PSL
UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à Inria Paris

Formal Verification of Rust Programs by Functional Translation

Soutenue par
Son HO
Le 09/12/2024

Ecole doctorale n° 386
Ecole Doctorale de Sciences Mathématiques de Paris-Centre

Spécialité
Informatique

Composition du jury :

Ralf, JUNG	Assistant Professor, ETH	*Rapporteur*
Christine, RIZKALLAH	Senior Lecturer, The University of Melbourne	*Rapporteur*
Cédric, FOURNET	Senior Principal Research Manager, Azure Research	*Examinateur*
Bryan, PARNO	Professor, Carnegie Mellon University	*Examinateur*
François, POTTIER	Directeur de recherche, Inria	*Examinateur*
Xavier, RIVAL	Directeur de recherche, Inria	*Examinateur*
Bruno, BLANCHET	Directeur de recherche, Inria	*Directeur de thèse*
Jonathan, PROTZENKO	Principal Researcher, Azure Research	Co-*Directeur de thèse*
Karthikeyan, BHARGAVAN	Directeur de recherche, Cryspen	Co-*Directeur de thèse*

Inria
INVENTEURS DU MONDE NUMÉRIQUE

Abstract

Software is plagued with bugs. Some of those bugs simply cause minor annoyances, but some others like Heartbleed or CrowdStrike are so critical that they make it to the headlines of the newspapers. As software has become pervasive in our lives, the presence of bugs which undermine its reliability is becoming an increasingly strong concern, leading to the development of techniques which enable ruling out entire classes of bugs when developing code. Among those techniques is program verification, which allows proving mathematical properties about the behavior of a piece of software regardless of a particular choice of inputs. While promising, program verification also tends to be extremely labor-intensive and as such hardly scales on real-world software. In this thesis, we thus explore the problem of developing techniques to implement verified, realistic, performant software, which can be deployed in real-world scenarios.

We first carry out three verification use cases targeting realistic software: Noise⋆, a verified compiler for the Noise family of protocols which generates performant C code with extensive correctness and security guarantees; the Zero-Cost Functors project, a set of language-based techniques that allow the programmer to modularly write and verify low-level, performant code while working at a high level of abstraction, and which allowed us to deploy verified cryptographic code in the Python standard library; a verified implementation of parts of the Dafny compiler inside of Dafny, which introduces techniques to stabilize SMT-based proofs. Those three use cases total more than 60k lines of code and proofs and introduce a set of techniques which allow program verification to scale more, while also revealing the fundamental limitations of the toolchains they rely on.

In the second part of this thesis, building on this practical verification experience, we introduce Aeneas, a new verification toolchain for Rust programs based on a lightweight functional translation. We leverage Rust's rich region-based type system to eliminate memory reasoning for a large class of Rust programs which includes loops and mutable borrows, by translating them to a pure lambda-calculus. Once extracted, this pure model allows the user to reason about the original Rust program through the theorem prover of their choice. Doing so, we relieve the proof engineer of the burden of memory-based reasoning, allowing them to instead focus on *functional* properties of their code. The translation mechanism itself relies on a symbolic execution which provably implements a borrow-checker for Rust; as such Aeneas is the only existing borrow-checker which comes with formal guarantees. As of today, Aeneas has backends for F⋆, Coq, HOL4 and Lean, which we present and evaluate.

Acknowledgements

I really had a great time doing this PhD, so much that I requested the permission to do it in four years instead of three. There are too many people I have to thank for all the good things I lived during this time and I hope I do not forget any of them.

I first wish to thank my PhD advisors, Jonathan Protzenko and Karthikeyan Bhargavan, who provided a constant attention and help throughout my PhD. First, I would like to thank Jonathan for his extreme availability despite the 9 hours (!) of time difference between Paris and Seattle, without which my PhD would have been drastically different and definitely a lot harder. I will never forget the image of you, or rather the absence of image of you, eating your breakfast while listening to my technical explanations, at an hour in the morning which is definitely not suitable for listening to technical explanations. My life as a PhD student would definitely not have been as easy without someone watching over my shoulders and actively working to always support me, starting with the internship at Microsoft during covid which I can't imagine how much effort you had to leverage to make it happen, or later when I moved to Seattle for a summer internship, providing me with a bed when I moved to my room, a car when I wanted to hike Mount Raineer, and pushing me to go spend some time in Hawaï (the fish were amazing). One of my then flatmates could not refrain her admiration when she asked me: "where the hell did you find an advisor like that?". Taking care of the *animaux infernaux* was also a pleasure, despite the fact that it cost me some sleep[1]. Karthik has also been of excellent advice throughout my PhD, always helping me to focus on the most important, while ignoring the irrelevant. I will never forget the adventures Théophile and I had in Rajasthan thanks to your very nice invitation to come spend a few weeks in India, and during which I managed to not get any heat stroke (not everybody was as lucky as me). Regarding Rajasthan, I also have to thank Prasad for almost organizing the trip for us, and for insisting on seeing the Taj Mahal at dawn so that we could properly enjoy the curved curves and the aligned lines (which we did). At this point I wish to emphasize that Karthik and Jonathan successfully managed to create an environment in which I could focus on research only, both enjoying a lot of freedom with regards to the topics I decided to work on while never feeling left on my own, and without worrying about other work-related issues. In particular, I never had to worry about financial constraints and I had the feeling I could go to whatever workshop or conference on the other side of the world that I wanted to attend, which may seem like a baseline but unfortunately is rather the exception than the rule.

I want to thank Bruno Blanchet for accepting to take on the burden of becoming

[1] I'm sorry for the plant, and I believe Laptop is sorry as well

my PhD advisor after Karthik took a sabbatical, and always patiently and carefully handling all the extremely annoying administrative issues I presented you, despite the fact that I know you do not particularly like this sort of work.

I wish to thank my jury members for accepting to review my PhD, starting with the referees. I really wish to thank Ralf Jung and Christine Rizkallah for accepting to take the time to thoroughly read this (heavy) manuscript: this really means a lot to me. I also wish to thank the other members of the jury: Cédric Fournet, Bryan Parno, François Pottier, and Xavier Rival. François and Xavier were actually also members of my *comité de suivi individuel*, for which I want to thank them as well.

I wish to thank the French administration for its wonderful rules which really helped a lot during the definitely not busy enough times which constituted the last months of my PhD. I always find it extremely fun to spend time thinking about how to satisfy ad-hoc conditions related to the composition of a jury, which were so constraining that I ended up not being able to include Jonathan and Karthik as proper members (!), and including them rather as invited members. However, even in kafkaiesque obscurity there is light, and I really have to thank DIENS' people, and in particular Eric Sinaman, for helping me navigate the PSL swamp and find a solution to this connundrum.

Needless to say that I really enjoyed my stay at Prosecco. It was a pleasure sharing a desk with you Théophile and I really appreciate your kindness and concern for others; I hope you will not miss too much the complaints I raised behind my screen during those long hours spent coding, always in a very soft voice of course. I also had an extremely good time in India fighting the deadly sidewalks with you, with the loads of cash we carried in our pockets because of those annoying bank card issues, and eating dressed as hobbos in those fancy Indian restaurants. A lot of the things I did would not have been possible without Aymeric's support and ability to work at undue hours (seriously man, do you sleep at times?). I also really enjoyed your incredible puns, all the beers we had while discussing how to reshape the world, the numerous discussions at your desk, as well as your almost tragic inability to say "no I don't have the time" when someone presents you with an interesting technical problem. There is nobody more perfect than Denis when it comes to forgetting science, especially when he walks into your office early in the morning saying: "Have you heard about [latest French political scandal]!? Incredible. I can't believe it.", before walking away without a word, his cup of tea in his hand of course. I'm also very happy that Guillaume joined the team after an improbable sequence of events: it really feels good to finally have a teammate going deep with me into the engineering problems of the projects I'm currently working on, and exchanging interesting views about the semantics of Rust and other technical points that, as you know, I always love discussing. I also want to thank all the other

people I enjoyed spending time with while at Inria, it was fun working with you, and having the traditional raclettes: Lucas (I will never forget those karaokes in Tokyo with MC Aymeric), Théo, Paul-Nicolas, Adrien, Justine, Vincent (the two of you), Louis, Yoann, Margot, Caroline.

I also wish to thank the team assistants who really did their best to make the administrative tasks as simple as possible whenever I travelled to a conference; in particular I wish to thank Christelle for her kindness and dedication.

I wish to thank Clément Pit-Claudel and Marianna Rappoport for the wonderful internship I had at AWS: it was very fun and I learnt a lot, despite the fact that verifying Dafny inside of Dafny itself was, well, painful as you know.

Doing research is also about meeting and discussing with people who have diverging views; I can't bear the environments with too uniform opinions which I have always, literally, fled. In this regard I had a lot of pleasure meeting Xavier Denis: I really appreciated all the discussions we had which invariably led me to the conclusion that "I don't think I'm wrong, but I don't think he's wrong either". It's too bad we never had an opportunity to work together, but I hope the future will fix that one day.

Even though my PhD kept me very busy, work did not consume all my life, and I wish to thank all the people who supported me outside of work.

I spent a lot of good time with you Victor, and we had wonderful trips together; behind the fun and the trips I could always count on you whenever I need someone, which is rare. I hope we'll continue have those board game evenings with Michel and Alexandre from time to time; Michel, let's find a way of hiking together. I'm happy I managed to convince you, Giuseppe, to come travel with Victor and I in Taiwan after the most intense lobbying campaign I've ever carried in my life; I only feel but regret about the fact that I couldn't help you bring this painting back (next time!). It is always a pleasure to have those long, deep and colourful conversations with you, Yuè: they are always extremely interesting and never boring. Manu, despite the distance you were always there when it was important, and I know you will always be there: I really want to thank you for that, is is extremely precious to me. Robin, I still don't understand how we ended up seeing each other so infrequently despite the fact that we literally lived next to each other; it's a good thing we eventually found bouldering as an excuse to spend some time together, and I hope we'll have more opportunities to spend week-end together like the one in Brittany with Jose. I've always had a lot of pleasure discussing all sorts of topics with you Laetitia, and I've been pleasantly surprised to see that you got an interest in topics related to my field; now I guess it is my turn to do the same (I wish I had more time...). Pierre-Cyril, it is always nice chatting with you about the wide variety of topics one can discuss with you: I hope we see each other

more in the future. Matthieu, I didn't expect all the things which were going to happen after we shared a room while preparing the *concours*, and Camille I'm very happy I met you thanks to Matthieu; it will always be great seeing you together with the twins, and I hope we will manage to make it happen despite the fact that we now traveled to different countries. I wish to thank the other members of the Team Bu Dong, Samuel and Agathe, for all the fun we had together along the years, especially when it comes to yelling with a microphone. I want to thank the climbing friends with whom I spent some good time every week at the end of my PhD - I wish it lasted longer: Laurène, Lola, Inna, Sophie, Sarah, Alex, Guillaume. I had a lot of pleasure meeting people at Nihao Paris, whom I wish to thank, starting with the dedicated organizer, Jean-Marie. I was also fortunate to keep some good old friends who, despite all the events of life, continue to be there in my life: William and Alexis.

Finally, I want to thank my family for always being there and supporting me. And I want to thank you, Jié, for being at my side: may it always be the case.

Contents

Part I

Introduction

In 2011, some former employees of the McAfee antivirus company decided to join forces to found their own cybersecurity company. For many years, they carried their business relatively unknown to the public by doing what every cybersecurity company does, such as providing security software, warning about security vulnerabilities [1] or investigating cyberattacks [2]. This situation dramatically changed thirteen years later, when, in the most ironic manner, a faulty update to one of their security software products caused a worldwide IT outage, resulting in flights and trains getting cancelled, financial transactions being blocked, and public services such as hospitals being heavily disrupted [3, 4]. Because of this unfortunate event, everybody now knows the name of CrowdStrike.

Alas, this is by far not the only occurrence of a severe bug found in hardware or software during the history of computer science. Just to name a few, a buffer overflow in the OpenSSL implementation of the TLS protocol caused the infamous Heartbleed security vulnerability [5]. An error in the floating point division of the Pentium processor in 1994 caused Intel to lose hundreds of millions of dollars in a massive recall of the faulty processors [6]. An integer overflow in the Inertial Reference System of the Ariane 5 rocket caused it to crash during its inaugural launch in 1996 [7], again causing the loss of hundreds of millions of dollars. Some bugs even caused the loss of lives, as experienced with the Therac 25 radiation therapy machine [8] and the TRP/2 radiation therapy software [9], whose software defects both led to the overdose of patients, leading to several deaths.

As software is pervasive in our lives today, controlling our cars, banking systems, medical devices, or communications, this state of affair can only cause serious concerns. In fact, in 1968, when computer software was still a young industry, participants to the first NATO Software Engineering Conference were already alarmed by what came to be known as the "software crisis" [10]:

> There is a widening gap between ambitions and achievements in software engineering. [...] Particularly alarming is the seemingly unavoidable fallibility of large software, since a malfunction in an advanced hardware-software system can be a matter of life and death, not only for individuals, but also for vehicles carrying hundreds of people and ultimately for nations as well.

As Dijkstra would put it later during his 1972 Turing Award lecture [11]:

> The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we

> had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.

More than 50 years later, after decades of hardware innovations which closely followed Moore's Law [12], software has become incomparably more complex than at the time of the software crisis, frequently reaching millions of lines of code, and even billions in the most extreme cases [13]. As a consequence, industrial software continues to be plagued with defects, up to the point where the average number of bugs per line of code has become a regular measure in the software industry [14]. In this context, it seems we are doomed to live in a world where missing error conditions cause planes to crash [15], incorrect conversions between metric and imperial units lead to space probes getting lost in space [16], or the risk of integer overflows forces engineers to hastily update the counter of views on YouTube because a Korean song is, definitively and in all respects, far too popular [17]. Or maybe not?

Back in the early days of computer science, in 1949, when computers were still using vacuum tubes [18] and programming meant painstakingly punching holes in tapes [19], Alan Turing himself was already thinking about the problem of ensuring that a procedure is correct [20]. His idea was to insert assertions in the code to help a human checker perform an external verification of its behavior. By doing so and probably unknowingly, he had become one of the pioneers of a field which would come to be known as *formal verification*.

Chapter 1

A Brief History of Formal Verification

1.1 The Early Days of Program Verification

Programmers were quick to realize that testing a program is not enough to ensure the absence of bugs. Rather, as Dijkstra famously put it, "program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence" [11]. This realization prompted computer scientists to look for software defects in a more systematic manner, in particular by designing methods to formally reason about programs and their behaviors. Some preliminary work followed Turing's 1949 paper [21–26], culminating in two seminal papers. In 1967, Floyd introduced the notion of *verification conditions*, which assert a relationship between preconditions and post-conditions [27]. Building on this work, Hoare laid out in 1969 the foundations of modern program verification by introducing what came to be known as *Hoare Logic* [28].

From there, researchers extended Hoare's work to build logics targeting programs of increasing expressivity. Among the practical verification work performed during the following years we can mention: the verification of a find function operating on an array and containing a loop [29], extensions of the Hoare Logic to reason about function calls [30] and parallel programs [31], or the proofs of correctness of a recursive quicksort function [32] and of a table lookup which used jump instructions [33]. Among the most influential work of that period, we can mention Dijkstra's paper which introduced the notion of *weakest preconditions* [34], and which allowed computing a logical predicate characterizing a whole function body, thus reducing its verification to a problem of entailment in a specific logic.

Despite the early promises of program verification and Hoare's initial optimism, who went as far as speculating that a proof of program correctness could be "hardly more laborious than the traditional practice of program testing" [29], the practical

results of program verification in the 1970s were extremely modest. For instance, the verification of a (minmax) SRLMT function made of 14 machine instructions and whose abstract definition is given by SRLMT(signal, limit) = MIN(limit, MAX(signal, −limit)) was considered state of the art in 1976 [35]. In view of such humble results, formal verification could not but attract strong scepticism from other computer scientists. Some of them, noting not only that the field had a hard time scaling to realistic programs, but also that many of such real-world programs did not admit formal properties that could be mathematically verified, advised for research on the topic to stop altogether [36]:

> Even [...] C.A.R. Hoare has been quoted [...] as saying "In many applications, algorithm plays almost no role, and certainly presents almost no problem." (We wish we could report that he thereupon threw up his hands and abandoned verification, but no such luck.)

At this time, the opponents of program verification were many and its proponents only a few [37]. Yet, some pioneers had laid out the foundations which would later lead to its first success.

1.2 The Emergence of Mechanized Reasoning

For program verification to become useful, it would need to scale. Interestingly, scalability did not necessarily mean automation, nor even the use of mechanized techniques. A notable example is given by Harlan D. Mills, then director of software engineering and technology at IBM's Federal Systems Division, who developed a methodology which by the mid-1980s was known as the "cleanroom" [38]. The cleanroom, if not program verification in the purest sense, consisted of taking inspiration from program verification to make code reviews a more systematic process. As such, programmers had to "prove" to their reviewers that their program was correct by going through it line by line and explaining as rigorously as possible why it was the case, very much like a mathematician detailing the proof of a theorem to some of her colleagues. This methodology led to some success, but at a cost (1000$ per line of code in the case of the space shuttle).

However, if we put the cleanroom methodology aside, for program verification to reach its first substantial achievements it would need to rely on mechanized proofs. The field of mechanized reasoning, that is, the use of computers to develop and verify mathematical proofs, emerged in the early days following two axes. First, the emergence of automated reasoning approaches which witnessed a flurry of activity during the 50s and 60s and, perhaps surprisingly, included almost all the earliest work on automated reasoning [39]. Second, the development of tools to actually perform mechanized

reasoning in a more or less interactive manner and by relying on a varying degree of automation. The latter really started at the end of 1960s and led in particular to the development of the first interactive theorem provers.

1.2.1 Automating Reasoning

In 1954, Martin Davis, working with the vacuum tube computer at the Institute for Advanced Study in Princeton, wrote an implementation of the procedure Presburger had designed to solve linear arithmetic problems. This was the first implementation of an automated decision procedure in the history of computer science and, as Davis himself acknowledged, there was a long road ahead [40]:

> Since it is now known that Presburger's procedure has worse than exponential complexity, it is not surprising that this program did not perform very well. Its great triumph was to prove that the sum of two even numbers is even.

Going beyond linear arithmetic laid the problem of reasoning about propositional calculus, which Newell, Shaw and Simon tackled in 1956 by introducing an incomplete procedure they called the Logic Theory Machine [41]. This procedure worked very much in the spirit of the proofs written in Principia Mathematica [42], the massive attempt from Whitehead and Russel to rebuild the foundations of mathematics, by deriving proofs from axioms and elementary reasoning rules like the modus ponens. The pen-and-paper effort of writing the three taxing volumes of the Principia had been so intense that Russel admitted later that his "intellect [had] never quite recovered from the strain" [43]. On their side, Newell, Shaw and Simon applied their Logic Theory Machine to the Principia; they succeeded in automatically proving 38 theorems from Chapter 2, to Russel's greatest satisfaction [44]:

> I am delighted to know that 'Principia Mathematica' can now be done by machinery. I wish Whitehead and I had known of this possibility before we wasted 10 years doing it by hand.

Propositional logic was of course not the end of the story: in order to write serious properties, one also needs functions and quantifiers; that is, first-order logic.

In 1957, Abraham Robinson [45] proposed a deviation from the then 'standard' view of deriving mathematical theorems from axioms and inference rules, used for instance in the Principia Mathematica and which then seemed natural to mathematicians. Instead of trying to replicate the process of a human being doing a proof in the style of the Logic Theory Machine, he advocated tackling the problem of computationally demonstrating

the unsatisfiability[1] of first-order logic formulae by designing techniques more suited for computers. He proposed the method of using *Skolem functions* to eliminate existential quantifiers, before using Herbrand's theorem, which in effect defines a procedure, on the resulting universally quantified formula. The procedure he described would work, after skolemization, by accumulating different instantiations of the formula until reaching a contradiction. Robinson drew a parallel of searching for instantiations to the intellectual process of a mathematician searching for a proof, and doing so insisted on the importance of properly selecting those instantiations.

Robinson's idea would prove extremely influential and was quickly put into practice, notably by Gilmore [46], and improved. Prawitz devised a method to "select" instantiations more efficiently [47]. In parallel, Martin Davis and Hilary Putnam noted that Gilmore's procedure was too slow at deciding whether a propositional logic formula is satisfiable or not [40], a problem which later came to be known as the *satisfiability problem*, or simply *SAT*. They proposed an improvement by introducing, first in an unpublished report to the NSA [48], then in a paper [49], a technique which came to be known as the *Davis-Putnam* procedure. This procedure was improved two years later with the introduction of the extremely influential DPLL (*Davis-Putnam-Logemann-Loveland*) algorithm [50].

More work followed to improve the performance further, in particular to efficiently combine Prawitz's ideas for the selection of instantiations with work on efficient SAT solving along the lines of the Davis-Putnam procedure. This led in particular to procedures based on *unification* algorithms [51–53].

A few years later, in 1965, John Alan Robinson[2] revolutionized the subject [40] by introducing a procedure based on a single rule of inference, called *resolution* [54], and which would crucially rely on unification. This rule was easily performable by computer, complete for first-order logic, and had the radical advantage, over the methods based on the Herbrand theorem, to not require the alternation between two phases (instantiating variables, then checking if the resulting propositional formula is unsatisfiable, then adding a different instantiation to the first, etc.).

Unfortunately, this was still too slow and more work was needed to cut the search space when tackling SAT problems. Early attempts included work by Robinson, Wos and Carson [55–57]; the field would however have to wait more than three decades for the next revolution to truly happen.

Subsequent work attempted to combine different first-order theories, eventually

[1] In order to prove that a mathematical formula is valid, a possibility is to show that its negation is unsatisfiable; hence the interest in proving unsatisfiability.

[2] Not to be mistaken for *Abraham* Robinson, the one who advocated using Skolem functions and Herbrand's procedure and that we mentioned above.

leading to the currently known Satisfiability Modulo Theory (SMT) solvers. Nelson and Oppen [58] proposed a method by which procedures can communicate information they individually derive about equalities between terms. For instance, a procedure for linear arithmetic could deduce from $i \leq j$ and $j \leq i$ that $i = j$, from which a procedure for the theory of arrays could deduce that $\texttt{get}\ (\texttt{set}\ a\ i\ 0)\ j = 0$, and so on, until reaching a contradiction. The Nelson-Oppen method explained how to combine individual procedures; subsequent work from Shostak [59] showed how to construct such procedures by detailing a particular strategy for deciding satisfiability of quantifier-free formulas[3] in certain kinds of theories.

Several breakthroughs starting at the end of the 1990s finally initiated what is now known as the "SAT revolution" [60, 61], which in turn allowed the "SMT revolution". They started in 1999 with the solver GRASP [62], which proposed a new architecture called *conflict-driven clause learning (CDCL)*. Two years later, the CDCL-solver Chaff [63] achieved significant performance gains[4] by putting a strong focus on better implementation techniques. From then, the field of SAT solving witnessed an explosion of improvements and industrial applications [61, 64].

The progress of SAT solving fueled the field of SMT solving, as the latter can be seen as a combination of SAT reasoning and theory reasoning. This led in particular to the *DPLL(T)* framework [65], which provides a way of integrating specialized theory solvers within a general purposed engine based on the DPLL procedure, and is used nowadays by several SMT solvers such as CVC4 [66] and Z3 [67]. The breakthroughs in SMT solving during the 2000s would have a direct impact on program verification, as we shall see later.

1.2.2 The Emergence of Interactive Tools

In parallel to efficient automated deduction procedures, computer science saw the emergence of tools to actually perform mechanized proofs. Some of those tools were purely dedicated to program verification. The earliest such tool was the program verifier developed by James C. King during his PhD [68], which turned annotated programs into verification conditions sent to an automated theorem prover (see below), and was specialized on the verification of programs performing integer arithmetic. The most challenging example verified by King was a program computing integer powers. King would later continue his work on program verification to lay the foundations of symbolic execution [69, 70]. Also worth mentioning are PL/CV [71], which allowed writing

[3] Where all variables are implicitly universally quantified

[4] Up to two orders of magnitude on difficult benchmarks, compared to other existing solvers.

assertions in an Algol-like programming language, or the Stanford Pascal Verifier, which operated on properly annotated Pascal programs [72]. More recently, we have seen the emergence of verification aware programming languages such as Idris [73], Dafny [74] or F* [75, 76].

Another big class of tools which emerged during those years were interactive theorem provers. One of the goals of developing theorem provers was of course to assist mathematicians in writing new proofs [77], or at least to help them filling the holes in proof outlines such as the ones found in textbooks [78]. This line of work would yield a flourishing research [39, 44, 79], reaching a point where there now exist libraries which formalize substantial portions of mathematics [80, 81], and proof mechanization sometimes catches up with the ongoing mathematical research [82–85]. However, and perhaps surprisingly given the name, the development of theorem provers was also highly motivated by the need for tools which could perform computer system verification, in particular during the 1970s and the 1980s [39]. A notable target was the verification of security properties such as isolation in the then new time-sharing operating systems [38].

At this stage, it is worth noting that the distinction between program verifiers, which allow verifying *programs*, and theorem provers, which allow general mathematical reasoning, is not always clear cut. For instance, the PL/CV program verifier that we mentioned above, and which targeted the verification of programs written in an Algol-like language, was directly inspired by the LCF theorem prover [86] (see below). As it targets a language which is essentially pure, it is actually very close to a theorem prover and as such was used to formalize a number of arithmetic facts [71]. At the opposite end, both the 'Pure LISP theorem prover' [87] and ACL2 ('Applicative Common Lisp') [88] can be considered as theorem provers [39], though their "logics" are (extensions of) a pure subset of Lisp. The Coq proof assistant [89], developed after 1984, was initially developed as a type-checker for a dependently-typed language called the *Calculus of Inductive Constructions* (CIC) [90], before being extended to allow the incremental construction of proof objects [39, 89]. More recently, Lean [91], developed after 2013 and also based on CIC, describes itself as both a programming language and a theorem prover.

As we noted earlier, the bulk of the development on interactive theorem provers, where *interactive* has to be understood in a very broad sense, started *after* the research in fully automated approaches had already produced notable results. This is not coincidental: if some theorem provers clearly benefited from the research in automated deduction to provide a high degree of automation, their development was also a reaction to the research in fully automated approaches [39]:

> Perhaps the most powerful driver of interactive theorem proving was not so much technology, but simply the recognition that after a flurry of activity in automated proving, with waves of new ideas like unification that greatly increased their power, the capabilities of purely automated systems were beginning to plateau.

As Robin Milner explained when asked about the development of his own prover[5] [92]:

> I greatly admired Robinson's resolution principle, a wonderful breakthrough; but in fact the amount of stuff you can prove with fully automatic theorem proving is still very small. So I was always more interested in amplifying human intelligence than I am in artificial intelligence. That means I began to be interested in how one could verify programs.

The first implementation of what resembles an interactive theorem prover was probably Paul Abrahams' Proofchecker in 1963 [39, 93], which he used, quite expectedly, for the verification of theorems from the Principia Mathematica. The first sustained effort in developing theorem provers closely followed, with the SAM (Semi-Automated Mathematics) family of provers, of which 5 different versions were developed between 1963 and 1969 [94]. From there the field witnessed an explosion of provers making different design choices with regards to such things as: their underlying logic, the level of automation, the proof language, etc. In 2006, Freek Wiedjik could write a survey to compare the Pythagorea's proof of the irrationality of $\sqrt{2}$ in a selection of 17 of the then most significant theorem provers [95]. Some of the most important ideas which underpin those provers had appeared quite early.

The idea of exploiting the *Curry-Howard* correspondence between propositions and types was introduced by one of the earliest provers, Automath, developed between 1967 and 1968 [96]. The idea is that if types are interpreted as propositions, constructing a proof simply consists in exhibiting an object of the proper type. A large family of provers reused this idea, though in different ways [39, 97] and for different logics; among those we can mention: νPRL [98], Coq [89], Agda [99], Twelf [100], and more recently Lean [101].

Another series of influential provers appeared in the following decade when Robin Milner, then interested in program verification, started in 1972 to work on the LCF provers[6] [86, 103]. Those provers introduced several important ideas. First, they used a powerful automatic simplification mechanism. Second, they introduced backward,

[5] The influential Edinburgh LCF, which he started in 1972

[6] The first one was based on Dana Scott's *Logic of Computable Functions* [102], hence the name

goal-directed proofs performed by means of *tactics*, where one proves theorems by applying transformations (implemented by the *tactics*) to a context made of a target proposition (the *goal*) and a list of assumptions. Finally, they used a clever encoding of objects of the logic as abstract objects inside the meta-language[7] of the prover. Being abstract, objects such as theorems could only be manipulated through functions which implemented the inference rules of the logic, ensuring that any theorem produced by the user could be derived from the axioms and the rules of the logic. This allowed implementing arbitrarily complex decision procedures without extending the trusted code base [39]. Milner's work led to the creation of the LCF *family* of provers, which notably include HOL4 [107], HOL Light [108], or Isabelle/HOL [109, 110].

"LCF-style" tactics have also become a standard ways of doing proofs within interactive theorem provers. For instance, we already mentioned the fact that the Coq proof assistant, which started as a type-checker for the Calculus of Inductive Constructions, later introduced tactics [39]. In the case of provers like Coq, tactics work by generating proof terms which are then checked by the kernel: similarly to provers of the LCF family, this allowed implementing arbitrarily complex decision procedures without compromising the trust in the prover.

Most of the theorem provers we mentioned above emphasized simple and secure foundations in particular by being built upon a (relatively) small trusted kernel, from where automation was gradually built; a consequence is that in the first years of their existence, provers like Coq and Isabelle/HOL had rather limited automation [39]. At the other end of the scale, other provers directly aimed for state of the art automation with less emphasis on trust, generally for the purpose of being applied immediately to interesting examples in program verification [39].

In 1971, Robert Boyer and J Strother Moore began working on a series of provers which, after several iterations, led to NQTHM ('new quantified THM'), often referred to as the 'Boyer-Moore theorem prover' [39]. It introduced several techniques to automate proofs by *induction*: the fact that recursive functions were defined by primitive recursions guided the automatic application of induction principles; the prover was also able to *generalize* the statements to be proved by strengthening the inductive hypotheses before performing the induction, while also relying on a systematic use of simplification.

The user interacted with the prover by stating a sequence of theorems to prove, as well as hints to indicate how to use the intermediate lemmas; the prover would then proceed by automatically proving the theorems one by one in order, crucially leveraging the intermediate lemmas provided by the user. The authors compared this approach to

[7] ML ('Meta-Language'), introduced for the special purpose of implementing Edinburgh, would evolve to have a life of its own [104–106]

the process of interacting with a mathematics student: "given the axioms of Peano, he could hardly be expected to prove (much less discover) the prime factorization theorem. However, he could cope quite well if given the axioms of Peano and a list of theorems to prove (e.g., "prove that addition is commutative," ... "prove that multiplication distributes over addition," [...])" [111]. This mostly automated way of doing proofs would later be complemented with low-level commands which enabled a more interactive experience [112]. The Boyer-Moore prover appeared to be quite practical and as such was used in several interesting use cases, both in software [113, 114] and hardware [115].

Moore would later collaborate with Matt Kaufmann to build ACL2 [116], 'A Computational Logic for Applicative Common Lisp', a successor of the Boyer-Moore prover which still has an active community today [117]. One important innovation was to not make any distinction between the *logic* and the *implementation language* (which are both pure subsets of Common Lisp). This allowed for efficient execution of functions inside the logic, and combined with the possibility of reasoning about ACL2's logic from within ACL2 itself allowed soundly adding custom extensions to the proof engine [118]. Due to its high-level of automation, ACL2 would find a number of applications in the industry [119].

A different series of provers, starting with EHDM in 1983 [120] and continuing later with PVS [121], integrated variations of Shostak's decision procedures for equality in a combination of theories [122], which as we have seen had laid some of the foundations of modern SMT solving. EHDM was used mechanize the proof of a fault-tolerant clock synchronization algorithm in 1991 [123] and initially introduced in an article by Lamport and Melliar-Smith [124], identifying several errors while doing so.

PVS, still under development today [125], is particularly notable in that it managed to combine both a high-level of automation *and* an expressive type system, contradicting the then common belief that it was possible to have one or the other, but not both [39]. It allowed a form of dependent types, where types are parameterized by terms, by supporting *predicate subtypes* (which restrict a type to a subset of this type by means of a predicate). The drawback of this expressivity is that it made typing undecidable; as a consequence, type-checking could require arbitrarily complex proof work from the user. Similar design choices were made later by the F* programming language [75], which combines an even more expressive dependent type system with the Z3 SMT solver. Automation put aside, users of PVS could use commands to interact with goals, in a fashion similar to tactic-based theorem provers, and could use a (restricted) language to combine proof commands into more powerful strategies. PVS found a number of applications, in particular in hardware [126]. More recently, F* permitted the verification of a large collection of low-level applications, in particular related to

cryptography [127, 128] as we shall see later.

1.3 First Applications

The progress in automated deduction and in the development of verification tools was accompanied by a number of applications, which boomed in the 1980s and the 1990s.

The first success came from not from the world of software but from hardware verification. A first partial success came from the SIFT project [129], an aircraft control computer commissioned by NASA and partially verified by means of the STP (Shostak Theorem Prover, a descendant of the the Boyer-Moore prover) and EHDM, which however drove much controversy as the verification process led to oversimplification, making it "unfit for purpose" [130]. Coming behind were Hunt's verification of a microprocessor with the Boyer-Moore prover in 1985 [115], the verification of parts of the Viper processor [131] with HOL (a successor of LCF) [132], or the verification by AMD of the floating-point division program of the AMD5k86 microprocessor using ACL2 [133], and which followed the Pentium bug [6]. In parallel, some companies like Intel validated hardware designs by resorting to model checking, a technique which emerged during the 1980s to analyze finite state systems [134–136]; Intel however used model checking more for debugging purposes to find design mistakes rather than as a verification technique [38].

With regards to software, formal verification also made some successful endeavours, for instance with the verification of a concurrent data-structure [137] and a minimalistic kernel [114], both with the Boyer-Moore prover, or the mechanization of a fault-tolerant synchronization algorithm with EDHM [123].

There was however still a huge gap between the ambitions of program verification practitioners and what was achievable in practice. For instance, they quickly understood that verifying a single program was not enough, and that in order to get the highest level of guarantees one needs to verify a whole stack comprising: the compiler from the source language to the assembly code, the operating system responsible for managing the software, and the hardware underneath [38]. The verification of the CLI stack with the Boyer-Moore prover in 1989 [113] was a first attempt in doing so, but on toy components.

In parallel to academic endeavours, industrial practitioners desiring stronger guarantees for their software resorted a wide range of methods loosely related to program verification and which can be grouped under the umbrella term of "formal methods". A driving force was the certification of critical-software [138–140], but sometimes, and

quite surprisingly, the goal was also to reduce the *cost* and *time to market*.

We mentioned the use of the "cleanroom" by Harlan D. Mills at IBM during the 1980s, which was very close to program verification, though with manual methods rather than mechanized tools. This was a rather isolated initiative, and most of the use of formal methods at that time consisted in using formal specification languages like Z [141], VDM [142] or B [143]. The goal of these languages was not to write proofs, which were completely absent, but rather to clarify what the software is supposed to do in order to catch mistakes early in the development process. The rationale was that the later a software mistake is found the more expensive and difficult it is to fix it; by catching mistakes early one could save time and money.

One of the most successful application of this methodology was the IBM CICS (Customer Information Control System) [144, 145], which used the Z notation on a system not considered as safety critical. IBM reported that using this methodology allowed them to reduce by half the normal number of customer-reported errors, and also reduce the total development cost of the release by 9% (which amounted to million of dollars) [146]. This use of formal methods on a commercial software was considered a massive success, and IBM was awarded, together with the Oxford University Computing Laboratory, the Queen's Award for Technological Achievement in 1992 [147]. IBM was not an exception: several other companies reported productivity benefits from the use of specification languages [148–151].

Interestingly, the use of a formal specification language without doing proofs was already enough to significantly increase confidence in software. A notable example is given by the company Praxis, a practitioner of VDM, which in the 1990s felt comfortable enough to engage in a project for the UK National Air Traffic Services (NATS) which exceeded the company's annual turnover, and contract to repair at no charge any major fault found in the following 5 years. In practice, the delivered system failed so rarely that after a few years NATS had to retrain their personnel in how to restart it [152].

Formal methods were of course applied to critical software belonging to a wide range of domains, and which include: nuclear plants [139, 148, 149, 153], avionics [154], or medical devices [155]. In case of the Sizewell B reactor in particular, one core issue was the reliability of compilers: software developers spent 18 person months proving the equivalence between the 26k lines of code written in their sources files, and the assembly resulting from compilation [153].

More closely related to program verification, we can mention the use of formal methods in railway systems, for instance to specify the SACEM embedded system which controls the speed of the trains on the RER A line in Paris in 1989 [138]. The goal was then to validate a new system which reduced the distance between two consecutive

trains in order to increase the traffic, while preserving the same global safety as before. The B method was used to specify part of the SACEM code deemed safety critical: annotations in the code were transformed into verification conditions which were proven "by hand" [138], and the objective of an increase of 25% of the traffic was achieved. A few years later, in 1998, the B method was used again, this time to fully verify the code of the driverless metro line 14 in Paris, which consisted in 86k lines of Ada [156]. The tool then worked by progressively *refining* [157, 158] abstract specifications into concrete code, and integrated solvers which allowed to automatically discharge 92% of the proofs [156]. The total cost of doing the proofs was evaluated to 7 person-months; interestingly, the confidence was such that *unit* tests were deemed unnecessary and removed altogether, resulting in a save in cost [156, 159][8].

During the 1990s, the use of formal methods gained popularity in the industry while still causing enough scepticism to push some of its proponents to publish position papers in their favour [160, 161]. The success of formal methods like with IBM CICS were however acknowledged, though with some reservations [162, 163]. But as we have seen above, the use of formal methods on realistic software mostly did not involve proofs, which then hardly scaled or only on a specific class of software, e.g., railway systems. The situation would dramatically change in the following decade.

1.4 Breakthroughs

In parallel to the SAT and SMT revolutions, the 2000s saw the emergence of highly automated tools which allowed analyzing programs at scale, sometimes by being fueled by SAT and SMT, sometimes by using orthogonal techniques.

The SLAM tool used model-checking to analyze Microsoft drivers, which had been a "major source of concern within [Microsoft] for many years" [164]. SLAM was designed to identify all potential bugs belonging to a certain class while limiting the amount of false positives; more recently, the Infer tool was designed to only report true positives at the cost of missing potential bugs [165], and is being routinely used at Meta to analyze millions of lines of code [166–168].

Some tools also relied on bounded model checking, by which a program is analyzed up to a specific number of unrollings of loops and recursive calls, for instance to analyze the safety of C programs [169], and by crucially relied on efficient SAT solvers [170].

Techniques based on abstract interpretation [171] allowed the fully automated analysis of code bases of hundreds of thousands lines of code, for instance with the

[8] As it is hard to test components separately, rather than the system as a whole, unit tests were expensive, and in particular more than the 7 person-months required to do the formal proofs [159].

Astrée analyzer [172, 173], which could in 2003 check the absence of run time errors in the 132k lines of C code of the Airbus A340 flight control software.

All the above-mentioned techniques were able to be made fully automatic by focusing on the verification of a restricted set of properties, and in particular runtime safety. The verification of more advanced properties such as functional correctness, by which a program not only runs safely but also behaves according to a well-defined specification, would know its first important success with the help of interactive theorem provers.

The first such success was given by CompCert in 2006 [174], the formally verified implementation of a realistic optimizing compiler for C, which was developed with the Coq proof assistant. A second success happened in 2009 with seL4, a complete, general purpose micro-kernel verified in Isabelle/HOL [175], and which could achieve a performance comparable to non-verified, high-performance L4 kernels. Both applications were considered as major breakthroughs and as opening a "new age of verification" [176]; as such they were both awarded the ACM Software System Award [176]. CompCert is now being used in avionics and in the domain of nuclear energy [177–179], while seL4 is being used in aerospace and in autonomous aviation [180].

As using interactive theorem provers could prove tedious, a new generation of automated tools also emerged, this time leveraging the benefits of the SMT revolution. This led for instance to Dafny [181], Chalice [182], Why3 [183], Viper [184], or F* [75]. Dafny and F* in particular were used to verify a wide range of realistic software [128, 185–187].

As of today, program verification has been able to produce a wide range of verified artifacts. Following CompCert and seL4, several verified compilers and micro-kernels have emerged, with for instance the CertiKOS micro-kernel [188] (verified in Coq), the CakeML compiler for ML [189] (verified in HOL4), and the Vélus compiler for Lustre [190] (in Coq). But program verification is not limited to compilers and micro-kernels: other examples include the miTLS verified implementation of TLS [191] written in F# and verified in F7 (a predecessor of F*), the HACL* verified cryptographic library [185] (F*), which contains more than 100k lines of verified C code and is notably being used in the Linux kernel, in Firefox and in the Python standard library, the Fiat-Crypto library of big numbers for cryptography, which is used by some of the cryptographic primitives in Google Chrome [192], and verified implementations of distributed protocols [186] and of distributed key-value stores [187]. It is worthy to note that 30 years after the endeavour of the CLI stack [113], realistic versions of most of the components needed for such a stack have been verified in one way or another [193].

In parallel, validation efforts evaluated the impact of program verification on software reliability. Both detractors [36] and proponents [160, 161] of formal methods identified

early in the history that program verification is fallible, in particular because all verification efforts rely on some assumptions that must be true for the verification to be valid; for instance, preconditions must be satisfied upon calling a function, formal developments make assumptions about the hardware or some system calls, the verification tools themselves may have bugs, etc. Several studies have showed that the use of program verification substantially improved the reliability of software *in practice*, in particular by analyzing the CompCert compiler [194] and the seL4 micro-kernel [195]. In the case of CompCert, authors of the CSmith fuzzer tested CompCert and other industrial C compilers; they note: "The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which CSmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task."

Following those academic success, program verification has actually evolved up to the point where there now exist industrial verification teams, which use verification tools such as Coq, Lean, HOL Light or Dafny on a daily basis. There are notably such teams at Galois [196–198], Meta [199], Amazon Web Service [200, 201], or ProvenRun [202] Yet, despite those success, the use of program verification is still a niche activity facing many challenges that must be overcome to gain wider adoption.

Chapter 2

The Ongoing Challenges of Formal Verification

Some early detractors of program verification prophesied that "no major programs [...] would ever be verified by man, woman, child, beast, or machine" [37]. If the past 20 years have proven them largely wrong, the field still faces much criticism and as such is forced to continuously justify its existence [151, 159]. Yet, some of the criticism is not unjustified.

The main challenge of program verification today is that it is an extremely labour intensive activity requiring a lot of expertise. We mentioned the breakthroughs that were the CompCert compiler and the seL4 micro-kernel. Verifying both programs, which have around 10k lines of code each, required 6 person-years to write 150k lines of Coq proofs in the case of CompCert [179], and 22 person-years to write 200k lines of Isabelle/HOL proofs in the case of seL4 [180]. Those two programs are however relatively simple with regards to the current industrial standards. The 10k lines of code of CompCert are hardly comparable to GCC's more than 15 millions of lines of code [203]. Verifying such an optimizing compiler was undoubtedly an impressive tour de force, yet it required some simplifying assumptions which could in theory be used to prove incorrect optimization passes [204][1]. In the case of seL4, the micro-kernel only runs on a single core[2]; its complexity can also not be compared to that of a full operating system.

Klein et al. analyzed the cost of developing and verifying seL4 and came to the conclusion that program verification, which allows developing software whose assurance level is higher than Common Criteria's EAL7 (the highest), is competitive with regards

[1] In particular, malloc() is assumed to always succeed and never return the null pointer; this could be used to prove that a pass which removes defensive checks against heap overflow is correct [204].

[2] The verification of a multicore version which uses a global lock is on hold [205].

to standard techniques allowing the development of software satisfying EAL6[3] [206]. If this may be used as an argument to promote program verification for the certification of critical software, we are far from being able to claim that verification can help reduce costs by catching bugs early, the same way specification languages helped companies like IBM and Praxis during the 1990s. As a consequence, we are today in an in-between situation: several verification success have demonstrated the feasibility of program verification on real-world projects, and the field has actually achieved the notable success of making it into the industry. However, the cost of verification is such that industrial practitioners are still few, and the size of the biggest verified software is still tiny compared to what is regularly developed nowadays.

The main reason behind the prohibitive cost of program verification is that formal verification requires carrying out even simple reasonings with an extremely high level of details. This is made worse by the problem of reasoning about concrete programs and their semantics. For instance, on a real computer, machine integers are bounded, and proof engineers have to explicitly reason about those bounds. Worse, modeling side effects can be hard and lead to tricky reasonings. In particular, many programs use pointers to do in-place updates: this simple problem has been a huge pain for program verification [168, 207], leading to a rich literature of techniques to efficiently model and reason about memory manipulating programs, including techniques such as dynamic frames [208], implicit dynamic frames [209], or separation logic [168, 210].

If many proof obligations are actually quite simple and easily disposed of, the task of verifying a realistic program requires discharging a huge amount of them; for instance, replaying the proofs of the VeriBetrKV distributed key-value store verified in Dafny [187] requires sending 5600 queries to the Z3 SMT solver [211]. This is a point about which both proponents and detractors of program verification readily agreed on. As Shostak had remarked back in the 70s, the mathematics involved in verification work is "wide" rather than conceptually "deep" [38]; he would leverage this insight to drive the development of efficient decision procedures. De Millo, Lipton and Perlis, this time arguing *against* program verification, would make the similar claim that program verification is "long and involved but shallow" [36].

Research to address this problem has thrived since the early days of verification, with from the beginning a tension between highly-automated decision procedures and less automated but more flexible techniques such as interactive theorem proving.

[3]Klein et al. claim that that the cost of developing the kernel with (resp., without) the tools was of $362/SLOC (Source Line of Code) (resp., $127/SLOC), to be compared with the $1 k/SLOC for EAL6 software usually found in the industry. This estimation is to be taken with a pinch of salt, as the salary of a PhD student is definitely lower than the salary of a software engineer, and as it doesn't include the cost of actually certifying the software; the order of magnitude is still informative.

The need to discharge a lot of mundane proof obligations naturally pushes people towards more automated techniques, and this has been a strong argument during the past 20 years for the development of verification tools based on SMT solvers [75, 181, 183], which have become the de facto decision procedures when one thinks of *automation* in program verification today. At the same time, today's SMT solvers can be hard to understand and control: as they are based on sensitive heuristics, small changes in queries can have huge consequences, with proofs suddenly taking a lot more time to complete, or breaking in unpredictable manners [196, 211–214]. This proved to be an important issue in practice; remarkably, a whole session in the first edition of the Dafny workshop was dedicated to this well-known problem of "proof stability" [215].

SMT solvers also have some more fundamental limitations. The fact that they manipulate formulae in first-order logic while being used in frameworks which support higher-order logics can lead to disconcerting failures when some properties get "lost in translation" [216, 217]. Another issue arises from the fact that SMT solvers are domain-specific in that they can handle a fixed number of theories (e.g., linear arithmetic, arrays, bitvectors, etc.). When verification requires reasoning about a theory which is not supported, or with inefficient heuristics[4], automation falls short and the user has to resort to writing tedious, manual proof scripts [214, 218, 219]. One consequence is that users of tools based on SMT solvers can not simply use them as black-boxes: they need to have some understanding of their inner workings to properly structure their proofs [214, 217].

In an ironic replay of what already happened during the 1970s, this state of affair can push people towards less automated but more interactive and flexible techniques [201]. The world is of course not rosy either when looking at those approaches. If their proponents have come up with a set of techniques to implement efficient, custom automation [220, 221], using interactive theorem provers still often requires writing tediously detailed proofs. Proof scripts consisting of calls to arcane tactics doing mysterious transformations also quickly lead to inscrutable "tactic soups" [222], as once parodied by Conor McBride's reference to the presumed EAR_OF_BAT_TAC tactic in LCF [39]. A good illustration of this phenomenon is given by the fact that Lean integrated a set of tactics specifically designed to help manipulate goals containing coercions between different types [223].

One consequence of those problems, both for SMT-based approaches and interactive theorem provers, is that writing and maintaining proofs at a large scale requires a lot of expertise [196, 206, 214, 224].

[4] Z3's heuristics for reasoning about non-linear arithmetic are for instance notoriously unstable and as a consequence often deactivated [214]

Aware of the shortcomings of those approaches, the proponents of both highly automated and more interactive techniques have been lurking at each other's side. Interactive theorem provers have been integrating support for highly automated procedures in their workflows, for instance with the use of hammers by which one can send a proof obligation to an automated theorem prover and in particular to SMT solvers, the most successful one being probably Sledgehammer for Isabelle/HOL [225]. A more recent technique has been the use of machine learning to automatically generate proof scripts, but this has mostly been confined to the use of theorem provers for mathematics, and not yet for program verification [226–228]. The Lean interactive theorem prover was also initially designed with the explicit goal of bridging the gap between interactive use and automation, in particular by means of a powerful meta-programming framework [229]. The other way around, highly automated tools have been adopting techniques coming from the world of interactive theorem proving. Why3 has integrated "tactics" to help better interact with the proof obligations, for instance by splitting goals [230], while adding support for what is known as *reflection*, to enable the implementation of custom procedures to complement SMT automation [219]. The F* verification oriented language has also integrated meta-programming facilities to implement custom automation in order to complement Z3 [218].

At the same time, the interactive features integrated in automated tools provide a very distinct experience from the one provided by interactive provers, the reverse being also true. For instance, the "tactics" provided by Why3 are rather simple and don't allow doing operations much more complex than splitting goals before querying SMT solvers; as such they hardly compare with the degree of interaction provided by even the first LCF provers. Similarly, if Meta-F* [218] introduced tactics inside the F* programming language, as of today those hardly provide an experience which can be considered as interactive; F* indeed doesn't allow iteratively stepping through the proofs, rather, the complete proof of a function body is processed in one go, and all the proof states displayed simultaneously in the editor. Looking at the opposite direction, if hammers have been a welcome addition to interactive provers, they use SMT solvers in a way which is quite different from tools built with SMT solvers at their core. In particular, a core component of Sledgehammer is the *fact selector* which attempts to discover which lemmas might be useful in the proof, and give all such lemmas to the SMT solver [231]. Dafny on its side doesn't attempt to automatically apply lemmas, which have to be manually instantiated by the user when writing proof scripts: the SMT solver is used mostly to bridge the holes in the proof laid out by the user. The F* programming language does allow the automatic instantiation of lemmas, but by using SMT *patterns* (a lemma is automatically instantiated if some term matching a

given pattern, which is stored in an E-graph, is found in the context), which operate in a manner radically different from the selection operated by Sledgehammer. It is also worth noting that the presence of baked-in SMT automation provides a quite different experience when using dependent types in Dafny or F* than when using a prover like Coq. For instance, the fact that F* uses an extensional type system allows smoothly using refinement types without thinking about the proofs of refinement when the solver automatically discharges them. At the opposite, in Coq, independently from the level of automation, refinements require explicit proof terms, making the use of dependent types a bit heavier in practice.

When looking back, the field of verification has made dramatic progress since Alan Turing's work on checking a large routine, and pioneers such as Floyd and Hoare laid out the foundations of modern program verification. There now exist working tools which have been applied to a wide range of realistic programs and are being routinely used in the industry. However, the field still faces many challenges as tools are hard to scale and require a huge amount of expertise; we have yet to create a new generation of tools that will push the boundaries of program verification.

Chapter 3

Thesis Overview and Contributions

The work we present here explores the problem of scaling program verification to a larger class of programs. In this thesis, we argue that **the use of techniques to simplify the modelling of program semantics, combined with reasoning frameworks based on custom, extensible automation, is a promising way of scaling verification to a large class of realistic programs**.

3.1 Moving Up the Stack of Verified Software with SMT-based Automation

Recent research aimed at verifying cryptographic implementations has revealed a sweet spot for program verification. Cryptographic *primitives* implement tricky mathematical specifications while at the same time aiming for extremely high performance; as such they are extremely low-level and error-prone, resulting in a high number of CVEs in common cryptographic libraries over the years. As cryptographic providers must also both be *agile*, by providing multiple implementations (e.g., Blake2b and SHA-256) for the same functionality (e.g., hashing), and *multiplexed*, by being able to choose between multiple (optimized) implementations of the same algorithm, developers of such libraries must implement and maintain a high number of implementations, increasing the risk of mistakes. Higher up the stack, *protocols* such as TLS can be extremely complex, with in particular non trivial state machines which are easy to get wrong [232]. Worse, any CVE found in such implementations leads to serious security vulnerabilities; case in point, the infamous Heartbleed vulnerability in the OpenSSL implementation of TLS has at its root a simple buffer overflow [5].

At the same time, reasoning about cryptographic primitive implementations is manageable with the existing verification tools. Indeed, verifying functional correctness

is often enough in giving a very high level of confidence in those implementations; other applications would require more: for instance, a verified distributed database would also require a property of eventual consistency to be meaningful. If cryptographic specifications can be subtle, they also fit well within the logic of proof assistants. Finally, cryptographic primitives have a limited use of effectful features: they are sequential, do not have a complex state, and do not manipulate complex data structures but rather only a few buffers.

This state of affair has made cryptographic implementations a prime target for program verification, starting with cryptographic *primitives*, but also somewhat extending to *protocols*. Work like FiatCrypto [192] has led to the verification of big-number libraries for cryptography, which are being used by several cryptographic primitives in Google Chrome. Broader in scope, Project Everest has led to the development of the HACL⋆ cryptographic library [127, 233], which contains a large collection of cryptographic primitives currently used by Firefox, Linux, WireGuard, or more recently the Python library, as well as verified protocol implementations like miTLS [191, 234].

If past work tackling the verification of cryptographic implementations can be considered an important success, many challenges remain to be addressed, in particular when moving up the software stack by going beyond *primitives*. For instance, if the miTLS [191] tour de force demonstrated that it is possible to verify protocol implementations, this verification was, in contrast to the numerous primitives implemented in HACL⋆, one-shot; as such it is unclear how to apply similar techniques to a larger class of programs. The verification effort of miTLS also suffered from several limitations. For instance, if miTLS came with computational security proofs, those did not go beyond the record layer. The Low⋆ verification framework, used to implement the HACL⋆ library, has also mostly been been used to verify functions which consume buffers: the problem of verifying complex data-structures and high-level APIs had not been tackled; miTLS, on its side, relied on the combination of several frameworks (F⋆ and Frama-C).

In view of those observations, we decided to tackle the problem of pushing the limits of what has been enabled by projects like HACL⋆, by moving up the software stack to extend program verification to a larger class of realistic programs. We do so through a series of verification projects.

Noise⋆. We note above that the verification of *protocol* implementations is still limited. As a consequence, for our first project, we decided to explore the problem of implementing secure high-level protocol implementations with the Noise⋆ project, which consists of a verified compiler targeting the Noise protocol framework. Noise defines a succinct notation and execution framework for a large class of secure channel protocols, some

of which are used in popular applications such as WhatsApp and WireGuard. This family of protocols currently defines 59 protocols, but could be extended in the future to support protocols with signatures and key encapsulation mechanisms. The Noise⋆ compiler takes *any* Noise protocol, and produces an optimized C implementation with extensive correctness and security guarantees. To this end, we formalize the complete Noise stack in F⋆, from the low-level cryptographic library to a high-level API. We write our compiler also in F⋆, prove that it meets our formal specification once and for all, and then specialize it on-demand for any given Noise protocol, relying on a novel technique called *hybrid embedding*. We thus establish functional correctness, memory safety and a form of side-channel resistance for the generated C code for each Noise protocol. We propagate these guarantees to the high-level API, using defensive dynamic checks to prevent incorrect uses of the protocol. Finally, we formally state and prove the security of our Noise code, by building on a symbolic model of cryptography in F⋆, and formally link high-level API security goals stated in terms of *security levels* to low-level cryptographic guarantees. This implementation is the first comprehensive verification result for a protocol compiler that targets C code, while providing a secure high-level API handling state machine transitions, peer and session management, state serialization and deserialization. It also represents a substantial amount of verification effort: Noise⋆ consists of more than 45k lines of F⋆ code and proofs, and every instantiation generates between 4k and 6k lines of low-level, specialized C code.

Zero-Cost Functors for Program Verification. When moving up to higher-level software quickly comes the problem of implementing generic code. Even more challenging in our case is the fact that we need this code to be low-level, efficient, and verified.

In this second project, we present the design, implementation and evaluation of a set of language-based techniques that allow the programmer to modularly write and verify code at a high level of abstraction, while retaining control over the compilation process and producing high-quality, zero-overhead, low-level code suitable for integration into mainstream software. We again implement those techniques within the F⋆ proof assistant, and specifically its shallowly-embedded Low⋆ toolchain that compiles to C. Through our evaluation, we establish that those techniques were critical in scaling the popular HACL⋆ library past 100,000 lines of verified source code, and brought about significant gains in proof engineer productivity. The exposition of this methodology converges on one final, novel case study: the streaming API, a finicky API that has historically caused many bugs in high-profile software. Using this approach, we manage to capture the streaming semantics in a generic way, and apply it "for free" to over a dozen use-cases. Six of those have made it into the reference implementation of the Python programming language, replacing the previous CVE-ridden code.

Stabilizing Proofs with Dafny-in-Dafny. The above-mentioned projects, by being implemented in F⋆, crucially rely on SMT automation. If this class of automation proved powerful to handle many reasonings and in particular arithmetic reasonings, it also suffers from several issues such as proof instabilities. The ability to use SMT solvers for different classes of programs, like compilers, is also unclear. In this third project, we tackle the problem of verifying parts of the Dafny compiler inside of Dafny itself, and design novel techniques to improve the proof stability by relying on Dafny's modules to implement induction principles.

3.2 Towards Better Scalability with the Verification of Rust Programs

The projects we presented in the previous section allowed us to push frameworks like Low⋆ to their extreme limits; scaling program verification further would require a new generation of tools. We already discussed the limitations of program verification today: let us now focus on the most important points that were revealed through the Noise⋆, zero-cost functors, and Dafny-in-Dafny projects, and which led to the design choices at the root of the toolchain we introduce in the second part of this thesis, the Aeneas framework.

One of the most critical issues we encountered is that reasoning about memory leads to a lot of tedious proof obligations. In the case of Low⋆ in particular, the use of dynamic frames [208] makes the state of Z3 grow extremely quicky, leading to slow, unstable proofs. A potential solution is to use separation logic [168, 210], and the Steel framework [235, 236] recently attempted to do so for F⋆, with some success [237]. But if the use of separation logic can make memory reasoning simpler, it doesn't remove that reasoning altogether. At the same time, one might notice that there are many situations where we *shouldn't have* to reason about memory at all, as aliasing is for instance non-existent or at least extremely limited; this is in particular the case in most of the code generated by the Noise⋆ compiler. Several lines of work have attempted to verify code which manipulates pointers and performs in-place updates while abstracting away reasoning about memory [238–241]. More recently, the emergence of the Rust programming language has opened new avenues for this research, as Rust is an expressive low-level programming language which allows precise memory manipulations, while enforcing a strict aliasing discipline through its linear type system and its mechanism of borrows. In particular, Electrolysis [242], later followed by RustHorn [243] and Creusot [244], made promising attempts at designing mechanisms

to capture the semantics of safe Rust code with *pure* models. This means that even though the verified code uses effectful features such as pointers and in-place updates, the verification itself can be performed on a model which, by the virtue of being pure, abstracts away boring, low-level memory details, allowing the proof engineer to focus instead on the functional behavior of the program.

We decided to follow this approach of generating pure models of safe Rust code, so as to simplify memory reasoning for a large class of programs. Memory is however not the only issue we encountered, as we also suffered from shortcomings stemming from the use of SMT-based automation. The sometimes extremely fast growth of SMT states, leading to unstable or long proofs, can be hard to cope with in practice. The recurrent need to discharge proof obligations which do not fit well within the theories natively supported by SMT solvers (e.g., non-linear arithmetic) also requires regularly writing detailed proof scripts. This combination of unstable proofs and manual reasonings is made worse by the lack of interactivity of SMT-based approaches, which leads to painful debugging sessions in the presence of failing or unstable proof obligations.

We remark that SMT solvers can be extremely good at automating a large class of proof obligations, which is crucial to enabling a smooth proof experience. For instance, we observe that arithmetic proof obligations are pervasive in HACL⋆, as, e.g., every array access requires a bounds check, while every addition requires an overflow check. In the general case, such reasoning can be highly non-trivial and require several steps to, for instance, unfold an invariant, derive crucial facts from several theories, combine them together and finally finish the proof with an arithmetic solver. As SMT solvers like Z3 can be excellent at automating proofs about linear arithmetic, their use can drastically ease the verification experience in those cases.

At the opposite, some other classes of problems are less well-handled by SMT solvers. For instance, in the case of F⋆, reasoning about equalities between sequences is performed by proving extensional equality (i.e., for every index, the cells are equal) and by leveraging SMT patterns; this unfortunately tends to make Z3's context grow extremely quickly. As a consequence, automation regularly falls short, forcing the user to resort to detailed proof scripts with, for instance, the use of the calc constructs which provides a convenient, through pedestrian, way of writing sequences of (in-)equalities. Similarly, reasonings about non-linear arithmetic is one of the motivations behind F⋆'s support for meta-programming [218].

At the same time we remark that when applying good proof engineering practice to make their development scale, proof engineers tend to only leverage a fraction of the power of SMT solvers. Because of the need to tightly control the SMT context

so that the proof time remains low and stable, a common good practice consists in restraining them by, for instance: deactivating unstable heuristics [214], and precisely controlling which facts are available to the solver at a given time (through the use of interfaces in F*, or with finer-grained opaque and reveal instructions in F* and Dafny). As a consequence, the class of problems automated by the use by SMT solvers is in practice not as large as it might seem; for instance, in the case of HACL*, we observe that aside from memory reasoning, SMT automation mostly helps with a class of linear arithmetic proof obligations which don't require the combination of many theories. Moreover, in this context, proof developments leveraging highly automated frameworks are actually not so far from developments leveraging less powerful automation; for instance, revealing the content of an invariant by means of the reveal command in Dafny is not that different from unfolding a definition in an interactive theorem prover. As such and in a way which is reminiscent of the work on the first program verifiers, today's use of automation seems closer to a way of helping skip over mundane proof steps (rewritings, substitutions, propositional reasoning, etc.) while still requiring manual proof scripts for any important reasoning step.

As a consequence, we ask: in the context of a strict development discipline in which proofs are highly structured, would it be possible to enable a proof experience which resembles the one permitted by SMT-based tools, but by using simpler, more controllable and interactive automation? And could we do so by using a framework which makes extensibility one of its key features, so as to allow extending the verifier with custom automation in order to tackle those theories that are not well-handled natively by more general solvers?

Following those observations, we decided to implement a new verification toolchain for Rust programs based on a lightweight functional translation and targeting various theorem provers. We dub this toolchain Aeneas.

The first contribution of Aeneas is a new approach to borrows and controlled aliasing. We propose the Low-Level Borrow Calculus (LLBC), an operational semantics that captures a large subset of Rust programs, and which naturally supports delicate patterns such as two-phase borrows and reborrows. Our semantics is value-based, meaning there is no notion of memory, addresses or pointer arithmetic. Our semantics is also ownership-centric, meaning that we enforce the soundness of borrows via a semantic criterion based on *loans* rather than through a syntactic type-based *lifetime* discipline. We claim that our semantics captures the *essence* of the borrow mechanism rather than its current implementation in the Rust compiler.

We then tweak LLBC to introduce LLBC$^\#$, a symbolic semantics for LLBC which

uses function signatures as summaries to approximate the borrow graph in the presence of function calls. This symbolic execution supports a join operation, which prevents an explosion in the number of states when handling disjunctions in the control-flow, and more importantly allows symbolically executing loops by means of fixed-point computations. We note that in effect, an interpreter for LLBC$^{\#}$ implements a borrow-checker for LLBC: if one symbolically executes all the functions in a program without the execution getting stuck, then the program is borrow-checked. We formalize this claim by proving that a borrow-checker based on our symbolic execution guarantees memory safety.

The last contribution of Aeneas is a translation from LLBC to a pure lambda-calculus, which crucially relies on a symbolic execution according to the semantics of LLBC$^{\#}$. This translation allows the user to reason about the original Rust program through the theorem prover of their choice, and fulfills our promise of enabling lightweight verification of a large class of Rust programs which can contain shared and mutable borrows, functions returning borrows, traits and loops. To deal with the well-known technical difficulty of handling functions which return mutable borrows, we rely on a new novel technical device called *backward functions*. As of today, Aeneas has backends for F*, Coq, HOL4 and Lean, that we present and evaluate.

Part II

Moving up the Software Stack with SMT-based Tools

Chapter 4

Noise*: Verified High-Performance Secure Protocol Implementations

We noted that the verification of cryptographic *primitives* has led to notable success, while the verification of higher-level *protocol* implementations is still limited. In this chapter, we thus study the problem of implementing secure high-level protocol implementations with the Noise* project, which consists in a verified compiler targeting the Noise protocol framework.

4.1 Introduction

Modern distributed applications rely on a variety of secure channel protocols, including TLS, QUIC, Signal, IPsec, SSH, WireGuard, OpenVPN, and EDHOC. Despite the similarity in their high-level goals, each of these protocols makes significantly different design choices based on the target network architecture, authentication infrastructure, and desired security goals. For example, the Transport Layer Security (TLS) protocol is used to secure live TCP connections between clients and servers using the X.509 public key infrastructure. In contrast, the Signal messaging protocol aims to provide strong confidentiality guarantees like post-compromise security [245] for long-running asynchronous messaging conversations between smartphones. All these protocols form a cornerstone of Internet security, so the correctness and security of their varied designs and diverse implementations is a tangible concern.

Security Analyses of Secure Channels. Several prior works establish security theorems for well-known secure channel protocols. However, as protocols get more complex, building and checking pen-and-paper proofs for complete protocols becomes infeasible. To address this, formal verification tools are now routinely applied to

obtain mechanized security proofs for cryptographic protocols. For example, tools like ProVerif [246] and Tamarin [247] have been used to automatically analyze protocols like TLS and Signal [248–250], by relying on abstract symbolic assumptions on the underlying cryptography. Computational provers like CryptoVerif [251] and Computational RCF [252] have also been used to verify some of these protocols, providing more precise security guarantees than symbolic tools, but requiring more human intervention [249, 250, 253, 254].

We refer the reader to [255] for a full survey of computer-aided cryptographic proofs. On the whole, verification tools have now reached a level of maturity that they can analyze the high-level design of most modern cryptographic protocols.

Verified Protocol Implementations. Even if the design of a protocol has been verified, writing a secure implementation remains a challenge. Protocol implementations have to account for many details that are left out of high-level security proofs, such as the crypto library, message formats, state machines, key storage and management, multiple concurrent sessions, and a high-level user-facing API that is easy for non-cryptographers to use. Each of these components has been subject to notable bugs resulting in embarrassing vulnerabilities like Heartbleed [5] and SMACK-TLS [232]. Many of these flaws were not found even through extensive testing.

In response, several works have sought to build high-assurance protocol implementations using formal verification tools. The most notable of these is miTLS [254], a verified reference implementation of the TLS 1.2 protocol in F#, built hand-in-hand with modular proofs of computational security at the code-level. Follow-up works verify efficient C implementations of various components of TLS 1.3, including the TLS packet formats [256], the cryptographic library [128], and the record layer [234]. Other works have built high-assurance protocol implementations in OCaml [257], JavaScript [249, 250], WebAssembly [258], and Java [259].

Despite these advances, verifying a full cryptographic protocol implementation written in a performance-oriented language like C is highly resource-intensive and can take years of work. Consequently such projects have only been attempted for important protocols like TLS. In this chapter, we tackle the problem of generalizing and scaling up the security analysis of protocol implementations in such a way that they can be applied to entire families of cryptographic protocols. Hence, we lower the human effort involved to build verified protocol libraries.

The Noise Protocol Framework. We target verified implementations of the Noise Protocol framework, which provides a general notation and execution rules for a large class of secure channel protocols. The Noise specification [260] currently describes 59

protocols, specifies message-level security properties for each of these protocols, and precisely defines all the cryptographic steps needed to send and receive protocol messages. Although these 59 protocols are centered around Diffie-Hellman and pre-shared keys, the specification language is itself extensible and can easily handle protocols with signatures and key encapsulation mechanisms in the future.

Noise is an ideal target for formal verification in that it covers a large class of similar protocols. For the same reason, it is a challenging target, since we would like to develop generic proofs that apply to all 59 Noise protocols and their implementations, rather than verify each protocol individually.

Several prior works present formal analyses for various Noise protocols [253, 261–263] and multiple open source libraries implement various subsets of Noise. However, until this work, there has been no verified implementation of Noise. Consequently, many security-critical protocol elements, including key management and state machines remain unstudied. Our goal is to develop a library of verified high-performance implementations of Noise protocols in C, with formal proofs of correctness and security that cover all these low-level details.

Our Approach. We build a verified implementation of Noise, following the methodology depicted in Figure 4.1. All our code is written and verified using the F⋆ programming language [75].

We first write a formal specification of Noise in F⋆ (middle column) by carefully encoding the message-level functions described in the Noise specification document [260] and linking them to F⋆ specifications of crypto algorithms. Our specification can be read as an *interpreter* for the Noise protocol notation, and we can use it to execute any Noise protocol. We extend this interpreter with F⋆ specifications of key validation and management and a high-level session API, both which are left unspecified in the Noise document. Hence, we obtain a full specification for the Noise protocol stack, starting from the crypto layer to the user-facing API (see Section 4.2).

Next, we write a low-level implementation of Noise (left column) using Low⋆ [264], a subset of F⋆, and prove that it matches the formal specification. We use the HACL⋆ verified cryptographic library to instantiate the cryptographic layer [127]. We develop a protocol compiler using a novel technique called *hybrid embedding* that allows us to write and verify generic code for all Noise patterns, prove them correct against the interpreter spec once and for all, and then specialize and compile the verified code into standalone C implementations for each Noise protocol (See Section 4.3).

On top of our protocol compiler, we design and build a session management layer that handles multiple sessions in parallel and handles error conditions. We write a verified key storage module that securely stores long-term keys both in-memory and

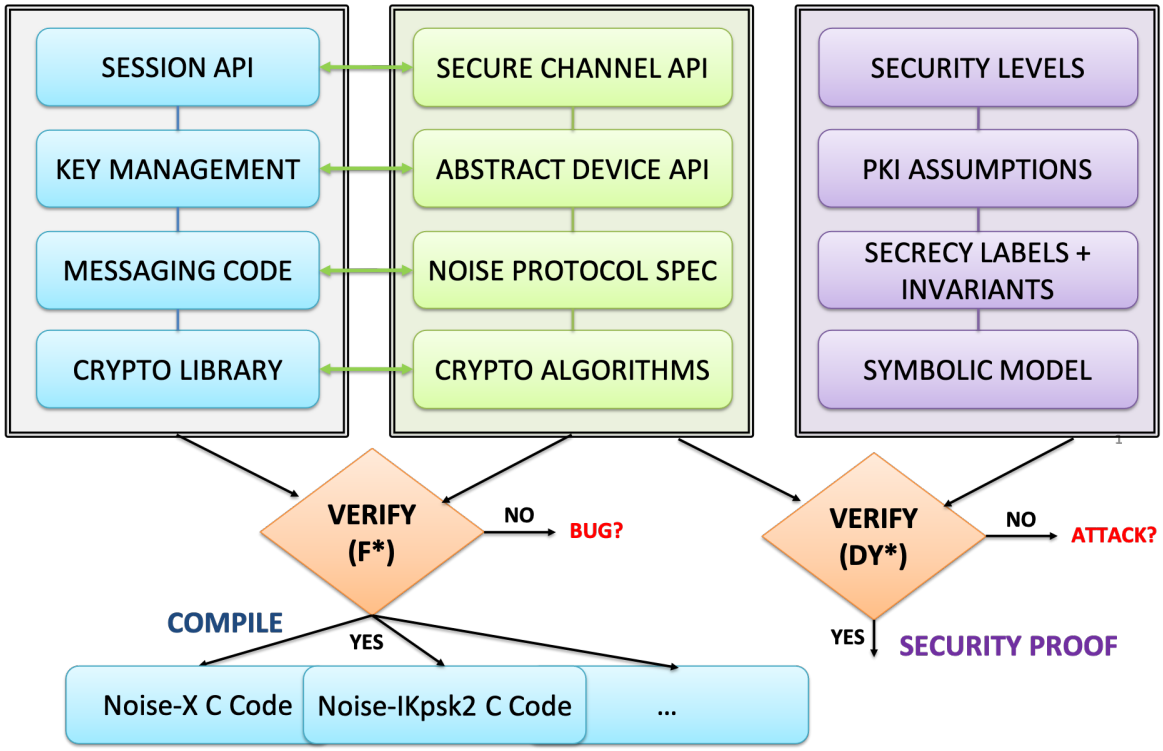

Figure 4.1: Noise* Architecture. Left: Noise protocol stack implemented in Low*; Middle: generic formal specification of Noise in F*; Right: security specifications for each layer using the DY* framework. After verification, the Low* code is specialized and compiled to obtain C code for each protocol.

on-disk. Finally, we build a verified high-level user-facing API that provides a simple, secure, misuse-resistant interface for applications (See Section 4.4).

Our Low⋆ code is verified with respect to our formal specification of Noise, but this does not mean that it is secure. For example, our protocol API may accidentally expose long-term keys to the adversary, or it may allow data from two sessions to be mixed up, which may not violate the Noise spec but would result in serious security vulnerabilities. To fill this gap, we extend our verification with a symbolic security analysis of the full protocol specification using a recent framework called DY⋆ [265]. We set and prove security goals for each layer in our implementation (right column), linking a symbolic model of cryptography all the way to verified high-level API security goals. Notably, our analysis is generic and verifies all Noise protocols in a single proof, unlike prior work which needed to run verification tools on each individual protocol. (See Section 4.5).

Finally, we demonstrate our framework by compiling verified implementations for all 59 Noise patterns and compare the results with prior work (See Section 4.6).

Contributions. The work in this chapter is adapted from a paper published at Security & Privacy in 2022 [266]. I personally came up with the idea of using *hybrid embeddings*, which are in effect an adaptation of Futamura's projection, formalized the specification of the Noise protocol in F⋆, and implemented and verified the protocol compiler written in Low⋆. Abhishek Bichhawat wrote the security proofs of the protocol functions, while I wrote the security proofs for the high-level API, which consist of: the state machines, the session management layer and the key-storage module.

4.2 A Formal Functional Specification of Noise

The Noise Protocol Specification [260] defines a succinct notation and precise execution rules for a family of secure channel protocols that primarily use Diffie-Hellman and pre-shared keys for confidentiality and authentication, yielding a total of 59 protocols with varying authentication and secrecy properties. We begin by an informal overview of the syntax and semantics of Noise protocols, before describing our formal specification of Noise in the F⋆ programming language [75].

4.2.1 Noise Protocol Notation

Three example Noise protocols are shown in Figure 4.2. The message sequence for each protocol is divided into three phases. The first phase (before the dotted line) consists of *pre-messages* exchanged by the two parties out-of-band before the protocol begins. The second phase is the main *handshake* where the two parties exchange fresh key

Protocol Name	Message Sequence	Payload Security Properties ← Auth	← Conf	→ Auth	→ Conf
X	← s				
	...				
	→ e, es, s, ss $[d_0]$	-	-	A1	C2
	→ $[d_1, d_2, ...]$	-	-	A1	C2
NX	→ e	A0	C0	A0	C0
	← e, ee, s, es $[d_0]$	A2	C1	A0	C0
	↔ $[d_1, d_2, ...]$	A2	C1	A0	C5
IKpsk2	← s				
	...				
	→ e, es, s, ss $[d_0]$	A0	C0	A1	C2
	← e, ee, se, psk $[d_1]$	A2	C4	A1	C2
	→ $[d_2]$	A2	C4	A2	C5
	↔ $[d_3, d_4, ...]$	A2	C5	A2	C5

Figure 4.2: Example Noise Protocols and Security Guarantees. X: a one-way authenticated encryption protocol; NX: an interactive Diffie-Hellman key exchange with an unauthenticated initiator; IKpsk2: an interactive mutually-authenticated key exchange using Diffie-Hellman and a pre-shared key. At each stage of a protocol, we note the expected authentication level (A0-A2) and confidentiality level (C0-C5) for messages in each direction (← / →).

material to establish a series of payload encryption keys with gradually stronger security guarantees. Once the handshake is complete, the protocol enters the third *transport* phase where both parties can freely exchange encrypted application messages in both directions.

The handshake is described as a sequence of messages between an initiator (I) and a responder (R), where each message is as a sequence of *tokens*. Each part Each participant maintains a *chaining key* k_i that it uses to derive the *payload encryption key* at each step; both of which are initially set to public constants derived from the protocol name. The chaining key evolves as each handshake token is processed.

Consider a handshake between I and R, where I has a static Diffie-Hellman key-pair (i, g^i) and generates an ephemeral key-pair (x, g^x); R has a static key-pair (r, g^r) and ephemeral key-pair (y, g^y); and the two may share a pre-shared key psk. Then the semantics of each token sent from I to R is as follows (tokens in the reverse direction are handled similarly):

- e: means that I includes g^x in the message;
- s: I includes its static public key (g^i) in the message, encrypted under the current payload encryption key;
- es: I computes the ephemeral-static Diffie-Hellman shared secret g^{xr} and mixes it

into the chaining key c_i, obtaining a new chaining key c_{i+1} and payload encryption key k_{i+1};

- se: I mixes the static-ephemeral shared secret g^{iy} into c_i;
- ee: I mixes the ephemeral-ephemeral secret g^{xy} into c_i;
- ss: I mixes the static-static shared secret g^{ir} into c_i;
- psk: I mixes the pre-shared key psk into c_i.

After processing each sequence of tokens according to the above rules, at the end of each message, the sender (I) also includes a (possibly empty) payload encrypted under the current payload encryption key. These payloads are implicit in Noise notation, but we note them explicitly ($d_0, d_1, ...$) in Figure 4.2.

On receiving a message constructed using the above rules, the responder R performs the dual operations to parse the remote ephemeral key (e), decrypt the remote static key (s), and computes the same sequence of chaining and payload encryption keys to decrypt the payload. In addition to the keys, each participant also maintains a hash of the protocol *transcript*, which is added as associated data to each encrypted handshake payload (to prevent handshake message tampering.)

Example X: One-Way Encryption. The protocol X is a one-way protocol that encrypts data in a single direction, from an initiator I to a responder R. As such, this protocol can be considered a replacement for constructions like NaCl Box [267] or HPKE [268] for encrypting files or one-way messages.

We now break down the notation for this protocol, which appears in Figure 4.2 under "message sequence". The pre-message token s, assumes that I has received R's *static* public key g^r before the handshake. The handshake itself consists of a single message (from I to S) with four tokens (e, es, s, ss) followed by an encrypted payload (d_0). Here, ephemeral-static Diffie-Hellman (es) serves to provide confidentiality for k_1 (even if I's static key were compromised), whereas static-static Diffie-Hellman (ss) is used to authenticate I. After the handshake, I can send any number data messages ($d_1, d_2...$) to R, using the final payload encryption key.

Example NX: Server-Authenticated Key Exchange. The protocol NX is a unilaterally authenticated key exchange protocol, where R is authenticated but I is not. Hence, this protocol can be seen as a replacement for TLS as it is used on the Web. The main difference from X is that it has no pre-messages, and has a second message that uses ephemeral-ephemeral Diffie-Hellman (ee) to provide forward secrecy.

We can extend NX to a mutually-authenticated protocol by adding a third handshake message that uses I's static key (se). This yields a different Noise protocol called XX, which is one of the protocols used in WhatsApp. Both NX and XX are single round-trip (1-RTT) protocols since the initiator has to wait for the response before it can send its first encrypted message. However, in scenarios where I already knows R's static public key (g^r) via a pre-message, it can use this prior knowledge to start sending data with the first message (0-RTT), but with different secrecy and confidentiality guarantees.

IKpsk2: Mutual-Authentication and 0-RTT. The IKpsk2 protocol, which is used by the WireGuard VPN, supports mutual authentication and 0-RTT by relying on both Diffie-Hellman and pre-shared keys, and hence provides some of the strongest security properties among all Noise protocols.

The protocol starts like X but includes authenticated messages in both directions; it uses four Diffie-Hellman operations and also a pre-shared key in the second message for additional protection against compromised static keys (and future quantum adversaries). Removing the psk token yields a protocol called IK, which is also used in WhatsApp.

4.2.2 Formalizing Noise in F*

We define a series of F* types that encode the syntax of Noise protocols. We define algebraic datatypes (enumerations) for pre-message and message tokens. We then define a handshake pattern as a record type containing a protocol name, a pre-message from I to R (premessage_ir), a pre-message from R to I (premessage_ri), and a list of handshake messages in alternating directions (first I to R, then R to I, and so on):

```
type premessage_token = | PS | PE
type message_token = | S | E | SS | EE | SE | ES | PSK

type handshake_pattern = {
  name : string;
  premessage_ir : option (list premessage_token);
  premessage_ri : option (list premessage_token);
  messages : list (list message_token);
}
```

We also define some convenient notations in F* to construct a handshake_pattern. For example, IKpsk2 is written as:

```
let pattern_IKpsk2 =
  hs "IKpsk2" [
    ~<<~ [PS];
```

```
    ~>~  [E; ES; S; SS];
    ~<~  [E; EE; SE; PSK]
  ]
```

The Noise specification defines a set of syntactic validity rules to ensure that the resulting protocols are implementable and secure. An example functional constraint is that a protocol should not use the token ee before e has been sent in both directions. A security constraint is that a session key based on a psk token should not be used for encryption unless an e has also been sent (otherwise there could be encryption nonce reuse.) We encode these rules as a boolean function over handshake patterns, and check that it holds for all 59 patterns.

```
val well_formed: handshake_pattern -> bool
```

Types for the Handshake State. To formalize the execution rules, we closely follow the Noise specification by defining the handshake state and functions over this state. Each type and function in our specification is parameterized by a config type specifying three cryptographic algorithms: a Diffie-Hellman group, an AEAD encryption scheme, and a hash algorithm:

```
type config = dh_alg * aead_alg * hash_alg
```

The cipher_state type consists of an AEAD key and a counter; it can be used for AEAD encryption and decryption:

```
type cipher_state = {
  k : option aead_key;
  n : nat;
}
```

The symmetric_state type represents the cryptographic state of a Noise handshake. It contains a hash of the protocol transcript (essentially all the message tokens processed so far), the current session key, called chaining_key in Noise, and a cipher_state (derived from the chaining_key) which is used for encrypting static keys and payloads during the handshake:

```
type symmetric_state (cfg : config) = {
  h : hash cfg;
  ck : chaining_key cfg;
  c_state : cipher_state;
}
```

The main handshake_state type contains the full state of a Noise handshake for a given participant; it includes the current symmetric_state and all the private, public, and shared keys currently known to the participant:

```
type handshake_state (cfg : config) = {
  sym_state : symmetric_state cfg;
  static : option (keypair cfg);
  ephemeral : option (keypair cfg);
  remote_static : option (public_key cfg);
  remote_ephemeral : option (public_key cfg);
  preshared : option preshared_key;
}
```

Message Processing Functions. The Noise specification document describes a series of functions over the three state objects, which we faithfully encode in F*. The highest-level operations defined by the document are functions for sending or receiving one handshake or data message. We describe the F* code for the handshake sending functions below.

First, we define a function that implements the sending operation for a single token as a case analysis over the 7 possible tokens (we show two cases below):

```
let send_message_token
  (cfg : config) (initiator is_psk : bool)
  (tk : token) (st:handshake_state cfg) :
  result (bytes * handshake_state cfg) =
  match tk with
  | S ->
    begin match st.static with
    | None -> Fail No_key
    | Some k ->
      begin match encrypt_and_hash cfg k.pub st.sym_state with
      | Fail x -> Fail x
      | Res (cipher, sym_st') ->
        Res (cipher, { st with sym_state = sym_st'; }) end
    end
  | EE -> dh_update cfg lbytes_empty st.ephemeral st.remote_ephemeral st
  | ...
  end
```

The function send_message_token takes as arguments: a config, a boolean flag indicating whether the sender is the initiator, a boolean flag indicating whether the current protocol uses psk, a token tk and a handshake state st. If the token is an S, the code finds the sender's static key (st.static), encrypts it and adds to the transcript hash

(encrypt_and_hash), returning the ciphertext (cipher) and the updated handshake state. If the token is an EE, the sender reads its ephemeral private key (st.ephemeral), the peer's ephemeral public key (st.remote_ephemeral) and calls the dh_update function that computes the Diffie-Hellman shared secret, mixes it into the current chaining_key, and returns an empty bytestring and the updated handshake state. The other cases are similar.

Building on this token-level function, we then write a function send_message_tokens that recursively calls send_message_token to process an arbitrary list of tokens, and use it to define a high-level function send_messagei for sending the i-th handshake message in a handshake_pattern.

A similar sequence of functions builds up to the top-level handshake receive function recv_messagei. Using these and other message-level functions in our specification, we can construct or process any pre-message, handshake message, or application data message in a Noise protocol.

Comparison with Prior Noise Models. Three features of our specification are notable. First, our F⋆ code is executable and precisely matches the Noise specification at the byte level. Indeed, by linking our specification code with the HACL⋆ cryptographic library, we are able to extensively test our specification against test-vectors from other Noise implementations. Second, we use recursive functions to model protocols and messages of arbitrary length, even though in practice, we may only care about the 59 protocols in the current Noise specification. Third, our code is structured as a *protocol interpreter*, and hence provides a single generic functional specification for *all* Noise protocols.

These three features are in contrast with prior formal models of Noise protocols that were written for various security analyses [253, 261–263]. These models ignore many low-level protocol details, are not precise at the byte level, and are not testable. Their modeling languages cannot handle generic recursion or protocol interpreters, and so require a separate model for each Noise protocol. We believe our F⋆ specification more closely captures the spirit of Noise and serves as a formal companion to the Noise specification.

4.2.3 Noise Protocol Security Guarantees

Different Noise protocols offer different security guarantees. Even within a single protocol, the confidentiality and authentication guarantees obtained by the initiator and responder often differ. These guarantees typically improve with each handshake message and stabilize after the handshake completes. For example, IKpsk2 allows

application data to be sent both during the handshake (d_0, d_1) and after the handshake $(d_2, d_3, ...)$, and each of these messages has different security guarantees. The Noise specification [260] defines 3 levels of authenticity (A0-A2) and 6 levels of confidentiality (C0-C5). Figure 4.2 lists the security levels at each stage of our three protocol examples, and Appendix A lists them for all 59 Noise patterns.

Payload Authentication Properties. The three authentication levels are:

- **A0**: No authentication
- **A1**: Sender authentication vulnerable to Key Compromise Impersonation (KCI) attacks
- **A2**: Sender authentication without KCI attacks

Consider a Noise protocol session between A and B, where B receives a message M at authentication level A2 (supposedly) from A. If B successfully decrypts this message, it has the guarantee that the message was indeed sent by A, unless the long-term static key of A (static Diffie-Hellman private key and/or PSK) has been compromised (i.e., leaked to the attacker) before the message was received. Authentication level A1 is weaker: it only guarantees message authenticity if the static keys of both A and B are non-compromised.

For a more formal illustration, in a prover like ProVerif, authentication level **1** would correspond to a security query written in terms of events triggered by the sender, receiver, and the adversary. The sender A triggers Sent(A,B,M) before sending a message; the receiver B triggers Recv(B,A,M) after processing the message; the adversary triggers LongTermCompromised(P) whenever it compromises the static keys of a principal P. (We assume that ephemeral keys are never compromised.) The resulting security query is written as follows:

```
query A:prin, B:prin, M:bitstring;
  event(Recv(B,A,M)) ⟹
    event(Sent(A,B,M)
    || event(LongTermCompromised(A))
    || event(LongTermCompromised(B))
```

The query for authentication level **2** simply removes the last line:

```
query A:prin, B:prin, M:bitstring;
  event(Recv(B,A,M)) ⟹
    event(Sent(A,B,M)
    || event(LongTermCompromised(A))
```

For reference, these queries correspond closely to the queries generated by a prior analysis of Noise in ProVerif [262].

For example, in NX, the initiator is never authenticated, so messages in the forward direction (→) in Figure 4.2 always have authentication level A0. The responder is fully authenticated and so its messages to the initiator are at level A2. In X and IK, the first message is authenticated by the initiator, but authentication is based on static-static Diffie-Hellman (ss), which means that if the responder's static key is compromised, an attacker can impersonate the initiator to the responder, resulting in a KCI attack. Hence, the authentication level is A1 for forward messages (→), until the third message when the static-ephemeral Diffie-Hellman (se) token strengthens the initiator's authentication level to A2.

Payload Confidentiality Properties. The six confidentiality levels, in increasing order of strength, are as follows:

- **C0**: No confidentiality
- **C1**: Confidentiality only against passive adversaries
- **C2**: Confidentiality against active adversaries, with weak forward secrecy against sender static compromise
- **C3**: Weak forward secrecy against sender and receiver static compromise
- **C4**: Strong forward secrecy unless sender static was compromised before message
- **C5**: Strong forward secrecy

Of these, the first two levels offer very weak confidentiality, in that an active network adversary can read a payload sent at level C0 or C1. Levels C2-C5 offer incremental degrees of forward secrecy, depending on which subset of static keys may be compromised and when. C2 offers confidentiality as long as the sender's ephemeral key and the recipient's static keys remain non-compromised. C3 additionally allows the receiver's static key to be compromised as long as the peer ephemeral public key at the sender corresponds to an non-compromised ephemeral private key at the recipient. C4 allows the sender and recipient's static keys to be compromised after the message is sent. C5 provides confidentiality even if the sender's static keys were compromised before the message was sent.

In a tool like ProVerif, encoding forward secrecy properties requires the use of phases to enforce an ordering between protocol messages and compromise events. For example, we would run the target protocol session in phase 0 and transition to phase 1 at the

end. We would then allow the attacker to compromise static keys in both phase 0 and phase 1, and state secrecy queries for each confidentiality level in terms of when these keys can be compromised. (As usual, we disallow the compromise of ephemeral keys.) Hence, to model level C4, we write a ProVerif query of the form:

```
query A:prin, B:prin, M:bitstring;
    (attacker_p1(M) && Sent(A,B,M)) ⟹
        event(LongTermCompromised_p0(B))
        || event(LongTermCompromised_p0(A))
```

That is, messages sent from A to B are confidential in phase 1 unless the static keys of A or B were compromised in phase 0. The query for C5 is stronger, it removes the last disjunct, and hence guarantees confidentiality even if A were compromised in phase 0 (when the session is still running.)

In Figure 4.2, X offers confidentiality at level C2 because there is no fresh ephemeral provided by the recipient. NX offers strong forward secrecy at level C5 for messages to the responder, but only level C1 for messages to the unauthenticated initiator. IKpsk2 provides level C5 confidentiality in both directions from the third message. However, the first message only offers level C2 (like X) and the second message only offers level C4 since an attacker who knows the responder's static private key and PSK will be able to forge the first message, record the second message, and later compromise the initiator's static key to obtain the session key and decrypt the payload.

We define an F* function that computes the authentication and confidentiality levels for each message in each handshake_pattern (see Appendix A). We confirm that it agrees with the Noise specification on the 38 protocols annotated in the document, and we also compute levels for the 21 PSK patterns not annotated in the specification. In Section 4.5, we show how these security levels are mapped to precise security goals stated as trace properties and we prove that our protocol specification meets these goals.

4.2.4 A High-Level API for Noise

A full protocol implementation has to handle many security-critical details beyond message processing. For example, in the NX protocol, when the initiator receives the responder's static key in the second message, it has to *validate* this key. Otherwise, there is no guarantee it is talking to the intended responder and all authenticity and confidentiality guarantees are lost. Similarly, in X and IKpsk2, the initiator static key needs to be validated against some database of known initiators. In PSK-based protocols like IKpsk2, the responder does not know what PSK to use until it sees the initiator's

static key; so we need a way for the responder to dynamically retrieve and validate a PSK based on a protocol message. An implementation that skips or incorrectly implements these key validation steps becomes vulnerable to serious attacks. However, none of these validation steps are documented in the Noise specification and so are left for the application layer to handle.

It is unrealistic to expect an application programmer who uses Noise to have the intimate knowledge needed about a specific Noise protocol in order to directly use the messaging functions, perform all the required validation steps, and know when it is safe to send or receive data.

We address this gap by formally specifying (and implementing) a high-level API that combines several layers: a session-based API that hides message-level protocol details, secure key storage with user-provided policies for key management, built-in validation steps and a defensive user-friendly interface that provides clear guidance on when it is safe to send or receive data over a Noise session. For example, sending secret application data after the first message of NX would be disastrous, but may be safe with IKpsk2. Section 4.4 describes our implementation of this high-level API in C.

4.3 Implementing a Noise Compiler in Low⋆

Our specification (Section 4.2) may run via the OCaml backend of F⋆, which we use for testing and spec-validation purposes. This execution path suffers, however, from slow performance: in F⋆ specifications, integers compile as infinite-precision bignums; sequences compile to persistent functional lists; and execution relies on OCaml's runtime system and garbage collector.

We now set out to write a low-level, efficient implementation of Noise protocols that does not suffer from such performance shortcomings. This section focuses on a novel technique called "hybrid embeddings", a key technical ingredient that allows us to author low-level code that remains parametric over the choice of Noise pattern, in a fashion similar to the interpreter. With hybrid embeddings, we verify the low-level code once then generate for free any number of specialized implementations for any Noise patterns: doing so, we minimize the verification effort while still guaranteeing low-level performance.

4.3.1 Warm-up: Low⋆ implementation of SS

For our efficient, low-level implementation of Noise protocols, we use Low⋆. Low⋆ is a subset of F⋆; or, said differently, Low⋆ is a *shallow embedding* of a well-behaved subset

of C into F*. Thanks to F*'s powerful effect system, Low* defines a CompCert-like C memory model, which captures heap- and stack-based allocations. A set of distinguished types, combinators and libraries provides users with working tools to operate on mutable arrays, machine integers, const pointers, and so on. Low* has been used for cryptographic libraries [127, 233], providers [128], protocol record layers [269, 270] and parsers [256].

In contrast to Section 4.2, where functions were pure, Low* functions use a new set of effects: Stack and ST. Consider the function that performs the required processing for the SS token.

```
inline_for_extraction noextract
let send_SS_m (nc: iconfig) (ssdhi: ssdh_impls nc)
 (smi: meta_info) (initiator: bool) (is_psk: bool)
 (st: valid_send_token_hsm nc is_psk SS smi):
 Stack (rtype (send_token_return_type smi is_psk SS))
 (requires (fun h ->
   live h st.static /\ live h st.remote_static /\
   not (g_is_null st.static) /\ not (g_is_null st.remote_static) /\
   loc_disjoint (loc st.static) (loc st.remote_static) /\ ... /\
   sym_state_invariant st.sym_state /\ nc.dh_pre /\ ...))
 (ensures (
   let st0_v = eval_handshake_state_m h0 st smi in
   let st1_v = eval_handshake_state_m h1 st (smi_init_sk smi) in
   let r_v = Spec.send_message_token initiator is_psk SS st0_v in
   match to_prim_error_code r, r_v with
   | CSuccess, Res (..., st1'_v) -> st1_v == st1'_v /\ ...
   | CDH_error, Fail DH -> True
   | _ -> False))
= [@inline_let] let priv = hsm_get_static st in
  [@inline_let] let pub = hsm_get_remote_static st in
  ssdhi_get_dh_update ssdhi BS.lbytes_empty smi priv pub st
```

Many of the parameters resemble the ones we saw earlier (Section 4.2). The iconfig, for *implementation* configuration, extends a spec-level config with low-level specific preconditions such as "our DH implementation requires AVX2". The ssdhi parameter contains our choice of implementation for cryptographic operations related to the symmetric state and DH; the Low* code is not only generic over the choice of algorithm (like the earlier specification), it is also generic over the choice of implementation. As an example, if the iconfig commits to Curve25519 for the DH algorithm, our code can operate either with HACL*'s Curve51 or Curve64 implementation. The smi parameter stands for "state meta-information"; it contains statically-known information, such as whether at this point of the handshake a symmetric key has been derived or not; and it

also contains the nonce (sequence number) to be used for the cryptographic operations. Finally, initiator and is_psk are similar to the parameters we saw earlier (Section 4.2).

The function signature exhibits typical features of Low*. The st argument represents the low-level state of the protocol, which can be reflected in a given heap h0 as a high-level state, using eval_handshake_state_m h0 st smi. The Stack return effect indicates that the function is valid vis-à-vis the C memory *and* only performs stack allocations (this latter restriction can be lifted by using the ST effect). The pre-condition covers spatial (disjointness) and temporal (liveness) preconditions; as well as functional correctness requirements, such as the symmetric state invariant and the implementation-specific preconditions. In the post-condition, we elide memory-related predicates (e.g.: only the protocol state is modified by a call to this function) for clarity. We focus instead on functional correctness: st0_v reflects low-level state st as a spec-level state before calling send_SS_m; similarly, st1 reflects st after calling the function. If we execute the interpreter on st0 and obtain st1', then both st1' and st1 coincide, i.e., if the specification guarantees success, so does the low-level implementation with the same result; if the specification errors out, so does the low-level implementation; no other outcome is allowed.

4.3.2 A Meta-Programmed Low* Implementation

Inspired by the generic spec-level interpreter, we now write an *even more generic* low-level function that not only works for any choice of algorithm, implementation, responder and PSK, but also works for any Noise token.

```
inline_for_extraction noextract
let send_message_token_m nc ssdhi smi initiator is_psk
  tk st out: (rtype (send_token_return_type smi is_psk tk))
= match tk with
  | S -> send_S_m nc ssdhi smi initiator is_psk st out
  | E -> send_E_m nc ssdhi smi initiator is_psk st out
  | ... -> ... (* identical for SS, EE, SE, ES, PSK *)
```

The send_message_token_m function above attains the same level of genericity as the specification. Even the return type of the function is generic: send_token_return_type captures the fact that SS returns an error code (for DHs that compute to 0), whereas S does not. (Here, our specification is more precise than the Noise specification, which leaves it up to the user to determine whether a DH that computes 0 is an error.) Our function can thus be used for *all* Noise protocols: the initial `match` acts as an interpreter, examines the Noise token, then dispatches execution to a suitable set of Low* functions.

Our style saves a tremendous amount of verification effort: rather than replicating the effort for 59 protocols, we extract the commonality, capture it with dependent types, and proceed to write send_message_token_m once and for all. The challenge now remains to ensure that the function generates valid C code that eliminates all runtime checks on the nature of the token.

To that end, we rely on implicit staging and compile-time partial evaluation via F*'s normalizer. (We use meta-level and compile-time interchangeably.) The first six parameters of the function are meta-parameters: once a Noise protocol is chosen, their concrete value is known at compile-time; and the F* compiler is capable of performing enough partial evaluation at compile-time that all uses of these parameters disappear *before* the code is even extracted to C. We indicate that the function features meta-level computations with the _m suffix.

Consider, for instance, the X protocol we saw earlier. At compile-time, we pick concrete values for the choice of algorithms (nc) and implementations (ssdhi). For the first handshake message, we call send_message_token_m, with smi.has_key = false, smi.nonce = 0, initiator = true, is_psk = false and of course tk = E. Thanks to the inline_for_extraction keyword, F* reduces the definition of send_message_token_m; the `match` reduces away, leaving only a call to send_E_m. This latter function itself further reduces: for instance, any statement of the form if is_psk disappears, meaning we ignore the symmetric key generation induced by PSK patterns. Once partial evaluation is done, the code contains only the bare minimum set of operations needed for the first token E of the X protocol, and all meta-parameters are gone.

4.3.3 Hybrid Embeddings

Looking back at Section 4.2, we can think of our earlier specification as an interpreter for Noise patterns; or, dually, as an evaluator defining the semantics of a deeply embedded domain-specific-language (DSL), in our case the language of Noise patterns. Unlike shallow embeddings, deep embeddings operate on a *representation* of the target language within the host language; doing so, they enjoy a great deal of flexibility since they are not confined to the syntax of the host language.

The `match` in the function above is a meta-level match that operates on the deeply embedded representation of Noise patterns, and gets partially evaluated away. We dub this style a hybrid embedding: the meta-level code operates over a deep embedding (the Noise patterns), but after partial evaluation, all that is left is a shallow embedding (the Low* code).

The hybrid style allows us to stage and automate the production of Low* code;

rather than writing Low* code by hand, we embed at meta-time a protocol compiler that executes on F*'s compile-time reduction facilities. This style is already useful for send_message_token_m; but there is no reason to limit ourselves to simple matches and ifs. We now show how to execute arbitrary pure F* code at meta-time, including recursion, to completely automate the production of a specialized Noise protocol instance.

```
[@@ strict_on_arguments [5]] inline_for_extraction noextract
let rec send_message_tokens_m (nc: iconfig) smi initiator
  is_psk pattern st outlen out =
 match pattern with
 | Nil → success _
 | tk :: pattern' →
   [@inline_let] let tk_outlen = token_message_vs nc smi tk in
   let tk_out = sub out 0ul tk_outlen in
   let r1 = send_message_token smi initiator is_psk tk st tk_out in
   if is_success r1 then
    let outlen' = outlen -! tk_outlen in
    let out' = sub out tk_outlen outlen' in
    [@inline_let] let smi' = send_token_update_smi is_psk tk smi in
    let r2 = send_message_tokens_m send_message_token
      smi' initiator is_psk pattern' st outlen' out' in
    compose_return_type smi is_psk true pattern' tk r2
   else
    compute_return_type smi is_psk true tk pattern' r1
```

The function above now operates over a *list* of tokens; that is, it generates Low* code for an entire Noise handshake pattern. Naturally, the function cannot extract as-is: operating over pure, persistent lists in low-level efficient, idiomatic C is a no-go; hence the noextract keyword. The goal is to ensure that the subset of send_message_tokens_m that performs a (pure) recursion over the argument pattern (denoted in bold) is always evaluated away at compile-time when applied to constant arguments. To this end, we allow F* to unfold recursive definitions (elided); to prevent infinite compile-time recursion, we restrict the unfolding to applications where the fifth argument (pattern) is concrete, via the strict_on_arguments keyword. The inline_let attribute indicates pure computations to be inlined at extraction-time. We use the keyword for meta parameters or constants computed from meta parameters.

The function is verified once and for all, meaning that we now have a verification statement for *any* list of noise tokens. At extraction-time, the user applies the function to five concrete arguments. If pattern is [E; ES; S; SS], then after a few steps of reduction, we obtain:

```
let r1 = send_message_token ... E ... in
if is_success r1 then ...
  let r2 = send_message_tokens_m ... [ ES; S; SS ] ... in ...
```

As computing E always succeeds, is_success r1 reduces to true, in turn eliminating the else branch entirely. Partial evaluation then continues until all meta-level code has disappeared; structural recursion over the list of tokens is over; and all that is left is a sequence of efficient Low* calls that implements the specification for the given list of tokens.

We use this style of hybrid embedding all throughout our low-level protocol code implementation, which allows us to substantially reduce the verification effort. The following section (Section 4.4) shows how to extend this style to generate the entire state machine of a Noise protocol.

4.3.4 Hybrid Type Definitions and Function Signatures

We use hybrid embeddings further to optimize internal type definitions and user-facing functions.

For type definitions, we insist on generating C code that contains no superfluous fields. This is useful not only in case the code's internals are audited; but also to ensure that no extra space is consumed in, e.g., the internal state of the handshake. To that end, our types reduce at meta-time; consider, for instance:

```
type handshake_state_t nc smi ... is_psk ... = {
  ...
  psk: if is_psk then lbuffer ... else unit;
  ...
}
```

If the chosen Noise protocol requires it, the psk field is an array of bytes. If the Noise protocol does not use a PSK, the meta-programmed type reduces to unit, which is then guaranteed to be eliminated by KReMLin [264], the Low*-to-C compiler. This eliminates an always-NULL, superfluous field.

For user-facing functions, we apply a similar design pattern and ensure that no "dummy" arguments are ever offered in the public API: such arguments cause user confusion, make code reviews more difficult, and generally diminish trust in our API. Anticipating slightly, consider this initialization function that we present as part of our user-facing API (Section 4.4):

```
let session_p_create (idc : valid_idc) (initiator : bool) ...
  (dvp : device_p idc) (peer_id : opt_pid_t idc initiator) : ST ... = ...
```

As mentioned in Section 4.2, we may not immediately know a peer's identity: whether peer_id is needed at initialization-time depends on the protocol. Rather than rely on an implicit invariant that peer_id will be ignored for some patterns, we instead rely on a meta-programmed type opt_pid_t. In the case of XX, the type opt_pid_t becomes unit. In the case of IKpsk2 for the initiator, the type becomes lbuffer uint8. KreMLin guarantees that function arguments of type unit are eliminated: this means we offer a custom API for each Noise protocol. This directly supports our goal of generating robust user-facing APIs that leave no room for user error.

4.4 A Complete Verified Noise Library Stack & API

Section 4.3 describes the core handshake actions, as captured by the Noise Protocol Framework. Yet, this forms only a small, core part of a Noise library. We now review the remainder of our Noise Protocol implementation and describe the many APIs and library features we wrote in order to provide a complete, self-contained, user-proof, verified Noise protocol stack.

A meta-programmed state machine. The core handshake actions (Section 4.3) each implement a single line of a Noise Pattern. We now tie together these individual protocol actions into two state machines: one for the initiator and one for the responder. These basic state machines are trivially induced by the steps of the handshake: they are linear, and each valid transition advances the initiator or responder to their next step.

The send_message_tokens_m function from Section 4.3 takes many run-time parameters; we group them in a single type definition, dubbed state_t. The state also holds the current step in the handshake, i.e., the current state of the machine. Continuing with hybrid embeddings, a generic function state_t_handshake_write_m advances the state machine, and returns a fresh state_t, for any choice of pattern, step i, or initiator *vs.* responder.

```
(* The low-level state machine type: encapsulates keypair, chaining
   hash state, symmetric state, current handshake step, psk, etc. *)
val state_t: isconfig -> initiator:bool -> Type0

(* Simplified signature *)
val state_t_handshake_write_m (isc: isconfig) (smi: smi)
  (i: nat { i < isc.pattern.messages })
  (payload_len: size_t) (payload: lbuffer uint8)
  (st: state_t isc (i%2=0) { ... })
  (outlen: size_t) (out: lbuffer uint8):
  Stack (s_result_code (st:state_t isc (i%2=0) { ... }))
```

The signature of the function is familiar; the earlier iconfig is now wrapped in an "implementation state config" isc, which contains the entire noise pattern, along with meta-parameters that determine the shape of the final C struct (Section 4.3.4). The function is once again written in the hybrid embedding style; the meta-parameter i allows the caller to specialize the function for the i-th step of the handshake; this in turns allows us to compute, at compile-time, whether the message originates from the initiator (i%2=0) or the responder (i%2=1). The meta-parameters also determine the nature of the return type, which is derived from the series of return types for each token. The function *returns* a fresh state st1 under the successful Res case. In a fashion similar to send_message_tokens_m, the low-level stateful function coincides with the outcome st1'_v of the spec-level interpreter. (Full definitions can be found in [271].)

The parameter i represents the current step of the handshake at meta-time; but this information is also carried at run-time within the state st. A static precondition requires the compile-time i to be consistent with the step stored at run-time within st. This key technical trick enables meta-time computations over the step i, which allows us to write a single transition function. The function can be specialized at meta-time for any step i; doing so produces a Low⋆ function that can only be called when the current run-time step coincides with the meta-time i.

Equipped with this extremely generic function, we now use the hybrid embedding style to meta-program state machine management: at compile-time, we generate a series of run-time tests for each (statically-known) possible state of the handshake; if a run-time test succeeds, the code proceeds to execute state_t_handshake_write_m, specialized at compile-time for the specific step of the handshake. The result is a higher-level function that can generate the state machine of either the initiator or the responder, for *any* Noise pattern. We have effectively embedded at meta-time within F⋆ a compiler that from a deeply embedded Noise pattern generates the corresponding shallowly-embedded Low⋆ state machine.

A user-proof state machine. As it stands, the state machine cannot be exposed to the user. First, it returns a new state, rather than modifying a heap-allocated state through a pointer; second, it does not record stuck states, meaning that the user can make a mistake by ignoring the Failure and calling the function a second time.

We now transform this low-level state machine into a user-proof one. In the process, we also enrich the API with features for device, peer and key management. We dub this second API layer the "device API". We encapsulate the earlier state_t in a device state dstate_t, which handles Low⋆ region-based memory management and ownership (elided), holds session and peer names (provided by the user), and maintains a device state for peer management.

```
[@CAbstractStruct] noeq type dstate_t idc =
  | Initiator: state:state_t idc.isc true -> session_name:name_t
    -> peer_name: name_t -> device: device_t -> ...
    -> dstate_t ...
  | Responder: state:state_t idc.isc false -> (* similar *)
noeq type dstate_p ... = B.pointer_or_null dstate_t
```

Introducing dstate_p, a potentially-null pointer, serves several purposes: the C code becomes more idiomatic, now manipulating a pointer to a structure instead of passing structures by value; we can now have a NULL case which accounts for errors, e.g. a point at infinity showing up at initialization time; and we can introduce a modicum of abstraction, by using the CAbstractStruct keyword which instructs KreMLin to only emit a typedef in the generated header, thus preventing clients from directly allocating or accessing a dstate_t. We lift the state machine to operate on dstate instead of state, and obtain the following signature.

```
val handshake_write_m (idc: valid_idc)
  (payload_len: size_t) (payload: lbuffer uint8 payload_len)
  (st: dstate_p idc) (outlen: size_t) (out: lbuffer uint8 outlen):
  ST ds_error_code_or_success (requires (fun h0 -> ...))
  (ensures (fun h0 res h1 ->
    (* omitted: modifies clause, liveness, invariants, etc. *)
    let st_v0 = dstate_p_v h0 st in
    let payload_v = as_seq h0 payload in
    let res_v = handshake_write payload_v st_v0 in
    match res with
    | CSuccess -> Res? res_v /\ (
      let Res (out'_v, st1'_v) = res_v in
      dstate_p_v h1 st == st1'_v /\ as_seq h1 out == out'_v /\
      ... /\ not (dstate_p_is_gstuck h0 st))
    | _ -> ... dstate_p_is_gstuck h1 st))
```

The only meta-parameter is idc, which subsumes the previous state config, and indicates whether we are compiling the initiator or the responder's state machine. Unlike state_t_handshake_write_m, this state machine from the device layer is safe to use from C. If an error happens, we modify the step number to a special value that indicates that the machine is stuck, before returning an error. Any further attempt to use this state will leave the machine in the error (stuck) state.

Device API and Session Management. In addition to the state machine, the device state dstate_t also encapsulates device management. A device holds a set of peers, along with a table that indexes them by identifier; it also holds the local static

identity, and provides a high-level API which enables the user to add, lookup, update or remove peers. Each peer contains detailed information, such as their remote static and pre-shared keys. The library is written from scratch, since the existing Low* libraries for, e.g., linked lists, were proof-of-concept-quality and not intended to be used within a large development. The result is a relatively simple API, wherein the user provides a private key, an implementation-specific prologue and a C string for the device name.

```
device_t *device_create(
  uint32_t prologue_len,
  uint8_t *prologue,
  const char *name,
  uint8_t *spriv);

peer_t *device_add_peer(
  device_t *dvp,
  const char *name,
  uint8_t *rs,
  uint8_t *psk);
```

Given a device, the user can create a new *session* with a chosen peer, in the role of either the initiator or the responder.

```
session_t *create_IKpsk2_initiator(device_t *d, uint32_t peer_id);
session_t *create_IKpsk2_responder(device_t *d);
```

We mention at the end of Section 4.2 that different Noise protocols handle identity management very differently; and that mishandlings can lead to serious vulnerabilities. We rule out these errors by construction in our API, using hybrid embeddings (Section 4.3.4) to meta-program the signature of the API functions. For instance, IKpsk2 demands a peer identity at initiator-creation time; this is reflected by the presence of the peer_id argument above. Conversely, for XX, both parties learn the remote's identity during the handshake, and the peer_id argument is absent from the C function signature.

This in turn begs the question of what should be an acceptable *policy* to deal with receiving a peer's public static key over the network, when the key is currently unknown to the device. The answer varies, and generally requires application-specific error handling. For instance, in the case of WireGuard, an unknown user simply cannot connect and the handshake is aborted. For WhatsApp, conversely, the application registers the peer with the device, and proceeds with the conversation.

In Noise*, we delegate these decisions to the user of our library via a *policy function* and a *certification function*. The former is a constant in practice, and simply determines

whether unknown keys may be accepted. The latter receives the decrypted payload of the message which should contain a certificate for the key, and from it determines whether to certify or invalidate a key. This behavior is triggered upon receiving an S token without a corresponding entry in the peer table.

Long-term key storage. To make sure our library is self-contained and ready to be used, Noise* incorporates a verified long-term (e.g., on-disk) key storage feature. Concretely, the device state can be serialized and deserialized, which includes peer list and static key. We use an AEAD construction, with the device and peer names as authenticated data. In order to avoid nonce reuse, each serialization generates a fresh nonce to be fed into the AEAD construction; the nonce is stored on disk, so that it can be reloaded at decryption-time. Our implementation comes with proofs of correctness for the parser and serializer, namely that they are the inverse of each other. Whether on-disk storage is enabled is up to the user; should they enable it via a meta-parameter, the resulting C code will contain, among other things, a create_device_from_secret that takes an encryption key, encrypted data, and returns a fresh device (or NULL if decryption failed). We delegate the handling of the on-disk encryption key to the user of our library.

A High-Level API with Message Encapsulation. To provide an industrial-grade, error-proof Noise library, there remains one last issue to address: right now, the user might inadvertently send messages at a lower level of confidentiality or authenticity than intended. This may happen either because the user has misunderstood the guarantees provided by a given Noise pattern; or because they sent early data in the handshake, before the full guarantees were established (Figure 4.2).

We revisit the Noise confidentiality levels (Figure 4.2) and expose an informative subset of them to the user: "public" (C0), "known remote replayable" (C2), "known remote weak forward" (C3) and "known remote strong forward" (C5). Then, we abstract away the type of messages and impose that the user go through a constructor and a destructor. These not only require the user to specify a level, but also to commit to a session and a peer, which rules out improper handling of data.

```
encap_message_t *pack_with_conf_level(
  uint8_t requested_conf_level,
  const char *session_name,
  const char *peer_name,
  uint32_t msg_len,
  uint8_t *msg);

bool unpack_message_with_auth_level(
```

```
  uint32_t *out_msg_len,
  uint8_t **out_msg,
  char *session_name,
  char *peer_name,
  uint8_t requested_auth_level,
  encap_message_t *emp);
```

Encapsulated messages can then be sent through an API that wraps handshake_write_m and takes care of packing and unpacking. When sending, we check that the session sn has reached *at least* the desired confidentiality level; when receiving, we check that the requested authentication level is *at most* the session's current level. The high-level rcode captures both state machine errors (stuck), and authenticity or confidentiality errors.

```
rcode session_write(
  encap_message_t *input,
  session_t *sn,
  uint32_t *out_len,
  uint8_t **out);

rcode session_read(
  encap_message_t **out,
  session_t *sn,
  uint32_t *inlen,
  uint8_t *input);
```

This concludes the tour of our Noise protocol implementation. From the protocol actions of Section 4.3, we derived a state machine implementation that properly handles failures and is entirely meta-programmed. We extend this state machine with runtime support for peer and device management, peer authentication policies, and on-disk long-term key storage. We expose the API via safe functions that perform confidentiality and authenticity run-time checks at the API boundary to rule out errors from unverified C clients. We obtain the first verified implementation for a full secure channel protocol stack, complete from cryptographic primitive to its user-facing API.

4.5 Symbolic Security Proofs for Noise*

As explained in Section 4.2, the Noise specification [260] describes the expected security guarantees for each Noise protocol in terms of authentication (A0-A2) and confidentiality levels (C0-C5). Several analyses have shown that various Noise protocols meet these guarantees against classic Dolev-Yao-style active network adversaries [272], using

symbolic analysis tools like ProVerif [262] and Tamarin [261]. Although these analyses provide comprehensive results for the protocol messaging code, they do not cover important details like message formats, protocol state machines, or key management, which are crucial to the security of full Noise implementations. In this section, we close this gap by proving the symbolic security of our F* Noise specification, relying on a framework called DY* [265].

4.5.1 Background on DY*

DY* Framework. DY* is a set of F* libraries that enables the symbolic security verification of protocol code written in F* [265]. In effect, we take our Noise protocol specification from Section 4.2 and replace all calls to concrete cryptography, random number generation, and state storage with the symbolic libraries provided by DY*, to obtain a *symbolic security specification* in F* that is functionally equivalent to our original specification. We then use the proof patterns provided by DY* to prove that our specification satisfies the security guarantees expected by Noise. Our proofs account for an unbounded number of protocol sessions and an active Dolev-Yao adversary [272].

DY* has previously been used to verify various protocols (including Signal [265]) but a key novelty of our approach is that we build a generic security proof for a Noise protocol interpreter to obtain security guarantees for *all* Noise protocols in one go. This kind of parameterized inductive proof is out of reach of tools like ProVerif and Tamarin, which instead have to rely on per-instance verification of each Noise protocol [261, 262]. The trade-off is that DY* is not as automated as these tools, and it does not yet support the verification of equivalence properties, needed to state goals like identity privacy.

We refer the reader to the DY* paper [265] and public code repository [273] for its detailed presentation. Below, we briefly discuss the main elements used in our Noise security proof.

Trace-Based Semantics. A DY* program consists of a set of stateful protocol functions (e.g. session_create, handshake_write) that can be executed by each protocol participant or *principal* (e.g. `"alice"`,`"bob"`) to initiate or continue any number of protocol sessions. Each session is locally identified by an integer sid; by convention, sid 0 is used for long-term keys.

The interleaved distributed execution of protocol sessions across multiple principals is modeled by an append-only global trace that records every message sent between principals, every freshly generated random value, every (long-term and ephemeral) session state stored by each principal, and every security event triggered by a principal to mark the progress of a protocol session. The index of an entry in the global trace

can be seen as a unique immutable timestamp, so we can state, for example, that an event was triggered at a particular trace index (event_at i (Send A B M)) and that this occurred *before* another event (event_at j (Recv B A M) $\wedge$ i $<$ j).

For example, in a run of the Noise IKpsk2 protocol between I and R, after I sends the first message, the global trace contains an entry for the generation of I's ephemeral key (x), the message from I to R, and I's handshake state after this message. Once R processes the first message and responds with the second message, the trace is extended by another entry for the responder ephemeral, the second message, and the handshake state stored at R. When the handshake is complete, both parties discard their session-specific handshake states and store new session states containing the final cipher states.

The attacker is modeled as an F* program that acts as a global scheduler: it drives the execution of all protocol sessions by calling protocol functions at different principals. It has all the capabilities of an active network attacker: it can read and write messages between any two principals in the global trace, it can generate its own fresh random values, and it can call cryptographic functions using values it has learned. The attacker can also dynamically compromise any session state stored at any principal to obtain its contents, and this action is marked with a new entry in the trace (compromised_at i "alice"sid). Hence, by compromising the long-term key session (sid=0) at a principal, the attacker can learn the principal's static Diffie-Hellman and pre-shared keys. Alternatively, by compromising a session corresponding to an ongoing Noise handshake, the attacker can learn the current handshake state, including any private ephemeral keys. However, the attacker cannot guess random values, or invert encryption unless it either has the key or has explicitly compromised it. The attacker's knowledge at a particular timestamp in the global trace is formalized by an inductive predicate: attacker_knows_at i m.

4.5.2 Formalizing Payload Security Goals as Trace Properties

We formalize each of the 3 authentication levels (A0-A2) and 6 confidentiality levels (C0-C5) of Noise as *trace properties*, i.e., predicates over the global trace.

Authentication Goals. Suppose that before sending an authenticated payload, each Noise participant A triggers an event of the form AuthSent A B M L indicating that it is sending a message M to B at authentication level L. After successfully processing an authenticated payload M in a session sid, the recipient B triggers an event AuthReceived B sid A M L. Then, the authentication goal for messages sent at Noise authentication level **1** can be written as a trace property:

```
1 let trace_property_A1 =
2   forall i sid A B M. event_at i (AuthReceived B sid A M 1) ==>
3     (exists j. j < i /\ event_at j (AuthSent A B M 1)) \/
4     (exists k. k < i /\ (compromised_at k A 0 \/
5                          compromised_at k B sid \/
6                          compromised_at k B 0))
```

This trace property says that whenever B accepts a message M from A at time i (with authentication level A1), either this must be an authentic message sent by A at time $j < i$, or else the static key of A or the ephemeral session state at B or the static key of B must have been compromised before i. The disjunct on line 5 indicates the possibility of a KCI attack: i.e., the loss of message authenticity when the recipient B's static key is compromised. To obtain the trace property for authentication level A2, we simply remove this disjunct (line 5) to require the absence of KCI attacks:

```
1 let trace_property_A2 =
2   forall i sid A B M. event_at i (AuthReceived B sid A M 2) ==>
3     (exists j. j < i /\ event_at j (AuthSent A B M 2)) \/
4     (exists k. k < i /\ (compromised_at k A 0 \/
5                          compromised_at k B sid)
```

Level A0 provides no guarantees:

```
1 let trace_property_A0 = True
```

Confidentiality Goals. Confidentiality guarantees are stated as predicates over the global trace that describe the conditions in which a protocol secret may become part of the attacker's knowledge. Suppose that each Noise participant A triggers an event ConfSent A sid B sid' M L before sending a fresh random secret message M at confidentiality level L to B, where sid and sid' are the session indexes at A and B. Then, the confidentiality level C4 is written as the following trace property:

```
1 let trace_property_C4 =
2   forall i j sid sid' A B M.
3     (event_at i ConfSent A sid B sid' M 4 /\
4      attacker_knows_at j M /\ i <= j) ==>
5     (exists k. k < i /\ (compromised_at k B 0 \/
6                          compromised_at k A 0)) \/
7     (exists l. l <= j /\ (compromised_at l A sid \/
8                           compromised_at l B sid')
```

This predicate says that if a secret message M sent at time i (and confidentiality level C4) from a session sid at A to a session sid' at B, and M subsequently becomes

known to the adversary at time j, then either the static key of A or the static key of B was compromised before the message was sent at i or else one of the two ephemeral protocol session states (sid, sid') was compromised before j.

The strongest variant of forward secrecy provided by Noise (C5) limits static key compromise to the recipient; that is, we drop the disjunct on line 5 (compromised_at k A 0) allowing the sender A's static key to be compromised at any time without affecting the confidentiality of M:

```
1 let trace_property_C5 =
2 forall i j sid sid' A B M.
3     (event_at i ConfSent A sid B sid' M 5 /\
4      attacker_knows_at j M /\ i <= j) ==>
5     (exists k. k < i /\ compromised_at k B 0) \/
6     (exists l. l <= j /\ (compromised_at l A sid \/
7                           compromised_at l B sid')
```

The trace properties for levels C1-C3 provide weaker forward secrecy guarantees than C4 by restricting the compromise scenarios in which confidentiality is guaranteed. In these scenarios, the sender does not know if the peer ephemeral public key it is using actually belongs to some recipient session sid' of B; instead this public key may have been provided by the attacker. So we use a different event ConfSentEph A sid B eph M L, where instead of the peer session sid', A marks the (possibly attacker-controlled) peer ephemeral key which it used to derive the encryption key.

The confidentiality guarantee of C1 then states that the message is secret only if this peer ephemeral key is confidential:

```
1 let trace_property_C1 =
2   forall i j sid eph A B M.
3     (event_at i ConfSentEph A sid B eph M 1 /\
4      attacker_knows_at j M /\ i <= j) ==>
5     (exists l. k <= j /\ (compromised_at k A sid \/
6                           (exists sk. eph = PK(sk) /\
7                            attacker_knows_at k sk)))
```

That is, we have no confidentiality if the attacker actively interferes with the session to provide its own public key eph. Note that this means that confidentiality is lost even if none of the recipient B's keys have been compromised.

C2 provides a stronger guarantee that links the confidentiality of the message to static key compromise at the recipient B:

```
1 let trace_property_C2 =
2 forall i j sid eph A B M.
```

```
3    (event_at i ConfSentEph A sid B eph M 2 /\
4     attacker_knows_at j M /\ i <= j) ==>
5    (exists k. k <= j /\ (compromised_at k A sid \/
6                          compromised_at k B 0))
```

That is, we have no confidentiality if the attacker compromises the recipient's static key or the sender's ephemeral key (before or after the message is sent), but the compromise of the sender's static key does not affect security.

C3 provides weak forward secrecy, which combines the guarantees of C1 and C2 to link message confidentiality to both the peer's ephemeral key and recipient's static key:

```
1  let trace_property_C3 =
2  forall i j sid eph A B M.
3    (event_at i ConfSentEph A sid B eph M 3 /\
4     attacker_knows_at j M /\ i <= j) ==>
5    (exists k. k <= j /\ (compromised_at k A sid \/
6                          (exists sk. eph = PK(sk) /\
7                           attacker_knows_at k sk /\
8                           compromised_at k B 0)))
```

That is, we have no confidentiality if the attacker first actively interferes with the session to provide its own ephemeral public key and then also compromises the recipient's static key (before or after the protocol message is sent).

C0 provides no guarantees:

```
let trace_property_C0 = True
```

Deriving Security Goals for each Noise Protocol. The overall security goal for our Noise specification is to prove that every global execution trace for every Noise protocol satisfies the 7 trace properties corresponding to A1-A2 and C1-C5. Hence, for each payload in a Noise protocol, we can look up the confidentiality and authentication level (from Appendix A) and map it to the corresponding trace property to obtain the precise security guarantee at sender and recipient.

Our way of encoding security goals as trace properties (sometimes called correspondence assertions [274]) is similar to how these goals are usually stated in protocol verification tools like ProVerif and Tamarin. Notably, these trace properties are defined independently of a specific Noise protocol or its F* code and only refer to events triggered during protocol execution. This allows our security goals to be independently audited and compared with other formulations. Indeed, the corresponding ProVerif query for authentication level A2 in prior work [262] (see Section 4.2.3) is almost identical (modulo syntax) to our trace property. However, the ProVerif queries for forward secrecy (C2-C5)

look different from our trace properties since they use phases (instead of timestamps) to enforce an order between messages and compromise events.

4.5.3 Security Proof for Noise⋆: Overview

Having stated our (trusted) security goals by mapping *levels* to trace properties, the next step is to prove that our security-oriented specification preserves a global trace invariant that implies these trace properties. This symbolic security proof in DY⋆ relies on two kinds of (untrusted) annotations: secrecy labels and authentication predicates. These must be provided by the programmer and are then verified by typechecking.

Secrecy Labels. Each bytestring (key, message, constant) used in the protocol must be annotated with a *secrecy label* that indicates which *sessions* of which *principals* are allowed to read them. For example, a static (long-term) Diffie-Hellman private key belonging to a principal named `"alice"` is given a label CanRead [P `"alice"`], indicating that it can be read by all sessions of `"alice"`, whereas an ephemeral private key that is only meant to be used in session sid is labeled CanRead [S `"alice"`sid]. A long-term pre-shared key between the principals `"alice"` and `"bob"` is given the label CanRead [P `"alice"`] ⊔ CanRead [P `"bob"`], where the join (⊔) operator indicates the union of the two labels. For succinctness, we can also write the above label as CanRead [P `"alice"`; P `"bob"`]. Constants and public bytestrings are labeled with Public, indicating that they can be read by any session, including by the attacker.

Secrecy labels are related by a reflexive, transitive relation can_flow i l1 l2 which says that a label l2 is stronger (more restrictive) than label l1 at a timestamp i. For example, the label Public can always flow to any other label, and CanRead [P p; P p'] can always flow to CanRead [P p]; but CanRead [S p sid] can only flow to Public at timestamp i if the event Compromise p sid occurs before i in the global trace.

The DY⋆ cryptographic API manipulates these labels and imposes a strict discipline on their usage, ensuring that secret data never flows to a public location. In particular, AEAD encryption returns a ciphertext labeled Public, but requires as a pre-condition that the label of the payload must flow to the label of the key. Computing a Diffie-Hellman shared secret between two private keys with labels l and l' yields a key with label $l \sqcup l'$, indicating that any session that can read one of the two private keys can know the shared secret. Calling a key derivation function (KDF) with two keys with labels l and l' yields a key with label $l \sqcap l'$, where the meet (⊓) operator indicates an intersection; only sessions that can read both inputs may read the result. Hence, KDF strengthens the label of a key by mixing in additional key material.

As a consequence of the secret labeling discipline, DY⋆ provides a generic secrecy

lemma stating that a secret with label l can only be obtained by the adversary at timestamp i if can_flow i l Public holds. This lemma can be instantiated to obtain strong protocol-specific security guarantees. For example, a Diffie-Hellman shared secret x with label CanRead[S "alice"sid] ⊔ CanRead[S "bob" sid'] is *forward secret*: an attacker can only obtain it if it specifically compromises sid (at alice) or sid' (at bob) before these sessions end and their state is deleted. Notably, compromising the long-term keys of alice or bob, or any other sessions at these principals does not help the adversary obtain x. Labels like these allow us to prove trace properties like strong forward secrecy (C5).

Authentication Predicates. DY* also defines a set of authentication predicates that can be instantiated for each protocol to enable the propagation of security invariants through cryptographic calls and events. For example, AEAD encryption has a pre-condition ae_pred that is intended to specify the conditions under which a message is allowed to be encrypted; this predicate becomes a post-condition for AEAD decryption. For Noise, we instantiate ae_pred to require that the sender must have triggered the AuthSent and ConfSent events, and consequently obtain the corresponding authentication guarantee at the recipient. Similarly, an event predicate event_pred states when an event may be triggered; we instantiate it to encode our authentication goals, requiring that the event AuthReceived can only be triggered if the corresponding authentication property holds. By instantiating these predicates and verifying that our protocol code still satisfies the resulting preconditions, we link protocol session state invariants with cryptographic guarantees to prove the target trace invariants for our Noise specification.

Structure of the Proof. We structure the symbolic security proof for our Noise specification in several steps:

- **Security Levels to Trace Invariants**: we write a generic function that maps any step of any Noise pattern to its corresponding *level*, as described in Figure 4.2, the full version of which is in Appendix A. We then extend the global trace invariant with the corresponding authentication or confidentiality trace properties for every Noise message sent and received at each level.

- **Security Levels to Key Secrecy Labels**: we map each payload security level to a predicate over the *secrecy label* of the AEAD key used to encrypt the payload. We show that, for each confidentiality and authentication level, the AEAD key secrecy label, the properties of AEAD encryption, and the generic secrecy lemma of DY* together imply the global trace invariant.

- **Handshake State Invariant**: to each state of the handshake, we associate

a label and we prove that in all runs of the protocol code, the resulting state matches its target label. We then prove that the label of the handshake state at a given protocol stage is always stronger than the target key secrecy label for that stage of the protocol.

- **High-Level API security**: our high-level API always preserves the handshake state invariant. In combination with the above sequence of proof steps, this allows us to prove that all reachable traces of our Noise protocol specification satisfy the level-based authentication and confidentiality guarantees of Noise. In particular, we prove that these security guarantees are correctly propagated all the way up to the user-facing API where they are exposed as understandable security guarantees.

To achieve the proof above, we build a new security-oriented specification of Noise that is provably equivalent to our original specification, but is annotated with labels and logical invariants that enable us to prove our security goals. The full proof development is in F*; we now describe each of the proof steps.

4.5.4 Security Proof: Handshake State Invariant

Labeling the Handshake State. In our security spec, we annotate every element of the handshake state with a secrecy label. The cipher_state and symmetric_state types are now parameterized by a timestamp i and a label l for the chaining key ck and AEAD key k:

```
type cipher_state (i:nat) (l:label) = {
  k: option (aead_key i l);
  n: nat; }

type symmetric_state (cfg:config) (i:nat) (l:label) = {
  h : hash cfg i Public;
  ck : chaining_key cfg i l;
  c_state : cipher_state i l; }
```

The full handshake state for a session sid at a protocol participant p is annotated with a security index. For each participating principal in the protocol, the index includes the name of the principal (p), its local session identifier (sid), the name of the peer (peer), and the secrecy label associated with the ephemeral key of the peer (peer_eph_label). Of these, the last two are optional, since they may only be available in later stages of protocols.

```
type index = {
  p: principal;
  sid: nat;
  peer: option principal;
  peer_eph_label: option label; }
```

Notably, the index does not contain the peer's local session identifier, since this value is unknown to p. All p knows is the remote ephemeral public key, and so we state our security properties in terms of what p knows about the security of this key, which is encapsulated in peer_eph_label.

Each handshake state is annotated with the current index idx and the current label l encoding the secrecy of the current chaining key and cipher state. Hence, in each run of a protocol at a principal p, we have an index and a label describing the current security guarantees.

```
type handshake_state (cfg:config) (i:nat) (l:label) (idx:index)
= {
sym_state : symmetric_state nc i l;
static : option (keypair cfg i (CanRead [P idx.p]));
ephemeral : option (keypair cfg i (CanRead [S idx.p idx.sid]));
remote_static : option (public_key cfg i (CanRead [P idx.peer]));
remote_ephemeral : option (public_key cfg i idx.peer_eph_label);
preshared : option (preshared_key cfg i idx.p idx.peer); }
```

In the handshake state, the local static and ephemeral keypairs have secrecy labels related to the current principal and session. Once we have validated the remote static key (see the certification function below), it is labeled with CanRead [P idx.peer]. However, the relationship between the remote ephemeral key label (idx.peer_eph_label) and the peer's identity is unknown. The pre-shared key, if it exists, has a label indicating that it is shared between the principal and its peer.

Computing Target Secrecy Labels. Given a Noise protocol (described as a handshake_pattern), and an index describing the current run, we can compute the target secrecy label for the handshake state at the initiator and responder at each stage of the protocol. Note that since the initiator and responder have different (partial) views of their peer's protocol state, the computed labels at the two ends may be different. In total, we compute four labels at each stage, two for‘ the initiator, and two for the responder:

- l_i: the current label at I;
- $l_i^{\leftarrow}$: the last label at which I received a message from R;

Protocol	Message Sequence	Stage	Initiator Handshake State Label l_i	$l_i^{\leftarrow}$	Responder Handshake State Label l_r	$l_r^{\rightarrow}$
X	$\leftarrow$ s	pre				
	$\rightarrow$ e, es, s, ss $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) $\sqcap$ (CanRead [P idx_i.p; P idx_i.peer])	-	(CanRead [P idx_r.p]$\sqcup$$idx_r$.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$
	$\rightarrow$ $[d_1, d_2, ...]$	2	$= l_i[1]$	Public	$= l_r[1]$	$= l_r[1]$
NX	$\rightarrow$ e $[d_0]$	1	Public	-	Public	Public
	$\leftarrow$ e, ee, s, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p]$\sqcup$$idx_r$.peer_eph_label)	Public
	$\rightarrow$ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$
	$\leftrightarrow$ $[d_3, ...]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$
IKpsk2	$\leftarrow$ s	pre				
	$\rightarrow$ e, es, s, ss $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) $\sqcap$ (CanRead [P idx_i.p; P idx_i.peer])	-	(CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$
	$\leftarrow$ e, ee, se, psk $[d_1]$	2	$l_i[1]$ $\sqcap$ (CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [P idx_i.p]$\sqcup$$idx_i$.peer_eph_label) $\sqcap$ (CanRead [P idx_i.p; P idx_i.peer])	$= l_i[2]$	$l_r[1]$ $\sqcap$ (CanRead [S idx_r.p idx_r.sid]$\sqcup$$idx_r$.peer_eph_label) $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) $\sqcap$ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$
	$\rightarrow$ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$
	$\leftrightarrow$ $[d_3, d_4, ...]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$

Figure 4.3: Target Security Labels Computed for Three Example Noise protocols (X, NX, and IKpsk2)

- l_r: the current label at R;
- $l_r^{\rightarrow}$: the last label at which R received a message from I.

When we wish to refer to the label at a particular stage n, we write $l_i[n]$ or $l_r[n]$. The sequence of computed labels for our three example Noise protocols X, NX, and IKpsk2 are shown in Figure 4.3, the full version of which is in Appendix B.

The target label computation faithfully follows the sequence of cryptographic operations. Every time new key material is added to the handshake state, the new label is a meet (or $\sqcap$) of the old label and the new key material. If the key material is a Diffie-Hellman secret, its label is a join (or $\sqcup$) of the labels of the two Diffie-Hellman private keys. Each participant knows the labels of its own static key (CanRead [P idx.p]) and its own ephemeral key (CanRead [S idx.p idx.sid]). After public key validation, it also knows the label of the peer's static key (CanRead [P idx.peer]), but it typically does not know the label of the peer's ephemeral key (idx.peer_eph_label). Hence, Diffie-Hellman operations involving the peer's ephemeral key result in labels that use idx.peer_eph_label as an opaque label.

Computing Target Labels for X. The protocol X has a single message with four tokens. At the initiator point of view, the token e does not affect the label; es changes the label to the secrecy label of the ephemeral-static Diffie-Hellman shared secret (CanRead [S idx_i.p idx_i.sid; P idx_i.peer]); s does not affect the label; ss changes the label to the meet of the previous label and the label of the static-static Diffie-Hellman shared secret (CanRead [P idx_i.p; P idx_i.peer]). Hence the label l_i after the first message is a meet of the labels of the two Diffie-Hellman shared secrets.

From the responder's point of view, the label l_r looks a bit different. Since the responder does not know the label of the initiator's ephemeral key, the label it computes for the

ephemeral-static shared secret is of the form (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label), where idx_r.peer_eph_label is the label of the peer's ephemeral private key. The label for the static-static shared secret is the same. Hence, for the responder, the key after the first message is only partially authenticated (level 1 in Noise terminology)

The last received label $l_i^{\leftarrow}$ is null since the initiator has not received any message, and $l_r^{\rightarrow}$ is the same as l_r.

Computing Target Labels for NX. For NX, the label computation is similar, except that the labels at the initiator and responder are even more asymmetric, since the initiator is unauthenticated. Hence, at the end of the protocol, the initiator has a precise label linking its session to the peer's identity (CanRead [S idx_i.p idx_i.sid; P idx_i.peer]), but the responder only has a weak label linking its session to *some* (potentially compromised) peer ephemeral key (CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label).

Computing Target Labels for IKpsk2. The computation of labels for IKpsk2 follows the same pattern as X and NX except that both parties are authenticated and their labels get stronger with each stage. Notably, at the end of the second message, the responder's label $l_r[2]$ has reached the maximum label for this pattern (it never changes thereafter). However, at this point, the last received label $l_r^{\rightarrow}[2]$ is still quite weak (since R has not yet received a message protected under the newest key). It is only when the responder receives a subsequent (data) message from the initiator that the two labels l_r an $l_r^{\rightarrow}$ coincide. It is this quirk of IKpsk2 that leads to the responder obtaining a slightly weaker forward secrecy guarantee (Noise level 4) at the end of the second message, and strong forward secrecy (level 5) after the third message.

Hence, for instance, after the second IKpsk2 message, the target handshake state label at an initiator with index idx_i is computed as follows:

(CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) $\sqcap$
(CanRead [P idx_i.p; P idx_i.peer]) $\sqcap$
(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label]) $\sqcap$
(CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label]) $\sqcap$
(CanRead [P idx_i.p; P idx_i.peer])

Each line of the label corresponds to some key material that has been mixed into the chaining key: ephemeral-static, static-static, ephemeral-ephemeral, and static-ephemeral Diffie-Hellman secrets, followed by a pre-shared key.

Proving the Handshake Secrecy Invariant. Our main secrecy invariant for the handshake state is that at each stage of the protocol its label must match the computed target label. We prove that the messaging functions in our Noise specification preserve

this invariant whenever they modify the handshake state. For example, the type of our labeled send_message_tokens function is as follows:

```
val send_message_tokens (cfg : config) (initiator is_psk : bool)
  (tokens : list token) (i : nat) (l : label) (idx : index)
  (st : handshake_state cfg i l idx) :
  (result
    (ciphertext : msg i Public *
     handshake_state cfg i (update_label l idx tokens initiator) idx))
```

The result type says that the new handshake state label (after the message is sent) can be computed from the old label, the index, the list of sent tokens, and the message direction. Separately, we show that this updated label corresponds exactly to the target label computed for this stage of the handshake pattern.

The type for receive_message_tokens is a bit more complicated since the index of the handshake state may change in the course of the function, if the message contains the peer's static or ephemeral key. Other than this detail, we again prove that it updates the handshake label in the same way from the prior label and received tokens. Hence, we prove that all our messaging functions preserve the handshake labeling invariant.

Establishing the Peer Ephemeral Invariant. The label of peer ephemeral key (idx.peer_eph_label) in the handshake state is (as yet) unrelated to the peer's identity. It means that the keys in the handshake state are linked to an untrusted remote ephemeral key, and hence are not forward secret. To obtain stronger forward secrecy guarantees, we need to establish an authentication invariant on the handshake state.

As described above, in addition to the target secrecy labels (l_i, l_r) for each handshake state at the initiator and responder, we also keep track of the label at which each participant received its last message ($l_i^{\leftarrow}$, $l_r^{\rightarrow}$). We then prove that if this last receive label is non-compromised at i (i.e., it does not flow to Public) then the remote ephemeral key label at i (idx.peer_eph_label) must be of the form CanRead [S idx.peer sid'] for some session sid' at the peer. In other words, the last received message conditionally attests to the authenticity of the peer ephemeral key. If the payload received with this message was protected with a strong label, we get a strong authentication guarantee for the peer ephemeral.

To obtain an authentication guarantee for the peer ephemeral key, we rely on the global AEAD predicate (ae_pred, mentioned in Section 4.5) to enforce that every encrypted handshake payload sent in each direction contains a transcript hash in the associated data, which uniquely captures all the ephemeral keys exchanged so far. Using this AEAD predicate at each decryption, the receive_message functions can establish and maintain the peer ephemeral invariant in the recipient's handshake state.

4.5.5 Security Proof: Handshake State Invariant to Trace Properties

Our next goal is to show that the handshake state invariant implies the trace properties corresponding to our authentication (A0-A2) and confidentiality goals (C0-C5). This proof is in three steps: (1) we map each authentication and confidentiality level to predicates on the secrecy label of an AEAD key; (2) we show that the handshake state invariant guarantees that the current AEAD key in the handshake state satisfies these key secrecy predicates; (3) we show that each key secrecy predicate implies the trace property for the corresponding level.

Mapping Levels to Key Secrecy Predicates. We define a series of security predicates in F*, one for each payload security level, stated in terms of the current global timestamp (i), security index (idx), and a handshake state label (l). The confidentiality predicates should be read from the viewpoint of the sender, whereas the authenticity predicates are from the viewpoint of the recipient. Each predicate has the same shape, represented by the predicate type below:

```
type security_pred = i:nat -> idx:index -> key_label:label -> Type
```

The three authentication predicates are as follows:

Level	Authentication Predicate (over i, idx, and l)
A0	$\top$
A1	can_flow i (CanRead [P idx.p; P idx.peer]) l
A2	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l

Each authentication predicate is stated in terms of the strength of the current key label l; that is, the conditions under which the current key may be known to the adversary. This in turn implies the conditions under which the messages received by idx.p may have been forged or tampered with.

For level A0, there are no authentication guarantees, and so the predicate is always $\top$.

For level A1, we require that l is at least as strong as the (static-static) label CanRead [P idx.p; P idx.peer], which means that the current key can only be known to the adversary if one of the two static keys (at idx.p or idx.peer) were currently known to the adversary.

For level A2, we strengthen the requirement by requiring that l is at least as strong as the (ephemeral-static) label CanRead [S idx.p idx.sid; P idx.peer]. This predicate requires that the AEAD key in the cipher state should be known only to the principal (idx.p)

Level	Confidentiality Predicate (over i, idx, and l)
C0	$\top$
C1	can_flow i (CanRead [S idx.p idx.sid] $\sqcup$ idx.peer_eph_label) l
C2	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l
C3	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l $\wedge$ can_flow i (CanRead [S idx.p idx.sid] $\sqcup$ idx.peer_eph_label) l
C4	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l $\wedge$ can_flow i (CanRead [S idx.p idx.sid] $\sqcup$ idx.peer_eph_label) l $\wedge$ (compromised_before i (P idx.p) $\vee$ compromised_before i (P idx.peer) $\vee$ ($\exists$sid'. peer_eph_label == CanRead [S idx.peer sid']))
C5	can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l $\wedge$ can_flow i (CanRead [S idx.p idx.sid] $\sqcup$ idx.peer_eph_label) l $\wedge$ (compromised_before i (S idx.p idx.sid) $\vee$ compromised_before i (P idx.peer) $\vee$ ($\exists$sid'. peer_eph_label == CanRead [S idx.peer sid']))

Figure 4.4: Confidentiality Predicates for each Noise Confidentiality Level

and its peer (idx.peer), and should be bound to the current session sid at idx.p. So, even an adversary who compromises a principal's static key cannot obtain the session key; the adversary must compromise either the principal's ephemeral key or the peer's static key. In particular, this forbids KCI attacks, since compromising the long-term keys of the principal idx.p does not break authentication.

The six confidentiality predicates are depicted in Figure 4.4, again stated in terms of the timestamp i, index idx, and the current handshake state label l.

As with authenticity, level C0 provides no guarantees.

For level C1, we require that the handshake state label l is at least as strong as the ephemeral-ephemeral label (CanRead [S idx.p idx.sid] $\sqcup$ idx.peer_eph_label) which means that the recipient (peer) is unauthenticated and hence could be played by an active attacker. This level only protects against passive adversaries. Note that idx.peer_eph_label is actually an optional value, so our the predicate definition implicitly says that this value must not be empty, otherwise the confidentiality predicate is false.

For level C2, we require that l is at least as strong as the ephemeral-static label CanRead [S idx.p idx.sid; P idx.peer], so we have confidentiality unless the sender's ephemeral key or the peer's static key are compromised.

For level C3, our requirement is a little stronger in that we require the current label to be stronger than both the ephemeral-static label (from 2) and the ephemeral-ephemeral label (from 1). This level provides weak forward secrecy, since the attacker can actively interfere with the session to insert its own ephemeral key.

For level C4, we strengthen the forward secrecy guarantee of level 3 by adding

conditions under which the peer ephemeral key is known to be secure. Unless the attacker has actively compromised one of the two static keys (before the session is complete), the peer_eph_label must be of the form CanRead [S idx.peer sid'] for some peer session sid'. Hence, we have forward secrecy if both static keys are non-compromised during the session.

Level C5 provides strong forward secrecy: the attacker must compromise either the sender's ephemeral key or the recipient's static key before the session is complete. This predicate is as follows:

```
can_flow i (CanRead [S idx.p idx.sid; P idx.peer]) l ∧
can_flow i (CanRead [S idx.p idx.sid] ⊔ idx.peer_eph_label) l ∧
(compromised_before i (S idx.p idx.sid) ∨
 compromised_before i (P idx.peer) ∨
 (∃ sid'. peer_eph_label == CanRead [S idx.peer sid']))
```

The first line of this predicate says that the handshake secrets should be readable only by the (authenticated) peer (idx.peer) and the current session idx.sid at idx.p. The second line says that the handshake secrets must also be bound to some peer ephemeral key. The last two lines provide strong forward secrecy: they say that unless the peer's long-term keys and the specific session idx.sid of idx.p was compromised (before the session is complete), the peer ephemeral key must have a label of the form CanRead [S idx.peer sid']. Since the key label is bound to specific sessions at both ends, compromising long-term keys after the session has no effect on key secrecy.

Handshake Invariant to Key Secrecy Predicates. Given the handshake state invariant (including the secrecy invariant and the peer ephemeral invariant), we prove that in each reachable handshake state the current handshake label satisfies the authenticity and confidentiality predicates described above for the security levels at the current stage of the protocol. In other words, we show that the secrecy label annotating the handshake state (and hence the label of its current AEAD key) is always stronger than the label expected by the Noise payload security level at the current stage of the protocol.

Key Secrecy Predicates to Trace Invariants. Message authenticity and confidentiality guarantees in secure channel protocols directly rely on the secrecy of the corresponding encryption key. In DY*, the AEAD encryption function only allows the encryption of a message under a key whose label is stronger than the message, and so the key label expresses an upper bound on the secrecy of messages. For our confidentiality proofs, we assume that the application always sends messages labeled with the current handshake state label, which is the same as the current AEAD key

label. Then, from using the confidentiality predicate on key labels for each level, we derive the confidentiality trace invariant for messages sent at this level as a corollary of the generic secrecy lemma of DY*.

For authentication, we instantiate the ae_pred pre-condition of AEAD encryption to ensure that each call to the AEAD encryption function is preceded by a security event AuthSent with the appropriate parameters. We also instantiate the event_pred pre-condition for security events to ensure that the AuthReceived function can only be called if the corresponding authentication trace property is satisfied. These predicates, along with the properties of AEAD encryption and decryption, and the authentication predicates on the key secrecy label allow us to prove the trace invariant for each authentication level.

This completes the security proof for the protocol code. In summary, we combine the handshake state invariant with key secrecy predicates to show that every reachable handshake state preserves the global trace invariant, which includes the confidentiality and authentication goals for each level.

4.5.6 Security Proof: High-Level API security

The final step of our proof involves propagating the protocol security guarantees up through the stack all the way to the high-level API. For most of our code, this propagation is relatively straightforward: we prove that our code does not accidentally break the labeling discipline, by storing a secret value in a public location, or mixing up data from different sessions.

The main security-critical step in this proof is the static key validation function provided by the device API. We assume that the certification function can take a potential public key, along with a (possibly-empty) certificate, and verify that it is indeed a static public key belonging to a given principal:

```
val certification_function: i:nat -> rs:bytes -> rcert:bytes ->
 option (peer:principal{is_public_key rs i (CanRead [P peer])})
```

We also propagate our secrecy labels through the device management API, by annotating all remote static and pre-shared keys stored in the device with the appropriate labels and ensuring that these labels are respected by the data structure and by the encrypted storage mechanism.

After all these steps, we obtain a high-level API that guarantees that each application message sent or received with the API meets high-level security properties expressed using a subset of the Noise security levels.

Component	F⋆ spec	Low⋆ code	DY⋆ proof
Core Protocol (Section 4.3)	1,095	15,506	1,792
Device Management (Section 4.4)	315	6,410	475
Session API (Section 4.4)	1,106	13,184	3,681

Figure 4.5: Size of the Noise⋆ codebase, *excluding* whitespace and comments. The total size of the codebase is 43kLOC.

Pattern	Noise⋆	Custom	Cacophony	NoiseExpl.	Noise-C
X	6677	N/A	2272	4955	5603
NX	5385	N/A	2392	4046	5065
XX	3917	N/A	1593	3149	3577
IK	3143	N/A	1357	2459	2822
IKpsk2	3138	3756	1194	2431	N/A

Figure 4.6: Performance Comparison, in handshakes / second. Benchmark performed on a Dell XPS13 laptop (Intel Core i7-10510U) with Ubuntu 18.04.

4.6 Evaluation and Comparison with Related Work

Size of the Codebase. Figure 4.5 measures the size of the F⋆ codebase for our Noise protocol implementation. This covers everything described in this paper. The core protocol code contains the Noise messaging functions. Device management includes long-term key storage and validation, including the encrypted storage and verified in-memory data structures, such as a linked list and an imperative map. Session API includes the two successive state machines and the high-level user-facing API code. For each component, we list the size of the high-level specification, the Low⋆ code, and the DY⋆ proof. All of the code listed here was written for the purposes of this paper. The total size is 43kLOC excluding whitespace and comments. As a point of comparison, HACL⋆ itself is 97kLOC, making Noise⋆ the second largest F⋆ project in the literature.

The Compiled C Library. Using the Noise⋆ compiler, we compile several specialized C implementations for each of the 59 Noise protocols. Representative code sizes are: 6,400 lines of C code for IKpsk2, 5,900 LoC for XX, and 4,900 LoC for X. Each Noise Protocol admits several implementations, depending on the choice of primitives (e.g. SHA2-256 vs. Blake2b), and the degree of optimization (e.g. Blake2b-portable vs. Blake2-AVX2). As a proof of concept, we ran a batch job that produced 472 implementations, out of several thousand possible choices [271]; the result is a net 3.2M lines of C code (including whitespace). In practice, a typical user would choose a Noise protocol, a set of primitives and a choice of optimization level, then would download the corresponding C implementation from Noise⋆, along with a custom distribution of HACL⋆ containing the relevant cryptographic primitives for the target platform, to obtain a small high-performance protocol implementation. Advanced users can extend our code-base and

compile it in different ways, to obtain a combination of Noise patterns, for example.

Proof Overhead. A popular way of measuring the human effort of verification is the proof-to-code ratio: how many lines of Low⋆ code did we write for each line of C that we produced. If we were to consider all 59 Noise patterns, this ratio would drop to 0.2, without even taking into account all the ciphersuite specializations we support. Conversely, if we only ever wanted generated code for a single Noise protocol, then the ratio jumps to nearly 7. A more realistic estimate is a proof-to-code ratio of 1, based on the 44kLOC of C code produced for the five patterns we actively test and benchmark. This is on par with (or even better than) mature F⋆ verification projects like HACL⋆.

Feature Comparison. We compare other Noise implementations in Figure 4.7. Noise⋆ generates specialized code, and is a compiler (C). WireGuard and Brontide are specialized, built-in (B) implementations for the purposes of a single application. Other implementations are interpreted (I). We count all patterns, even those that do not appear in the Noise Protocol Framework.

An implementation offers a *Lean API* if it establishes a clear abstraction boundary that strives to prevent user mistakes. Details vary; here, the presence of a state machine with abstract send and receive functions is enough to qualify as a "Lean API". WireGuard and Brontide are omitted, since they use a single Noise protocol and therefore leverage that fact to traverse abstraction boundaries. An implementation successfully handles *Early Data* if it allows the user to use early message payloads, while preventing confidentiality issues one way or another. WireGuard uses a custom scheme that has been carefully audited; Brontide prevents sending payloads altogether before the handshake is finished. An implementation with *Key Validation* provides a way of validating keys upon receiving them, e.g. by calling a user-provided function. Finally, an API with *Key Storage* provides a long-term, secure way of storing and retrieving preshared or remote static keys.

Code sizes vary according to the feature set and the language used. For Noise⋆, we list the average size of a *single*, specialized C implementation. Noise⋆ is larger than e.g. Cacophony or Noise Explorer, because of a more verbose language (C) and a larger feature set. Noise⋆ is smaller than Noise-C or Noise Java. Our choice of generating C code will, we hope, facilitate integration in existing codebases. We remark that not all implementations support the same cipher suites; this depends on the choice of the underlying cryptographic library. We are here limited by e.g. the absence of Curve448 in HACL⋆; fortunately, none of the applications we studied require it (this includes WhatsApp, not counted in this table).

Performance comparison. We compare the speed of our code with other Noise

Implementation	Type	Language	Patterns	Lean API	Early Data	Key Valid.	Key St.	Verif.	Code Size
Noise*	C	C	59	Y	Y	Y	Y	Y	~5000
Cacophony	I	Haskell	59	Y	N	N	N	N	~1800
Noise-C	I	C	40	N	N	N	Y	N	~9000
WireGuard	B	C	1	-	-	Y	N	N	~2700
Snow	I	Rust	59	N	N	N	N	N	~3400
Noise Explorer	I	Rust/Go	50	Y	N	N	N	N	~900
Brontide	B	Go	1	-	-	N	N	N	~750
Noise Java	I	Java	40	N	N	N	N	N	~8000

Figure 4.7: Noise Implementations Comparison. For Type: C = compiler, I = interpreter, B = builtin, i.e., a custom implementation.

Project	Model	Tool	# Patterns	Code Gen	Model Size	Verif. Time
Noise*	S	F*	59	Y	~680	10s
Vacarme [261]	S	Tamarin	53	N	~270	1.4 days
Noise Explorer [262]	S	ProVerif	57	Y	~650	0.5-24h
fACCE [263]	C	Manual	8	N	-	-
Dowling et al. [275]	C	Manual	1	N	-	-
WireGuard-CV [253]	C	CryptoVerif	1	N	~5000	1.5h

Figure 4.8: Noise Analysis Comparison. For Model: S = symbolic, C = computational. Model Size and Verification Time are per pattern.

implementations in Figure 4.6. We compiled the C code for Noise-C and Noise* using gcc 7.5.0. We used QEMU to run WireGuard for benchmarking, the Criterion 0.3.3 crate to benchmark the Rust code and the Criterion 1.5.9.0 package to benchmark the Haskell code. We observe that our Noise* implementation either beats existing implementations for handshakes per second; or is competitive with the state-of-the art IKpsk2 implementation from WireGuard. Detailed analysis reveals that without a GC (e.g. Noise-C), the performance is dominated by the DH computations.

Security Analysis Comparison. Figure 4.8 compares our symbolic security analysis with prior formal proofs of Noise protocols. The closest related works are Noise Explorer and Vacarme, which both analyze (almost all) Noise protocols against Dolev-Yao attackers. Noise Explorer [262] compiles each handshake pattern to a ProVerif model and verifies it against a series of reachability queries corresponding to the different Noise secrecy and authenticity levels. The analysis of each protocol takes between 30 minutes and 24 hours. Vacarme generates Tamarin models for each protocol, and analyzes it against the strongest threat model supported by the protocol. Analyzing 53 protocols takes a total of 74 CPU days.

The key difference in our approach is that we verify a generic executable Noise specification using a modular, semi-automated proof technique based on dependent types. Hence, we are able to verify the whole protocol specification in about 9 minutes, which amounts to 10s per pattern. Furthermore, our proofs are for an executable specification of the whole Noise protocol stack, whereas Noise Explorer and Vacarme only focus on the protocol messaging code. Conversely, the protocol-level verification results of Vacarme are stronger than ours, since Tamarin can handle equivalence properties like anonymity and has a more precise model of Diffie-Hellman.

The security proof overhead for Noise⋆ can be estimated by the ratio between our DY⋆ proof and the functional specification, which is 2.4. Note however, that this is a proof for all 59 patterns, and is still just a fraction of the effort of developing the Low⋆ implementation.

Figure 4.8 also notes other work on the computational analysis of Noise protocols: [263] defines a new security model for cryptographically analyzing multiple Noise protocols using pen-and-paper proofs; [275] describes a manual proof of WireGuard, including an analysis of IKpsk2; [253] presents a mechanized cryptographic proof of WireGuard using CryptoVerif. These works use a more precise cryptographic model than the symbolic models in our work or in Vacarme. However, the work needed to prove each protocol is significantly higher. Linking our verified implementations to computational proofs is an interesting direction for future work.

Other Related Work. Apart from work on Noise, prior works have investigated the automatic generation of protocol code from verified high-level protocol specifications, yielding implementations in Java [276], OCaml [277, 278], and F# [279, 280]. Each of these tools has been applied to a handful of protocols; the generated protocol code is tuned for correctness rather than performance and relies on unverified cryptographic libraries. In contrast, by relying on the F⋆ ecosystem, we generate high-performance C code that is provably correct, memory safe, and linked to a verified cryptographic library. Furthermore, the flexibility and succinctness of the Noise specification language enables us to automatically generate verified implementations for 59 distinct protocols, yielding a comprehensive protocol library. Other prior works have focused on efficient code generation for specialized cryptographic constructions like multi-party computation and zero-knowledge proofs; we refer the reader to [255, 281] for a survey of this line of work. Finally, a long line of work has investigated techniques for directly verifying cryptographic protocol implementations written in F# [252, 282–284], F⋆ [234, 285], Java [259, 286], and C [287–289]. In these settings, each protocol implementation must be verified independently, whereas our compiler-based approach allows us to verify a large class of protocol implementations once and for all.

4.7 Conclusion

We have presented a Noise Protocol Compiler embedded within F⋆. Our compiler is verified once; then, for any choice of Noise Protocol and matching cryptographic implementations, it produces an efficient, low-level implementation in C. We generate not only protocol transitions, but also the entire protocol stack, including state machine, device and session management, user-configurable key policies, long-term key storage, and dynamic security levels. At all layers, we guard against user error by providing robust APIs. We go beyond the usual trifecta of memory safety, functional correctness and side-channel resistance, by connecting our verified verified stack to a symbolic security proofs based on the DY⋆ framework. None of these results affect performance, as our C code beats most existing implementations.

Chapter 5

Modularity and Zero-Cost Abstractions for Program Verification

With the Noise* project, we explored the problem of moving up the software stack by going beyond cryptographic primitives and developing secure protocol implementations. When moving up to higher-level software however quickly comes the problem of implementing generic code. Even more challenging in our case is the fact that we need this code to be low-level, efficient, and verified. We now turn to this problem with a project which introduces zero-cost functors for program verification in F*.

5.1 Introduction

Within the span of a few years, formal verification has gone mainstream. Previously confined to academic circles, the idea of proving properties about security-critical code is now widely accepted. Case in point: a major cloud company like Amazon will pay for a full sponsored article in the Wall Street Journal [290], touting the benefits of formal verification for its cloud computing unit.

Such security-critical code often lies on the critical path of larger subsystems; users therefore expect security-critical code to be not only secure and reliable, but also fast. To that effect, programmers continue to resort to low-level programming idioms and manual memory management, which allows them to exert fine-grained control on the structure of their code, and hence squeeze every last inch of performance out of it [291], sometimes directly leveraging hardware facilities to do so [292, 293]. This unfortunately comes at a cost; taming the complexity of such programs is error-prone, leading to abundant mistakes with dire consequences [294–308].

Aiming to address this problem, formal verification practitioners have thus focused

on code that is both security-critical and low-level. Success stories include verified cryptography, with e.g., HACL⋆/EverCrypt [127, 128, 233] (integrated into the Linux kernel, Mozilla Firefox, and the Tezos blockchain), or Fiat Cryptography [192] (integrated into Google's BoringSSL cryptographic library); verified parsers, with e.g., EverParse [256, 309] (integrated into Microsoft's Hyper-V network virtualization stack); verified kernels, such as CertiKOS [310, 311] or seL4 [312]; and many more.

But for all the success stories, it remains a technical challenge to author and verify a system that is simultaneously large-scale, low-level, and performant. Large-scale verification projects abound; one must only think of e.g., Mathlib [313], a large collection of mathematical proofs and theorems written in Lean3, which recently crossed the million-line threshold. Large-scale *and* low-level verified projects are not unheard of: seL4 [314], based on a dialect of Haskell, or CertiKOS [311], written in Coq, both demonstrate that one can write non-trivial pieces of software (such as OSes) that deal with low-level concerns and deliver *reasonable* performance. But when it comes to large-scale, low-level *and* competitively performant verification projects, few candidates come to mind. One reason is that verification remains onerous: expert proof engineers are rare, and their task is hard enough; as such, advances in proof engineering and reusable abstractions are badly needed to increase productivity. Nowhere is this more salient than when trying to verify low-level code. Verification calls for high-level abstractions and extreme modularity, while low-level efficient code calls for breaking up those very abstractions barriers. This, in our opinion, has hindered the development of large-scale, low-level, and efficient libraries.

High-level abstractions are well-known to functional programmers; they may include type-level abstraction and polymorphism; module interfaces and functors; type classes. They are also known to the mythical "real-world" programmers: templates and concepts in C++, or traits in Rust, also support modularity in the large. But sooner or later, the programmer will, on the quest to ultimate performance, pull low-level tools from their arsenal. The infamous C preprocessor oftentimes makes an appearance, with tricks so frightening that basic decency prevents us from describing them here [315]. And these are not just simple, straightforward patterns such as loop unrolling; entire polymorphic data structures are emulated using the C preprocessor. This route usually ends in pain and suffering, with unmaintainable code, subtle mistakes, and generally, the inability to reason about such code. There is thus a tension between going low-level for efficiency, and introducing high-level concepts, abstraction and modular boundaries to make the code easier to reason about. This tension is heightened in the context of verification: the need for modular, high-level code is even greater, so as to ease verification; but the pressure for efficient, low-level code is also stronger, to meet practitioners' performance

requirements and thus give our verified code the chance to be integrated and then deployed in mainstream software.

In this chapter, we set out to have our cake and eat it. That is, to have efficient, low-level, verified code, and to do so *at a large scale*, in a verified software project that exceeds 100,000 lines of verified code. Worded differently, we want to reconcile the modularity of, say, SML or OCaml's functors, or Haskell's type classes, with the efficiency of Rust traits and the infamous "zero-cost abstraction" of C++ templates, *for verified code*. And we want to have our cherry on top of the cake: we set out to do so *without extending the Trusted Computing Base (TCB)* of the tools we use. We design, implement and evaluate our techniques within the F* dependently-typed proof assistant, which culminate in the following contributions.

First (Section 5.3), we propose a proof engineering methodology that allows one to structure their verified code as they would, say, with functors, all the while still producing idiomatic, low-level code with readable functions and no runtime overhead of any kind.

Second (Section 5.4), we observe that using this methodology is burdensome in practice, because structuring the code to fit our proof engineering pattern requires a fair amount of bookkeeping. We thus automate our methodology by designing a DSL that guides an automated code-rewriting transformation which automatically applies the pattern from Section 5.3 to the user's code. In practice, this allows the user to write their code in a modular, high-level, natural style that emulates ML's functors, while relying on inlining and partial evaluation to eliminate the high-level abstractions and make our discipline truly, a zero-cost abstraction. The DSL is interpreted via meta-programming, specifically, via elaborator reflection; in essence, we script the compiler, and add an early compilation stage that takes our functor DSL and evaluates it away. The techniques we introduce are implemented in user-space, meaning we do not modify the compiler and leave the TCB intact, so as to provide the same guarantees as code written without our libraries.

Third (Section 5.5), we explain how several algorithms previously released via the HACL* and EverCrypt projects were, in reality, relying on our techniques to scale up, and to avert engineering and usability disasters. We review a series of case studies and show how several cryptographic primitives can be implemented using our DSL so as to maximize code sharing and minimize maintenance.

Fourth (Section 5.6), and final, we examine a large case study: the streaming API, a cryptographic construct that transforms an unsafe, block-based algorithm into a safe, high-level API by means of an internal buffer. With our DSL, we write a generic streaming API once, then instantiate it "for free" over any unsafe block-based algorithm.

Out of a dozen instantiations of our streaming functor, six have been integrated into the reference implementation of the Python programming language. This case study is a contribution on its own: to the best of our knowledge, no one had precisely described, captured with dependent types and implemented generically what it means to turn a block-based algorithm into a streaming API.

Our evaluation section quantifies the improvements in programmer productivity and effectiveness stemming from the use of our methodology. We have evaluated our techniques on the HACL⋆ project, and found that they were the key ingredient that allowed HACL⋆ to cross the barrier of 100,000 lines of verified source F⋆ code. Without our work, modularizing and scaling up the codebase would have been impossible.

We conclude and observe that while our case studies focus on cryptographic code, our techniques are general and can be applied to data structures, or more generally, any situation that calls for modular proofs of low-level programs, as evidenced by our choice of running example (Section 5.3).

Contributions. The work in this chapter is adapted from a paper published at ICFP in 2023 [316]. Jonathan came up with the idea of using the "functor" encoding and of automating it by using meta-programming, and applied this methodology to several cryptographic primitives. I extended the methodology to apply it to a larger collection of primitives, in particular on the implementations required by the Noise⋆ project; this required a deep rework of the original version of the streaming hash APIs. Aymeric later added more primitives. I also used some of the techniques mentioned in this chapter directly in the Noise⋆ project to make the implementation generic in, say, the cryptographic primitive implementations or the peer identifiers. More precisely, I used the idea of writing generic mk_ functions that are later specialized, as we do in Section 5.3.2, but where the mk_ functions are parameterized in a style closer to the "functor" parameters of Section 5.3.1 (i.e., without an index), because it didn't leverage the automation that we introduce in the Section 5.4. I decided not to mention those techniques in the previous chapter as a detailed description fits more naturally here.

5.2 Background

In this section, we introduce the background required to understand our methodology. We start with an overview of our verification environment: F⋆ (Section 5.2.1). We then present a well-known technique to encode functors with dependent types (Section 5.2.2) that we build upon in the later sections.

5.2.1 F*, Low*, Meta-F*

F* is a state-of-the-art verification-oriented programming language. Hailing from the tradition of ML [317], F* features dependent types, refinement types, and a user-extensible effect system [318], which allows reasoning about IO, concurrency, divergence, various flavors of mutability, or any combination thereof. For verification, F* uses a weakest precondition calculus based on Dijkstra Monads [319, 320], which synthesizes verification conditions that are then discharged to the Z3 SMT solver [321]. Proofs in F* typically are a mixture of manual reasoning (calls to lemmas), semi-automated reasoning (via tactics [218]) and fully automated reasoning (via SMT).

Low* is a subset of F* that exposes a carefully curated subset of the C language. Using F*'s effect system, Low* models the C stack and heap, and allocations in those regions of the memory. Low* also models data-oriented features of C, such as arrays, pointer arithmetic, machine integers with modulo semantics, const pointers, and many others via a set of distinguished libraries. Programming in Low* guarantees spatial safety (no out-of-bounds accesses), temporal safety (no double frees, no use-after free) and a form of side-channel resistance [127, 322]. All of these guarantees are enforced statically and incur no run-time checks. To provide a flavor of programming in Low*, we present the swap function below. We first focus on the various typical Low* features of this function signature.

```
let swap (x y : pointer U32.t) : ST unit (requires ...) (ensures ...) =
  let xv = deref x in
  let yv = deref y in
  upd x yv;
  upd y xv
```

Functions in Low* are annotated with their return effect, in this case ST, which indicates that the function may perform heap allocations[1]. Functions without a return effect are understood to be total. The input parameters have type pointer U32.t, i.e., pointers to 32-bit unsigned machine integers with modulo semantics. Functions are specified using pre and post-conditions, which we omit here and whose explanation we defer until Section 5.6. Finally, the implementation of swap simply dereferences x and y (deref), then updates them while swapping their values (upd).

Erasure and extraction in F* follows Letouzey's extraction principles for Coq [323]. After type-checking and performing partial evaluation, F* *erases* computationally-

[1] Low* actually distinguishes two stateful effects, Stack for functions which only allocate on the stack (no memory leaks), and ST for functions which also allocate on the heap. In this chapter, we only use ST for the purpose of simplicity.

irrelevant code and performs extraction to an intermediary representation dubbed the "ML AST".

For erasure, F⋆ eliminates type refinements, pre- and post-conditions, and generally replaces computationally irrelevant terms with units. F⋆ also removes calls to (pure) unit-returning functions, which means that calls to lemmas are also eliminated. For extraction, F⋆ ensures that the "ML AST" features only prenex polymorphism (i.e., type schemes), and that it is annotated with classic ML types. In the context of this chapter, we are only concerned with the generation of C code, which is possible only on a subset of the "ML AST"; when extracting for C, a battery of checkers verifies that the code is in the proper subset.

KaRaMeL [322] compiles the "ML AST" to *readable*, *auditable* C by using a series of small, composable passes. The KaRaMeL preservation theorem [322] states that the safety guarantees in Low⋆ carry over to the generated C code. We show below the result of compiling swap to C.

```
void swap(uint32_t *x, uint32_t *y) {
  uint32_t xv = *x;
  uint32_t yv = *y;
  *x = yv;
  *y = xv;
}
```

5.2.2 Encoding Functors With Dependent Types

We now illustrate the challenge of combining generic, modular programming (good for proofs) with low-level compilation (good for efficiency). We start with a running example that we will reuse in Section 5.3: an imperative key-value map implemented using an associative list. For simplicity of exposition, we use standard algebraic datatypes, such as list. Low⋆ features low-level data structures, notably linked lists; however, these would significantly complicate our running example with notions of memory footprints and memory reasoning. We thus stick with list for the chapter, and provide a complete low-level example relying on linked lists in the supplementary material.

To enable code reuse, we wish to make the associative list generic in the type of its keys and values. If we were to use a language like OCaml or Haskell we would naturally implement this map by using a functor or type classes. Listing 1 illustrates this with an OCaml functor named MkMap, which takes an argument EqType containing a type for keys k, and a corresponding decidable equality. The MkMap functor implements find using a loop and mutable references, generically, for any type of keys k and corresponding equality eq.

```
1  module type Map = sig
2    type k
3    val find: k -> (k * 'a) list -> 'a option
4  end
5
6  module type EqType = sig
7    type t
8    val eq: t -> t -> bool end
9
10 module MkMap (E : EqType) :
11   Map with type key = E.t = struct
12   type k = E.t
13   let find x ls =
14     let b = ref true in
15     let lsp = ref ls in
16     while !b do
17       match !lsp with
18       | [] -> b := false
19       | (x', _) :: tl ->
20         if E.eq x x' then b := false
21         else lsp := tl done;
22     match !lsp with
23     | [] -> None
24     | (_, y) :: _ -> Some y
25 end
```

Listing 1: An Associative Map Implemented in OCaml

We want to attain the same modularity when verifying code in a prover like F⋆. As a first attempt, we can reuse a well-known technique [324, 325] to encode this OCaml functor using dependent types (Listing 2). The Map module signature becomes a record map, and the type k of keys becomes a record field. Since this is a dependent record, eq may refer to k. We implement the MkMap functor with the mk_map function, which receives an instance of eq_type along with a type a. The return type of mk_map uses a refinement: a value m:map has type m:map a{m.k == e.t} if it satisfies the logical predicate m.k == e.t; this equation exactly encodes the condition type key = E.t of the OCaml code (line 11). Finally, Low⋆ uses a special while combinator for loops, which takes two closures as inputs, for the loop condition and the loop body respectively; the implementation of find otherwise mimics its OCaml counterpart. As the code is stateful, i.e., it lives in the ST effect, it requires annotations such as pre- and post-conditions; at this stage, we are concerned with the shape of the code and not its correctness: we thus omit them for simplicity.

Even when assuming that all data structures are suitably low-level, the issue remains that the implementation of find manipulates dictionaries (e.g., instances of eq_type). Note that this is not specific to our encoding: we would have the same problems had

```
type map (a : Type) = {
  k: Type;
  find: k -> list (k * a) -> ST (option a) ... }

type eq_type = { t: Type; eq: t -> t -> bool; }

let mk_map (e : eq_type) (a : Type) :
  m:map a{m.k == e.t} = {
  k = e.t;
  find = (fun x ls ->
    let b = alloc true in
    let lsp = alloc ls in
    while (fun () -> !* b)
      (fun () ->
        let ls = !* lsp in
        match ls with
        | [] -> upd b false
        | (x', _) :: tl ->
          if e.eq x x' then upd b false
          else upd lsp tl);
    match !* lsp with
    | [] -> None | (_, y) :: _ -> Some y) }
```

Listing 2: An Associative Map Implemented in F⋆

we used functors or type classes, and this is the case for the OCaml implementation of the map. Dictionary-passing is problematic because it has a cost at runtime. Worse, our implementation doesn't fit in the Low⋆ subset and thus can't be extracted to C; indeed, the resulting code would manipulate records with fields containing types, which is not supported in C. We show in the next section how we solved this problem.

5.3 Writing Low-Level, Modular Code

We showed in Section 5.2.2 how one can achieve the same level of modularity and genericity in F⋆ as in a regular, high-level programming language like OCaml, by encoding functors with dependent types by means of an already known technique. But now we ask: how can one turn this into idiomatic, efficient low-level code? In the coming section, we answer by introducing new methods which build upon the technique explained in Section 5.2.2. We stick to the same running example, that is an imperative key-value map, and for the purpose of illustration, we assume once again that all data structures, such as list, are suitably low-level.

```
1  (* Map instantiation *)
2  let str_eqty : eq_type = { t = string; eq = String.eq; }
3  let ifind = (mk_map str_eqty int).find
4
5  (* After partial evaluation *)
6  let ifind (x: string) (ls: list (string * int)): option int =
7    let b = alloc true in let lsp = alloc ls in
8    while (fun () -> !* b)
9      (fun () ->
10      let ls = !* lsp in
11      match ls with
12      | [] -> upd b false
13      | (x', _) :: tl ->
14        if String.eq x x' then upd b false
15        else upd lsp tl);
16   match !* lsp with
17   | [] -> None
18   | (_, y) :: _ -> Some y
```

Listing 3: find after Instantiation (Top), then Partial Evaluation (Bottom)

5.3.1 Making Functors Zero-Cost: A First Attempt

We now present a first naive technique that allows the user to generate specialized Low⋆ code (i.e., without dictionary-passing), at the expense of code size explosion. The key idea is to perform partial evaluation at extraction time to inline all uses of eq_type (and a). To do so, we can leverage the F⋆ *normalizer* to symbolically reduce terms. The normalizer is not an F⋆ specificity; it is at the core of dependent type systems, and therefore a component of the type-checker of any dependently typed language. As such, this component is part of the TCB of type-theory-based proof assistants.

The user proceeds as follows. First, they pick concrete values for the functor arguments. In our example (Listing 3), the user picks str_eqty and int for the mk_map parameters e:eq_type and a:Type, respectively. Then, the user applies those arguments to the functor itself, hence defining an *i*nstantiated version of find, dubbed ifind (line 3). The normalizer then kicks in and β-reduces the application of ifind to its concrete arguments. By inlining the body of find, then by simplifying some terms like the projection str_eqty.eq, all uses of records inside ifind are removed; the resulting specialized ifind is indistinguishable from a direct, monomorphic implementation of find. We show the result of partial evaluation in Listing 3.

This approach therefore allows us to turn our functors into zero-cost abstractions. The caveat with this style, however, is that *every single function* needs to be inlined, except for the top-level functions that make up the API entry points. This is fine for our small example; but in a real-world development, this leads to both code size

```
type dv = {
  pid : Type;
  send : pid -> list (pid * ckey) -> bytes -> option bytes;
  recv : pid -> list (pid * ckey) -> bytes -> option bytes; }

type cipher = {
  enc : ckey -> bytes -> bytes;
  dec : ckey -> bytes -> option bytes; }

let mk_dv (m : map ckey) (c : cipher) : d:dv{d.pid == m.k} = {
  pid = m.k;
  send =
    (fun id dv plain ->
     Option.map (m.find id dv) (fun sk -> c.enc sk plain));
  recv =
    (fun id dv secret ->
     Option.map (m.find id dv) (fun sk -> c.dec sk secret)) }
```

Listing 4: Implementation in F* of a Peer Device for a Secure Channel Protocol

explosion (we insert a copy of find's body at each call-site), and unacceptable code quality (implementing an algorithm as a single 20,000-line C function is generally frowned upon). To illustrate this more concretely, we introduce in Listing 4 a client of find, known as a "device", a high-level data structure used in communication protocols to store a map from peer identifiers to session keys, i.e., a map from unique participant identifiers to the cryptographic keys used for secure communications.

A device should implement two functions to communicate with participants. The send function takes as arguments a peer identifier id, a map from peer identifiers to cryptographic keys (of type ckey), and a message plain. It looks up the key associated to id, and finally uses it to encrypt plain. The recv function performs the dual operation, i.e., it searches for the key to decrypt a message received from a known peer. The choice of the peer identifier type pid is orthogonal to the implementation of a device dv; we can therefore write a generic implementation parametric in pid and accordingly in the map from peer identifiers to cryptographic keys. Furthermore, this device can be useful in a variety of contexts and with a range of ciphersuites, and should thus be independent of the specifics of any cryptographic encryption algorithm: we also parameterize the implementation mk_dv with the encryption and decryption functions, encapsulated in a record of type cipher.

Equipped with a generic device, we can, as in the map example, instantiate it for a specific choice of peer identifiers and cryptographic functions, before applying partial evaluation to get specialized code which does not manipulate dictionaries. Unfortunately, doing so would lead us into the pitfall we mentioned earlier, where the code for find is duplicated in both the instantiations for send and recv. In our experience interacting

with maintainers of some of the most popular open-source projects, such aesthetic *faux-pas* are bad enough that a practitioner will dismiss our code as 'not serious' and 'too verbose', raising barriers to its integration to existing codebases. In this regard, we insist on the fact that the HACL⋆ code, part of which we applied our methodology on (Section 5.5), was deployed in real-world projects such as the NSS library or Python.

5.3.2 A General Rewriting Pattern for Fine-Tuned Code Generation

When specializing functions like send and recv, we want them to call the same *specialized version* of find, rather than duplicate the body of find. In effect, we want to perform whole-program specialization (in the style of MLton [326]) while preserving the shape of the *static* call-graph (in order to give the programmer enough control so as to generate palatable code). To do so, we propose a *modular* approach that allows us to rewrite *each function in isolation* without knowing yet how the function will later be instantiated, all the while avoiding the need for inlining everything. We proceed as follows. Instead of using a dependent record, for each function, we add additional parameters that stand in for the callees that need to be specialized; and we re-implement the function body to refer to those arguments. For instance, send and recv become the mk_send and mk_recv functions in Listing 5. Note that a function is parameterized with *exactly* its callees: for instance send is parameterized by enc but not dec, while it is the converse for recv. We intentionally refrain from using a record ("functor")-based encoding like in the previous sections: this would rapidly lead to a proliferation of type definitions, as there would typically be one record per definition. This would make both programming and maintaining our codebase tedious, as the addition or modification of any element in the record would require changing all occurrences across the call-graph.

Anticipating a bit on the automation we introduce in Section 5.4, we request that the polymorphism be prenex, i.e., that all type parameters be captured by the first argument; this does not restrict expressivity, and allows us to avoid having extra type-level dependencies across function arguments which would be difficult to automatically handle. More specifically, we keep the generic types in a record, that we call the "index" and make the first parameter of the function. This index must capture all the choices of parametricity for the types. In practice, as we'll see in concrete, real-world examples in Section 5.5, we often pick the index to be an enumeration, but this is not a requirement of our approach. The index can also range over an infinite number of elements, as is the case for the generic type pid in Listing 5. In this specific example, since we are only parametric in one type, we dispense with a record type and parameterize our functions

```
let mk_send (pid : Type)
  (find : pid -> list (pid * ckey) -> option ckey)
  (enc : ckey -> bytes -> bytes)
  (id : pid) (dv : list (pid * ckey)) (plain : bytes) : option bytes =
  match find id dv with
  | None -> None
  | Some sk -> Some (enc sk plain)

let mk_recv (pid : Type)
  (find : pid -> list (pid * ckey) -> option ckey)
  (dec : ckey -> bytes -> option bytes)
  (id : pid) (dv : list (pid * ckey)) (plain : bytes) : option bytes =
  match find id dv with
  | None -> None
  | Some sk -> dec sk plain
```

Listing 5: Parameterizing send and recv by their Callees

over pid directly.

We also apply this approach to the find function previously presented. This function is parametric in two types: the type of keys k, and the type of values v of the map. We collect both types in a record of type mindex, which becomes the first argument of mk_find, presented in Listing 6.

Importantly, we drop the "functor" encoding for the functions but not the types; i.e., we use a record which holds all the type parameters. Keeping this encoding for types doesn't lead to the same proliferation of records as for functions. Indeed, type parameters tend to be fewer, change less often, and their parameterization tends to be more uniform accross functions.

We finally show an instantiation of those generic definitions in Listing 7, where aes_enc and aes_dec are encryption/decryption functions for AES-GCM, one of the most widely used authenticated encryption algorithm. We omit their implementation, which is irrelevant for presentation purposes; they can be provided by a separate cryptographic library. With this new encoding, we can individually unfold the definitions of mk_find, mk_send and mk_recv before simplifying the projections over record fields, e.g., i.k, while preserving the call graph; we show the result of the partial evaluation in Listing 7. Note in particular that the definition of mk_find is not inlined in the resulting isend and irecv; both functions instead call the specialized ifind.

Our goals are met: we have described a general rewriting pattern that allows us to write generic implementations, that can be specialized for a choice of types (the "index"), while preserving the shape of the static call-graph and hence produce high-quality low-level code.

```
type mindex = { k : Type; v : Type }

let mk_find (i: mindex) (eq: i.k -> i.k -> bool) (x: i.k)
            (ls: list (i.k * i.v)) : option i.v =
  let b = alloc true in
  let lsp = alloc ls in
  while (fun () -> !* b)
    (fun () -> let ls = !* lsp in
     match ls with
     | [] -> upd b false
     | (x', _) :: tl ->
       if eq x x' then upd b false
       else upd lsp tl);
  match !* lsp with
  | [] -> None
  | (_, y) :: _ -> Some y
```

Listing 6: Rewriting find to Follow a Systematic Pattern

```
(* Instantiation *)
let i = { k = string; v = ckey; }
let ifind = mk_find i String.eq
let isend = mk_send string ifind aes_enc
let irecv = mk_recv string ifind aes_dec

(* After partial evaluation *)
let ifind x ls =
  let b = alloc true in
  let lsp = alloc ls in
  while (fun () -> !* b)
    (fun () ->
     let ls = !* lsp in
     match ls with
     | [] -> upd b false
     | (x', _) :: tl ->
       if String.eq x x' then upd b false
       else upd lsp tl);
  match !* lsp with
  | [] -> None
  | (_, y) :: _ -> Some y

let isend id dv plain =
  Option.map (ifind id dv) (fun sk -> Some (aes_enc sk plain))

let irecv id dv secret =
  Option.map (ifind id dv secret) (fun sk -> aes_dec sk secret)
```

Listing 7: Instantiation (Top) and Partial Evaluation (Bottom) of the Map and Device Functions

Discussion Previous work in F⋆ used a precursor to the techniques we present in this section. In particular, HACL⋆ [127, 233] made heavy use of specialization and partial evaluation to factor out large pieces of code, for instance by writing a single generic implementation of Poly1305 for three variants (C, C+AVX, C+AVX2). However, this early style had two issues. First, it led to code size explosion due to excessive inlining, which was solved by manually introducing alternating levels of generic and specialized functions, a tedious and time-consuming task. Second, it relied on closed enumerations (i.e., an inductive with constant constructors) as opposed to the open-ended "indices" that we introduce in the present section. The first point is addressed by our DSL, the rewriting tactic and the systematic higher-order pattern it produces. Regarding the second point, parameterizing over closed enumerations is a legacy style (Section 5.5) that is acceptable as long as the user is adamant that no further cases will be added. Indeed, adding a new case to the enumeration entails a re-verification of the generic code, affecting modularity. We strongly encourage users to try to define a generic index type (i.e., at type Type), which provides more flexibility, modularity, and allows the user to trivially add new specializations without affecting the generic code. This requires, however, more thought on the part of the user to correctly define the index type. Overall, this chapter groups in one place the culmination of all the techniques which were introduced to make the Everest project [327] scale up to its current size.

5.4 Meta-Programmed Static Call-Graph Rewriting

In the previous section, we identified a programming pattern that allowed us to modularly write verified code, in a way reminiscent of ML functors, by rewriting our low-level functions into a higher-order form that lends itself to code specialization via partial application. In practice, manually writing code which uses this pattern requires a fair amount of tedious, administrative work. In this section, we thus set out to relieve the user from this burden by designing a small DSL, to be more precise a subset of Low⋆ extended with a mechanism of annotations, by which the user can write code in a natural style before calling a rewriting procedure which automatically turns this code into a higher-order form. To do so, we i) propose a small usability tweak to make parameterization easier, then ii) formally define our rewriting rules, and iii) devise a frontend language that allows the user to express their intent via a mechanism of annotations. Our rewriting rules are interpreted by a custom pre-processing phase implemented via elaborator reflection, i.e., "scripting the compiler". In effect, we are adding a user-defined early compilation stage.

```
type mindex = { k : Type; v : Type }
assume val eq (i : mindex) : i.k -> i.k -> bool

let find (i : mindex) (x : i.k) (ls : list (i.k * i.v)) : option i.v =
  let b = alloc true in
  let lsp = alloc ls in
  while (fun () -> !* b)
    (fun () ->
     let ls = !* lsp in
     match ls with
     | [] -> upd b false
     | (x', _) :: tl ->
       if eq i x x' then upd b false
       else upd lsp tl);
  match !* lsp with
  | [] -> None
  | (_, y) :: _ -> Some y
```

Listing 8: Hoisting Callee Arguments from find

5.4.1 A Declarative Style for Callee Arguments

The higher-order, rewritten functions presented in Section 5.3.2 allow us to write low-level, verified code in a modular fashion. However, there remains a usability problem. The functions that we parameterize over, like eq, need to be brought in scope frequently, as eq has many callers. This is currently achieved by making *every function* in our development that needs it parametric over eq, which incurs a non-trivial amount of boilerplate. Even worse, in the case of an actual algorithm, e.g., Curve25519 (Section 5.5.2), we parameterize the algorithm over a dozen operations. Asking the user to add as many arguments to every declaration *and* call-site would be too onerous.

To alleviate these concerns, we propose to adopt a more declarative style. For presentation purposes, let us reuse the map example from the previous section. Instead of explicitly parameterizing the definitions (e.g., find) with their generic parameters (e.g., the decidable equality eq), we introduce the parameters of our implementations as top-level declarations annotated with the assume qualifier, as shown in Listing 8. We achieve the same effect as before: the declaration is in scope for our entire development. But this time, we avoid the syntactic overhead. Once this declaration is in the scope of find, it can be freely used and referred to in the body of the function. In practice, the index is an implicit argument, which further reduces the syntactic burden.

With this approach, changing the signature of eq becomes less dreary. Instead of performing modifications in all functions relying on eq, it suffices to tweak its top-level declaration. The reader might wonder why one would need to change the type of eq; while this example is overly simple for presentation purposes, making minor

modifications to specifications to, say, add a missing invariant or fix a mistake in a precondition is common when doing incremental verification. Leveraging F*'s SMT-backed automation, small changes to the callee often do not require modifying the callers.

An assumed declaration in F* is tantamount to introducing a hole in our code. Trying to generate C code containing such a hole would lead to C extern declarations, and raise compilation failures unless an external definition is provided by the user. In the following section, we will describe how to fill this hole, and ensure that the provided definition matches the assumed function type.

5.4.2 Static Call-Graph Rewriting

While hoisting callee arguments to assumed top-level declarations reduces code clutter, it only alleviates some of the burden that a programmer is facing when using our methodology. Relying on top-level assumed declarations for callees is not always desirable. In our map and device example, while send and recv are parametric in find, find itself is implemented in the module; adding an assumed type declaration would be redundant. We would rather preserve the existing definition of find, and automatically rewrite, e.g., send into its mk_send counterpart that takes find as an argument. We show in this section how to reach this goal using metaprogramming.

Rewriting, Formally Following the programming pattern described in Section 5.3.2, we assume that every function node g_i in the static call-graph is parameterized over an argument $\mathsf{idx} : t_{\mathsf{idx}}$ that represents the specialization index, and that this argument appears in first position.

At definition site, every function definition $\mathsf{let}\ f\ \mathsf{idx}\ \overline{x} =$ is replaced by $\mathsf{let}\ \mathsf{mk_f}\ \mathsf{idx}\ (g_1 : t_{g_1}\ \mathsf{idx})\ ...(g_n : t_{g_n}\ \mathsf{idx})\ \overline{x} =$. The g_i represent all the callees in the body of f. The t_{g_i} are the types of the original g_i, abstracted over the index idx, that is, if the type of g_i was the dependent arrow $\mathsf{idx} : t_{\mathsf{idx}} \to t$, then t_{g_i} is the dependent function $t_{g_i} = \lambda(\mathsf{idx} : t_{\mathsf{idx}}).\ t$, which allows us to write the application $t_{g_i}\ \mathsf{idx}$. At call-site, when encountering a call $g_i\ \mathsf{idx}\ \overline{e}$, the call becomes $g_i\ \overline{e}$ and references the bound variable g_i instead of the global name.

Taking our running example, we have $t_{\mathsf{idx}} = \mathsf{mindex}$, and $\mathsf{eq} : i : t_{\mathsf{idx}} \to i.k \to i.k \to \mathsf{bool}$. Our goal is to make sure that find becomes parameterized over an argument eq specialized for the *same value of the index* as find. To achieve that, we pick $t_{\mathsf{eq}} = \lambda(i : t_{\mathsf{idx}}).\ i.k \to i.k \to \mathsf{bool}$, and thus rewrite find into $\mathsf{let}\ \mathsf{mk_{find}}\ (i : t_{\mathsf{idx}})\ (eq : t_{\mathsf{eq}}\ \mathsf{idx})$, which then reduces into $\mathsf{let}\ \mathsf{mk_{find}}\ (i : t_{\mathsf{idx}})\ (eq : i.k \to i.k \to \mathsf{bool})$, where the index i is the

```
Section map.
 Record mindex := { k : Set; v : Set }.
 Variable eq : forall i:mindex, i.(k) -> i.(k) -> bool.

 Definition find (i : mindex) (x : i.(k)) (ls : list (i.(k) * i.(v))) :
   option i.(v) :=
   ...
End map.
```

Listing 9: An Equivalent of find Using Coq's Section Variables

same everywhere, meaning that eq is specialized for the same choice of types as find.

Recursively Traversing the Call-Graph The rewriting presented above is highly modular; it allows us to rewrite each function in isolation. Following the same process as for find, we notice when rewriting send that it should be parameterized by a specialized version of find itself. Empirically, we observe the composition of parametric functions to be a common pattern. Instead of manually applying our rewriting to send, recv, and find, we recursively traverse the call-graph, automatically performing rewriting on the definitions of all callees of the function being rewritten. Using this approach, a user only needs to invoke the rewriting on the API endpoints of their library, i.e., specific top-level functions. When encountering a top-level assume declaration, as described in Section 5.4.1, the traversal stops. Callers end up with the correct additional parameters, and it will be up to the user to exhibit suitable instantiations for the assumed functions.

Section Variables Our mechanism is very similar to the section variables mechanism provided by provers such as Coq and Lean. For instance, it would be possible to automatically parameterize find with eq by using a section in which eq is declared as a Variable; we show such an example for Coq in Listing 9. The section mechanism in its current shape would however lead to *slightly* more work on the user side: it would work for all the definitions that we mark as assume in F⋆, as we would simply declare them as section variables, but doesn't provide a straightforward way of parameterizing functions like send and recv with find. Indeed, we would need to both define mk_find *and* declare find as a section variable for send and recv to use it, so that they get correctly parameterized; with our call-graph rewriting we write a single definition for find.

5.4.3 Fine-Grained Code Specialization

While inlining all functions, as explained in Section 5.3.1, is not desirable, specializing all functions in the call-graph can also conflict with a programmer's intent. Many

```
let while_cond (b: pointer bool) (_:unit) = !*b

let while_body (i: mindex) (b: pointer bool) (lsp: list (i.k * i.v))
               (x:i.k) (_:unit) =
  let ls = !* lsp in
  match ls with
  | [] -> upd b false
  | (x', _) :: tl ->
    if eq x x' then upd b false
    else upd lsp tl

let find (i : mindex) (x : i.k) (ls : list (i.k * i.v)) : option i.v =
  let b = alloc true in
  let lsp = alloc ls in
  while (while_cond b) (while_body i b lsp x);
  match !* lsp with
  | [] -> None
  | (_, y) :: _ -> Some y
```

Listing 10: Hoisting Loop Closures

functions, e.g., alloc and upd from the standard library, are not parametric and thus do not require specialization; furthermore, to reduce the size of proof contexts and ease verification, programmers often rely on auxiliary functions that are expected to be inlined at extraction-time. Consider for instance the while combinator used to implement find. While inlining the closures for the loop condition and the loop body is reasonable for small examples, a programmer might find it useful to hoist them for verification purposes, as shown in Listing 10, while unfolding them at extraction-time to retrieve idiomatic code.

Designing generic heuristics to determine which functions should be specialized or inlined is tricky; getting them wrong risks alienating developers when they do not obtain the shape of the code they expect. Instead of a generic solution, we prefer to leverage programmers' knowledge of their code. Using F⋆'s annotation system, we provide two attributes, Specialize and Eliminate, that enable a fine-grained control on the rewritings performed by our approach.

Before rewriting, declarations annotated with Eliminate are preprocessed; their top-level declarations are removed, and their definitions are inlined at the different call-sites.[2] After preprocessing, instead of rewriting each function definition and callee as described in Section 5.4.2, we limit the code transformation to functions annotated with the Specialize attribute.

[2]In practice, we use a more efficient implementation strategy that allows us to perform our code rewriting in a single pass, and allows us to avoid traversing big terms that have already undergone a first round of inlining. The commented implementation of the tactic contains all of the details [328].

```
type mindex = { k : Type; v : Type }

[@Specialize] assume val eq (i : mindex): i.k -> i.k -> bool
[@Eliminate] let while_cond (b: pointer bool) (_:unit) = !*b

[@Eliminate]
let while_body (i: mindex) (b: pointer bool)
  (lsp: list (i.k * i.v)) (x:i.k) (_:unit) =
  ... (* elided, same as Listing 10 *)

[@Specialize] let find (i : mindex) (x : i.k)
  (ls : list (i.k * i.v)) : option i.v =
  let b = alloc true in let lsp = alloc ls in
  while (while_cond b) (while_body i b lsp x);
  match !* lsp with
  | [] -> None | (_, y) :: _ -> Some y

(* After rewriting *)
type mindex = { k : Type; v : Type }

let mk_find (i: mindex) (eq: i.k-> i.k -> bool)
            (x: i.k) (ls: list (i.k * i.v)): option i.v =
  let b = alloc true in let lsp = alloc ls in
  while (fun () -> !* b)
    (fun () -> let ls = !* lsp in
     match ls with
     | [] -> upd b false
     | (x', _) :: tl ->
       if eq x x' then upd b false
       else upd lsp tl);
  match !* lsp with
  | [] -> None
  | (_, y) :: _ -> Some y
```

Listing 11: Fine-Grained Control on Code Transformation

We show in Listing 11 a complete example using the different features presented in this section. The code on the right corresponds to the F* code on the left, after automatically rewriting find. As while_body and while_cond are annotated with Eliminate, they are inlined during preprocessing. The eq function declaration is annotated with the Specialize attribute; it therefore appears as an argument to mk_find. Other functions, i.e., alloc and upd take their roots in F*'s standard library, and are not annotated with any of our custom attributes. They are therefore ignored and left as-is while statically rewriting the call-graph. In real developments (see sections 5.5 and 5.6), annotating the functions proved to be extremely straightforward and light. In return, it allowed us to automatically transform the code into a higher-order version, which represents a fair amount of work when performed manually.

5.4.4 Implementation in Meta-F*

We have implemented this call-graph rewriting using syntax inspection, term generation and definition splicing in Meta-F* [218]. Meta-F* allows the programmer to script the F* compiler using *user-written F* programs*, a technique known as elaborator reflection and pioneered by Lean [329] and Idris [330]. This approach means that any fresh term generated by a meta-program must be re-checked for soundness; we therefore do not prove any results about our procedure and let F* validate the terms we produce. When calling our procedure, the user passes the roots of the call-graph traversal, i.e., the API endpoints of their library, along with the type of the index. The procedure traverses the call-graph, generates rewritten variants of all the definitions, and inserts them at the current program point.

We insist on the fact that the entire rewriting procedure was implemented in user-space and does not need to be trusted. One might wonder how we enforce that the types of the generated, higher-order definitions are correct. Indeed, if our meta-program generates definitions which don't have the correct type, successively type-checking those definitions against those types doesn't give us any guarantees. In practice, we check this later when *instantiating* those higher-order definitions, by annotating their specializations: if the types generated by our meta-program were incorrect, type checking would fail at this stage. As we use helpers to factor out types between the generic and the specialized definitions, annotating those instantiations doesn't create any burden on the user side. Finally, one last point of concern would be that our rewriting procedure transforms the functions in such a way that the generated C code has an unexpectedly poor performance. We note that, due to the nature of the transformations we perform, which consist, after instantiation and partial evaluation, in specializing part of code, this shouldn't happen in practice. Of course, this doesn't dispense us from benchmarking the code, which we do. As our technique gives users fine-grained control on the shape of the generated code, it is also possible to tune the output to reach the desired performance. In particular, we did not see any noticeable change of performance after adapting the code from HACL* to use the present technique (Section 5.5). The interested reader can find our entire, generously commented implementation online [331].

Comparison with existing techniques Our mechanism shares similarities with other type specialization techniques. Specifically, Haskell's SPECIALIZE pragma and Rust's trait system attempt to solve very similar problems, albeit as a trusted whole-program monomorphization pass within their respective compilers, as opposed to a source-to-source rewriting pass. Putting aside the problem of working within a proof assistant, we note that by contrast our technique is 1) untrusted and thus doesn't

require extending the F* compiler, 2) allows specializing over values, functions, while leveraging general-purpose dependent types, and 3) gives the user fine-grained control on how the specialized call-graph should look like, in particular for the purpose of outputting a readable program.

Adaptability of our technique to other proof assistants. While we implemented our approach in F⋆, our techniques are not tied to one particular language. Our work focuses on the verification of shallowly embedded programs; although a discussion of the advantages and disadvantages of the use of deep embeddings or shallow embeddings is out of scope of this chapter, it is worth noting that the extraction of shallowly embedded programs has been used in many other verification projects, relying on a variety of proof assistants [192, 238, 332–335]. Restricting our scope to the verification of shallowly embedded programs, our approach needs the following key ingredients to be applicable. (1) We need to be able to encode functors, and as we explained in Section 5.2 there exists a well known technique to do so in dependently-typed languages such as Coq, Lean or Idris. (2) We need elaborator reflection to implement the rewriting procedure; some languages like Lean or Idris provide it in their meta-language, while some other tools like Coq would require writing a plugin. (3) We need the ability to partially evaluate the specialized programs, which is a common feature of the aforementioned tools. (4) We need an extraction mechanism, which is supported for instance by Coq, Lean and Idris; in particular, we note that Lean and Idris support the extraction to a low-level language such as C or C++. We thus conclude that our methodology could be ported to either one of these proof assistants without fundamental difficulties.

5.5 Application to the HACL⋆ Cryptographic Library

We introduced our approach on a small example in the previous sections. We now demonstrate its applicability on real-world examples by presenting its use on heavily optimized implementations of cryptographic primitives inherited from the HACL⋆ [127, 233] and EverCrypt [128] libraries. HACL⋆ is a cryptographic library written in F⋆ which compiles to C; it offers vectorized versions of many algorithms via C compiler intrinsics, e.g., for targets that support AVX, AVX2 or ARM Neon. EverCrypt is a high-level API that multiplexes between HACL⋆ and Vale-Crypto [336, 337], a library of verified primitives implemented in assembly; it supports dynamic selection of algorithms and implementations based on the target CPU's feature set. Combined with EverCrypt, HACL⋆ features 105k lines of F* code for 72k lines of generated C code (excluding comments and whitespace, as well as the Vale assembly DSL). Those case studies

are not new, but were adapted to apply our technique. We explain in this section how we achieved this, and by doing so show how they stress all the requirements which motivated our new approach and which we described in the past sections; that is, the need for 1. zero-cost abstractions which provide high-level modularity and composability; 2. a fine-grained control on the shape of the generated code to obtain efficient and idiomatic implementations; 3. a flexible and lightweight approach which limits the amount of boilerplate and handles a wide range of scenarios. We detail in Section 5.7.1 the limitations of the previous techniques, and the consequent benefits we got by applying our new approach.

5.5.1 Hardware-Specialized Code: ChaCha20-Poly1305

We first present the application of our approach on one of our simplest examples, the ChaCha20-Poly1305 cryptographic construction. This case study illustrates how we used our approach to generate, from a single generic implementation, optimized code specialized for specific hardware targets. ChaCha20-Poly1305 is an algorithm for authenticated encryption with additional data (AEAD). The specifics of the construction are orthogonal to this chapter; for presentation purposes, it is sufficient to know that it combines two cryptographic primitives: the ChaCha20 stream cipher, and the Poly1305 message authentication code (MAC).

Depending on the hardware used, both ChaCha20 and Poly1305 admit several implementations. In particular, these primitives are especially well-suited to SIMD vectorization, by which we apply an operation (e.g., multiply by a constant) on all the elements of a vector at the same time, and can be highly optimized when such instructions are available. Previous work on HACL⋆ [233] demonstrated how to write and verify generic, vectorization-agnostic implementations of these algorithms, which could be specialized by partial evaluation to provide idiomatic C implementations. The approach then used to make the implementation generic was plagued with various issues, whose detailed description we defer until Section 5.7; in short, it struggled with scalability.

We now show how we implemented the cryptographic construction in our DSL. We mentioned earlier (Section 5.3.2) that the index captures the set of possible specializations. Our running example admitted an infinite set of possible specialization choices, as long as the key type admitted a decidable equality. In the example below, we only capture a *finite* set of possible specialization choices, which we express via a finite enumeration of type arch_index.

```
type arch_index = | V32 | V128 | V256
```

```
[@Specialize]
assume val chacha20_encrypt: w:arch_index -> chacha20_encrypt_st w

[@Specialize]
assume val do_poly1305: w:arch_index -> poly1305_st w

let aead_encrypt (w:arch_index) ... =
  chacha20_encrypt w ...;
  do_poly1305 w ...
```

To parameterize over both primitives, we rely on abstract signatures for ChaCha20 and Poly1305, as described in Section 5.4.1. The types chacha20_encrypt_st and poly1305_st correspond to the function types of both primitives, where the type of the arguments (e.g., the Poly1305 context) depend on the w: arch_index parameter. Both functions are annotated with the Specialize attribute, indicating that they are parameters of the implementation. As aead_encrypt calls these two functions, our rewriting procedures generates a higher-order combinator mk_aead_encrypt which requires two functions for chacha20_encrypt and do_poly1305. The aead_decrypt function is rewritten in a similar manner. The last step is to instantiate this combinator with different existing implementations for both primitives, for instance one specialized for 128-bit vectorization.

```
let aead_encrypt : aead_encrypt_st V128 =
  mk_aead_encrypt V128 do_poly1305_128 chacha20_encrypt_128
```

The resulting C code is idiomatic, and close to what one would expect from hand-written C code, albeit with formal guarantees about its correctness and constant-time execution. Case in point, the corresponding code in HACL* was previously integrated into Firefox [233].

5.5.2 Composing Implementations: Curve25519

We saw in the previous section a first application of the basic features of our approach. In this section, we demonstrate how our technique gives us *composability* on a real-world example, allowing us to simplify a collection of verified implementations of a widely used elliptic curve, Curve25519 [338].

The specifics of the algorithm are out of scope for this chapter; in this presentation, it suffices to say that it relies on modular arithmetic in a mathematical field, which admits two implementations based on different representations of the field elements.

Furthermore, one of these representations relies on a set of primitives (e.g., addition) which themselves admit two different implementations, one in Low*, and one in Vale assembly when specific hardware instructions are available.

Previous work on EverCrypt [128] provided a single verified client-facing API multiplexing between different implementation, that is, an API which selects the best implementation depending on the hardware available; these implementations however lived side by side, duplicating a lot of code. Using our approach, we now show how we reduce code redundancy, by aggressively sharing more code between those different implementations, and only specializing between the different field representations and implementations *a posteriori*. Providing a single generic implementation that will be automatically specialized reduces the maintenance cost of the HACL* codebase, while also simplifying the development of algorithmic improvements across our different versions. Using OCaml syntax, the end result allows users to pick between three different versions of Curve25519:

- module Curve64Lowstar = Curve25519(Field64(CoreLowstar))
- module Curve64Vale = Curve25519(Field64(CoreVale))
- module Curve51 = Curve25519(Field51)

An important point to notice is that we leverage our DSL to organize our implementation into three layers, that we later compose with each other. For presentation purposes, we present here a simplified version of Curve25519 which omits several layers and functions. We refer the interested reader to the HACL* repository [339] for our complete implementation.

Composing Abstractions. Curve25519 exposes several functionalities, including scalarmult, which performs scalar multiplication on the elliptic curve. This function calls into encode_point, which itself relies on the field addition fadd. All these functions are parameterized by an index corresponding to the field representation, of type field_index. For clarity of the generated code, we wish to avoid inlining any of these functions; we thus annotate each definition with the Specialize attribute.

```
type field_index = | F51 | F64

[@Specialize] assume val fadd (i:field_index) -> fadd_t i

[@Specialize] let encode_point (i: field_index) ... = ... fadd ...
[@Specialize] let scalarmult (i:field_index) ... = ... encode_point ...
```

The code is rewritten as one might expect. We can provide *multiple* specializations for one choice of index. If i is F64, we can generate both encode_point_64_lowstar and encode_point_64_vale:

```
let encode_point_64_lowstar = mk_encode_point F64 Lowstar.Field64.fadd ...
let encode_point_64_vale = mk_encode_point F64 Vale.Field64.fadd ...
```

In addition, we can generate encode_point_51 (elided). This in turns allows us to generate three versions for scalarmult:

```
let scalarmult_51 = mk_scalamult F51 encode_point_51 ...
let scalarmult_64_lowstar = mk_scalamult F64 encode_point_64_lowstar ...
let scalarmult_64_vale = mk_scalamult F64 encode_point_64_vale ...
```

In effect, leveraging the composability permitted by our approach, the Curve25519 implementation is organized into three layers: the field arithmetic, the field encoding (F51 or F64), and the elliptic curve operations.

5.5.3 A Highly Parametric Example: The HPKE Construction

We now present the culmination point of our series of cryptographic primitives: HPKE (Hybrid Public-Key Encryption) [268], a recent cryptographic construction that combines AEAD (Authenticated Encryption with Additional Data), DH (Diffie-Hellman), and hashing. The implementation of HPKE ticks several of the boxes that we wished to cover with our technique, that is: we build on top of several functionalities, each of these functionalities can be instantiated with several algorithms (e.g., Curve25519 or P256 for DH, ChaCha20-Poly1305 or AES-GCM for AEAD), and every algorithm admits several implementations; we have a complex call graph divided into several layers. Omitting several definitions for brevity, we structure the code as follows, using hpke_alg as our index.

```
type aead_alg = AES128_GCM | AES256_GCM | CHACHA20_POLY1305
type hpke_alg = dh_alg * aead_alg * hash_alg

type key_aead (alg: hpke_alg) = lbuffer U8.t (key_len (snd3 alg))

[@Specialize] assume val sign: (alg:hpke_alg) -> sign_t alg
[@Specialize] assume val enc: (alg:hpke_alg) -> enc_t alg

[@Eliminate]
let helper (alg: hpke_alg): helper_t alg =
  fun ...  -> ... sign alg ...
```

```
[@Specialize]
let hpke_sealBase (alg: hpke_alg): hpke_sealBase_t alg = fun ... ->
  ... helper alg ...
  ... enc alg ...
```

The index hpke_alg is a triple that captures all possible algorithm choices prescribed by the HPKE RFC. We thus write specifications, lemmas, helpers, and types parametrically over the index as standalone definitions. The key_aead type, for example, is parametric over triplets of algorithms, and defines a low-level key to be an array of bytes whose length is the key length for the chosen AEAD. The same systematic parameterization over hpke_alg can be carried to functions and their types, e.g., hpke_sealBase, which encrypts and authenticates a plaintext. We use small helpers, e.g., helper, to make verification robust in the presence of an SMT solver and ensure modularity of the proofs, as explained in Section 5.4.3; because we want to evaluate them away at extraction time, we mark them with the Eliminate attribute.

One possible specialization, out of hundreds of possible options, is to pick the F51 version Curve25519 for DH (Section 5.5.2), the AVX 128-bit variant of Chacha20-Poly1305 for AEAD (Section 5.5.1), and SHA2-256 for hashing.

```
let alg = (DH_CURVE25519, CHACHA20_POLY1305, SHA2_256)
let sealBase = mk_hpke_sealBase alg aead_encrypt_cp128 scalarmult_51 ...
```

The HPKE example is emblematic of our modularity pattern. It allows the programmer to author their verified code while thinking about the choice of *functionalities*; picking concrete implementations for each functionality and specializing the code accordingly is left to a later phase, and is entirely handled by our automated rewriting. All the user has to do is pick their particular choice of algorithms and implementations, and enjoy the resulting specialized HPKE.

Out of hundreds of possible choices, the HACL⋆ library provides 30 different variants of HPKE. Adding a new variant requires minimal effort; furthermore, with our methodology, each variant lives in its own separate file, which can then be compiled with exactly the right compiler options without any danger of miscompilation.

5.6 A Generic State Machine: the Streaming API

In the previous section, our technical contributions consisted of honing the proofs and restructuring the codebase of *pre-existing* algorithms, solving deep technical roadblocks in the process. In this section, we describe a novel case study that was enabled by

the present work. We first explain the nature of the problem; then show how our methodology came in judiciously and allowed us to structure our code to achieve maximum modularity.

We want to emphasize that this case study is an important contribution, *on its own*, for two reasons. First, it encompasses all the difficulties of carrying out large-scale verification of low-level code: the development is built on top of already complex implementations (i.e., the HACL⋆ hashes); it is divided into several modular layers, which must each be specialized in a myriad of ways; finally, unverified implementations of this code have historically caused critical bugs in high-profile software [300, 301], and this complexity pervades our invariants, which were subtle and difficult to get right. Second, and perhaps more importantly: the cryptographic community has some folk knowledge of what a block algorithm is; but as far as we know, this folk knowledge was never distilled into formal, precise language, like we do here.

5.6.1 Illustrating Streaming APIs with the Hash example

Many cryptographic algorithms offer identical or similar *functionalities*. For example, SHA2 [340], SHA3 [341], and Blake2 [342, 343] (in no-key mode) all implement the *hash* functionality, taking an input text to compute a resulting digest. As another example, HMAC [344], Poly1305 [345], GCM [346], and Blake2 implement the *message authentication code (MAC)* functionality, taking an input text and a key to compute a digest.

At a high level, these functionalities are simply black boxes with one or two inputs, and a single output. Taking HACL⋆'s SHA2-256 implementation as an example, this results in a natural, self-explanatory C API:

```
void sha2_256(uint8_t *input, uint32_t input_len, uint8_t *dst);
```

This "one-shot" API, however, places unrealistic expectations on clients of this library. For instance, the TLS protocol, widely-used to secure internet communications, computes repeated intermediary hashes of the handshake data transmitted so far. Using the one-shot API would be grossly inefficient, as it would require re-hashing the entire handshake data every single time. In other situations, merely hashing the concatenation of two non-contiguous arrays with this API requires a full copy into a contiguous array.

Cryptographic libraries thus need to provide a different API that allows clients to perform *incremental* hash computations. A natural candidate for this is the block API: all of the algorithms we mentioned above are block-based, meaning that, under the hood, they follow the state machine from Figure 5.1: after allocating an internal state

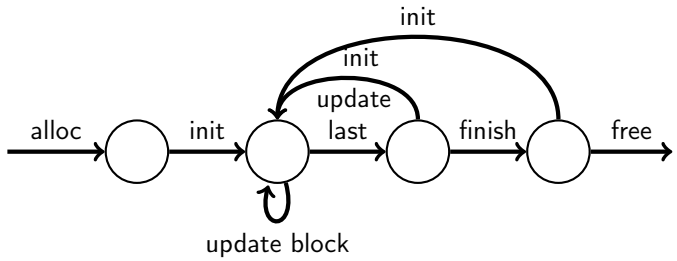

Figure 5.1: State Machine of an Error-Prone Block-Based API

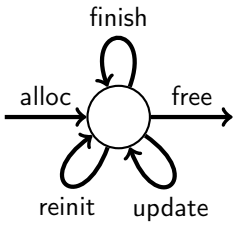

Figure 5.2: State Machine of a Safe, Streaming API

(alloc), they initialize it (init), process the data (update_block) block by block (for an algorithm-specific block size), perform some special treatment for the leftover data (update_last), then extract the internal state (finish) onto a user-provided destination buffer, which then holds the final digest. Revealing this API allows clients to feed their data into the hash *incrementally*, meaning that at first glance, our earlier issues are solved as we have found a way to hash data block by block without holding onto the entire input.

Unfortunately, this block API is wildly unsafe to call from unverified C code. First, it requires clients to maintain a block-sized buffer that, once full, must be emptied via a call to update_block. This entails non-trivial modulo-arithmetic computations and pointer manipulations, which are error-prone [300, 301]. Second, clients can easily violate the state machine. For instance, when extracting an intermediary hash, clients must remember to copy the internal hash state, call the sequence update_last and finish on the copy, free that copy, and only then resume feeding data into the original hash state. Third, algorithms exhibit subtle differences: for instance, Blake2 must not receive empty data for update_last, while SHA2 does not suffer from this restriction. In short, the block API is error-prone, confusing, and likely to lead to programmer mistakes.

We thus wish to take all of the block-based algorithms, and devise a way to wrap their respective block APIs into a uniform, safe API that eliminates all of the pitfalls above. We dub this safe API the streaming API (Figure 5.2): it has a degenerate state machine with a single state; it performs buffer management under the hood; it hides the differences between algorithms; and performs necessary copies as-needed when a digest needs to be extracted.

Writing and verifying a copy of the streaming API for each one of the eligible algorithms would be tedious, not very much fun, and bad proof engineering. Instead, we apply the methodology exposed throughout this chapter, and set out to write a generic API transformer that turns any block algorithm into its safe, streaming counterpart. We begin with a description of a block algorithm's stateful API and intended specification

```
type state_index = {
  s : Type; (* Low-level type *)
  t : Type; (* A pure representation of an s *)
  footprint : mem -> s -> Ghost loc;
  invariant : mem -> s -> Ghost Type;

  (* Reflect an s in a memory snapshot as a pure value *)
  v : mem -> s -> Ghost t;

  (* Adequate framing lemmas *)
  frame_invariant:
    ss:state_index -> l:loc -> s:ss.s -> h0:mem -> h1:mem ->
    Lemma
    (requires
      ss.invariant h0 s /\
      loc_disjoint l (ss.footprint h0 s) /\
      modifies l h0 h1)
    (ensures
      ss.invariant h1 s /\
      ss.v h0 s == ss.v h1 s /\
      ss.footprint h1 s == ss.footprint h0 s);

  ... (* Omitted: additional lemmas *) }

(* Stateful operations *)
[@Specialize] assume val malloc (#i: state_index) ... : ST ...
[@Specialize] assume val free (#i: state_index) ... : ST ...
[@Specialize] assume val copy (#i: state_index) ... : ST ...
... (* Omitted *)
```

Listing 12: The stateful API

using our DSL – this will be our "functor argument".

5.6.2 The Essence of Stateful Data

Before we get to the block API itself, we need to capture a more basic notion, that of an abstract piece of data that lives in memory, composes with the Low⋆ memory model and modifies-clause theory [347], and supports basic operations such as allocation, de-allocation, and copy. This is the *stateful* API presented in Listing 12.

We parameterize the implementation with the record state_index. We mentioned earlier that the index captures the space of all possible instantiations – here, this space is constrained by the presence of valid specifications that satisfy the behavioral lemmas we require. This is an extension of our previous style: the index bundles up in one record all of the type-level arguments to our functions. The specifications are used only in the proofs and are not relevant at runtime; for this reason we put them in the index and mark them as ghost by using the Ghost effect. Similarly to frameworks like

Why3 or Dafny, ghost code (and variables) in F⋆ is computationally irrelevant code; as such it must obey some restrictions, for instance, it is possible to convert a non-ghost value to a ghost value, but not the other way around. At extraction time, ghost code is erased, typically by being replaced with unit values (which are later eliminated). As the specifications are grouped in the index, they also do not undergo code specialization and higher-order rewriting, and do not need to be annotated with our DSL. This establishes a distinction between erased arguments (types, specifications, lemmas), which are handled via regular polymorphism and as such appear within the index, and run-time functions, which must undergo rewriting, higher-order parameterization, and as such rely on assume val and our rewriting mechanism.

Importantly, we saw in Section 5.5 the use of closed enumeration types for the choice of the index, by which we allow a *finite* set of possible specializations. In the present case, due to the highly generic nature of our code we need to use an open ended parameterization (i.e., a record), by which the index captures an *infinite* set of possible choices of specialization.

The state_index record contains a low-level type s (e.g., lbuffer U8.t 64ul, an array of length 64 containing bytes) which comes with an abstract footprint (e.g., the extent of that array in memory), and an abstract invariant (e.g., the array is live). The footprint and the invariant live in the Ghost effect, meaning they are computationally irrelevant and thus erased at extraction. The low-level type can be reflected as a pure value of type t (e.g., a sequence) using a ghost function v (e.g., as_seq, which interprets arrays as pure sequences). Outside of state_index, we declare some administrative lemmas which allow harmonious interaction with Low⋆'s modifies-clause theory; for instance, the frame_invariant lemma which we need because of the specificities of the Low⋆ memory model: under the pre-condition that the state invariant holds in an initial memory snapshot h0, and that the memory locations modified between h0 and h1 are disjoint from the state footprint, then the invariant also holds in h1 and the (pure reflection of the) state and the footprint are left unchanged; we automate its application with an SMT pattern (elided), which indicates to Z3 when to instantiate this lemma. The stateful operations allow, respectively, allocating a fresh state on the heap; freeing a heap-allocated state; and copying the state.

As we need two different stateful objects for our block implementation, states and keys (see 5.6.3), we actually declare two stateful APIs, in modules State and Key respectively; note that in practice we factor out the types of the declarations, so as not to duplicate code. Writing instances of the stateful APIs is easy, the most complex one being the internal state of Blake2 which occupies 46 lines of code, with all proofs going through automatically.

5.6.3 The Essence of Block Algorithms

We now capture the essence of a block algorithm by authoring an API that encapsulates a block algorithm's types, representations, specifications, lemmas, and stateful implementations in one go. We need the block API to capture four broad traits of a block algorithm, namely i) explain the runtime representation and spatial characteristics of the block algorithm, ii) specify as pure functions the transitions in the state machine, iii) reveal the block algorithm's central lemma, i.e., processing the input data block by block is the same as processing all of the data in one go, and iv) expose the low-level run-time functions that realize the transitions in the state machine. The result appears in Listing 13; for conciseness, we omit the full statement of the fold lemma, as well as the stateful type of the remaining transitions of the state machine. Similarly to the stateful API, we gather the specification of the block API in the index, that is in the record state_index. The actual definition is about 150 lines of F*, and appears in the anonymous supplement.

Run-time characteristics. A block algorithm revolves around its state, which implements the State stateful API. It may need to keep a key at run-time (km = Runtime, e.g., Poly1305), or keep a ghost key for specification purposes (km = Erased, e.g., keyed Blake2), or may need no key at all, in which case the key field is a degenerate instance of the Key stateful API, such that key.s = unit.

Specification. Using state.t, i.e., the algorithm's state reflected as pure value, we specify each transition of the state machine at lines 15-20. Importantly, rather than specify an "update block" function, we use an "update multi" function that can process multiple blocks at a time. We do not impose any constraints on how update_multi is authored, we only request the lemma update_multi_is_a_fold (line 23), which states that it obeys the fold law:

```
update_multi_s ((update_multi_s s l1 b1) (l1 + length b1) b2) ==
  update_multi_s s l1 (concat b1 b2)
```

This style has several advantages. First, this leaves the possibility for optimized algorithms that process multiple blocks at a time to provide their own update_multi function, rather than being forced to inefficiently process a single block. For non-optimized algorithms that are authored with a stateful update_block, we provide a higher-order combinator that derives an update_multi function and its correctness lemma automatically. Second, by abstracting over how blocks are processed, we capture a wide range of behaviors. For instance, Poly1305 has immutable internal state for

storing precomputations, along with an accumulator that changes with each call to update_block: we simply pick state.t to be a pair, where the fold only operates on the second component.

The block lemma. The spec_is_incremental lemma captures the key correctness condition and ties all of the specification functions together; by doing so it also specifies the order of the transitions of the state machine. For a given piece of data, the result hash1, obtained via the incremental state machine from Figure 5.1, is the same as calling the one-shot specification spec_s. This lemma relies on a helper, split_at_last, which splits a sequence into a series of blocks and a rest, and was carefully crafted to subsume the different behaviors between Blake2 and other block algorithms; in particular, it makes sure the rest is not empty unless the initial sequence is empty, so that update_last is never called on an empty sequence in the case of Blake2.

```
let split_at_last (block_len: U32.t) (b:seq U8.t) =
  let n = length b / block_len in
  let rem = length b % block_len in
  let n = if rem = 0 && n > 0 then n-1 else n in
  let blocks, rest = split b (n * l) in blocks, rest
```

Stateful implementations. We now zoom in on the update_multi low-level signature, which describes a block's algorithm run-time processing of multiple blocks in one go (Listing 13). This function is characterized by the spec-level update_multi_s; under the proper preconditions (elided here), it only affects the memory locations of the state s (line 37), preserves the footprint (line 38) and the invariant (line 39), and updates the state according to the pure spec (line 42).

The combination of the lemma spec_is_incremental along with the Low⋆ signatures of update_multi and others restricts the API in a way that the only valid usage is dictated by Figure 5.1. Designing this API while looking at a wide range of algorithms forced us to come up with a precise, yet general enough description of what a block algorithm is. We have been able to author instances of this API for SHA3, SHA2 (4 variants), Blake2 (4 variants), Poly1305 (3 variants), and legacy algorithms MD5 and SHA1. This includes the vectorized variants of these algorithms, when available. By materializing those instances, we were able to tie together a whole class of algorithms under a single unifying interface, therefore materializing the (informal) claim from the cryptographic community that "these are all blocks algorithms".

5.6.4 A Streaming API

Equipped with an accurate and precise description of what a block algorithm is, we are now ready to use our approach to write an API transformer that takes an instance of the block API, implementing the state machine from Figure 5.1, and returns the safe API from Figure 5.2. We now present the definition of the run-time state of streaming API. The state is naturally parameterized over a block_index, and wraps the block algorithm's state with several other fields.

```
[@CAbstractStruct]
type state_s (bi: block_index) = {
  block_state: bi.state.s;
  buf: buffer U8.t { length buf = bi.block_len };
  total_len: U64.t;
  seen: erased (seq U8.t);
  p_key: optional_key bi.km bi.key; }

let state (bi: block_index) = pointer (state_s bi)
```

The CAbstractStruct attribute ensures that the C code below will appear in the header. This pattern is known as "C abstract structs" and is commonly used by C programmers to provide a modicum of abstraction: the client cannot allocate structs or inspect private state, since the definition of the type is not known; it can only hold pointers to that state, which forces them to go through the API.

```
struct state_s;
typedef struct state_s *state;
```

First, buf is a block-sized internal buffer, which relieves the client of having to perform modulo computations and buffer management. Once the buffer is full, the streaming API calls the underlying block algorithm's update_multi function, which effectively folds the blocks into the block_state. The total_len field keeps track of how much data has been fed so far, information that is needed for many block-based algorithms, notably hashes which encode the length of the input as part of the final block in order to rule out padding attacks.

The most subtle point is the use of a ghost (i.e., computationally irrelevant and erased at extraction time) sequence of bytes, seen, which keeps track of the past, i.e., the bytes we have fed so far into the hash. This is reflected in the invariant, which states that if we split the input data into blocks, then the current block algorithm state is the result of accumulating all the blocks into the block state; the rest of the data that doesn't form a full block is stored in buf.

```
let state_invariant (bi: block_index) (h:mem) (s:state bi) =
  let s = deref h s in
  let State block_state buffer total_len seen key = s in
  let blocks, rest = split_at_last (U32.v bi.block_len) seen in
  (* omitted *) ... /\
  bi.state.v h block_state ==
    bi.update_multi_s (bi.init_s (optional_reveal h key)) 0 blocks /\
  slice (as_seq h buffer) 0 (length rest) == rest
```

The finish function takes a block specification bi. Under the hood, it calls State.copy to avoid invalidating the block_state; then update_last followed by finish, the last two transitions of Figure 5.1. Thanks to the correctness lemmas in the block API along with the invariant, finish states that the digest written in dst is the result of applying the full block algorithm to the data that was fed into the streaming state so far.

```
[@Specialize]
val finish (bi:block_index):
  s:state bi ->
  dst:buffer U8.t{len dst == bi.out_len} ->
  ST unit
  (requires
    fun h0 -> ... (* omitted *))
  (ensures
    fun h0 s' h1 ->
    ... /\ as_seq h1 dst == bi.spec_s (get_key h0 s) (get_seen h0 s))
```

One point of interest is the usage of a *ghost selector* get_seen, which in any heap returns the bytes seen so far. We have found this style the easiest to work with, as opposed to a previous iteration of our design where the user was required to materialize the previously-seen bytes as a ghost argument to the stateful functions, such as finish above. The previous iteration placed a heavy burden on clients, who were required to perform some syntactically heavy book-keeping to thread this argument through function calls; the present style is much more lightweight.[3]

This streaming API has one limitation, in that we cannot prove the absence of memory leaks. This is a fundamental limitation of using the ST effect in Low⋆. However, this can be easily addressed with manual code review or off-the-shelf tools, such as clang's –fsanitize=memory.

[3]The anonymous supplement contains an in-depth explanation of the respective merits of the three styles we considered, in file Hacl.Streaming.Functor.fsti.

A Note on Properly Compiling the State Type An interesting technicality is that the state type, as introduced above, generates runtime casts due to the Letouzey-style extraction pipeline of F⋆, and as such, does not compile to C. Casts between values of different types and sizes are admissible when extracting to OCaml, owing to its universal boxed value representation (as long as one is willing to use Obj.magic). But C has no ⊤ type, meaning that such casts are rejected by KaRaMeL.

Looking closely at state_s, we remark that it is parameterized by a value, not a type. It therefore won't extract to a definition of the form type αt. Second, it uses a type-level field projection for block_state, which is also not part of the simple grammar of types of either OCaml or C.

We *do* instantiate state_s over a specific choice of argument bi. But inductive types are typed nominally, and an application of state_s to its argument generates a type instantiation, not a fresh, specialized state type definition. This is in contrast to a type abbreviation, which, being typed structurally, would simply reduce away, circumventing this issue.

We could rewrite this type too, using our tactic, but there is actually a simpler way. We add a seemingly useless type (*not* value!) parameter to state_s:.

```
[@CAbstractStruct]
type state_s' (bi: Block.block_index)
              (s: Type { s == bi.state.s }) = {
  block_state: s;
  ... (* rest as before *) }

let state_s bi = state_s' bi bi.state.s
```

From the point of view of type-checking, this is strictly equivalent to the previous definition. But from the point of view of extraction, after erasure, bi becomes an unused, erased type parameter of state_s' (it eventually gets eliminated), while s, at Type, becomes a regular parameter of the (extracted) data type state_s'. Uses of state_s', via the state_s wrapper, become regular type applications. This means that the resulting code contains no casts, and simply relies on parameterized data types, which are handled by KaRaMeL and monomorphized via a whole-program compilation pass.

This rewriting trick significantly improves the quality of the generated code, and to the best of our knowledge, had never been documented before.

A Note on Additional Compile-Time Parameters In addition to types and lemmas, we also add, within our index, extra parameters that reduce at compile-time using normal reduction mechanisms. These act as supplemental "tweaking knobs" that

control the shape of the produced code. An example is the block size, which is specific to each algorithm, reduces using normal partial evaluation mechanisms, and eventually generates stack-allocated arrays of the correct block size (rather than with a run-time dynamic check).

Another one of these knobs is the key management policy, which is another choice the user can tweak when instantiating the streaming API.

```
type key_management = | Runtime | Erased

let optional_key (km: key_management) (key: Key.state_index) : Type =
  match km with | Runtime -> key.s | Erased -> Ghost.erased key.t
```

The km parameter of the block API exists only at compile-time, not at run-time. All of its uses are partially evaluated away. It allows the block algorithm to indicate whether it needs a key. In the streaming code, every reference to key goes through a wrapper like the one above. After partial evaluation, the optional_∗ wrappers reduce to either a proper key type, or to a ghost value, which then gets erased to unit. This allows, for instance, generating either an init function that does *not* take a key (hash functionality), or an init function that *does* take a key (MAC functionality). Thanks to the various unit-elimination optimizations of KaRaMeL, the former case results in no superfluous fields in the state type, nor superfluous arguments to the API functions.

5.7 Evaluation

We now evaluate the efficiency of our approach. Recall that our original goal was to support authoring large-scale, low-level verified software; in this section, we therefore focus on proof engineering and programmer productivity metrics. Our case studies involve pre-existing algorithms from the HACL⋆ project; the run-time performance of the code is thus that of the underlying cryptographic algorithms, for which we did not observe noticeable changes in performance after we updated the code. We therefore leave a crypto-oriented performance discussion to the original HACL⋆ paper [233]. In total, the modifications we performed had an impact on 30k lines of the C code generated by compiling the HACL⋆ library.

5.7.1 Core Algorithms: ChaCha20-Poly1305, Curve25519, HPKE

Qualitative Study The HACL⋆ library originally featured ChaCha20-Poly1305 and Curve25519, in multiple variants, but got by without the use of our code rewriting tactic. For Curve25519, the original code was playing build system tricks, and would tweak the

include path to select, say, one implementation of the Curve field over another. Needless to say, this did not scale. Every tweak to the include path invalidates intermediary build files, with two consequences: first, the build time rapidly skyrockets; second, the limitation carries over to verified clients of HACL⋆ which in turn need to play the same include path tricks if they want to use such algorithms.

In the case of ChaCha20-Poly1305, the existing code was in better shape, but not by much. It relied on a static dispatch style (not described here), which came with severe limitations. Notably, it imposed that all variants of the same algorithm be in one C file. This made regular C and vectorized implementations appear in the same file; as the vectorized version would mandate special compiler flags (here, −mavx −mavx2), the C compiler would happily use AVX2 instructions for the non-vectorized, regular C version, causing illegal instruction errors later on [348].

We upgraded both of these algorithms within the HACL⋆ codebase to use our code-rewriting tactic, which addressed all of the roadblocks above, and resulted in significantly improved programmer experience and productivity. Our techniques also paved the way for the HPKE implementation in HACL⋆. Before our work, HACL⋆ could not distinguish between a notion of *algorithm* (e.g., P-256 *vs.* Curve25519) and multiple *implementations* (e.g., Curve25519-64 *vs.* Curve25519-51) of said algorithm. This made a modular and specializable HPKE impossible to author. Using our framework, the HACL⋆ authors were able to express HPKE naturally, modularly and generically, while allowing more than 60 possible choices of algorithms and corresponding implementations, each in their own file. This simply could not have happened without the principles exposed in this article.

Quantitative Study In the design of elaborator reflection, the user (i.e., the tactic) is allowed to create ill-typed terms. The API does not statically enforce the creation of well-typed terms; it simply re-checks user-provided terms before they are added to the context. This means that the rewritten terms produced by our tactic need to be re-checked by the F⋆ typechecker.

We measure the verification overhead that comes from re-verifying those rewritten terms. Specifically, Table 5.3 measures the overhead incurred by re-checking the tactic-generated definitions, relative to the total verification time for a given algorithm. In most cases, the overhead is $< 100\%$, because we don't rewrite lemmas and proofs. We need to investigate why HPKE is an outlier; we suspect the Z3 solver might be overly sensitive to the shape of the proof obligations it receives; since we rewrite the call-graph, the resulting proof obligations are slightly different from the ones generated by the original call-graph.

One might wonder about the impact of our approach on programmer productivity. Indeed, re-verifying the terms has a non-negligible impact on the build time which, in turn, affects programmer productivity. In practice this did not prove to be an issue: we generally need fast incremental builds (and in particular, fast type-checking of the code) when working on the *generic* definitions and their proofs (i.e., the functions implemented in the DSL), or when working on the clients of the specialized instantiations *after* we run the call-graph rewriting and verified the result.

5.7.2 Implementation and Usability of our DSL

Tactics are not part of the trusted computing base (Section 5.2); unlike, say, MTac2 [349], Meta-F⋆ [218] does not allow the user to prove properties about tactics, trading provable correctness for ease-of-use and programmer productivity. This begs the question of the reliability of the tactic, since it's not formally shown to always generate well-typed terms. Debugging took place in two phases. First, type-checking the implementation of the tactic itself, which was easy, as there were no deep proof obligations, only ML-like type-checking. We note that our tactic, at 620 lines, (including whitespace and comments) is the third largest Meta-F⋆ program written to date. Second, type-checking the output of the tactic. We did so by inspecting the generated definitions and type-checking them like regular terms in the interactive mode, which quickly revealed the source of bugs. We debugged the tactic on Curve25519, our most complex example; once debugged, the tactic never generated ill-typed code and was used successfully by other co-authors.

In the years since we implemented this tactic, it has come to be used in numerous places in HACL⋆ and has been the workhorse of many verified algorithms. The tactic now executes natively, leveraging the F⋆ compiler's ability to dynlink natively-compiled tactics, similar to Coq's native_compute. The running-time of the tactic itself is not noticeable.

5.7.3 Streaming API

To evaluate the applicability of the streaming API, we compare lines of code (LoC) for the F⋆ source code and the final C code as a proxy for programmer effort. While not ideal, this metric has been used by several other papers [127, 128, 233] and provides a coarse estimate of the proof engineering effort. Our point of reference is a previous, non-generic streaming API that previously operated atop the EverCrypt agile hash layer.

Table 5.4 presents the evaluation. For the old streaming API, the proof-to-code ratio was 1.11, i.e., each line of generated C code required more than one line of

F* code. Capturing the block API and implementing the streaming API uses 1667 lines of F* code. The extra verification effort is quickly amortized across the 14 applications of the streaming API, which each requires a modest amount of proofs to match the exact signature of the block API. Out of those, six were integrated into the reference implementation of the Python programming language. Poly1305 and Blake2 were originally authored without bringing out the functional, fold-like nature of the algorithms, which led to some glue code and proofs to meet the block API. Altogether, we obtain a proof-to-code ratio of 0.51, which we interpret to coarsely mean a 2x improvement in programmer productivity. We expect this number to further decrease, as more applications of the streaming API follow.

For execution times, we present the verification time of the API itself, and the verification time of each of the instances, including glue proofs. Compared to fully verifying Blake2 (7.5 minutes), or Poly1305 (~14 minutes), the verification cost is modest. Applying the streaming API to a type class argument incurs no verification cost, so the extraction column measures the cost of partial evaluation and extraction to the ML AST, which is negligible.

5.8 Related Work

Automating the generation of low-level code is a common theme among several software verification projects. We now review several related efforts not discussed earlier in the chapter.

A rich overview of the topic of proof engineering can be found in Ringer et al.'s survey [224]. We however note that this survey focuses on techniques for verifying large-scale *proof* developments without efficient, readable extraction being a concern. Furthermore, it especially focuses on Interactive Theorem Provers, and leaves out of scope program verifiers based on constraint solvers, which require a different set of techniques to tame the solver's complexity. In this regard, the present work explores a complementary facet of the art of proof engineering.

Fiat Cryptography [192] relies on a combination of partial evaluation and certified compilation phases to compile a generic description of a bignum arithmetic routine to an efficient, low-level imperative language, which is then output as either C or assembly. In this approach, the specifications are declarative, and do not impose any choice of representation. Conversely, in HACL*, the decision is made by the programmer, who *manually* refines a high-level mathematical specification into an implementation that picks word sizes and representations. While the approach of Fiat Cryptography is automated, it relies on fine-grained control of the compilation toolchain; for instance,

a key compilation step is bounds inference, which picks integer widths to be used by the rest of the compilation phases. By contrast, we do not customize the extraction procedure of F⋆, nor extend KaRaMeL with dedicated phases. Another difference to highlight is that FiatCrypto, to the best of our knowledge, focuses on the core bignum subset of operations, and offers neither a high-level Curve25519 API, or other families of algorithms. In our work, we operate at higher levels up the stack, tackling high-level API transformers and complete algorithms, "in the large".

Jasmin [350, 351] is a framework for developing high-speed, verified cryptographic implementations. Jasmin provides a low-level DSL with features such as loops or procedures, and has been used to verify a range of cryptographic algorithms. However it lacks the higher-level abstraction features provided by our approach to author generic, specializable implementations. Jasmin relies on verified compilation using Coq to generate optimized assembly code semantically equivalent to code verified in the Jasmin DSL. In contrast, the extraction procedure of F⋆, in line with several other proof assistants, is trusted; this problem is orthogonal to our approach, and could be addressed through advances in verified program extraction [323, 352].

Recent work by Pit-Claudel et al. [333, 334] proposes correct-by-construction pipelines to generate efficient low-level implementations from non-deterministic functional high-level specifications. The process is end-to-end verified as it relies on Bedrock [353, 354]. In Pit-Claudel's work, compilation and extraction are framed as a backwards proof search and synthesis goal. Handling non-determinism has not been done at scale with F⋆; however, the algorithms we study are fully deterministic. Pit-Claudel's approach is DSL-centric: the user is expected to augment the compiler with new synthesis rules for each new flavor of specifications. In our work, we reuse the existing extraction facility of F⋆, which we treat as a black box. We rely on many whole-program compilation phases, such as the various compilation schemes for data types, monomorphization and unused argument elimination; to the best of our knowledge, Pit-Claudel's toolchains do not support such whole-program transformations that require non-local decisions.

Appel et al. [355] verify the equivalent of our streaming API applied to SHA2-256; specifically, the version found within OpenSSL. This work is end-to-end verified, by virtue of using VST [356]. In its current form, the development is geared towards SHA-256 only, and supports neither higher-order, modular reasoning, nor code generation "for free" for multiple algorithms.

Lammich [335] uses an approach very similar to Pit-Claudel *et.al.*, and refines Isabelle/HOL specifications down to efficient, Imperative/HOL code. The code is then extracted to a functional programming language, e.g. OCaml or SML, and compiled

by an off-the-shelf compiler. This means that unlike Pit-Claudel, Lammich still relies on a built-in extraction facility. This is a promising approach, and seems applicable to the original HACL⋆ code: it would be worthwhile to try to refine HACL⋆ specifications automatically to the Low⋆ code. In the case of the various "functors" we describe, however, we suspect "explaining" how to refine the specification into the exact API we want would require the same amount of work as writing the functors directly.

In Haskell, the cryptonite library [357] offers an abstract hash interface using type classes, along with several instances of this type class. The high-level idea is the same: unifying various hash algorithms under a single interface. One natural advantage of our work is that it comes with proofs, meaning that clients can be verified on top of HACL⋆ and shown to not misuse our APIs, before being also extracted to C. Perhaps more to the point, our approach also has unique constraints: no matter how efficient functional code may be, we insist on generating idiomatic C code: adoption by Firefox, Linux and others comes at that cost. The code produced by our toolchain cannot afford to have run-time dictionaries or function pointers. We therefore *must* partially evaluate away the abstractions before C code generation.

Cogent [240] is a purely functional language with linear types that was used to implement verified file systems [241]. By restricting the language's expressiveness, its authors simplify reasoning, and allow compiling Cogent programs to efficient C code by means of a self-certifying compiler. Though Cogent provides features such as polymorphic, higher-order functions, as well as the possibility of parameterizing code with abstract definitions, it doesn't provide the equivalent of 0-cost functors like our approach. Our approach seems a natural fit to verify applications such as file systems, provided the effect system we use is adequate. In this regard, it might be interesting to use our rewriting mechanism with programs written in Steel [236], a separation logic framework implemented for F⋆ and which also supports extraction to C through KaRaMeL; doing so would only require minor modifications of our rewriting procedure.

5.9 Conclusion

Software verification is entering new territory, with proof developments now routinely topping 100,000 lines of code. And when it comes to verifying security-critical code, the resulting software artifact has to be not only verified, but also low-level and fast. For those projects, there is a growing need of foundational proof development design patterns.

In this chapter, we designed, implemented and evaluated a new methodology that relies on elaborator reflection to add a custom pre-compilation stage. That early stage

interprets user-provided annotations (in effect, a DSL) and rewrites the code accordingly. We showed that this provides significant gains in terms of proof engineer productivity, allowing not only existing algorithms in HACL* to be rewritten in a form that tames previous complexity, but also allows us to explore and analyze new algorithms, such as the streaming API.

The benefits of our approach are very concrete: we were able to implement, verify, and instantiate the streaming API in a modular way, which resulted in code that was high-quality enough to pass muster with the Python maintainers, and should be included in the upcoming Python 3.12.

```
type block_index = {
  km: key_management; (* km = Runtime \/ km = Erased *)
  state: State.state_index; (* State spec *)
  key: Key.state_index; (* Key spec *)
  (* Some lengths used in the specs *)
  max_input: x:nat { 0 < x /\ x < pow2 64 };
  out_len: x:U32.t { U32.v x > 0 };
  block_len: x:U32.t { U32.v x > 0 };

  (* The one-shot specification *)
  spec_s: key.t -> input:seq U8.t{length input <= max_input}) ->
    out:seq U8.t{length out == U32.v out_len};

  (* The block specification *)
  init_s: key.t -> state.t;
  update_multi_s: state.t -> prevlen:nat ->
    s:seq U8.t{length s % U32.v block_len = 0}) -> state.t;
  update_last_s: state.t -> prevlen:nat ->
    s:seq U8.t{length s <= U32.v block_len} -> state.t;
  finish_s: key.t -> state.t -> s:seq U8.t { length s = U32.v out_len };

  (* update_multi_s respects the fold law... *)
  update_multi_is_a_fold: ... -> Lemma ...;
  (* ...and central correctness lemma of a block algorithm *)
  spec_is_incremental: key:key.t ->
    input:seq U8.t{length input <= max_input} -> Lemma (
      let bs, l = split_at_last (U32.v block_len) input in
      let hash0 = update_multi_s (init_s key) 0 bs in
      let hash1 = finish_s key (update_last_s hash0 (length bs) l) in
      hash1 == spec_s key input); }

[@Specialize]
assume val update_multi (bi:block_index) (s:bi.state.s) (prevlen:U64.t)
  (blocks:buffer U8.t { length blocks % U32.v bi.block_len = 0 })
  (len: U32.t { U32.v len = length blocks /\ ... (* omitted *) })
  ST unit (requires ... (* omitted *)) (ensures (fun h0 _ h1 ->
    modifies (bi.state.footprint h0 s) h0 h1 /\
    bi.state.footprint h0 s == bi.state.footprint h1 s /\
    bi.state.invariant h1 s /\
    bi.state.v i h1 s ==
      bi.update_multi_s (bi.state.v i h0 s) (U64.v prevlen)
                        (as_seq h0 blocks) /\ ...));
... (* Omitted: rest of the block API, e.g. init, finish... *)
```

Listing 13: The block API

Figure 5.3: Cost of Verifying the Tactic-Rewritten Call Graphs

Algorithm	**Verif. time**
Chacha20	6s (+27%)
Poly1305	43s (+17%)
ChaCha-Poly	19s (+34%)
Curve25519	74s (+88%)
HPKE	231s (+132%)

Figure 5.4: Quantifying the Impact of the Streaming API

	F⋆ LoC	**C LoC**	**verif.**	**extract.**
EverCrypt hashing (old)	848	798		
API and interfaces	1667	0	103.5s	0
EverCrypt hashing (new)	128	929	20.3s	8.3s
MD5, SHA1, SHA2 (×4)	231	1577	39.4s	13.6s
Poly1305 (×3)	452	998	55.7s	6.4s
Blake2s (×2), Blake2b (×2)	874	4761	115.1s	9.2s
Total	4200	8265	334s	47.6s

Chapter 6

Incremental Proofs in Dafny with Module-Based Induction

The Noise* and zero-cost functors projects, by being implemented in F*, crucially rely on SMT automation. If this class of automation proved powerful to handle many reasonings and in particular arithmetic reasonings, it also suffers from several issues such as proof instabilities. The ability to use SMT solvers for different classes of programs, like compilers, is also unclear. In this third project, we tackle the problem of verifying parts of the Dafny compiler inside of Dafny itself, and design novel techniques to improve the proof stability by relying on Dafny's modules to implement induction principles.

6.1 Introduction

Writing a mechanized proof generally implies working in an incremental manner, by first laying out a simplified version of the problem under study and shaping the proofs, then gradually complexifying the definitions while updating those proofs. For instance, to verify a compiler one might start with a simplified version of the source AST that omits the complex cases, before adding those cases one by one. Highly automated theorem provers like Dafny make the first iterations pleasant as the prover manages to discharge most of the then-simple proof obligations. Unfortunately, as the problem gets more complicated, automated proofs can become unstable or simply break. Worse, most automated theorem provers only offer limited tools to debug unstable or failing proofs: unlike an ITP like Coq, an ATP like Dafny does not show a goal or a proof context along with each failure. In an attempt to recover the benefits of ITPs, we demonstrate in this short chapter how to structure inductive proofs in Dafny by using modules to encode Coq-like induction principles. We report on our experience of iterating through

```
1  // Dafny
2  datatype LL<T> =
3  | Nil | Cons(h: T, t: LL<T>)
4
5  function App<T>(
6    l0: LL<T>, l1: LL<T>) : LL<T>
7  { match l0
8    case Nil => l1
9    case Cons(h, t) =>
10   Cons(h, App(t, l1)) }
11
12 lemma Assoc<T>(l0: LL<T>,
13   l1: LL<T>, l2: LL<T>
14 ) ensures App(App(l0, l1), l2)
15       == App(l0, App(l1, l2))
16 { match l0
17   case Nil => // Nothing to do
18   case Cons(h, t) =>
19   Assoc(t, l1, l2); }
```

```
(* Coq *)
Inductive ll T :=
| Nil | Cons : T -> ll T -> ll T.

Fixpoint app {T} (l0 l1: ll T) : ll T :=
  match l0 with
  | Nil => l1
  | Cons h t => Cons h (app t l1)
  end.


Lemma assoc {T} (l0 l1 l2 : ll T) :
  app (app l0 l1) l2 = app l0 (app l1 l2).
Proof.
  induction l0 as [|hd tl IH]; intros; simpl.
  + reflexivity. (* Nil case *)
  + rewrite IH. reflexivity. (*Cons case *)
Qed.
```

Listing 14: Proof that List Concatenation is Associative (Left: Dafny, Right: Coq).

the proofs of a prototype of a self-hosted compiler for the Dafny language, and on the benefits in terms of proof maintainability and stability.

Contributions. The work in this chapter comes from a collaboration with Clément Pit-Claudel, that we presented at the Dafny Workshop in 2014 [358]. With regards to the self-hosted compiler which is mentioned throughout the chapter [359], Clément wrote a first version of the semantics of (a subset of) Dafny, which I then expanded and used to implement and verify several compilation passes. During this process I came up with the idea of using modules to encode inductive proofs in Dafny.

6.2 Inductive Proofs in Dafny and Coq

Comparing Dafny and Coq. Let us start with a minimal example: a proof that list concatenation is associative, in Dafny and Coq (listing 14). In Dafny, we define the concatenation as App ("append") and do the proof by induction in Assoc. Dafny automatically discharges the Nil case. In the Cons case, we use a recursive call (line 19) to invoke the induction hypothesis.

In Coq, we use induction l0 as ... to invoke the induction principle that Coq automatically derived from the definition of ll, which, after a call to simpl to simplify the context, gives us two goals:

$\texttt{T} : \texttt{Type}, l_1 : \texttt{ll T}, l_2 : \texttt{ll T} \vdash \texttt{app}\ l_1\ l_2 = \texttt{app}\ l_1\ l_2$

for the Nil case, and:

$\ldots$, IH : $\forall l_1\ l_2.\ \texttt{app}\ (\texttt{app}\ t\ l_1)\ l_2 = \texttt{app}\ t\ (\texttt{app}\ l_1\ l_2)$
$\vdash \texttt{Cons}\ h\ (\texttt{app}\ (\texttt{app}\ t\ l_1)\ l_2) = \texttt{Cons}\ h\ (\texttt{app}\ t\ (\texttt{app}\ l_1\ l_2))$
for the Cons case.

From there, it is easy to determine how to invoke the induction hypothesis (rewrite IH, with unification filling in the arguments) in the Cons case. We finally conclude both goals by reflexivity. In contrast, in Dafny: 1. we have to write the inductive structure by hand; 2. we are not shown goals; 3. we cannot use unification to instantiate the induction hypothesis, and must instead specify arguments to the recursive call. This isn't an issue when doing simple proofs, but is a significant burden when working on more realistic cases. In particular, Dafny doesn't provide much information to the user when a proof breaks, while Coq displays the precise goal on which it got stuck. As a result, the user often spends a significant amount of time debugging broken proofs to understand *which* proof obligation failed, before actually spending time on fixing it.

Another issue is proof duplication. We notice that in practice most proofs about lists follow the same structure: we perform an induction, and in the Cons case call the induction hypothesis on the tail of the list; writing the inductive structure and specifying the arguments to the recursive call by hand leads to a lot of boilerplate in Dafny.

One final issue is proof evolution. In Coq, adding an additional constructor, maybe `Snoc : ll T → T → ll T`, would lead to a failed proof with a new unsolved goal clearly identifying what is missing. Dafny, in contrast, would first try to derive a contradiction in the missing case (Snoc), and having failed to do that would report the missing case (without showing a proof context). This contradiction proof can be costly: in extreme cases, e.g. with many other constructors, it can fail to complete and we may simply get an unspecific proof failure for the whole lemma, without further details. A solution might be to forbid the users from omitting cases in a match; this is however not desirable in practice, as programmers often use this convenience to avoid considering many irrelevant cases in their proofs.

Using an induction principle in Dafny. We propose to structure the proofs with an induction principle, by which we factor out the structure of the proofs and decompose the various proof obligations, leading to less mundane work, a better debugging experience and finer control over verification times. The use of induction principles is inspired by provers like Coq which generate them for free; in the case of Dafny we have to write them by hand. [1]

[1] The induction principle we introduce here is actually simplistic for the purpose of clarity. For more realistic versions, see Section 6.3.

```
abstract module ListInduction {
  predicate P<T>(ls: LL<T>)

  lemma Induct_Nil<T>() ensures P<T>(Nil)

  lemma Induct_Cons<T>(h: T, t: LL<T>)
    requires P(t)
    ensures P(Cons(h, t))

  lemma Induct<T>(ls: LL<T>) ensures P(ls) {
    match ls
    case Nil => Induct_Nil<T>();
    case Cons(h, t) => Induct(t); Induct_Cons(h, t); } }

module AppAssoc refines ListInduction {
  predicate P ... { forall l1, l2 :: App(App(ls, l1), l2) == App(ls, App(l1, l2)) }

  // "..." is Dafny syntax to reuse the
  // signatures defined in the module
  // ListInduction
  lemma Induct_Nil ... {}
  lemma Induct_Cons ... {} }
```

Listing 15: Proof of Associativity of List Concatenation with a Module-Based Induction Principle.

We proceed by defining an induction principle for lists in the form of an abstract module (ListInduction in listing 15). We declare the target property that we wish to prove by induction through an abstract predicate P, together with the rules it must satisfy: Induct_Nil states that P must be true for the empty list, and Induct_Cons states that it must be true on non-empty lists provided it its true on their tails. Those abstract declarations act like holes: the corresponding proofs are to be filled later. Given those assumptions, we can prove by induction once and for all that P always holds (lemma Induct).

This abstract module provides a generic structure for all the inductive proofs for lists; in particular we can use it to prove associativity. We do so by defining a module named AppAssoc which refines ListInduction. This time, we have to fill in the blanks: we state the associativity property by providing a definition for P, and write proofs for Induct_Nil and Induct_Cons; as Dafny manages to discharge the proofs automatically, they are empty. Note that because the proofs are empty, Dafny actually allows us to omit those lemmas, which is in practice very useful when there are a lot of trivial cases; Dafny would however report an error if it fails to prove on its own a lemma that we omitted. The theorem we want is finally given by AppAssoc.Induct, that we can use without additional work. Using an induction principle for this simple example might seem overkill; we illustrate the benefits on more realistic examples in the next sections.

6.3 Applying the Induction Principle on Mini-Dafny

6.3.1 Verifying IsPure

We explained how to define and use an induction principle on the simplistic example of list concatenation. Let us now illustrate how it can be adapted to a more interesting example, namely verifying micro-passes of a compiler for a simple language based on Dafny that we call mini-Dafny. We make the whole development available in the companion artifact [360]. We adapted this language from a work-in-progress verified compiler for the Dafny programming language [359]. The problem of verifying multiple compilation passes, which involved repeatedly proving inductive theorems involving the same, big function (InterpStmt is around 1000 LoCs in [359]) provided the initial motivation for the present work.

```
1 function InterpStmt(s: Stmt, ctx: Context):
2   Result<(int, Context)> {
3   match s {
4   case Bind(bvar, bval, body) =>
5     // ':-' below is a monadic bind
6     var (bvalv, ctx1) :- InterpStmt(bval, ctx);
7     var ctx2 := ctx1[bvar := bvalv];
8     var (bodyv, ctx3) :- InterpStmt(body, ctx2);
9     var ctx4 := ctx1 + (ctx3 - {bvar});
10    Success((bodyv, ctx4))
11
12  case Seq(s1, s2) =>
13    var (_, ctx1) :- InterpStmt(s1, ctx);
14    InterpStmt(s2, ctx1)
15
16  ... /* Omitted */ } }
```

We define the semantics of mini-Dafny with an interpreter (InterpStmt). For simplicity, values are integers, and we omit all side effects but in-place updates to local variables. InterpStmt takes as inputs a statement and a context, which is a map from variable names to integer values, and returns the result of evaluating the statement together with an updated context. In order to evaluate a variable declaration (Bind case), we first evaluate the bound value bval (line 6), where :− is a bind for the error monad, meaning the statement var x :− y; st is desugared to match y { case Fail e => Fail e; case Return(x) => st; }. We then augment the context with a new binding for bvar (line 7), evaluate the body in this new context (line 8), and finally reset the value bound to bvar (if this binding exists in the initial context),

so that the bound variable doesn't escape its scope (line 9). Specifically, at line 9, ctx3 − {bvar} is the map ctx3 where we remove the binding for bvar (if it exists), and ctx1 + (ctx3 − {bvar}) is ctx1 extended with the bindings from ctx3 − {bvar} (if a binding exists in both maps, we take the one from ctx3 − {bvar}). Finally, evaluating a sequence of statements (Seq case) simply requires chaining contexts between the statements of the sequence.

Given those semantics for mini-Dafny, we can verify a first micro-pass which rewrites statements of the form 0 ∗ s or s ∗ 0 to 0, provided s is pure (i.e., it doesn't update any local variable); note that in mini-Dafny we mix statements and expressions. We first define a predicate IsPure(s: Stmt, locals: set<string>) which states that statement s doesn't update variables but the ones listed in locals; we use locals to track variables bound in declarations and whose updates won't escape their scope. In particular, if IsPure(s, {}) is true then s doesn't have side effects. Looking at the definition, a declaration (Bind) is pure if the bound statement bval only updates variables from locals, and if its body only updates variables from the set {bvar} + locals. An in-place update (Assign) is pure if it updates the value of a variable from locals. A sequence is pure if it is made of pure statements. We omit the other, straightforward cases. For instance, x := 3 is not pure, while var x := 0; x := 3 is pure because x is locally bound and won't escape its scope.

```
predicate IsPure(
  s: Stmt, locals: set<string> := {}) {
  match s
  case Bind(bvar: Var, bval: Stmt, body: Stmt) =>
    IsPure(bval, locals) && IsPure(body, {bvar} + locals)
  case Assign(avar, aval) =>
    avar in locals && IsPure(aval, locals)
  case Seq(s1, s2) =>
    IsPure(s1, locals) && IsPure(s2, locals)
  ... /* Omitted */ }
```

Now suppose we want to prove the correctness of IsPure, meaning that if a statement is pure in the sense of IsPure then evaluating it leaves the context unchanged. As the proof proceeds by induction over mini-Dafny statements [2] we introduce an induction principle to reason over the mini-Dafny AST.

```
predicate P(st: S, s: Stmt)
predicate P_Step(st: S, s: Stmt, st1: S, v: V)
... // Some definitions omitted
```

[2] mini-Dafny doesn't have loops, which would require induction over semantic derivations.

```
lemma P_Step_Sound(st: S, s: Stmt, st1: S, v: V)
  requires P_Step(st, s, st1, v)
  ensures P(st, s)

lemma InductSeq_Step(
  st: S, s: Stmt, s1: Stmt, s2: Stmt, st1: S, v1: V)
  requires s == Seq(s1, s2)
  requires P_Step(st, s1, st1, v1)
  requires P(st1, s2)
  ensures P(st, s)

... // Omitted: lemmas for the various inductive cases
```

Defining an induction principle for mini-Dafny requires a bit more work than for lists. We require an abstract P(st: S, s: Stmt) predicate which states the target property for statement s in state st. Importantly, we use an abstract type S for the states, because the user might want to carry more information than just a single context. We do the same for values, for similar reasons. We also require an auxiliary predicate P_Step to mention intermediary steps of execution. The P_Step(ctx: S, s: Stmt, ctx1: S, v: int) predicate states that evaluating s starting in state ctx succeeds and yields a new state ctx1 and a value v; as we need to link it to P somehow, we also require that P_Step implies P through the (abstract) lemma P_Step_Sound. We then decompose the inductive cases into precise lemmas. For instance, InductSeq_Step states that, if the target property holds for the execution of s1 starting in st, and also holds for the execution of s2 starting in the state resulting from executing s1, then it holds for the whole sequence s1; s2, starting in st. We finally show how to use this induction principle for the proof of correctness of IsPure.

```
1  datatype S =
2    S(locals: set<string>, ctx: Context)
3  type V = int
4
5  predicate SameCtxs(locals, ctx, ctx1) {
6    && ctx1.Keys == ctx.Keys
7    && ctx1 - locals == ctx - locals }
8
9  predicate P_Step(st, s, st1, v) {
10   && IsPure(s, st.locals)
11   && st.ctx.Keys >= st.locals
12   && InterpStmt(s, st.ctx) == Success((v, st1.ctx))
13   && st1.locals == st.locals
14   && SameCtxs(st.locals, st.ctx, st1.ctx) }
15
16 predicate P(st, s) {
17   IsPure(s, st.locals) ==>
18   st.ctx.Keys >= st.locals ==>
19   match InterpStmt(s, st.ctx) {
20     case Failure _ => true
21     case Success((_, ctx1)) =>
22       SameCtxs(st.locals, st.ctx, ctx1)
23   } }
```

We define the state as a pair of a set of variable names and a context. The predicate P states that if s only updates variables from st.locals according to IsPure (line 17), and if the context has bindings for the variables listed in st.locals (line 18), then evaluating s starting in st.ctx yields (if it succeeds) a context which is unchanged but on the variables listed in st.locals (line 22). We need the condition st.ctx.Keys $\geq$ st.locals to ensure that InterpStmt won't fail while accessing a variable listed in st.locals because it is undefined. The predicate P_Step is similar to P except for the condition that executing s in st must succeed, yielding v st1 (line 12), and for the conjunctions which replace implications (this is slightly technical: suffices to say that P_Step must *unconditionally* state that the execution succeeds; we omit the rule which enforces this). Overall, instantiating the induction principle for IsPure is straightforward, and all the proofs go through automatically.

6.3.2 Experience Using the Induction Principle

We now report on our experience of using the induction principle in practice. We applied the induction principle to several proofs, namely: 1. IsPure; 2. EliminateMulZero: a micro-pass which simplifies statements of the form $0 * s$ or $s * 0$ to 0, provided s is pure;

3. UnchangedVar: a predicate which states that a specific variable is left unchanged by an statement.

We made several changes to the mini-Dafny AST in order to evaluate the cost of updating the proofs: 1. we updated Seq to contain an arbitrary number of statements (`Seq(Stmt, Stmt)` → `Seq(seq<Stmt>)`). 2. we updated Bind (and Assign) to allow multiple declarations (assignments, respectively) at once.

We initially introduced UnchangedVar to reason about variable inlining when working on the more mature version of the compiler [359]. Though simple in appearance, this property is actually subtle and led to expensive proofs; we thus resorted to using a module-based induction principle. In practice, this approach allowed us to dramatically decrease the time we spent on maintaining the proofs.

Factoring out proofs. By using an induction principle we don't have to write the inductive structure of the proofs by hand, and even get automatic variable introduction; after we paid the cost of writing this principle, we thus recover similar advantages to using a tactic like induction in Coq. Instantiating the induction principle by providing definitions for the abstract declarations (e.g., P) requires some boilerplate, especially as we introduced more abstract definitions with each language extension. In practice, however, this work was straightforward, especially as mistakes done when instantiating the induction principle were easy to debug and fix, and in particular easier to fix than the version of the proofs which did not use an induction principle. We also noticed that our instantiations shared similarities: we might leverage this fact to reduce the work even further in the future.

Easy debugging. Because we wrote the rules required by the induction principle so that they are small and precise, we were able to quickly pinpoint the reasons behind a failure whenever a proof broke. In particular, Dafny would tell us which specific lemma (and thus inductive case) failed. This allowed us to easily fix proofs when updating the language, and proved useful when sketching the proofs in the first place, as we could quickly iterate by adjusting the way we stated the properties we targeted to prove until we got them right.

Smooth iterations. Updating the mini-Dafny language required us to update the induction principle several times, either by modifying specific rule statements, or by adding more rules and abstract definitions. As a result, we could focus on specific changes while updating the proofs, and the fact that the rules are small and simple made them more stable. In practice, updating the mini-Dafny language only required us to provide definitions for the new declarations we introduced in the induction principle, and to add *one* assertion at one location in the proof of VarUnchanged, to guide the

SMT solver in its search.

6.4 Related Work

Previous work explored the design of proofs robust to changes, typically by introducing abstractions and interfaces [206, 361–363]. Some work explored the problem of automating inductive proofs altogether by means of heuristics [364–369]; in our case, we target properties which are usually too complex to be fully automated. Other work explored the problem of writing usable inversion theorems but for ITPs like Coq, for instance to control the size of the generated proof terms [370, 371]. An obvious way of overcoming the limitations described in Section 6.2 is to extend ATPs with tactics, as done in [218, 372, 373]. In the context of the present work extending the Dafny prover was however not an option, but it would be interesting to investigate how our approach compares with using tactics in a tool which supports both (like F* or Why3). Interestingly, if the use of heavy automation has been promoted a lot to stabilize proofs when using ITPs [231, 374–377], much less work went into the problem of stabilizing proofs which already relied on a high level of automation. We can however mention attempts to stabilize the proof search itself [378] or preserve VC transformations and proof attempts [379]. Related to the problem of proof maintenance, some recent work explored the problem of proof repair [380, 381], but in the context of tactic-based proof assistants like Coq and not in highly automated theorem provers like Dafny. Finally, it is worth noting that other works tackled the problem of studying language semantics in an ATP [382–384]. We however note that none of those targeted the verification of a multi-pass compiler for a realistic language, which on our side provided the original motivation for introducing our encodings of induction principles [359].

6.5 Conclusion

We demonstrated how to encode an induction principle in Dafny by means of an abstract module. By applying this induction principle to case studies taken from iterations over the proofs of correctness of a compiler for the mini-Dafny language, we showed that our technique has clear benefits to help the user factor out and maintain proofs. In effect, by structuring inductive proofs we relieve the user from mundane work spent on structuring those proofs, and allow them to focus instead of their core. By decomposing inductive proof obligations into small and precise lemmas, we also make it easy to pinpoint the reason behind proof failures. As future work, we are planning to investigate how to automate the process of generating induction principles, as is done in ITPs like Coq.

Part III

Towards Better Scalability with Rust Verification

Chapter 7

Introduction

The Noise*, zero-cost functors and Dafny-in-Dafny projects we presented in the previous chapters allowed us to push frameworks like Low* to their extreme limits; scaling program verification further would require a new generation of tools. By leveraging the practical experience we gained through those projects, we thus decided to work on the creation of a new toolchain which targets Rust programs, Aeneas. The Aeneas framework crucially leverages the Rust type system to generate pure models of programs and their semantics, which allows drastically simplifying reasoning about memory, and takes advantage of the custom, extensible automation made possible by the meta-programming languages of interactive theorem provers.

The Rust programming language continues to rise in popularity. Rust has by now become the darling of developers, voted the most admired language for the 8th year in a row [385]; a favorite of governments, who promote safe languages in the context of cybersecurity [386]; an industry bet, wherein large corporations are actively migrating to Rust [387]; and an active topic in the programming languages research community.

Two research directions have emerged over the past few years. First, many tools now set out to verify Rust programs and prove properties about their behavior, e.g., show that a Rust program matches its specification. But to confidently reason about Rust programs, one needs a solid semantic foundation to build upon. This is the second research direction, namely, understanding the semantics of Rust itself, clarifying the language specification, and showing the soundness of Rust's type system.

For Rust verification, we find tools such as Creusot [244], Prusti [388], Verus [389], or hax [390]. All of those leverage one key insight: the strong ownership discipline enforced by the Rust type system makes verification easier. For Creusot and Verus, this observation turns into a first-order logical encoding of Rust programs that can then be discharged to an SMT solver. For Prusti, the Rust discipline guides the application of

separation logic rules in the underlying Viper framework [391]. And for hax, restricting programs to a pure value-passing subset of Rust allows writing an almost identity-like translation to backends such as F* [75] or ProVerif [392].

For Rust semantics, RustBelt [393] aims to prove the soundness of Rust's type system using a minimalistic model called λ_{Rust}, whose operational semantics is defined by compilation to a core language. The MIRI project [394] aims to provide a reference operational semantics of Rust, even in the presence of unsafe code. Stacked [395] and Tree Borrows [396] aim to clarify the aliasing contract between the programmer and the Rust compiler.

At the intersection of these two axes is RustHornBelt [397], which aims to prove that the RustHorn [243] logical encoding of Rust programs (as used in Creusot) is sound with regards to the semantics of λ_{Rust}, and consequently, that properties proven thanks to the logical encoding hold for the original program.

We propose a new approach to understanding and verifying Rust programs. At the heart of our methodology is a lightweight functional translation of Rust programs. We eschew the complexity of connecting to a separation-logic based backend [393], or relying on prophecy variables to produce a logical encoding [243, 397]. Instead, we synthesize a pure, functional, executable equivalent of the original Rust program, thus producing a lambda-term that does not rely on memory or special constructs. Our translation handles shared, mutable, two-phase and re-borrows, and thus accounts for a very large fraction of typical Rust programs. We call the conceptual framework, as well as the companion tool, Aeneas.

The key point of this work is that we give a *functional* semantics, and thus a *pure* translation, to the subset of Rust we consider. Concretely, we define LLBC, the Low-Level Borrow Calculus, to model that subset. Then, we give it an operational semantics that is functional in nature. We do not rely on memory, addresses or pointer arithmetic; rather, we map variables to values, and track aliasing in a very fine-grained manner, allowing us to handle delicate patterns such as reborrows or two-phase borrows. We claim that our operational semantics captures the *essence* of borrowing; that is, it does not simply apply the rules dictated by Rust's lifetime discipline. Rather, it establishes what is allowed with regards to ownership in the presence of moves, borrows and copies. As such, our semantics can account not only for the current borrow-checker's behavior, but also for its future evolutions, such as Polonius [398].

We then tweak our semantics to abstract the aliasing graph in the presence of function calls; to do so, we use the type of functions as summaries, resulting in LLBC$^\#$, the *symbolic semantics* of LLBC. In effect, this symbolic semantics defines a borrow-checker for LLBC programs, that is, if we successfully execute all the functions in a

program by using the symbolic semantics, then this program is safe to execute.

Our functional translation then simply consumes the execution traces of the symbolic semantics to construct a pure, executable program that is functionally equivalent to the source Rust. To overcome the key difficulty of terminating a borrow, we rely on a technical innovation called *backward functions*, which obviates the need for prophecy variables (as in RustHorn). We wish to emphasize that our functional translation is completely generic. While we demonstrate it by printing our pure programs in Lean syntax, many other options are possible, and we have successfully implemented backends for F*, Coq and HOL4.

Finally, we set out to formalize the fact that LLBC$^\#$ instead defines a borrow-checker for LLBC, and introduce new proof techniques to do so. First, we establish that LLBC, despite using some mildly exotic features, is a reasonable model of execution and really does connect to a traditional heap-and-addresses model of execution. Second, we show that the symbolic semantics of LLBC correctly approximates its concrete semantics, and thus that successful executions in the symbolic semantics guarantee the soundness of concrete executions – in short, that the symbolic interpreter acts as a borrow checker.

Structure of the Following Chapters. The following chapters are organized as follows. We first give a primer of Aeneas's functional translation in Chapter 8. The translation relies on a symbolic evaluation, which itself requires introducing a way of tracking the borrow graph of a program while abstracting away the heap. We solve the latter by introducing LLBC in Chapter 9, a semantics for safe Rust, before tweaking this semantics to introduce the *symbolic* semantics for LLBC (LLBC$^\#$) in Chapter 10. We prove the soundness of the symbolic semantics in Chapter 11, through a theorem which states that a program which is successfully evaluated following the symbolic semantics is *memory safe*, and its evaluation following the semantics of LLBC is in bisimulation with an evaluation using a more standard semantics which explicitly models the heap through a CompCert-like memory model. Building on this soudness theorem, we extend the symbolic semantics in Chapter 12 with a join operation which allows us to support loops. In order not to overwhelm the reader with technical details, we only present our proof methodology and the important theorems, and leave the detailed proofs with the necessary intermediate lemmas in appendix. We finally introduce the (trusted) translation, which is based on a symbolic execution, in Chapter 13, and evaluate the framework in Chapter 14.

Contributions. The work presented here is based on two papers presented at ICFP in 2022 and 2024 [399, 400]. I personally came up with the idea of using forward and backward functions to write pure models of Rust programs, then designed the semantics

of LLBC and LLBC$^{\#}$ to make this translation possible. In particular, I came up with the idea of using the trace of a symbolic execution to generate a functional translation, of modelling loans and borrows as first-class values, and of introducing *region abstractions* to abstract away lifetime constraints. I wrote most of the implementation of the first version of Charon and Aeneas, which was presented with the 2022 paper. Sidney Congard did a preliminary exploration of a potential join operation for LLBC$^{\#}$; I completely reworked those ideas, resulting in the join operation we present here, as well as the fixed-point computations that allow us to support loops. I wrote the first version of the extension which introduced support for loops, which was later modified by Aymeric Fromherz. With regards to the soundness of LLBC$^{\#}$, I personally came up with the idea of using hybrid semantics and defining relations between different languages as sequences of elementary transformations, and wrote the proofs.

Merging the two papers required modifications. The first paper introduced the initial versions of LLBC and LLBC$^{\#}$, and the translation. The second paper revisited those semantics to establish a soundness result our symbolic borrow-checking, and expanded LLBC$^{\#}$ to support control-flow joins and loops. In the following chapters, I present the most recent version of the semantics, which required updating the rules for the translation (Chapter 13). The section about the backends is novel (Section 14.3).

Chapter 8

Aeneas and its Functional Translation, by Example

Before jumping into the various facets of our formalism, we keep an eye on the prize, and immediately showcase how Aeneas translates Rust programs to pure equivalents. In this section, and for the remainder of the thesis, we use Lean syntax for our functional translation; it greatly resembles OCaml and other ML languages, and as such should be familiar to the reader. A brief note about terminology: we adopt the view of Matsakis [401], and refer to *regions*, emphasizing that a region encompasses a set of borrows and loans at a given program point. The Rust compiler and documentation, however, refer to *lifetimes*, which conveys the idea of a syntactic bracket, and a specific implementation technique to enforce soundness. In this work, whenever we talk about Rust specifically, we use "lifetime"; whenever we emphasize our semantic view of ownership, we use "region".

8.1 Mutable Borrows, Functionally

To warm up, we consider an example that, albeit small, showcases many of Rust's features, including its ownership mechanism. In the Rust program below, incr increments a reference, and test_incr acts as a representative caller of the function.

```
1 fn incr(x: &mut i32) {
2   *x = *x + 1;
3 }
4
5 fn test_incr() {
6   let mut y = 0i32;
7   incr(&mut y);
```

```
8   assert!(y == 1);
9 }
```

The incr function operates by reference; that is, it receives the address of a 32-bit signed integer x, as indicated by the & (reference) type. In addition, incr is allowed to modify the contents at address x, because the reference is of the mut (mutable) kind, which permits memory modification. Finally, the Rust type system enforces that mutable references have a unique owner: the definition of incr type-checks, meaning that the function not only *guarantees* it does not duplicate ownership of x, but also can *rely* on the fact that no one else owns x.

In test_incr, we allocate a mutable value (`let mut`) on the stack; upon calling incr, we take a mutable reference (&mut) to y. Statically, y becomes unavailable as long as &mut y is active. In Rust parlance, y is *mutably borrowed* and its ownership has been transferred to the mutable reference. To type-check the call, the type-checker performs a lifetime analysis: the incr function has type $(\&\alpha \mathsf{mut\ i32}) \to ()$, and the &mut y borrow has type $\&\beta \mathsf{mut\ i32}$; both α and β are lifetime variables.

For now, suffices to say that the type-checker ascertains that the lifetime β of the mutable borrow satisfies the lifetime annotation α in the type of the callee, and deems the call valid. Immediately after the call, Rust *terminates* the region β, in effect *relinquishing* ownership of the mutable reference $\&\beta \mathsf{mut\ y}$ so as to make y usable again inside test_incr. This in turn allows the `assert` to type-check, and thus the whole program. Undoubtedly, this is a very minimalistic program; yet, there are two properties of interest that we may want to establish already. The obvious one: the assertion always succeeds. More subtly, doing so requires us to prove an additional property, namely that the addition at line 2 does not overflow.

The key insight of AENEAS is that even though the program manipulates references and borrows, none of this is informative when it comes to reasoning about the program. More precisely: x and y are uniquely owned, meaning that there are no stray aliases through which x or y may be modified; in other words, to understand what happens to y, it suffices to track what happens to &mut y, and therefore to x. Feeding this program to AENEAS generates the following translation, where + is syntactic sugar for an AENEAS primitive that captures the semantics of error-on-overflow in Rust.

```
1 def incr (x : I32): Result I32 :=
2   x + 1#i32 -- evaluates to `fail` in case of overflow
3
4 def test_incr : Result Unit := do
5   let y <- incr x -- monadic bind
6   massert (y = 1#i32) -- monadic assert
```

```
7
8  #assert (test_incr = ok ())
```

This program is semantically equivalent to the original Rust code, but does not rely on the memory: we have leveraged the precise ownership discipline of Rust to generate a functional, pure version of the program. In hindsight, the usage of references in Rust was merely an implementation detail, which is why incr becomes a simple (possibly-overflowing) addition. Should the call to incr (line 7) succeed, its result is bound to y (line 5); the `assert` simply becomes a boolean test that may generate a failure in the error monad.

For the purposes of unit-testing, Aeneas can insert an additional assertion for properly annotated test functions of type unit → unit; the prover shows instantly that our test always succeeds. (In Lean, we execute this assertion directly on the normalizer; Aeneas produces an executable translation, not a logical encoding.) In the remainder of this section, we use ←, Lean's bind operator in the error monad. Other supported backends like F*, Coq and HOL4 also provide this notation.

8.2 Returning a Mutable Borrow and the Use of Backward Functions

Rust programs, however, rarely admit such immediate translations. To see why, consider the following example, where the choose function returns a borrow, as indicated by its return type &αmut.

```
1  fn choose<'a, T>(b: bool, x: &'a mut T, y: &'a mut T) -> &'a mut T {
2    if b { return x; } else { return y; }
3  }
4
5  fn test_choose() {
6    let mut x = 0i32;
7    let mut y = 0i32;
8    let z = choose(true, &mut x, &mut y);
9    *z = *z + 1;
10   assert!(*z == 1);
11   assert!(x == 1);
12   assert!(y == 0);
13 }
```

The choose function is polymorphic over type T and lifetime α; the lifetime annotation captures the expectation that both x and y be in the same region. At call site, x and y

are borrowed (line 8): they become unusable, and give birth to two intermediary values &mut x and &mut y of type &αmut i32. The value returned by choose also lives in region α, i.e., z also has type &αmut i32. The usage of z (lines 8-10) is valid because the region α still exists; the Rust type-checker infers that region α ought to be terminated after line 10, which ends the borrows and therefore allows the caller to regain full ownership of x and y, so that the asserts at lines 11-12 are well-formed.

At first glance, it appears we can translate choose to an obvious conditional. But if we reason about the semantics of choose from the caller's perspective, it turns out that the intuitive translation is not sufficient to capture what happens, e.g., the fact that y is updated while x is being left unchanged and that we observe at lines 11 and 12. To solve this issue we observe that at call site, choose is an opaque, separate function, meaning the caller cannot reason about its precise definition – all that is available is the function type. This type, however, contains precise region information, which one can use to summarize its behavior. When performing the function call, the ownership of x and y is transferred to region α in exchange for z; symmetrically, when the lifetime α terminates, z is relinquished to region α in exchange for regaining ownership of x and y. The former operation flows *forward*; the latter flows *backward*. Using a separation-logic oriented analogy: borrows and regions encode a magic wand that is introduced in a function call and eliminated when the corresponding region terminates.

Our point is: both function call and region termination are semantically meaningful. With the choose example, AENEAS emits a *forward* function which itself returns a *backward* continuation. The forward function is used to model the function call at line 8, while the backward continuation is used to propagate changes back into x and y when lifetime α terminates.

```
1  def choose {T : Type} (b : Bool) (x : T) (y : T) :
2    Result (T × (T → T × T)) :=
3    if b
4    then ok (x, fun z => (ret, y))
5    else ok (y, fun z => (x, ret))
6
7  let test_choose : Result Unit := do
8    let (z, choose_back) <- choose true 0#i32 0#i32;
9    let z1 <- z + 1#i32
10   massert (z1 = 1#i32) -- monadic assert
11   let (x, y) := choose_back z1
12   massert (x = 1#i32)
13   massert (y = 0#i32)
14   ok ()
15
```

```
16 #assert (test_choose = ok ())
```

The call to choose becomes a call to the forward function choose (line 8); we bind the result of the addition (provided no overflow occurs) to z (line 9); then, per the rules of Rust's type-checker, region α terminates which compels us to call the backward function choose_back. The intuitive effect of calling choose_back is as follows: we relinquish z, which was in region α; doing so, we propagate any updates that may have been performed through z onto the variables whose ownership was transferred to α in the first place, namely x and y. This bidirectional approach is akin to lenses [402], except we propagate the output back to possibly-many inputs; in this case, z is a view over either x or y, and the backward function reflects the update to z onto the original variables. Thus, both variables are re-bound (line 11), before chaining the two asserts (lines 12 and 13).

From the caller's perspective, the computational content of choose is unknown; but the signature of choose reveals the effect it may have onto its inputs x and y, which in turns allows us to derive the type of the backward and forward functions from the signature of choose itself. The result is a modular, functional translation that does not rely on any sort of cross-function inlining or whole-program analysis. To synthesize the backward function, it suffices to invert the direction of assignments; in one case, z flows to x and y remains unchanged; the other case is symmetrical.

8.3 Functions with no Output Borrows

Let us now revisit to the example of the incr function. We presented it as a Rust function which doesn't require the use of backward functions in the generated model; it actually does. The translation performed by Aeneas is generic in the type of the function, and the emitted model always has one backward function per region present in the signature. In the case of incr<α>(x: &mut i32), the translation should return a value of type Unit (for the output of incr), together with a backward function for the lifetime α. As incr doesn't return any mutable borrow for α, contrary to choose, this backward function doesn't actually consume any inputs; as such it is not a function but a *value* of type i32. As a consequence, the translation of incr should have the output type Result (Unit × I32), and this is actually the pure model initially generated by Aeneas.

```
def incr (x : I32): Result (Unit × I32) := do
  let x1 <- x + 1#i32
  ok ((), x1)
```

Finally, because we are concerned with generating idiomatic code, we implemented several micro-passes to improve the quality of the translation before emitting it. Those include a unit elimination pass which simplifies the type Result (Unit × I32), yielding the pure model which is actually output by AENEAS and that we showed at the beginning of the section.

8.4 Recursion and Data Structures

It might not be immediately obvious that this translation technique scales up beyond toy examples; we now crank up the complexity and show how AENEAS can handle a wide variety of idioms while still delivering on the original promise of a lightweight functional translation. Our next example is list_nth, which allows taking a mutable reference to the n-th element of a list, mutating it, and regaining ownership of the list.

```
enum List<T> {
  Cons(T, Box<List<T>>),
  Nil,
}

fn list_nth_mut<'a, T>(l: &'a mut List<T>, i: usize) -> &'a mut T {
  match l {
    Nil => { panic!() }
    Cons(x, tl) => {
      if i == 0 { x }
      else { list_nth_mut(tl, i - 1) }
    }
  }
}

fn sum(l: & List<i32>) -> i32 {
  match l {
    Nil => { return 0; }
    Cons (x, tl) => { return *x + sum(tl); }
  }
}

fn test_nth() {
  let mut l = Cons (1, Box::new(Cons (2, Box::new(Nil))));
  let x = list_nth_mut(&mut l, 1);
  *x = *x + 1;
  assert!(sum(&l) == 4);
}
```

This example relies on several new concepts. Parametric data type declarations (line 1) resemble those in any functional programming language such as OCaml or SML. The Box type denotes a heap-allocated, uniquely-owned piece of data. Without the Box indirection, List would describe a type of infinite size and would be rejected. Immutable (or shared) borrows (line 16) do not sport a mut keyword; they do not permit mutation, but the programmer may create infinitely many of them. Only when all shared borrows have been relinquished does full ownership return to the borrowed value.

The complete translation is as follows:

```
1  inductive List (T : Type) :=
2  | Cons : T -> List T -> List T
3  | Nil : List T
4
5  def list_nth_mut {T : Type} (l : List T) (i : Usize) :
6    Result (T × (T -> List T)) :=
7    match l with
8    | Cons x tl =>
9      if i = 0#usize
10     then ok (x, fun x' => Cons x' tl)
11     else do
12       let i1 <- i - 1#usize
13       let (v, list_nth_mut_back) <- list_nth_mut tl i1
14       ok (v, fun v' => Cons x (list_nth_mut_back v'))
15   | Nil => fail
16
17 def sum (l : list_t i32) : Result I32 :=
18   match l with
19   | Cons x tl => do
20     let i <- sum tl
21     x + i
22   | Nil => ok 0#i32
23
24 def test_nth : Result Unit := do
25   let l := Cons (1, Cons (2, Nil))
26   let (x, list_nth_mut_back) <- list_nth_mut l 1
27   let x1 <- x + 1#i32
28   let l1 := list_nth_mut_back x1
29   let i <- sum l1
30   massert (not (i = 4))
31
32 #assert (test_nth_fwd = Return ())
```

We first focus on the caller's point of view. Continuing with the lens analogy, we focus on (or "get") the n-th element of the list via a call to list_nth_mut (line 26);

modify the element (line 27); then close (or "put" back) the lens, and propagate the modification back to the list via a call to list_nth_mut_back (line 28). The backward continuation is of particular interest. In the Nil case, it simply updates the list to replace the head (x) with its new value (x', at line 10). In the Cons case, it updates the tail of the list by using the backward continuation returned by the recursive call at line 13, and reconstructs the complete list by consing the (unchanged) head value (line 14)[1].

8.5 Loops

The example above implements list_nth_mut as a recursive function; a more idiomatic version would implement it with a loop instead, as shown below.

```
fn list_nth_mut<'a, T>(mut l: &'a mut List<T>, i: usize) -> &'a mut T {
  loop {
    match l {
      Nil => { panic!() }
      Cons(x, tl) => {
        if i == 0 { return x; }
        else {
          l = tl;
        } } } } }
```

The translation is as follows:

```
def list_nth_mut_loop {T : Type} (l : List T) (i : U32) :
  Result (T × (T → List T)) :=
  match l with
  | Cons x tl =>
    if i = 0#u32
    then
      ok (x, fun x' => Cons x' tl)
    else do
      let i1 ← i - 1#u32
      let (v, back) ← list_nth_mut_loop tl i1
      ok (v, fun v' => Cons x (back v'))
  | Nil => fail
```

[1] If the list_nth_mut function is structurally terminating and is thus accepted with no difficulty by a theorem prover like Lean, one may wonder how we handle definitions which do not have this property. In the case of the Lean backend, we implemented a custom elaboration triggered by the attribute divergent, and which leverages the fact that our functions live in an error monad to encode them by using a custom fixed-point operator. As this custom elaboration also proves and registers on the fly the unfolding lemmas expected by the user, this method allows us to define (potentially) partial functions in a lightweight manner. In practice, the user can perform their proofs without even being aware of the existence of this custom elaboration; we elide the precise details of this encoding here.

```
def list_nth_mut {T : Type} (l : List T) (i : U32) :
  Result (T × (T -> (List T))) :=
  list_nth_mut_loop l i
```

This time, AENEAS introduces an auxiliary, recursive definition list_nth_mut_loop to model the body of the loop. The function list_nth_mut then simply calls this function, doing nothing else because its body is only made of this loop. The model of the loop itself is actually exactly the same as the translation of the recursive version of list_nth_mut, which is perhaps not surprising given the fact that this function is tail-call recursive.

Traits. We end up this tour of AENEAS' translation by looking at traits, which implement a typeclass system for Rust.

```
1  trait Counter {
2      fn incr<'a>(&'a mut self) -> usize;
3  }
4
5  impl Counter for usize {
6      fn incr(&mut self) -> usize {
7          let x = *self;
8          *self += 1;
9          x
10     }
11 }
12
13 fn use_counter<'a, T: Counter>(cnt: &'a mut T) -> usize {
14     cnt.incr()
15 }
```

In the snippet of code above, Counter defines a trait which requires a single method incr. We implement an instance of Counter for the type usize on lines 5-11, which simply increments self and returns its former value. Finally, we define a polymorphic function use_counter which takes as input a mutable borrow to an element of a type T which must implement the trait Counter, as indicated by T : Counter. It then uses this trait obligation to call the incr method with cnt at line 14. We show the result of the translation below.

```
1  structure Counter (Self : Type) where
2    incr : Self -> Result (Usize × Self)
3
4  def CounterUsize.incr (self : Usize) : Result (Usize × Usize) := do
```

```
5     let self1 <- self + 1#usize
6     Result.ok (self, self1)
7
8   def CounterUsize : Counter Usize := {
9     incr := CounterUsize.incr
10  }
11
12  def use_counter {T : Type} (CounterInst : Counter T) (cnt : T) :
13    Result (Usize × T) :=
14    CounterInst.incr cnt
```

The trait Counter is modeled as a structure containing one field, for the method incr; we resort to using structures rather than typeclasses because we have no way of ensuring that the typeclass inference implemented by the backend will yield the same result as the trait resolution performed by Rust. Here, we crucially leverage the modular nature of the translation: the type of incr in Rust gives us all the information we need; in particular, in the pure world, we can model incr with a function of type Self → Result (Usize × Self). The translation of the instance of Counter for usize is straightforward (lines 4-10). Finally, the translation of use_counter is a function which explicitly receives an input of type Counter T, which it then uses to model the call to the incr method.

Chapter 9

An Ownership-Centric Semantics for Rust

Before explaining the functional translation above, we must first define our input language and its operational semantics. We first present a series of short snippets of Low-Level Borrow Calculus (LLBC) code, and show in comments how our execution environments model the effect of each statement. LLBC is a cleaned up version Rust's Mid-level Intermediate Representation (MIR), one of the intermediate languages used by the Rust compiler; in particular, LLBC uses high-level syntax constructs such as `if then else`, loops, breaks and continues instead of MIR's gotos. As such, LLBC can be seen as a desugared version of Rust's syntax AST, where for instance copies and moves are explicit. We then present the semantics of LLBC in a systematic manner in Section 9.2.

9.1 The Low-Level Borrow Calculus - Examples

9.1.1 Mutable Borrows

We consider the snippet of code below. After line 1, x points to 0, which we write $x \mapsto 0$. At line 2, px *mutably* borrows x. As we mentioned earlier, a mutable borrow grants exclusive ownership of a value, and renders the borrowed value unusable for the duration of the borrow. We reflect this fact in our execution environment as follows: x is marked as "loaned-out", in a mutable fashion, and px is known to be a mutable borrow. Furthermore, ownership of the borrowed value now rests with px, so the value within the mutable borrow is 0. Finally, we need to record that px is a borrow *of* x: we issue a fresh loan identifier ℓ that ties x and px together. The same operation is repeated at line 3. Value 0 is now held by ppx, and px, too, becomes "loaned out".

```
1 let mut x = 0;          // x ↦ 0
2 let mut px = &mut x;    // x ↦ loan^m ℓ,   px ↦ borrow^m ℓ 0
3 let ppx = &mut px;      // x ↦ loan^m ℓ,   px ↦ loan^m ℓ',     ppx ↦ borrow^m ℓ' (borrow^m ℓ 0)
```

Our environments thus precisely track ownership; doing so, they capture the aliasing graph in an exact fashion. Another point about our style: this representation allows us to adopt a *focused* view of the borrowed value (e.g. 0), solely through its owner (e.g. ppx), without worrying about following indirections to other variables. We believe this approach is unique to our semantics; it has, in our experience, greatly simplified our reasoning and is crucial to making the symbolic semantics and the functional translation work(Chapter 13).

We remark that our style departs from Stacked Borrows [395], where the modified value remains with x. We also note that our formalism cannot account for unsafe blocks; allowing unfettered aliasing would lead to potential cycles, which we cannot represent. This is an intentional design choice for us: we circumscribe the problem space in order to achieve an intuitive, natural semantics and a lightweight functional translation. Aeneas shines on safe Rust programs, and can be complemented by more advanced tools such as RustBelt for unsafe parts.

9.1.2 Shared borrows

Shared borrows behave more like traditional pointers. Multiple shared borrows may be created for the same value; each of them grants read-only access to the underlying value. The owner also retains a read-only access to the borrowed value; regaining the full ownership requires terminating all of the borrows. In the example below, the value $(0, 1)$ is borrowed in a shared fashion, at line 2. This time, the value remains with x; but taking an immutable reference to x still requires book-keeping. We issue a new loan ℓ, and record that px1 is now a shared borrow associated to loan ℓ; to understand which value px1 points to, we simply look up in the environment who is the owner of ℓ, and read the associated value. Repeated shared borrows are permitted: at line 3, we create a new shared borrow of x; as x is already borrowed we do not need to update its value and simply reuse the loan identifier ℓ. At line 4, we anticipate on our internal syntax, where moves and copies are explicit, and copy the first component of x. Values that are loaned immutably, like x, can still be read; in the resulting environment, y points to a copy of the first component, and bears no relationship whatsoever to x. Finally, at line 5, we *reborrow* (through px1) the first component of the pair only. First, to dereference px1, we perform a lookup and find that x owns ℓ. Then, we perform book-keeping and update the value loaned by x, so as to reflect that its first component

has been loaned out, introducing a fresh loan identifier ℓ' at the same time.

```
1 let x = (0, 1);      // x ↦ (0, 1)
2 let px1 = &x;        // x ↦ loan^s ℓ (0, 1),   px1 ↦ borrow^s ℓ
3 let px2 = &x;        // x ↦ loan^s ℓ (0, 1),   px1 ↦ borrow^s ℓ,   px2 ↦ borrow^s ℓ
4 let y = copy x.0;    // x ↦ loan^s ℓ (0, 1),   px1 ↦ borrow^s ℓ,   px2 ↦ borrow^s ℓ,   y ↦ 0
5 let z = &(*px1.0); // x ↦ loan^s ℓ ((loan^s ℓ' 0), 1),  px1 ↦ ...,  px2 ↦ ...,  y ↦ 0,  z ↦ borrow^s ℓ'
```

In our presentation, shared borrows behave like pointers, and every one of them is statically accounted for via the loan identifier attached to the borrowed value. The reader might find this design choice surprising: indeed, in Rust, shared borrows behave like immutable values and we ought to be able to treat them as such. Recall, however, that one of our key design goals is to give a *semantic* explanation of borrows; as such, our precise tracking of shared borrows allows us to know precisely when all aliases have been relinquished, and full ownership of the borrowed value has been regained, as we shall see in Section 9.1.4.

Finally, we reiterate our remark that our formalism allows keeping track of the aliasing graph in a precise fashion; the discipline of Rust bans cycles, meaning that the aliasing graph is always a tree. This style of representation resembles Mezzo [403], where loan identifiers are akin to singleton types, and entries in the environment are akin to permissions.

9.1.3 Reborrows

We now consider a particularly tricky example accepted by the Rust compiler. While the complexity seems at first gratuitous, it turns out that the pattern of borrowing a dereference (i.e., `&mut (*px)`) is extremely common in Rust. The reason is subtle: in the post-desugaring MIR internal Rust representation, moves and copies are explicit, meaning function calls of the form f(move px) abound. Such function calls *consume* their argument, and render the user-declared reference px unusable past the function call. To offer a better user experience, Rust automatically "reborrows" the content pointed to by px, and rewrites the call into `f(move (&mut (*px)))` at desugaring-time. Thus, only the intermediary value is "lost" to the function call; relying on its lifetime analysis, the Rust compiler concludes that the user-declared reference px remains valid past the function call, hence making the programmer's life easier.

Even more twisted, reborrowing (part of) oneself, as done at line 3, is also a very common pattern, in particular when visiting recursive data-structures. When recursively diving into a list l, in the Cons case, we need to update l so that it points to its tail; in practice this is tantamount to reborrowing oneself. Capturing the semantics of such an update must be done with great care, in order to preserve precise aliasing information.

We propose an example that shows how to reborrow oneself. Rust accepts this program; we now explain with our semantics *why* it is sound. In the example below, at line 2, the environment offers no surprises. Justifying the write at line 3 requires care. We borrow $*px$, which modifies px to point to $\mathsf{borrow}^m\ \ell'\ 0$, and returns $\mathsf{loan}^m\ \ell'$; the value about to be overwritten is stored in a fresh, anonymous variable _, and $\mathsf{loan}^m\ \ell'$ gets written to px.

```
1 let mut x = 0;        // x ↦ 0
2 let mut px = &mut x; // x ↦ loan^m ℓ,   px ↦ borrow^m ℓ 0
3 px = &mut (*px);      // x ↦ loan^m ℓ,   _ ↦ borrow^m ℓ (loan^m ℓ'),   px ↦ borrow^m ℓ ′0
4                       // After ending ℓ′:
5                       // x ↦ loan^m ℓ,   _ ↦ borrow^m ℓ 0,             px ↦ ⊥
6                       // After ending ℓ:
7 assert!(x==0);        // x ↦ 0,          _ ↦ ⊥,                        px ↦ ⊥
```

Saving the old value is crucial for line 7. For the assertion, we need to regain full ownership of x. To do so, we first terminate ℓ'. This *reorganizes* the environment, with two consequences. First, px becomes unusable, which we write $px \mapsto \bot$. Second, the anonymous value, which we had judiciously kept in the environment, becomes $\mathsf{borrow}^m\ \ell\ 0$; this yields the environment at line 5. We reorganize the environment again, to terminate ℓ; the effect is similar, and results in $x \mapsto 0$, i.e. full ownership of x; we get the final environment of line 7. This example illustrates a key characteristic of our approach, which is that we reorganize borrows in a *lazy* fashion, and don't terminate a borrow until we need to get the borrowed value back.

9.1.4 Lazy Borrow Semantics - An Illegal Borrow

We offer an final example which leverages our reorganization rules; furthermore, the example illustrates how our semantics reaches the same conclusion as rustc, though by different means, on a borrowing error. At line 2, a borrow is introduced, which results in a fresh loan ℓ. To make progress, we terminate borrow ℓ at line 4. Line 5 then type-checks, with a fresh borrow ℓ'. Then, we error out at line 7: px_1 has been terminated, and we cannot dereference $\bot$.

The Rust compiler proceeds differently, and implements an analysis which requires computing borrow constraints for an entire function body. The compiler notices that lifetime ℓ must go on until line 7 (because of the `assert`), which prevents a new borrow from being issued at line 5. Rust thus ascribes the error to an earlier location than we do, that is, line 5. We remark that the Rust behavior is semantically equivalent to ours; however, our lazy approach which terminates borrows only as needed has the advantage

that evaluation can proceed in a purely forward fashion, without requiring a non-local analysis or backtracking. This on-demand approach is similar to Stacked Borrows.

```
1 let mut x = 0;      // x ↦ 0
2 let px1 = &mut x; // x ↦ loan^m ℓ,   px1 ↦ borrow^m ℓ 0
3                   // After ending ℓ:
4                   // x ↦ 0,   px1 ↦ ⊥
5 let px2 = &mut x; // aeneas: x ↦ loan^m ℓ',   px1 ↦ ⊥,   px2 ↦ borrow^m ℓ' 0
6                   // rustc: error: cannot borrow `px1` as mutable more than once at a time
7 assert!(*px1 == 0); // aeneas: error, attempt to deference unusable variable px1
```

9.1.5 Two-Phase Borrows

We finally review *two-phase* borrows, which are introduced by the Rust compiler at desugaring time. A two-phase borrow starts as a shared borrow in a "reservation phase", and later gets activated into a full mutable borrow. Reserved borrows enable a variety of very common idioms [404] without resorting to more advanced desugarings. Below, we create a two-phase borrow at line 3. From the point of view of the lender, a two-phase borrow acts as a shared borrow; the value of x, which is already immutably borrowed, remains unchanged. However, px2 maps to the *reserved* borrow $\mathsf{borrow}^r\ \ell$. At line 4 we evaluate the assertion as before. However, at line 9 we need to use the two-phase borrow to perform an in-place update. We thus reorganize the environment by ending the shared borrow ℓ contained by px1 at line 6. There now only remains a single borrow pointing to x, the reserved borrow of px2, that we can promote to a mutable borrow at line 8. This allows us to evaluate the in-place update at line 9.

```
1 let mut x = 0;           // x ↦ 0
2 let px1 = &x;            // x ↦ loan^s ℓ 0,   px1 ↦ borrow^s ℓ
3 let px2 = &two-phase x; // x ↦ loan^s ℓ 0,   px1 ↦ borrow^s ℓ,   px2 ↦ borrow^r ℓ
4 assert!(*px1 == 0);      // x ↦ loan^s ℓ 0,   px1 ↦ borrow^s ℓ,   px2 ↦ borrow^r ℓ
5                          // After ending the shared borrow of px1:
6                          // x ↦ loan^s ℓ 0,   px1 ↦ ⊥,   px2 ↦ borrow^r ℓ
7                          // After promoting the reserved borrow of px2:
8                          // x ↦ loan^m ℓ,   px1 ↦ ⊥,   px2 ↦ borrow^m ℓ 0
9 *px2 = 1;                // x ↦ loan^m ℓ,   px1 ↦ ⊥,   px2 ↦ borrow^m ℓ 1
```

9.2 The Low-Level Borrow Calculus - Rules

We now formally introduce and define our semantics of Rust programs. We start with the Low-Level Borrow Calculus ("LLBC", Figure 9.1), our source language. LLBC is in large part inspired by MIR, Rust's post-desugaring internal representation, notably: all

τ	::=	type
	$\mathsf{bool} \mid \mathsf{uint32} \mid \mathsf{int32} \mid ...$	literal types
	$\&^\rho \mathsf{mut}\, \tau \mid \&^\rho \tau$	mutable, immutable (shared) borrow
	$\alpha, \beta, ...$	type variables
	$\tau_0 + \tau_1$	sum
	$(\vec{\tau})$	pair ($\mathsf{len}(\vec{\tau}) = 2$) or unit ($\mathsf{len}(\vec{\tau}) = 0$)
	$\mathsf{Box}\, \tau$	boxed type
	$\mu X.\, \tau$	equirecursive type
s	::=	statement
	$\emptyset$	empty statement (nil)
	$s;\ s$	sequence (cons)
	$p := rv$	assignment
	$\mathsf{if}\, op\, \mathsf{then}\, s\, \mathsf{else}\, s$	conditional
	$\mathsf{match}\, p\, \mathsf{with}\, \overrightarrow{C \to s}$	data type case analysis
	$\mathsf{free}\, p$	free
	return	function exit
	panic	unrecoverable error
	$\mathsf{loop}\, s$	loop
	$\mathsf{break}\, i \mid \mathsf{continue}\, i$	break, continue to an outer loop
	$p := f(\vec{op})$	function call
x		variable
p	$::= P[x]$	place
rv	::=	assignable "r" values
	op	operand
	$\&\mathsf{mut}\, p \mid \&\, p$	mutable, immutable (shared) borrow
	$\&\mathsf{reserved}\, p$	reserved (two-phase) borrow
	$!op \mid op + op \mid op - op \mid ...$	operators
	$\mathsf{new}\, op$	new
op	::=	operand
	$\mathsf{move}\, p$	ownership transfer
	$\mathsf{copy}\, p$	scalar copy
	$\mathsf{true} \mid \mathsf{false} \mid n_{\mathsf{i32}} \mid n_{\mathsf{u32}} \mid ...$	literal constants
	$\mathsf{Left}\, op \mid \mathsf{Right}\, op$	sum constructor
	$(\vec{op})$	pair ($\mathsf{len}(\vec{op}) = 2$) or unit ($\mathsf{len}(\vec{op}) = 0$)
P	::=	path
	$[.]$	base case
	$*P$	deref
	$P.f$	field selection
D	::=	top-level declaration
	$\mathsf{fn}\, f\, \langle \vec{\rho},\ \vec{\tau} \rangle\ (\vec{x}_{\mathsf{arg}} : \vec{\tau})\ (\vec{x}_{\mathsf{local}} : \vec{\tau})\ (x_{\mathsf{ret}} : \tau)\ \{\ s\ \}$	function declaration

Figure 9.1: The Low-Level Borrow Calculus: Syntax

local variables $\vec{x}_{\mathsf{local}}$ are bound at the beginning of the function declaration; returning a value from a function amounts to writing into the special variable x_{ret}, followed by return; all sub-expressions have been named so as to fit within MIR's statement/r-value/operand categories; and all variables within expressions have been desugared to either a move, a copy or a borrow. However, and unlike MIR, LLBC retains some high-level constructs: control-flow remains structured (LLBC statements are thus the fusion of MIR's statements and terminators); and case analysis on data types is exposed via a limited form of (complete) pattern matching, as opposed to a low-level integer switch on the tag. In order to make the proofs in the later sections simpler (Chapter 11), we also restrict data types to sums and pairs; a more general presentation of the formalism with regards to datatypes was initially introduced in [399]. We remark that data types may match on a *path* only; this merely imposes that the scrutinee be let-bound before examining it, something that MIR does internally. We also use pure expressions to allocate data types, rather than the progressive (mutable) initialization pattern used by MIR; and we see structures as data types equipped with a single constructor, for conciseness. Finally, we do not formalize traits in our semantics, though the implementation supports them. Staying close to MIR is a design choice in line with other Rust-related works [395]; it allows for fewer, simpler rules, and a more precise description of what happens from the point of view of ownership.

At the heart of LLBC is a notion of *place*, i.e. the combination of a base variable (e.g. x) and a series of field offsets and indirections known as a *path* (e.g. $*\_$.f). A place is akin to the notion of "lvalue" in, e.g., C. Assigning or returning from a function can only be done into a *place*. The grammar of rvalues and operands (Rust's limited form of expression) is very explicit, in that every use of a variable is performed through a copy, a move, or a borrow of a place.

9.3 A Structured Memory Model

Rust marries high-level concepts, such as ownership, a strong notion of value, and data types, with low-level concepts such as moves and copies, paths through a base address, and modifications at depth throughout the store. We propose a semantics that operates exclusively in terms of values (that is, no memory addresses), yet still permits fine-grained memory mutations as allowed by Rust.

Figure 9.2 presents our environments, which we write Ω, and our values, which we write v. We do not distinguish between states and environments, and use the two terms interchangeably. The state maps variable names x to values v: we have no notion of arbitrary memory addresses, or pointer arithmetic. Our values v are carefully crafted

v	::=		value
		$\mathsf{true} \mid \mathsf{false} \mid n_{\mathsf{i32}} \mid n_{\mathsf{u32}} \mid \ldots$	literal constants
		$\mathsf{Left}\, v \mid \mathsf{Right}\, v$	sum value
		$()$	unit
		(v_0, v_1)	pair
		$\bot$	bottom (invalid) value
		$\mathsf{loan}^m\, \ell$	mutable loan
		$\mathsf{borrow}^m\, \ell\, v$	mutable borrow
		$\mathsf{loan}^s\, \ell\, v$	shared loan
		$\mathsf{borrow}^s\, \ell$	shared borrow
		$\mathsf{borrow}^r\, \ell$	reserved borrow
r	::=		result
		$()$	unit
		return	successful, possibly-early exit
		panic	unrecoverable error state
		$\mathsf{break}\, i \mid \mathsf{continue}\, i$	break, continue
id	::=		environment binding identifier
		x	local variable identifier
		$\_$	anonymous identifier
Ω^{LLBC}	::=	$\{\ \mathsf{env} : id \underset{\text{partial}}{\longrightarrow} v,\ \mathsf{stack} : [[x]]\ \}$	state

Figure 9.2: The Low-Level Borrow Calculus: Environments and Values

to model the semantics of borrows and ownership tracking in Rust; several of them already appeared in our earlier examples.

The combination of places, environments and values allows us to define reads and writes already. Reads and writes are defined in terms of our *structured* memory model: we do not have any notion of memory address, but *do* have a notion of path combined with a base "address" (variable) x, that is, a place; this permits reads and writes, at depth, through references. We present the rules for reading (R-*) in Figure 9.3 and writing (W-*) in Figure 9.4.

For *reading*, we write $\vdash \Omega(p) \overset{\mathsf{k}}{\Rightarrow} v$, meaning reading from Ω at place p with capability k produces v. The capability $k \in \{\mathsf{mut}, \mathsf{imm}, \mathsf{mov}\}$ constrains the way we access the place, and depends on the expression we evaluate; they are necessary mostly for *writing*, but we use them also for the reading rules for consistency. We use imm when creating a shared borrow or when copying a value (E-SharedBorrow, E-Copy), mut when creating a mutable borrow (E-MutBorrow), and mov when moving a value (E-Move). We can dereference a shared borrow only with capability imm (R-Deref-SharedBorrow): as mov and mut both require a mutable access, we need to end the loan before. Similarly, moving a value out of a borrow is not permitted: we can dive into a mutable borrow

$$\textsc{Read}\ \frac{p = P[x] \qquad \Omega(x) = v \qquad \Omega \vdash P(v) \overset{k}{\Rightarrow} v'}{\vdash \Omega(p) \overset{k}{\Rightarrow} v'}$$

$$\textsc{R-Base}\ \frac{}{\Omega \vdash [.](v) \overset{k}{\Rightarrow} v}$$

$$\textsc{R-ProjSum-Left}\ \frac{\Omega \vdash P(v) \overset{k}{\Rightarrow} v'}{\Omega \vdash P((\mathsf{Left}\ v).0) \overset{k}{\Rightarrow} v'}$$

$$\textsc{R-ProjSum-Right}\ \frac{\Omega \vdash P(v) \overset{k}{\Rightarrow} v'}{\Omega \vdash P((\mathsf{Right}\ v).0) \overset{k}{\Rightarrow} v'}$$

$$\textsc{R-ProjPair-Left}\ \frac{\Omega \vdash P(v) \overset{k}{\Rightarrow} v'}{\Omega \vdash P((v, v_1).0) \overset{k}{\Rightarrow} v'}$$

$$\textsc{R-ProjPair-Right}\ \frac{\Omega \vdash P(v) \overset{k}{\Rightarrow} v'}{\Omega \vdash P((v_0, v).1) \overset{k}{\Rightarrow} v'}$$

$$\textsc{R-SharedLoan}\ \frac{P \neq [.] \qquad \Omega \vdash P(v) \overset{\mathsf{imm}}{\Rightarrow} v'}{\Omega \vdash P(\mathsf{loan}^s\ \ell\ v) \overset{\mathsf{imm}}{\Rightarrow} v'}$$

$$\textsc{R-Deref-Box}\ \frac{\Omega \vdash P(v) \overset{k}{\Rightarrow} v'}{\Omega \vdash P(*(\mathsf{Box}\ v) \overset{k}{\Rightarrow} v'}$$

$$\textsc{R-Deref-SharedBorrow}\ \frac{\mathsf{loan}^s\ \ell\ v \in \Omega \qquad \Omega \vdash P(\mathsf{loan}^s\ \ell\ v) \overset{\mathsf{imm}}{\Rightarrow} v'}{\Omega \vdash P(*(\mathsf{borrow}^s\ \ell)) \overset{\mathsf{imm}}{\Rightarrow} v'}$$

$$\textsc{R-Deref-MutBorrow}\ \frac{\Omega \vdash P(v) \overset{\mathsf{imm},\mathsf{mut}}{\Rightarrow} v'}{\Omega \vdash P(*(\mathsf{borrow}^m\ \ell\ v)) \overset{\mathsf{imm},\mathsf{mut}}{\Rightarrow} v'}$$

Figure 9.3: Read Rules (LLBC)

with capabilities imm and mut but not mov (R-Deref-MutBorrow).

Accessing a place requires looking up the "base pointer" (variable) x found in p (Read), then deferring to an auxiliary judgment of the form $\Omega \vdash P(v_x) \overset{k}{\Rightarrow} v$, meaning following path P into v_x with capability k produces value v. We can follow path P as long as the value v_x is of the right shape (R-Deref-MutBorrow, R-ProjPair-Left, R-ProjSum-Left, etc.). Dereferencing a mutable borrow simply requires diving into the borrowed value, since the mutable borrow has an exclusive access to the value it points to (R-Deref-MutBorrow). Conversely, reading from a shared borrow requires looking up the owner of the loan to find the value being pointed to (R-Deref-SharedBorrow).

Rule R-Base is our base case: if we have reached the end of the path P, we simply return the value found there. One subtlety occurs in the case of shared *loans*. Rule R-SharedLoan permits reading from a value that is currently immutably borrowed, which is allowed in Rust. However, we only do so if necessary; that is, if we must follow further indirections in P (i.e., $P \neq [.]$). If there are no further indirections (i.e., $P = [.]$), R-Base kicks in and returns a value of the form loan^s. This is intentional, and will prove useful when creating shared borrows (E-SharedBorrow), as we shall see shortly.

For *writing*, we write $\vdash \Omega[p \leftarrow v] \overset{k}{\Rightarrow} \Omega'$, meaning assigning value v into Ω at place p with capability k produces an updated environment Ω' (Write). As before, we follow the structure of v_x (Write), and defer to an auxiliary judgment of the form

$$\textsc{Write}\ \frac{p = P[x] \qquad \Omega(x) = v \qquad \Omega \vdash P(v) \leftarrow w \overset{\mathsf{k}}{\Rightarrow} (v', \Omega') \qquad \Omega'' = (\Omega'(x) := v')}{\vdash \Omega[p \leftarrow w] \overset{\mathsf{k}}{\Rightarrow} \Omega''}$$

$$\text{W-}\textsc{Base}\ \frac{}{\Omega \vdash [.](v : \tau) \leftarrow (w : \tau) \overset{\mathsf{k}}{\Rightarrow} (w, \Omega)}$$

$$\text{W-}\textsc{ProjSum-Left}\ \frac{\Omega \vdash P(v) \leftarrow w \overset{\mathsf{k}}{\Rightarrow} (v', \Omega')}{\Omega \vdash P((\mathsf{Left}\, v).0) \leftarrow w \overset{\mathsf{k}}{\Rightarrow} (\mathsf{Left}\, v', \Omega')}$$

$$\text{W-}\textsc{ProjSum-Right}\ \frac{\Omega \vdash P(v) \leftarrow w \overset{\mathsf{k}}{\Rightarrow} v' \dashv \Omega'}{\Omega \vdash P((\mathsf{Right}\, v).0) \leftarrow w \overset{\mathsf{k}}{\Rightarrow} (\mathsf{Right}\, v', \Omega')}$$

$$\text{W-}\textsc{ProjPair-Left}\ \frac{\Omega \vdash P(v) \leftarrow w \overset{\mathsf{k}}{\Rightarrow} (v', \Omega')}{\Omega \vdash P((v, v_1).0) \leftarrow w \overset{\mathsf{k}}{\Rightarrow} ((v', v_1), \Omega')}$$

$$\text{W-}\textsc{ProjPair-Right}\ \frac{\Omega \vdash P(v) \leftarrow w \overset{\mathsf{k}}{\Rightarrow} (v', \Omega')}{\Omega \vdash P((v_0, v).1) \leftarrow w \overset{\mathsf{k}}{\Rightarrow} ((v_0, v'), \Omega')}$$

$$\text{W-}\textsc{SharedLoan}\ \frac{P \neq [.] \qquad \Omega \vdash P(v) \leftarrow w \overset{\mathsf{imm}}{\Rightarrow} (v', \Omega')}{\Omega \vdash P(\mathsf{loan}^s\, \ell\, v) \leftarrow w \overset{\mathsf{imm}}{\Rightarrow} (\mathsf{loan}^s\, \ell\, v', \Omega')}$$

$$\text{W-}\textsc{Deref-Box}\ \frac{\Omega \vdash P(v) \leftarrow w \overset{\mathsf{k}}{\Rightarrow} (v', \Omega')}{\Omega \vdash P(*(\mathsf{Box}\, v)) \leftarrow w \overset{\mathsf{k}}{\Rightarrow} (\mathsf{Box}\, v', \Omega')}$$

$$\text{W-}\textsc{Deref-SharedBorrow}\ \frac{\mathsf{loan}^s\, \ell\, v \in \Omega \qquad \Omega \vdash P(\mathsf{loan}^s\, \ell\, v) \leftarrow w \overset{\mathsf{imm}}{\Rightarrow} (v', \Omega'[\mathsf{loan}^s\, \ell\, v'']) \qquad \Omega'' = \Omega'[\mathsf{loan}^s\, \ell\, v']}{\Omega \vdash P(*(\mathsf{borrow}^s\, \ell)) \leftarrow w \overset{\mathsf{imm}}{\Rightarrow} (\mathsf{borrow}^s\, \ell, \Omega'')}$$

$$\text{W-}\textsc{Deref-MutBorrow}\ \frac{\Omega \vdash P(v) \leftarrow w \overset{\mathsf{imm},\mathsf{mut}}{\Rightarrow} (v', \Omega')}{\Omega \vdash P(*(\mathsf{borrow}^m\, \ell\, v)) \leftarrow w \overset{\mathsf{imm},\mathsf{mut}}{\Rightarrow} (\mathsf{borrow}^m\, \ell\, v', \Omega')}$$

Figure 9.4: Write Rules (LLBC)

$\Omega \vdash P(v_x) \leftarrow v \overset{\mathsf{k}}{\Rightarrow} (v'_x, \Omega')$, which from v_x computes an updated value v'_x where only the sub-expression selected by P is updated with v, together with an updated environment Ω'. We update x's entry in the environment to map to the new value v'_x, denoted $\Omega[x \mapsto v'_x]$ (Write). Importantly, the auxiliary judgement needs to compute both an updated value *and* an updated environment, because following shared borrows might require looking up and thus updating a value which is not the one under scrutinee. Also note that the writing judgement is used to evaluate assignments (E-Assign) but also to do the necessary book-keeping when, e.g., evaluating right-values. For instance, creating a shared borrow requires using the writing judgement to update the environment by inserting a shared loan at the proper place (E-SharedBorrow). As before, the shape of P determines which rule applies: we may only dereference borrows or boxes (W-Deref-SharedBorrow, W-Deref-MutBorrow, W-Deref-Box) and eventually apply W-Base. We elide the remaining rules (tuples, sums, etc.).

9.4 Semantics of Ownership and Borrows

At the heart of our operational semantics is our treatment of borrows, which captures ownership transfer.

9.4.1 Right-values

We now introduce the rules to evaluate rvalues in Figure 9.5, and describe in detail the essential operations: moves, copies and borrows. Our rules start with E-, for evaluation rules, and the indications **LLBC only** can be ignored for now.

When moving a value at place p (E-Move), we simply replace this value with $\bot$. We disallow moving: $\bot$, already-borrowed values (no loans), or reserved borrows. We also forbid moving *through* a dereference. The former prevents invalidating stray borrows; simply said, if a value has non-terminated borrows, we cannot obtain full ownership of it in order to perform the move. The latter replicates Rust's constraint that no moves are allowed under a borrow.

For copies (E-Copy), we rely on an auxiliary judgment of the form $\vdash \mathsf{copy}\, v = v'$, meaning copying v in produces v'. Copying a value produces the same value except for shared loans, in which case we simply copy the shared value without performing any shared-loan tracking (Copy-SharedLoan); the ownership information that regards the old value is irrelevant for the newly-copied value. The copy judgement constrains the copyable values by omission: we allow copying, e.g., a shared borrow (Copy-SharedBorrow), but disallow copying mutable or reserved borrows, mutably

E-MutBorrow **(LLBC only)**

$$\frac{\vdash \Omega(p) \overset{\mathsf{mut}}{\Rightarrow} v \qquad \bot, \mathsf{loan}^{s,m}, \mathsf{borrow}^{r} \notin v \qquad \ell \text{ fresh} \qquad \vdash \Omega[p \leftarrow \mathsf{loan}^{m}\,\ell] \overset{\mathsf{mut}}{\Rightarrow} \Omega'}{\Omega \vdash \&\mathsf{mut}\,p \Downarrow (\mathsf{borrow}^{m}\,\ell\ v,\ \Omega')}$$

E-SharedBorrow **(LLBC only)**

$$\frac{\vdash \Omega(p) \overset{\mathsf{imm}}{\Rightarrow} v \qquad \bot, \mathsf{loan}^{m}, \mathsf{borrow}^{r} \notin v \qquad \vdash \Omega[p \leftarrow v'] \overset{\mathsf{imm}}{\Rightarrow} \Omega' \qquad v' = \begin{cases} \mathsf{loan}^{s}\,\ell\,v'' & \text{if } v = \mathsf{loan}^{s}\,\ell\,v'' \\ \mathsf{loan}^{s}\,\ell\,v & \ell \text{ fresh otherwise} \end{cases}}{\Omega \vdash \&\,p \Downarrow (\mathsf{borrow}^{s}\,\ell,\ \Omega')}$$

E-ReservedBorrow **(LLBC only)**

$$\frac{\vdash \Omega(p) \overset{\mathsf{mut}}{\Rightarrow} v \qquad \bot, \mathsf{loan}^{m}, \mathsf{borrow}^{r} \notin v \qquad \vdash \Omega[p \leftarrow v'] \overset{\mathsf{mut}}{\Rightarrow} \Omega' \qquad v' = \begin{cases} \mathsf{loan}^{s}\,\ell\,v'' & \text{if } v = \mathsf{loan}^{s}\,\ell\,v'' \\ \mathsf{loan}^{s}\,\ell\,v & \ell \text{ fresh otherwise} \end{cases}}{\Omega \vdash \&\mathsf{reserved}\,p \Downarrow (\mathsf{borrow}^{r}\,\ell,\ \Omega')}$$

E-Move

$$\frac{\vdash \Omega(p) \overset{\mathsf{mov}}{\Rightarrow} v \qquad \bot, \mathsf{loan}^{s,m}, \mathsf{borrow}^{r} \notin v \qquad \vdash \Omega[p \leftarrow \bot] \overset{\mathsf{mov}}{\Rightarrow} \Omega'}{\Omega \vdash \mathsf{move}\,p \Downarrow (v,\ \Omega')}$$

E-Copy

$$\frac{\vdash \Omega(p) \overset{\mathsf{imm}}{\Rightarrow} v \qquad \vdash \mathsf{copy}\,v = v'}{\Omega \vdash \mathsf{copy}\,p \Downarrow (v',\ \Omega)}$$

E-Pair

$$\frac{\Omega_0 \vdash op_0 \Downarrow (v_0,\ \Omega_1) \qquad \Omega_1 \vdash op_1 \Downarrow (v_1,\ \Omega_2)}{\Omega_0 \vdash (op_0, op_1) \Downarrow ((v_0, v_1),\ \Omega_2)}$$

E-Sum-Left

$$\frac{\Omega \vdash op \Downarrow (v,\ \Omega')}{\Omega \vdash \mathsf{Left}\ op \Downarrow (\mathsf{Left}\,v,\ \Omega')}$$

E-Sum-Right

$$\frac{\Omega \vdash op \Downarrow v \dashv \Omega'}{\Omega \vdash \mathsf{Right}\ op \Downarrow (\mathsf{Right}\,v,\ \Omega')}$$

Figure 9.5: LLBC: Rules to Evaluate Rvalues

$$\textsc{Copy-SharedBorrow}\ \frac{}{\vdash \mathsf{copy}\ \mathsf{borrow}^s\ \ell = \mathsf{borrow}^s\ \ell} \qquad \textsc{Copy-SharedLoan}\ \frac{\vdash \mathsf{copy}\ v = v'}{\vdash \mathsf{copy}\ \mathsf{loan}^s\ \ell\ v = v'}$$

$$\textsc{Copy-Scalar}\ \frac{v \text{ literal value (boolean, integer, etc.)}}{\vdash \mathsf{copy}\ v = v} \qquad \textsc{Copy-Sum}\ \frac{C = \mathsf{Left} \ \vee\ C = \mathsf{Right} \quad \vdash \mathsf{copy}\ v = v'}{\vdash \mathsf{copy}\ C\ v = C\ v'}$$

$$\textsc{Copy-Pair}\ \frac{\vdash \mathsf{copy}\ v_0 = v_0' \quad \vdash \mathsf{copy}\ v_1 = v_1'}{\vdash \mathsf{copy}\ (v_0,\ v_1) = (v_0',\ v_1')}$$

Figure 9.6: LLBC: Rules to Evaluate Copy

loaned values, boxes or $\bot$, as there are no corresponding rules.

For mutable borrows (E-MutBorrow), we disallow: borrowing already-borrowed values (no $\mathsf{loan}^{s,m}$); borrowing moved, uninitialized values (no $\bot$) or reserved borrows; and borrowing through a shared borrow, as constrained by the use the the mov capability (R-Deref-SharedBorrow, R-Deref-MutBorrow). If these premises are satisfied, we update the environment by marking p as loaned with identifier ℓ. The borrow evaluates to $\mathsf{borrow}^m\ \ell\ v$, which embodies exclusive access to value v thanks to the loan ℓ.

For immutable borrows (E-SharedBorrow), we disallow moved or uninitialized values (no $\bot$) and reserved borrows, but rule out mutable loans only: it is always legal in Rust to create another shared borrow from a value that has already been shared. The borrow evaluates to $\mathsf{borrow}^s\ \ell$, a borrow with a non-exclusive access granted by ℓ, where ℓ may have two origins. Either the value at place p is already immutably borrowed for some loan ℓ, in which case we simply reuse this identifier. Or this value is not borrowed, in which case we update the environment to mark the value as shared by inserting a fresh loan identifier ℓ.

Reserved (two-phase) borrows (E-ReservedBorrow) work similarly to immutable borrows, except that we use the capability mut instead of imm, as those borrows may later be promoted to full mutable borrows.

9.4.2 Statements

We are now ready to define the semantics of statements (Figures 9.7, 9.8, 9.9).

We start with the assignments (E-Assign). We reduce rvs first, and remark that to obtain v, the various rules for the rv syntactic category must succeed. For instance, if

$$\textsc{E-Reorg}\ \frac{\Omega_0 \hookrightarrow \Omega_1 \qquad \Omega_1 \vdash s \rightsquigarrow (r, \Omega_2)}{\Omega_0 \vdash s \rightsquigarrow (r, \Omega_2)} \qquad \textsc{E-Return}\ \frac{}{\Omega \vdash \mathsf{return} \rightsquigarrow (\mathsf{return}, \Omega)} \qquad \textsc{E-Panic}\ \frac{}{\Omega \vdash \mathsf{panic} \rightsquigarrow (\mathsf{panic}, \Omega)}$$

$$\textsc{E-Seq-Unit}\ \frac{\Omega_0 \vdash s_0 \rightsquigarrow ((), \Omega_1) \quad \Omega_1 \vdash s_1 \rightsquigarrow (r, \Omega_2)}{\Omega_0 \vdash s_0; s_1 \rightsquigarrow (r, \Omega_2)} \qquad \textsc{E-Break}\ \frac{}{\Omega \vdash \mathsf{break}\, i \rightsquigarrow (\mathsf{break}\, i, \Omega)} \qquad \textsc{E-Continue}\ \frac{}{\Omega \vdash \mathsf{continue}\, i \rightsquigarrow (\mathsf{continue}\, i, \Omega)}$$

$$\textsc{E-Seq-Propagate}\ \frac{r \in \{\mathsf{return}, \mathsf{panic}, \mathsf{continue}\, i, \mathsf{break}\, i\} \quad \Omega \vdash s_0 \rightsquigarrow (r, \Omega')}{\Omega \vdash s_0; s_1 \rightsquigarrow (r, \Omega')}$$

$$\textsc{E-IfThenElse-T}\ \frac{\Omega \vdash op \Downarrow (v, \Omega') \quad v = \mathsf{true} \vee v = \mathsf{loan}^s\, \ell\, \mathsf{true} \quad \Omega' \vdash s_1 \rightsquigarrow (r, \Omega'')}{\Omega \vdash \mathsf{if}\, op\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2 \rightsquigarrow (r, \Omega'')}$$

$$\textsc{E-IfThenElse-F}\ \frac{\Omega \vdash op \Downarrow (v, \Omega') \quad v = \mathsf{false} \vee v = \mathsf{loan}^s\, \ell\, \mathsf{false} \quad \Omega' \vdash s_2 \rightsquigarrow (r, \Omega'')}{\Omega \vdash \mathsf{if}\, op\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2 \rightsquigarrow (r, \Omega'')}$$

$$\textsc{E-Match}\ \frac{\vdash \Omega(p) \overset{\mathsf{imm}}{\Rightarrow} v \qquad v = C\, v' \vee v = \mathsf{loan}^s\, \ell\, (C\, v') \qquad (C = \mathsf{Left} \wedge s = s_1) \vee (C = \mathsf{Right} \wedge s = s_2) \qquad \Omega \vdash s \rightsquigarrow (r, \Omega')}{\Omega \vdash (\mathsf{match}\, p\, \mathsf{with} \mid \mathsf{Left}\ \Rightarrow s_1 \mid \mathsf{Right}\ \Rightarrow s_2) \rightsquigarrow (r, \Omega')}$$

$$\textsc{E-Assign}\ \textbf{(LLBC only)}\ \frac{\Omega \vdash rv \Downarrow (v, \Omega') \qquad \vdash \Omega'(p) \overset{\mathsf{mut}}{\Rightarrow} v_p \qquad v_p \text{ has no outer } \mathsf{loan}^{s,m} \qquad \vdash \Omega'[p \leftarrow v] \overset{\mathsf{mut}}{\Rightarrow} \Omega'' \qquad \Omega''' = \Omega'', \ \_ \rightarrow v_p}{\Omega \vdash p := rv \rightsquigarrow ((), \Omega''')}$$

$$\textsc{E-Box-New}\ \textbf{(LLBC only)}\ \frac{\Omega \vdash op \Downarrow (v, \Omega')}{\Omega \vdash \mathsf{new}\, op \rightsquigarrow (\mathsf{Box}\, v, \Omega')} \qquad \textsc{E-Box-Free}\ \frac{\vdash \Omega(p) \overset{\mathsf{mov}}{\Rightarrow} \mathsf{Box}\, v \quad \Omega \vdash *p := \bot \Downarrow ((), \Omega') \quad \vdash \Omega[p \leftarrow \bot] \overset{\mathsf{mov}}{\Rightarrow} \Omega''}{\Omega \vdash \mathsf{free}\, p \rightsquigarrow ((), \Omega'')}$$

Figure 9.7: LLBC: Rules to Evaluate Statements

the right-hand side is a move, then E-Move enforces all of its preconditions. This means E-Assign operates with ownership of v, which maps to our intuition for assignments in the presence of ownership and, naturally, also corresponds to the Rust semantics. What we do enforce, however, is that the value v_p found at place p (on the left-hand side of the assignment) should not have any outer loans, where an outer loan is a loan value which is itself not inside a (mutable) borrow (for instance, $(0, \mathsf{loan}^m\,\ell)$ contains an outer loan, while $\mathsf{borrow}^m\,\ell\,(\mathsf{loan}^m\,\ell')$ doesn't). Overwriting a value that is currently loaned-out would violate safety; we need to rule this out. More precisely: loans may only appear behind borrow indirections; the value itself that is being overwritten may not contain any loan. The assignment rule ends by writing v at the new place (using Write). The rule for function call (E-Call) is identical, except that it deals with binding the arguments, locals and return variable.

One key point of E-Assign is that we retain the old value in the environment Ω'', inside an anonymous variable _ not accessible to the user-written program. The only purpose of this variable is to avoid discarding useful ownership knowledge; operationally, this operation is ghost: the need to retain old values disappears once we explicitly model the memory, as we will see in the proofs of soundness.

Going back to the reborrow example from Section 9.1.3, the possibility of writing to a value containing an *inner* loan is what allows us to write into px at line 3. On the other hand, the proof of soundness requires us to forbid overriding an *outer* loan. Finally, storing the old value of px inside an anonymous variable is crucial to not lose information about the borrow graph.

```
1 let mut x = 0;          // x ↦ 0
2 let mut px = &mut x;    // x ↦ loan^m ℓ,   px ↦ borrow^m ℓ 0
3 px = &mut (*px);        // x ↦ loan^m ℓ,   _ ↦ borrow^m ℓ (loan^m ℓ'),   px ↦ borrow^m ℓ '0
4                         // After ending ℓ':
5                         // x ↦ loan^m ℓ,   _ ↦ borrow^m ℓ 0,             px ↦ ⊥
6                         // After ending ℓ:
7 assert!(x==0);          // x ↦ 0,          _ ↦ ⊥,                        px ↦ ⊥
```

Most of the remaining rules for statements are straightforward; we illustrate matches and skip the rest. For matches (E-Match), we peek at the enum tag via $\overset{\mathsf{imm}}{\Rightarrow}$; actual transfer of ownership with moves and copies takes place in the suitable branch while executing the statement s. We note that in general, our Read judgment may return values of the form loan^s: this is useful e.g., to enforce that a value is not loaned out, as in the premise of E-Move. For matches, however, we merely need to read the enum tag; hence the case disjunction on the shape of the scrutinee in the premise. At this point it is useful to note that the rules to create shared borrows were designed so that it is not possible to stack an arbitrary number of shared loans.

E-PushStack
$$\frac{\Omega' = \{\ \Omega \text{ with } \mathsf{env} = [\overrightarrow{x \to v}] \mathbin{+\!\!+} \Omega.\mathsf{env},\ \mathsf{stack} = [\overrightarrow{x}] :: \Omega.\mathsf{stack}\ \}}{\vdash \mathsf{push\_stack}\ [\overrightarrow{x \to v}]\ \Omega\ =\ \Omega'}$$

E-PopStack
$$\frac{\begin{array}{c}\Omega_0.\mathsf{stack} = ([x_{ret}] \mathbin{+\!\!+} [\overrightarrow{x_i}]) :: \mathsf{stack}' \\ \forall\, i, \Omega_i \vdash x_i := \bot \Downarrow ((), \Omega_{i+1}) \qquad \Omega_m.env = [x_{ret} \to v_{ret}] \mathbin{+\!\!+} [\overrightarrow{x_i \to v_i}] \mathbin{+\!\!+} \mathsf{env}' \\ \bot, \mathsf{loan}, \mathsf{borrow}^r \notin v_{ret} \qquad \Omega_{end} = \{\ \Omega_m \text{ with } \mathsf{stack}\ =\ \mathsf{stack}',\ \mathsf{env}\ =\ \mathsf{env}'\ \}\end{array}}{\vdash \mathsf{pop\_stack}\ \Omega\ =\ (v_{ret},\ \Omega_{end})}$$

E-Call
$$\frac{\begin{array}{c}\overrightarrow{\rho} \text{ fresh} \\ f\langle \_^{\rightarrow}, \vec{\tau}\rangle = \mathsf{fn}\ \langle \_^{\rightarrow}\rangle\ (\overrightarrow{x_i} : \overrightarrow{\tau_i})\ (\overrightarrow{y_j} : \overrightarrow{\tau_j})\ (x_{\mathsf{ret}} : \tau)\ \{\ s\ \} \qquad \forall\, j \in [0; m[, \Omega_j \vdash op_j \Downarrow (v_j, \Omega_{j+1}) \\ \vdash \mathsf{push\_stack}\ \left([x_{\mathrm{ret}} \to \bot] \mathbin{+\!\!+} [\overrightarrow{x_j \to v_j}] \mathbin{+\!\!+} [\overrightarrow{y_k \to \bot}]\right) \Omega_m = \Omega_{begin} \\ \Omega_{\mathrm{begin}} \vdash body \rightsquigarrow (r, \Omega_{\mathrm{end}}) \\ (r', \Omega_f) = \begin{cases} (\mathsf{panic}, \Omega_{\mathrm{end}}) & \text{if } r = \mathsf{panic} \\ ((), \Omega'') & \text{if } r = \mathsf{return} \wedge \vdash \mathsf{pop\_stack}\ \Omega_{\mathrm{end}} = (v,\ \Omega')\ \wedge\ \Omega' \vdash p := v \rightsquigarrow ((), \Omega'') \end{cases}\end{array}}{\Omega_0 \vdash p := f\langle \_^{\rightarrow}, \vec{\tau}\rangle(\overrightarrow{op_j}) \rightsquigarrow (r', \Omega_f)}$$

Figure 9.8: LLBC: Rules to Evaluate Function Calls

E-Loop-Break-Inner
$$\frac{\Omega \vdash s \rightsquigarrow (\mathsf{break}\,0,\ \Omega)}{\Omega \vdash \mathsf{loop}\,s \rightsquigarrow ((),\ \Omega)}$$

E-Loop-Break-Outer
$$\frac{\Omega \vdash s \rightsquigarrow (\mathsf{break}\,i+1,\ \Omega)}{\Omega \vdash \mathsf{loop}\,s \rightsquigarrow (\mathsf{break}\,i,\ \Omega)}$$

E-Loop-Continue-Inner
$$\frac{\Omega \vdash s \rightsquigarrow (\mathsf{continue}\,0,\ \Omega') \qquad \Omega' \vdash \mathsf{loop}\,s \rightsquigarrow (r,\ \Omega'')}{\Omega \vdash \mathsf{loop}\,s \rightsquigarrow (r,\ \Omega'')}$$

E-Loop-Continue-Outer
$$\frac{\Omega \vdash s \rightsquigarrow (\mathsf{continue}\,i+1,\ \Omega)}{\Omega \vdash \mathsf{loop}\,s \rightsquigarrow (\mathsf{continue}\,i,\ \Omega)}$$

E-Loop-Panic
$$\frac{\Omega \vdash s \rightsquigarrow (\mathsf{panic},\ \Omega)}{\Omega \vdash \mathsf{loop}\,s \rightsquigarrow (\mathsf{panic},\ \Omega)}$$

E-Loop-Return
$$\frac{\Omega \vdash s \rightsquigarrow (\mathsf{return},\ \Omega)}{\Omega \vdash \mathsf{loop}\,s \rightsquigarrow (\mathsf{return},\ \Omega)}$$

Figure 9.9: LLBC: Additional Rules to Evaluate Loops

$$\textsc{Reorg-End-MutBorrow}\ \textbf{(LLBC only)}\quad \frac{\text{hole of } \Omega[\mathsf{loan}^m\ \ell,\ .] \text{ not inside a borrowed value} \qquad \mathsf{loan}^{s,m} \notin v}{\Omega[\mathsf{loan}^m\ \ell,\ \mathsf{borrow}^m\ \ell\ v] \hookrightarrow \Omega[v, \bot]}$$

$$\textsc{Reorg-End-SharedReservedBorrow}\ \textbf{(LLBC only)}\quad \frac{\text{hole of } \Omega[.] \text{ not inside a borrowed value} \qquad \mathsf{loan}^s\ \ell\ v' \in \Omega[v] \qquad v = \mathsf{borrow}^{s,r}\ \ell}{\Omega[v] \hookrightarrow \Omega[\bot]}$$

$$\textsc{Reorg-End-SharedLoan}\quad \frac{\mathsf{borrow}^{s,r}\ \ell \notin \Omega[\mathsf{loan}^s\ \ell\ v]}{\Omega[\mathsf{loan}^s\ \ell\ v] \hookrightarrow \Omega[v]}$$

$$\textsc{Reorg-Activate-Reserved}\ \textbf{(LLBC only)}\quad \frac{\mathsf{loan}, \mathsf{borrow}^r \notin v \qquad \text{hole of } \Omega[., \mathsf{borrow}^r\ \ell] \text{ not inside a shared value} \qquad \ell \notin \Omega[., .], v}{\Omega[\mathsf{loan}^s\ \ell\ v, \mathsf{borrow}^r\ \ell] \hookrightarrow \Omega[\mathsf{loan}^m\ \ell, \mathsf{borrow}^m\ \ell\ v]}$$

$$\textsc{Reorg-Seq}\quad \frac{\Omega_0 \hookrightarrow \Omega_1 \qquad \Omega_2 \hookrightarrow \Omega_2}{\Omega_0 \hookrightarrow \Omega_2}$$

$$\textsc{Reorg-None}\quad \frac{}{\Omega \hookrightarrow \Omega}$$

Figure 9.10: LLBC: Reorganization Rules

9.5 Reorganizing Environments and Terminating Borrows

We now present the final conceptual portion of our operational semantics: reorganizing the environment, which we used in our earlier examples to terminate borrows. We present rules in a declarative style, to highlight the *semantics* of Rust as opposed to the *implementation* of borrow-checking. A consequence of our declarative approach is that we do not need to follow Rust's behavior to the letter; rather, in the implementation, we reorganize borrows in a lazy fashion, and don't terminate a borrow unless we need to get the borrowed value back. Concretely, our rules allow reorganization before every statement (E-Reorg). We claim that this captures a general semantics of borrows; we substantiate that claim by showing, in Chapter 14, how our semantics can validate a Rust program rejected by the current Rust borrow checker but accepted by Polonius, an ongoing rewrite of the borrow checker to allow for a larger class of Rust programs to be accepted.

We define reorganizing via a set of rewriting rules that operate on the environment Ω (Figure 9.10). Since these rules are syntactic in nature, we rely on value contexts $V[v]$ and environment contexts $\Omega[v]$, rather than our earlier semantic notions of reads,

writes and ghost updates. We omit administrative rules for re-ordering environments at will. Our judgments are of the form $\Omega \hookrightarrow \Omega'$, meaning Ω may be reorganized into Ω'. We indulge in some syntax overload; whenever used on the left-hand side of $\hookrightarrow$, we understand $\Omega[x \mapsto v]$ to pattern-match on Ω to select a mapping. This considerably simplifies notation.

Our rules either render a value unusable ($\bot$), or strengthen it (borrow^m, in the case of reserved borrows) For these reasons, we generally demand full ownership of the value in $V[.]$ (or $\Omega[.]$); this is the reason behind the premises of the shape "hole of $\Omega[.]$ not inside a borrowed value". The hole of $\Omega[.]$ is not inside a borrowed value if $\Omega[.]$ is not of the shape $\Omega[.] = \Omega'[\mathsf{borrow}^m\, \ell\ (V[.])]$ or $\Omega[.] = \Omega'[\mathsf{loan}^s\, \ell\, (V[.])]$ (for some $\Omega'[.]$, $V[.]$ and ℓ). For instance, Reorg-End-MutBorrow states that we can end a mutable borrow which is not inside a mutable borrow or a shared loan; i.e., if it is not itself borrowed. Similarly, the hole of $\Omega[.]$ is not inside a shared value if $\Omega[.]$ is not of the shape $\Omega[.] = \Omega'[\mathsf{loan}^s\, \ell\, V[.]]$ (for some $\Omega'[.]$, $V[.]$ and ℓ). We use this in Reorg-Activate-Reserved to forbid activating a reserved borrow if its corresponding loan is itself immutably borrowed, as otherwise it would allow us to mutate an immutably borrowed value. Doing so, we precisely capture the constraints of Rust with regards to reborrows. We now review the rules.

When ending a shared borrow (Reorg-End-SharedReservedBorrow), we render the borrow unusable henceforth, and replace it with $\bot$. The rule also applies to reserved borrows which haven't been promoted to mutable borrows. When ending a shared loan (Reorg-End-SharedLoan), we check that there doesn't remain any corresponding borrows in the environment, then remove the shared loan marker.

When ending a mutable borrow (Reorg-End-MutBorrow), we enforce that we own the value we are about to return (i.e., there are no loans inside). The borrow then becomes unusable by being replaced with $\bot$, and the borrowed value is returned to its rightful owner.

This high-level approach to the Rust semantics allows us to very naturally account for an oft-used Rust feature, namely two-phase borrows, which are introduced in many places when desugaring to MIR and enable a variety of very common idioms [404] without resorting to more advanced desugarings. We account for those through what we call *reserved* borrows. Reserved borrows are created just like shared borrows (E-ReservedBorrow). However, reserved borrows cannot be copied, dereferenced, or written into. Therefore, the only way to use a reserved borrow is to strengthen it into a mutable borrow, which is legal, as long as all other (shared or reserved) borrows have ended (Reorg-Activate-Reserved).

These rules are declarative and non-ordered; in practice, our tool performs a syntax-

directed reorganization guided by the various preconditions on our rules. For instance, whenever $\mathsf{loan}^{s,m} \notin v$ appears as a premise, we perform a traversal of v to end whichever loans we encounter.

Chapter 10

Symbolic Semantics (LLBC$^{\#}$)

Our semantics allows us to keep track of borrows and ownership in an exact fashion. We now ask: if we adopt a modular approach and treat function calls as opaque, how much can we leverage borrows and regions to still enable precise tracking of ownership and aliasing? We answer that question with a region-centric shape analysis that abstracts away the effect of a function call on the ownership graph, via a notion of *region abstraction*. We dub the result our "symbolic semantics", or LLBC$^{\#}$; it is, obviously, less precise than our earlier concrete semantics; yet, it contains enough ownership and aliasing information that we can generate a functional translation from it. Our symbolic semantics very much resembles the concrete semantics; this time, however, we turn out attention to the region information provided by function signatures to abstract away subsets of the ownership graph.

In this section, we introduce a few additional restrictions on the subset of Rust we can handle. We temporarily assume every disjunction in the control-flow is in terminal position; this is a strong restriction, which in practice requires duplicating the continuation of conditionals and matches, but that we will lift in Section 12.1. We also temporarily assumed the code doesn't contain loops; we defer the support for loops in the symbolic semantics to Section 12.2; this first version of the semantics *does*, however, support recursive functions. We disallow nested borrows in function *signatures*, but users can still manipulate arbitrarily nested borrows within function bodies. We also disallow instantiating a polymorphic function with a type argument that contains a borrow, and do not allow type declarations that contain borrows. We believe most of the latter issues can be addressed with moderate modifications to LLBC$^{\#}$, and leave them as future work. Finally, let us reiterate the fact that we do not model traits in our semantics, though traits are supported by the implementation.

10.1 Symbolic Semantics by Example

10.1.1 Symbolic Values and Matches

A first concept we need to add to our toolkit is that of a symbolic value, that is, a value which is not statically known; we write $(\sigma : \tau)$ to denote a symbolic value σ of type τ. We now illustrate how symbolic values behave, notably in the presence of `match`es, which refine our static knowledge.

```
1  fn f(mut o: Option<i32>) {
2    // Initial state:
3    // o ↦ (σ : Option i32)
4    let po = &mut o;
5    // o ↦ loan^m ℓ;   po ↦ borrow^m ℓ (σ : Option i32)
6
7    match *po {
8      None => {
9        // o ↦ loan^m ℓ;   po ↦ borrow^m ℓ (None)
10       panic!();
11     }
12
13     Some => {
14       // o ↦ loan^m ℓ;   po ↦ borrow^m ℓ (Some (σ' : i32))
15       let r = &mut (*po).Some.0;
16       // o ↦ loan^m ℓ;   po ↦ borrow^m ℓ (Some (loan^m ℓ'));   r ↦ borrow^m ℓ' (σ' : i32)
17       *r = 1;
18       // o ↦ loan^m ℓ;   po ↦ borrow^m ℓ (Some (loan^m ℓ'));   r ↦ borrow^m ℓ' 1
19       *po = None;
20       // o ↦ loan^m ℓ;   po ↦ borrow^m ℓ (None);   r ↦ ⊥
21     }
22   }
23 }
```

Our semantics only models pairs and sums; in the snippet of code above Option i32 is syntactic sugar for i32 + (), while Some and None are sugar for Left and Right, respectively. The symbolic value σ stands in for the function parameter whose concrete value is unknown at run-time; σ can be borrowed mutably like any regular value (line 4).

At line 7, we perform a case analysis; at this stage, all we know is that the scrutinee ∗po evaluates to the symbolic value σ, of the correct type Option i32. In order to check the branches, we treat each one of them individually, in each case refining σ with a more precise value according to the constructor of the branch. Simply said, in the None case, we replace every occurrence of σ with None, and in the Some case, we replace every occurrence of σ with Some $(\sigma' : \text{i32})$, where σ' is a fresh symbolic value.

More interesting pointer manipulations follow in the Some branch. We borrow the value within the option via r, using a projector syntax inspired by MIR's internal representation of projectors. This borrowing incurs no loss in precision in our alias tracking: because we refined σ earlier, we know that *both* o and po are unusable as long as r lives.

Importantly, this precise tracking of the borrow graph forbids changing the enum variant of o, while its value or one of its fields is borrowed: this disallows leftover borrows pointing to data of the wrong (previous) type, which would make our execution unsound. More precisely, when the user changes the enum variant at line 19, our symbolic semantics requires to give up ownership of r, in order to regain $po \mapsto \mathsf{borrow}^m\ \ell\ (\mathsf{Some}\ 1)$, which by virtue of containing no "outer" loans makes the update valid.

10.1.2 Function Calls: Single Region

We now switch from the callee to the caller's perspective, and turn our attention to function calls. We introduce a new concept of *region abstractions* to our borrow graph. An abstraction owns borrows and loans, but does so abstractly; that is, we have no aliasing information about values in an abstraction, and in particular we don't know which borrow corresponds to which loan. Region abstractions allow us to retain ownership and aliasing information in the presence of function calls; they are introduced when a call takes place, upon which they assume ownership of the call's arguments; they are terminated whenever the caller relinquishes ownership of (part of) the return value, upon which ownership flows back to the original arguments.

Before modifying the semantics from Chapter 9, we illustrate region abstractions with an example. We revisit our earlier test_choose function (Section 8.2).

```
1  let mut x = 0;
2  let mut y = 0;
3  let px = &mut x;
4  let py = &mut y;
5  // x ↦ loan^m ℓ_x,   y ↦ loan^m ℓ_y,   px ↦ borrow^m ℓ_x 0,   py ↦ borrow^m ℓ_y 0
6  let pz = choose(true, move px, move py);
7  // x ↦ loan^m ℓ_x,   y ↦ loan^m ℓ_y,   px ↦ ⊥,   py ↦ ⊥,
8  // A(ρ) {  borrow^m ℓ_x 0,   borrow^m ℓ_y 0,   loan^m ℓ_r  }
9  // pz ↦ borrow^m ℓ_r (σ : u32),
10 *pz = *pz + 1;
11 // x ↦ loan^m ℓ_x,   y ↦ loan^m ℓ_y,   px ↦ ⊥,   py ↦ ⊥,
12 // A(ρ) {  borrow^m ℓ_x 0,   borrow^m ℓ_y 0,   loan^m ℓ_r  }
13 // pz ↦ borrow^m ℓ_r (σ' : u32)
14
15 // Step 1:
```

```
16 // x ↦ loan^m ℓ_x,   y ↦ loan^m ℓ_y,   px ↦ ⊥,   py ↦ ⊥,
17 // A(ρ) {   borrow^m ℓ_x 0,   borrow^m ℓ_y 0,   σ'   }
18 // pz ↦ ⊥
19
20 // Step 2:
21 // x ↦ loan^m ℓ_x,   y ↦ loan^m ℓ_y,   px ↦ ⊥,   py ↦ ⊥,
22 // _ ↦ borrow^m ℓ_x σ_x,   _ ↦ borrow^m ℓ_y σ_y
23 // pz ↦ ⊥
24
25 // Step 3:
26 // x ↦ σ_x,   y ↦ loan^m ℓ_y,   px ↦ ⊥,   py ↦ ⊥,
27 // _ ↦ ⊥,   _ ↦ borrow^m ℓ_y σ_y
28 // pz ↦ ⊥
29 assert!(x == 1);
```

Up to line 4, the usual set of rules apply and yield an environment that is consistent with Chapter 9. Our abstract rules come in at line 6, where we are faced with a function call. We now need to abstract the call, that is, precisely capture how the function call affects the borrow graph, without looking at the definition of the function itself. To do so, we have only one piece of information at our disposal: the type of choose, namely $\mathsf{fn}\langle\rho\rangle(\mathsf{bool}, \&^{\rho}\mathsf{mut}\ \mathsf{u32}, \&^{\rho}\mathsf{mut}\ \mathsf{u32}) \to \&^{\rho}\mathsf{mut}\ \mathsf{u32}$.

The type of choose conveys two key pieces of information: first, it *consumes* two mutable borrows in order to *produce* a fresh (abstract) return value; second, the borrows and the return value belong to the same *region* ρ. This region gives us the constraint that, for as long as the output borrow is alive, we have to consider that the input borrows as alive; as such it gives us a summary of the function. We proceed as follows. We allocate a fresh region abstraction $A(\rho)$, which owns the consumed arguments pertaining to region ρ; in our case, $\mathsf{borrow}^m\ \ell_x\ 0$ and $\mathsf{borrow}^m\ \ell_y\ 0$. (In the case of multiple regions appearing inside the function type, we need to project the ownership of the arguments along their respective regions; we handle this case formally in Section 10.2.) We know that the return value pz has type $\&^{\rho}\mathsf{mut}\ \mathsf{u32}$; furthermore, the region in the type tells us that the owner of this abstract value is the abstraction $A(\rho)$. Necessarily, pz has a value of the shape $\mathsf{borrow}^m\ \ell_r\ (\sigma : \mathsf{u32})$, for some fresh loan identifier ℓ_r and *symbolic value* σ; because the borrow ℓ_r belongs to region ρ, its corresponding loan is placed inside the region abstraction $A(\rho)$. We obtain the environment at lines 7-9, where the ownership of both px and py has been transferred to the region abstraction, and where pz has full ownership of a value loaned from the region abstraction. Intuitively, a region abstraction is a bag containing borrows (what has been consumed) and loans (what has been produced).

At line 10, the mutation type-checks, and does not affect the abstract environment:

the symbolic value σ borrowed through pz is simply replaced by a fresh symbolic value σ' stemming from the addition. At that stage, we cannot read from x since it is mutably loaned; we therefore need to reorganize the environment to make the assertion at line 29 succeed. Since we do not have any precise knowledge about the aliasing relationship between x, y and pz, we cannot return ownership to x directly; we must return ownership *en masse* by terminating region $A(\rho)$. We do so by terminating the borrow for pz, which returns the abstract value σ' to $A(\rho)$ (step 1, lines 15-18). Now that $A(\rho)$ has no outstanding loans left, we can terminate $A(\rho)$ itself. This reintroduces in the environment borrows l_x and l_y with fresh values, and replaces the borrowed values they held (0 in both cases) with fresh symbolic values to account for potential modifications (lines 20-23). These borrows are placed in anonymous variables, i.e. they are not directly accessible to the user; they once again solely ensure we do not lose ownership information. A final reorganization of the environment terminates ℓ_x, and makes x usable again (lines 25-28).

Discussion. We see our region abstractions as a form of magic wands; a function call consumes part of the memory, and returns a magic wand (the region abstraction) along with its argument (the returned value). Regaining ownership of the consumed memory requires applying the magic wand to its argument, hence surrendering access to the returned value.

10.1.3 Function Calls: Multiple Regions

We now study a call to the swap function, which permutes the two components of a single tuple, located in two different regions.

$$\mathsf{fn\ swap}\langle\alpha,\beta\rangle(z : (\&^{\alpha}\mathsf{mut\ u32}, \&^{\beta}\mathsf{mut\ u32})) \to (\&^{\beta}\mathsf{mut\ u32}, \&^{\alpha}\mathsf{mut\ u32})$$

We examine a call `let r = swap (move z)` in the following environment:

$$x \mapsto \mathsf{loan}^m\ \ell_x, \quad y \mapsto \mathsf{loan}^m\ \ell_y, \quad z \mapsto (\mathsf{borrow}^m\ \ell_x\ 0, \mathsf{borrow}^m\ \ell_y\ 0)$$

This time, the presence of two regions forces us to be more precise. We dispatch each component of the argument and each component of the returned value to its respective region abstraction. We get the following environment after the function call, where the value _ inside a region stands for a value which is ignored, because it belongs to a different region:

$$\begin{aligned}
&x \mapsto \mathsf{loan}^m\ \ell_x, \quad y \mapsto \mathsf{loan}^m\ \ell_y, \quad z \mapsto \bot,\\
&A(\alpha)\{\quad (\mathsf{borrow}^m\ \ell_x\ 0,\ _), \quad (_,\ \mathsf{loan}^m\ \ell_r) \quad\},\\
&A(\beta)\{\quad (_,\ \mathsf{borrow}^m\ \ell_y\ 0) \quad (\mathsf{loan}^m\ \ell_l,\ _) \quad\},\\
&r \mapsto (\mathsf{borrow}^m\ \ell_l\ \sigma_l,\ \mathsf{borrow}^m\ \ell_r\ \sigma_r)
\end{aligned}$$

v	::=		value
		...	
		σ	symbolic value
		_	ignored value (only used in region abstractions)
A	::=	$A_{id}\,\{\,[v]\,\}$	region abstraction
Ω^{LLBC}	::=	$\{$	state
		$\mathsf{env} : id \underset{\text{partial}}{\longrightarrow} v,$	
		$\mathsf{abs} : A_{id} \underset{\text{partial}}{\longrightarrow} A,$	
		$\mathsf{stack} : [[x]]\ \}$	

Figure 10.1: LLBC$^\#$: States and Values

In the resulting environment, z has been consumed. The return value r contains a pair of mutable borrows whose corresponding loans are dispatched between the region abstractions for α and β. We also note that the two regions are completely independent; for instance, once can end $A(\alpha)$ to get access back to x:

$$
\begin{aligned}
&x \mapsto \mathsf{loan}^m\,\sigma_x, \quad y \mapsto \mathsf{loan}^m\,\ell_y, \quad z \mapsto \bot,\\
&\_ \mapsto \bot,\\
&A(\beta)\{ \quad (\_,\ \mathsf{borrow}^m\,\ell_y\ 0) \quad (\mathsf{loan}^m\,\ell_l,\ \_) \quad \},\\
&r \mapsto (\mathsf{borrow}^m\,\ell_l\ \sigma_l,\ \bot)
\end{aligned}
$$

10.2 From Concrete to Symbolic Semantics

We define our symbolic semantics, LLBC$^\#$, mostly as an extension of our earlier formalism, along with a new rule for function calls. First, we extend values and environments to account for symbolic values, denoted σ (Figure 10.1), as well as region abstractions.

We present the additional rules for LLBC$^\#$ in Figure 10.2. We also slightly modify the evaluation statement: a statement now evaluates to a *set* of tagged states (i.e., pairs of a control-flow tag and a state). For instance, E-Seq-Symbolic states that when evaluating a sequence s0; s1 we evaluate the first statement then, for all the successful evaluations which do not break the control-flow (i.e., which have the tag unit), we evaluate the second statement. We introduce new reorganization rules to "expand" symbolic values; for instance, Reorg-SymbolicPair allows expanding a symbolic value $\sigma : (\tau_0, \tau_1)$ of type pair into a pair of fresh symbolic values $\sigma_0 : \tau_0$ and $\sigma_1 : \tau_1$, by substituting σ with $(\sigma_0 : \tau_0, \sigma_1 : \tau_1)$ in the environment. An enum can not be expanded

$$\textsc{Reorg-End-MutBorrow}\#\quad \frac{\text{hole of } \Omega[\mathsf{loan}^m\,\ell,\ .] \text{ not inside a borrowed value or a region abstraction} \qquad \text{no loan, } \mathsf{borrow}^r \in v}{\Omega[\mathsf{loan}^m\,\ell,\ \mathsf{borrow}^m\,\ell\ v] \hookrightarrow \Omega[v, \bot]}$$

$$\textsc{Reorg-End-SharedReservedBorrow}\#\quad \frac{\text{hole of } \Omega[.] \text{ not inside a borrowed value or a region abstraction}}{\Omega[\mathsf{borrow}^{s,r}\,\ell] \hookrightarrow \Omega[\bot]}$$

$$\textsc{Reorg-End-Abstraction}\quad \frac{\text{no borrows, loans} \in \overrightarrow{v},\ \overrightarrow{v'} \qquad \overrightarrow{\sigma} \text{ fresh}}{\Omega,\ A\,\{\ \overrightarrow{v},\ \overrightarrow{\mathsf{borrow}^s\,\ell},\ \overrightarrow{\mathsf{borrow}^m\,\ell'\ (v' : \tau)}\ \} \hookrightarrow \Omega,\ \overrightarrow{\_ \to \mathsf{borrow}^s\,\ell},\ \overrightarrow{\_ \to \mathsf{borrow}^m\,\ell'\ (\sigma : \tau)}}$$

$$\textsc{E-IfThenElse-Symbolic}\quad \frac{\Omega \vdash op \Downarrow (v,\ \Omega') \qquad v = (\sigma : \mathsf{bool}) \vee v = \mathsf{loan}^s\,\ell\,(\sigma : \mathsf{bool}) \qquad \Omega_0 = \Omega'[\mathsf{true}/\sigma] \qquad \Omega_1 = \Omega'[\mathsf{false}/\sigma] \qquad \Omega_0 \vdash s_0 \rightsquigarrow S_0^\# \qquad \Omega_1 \vdash s_1 \rightsquigarrow S_1^\#}{\Omega \vdash \mathsf{if}\ op\ \mathsf{then}\ s_0\ \mathsf{else}\ s_1 \rightsquigarrow S_0^\# \cup S_1^\#}$$

$$\textsc{E-Match-Symbolic}\quad \frac{\vdash \Omega(p) \overset{\mathsf{imm}}{\Rightarrow} v \qquad v = (\sigma : \tau_0 + \tau_1) \vee v = \mathsf{loan}^s\,\ell\,(\sigma : \tau_0 + \tau_1) \qquad \sigma_0,\ \sigma_1 \text{ fresh} \qquad \Omega_0 = \Omega'[\mathsf{Left}\,(\sigma_0 : \tau_0)/\sigma] \qquad \Omega_1 = \Omega'[\mathsf{Right}\,(\sigma_0 : \tau_0)/\sigma] \qquad \Omega_0 \vdash s_0 \rightsquigarrow S_0^\# \qquad \Omega_1 \vdash s_1 \rightsquigarrow S_1^\#}{\Omega \vdash (\mathsf{match}\,p\,\mathsf{with} \mid \mathsf{Left}\ \Rightarrow s_0 \mid \mathsf{Right}\ \Rightarrow s_1) \rightsquigarrow S_0^\# \cup S_1^\#}$$

$$\textsc{E-Seq-Symbolic}\quad \frac{\Omega \vdash s_0 \rightsquigarrow \{((), \Omega_i)\} \cup S^\# \qquad \forall\, r \in S^\#,\ \forall\, \Omega,\ r \neq ((), \Omega) \qquad \forall\, i,\ \Omega_i \vdash s_1 \rightsquigarrow S_i^\#}{\Omega \vdash s_0;\ s_1 \rightsquigarrow S^\# \cup (\bigcup_i S_i^\#)}$$

$$\textsc{Copy-Symbolic}\quad \frac{\sigma' \text{ fresh}}{\vdash \mathsf{copy}\,\sigma = \sigma'} \qquad \textsc{Reorg-SymbolicBox}\quad \frac{\sigma' \text{ fresh}}{\Omega[\sigma : \mathsf{Box}\,\tau] \hookrightarrow \Omega[\mathsf{Box}\,\sigma']} \qquad \textsc{Reorg-SymbolicPair}\quad \frac{\sigma_0,\ \sigma_1 \text{ fresh}}{\Omega[\sigma : (\tau_0,\ \tau_1)] \hookrightarrow \Omega[(\sigma_0,\ \sigma_1)]}$$

Figure 10.2: LLBC$^\#$: Additional Rules

during a reorganization; only `match`es allow doing so. In this case we continue the evaluation inside the branches after substituting the enum value into either of its variants (E-Match-Symbolic).

We also replace the rules to end borrows with more general versions to account for abstractions (Reorg-End-MutBorrow# and Reorg-End-SharedReservedBorrow#). Those rules forbid ending directly a borrow which is inside a region abstraction; one needs to end the whole region abstraction first, with a new reorganization rule Reorg-End-Abstraction. This rule states that, if there doesn't remain any loans in the region abstraction, we can end it to reintroduce its borrows in the environment inside anonymous variables, replacing the mutably borrowed values with fresh symbolic values.

Function calls (E-Call-Symbolic) require several steps. First, we evaluate all the operands to compute the input values. We then introduce fresh region abstractions for the set of regions $\overrightarrow{\rho}$ in the signature and whose content is defined through an auxiliary judgement (InstSig). We need to dispatch, or "project", the input borrows into the region abstractions they belong to; this is done through an auxiliary device $\mathsf{proj\_input}$ (Figure 10.4). The expression $\mathsf{proj\_input}\ \rho\ (v : \tau)$ defines the projection of the borrows belonging to region ρ of the value v seen as having type τ; the type τ, in which borrows are annotated with regions, is derived from the type of the function. The rules are straightforward: if τ does not contain the region ρ following which we are projecting, we ignore the value (ProjInput-Ignore); we project borrows belonging to ρ (ProjInput-Shared, ProjInput-Mut); we independently project the fields of ADTs (ProjInput-Pair). We also need to introduce borrows and loans for the output value; this is done through an auxiliary projector $\mathsf{proj\_output}$ (Figure 10.5). At the difference of $\mathsf{proj\_input}$, $\mathsf{proj\_output}$ takes as input not a value but the type of the output value, and evaluates to an output value (containing fresh borrows) as well as sets of loans, grouped according to the regions they belong to; the rules are similar to the ones defining $\mathsf{proj\_input}$.

Let us revisit the example from Section 10.1.3. The function `swap` has the following signature:

$$\mathsf{fn\ swap}\langle\alpha, \beta\rangle(z : (\&^{\alpha}\mathsf{mut\ u32}, \&^{\beta}\mathsf{mut\ u32})) \to (\&^{\beta}\mathsf{mut\ u32}, \&^{\alpha}\mathsf{mut\ u32})$$

We examine a call `let r = swap (move z)` in the following environment:

$$x \mapsto \mathsf{loan}^{m}\ \ell_x, \quad y \mapsto \mathsf{loan}^{m}\ \ell_y, \quad z \mapsto (\mathsf{borrow}^{m}\ \ell_x\ 0, \mathsf{borrow}^{m}\ \ell_y\ 0)$$

We introduce regions for α and β, and need to dispatch its input $(\mathsf{borrow}^{m}\ \ell_x\ 0, \mathsf{borrow}^{m}\ \ell_y\ 0)$

E-Call-Symbolic **(LLBC$^{\#}$ only)**

$$\frac{f\langle \vec{\rho},\ \vec{\tau}\rangle = \mathsf{fn}\ \langle\vec{\rho}\rangle\ (\overrightarrow{x} : \overrightarrow{\tau})\ (\overrightarrow{y} : \overrightarrow{\tau'})\ (x_{\mathsf{ret}} : \tau_{\mathsf{ret}})\ \{\ s\ \} \qquad \Omega_j \vdash op_j \Downarrow (v_j,\ \Omega_{j+1}) \\ \overrightarrow{\rho}\ \text{fresh} \qquad \overrightarrow{A_{\mathsf{sig}}(\rho)},\ v_{\mathsf{out}} = \mathsf{inst\_sig}(\Omega_n, \vec{\rho}, \vec{v}, \tau_{\mathrm{ret}}) \qquad \Omega_n,\ \overrightarrow{A_{\mathsf{sig}}(\rho)} \vdash p := v_{\mathsf{out}} \rightsquigarrow ((),\ \Omega')}{\Omega_0 \vdash p := f\langle \_,\ \vec{\tau}\rangle(\overrightarrow{op}) \rightsquigarrow ((),\ \Omega')}$$

InstSig

$$\frac{\begin{array}{c}\forall\ \rho,\ A^{\mathrm{in}}(\rho) = \{\mathsf{proj\_input}\ \rho\ v_i\} \\ \mathsf{proj\_output}\ \vec{\rho}\ \tau = (v_{\mathsf{out}},\ \overrightarrow{A^{\mathrm{out}}(\rho)}) \\ \forall\ \mathsf{borrow}^s\ \ell \in \vec{v},\ \exists\ v,\ \mathsf{loan}^s\ \ell\ v \in \Omega \\ \forall\ \rho,\ A_{\mathsf{sig}}(\rho) = A^{\mathrm{in}}(\rho) \cup A^{\mathrm{out}}(\rho)\end{array}}{\mathsf{inst\_sig}\ (\Omega,\ \vec{\rho},\ \vec{v},\ \tau) = \overrightarrow{A_{\mathsf{sig}}(\rho)},\ v_{\mathsf{out}}}$$

Figure 10.3: LLBC$^{\#}$: Function Calls

accordingly. From the signature of swap, we learn that, from the point of view of swap, the input has type $(\&^{\alpha}\mathsf{mut\ u32}, \&^{\beta}\mathsf{mut\ u32})$. We project following α:

$$\begin{aligned}&\mathsf{proj\_input}\ \alpha\ ((\mathsf{borrow}^m\ \ell_x\ 0, \mathsf{borrow}^m\ \ell_y\ 0) : (\&^{\alpha}\mathsf{mut\ u32}, \&^{\beta}\mathsf{mut\ u32})) \\ &= (\mathsf{proj\_input}\ \alpha\ (\mathsf{borrow}^m\ \ell_x\ 0 : \&^{\alpha}\mathsf{mut\ u32}),\ \mathsf{proj\_input}\ \alpha\ (\mathsf{borrow}^m\ \ell_y\ 0 : \&^{\beta}\mathsf{mut\ u32})) \\ &= (\mathsf{borrow}^m\ \ell_x\ \_,\ \_)\end{aligned}$$

The projection of the input value following β is similar. We now need to compute the output. We have, for the first element of the returned pair:

$$\mathsf{proj\_output}\ \{\alpha, \beta\}\ (\&^{\beta}\mathsf{mut\ u32}) = (\mathsf{borrow}^m\ \ell_l\ \sigma_l,\ \{A^{\mathrm{out}}(\alpha) = \{\},\ A^{\mathrm{out}}(\beta) = \{\mathsf{loan}^m\ \ell_l\}\})$$

Similarly, for the second element:

$$\mathsf{proj\_output}\ \{\alpha, \beta\}\ (\&^{\alpha}\mathsf{mut\ u32}) = (\mathsf{borrow}^m\ \ell_r\ \sigma_r,\ \{A^{\mathrm{out}}(\alpha) = \{\mathsf{loan}^m\ \ell_r\},\ A^{\mathrm{out}}(\beta) = \{\}\})$$

This gives us, for the whole pair:

$$\begin{aligned}&\mathsf{proj\_output}\ \{\alpha, \beta\}\ (\&^{\alpha}\mathsf{mut\ u32},\ \&^{\beta}\mathsf{mut\ u32}) = \\ &\quad((\mathsf{borrow}^m\ \ell_l\ \sigma_l,\ \mathsf{borrow}^m\ \ell_r\ \sigma_r),\ \{A^{\mathrm{out}}(\alpha) = \{\mathsf{loan}^m\ \ell_r\},\ A^{\mathrm{out}}(\beta) = \{\mathsf{loan}^m\ \ell_l\}\})\end{aligned}$$

Putting everything together, we get the environment shown in Section 10.1.3.

$$\textsc{ProjInput-Ignore}\ \frac{\rho \notin \tau}{\mathsf{proj\_input}\ \rho\ (v : \tau) = \_} \qquad \textsc{ProjInput-Pair}\ \frac{\mathsf{proj\_input}\ \rho\ v_0 = v_0' \qquad \mathsf{proj\_input}\ \rho\ v_1 = v_1'}{\mathsf{proj\_input}\ \rho\ (v_0,\ v_1) = (v_0',\ v_1')}$$

$$\textsc{ProjInput-Shared}\ \frac{\text{no borrows} \in \tau}{\mathsf{proj\_input}\ \rho\ (\mathsf{borrow}^s\ \ell : \&\rho\,\tau) = \mathsf{borrow}^s\ \ell}$$

$$\textsc{ProjInput-Mut}\ \frac{\text{no borrows} \in v}{\mathsf{proj\_input}\ \rho\ (\mathsf{borrow}^m\ \ell\ v : \&\rho\,\mathsf{mut}\,\tau) = \mathsf{borrow}^m\ \ell\ \_}$$

Figure 10.4: LLBC$^{\#}$: Input Projections

$$\textsc{ProjOutput-Symbolic}\ \frac{\sigma \text{ fresh} \qquad \text{no borrows} \in \tau}{\mathsf{proj\_output}\ \vec{\rho}\ \tau = (\sigma,\ \overrightarrow{A(\rho)})} \qquad \textsc{ProjOutput-Pair}\ \frac{\begin{array}{c}\mathsf{proj\_output}\ \vec{\rho}\ \tau_0 = (v_0,\ \overrightarrow{A_1(\rho)}) \\ \mathsf{proj\_output}\ \vec{\rho}\ \tau_1 = (v_1,\ \overrightarrow{A_2(\rho)}) \\ \forall\ \rho,\ A(\rho) = A_1(\rho) \cup A_2(\rho)\end{array}}{\mathsf{proj\_output}\ \vec{\rho}\ (\tau_0,\ \tau_1) = ((v_0,\ v_1),\ \overrightarrow{A(\rho)})}$$

$$\textsc{ProjOutput-Shared}\ \frac{\text{no borrows} \in \tau \qquad \sigma,\ \ell \text{ fresh} \qquad A(\rho) = \{\ \mathsf{loan}^s\ \ell\,\sigma\ \} \qquad \forall\ \rho' \neq \rho,\ A(\rho') = \{\}}{\mathsf{proj\_output}\ \vec{\rho}\ (\&\rho\,\tau) = (\mathsf{borrow}^s\ \ell,\ \overrightarrow{A(\rho)})}$$

$$\textsc{ProjOutput-Mut}\ \frac{\text{no borrows} \in \tau \qquad \sigma,\ \ell \text{ fresh} \qquad A(\rho) = \{\ \mathsf{loan}^m\ \ell\ \} \qquad \forall\ \rho' \neq \rho,\ A(\rho') = \{\}}{\mathsf{proj\_output}\ \vec{\rho}\ (\&\rho\,\mathsf{mut}\,\tau) = (\mathsf{borrow}^m\ \ell\ \sigma,\ \overrightarrow{A(\rho)})}$$

Figure 10.5: LLBC$^{\#}$: Output Projections

Chapter 11

Soundness: LLBC$^{\#}$ Defines a Borrow-Checker

In the previous chapter we introduced LLBC, an operational semantics for safe Rust focused on the semantics of borrows. By tweaking the semantics of LLBC we also produced LLBC$^{\#}$, a symbolic semantics for LLBC. We claimed that a symbolic interpreter for LLBC$^{\#}$ implements a borrow-checker for Rust; in this chapter we set out to substantiate this claim. Doing so requires proceeding in several steps.

First, the soundness of LLBC$^{\#}$ is predicated on the LLBC model being a sound foundation. LLBC has several unusual features, such as attaching values to pointers rather than to the underlying memory location, or not relying on an explicit heap; right now, it requires a leap of faith to believe that this is an acceptable way to model Rust programs. The value of LLBC is that it explains, checks and provides a semantic foundation for reasoning about many Rust features; but the drawback is that one has to trust that this semantics makes sense. We remark already that this question is orthogonal to the RustBelt line of work. RustBelt attempts to establish the soundness of Rust's type system with regards to λ_{Rust}, whose classic, unsurprising semantics does not warrant scrutiny. Clarifying the link between LLBC and a standard heap-and-addresses model is not just a matter of theory: once the Rust compiler emits LLVM bitcode, the heap and addresses become real. Finally, we note that the essential property of a borrow-checker is to ensure memory safety; as a consequence, we can actually not specify what it means for LLBC$^{\#}$ to define a borrow-checker in the first place, without first linking LLBC to more standard semantics which explicitly model the heap.

The second issue relates to LLBC$^{\#}$ itself. LLBC$^{\#}$ shares most of its semantics with LLBC with the exception of function calls, where it uses function signatures as summaries to approximate parts of the borrow graph by leveraging the so-called region abstractions. In spite of carefully-crafted rules that emphasize readability and simplicity,

the formal argument that it correctly approximates LLBC is highly non trivial.

In this chapter, we set out to address those two issues, and introduce new proof techniques to do so. First, we establish that LLBC is, indeed, a reasonable model of execution, and that in spite of some mildly exotic features, it really does connect to a traditional heap-and-addresses model of execution. Second, we show that the symbolic semantics of LLBC correctly approximates its concrete semantics, and thus that successful executions in the symbolic semantics guarantee the soundness of concrete executions – in short, that the symbolic interpreter acts as a borrow checker.

Because LLBC can, in some situations, analyze code more precisely than the Rust borrow-checker, we are therefore able to prove the soundness of more programs than the Rust compiler itself. We thus hope that this work sheds some light on potential improvements to the current implementation of the borrow-checker, backed by an actual formal argument.

More precisely, we do the following:

We introduce PL, for "pointer language", which uses a traditional, explicit heap inspired by CompCert's C memory model, and show that it refines LLBC (Section 11.2). This allows us to establish that a low-level model (where values live at a given address) refines the Rust model given by LLBC (where borrows hold the value they are borrowing). These two semantics have opposite perspectives on what it *means* to have a pointer.

We prove that LLBC itself refines the symbolic version of LLBC. Combined with the previous result, this allows us to precisely state why LLBC$^\#$ is a borrow-checker for LLBC: if an LLBC$^\#$ execution succeeds, then any corresponding low-level PL execution is safe for all inputs. We obtain this result by a combination of forward simulations, along with the determinism of the target (PL) language; we also reason about what it means for a low-level (PL) initial state to satisfy a function signature with ownership constraints in LLBC$^\#$.

To conduct these proofs of refinement, we introduce a novel proof technique that relies on the fact that our languages operate over the same grammar of expressions, but give it different *meanings*, or *views* (Section 11.1). For instance: both our heap-and-address (PL) and borrows-and-loans (LLBC) views operate over the same program syntax, but have different types for their state and reduction rules. Rather than go full-throttle with a standard compilation-style proof of simulation, we reason modularly over local or pointwise transformations that rewrite one piece of state to another – proofs over those elementary transformations are much easier. We then show that two states that are related by the transitive closure of these elementary transformations continue to relate throughout their execution, in the *union* of the semantics, ultimately giving a proof of refinement.

In order not to overwhelm the reader with technical details, we only present our proof methodology and the important theorems, and leave the detailed proofs with the necessary intermediate lemmas in appendix.

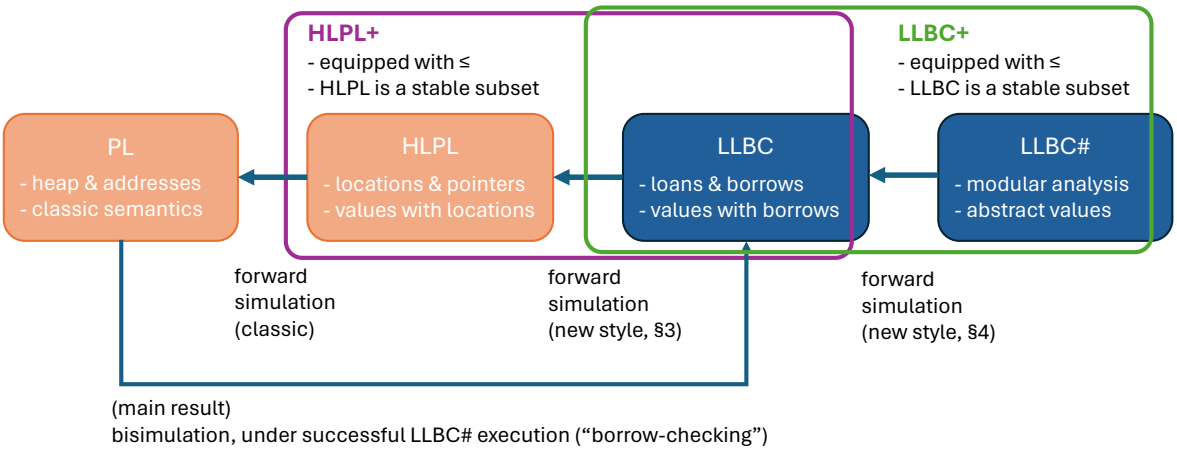

Figure 11.1: The architecture of our proof

11.1 A Generic Approach to Proving Language Simulations

As is standard in this kind of work, we go through several intermediate languages in order to relate our high-level (LLBC$^{\#}$) and low-level (PL) languages (Figure 11.1). This allows for modular reasoning, where each step focuses on a particular semantic concept.

To conduct these various refinement steps, we design a new, reusable proof methodology that allows efficiently establishing simulations between languages via the use of local and pointwise reasoning. We use this generic methodology repeatedly throughout the rest of the paper, so as to make our various reasoning steps much more effective; we now present the idea in the abstract, and apply it to our use-cases in subsequent sections (Section 11.2, Section 11.3).

Our approach consists of a generic proof template for establishing simulations between two languages L_l and L_h that share a grammar of statements, but whose semantics and notions of states differ. To simplify the presentation, we focus on forward simulations, but the methodology can be easily adapted to work dually for backward simulations. We briefly touch on this at the end of this section, and later discuss why we focus on forward simulations in this work, in Section 11.2.1.

Formally, we write Ω_l, Ω_h for low-level and high-level states respectively; $\leq$, for a given relation that *refines* a high-level state into a low-level state, i.e., $\Omega_l \leq \Omega_h$;

and $\Omega_l \vdash_l e \Downarrow_l (e', \Omega_l')$ (respectively, h), for the reduction of expressions into another expression and resulting state. We do not make any assumptions about what kind of reduction $\Downarrow$ is (small or big step); we simply assume one can perform inductive reasoning over it. We may omit some of the indices when clear from the context.

We aim to establish the following property:

Definition 1 (Forward Simulation)**.** *For all states Ω_l, Ω_h:*

$$\begin{aligned}
&\Omega_l \leq \Omega_h \Rightarrow \\
&\forall\, e\, e'\, \Omega_h',\ \Omega_h \vdash_{\mathsf{h}} e \Downarrow_h (e', \Omega_h') \Rightarrow \\
&\exists\, \Omega_l',\ \Omega_l \vdash_{\mathsf{l}} e \Downarrow_l (e', \Omega_l') \wedge \Omega_l' \leq \Omega_h'
\end{aligned}$$

Proving this property in a direct fashion can be tedious. When states Ω_l and Ω_h heavily differ, for instance because they operate at different levels of abstraction, the state relation commonly consists of complex global invariants, which are tricky to both correctly define and reason about, and oftentimes require maintaining auxiliary data structures, such as maps between high and low, for the purposes of the proof.

To circumvent this issue, our approach relies on two key components. First, instead of a global relation between states Ω_l and Ω_h, we define a set of small, local rules, the transitive closure of which constitutes $\leq$ (Section 11.1.1). These can then be reasoned upon individually.

Second, we add the ability to reason about states that contain a mixture of high and low, which we dub "hybrid" (Section 11.1.2). These now occur because our rewritings operate incrementally, and thus may give rise to states that belong neither to L_l nor L_h. Since the proof of forward simulation involves an induction on $\Downarrow$, we now must reason about the reduction of terms in hybrid states. Note that, if the empty states in L_l and L_h are related, we retrieve standard simulation properties, namely that the execution of a closed program in L_l is related to its execution in L_h. This will be the case for the state relations in Section 11.2 and Section 11.3.

11.1.1 Local State Transformations

To illustrate the idea of state relations based on local transformations, we take the simplistic example of a state that contains two integer variables x and y; in Ω_l, x and y contain concrete values; in Ω_h, x and y contain symbolic (or abstract) values. This is a much-simplified version of the full proof we later develop in Section 11.3; the setting of abstract/concrete values makes it easy to illustrate the concept. In this example,

our goal is to prove that L_h implements a sound symbolic execution for L_l. Instead of defining a global relation using universal quantification on all variables, we instead define a local relation that, for a given variable, swaps its concrete value n and its corresponding abstract version σ in L_h. We concisely write $\Omega[n] \leq \Omega[\sigma]$, leveraging a state-with-holes notation for Ω.

Establishing $L_l \leq L_h$ involves repeatedly applying this relation; to either x followed by y, or y followed by x; the locality of the transformation enables us to reason modularly about both variables. Naturally, once the reasoning becomes more complex, the ability to consider a single transformation at a time is crucial.

We remark that these individual transformations are non-directed, and can be read either left-to-right or right-to-left. In our example, when read left-to-right, we have an abstraction; when read right-to-left we have a concretization. This supports our later claim that this methodology works for both forward and backward simulations.

11.1.2 Reasoning over a Superset Language

By defining the state relation $\leq$ as the reflexive, transitive closure of local relations, we can now attempt to prove forward simulation by a standard induction on the evaluation in L_h, followed by an induction on the $\leq$ relation.

However, one problem arises: intermediate states do not belong to either language, and hence do not have semantics. To see why, consider in our proof the induction step corresponding to the transitivity of the relation, where we have an intermediate state Ω_m such that $\Omega_l \leq \Omega_m \leq \Omega_h$, and we want to use an induction hypothesis on $\Omega_m \leq \Omega_h$. We do so in order to establish that if $\Omega_h \vdash_{\mathsf{h}} e \Downarrow_h (e', \Omega'_h)$ and $\Omega_m \leq \Omega_h$, then $\Omega_m \vdash e \Downarrow (e', \Omega'_m)$ and $\Omega'_m \leq \Omega'_h$ – this is the induction on the size of $\leq$ for the purposes of establishing the forward simulation. This is fine, except reduction is not defined for hybrid states like Ω_m which contain a mixture of high-level and low-level.

To address this issue, we instead consider a superset language L^+ that contains semantics from both L_l and L_h.

Coming back to our previous example, let us assume that we want to update variable x. If x has already been rewritten, we can rely on the corresponding semantic rule in L_h, otherwise we execute the program according to the semantics in L_l. For our simplistic example, no further rules are needed, and the strict union suffices, i.e. $L^+ = L_l \cup L_h$. We will see shortly that this strict union is not always adequate in the general case.

With this approach, we establish the following theorem for two concrete languages L_l and L_h:

Theorem 1 (Forward Simulation on Superset Language)**.** *For all L^+ states Ω_1, Ω_2, we*

have:

$$\begin{aligned}
&\Omega_1 \leq \Omega_2 \Rightarrow \\
&\forall\, e\, e'\, \Omega'_2,\, \Omega_2 \vdash_+ e \Downarrow_+ (e', \Omega'_2) \Rightarrow \\
&\exists\, \Omega'_1,\, \Omega_1 \vdash_+ e \Downarrow_+ (e', \Omega'_1) \wedge \Omega'_1 \leq \Omega'_2
\end{aligned}$$

Unfortunately, instantiating this theorem with states $\Omega_1 = \Omega_l \in L_l$ and $\Omega_2 = \Omega_h \in L_h$ does not allow us to derive Theorem 1, which was our initial goal. Indeed, Ω_1 initially belongs to L_l but reduces with $\Downarrow_+$, i.e., with the union of the semantics, and so far nothing allows us to conclude that the resulting Ω'_1 is still in L_l.

To ensure that this execution is valid with respect to L_l, we need to restrict L^+ by excluding a set of rules R from L_h so that L_l and L^+ satisfy the following property.

Definition 2 (Stability)**.** *Given two languages L_l, L^+ such that L^+ is a superset of L_l, we say that L_l is a stable subset of L^+ if, for all $e, e', \Omega \in L_l, \Omega^+ \in L^+$, if $\Omega \vdash_+ e \Downarrow_+ (e', \Omega^+)$, then $\Omega^+ \in L_l$ and $\Omega \vdash_l e \Downarrow_l (e', \Omega^+)$*

Combined with stability, Theorem 1 allows us to directly derive that L_l and L^+ satisfy a forward simulation relation. To conclude, the last remaining step is to establish the same property between L^+ and L_h, which only requires reasoning about the excluded rule set R.

The earlier simplistic example of concrete/abstract integers requires no particular care when defining L_+, and the union of the rules suffices (i.e., $R = \varnothing$). But for our real use-case, one can see from Figure 11.1 that the semantics of HLPL$^+$ exclude some of the rules from LLBC. We also point out that for our real use-cases, the superset L_+ requires additional administrative rules, to make sure high and low compose (as we will see shortly).

Forward *vs.* backward. While the presentation focused on forward simulations, the approach can be easily adapted to backward simulations. Leveraging the duality between both relations, it is sufficient to exclude a set of rules R from L_l instead of L_h so that L_h becomes a stable subset of L^+, and to similarly conclude by reasoning over the rules in R.

11.2 A Heap-and-Addresses Interpretation of Valued Borrows

Equipped with our generic proof methodology, we now turn to the Low-Level Borrow Calculus (LLBC). To remain conceptually close to Rust, we designed LLBC to operate on states containing loans and borrows. While this helps understanding and explaining the language from the programmer's perspective, this model departs from standard operational semantics for heap-manipulating programs, which commonly rely on a low-level model based on memory addresses and offsets [393, 405, 406]. In this section, we thus bridge the abstraction gap, by relating LLBC to a standard, low-level operational semantics, therefore establishing LLBC as a sound semantic foundation for Rust programs. To do so, we introduce the Pointer Language (PL), a small language explicitly modeling the heap using a model inspired by CompCert's C memory model, and establish a relation between LLBC and PL that demonstrates that LLBC's borrow-centric view of memory is compatible with classic pointers.

11.2.1 Forward *vs* Backward

In our proof, the source and target language are the same – we are not doing a compiler correctness proof. Rather, we reconcile two models of execution over the same syntax of programs, i.e., given a PL state that *concretizes* the LLBC state, the program computes in PL the same result as in LLBC. This is similar to the original circa-2008 style of CompCert proofs, and qualifies as a forward simulation [405].

Indeed, at this stage, it is not true that every PL execution can be simulated backwards by an LLBC execution; simply said, a PL program could be safe for reasons that cannot be explained by LLBC's borrow semantics. We will see in the next section (Section 11.3.5) how we can obtain the backwards direction, provided the program is borrow-checked – a result that is akin to a typing result. For now, we simply seek to establish that the execution model of LLBC is a correct restriction of a traditional heap-and-addresses model. To that end, we use the methodology from Section 11.1, and for now only use the forward direction it gives us.

11.2.2 Difficulties and Methodology

Instead of environments (we use the terms environments and states interchangeably) containing borrows and loans, the PL language operates on an explicit heap adapted from the CompCert memory model [406]. In particular, loan identifiers are replaced by

v	$::=$		value
		$\mathsf{true} \mid \mathsf{false} \mid n_{\mathsf{i32}} \mid n_{\mathsf{u32}} \mid \ldots$	literal constants
		$\mathsf{Left}\, v \mid \mathsf{Right}\, v$	sum value
		$()$	unit
		(v_0, v_1)	pair
		$\bot$	bottom (invalid) value
		$\mathsf{loc}\, \ell\, v$	location
		$\mathsf{ptr}\, \ell$	pointer

$\Omega^{\text{HLPL}} ::= \{\ \mathsf{env} : id \underset{\text{partial}}{\longrightarrow} v,\ \mathsf{stack} : [[x]]\ \}$ state

Figure 11.2: HLPL: Environments and Values

memory addresses, consisting of a block identifier and an offset, and the memory layout is made semi-explicit, by including a notion of *size* for each type.

When attempting to relate LLBC to a lower-level language like PL, two main difficulties arise. First, manipulating an explicit heap requires operating on sequences of words, which need to be reconciled with more abstract values. Second, one needs to relate borrows and loans to low-level pointers. The core of the difficulty lies in the fact that in LLBC, mutable borrows "carry" the value they borrow, making it hard to reason about the provenance of a value in the presence of reborrows.

This is where our methodology comes in (Section 11.1). To make the proof simpler and more modular, we proceed in two steps, via the addition of an intermediary language dubbed HLPL, for high-level pointer language.

11.2.3 An Intermediary Language: HLPL

As before, the program syntax remains the same; what differs now is that HLPL states are half-way between PL states and LLBC states. HLPL states no longer feature borrows and loans; but they retain an abstract notion of *pointers* and *locations*, denoted respectively $\mathsf{ptr}\, \ell$ and $\mathsf{loc}\, \ell\, v$.

The simulation from HLPL to PL is standard and only consists in materializing locations as a global heap, and mapping pointers to corresponding addresses. We instead focus on where the challenge lies, namely, going from HLPL (value is with the location) to LLBC (value is with the borrow).

In addition to the program syntax (same as LLBC) and definition of states (Figure 11.2), we define the operational semantics of HLPL in Figure 11.3. HLPL shares most of its semantics with LLBC with the exception that it uses *pointer* and *location* values; in order to go from LLBC to HLPL, we simply replace the rules which manipulate borrows and loans (E-MutBorrow, E-SharedBorrow, etc.) with rules handling

$$\textsc{E-Ptr}\ \frac{\vdash \Omega(p) \overset{\mathsf{imm,mut}}{\Rightarrow} v \qquad \vdash \Omega(p) \leftarrow v' \overset{\mathsf{imm,mut}}{\Rightarrow} \Omega' \qquad v' = \begin{cases} \mathsf{loc}\,\ell\,v'' & \text{if } v = \mathsf{loc}\,\ell\,v'' \\ \mathsf{loc}\,\ell\,v & \ell \text{ fresh otherwise} \end{cases}}{\Omega \vdash \{\&\,p, \&\mathsf{mut}\,p, \&\mathsf{reserved}\,p\} \Downarrow (\mathsf{ptr}\,\ell, \Omega')}$$

$$\textsc{Reorg-End-Ptr}\ \frac{}{\Omega[\mathsf{ptr}\,\ell] \hookrightarrow \Omega[\bot]}$$

$$\textsc{Reorg-End-Loc}\ \frac{\mathsf{ptr}\,\ell \notin \Omega[\mathsf{loc}\,\ell\,v]}{\Omega[\mathsf{loc}\,\ell\,v] \hookrightarrow \Omega[v]}$$

$$\textsc{HLPL-E-Move}\ \frac{\vdash \Omega(p) \overset{\mathsf{mov}}{\Rightarrow} v \qquad \bot, \mathsf{loc} \notin v \qquad \vdash \Omega[p \leftarrow \bot] \overset{\mathsf{mov}}{\Rightarrow} \Omega'}{\Omega \vdash \mathsf{move}\,p \Downarrow (v, \Omega')}$$

$$\textsc{HLPL-E-Box-New}\ \frac{\Omega \vdash op \Downarrow (v, \Omega') \qquad l^b \text{ fresh}}{\Omega \vdash \mathsf{new}\,op \Downarrow (\mathsf{ptr}\,\ell, (\Omega', l^b \to v))}$$

$$\textsc{HLPL-E-Box-Free}\ \frac{\vdash (\Omega, \ell^b \to v)(p) \overset{\mathsf{mov}}{\Rightarrow} \mathsf{ptr}\,\ell \qquad \text{no locations in } v \qquad \mathsf{ptr}\,\ell^b \notin \Omega \qquad \vdash \Omega(p) \leftarrow \bot \overset{\mathsf{mov}}{\Rightarrow} \Omega'}{\Omega, l \to v \vdash \mathsf{free}\,p \Downarrow ((), \Omega', \_ \to v)}$$

$$\textsc{HLPL-E-Assign}\ \textbf{(HLPL only)}\ \frac{\Omega \vdash rv \Downarrow (v, \Omega') \qquad \vdash \Omega'(p) \overset{\mathsf{mut}}{\Rightarrow} v_p : \tau \qquad v_p \text{ has no } \mathsf{loc} \qquad \vdash \Omega'(p) \leftarrow v \overset{\mathsf{mut}}{\Rightarrow} \Omega'' \qquad \Omega''' = \Omega'', \_ \to v_p}{\Omega \vdash p := rv \Downarrow ((), \Omega''')}$$

$$\textsc{R-Loc}\ \frac{P \neq [.] \qquad \Omega \vdash P(v) \overset{\mathsf{imm}}{\Rightarrow} v'}{\Omega(p) \vdash P(\mathsf{loc}\,\ell\,v) \overset{\mathsf{imm}}{\Rightarrow} v'}$$

$$\textsc{R-Deref-Ptr-Loc}\ \frac{\mathsf{loc}\,\ell\,v \in \Omega \qquad \Omega \vdash P(\mathsf{loc}\,\ell\,v) \overset{\mathsf{imm,mut}}{\Rightarrow} v'}{\Omega \vdash P(*(\mathsf{ptr}\,\ell)) \overset{\mathsf{imm,mut}}{\Rightarrow} v'}$$

$$\textsc{R-Deref-Box-Id}\ \frac{\Omega(\ell^b) = v \qquad \Omega \vdash P(v) \overset{\mathsf{k}}{\Rightarrow} v'}{\Omega \vdash P(*(\mathsf{ptr}\,\ell^b)) \overset{\mathsf{k}}{\Rightarrow} v'}$$

$$\textsc{W-Loc}\ \frac{P \neq [.] \qquad \Omega \vdash P[v] \leftarrow w \overset{\mathsf{imm}}{\Rightarrow} (v', \Omega')}{\Omega(p) \vdash P(\mathsf{loc}\,\ell\,v) \leftarrow w \overset{\mathsf{imm}}{\Rightarrow} (\mathsf{loc}\,\ell\,v', \Omega')}$$

$$\textsc{W-Deref-Ptr-Loc}\ \frac{\mathsf{loc}\,\ell\,v \in \Omega \qquad \Omega \vdash P(\mathsf{loc}\,\ell\,v) \leftarrow w \overset{\mathsf{imm,mut}}{\Rightarrow} (v', \Omega'[\mathsf{loc}\,\ell\,v'']) \qquad \Omega'' = \Omega'[\mathsf{loc}\,\ell\,v']}{\Omega \vdash P(*(\mathsf{ptr}\,\ell)) \leftarrow w \overset{\mathsf{imm,mut}}{\Rightarrow} \mathsf{ptr}\,\ell \dashv \Omega''}$$

$$\textsc{W-Deref-Box-Id}\ \frac{\Omega(\ell^b) = v \qquad \Omega \vdash P(v) \leftarrow w \overset{\mathsf{k}}{\Rightarrow} (v', \Omega') \qquad \Omega'' = (\Omega'(l^b) := v')}{\Omega \vdash P(*(\mathsf{ptr}\,\ell^b)) \leftarrow w \overset{\mathsf{k}}{\Rightarrow} (\mathsf{ptr}\,\ell^b, \Omega'')}$$

$$\textsc{Copy-Loc}\ \frac{\vdash \mathsf{copy}\,v = v'}{\vdash \mathsf{copy}\,(\mathsf{loc}\,\ell\,v) = v'}$$

$$\textsc{Copy-Ptr}\ \frac{}{\vdash \mathsf{copy}\,(\mathsf{ptr}\,\ell) = \mathsf{ptr}\,\ell}$$

Figure 11.3: HLPL: Semantics

pointers and locations. Pointers (respectively, locations) behave basically like shared borrows (respectively, loans), in that the value lives with the location. Unlike shared borrows, we permit modifications through the pointer. In particular, in HLPL we evaluate borrowing expressions like &mut p and &p with E-Ptr, which is very similar to E-SharedBorrow. We remark that HLPL retains some structure still: one can only update a value $x \to \mathsf{loc}\,\ell\,0$ via a corresponding pointer $p \to \mathsf{ptr}\,\ell$, e.g., to obtain $x \to \mathsf{loc}\,\ell\,1$ (W-Deref-Ptr-Loc, in Appendix). Should one want to update x *itself*, there must be no outstanding pointers to it (Reorg-End-Ptr), and the location itself must have been forgotten (Reorg-End-Loc). Just like in LLBC, these reorganizations may happen at any time, and as in LLBC, a poor choice of reorganizations may lead to a stuck execution.

Let us contrast the semantics of HLPL and LLBC with an example. We annotated the snippet of code below with *LLBC* environments. This example doesn't introduce difficulties compared to what we have seen before. One has however to note that the reborrow on line 3 introduces an indirection between the value of px and x; in order to see that px actually borrows x, one has to follow ℓ_1 then ℓ_0.

```
1 x  = 0;            // x ↦ 0
2 px = &mut x;       // x ↦ loan^m ℓ0,   px ↦ borrow^m ℓ0 0
3 px = &mut (*px); // x ↦ loan^m ℓ0,   _ ↦ borrow^m ℓ0 (loan^m ℓ1),   px ↦ borrow^m ℓ1 0
4 assert!(x = 0);  // x ↦ 0,   _ ↦ ⊥,   px ↦ ⊥
```

Let us now look at the same example, but annotated with *HLPL* environments.

```
1 x  = 0;            // x ↦ 0
2 px = &mut x;       // x ↦ loc ℓ 0,   px ↦ ptr ℓ
3 px = &mut (*px); // x ↦ loc ℓ 0,   _ ↦ ptr ℓ,   px ↦ ptr ℓ
4 assert!(x = 0);  // x ↦ loc ℓ 0,   _ ↦ ptr ℓ,   px ↦ ptr ℓ
```

At line 2, we use E-Ptr to introduce a fresh location in x and evaluate &mut x to a fresh pointer. Contrary to what happens in LLBC, in HLPL we leave pointed values in place. At line 3, we use E-Ptr again, but this time dereference px (which maps to $\mathsf{ptr}\,\ell$), which yields the value of x ($\mathsf{loc}\,\ell\,0$), and create another pointer for this location. Because this value already contains a *location*, we do not introduce a fresh location and simply evaluate &mut (*px) to $\mathsf{ptr}\,\ell$. We now evaluate the assignment to px (left hand side of px = &mut (*px)): we move the current value of px ($\mathsf{ptr}\,\ell$) to a fresh anonymous value, then override the value of px with the result of evaluating the right-hand side ($\mathsf{ptr}\,\ell$).

Finally, at line 4, we need to read x. We can read *through* locations; we do not need to end any pointer at this point and leave the environment unchanged. The same

$$\textsc{Le-SharedReserved-To-Ptr}\quad \frac{}{\Omega[\mathsf{ptr}\,\ell] \leq \Omega[\mathsf{borrow}^{s,r}\,\ell]}$$

$$\textsc{Le-MutBorrow-To-Ptr}\quad \frac{\ell \notin \Omega[.,.] \qquad \ell \notin v}{\Omega[\mathsf{loc}\,\ell\,v,\ \mathsf{ptr}\,\ell] \leq \Omega[\mathsf{loan}^{m}\,\ell,\ \mathsf{borrow}^{m}\,\ell\ v]}$$

$$\textsc{Le-RemoveAnon}\quad \frac{\text{no borrow, loan, location, pointer} \in v}{\Omega \leq \Omega,\ \_ \to v}$$

$$\textsc{Le-Merge-Locs}\quad \frac{\forall\, v', \mathsf{loc}\,\ell'\,v' \notin \Omega[\mathsf{loc}\,\ell\,v] \qquad \Omega' = \left[\ell \middle/ \ell'\right](\Omega[\mathsf{loc}\,\ell\,v])}{\Omega' \leq \Omega[\mathsf{loc}\,\ell\,(\mathsf{loc}\,\ell'\,v)]}$$

$$\textsc{Le-SharedLoan-To-Loc}\quad \frac{\mathsf{borrow}^{s,r}\,\ell \notin \Omega[\mathsf{loc}\,\ell\,v]}{\Omega[\mathsf{loc}\,\ell\,v] \leq \Omega[\mathsf{loan}^{s}\,\ell\,v]}$$

$$\textsc{Le-Box-To-Loc}\quad \frac{\ell^b \text{ fresh}}{\Omega[\mathsf{ptr}\,\ell^b],\ l^b \to v \leq \Omega[\mathsf{Box}\,v]}$$

$$\textsc{Le-Subst}\quad \frac{\ell' \notin \Omega}{\left[\ell' \middle/ \ell\right]\Omega \leq \Omega}$$

Figure 11.4: The $\leq$ Relation on HLPL$^+$ states

happens with shared borrow in LLBC: we can read *through* shared values. However, if we were to directly update x, for instance by evaluating $\mathsf{x} = 1$, we would need to end the location with Reorg-End-Loc, which would require ending the pointers first by using Reorg-End-Ptr, yielding the environment: $x \mapsto 0, \quad \_ \mapsto \bot, \quad px \mapsto \bot$.

It is to be noted that E-Ptr applies both to mutable borrows and shared borrows: if we update the example above to replace the occurrences of &mut with &, we get exactly the same environments at each program point.

Finally, boxes in HLPL are handled differently than in LLBC. In LLBC, when we allocate a new box with value v, we simply create a value $\mathsf{Box}\,v$ (E-Box-New). In HLPL, we create a pointer and introduce a fresh box identifier in the environment; this allows us to stay close to PL which allocates a fresh memory block.

We now leverage the proof methodology of Section 11.1 to show that there is a forward simulation from LLBC to HLPL.

11.2.4 The $\leq$ Relation Between HLPL and LLBC States

Following Section 11.1, the first step consists in introducing a series of local rewriting rules (Figure 11.4, which defines the relation on *hybrid* states, that we dub hlpl+ states), whose transitive closure, written $\leq$, relates an HLPL state Ω_{hlpl} to an LLBC state Ω_{llbc}. The relation $\Omega_{\text{hlpl}} \leq \Omega_{\text{llbc}}$ can be read in both directions; but since we are concerned here with a forward simulation, we read these rules right to left, that is, we *gradually* transform borrows and loans (from LLBC) into pointers and locations (from HLPL).

In effect, this amounts to losing information about the nature of the borrows, in order to only retain an aliasing graph. Continuing with the right to left intuition,

Le-MutBorrow-To-Ptr states that we can collapse a pair of a mutable loan and its corresponding borrow to a location and a pointer. Notice how the value v moves from being attached to the borrow to being attached to the location; this is the crucial rule that moves from a borrow-centric view to a location-centric view.

Going back to the reborrowing example from above, we have at line 2 that the LLBC state $(x \mapsto \mathsf{loan}^m\ \ell_0,\ px \mapsto \mathsf{borrow}^m\ \ell_0\ 0)$ is related to the HLPL state $(x \mapsto \mathsf{loc}\,\ell\,0,\ px \mapsto \mathsf{ptr}\,\ell)$ by Le-MutBorrow-To-Ptr. The reborrow at line 3 is more interesting; starting from the LLBC state we have (we are using the notation: $\Omega \geq \Omega' \iff \Omega' \leq \Omega$):

$$
\begin{aligned}
& x \mapsto \mathsf{loan}^m\ \ell_0, \quad \_ \mapsto \mathsf{borrow}^m\ \ell_0\ (\mathsf{loan}^m\ \ell_1), \quad px \mapsto \mathsf{borrow}^m\ \ell_1\ 0 \\
& \geq x \mapsto \mathsf{loan}^m\ \ell_0, \quad \_ \mapsto \mathsf{borrow}^m\ \ell_0\ (\mathsf{loc}\,\ell_1\,0), \quad px \mapsto \mathsf{ptr}\,\ell_1 && (1) \\
& \geq x \mapsto \mathsf{loc}\,\ell_0\,(\mathsf{loc}\,\ell_1\,0), \quad \_ \mapsto \mathsf{ptr}\,\ell_0, \quad px \mapsto \mathsf{ptr}\,\ell_1 && (2) \\
& \geq x \mapsto \mathsf{loc}\,\ell_0\,0, \quad \_ \mapsto \mathsf{ptr}\,\ell_0, \quad px \mapsto \mathsf{ptr}\,\ell_0 && (3)
\end{aligned}
$$

By Le-MutBorrow-To-Ptr, we convert the pair borrow/loan for ℓ_1 to a pair pointer/location at step (1). We use Le-MutBorrow-To-Ptr again at step (2), this time for ℓ_0. We now get two stacked locations in `x`, that we "merge" together by using Le-Merge-Locs at step (3). This yields the same HLPL environment as at line 3.

Finally, we note that the states after the `assert` at line 4 are not related: we can not transform the LLBC state $(x \mapsto 0,\ \_ \mapsto \bot,\ px \mapsto \bot)$ to the HLPL state $(x \mapsto \mathsf{loc}\,\ell\,0,\ \_ \mapsto \mathsf{ptr}\,\ell,\ px \mapsto \mathsf{ptr}\,\ell)$ by using the rules for $\leq$. In particular, we note that the HLPL semantics allows strictly *more* behaviors than LLBC, as we *don't have* to end the pointers and the location to read `x`. However, we *can* end them to yield the HLPL state $x \mapsto 0,\ \_ \mapsto \bot,\ px \mapsto \bot$, which is related to the LLBC state at the same program point (they are actually the same).

Converting shared borrows to pointers is even simpler. This time we convert each shared borrow to a pointer with Le-SharedReserved-To-Ptr, then convert the shared loan to a location with Le-SharedLoan-To-Loc; we have to convert the borrows first because of the premise $\mathsf{borrow}^s\ \ell \notin \Omega$ in Le-SharedLoan-To-Loc.

Let us illustrate the conversion of shared borrows with an example. Below, we annotated each program point with *LLBC* environments.

```
1  x  = 0;          // x ↦ 0
2  px = &x;         // x ↦ loan^s ℓ 0,   px ↦ borrow^s ℓ
3  px = &(*px);     // x ↦ loan^s ℓ 0,   _ ↦ borrow^s ℓ,   px ↦ borrow^s ℓ
4  assert!(x = 0); // x ↦ loan^s ℓ 0,   _ ↦ borrow^s ℓ,   px ↦ borrow^s ℓ
```

Because in HLPL mutable borrows and shared borrows are evaluated with the same rule (E-Ptr), the HLPL environments are the same as in the mutable case.

```
1  x  = 0;          // x ↦ 0
2  px = &x;         // x ↦ loc ℓ 0,   px ↦ ptr ℓ
3  px = &(*px);     // x ↦ loc ℓ 0,   _ ↦ ptr ℓ,   px ↦ ptr ℓ
4  assert!(x = 0);  // x ↦ loc ℓ 0,   _ ↦ ptr ℓ,   px ↦ ptr ℓ
```

For the reborrow at line 3, we have:

$$
\begin{aligned}
& x \mapsto \mathsf{loan}^s\, \ell\, 0, \quad \_ \mapsto \mathsf{borrow}^s\, \ell, \quad px \mapsto \mathsf{borrow}^s\, \ell \\
\geq\ & x \mapsto \mathsf{loan}^s\, \ell\, 0, \quad \_ \mapsto \mathsf{borrow}^s\, \ell, \quad px \mapsto \mathsf{ptr}\, \ell && (1) \\
\geq\ & x \mapsto \mathsf{loan}^s\, \ell\, 0, \quad \_ \mapsto \mathsf{ptr}\, \ell, \quad px \mapsto \mathsf{ptr}\, \ell && (2) \\
\geq\ & x \mapsto \mathsf{loc}\, \ell\, 0, \quad \_ \mapsto \mathsf{ptr}\, \ell, \quad px \mapsto \mathsf{ptr}\, \ell && (3)
\end{aligned}
$$

We also provide a few administrative rules. For instance, Le-RemoveAnon allows to remove an anonymous value from the context, as long as it doesn't contain loans, borrows, locations or pointers; such values are de facto useless, and this rule allows us to formally ignore them. We transform LLBC boxes into an HLPL boxes with Le-Box-To-Loc, which introduces a fresh box identifier in the environment. The rule Le-Subst also allows us to substitute identifiers, which is necessary in the proof of the forward simulation between HLPL and LLBC.

11.2.5 Working in HLPL$^+$

Naturally, reasoning about $\leq$ (in order to establish the forward simulation, as we do in the next paragraph) is conducted via reasoning by induction. Specifically, we do the proof by induction on the evaluation derivation, then in each sub-case do an induction on $\leq$. This is the essence of our proof technique, which emphasizes local and pointwise reasoning rather than global invariants. In particular, we took great care to define the rules of $\leq$ either as *local* transformations (by defining them with states with holes), or as *pointwise* transformations (Le-Merge-Locs).

As we explained earlier (Section 11.1), this leads us to reason about hybrid states that contain both loans and borrows (like LLBC states) as well as locations and pointers (like HLPL states). We call such states HLPL$^+$ states, and per Section 11.1 set out to give an operational semantics to HLPL+.

Since HLPL$^+$ shares the same syntax as HLPL and LLBC, it suffices to take the union of the rules from HLPL and LLBC, *adding* HLPL$^+$-specific rules (Figure 11.5), and *excluding* the rules marked **(LLBC only)**. The LLBC-only rules introduce new loans and borrows; by excluding those, we get that HLPL is a stable subset of HLPL$^+$ (Definition 2).

HLPL+-E-Ptr

$$\frac{\vdash \Omega(p) \overset{\mathsf{imm},\mathsf{mut}}{\Rightarrow} v \qquad \vdash \Omega(p) \leftarrow v' \overset{\mathsf{imm},\mathsf{mut}}{\Rightarrow} \Omega' \qquad v' = \begin{cases} \mathsf{loc}\,\ell\,v'' & \text{if } v = \mathsf{loc}\,\ell\,v'' \\ \mathsf{loc}\,\ell\,v'' & \text{if } v = \mathsf{loan}^s\,\ell\,v'' \\ \mathsf{loc}\,\ell\,v & \ell \text{ fresh otherwise} \end{cases}}{\Omega \vdash \{\&\,p,\ \&\mathsf{mut}\,p,\ \&\mathsf{reserved}\,p\} \Downarrow (\mathsf{ptr}\,\ell,\ \Omega')}$$

HLPL+-E-Assign

$$\frac{\begin{array}{c}\Omega \vdash rv \Downarrow (v,\ \Omega') \qquad \vdash \Omega'(p) \overset{\mathsf{mut}}{\Rightarrow} v_p \\ v_p \text{ has no outer } \mathsf{loan},\ \text{no } \mathsf{loc} \\ \vdash \Omega'(p) \leftarrow v \overset{\mathsf{mut}}{\Rightarrow} \Omega'' \qquad \Omega''' = \Omega'',\ \_ \to v_p\end{array}}{\Omega \vdash p := rv \Downarrow ((),\ \Omega''')}$$

R-Deref-Ptr-SharedLoan

$$\frac{\mathsf{loan}^s\,\ell\,v \in \Omega \qquad \Omega \vdash P(\mathsf{loan}^s\,\ell\,v) \overset{\mathsf{imm}}{\Rightarrow} v'}{\Omega \vdash P(*(\mathsf{ptr}\,\ell)) \overset{\mathsf{imm}}{\Rightarrow} v'}$$

HLPL+-E-Move

$$\frac{\begin{array}{c}\vdash \Omega(p) \overset{\mathsf{mov}}{\Rightarrow} v \\ \bot, \mathsf{loan}, \mathsf{borrow}^r, \mathsf{loc} \notin v \\ \vdash \Omega[p \leftarrow \bot] \overset{\mathsf{mov}}{\Rightarrow} \Omega'\end{array}}{\Omega \vdash \mathsf{move}\,p \Downarrow (v,\ \Omega')}$$

W-Deref-Ptr-SharedLoan

$$\frac{\mathsf{loan}^s\,\ell\,v \in \Omega \qquad \Omega \vdash P(\mathsf{loan}^s\,\ell\,v) \leftarrow w \overset{\mathsf{imm}}{\Rightarrow} (v',\ \Omega'[\mathsf{loan}^s\,\ell\,v'']) \qquad \Omega'' = \Omega'[\mathsf{loan}^s\,\ell\,v']}{\Omega \vdash P(*(\mathsf{ptr}\,\ell)) \leftarrow w \overset{\mathsf{imm}}{\Rightarrow} (\mathsf{ptr}\,\ell,\ \Omega'')}$$

HLPL+-Reorg-End-SharedLoan

$$\frac{\mathsf{borrow}^{s,r}\,\ell,\ \mathsf{ptr}\,\ell \notin \Omega[\mathsf{loan}^s\,\ell\,v]}{\Omega[\mathsf{loan}^s\,\ell\,v] \hookrightarrow \Omega[v]}$$

Figure 11.5: Additional Rules for HLPL$^+$

The HLPL$^+$ rules are in Figure 11.5. We replace E-Ptr with HLPL+-E-Ptr, which is slightly more general to account for hybrid states; namely, we might want to create a pointer to a shared loan, in which case we do not introduce a fresh location. In a similar vein, R-Deref-Ptr-SharedLoan deals with a hybrid state where the state still contains a loan, but the value being reduced is an HLPL pointer, not a borrow. On its side, HLPL+-E-Assign shows how to add extra preconditions to extend the "no outstanding borrows" condition (required in LLBC for soundness) to a hybrid world in which there might be locations, too. We skip the other rules, which are straightforward.

Equipped with our individual rewriting rules (which form $\leq$) and a semantics in which those rules operate (HLPL$^+$, the union of HLPL and LLBC, crafted to make HLPL a stable subset), we now prove that reduction preserves $\leq$.

Theorem 2 (Eval-Preserves-HLPL+-Rel)**.** *For all Ω_l, Ω_r HLPL$^+$ states, we have:*

$$\Omega_l \leq \Omega_r \Rightarrow \forall\ s\ r\ \Omega'_r,\ \Omega_r \vdash_{\mathsf{hlpl+}}\ s \rightsquigarrow (r,\ \Omega'_r) \Rightarrow$$
$$\exists\ \Omega'_l,\ \Omega_l \vdash_{\mathsf{hlpl+}}\ s \rightsquigarrow (r,\ \Omega'_l) \wedge \Omega'_l \leq \Omega'_r$$

The proof is in Appendix C; it consists in a nested case analysis; first, for each reduction step; then, for each $\leq$ step. In particular, we use the fact that if two states are related by one of the $\leq$ rules, then the structure enforced by this rule is generally preserved after the evaluation.

11.2.6 From HLPL$^+$ to HLPL

HLPL$^+$ is merely a proof device; our ultimate goal is to relate LLBC to HLPL, not HLPL$^+$. Because we excluded (above) from HLPL$^+$ those rules that might create new loans or borrows, we trivially have the fact that the semantics of HLPL and HLPL$^+$ coincide on HLPL states (i.e., HLPL$^+$ states that don't contain loans or borrows, and thus belong to the HLPL subset of HLPL$^+$); that is, that HLPL is a stable subset of HLPL$^+$ in the sense of Definition 2. Some subtleties arise from the fact that we replaced some HLPL rules with more general HLPL$^+$ versions to account for hybrid states, but those rules were carefully crafted to coincide on non-hybrid states; we refer the interested reader to Appendix C for a detailed discussion.

From this we deduce that there is a forward relation from HLPL$^+$ to HLPL:

Theorem 3 (Forward Relation for HLPL and HLPL$^+$)**.** *For all Ω_l HLPL state, Ω_r HLPL$^+$ state:*

$$\Omega_l \leq \Omega_r \Rightarrow \forall\ s\ r\ \Omega'_r,\ \Omega_r \vdash_{\mathsf{hlpl+}}\ s \rightsquigarrow (r,\ \Omega'_r) \Rightarrow \exists\ \Omega'_l,\ \Omega_l \vdash_{\mathsf{hlpl}}\ s \rightsquigarrow (r,\ \Omega'_l) \wedge \Omega'_l \leq \Omega'_r$$

11.2.7 From LLBC to HLPL$^+$

We have just shown that the lower bound, Ω_l, remains in HLPL (by virtue of stability, Definition 2), and therefore $\leq$ can preserve the relation between HLPL and HLPL$^+$. It now remains to show the link between HLPL$^+$ and LLBC, i.e., the upper bound, Ω_h, reduces in LLBC in a way that preserves $\leq$.

Because HLPL$^+$ excludes several rules from LLBC (remember that some rules are marked **(LLBC only)**), LLBC is not a subset of HLPL$^+$, which prevents us from deriving that result instantly. Instead, we remark that we can still reconstruct the missing LLBC semantics using well-chosen combinations of evaluation rules from HLPL$^+$ and refinement rules from $\leq$. For instance, E-MutBorrow states that evaluating $\&\mathsf{mut}\, p$ in LLBC leads to a state with a mutable loan and a mutable borrow. We can build the same state by using the HLPL$^+$ rule E-Ptr to introduce a pointer and a location, then by using Le-MutBorrow-To-Ptr to transform this state into a *related* state (in the sense of $\leq$), by converting this pointer and this location to a borrow and a loan. This means that a reduction in LLBC can always be completed to correspond to a reduction in HLPL$^+$, which allows us to prove the following theorem stating that, given an LLBC state Ω, evaluating a statement following the semantics of LLBC leads to a state *in relation* with the state resulting from the HLPL$^+$ semantics.

Theorem 4 (Eval-LLBC-Preserves-Rel)**.** *For all Ω LLBC state we have:*

$$\forall\ s\, r\, \Omega_r,\ \Omega \vdash_{\mathsf{llbc}}\ s \rightsquigarrow (r,\ \Omega_r) \Rightarrow \exists\ \Omega_l,\ \Omega \vdash_{\mathsf{hlpl+}}\ s \rightsquigarrow (r,\ \Omega_l) \wedge \Omega_l \leq \Omega_r$$

Putting 3 and 4 together and using the transitivity of $\leq$ we finally get the preservation theorem we were aiming at, LLBC is in forward simulation with HLPL:

Theorem 5. *For all Ω_l HLPL state, Ω_r LLBC state we have:*

$$\forall\ s\, r\, \Omega'_r,\ \Omega_r \vdash_{\mathsf{llbc}}\ s \rightsquigarrow (r,\ \Omega'_r) \Rightarrow \exists\ \Omega'_l,\ \Omega_l \vdash_{\mathsf{hlpl}}\ s \rightsquigarrow (r,\ \Omega'_l) \wedge \Omega'_l \leq \Omega'_r$$

11.2.8 Form of Our Theorems

We take great care to start from *any* initial states Ω_l and Ω_r rather than requiring a program execution with a closed term and with an empty state (i.e., a "main" function). The reason is, as mentioned before, that we see the LLBC$^\#$ execution as borrow-checking, which we aim to perform modularly at the function-level granularity. This design for our theorems will later on allow us, as we connect LLBC$^\#$ all the way down to PL, to reason about the execution of a PL function that starts in an initial state *compatible with the lifetime signature* of the function in LLBC (Section 11.3.5).

Should we wish to do so, and using the fact that empty states are in relation with each other, we can specialize our main theorem to closed programs executing in the empty state.

11.2.9 The Pointer Language (PL)

So far, we have only proven the central arrow of Figure 11.1. We now describe the simulation between HLPL and PL; we do so briefly, since the techniques are standard, and do not leverage our new proof methodology. The semantics of PL and the full proof are in Appendix D. The PL language uses an explicit heap adapted from the CompCert memory model; this is similar to RustBelt, except we have no extra state to account for concurrency. As before, we retain the same syntax of programs; this is why we can, ultimately, conclude that LLBC is a sound restriction over the execution of PL programs.

For the proof of simulation between PL and HLPL, we have no choice but to introduce a global map from location identifiers to concrete addresses, along with an explicit notion of heap. In short, we adopt the traditional global-invariant style of proof; this is the only arrow from Figure 11.1 where we cannot apply our methodology. However, as we took care to design HLPL so that its pointers leave values in place, the structure of HLPL states and PL states is very close; as a consequence the proof is technical, but straightforward. It crucially leverages the fact that the rules of HLPL were carefully crafted so that it is not possible to move a location (see the premises of HLPL-E-Move or HLPL-E-Assign for instance), as it would break the relation between the HLPL locations and the PL addresses. Moving up the hierarchy of languages, this is the reason why we forbid moving *outer* loans in the LLBC rules, that is loans which are not inside a borrow (see E-Assign in particular). We elide the exact statements of the theorems, which can be found in Appendix D.

11.2.10 Divergence and Step-Indexed Semantics

The forward simulation theorem between PL and LLBC relates terminating executions. Anticipating on the next section where we will consider LLBC$^{\#}$ executions as borrow-checking certificates, this gives us in particular that the existence of safe executions for LLBC implies that the PL executions are safe. Unfortunately, as LLBC is defined in a big-step fashion, one problem arises. Big-step semantics do not allow to reason about programs that safely diverge, that is, programs that never get stuck or crash, but do not terminate: these programs cannot be distinguished from stuck programs; in both cases, no evaluation exists. This is a known problem when studying type systems;

$$\frac{}{\Omega \vdash s \underset{0}{\rightsquigarrow} \infty}\ \text{E-Step-Zero} \qquad \frac{}{\Omega \vdash \mathsf{return} \underset{n+1}{\rightsquigarrow} (\mathsf{return}, \Omega)}\ \text{E-Step-Return}$$

$$\frac{\Omega_0 \vdash s_0 \underset{n+1}{\rightsquigarrow} ((), \Omega_1) \qquad \Omega_1 \vdash s_1 \underset{n+1}{\rightsquigarrow} res}{\Omega_0 \vdash s_0; s_1 \underset{n+1}{\rightsquigarrow} res}\ \text{E-Step-Seq-Unit}$$

$$\frac{\Omega \vdash s \underset{n}{\rightsquigarrow} (\mathsf{continue}\, 0,\, \Omega') \qquad \Omega' \vdash \mathsf{loop}\, s \underset{n}{\rightsquigarrow} (r,\, \Omega'')}{\Omega \vdash \mathsf{loop}\, s \underset{n+1}{\rightsquigarrow} (r,\, \Omega'')}\ \text{E-Step-Loop-Continue-Inner}$$

$$\frac{\begin{array}{l} f\langle \_, \vec{\tau} \rangle = \mathsf{fn}\ \langle \_ \rangle\ (\overrightarrow{x_i} : \overrightarrow{\tau_i})\ (\overrightarrow{y_j} : \overrightarrow{\tau_j})\ (x_{\mathsf{ret}} : \tau)\ \{\ s\ \} \qquad \forall\, j, \Omega_j \vdash op_j \to (v_j,\ \Omega_{j+1}) \\ \quad \vdash \mathsf{push\_stack}\ \left([x_{\mathsf{ret}} \to \bot] \mathbin{+\!\!+} [\overrightarrow{x_j \to v_j}] \mathbin{+\!\!+} [\overrightarrow{y_k \to \bot}]\right) \Omega_m = \Omega^{(0)} \\ \Omega^{(0)} \vdash body \underset{n}{\rightsquigarrow} res \qquad res' = \begin{cases} (\mathsf{panic}, \Omega^{(1)}) & \text{if } res = (\mathsf{panic}, \Omega^{(1)}) \\ ((), \Omega^{(3)}) & \text{if } res = (\mathsf{return}, \Omega^{(1)}) \wedge \\ & \quad \vdash \mathsf{pop\_stack}\ \Omega^{(1)} = (v, \Omega^{(2)}) \wedge \\ & \quad \Omega^{(2)} \vdash p := v \rightsquigarrow ((), \Omega^{(3)}) \\ \infty & \text{if } res = \infty \end{cases} \end{array}}{\Omega_0 \vdash p := f(\overrightarrow{op_j}) \underset{n+1}{\rightsquigarrow} res'}\ \text{E-Step-Call}$$

Figure 11.6: Semantics of LLBC with Step-Indexing (Selected Rules)

previously proposed workarounds include defining semantics coinductively to model divergence [407, 408].

To avoid tricky coinductive reasoning, we instead rely on step-indexed semantics. We present in Figure 11.6 several of the updated rules for the step-indexed LLBC. As for related work, we add a step index to the judgment which evaluates statements (e.g., E-Step-Seq-Unit). This step index can be seen as a standard notion of fuel [409]; when the index is equal to zero, i.e., the execution is out of fuel, we stop the evaluation and return the ∞ value (E-Step-Zero). Otherwise, when evaluating possibly non-terminating statements (e.g., function calls as described by E-Step-Call), we decrement the step index in the recursive evaluation (e.g., the evaluation of the function body).

Building on this semantics, we can now define the following evaluation judgment that hides the step indices, and returns either ∞ for diverging computations, or the previously seen pair of a control flow tag and environment:

$$\Omega \vdash s \rightsquigarrow \infty := \forall\, n, \Omega \vdash s \underset{n}{\rightsquigarrow} \infty$$

$$\Omega \vdash s \rightsquigarrow res := \exists\, n, \Omega \vdash s \underset{n}{\rightsquigarrow} res \ \wedge\ res \neq \infty$$

We follow a similar approach to extend the PL and HLPL languages to model divergence. Adapting the proofs and theorems from §Section 11.2 to the step-indexed semantics and proving Theorem 6 is straightforward, and we omit the complete presentation for brevity. The core idea is that evaluations for the same program have identical step indices in PL, HLPL, and LLBC: the step index only decrements when entering a function call (or a loop), which is shared across the three languages.

Theorem 6. *For all Ω_l PL state and Ω_h LLBC state, we have:*

$$\Omega_l \leq \Omega_h \Rightarrow \forall\ s,\ \Omega_h \vdash_{\mathsf{llbc}}\ s \rightsquigarrow \infty \Rightarrow \Omega_l \vdash_{\mathsf{pl}}\ s \rightsquigarrow \infty$$

11.3 LLBC$^{\#}$ is a sound approximation, a.k.a., borrow-checker for LLBC

In the previous section we have showed that LLBC is a reasonable model of execution for Rust programs, as it can indeed be related to a traditional heap-and-addressses model of execution. In this section, we aim to study the claim that an interpreter for symbolic semantics, as defined by LLBC$^{\#}$, defines a sound borrow checker. Borrow checking is, conceptually, similar to type checking. A sound borrow checker for LLBC therefore ensures that, for a given program, if the borrow checker succeeds, then the program executes safely. In our setting, the definition of the borrow checking rules is however non-standard: instead of a set of inference rules, the borrow checker is formalized using a symbolic semantics. Importantly, this semantics is not deterministic; its implementation relies on several heuristics to choose the right rules to apply. This is to be contrasted with the current implementation of borrow-checking in the Rust compiler: while deterministic, it relies on a mostly lexical lifetime mechanism; we claim that our approach emphasizes *semantic* rather than *syntactic* borrow-checking.

We therefore aim to establish a forward simulation from LLBC$^{\#}$ to LLBC, that is, that for all successful LLBC$^{\#}$ evaluations for a given program, there exists a related, valid execution in LLBC. By composing this property with the forward simulation proven in the previous section, we therefore obtain that if the borrow-checker (LLBC$^{\#}$) succeeds for a given program, then this program safely executes at the PL level. The forward simulation from LLBC$^{\#}$ to LLBC also allows us to strengthen the relation between PL and LLBC: by definition of the simulation, successfully borrow checking a program implies the existence of a safe LLBC evaluation, which then allows us to

conclude that we actually have a bisimulation between PL and LLBC for programs that pass borrow checking.

In the rest of this section, we focus on applying the proof methodology from Section 11.1 to prove that LLBC and LLBC$^\#$ admit a forward simulation. We then show how to combine this result with the determinism of PL to obtain a bisimulation for PL and LLBC, under successful borrow-checking.

11.3.1 Simulation Relation

Considering LLBC$^\#$ as a borrow checker, we now aim to establish a property akin to type safety. As LLBC$^\#$ is defined as a semantics instead of a set of inference rules, this corresponds to a forward simulation. We assume that programs are executing in an environment $\mathcal{P}$, which consists of a set of function definitions alongside their signature. Formally, we aim to prove the following property:

Theorem 7. *For all states Ω and $\Omega^\#$, statement s, and $S^\#$ set of tagged states (i.e., pairs of a control-flow tag and a state), we have:*

$$
\begin{aligned}
&(\forall\, f \in \mathcal{P},\ \mathsf{borrow\_checks}\ f) \Rightarrow \Omega \leq \Omega^\# \Rightarrow \Omega^\# \vdash_{\mathsf{llbc}\#}\ s \rightsquigarrow S^\# \Rightarrow \\
&(\Omega \vdash_{\mathsf{llbc}}\ s \rightsquigarrow \infty)\ \vee\ (\exists\, \Omega_1,\ \Omega \vdash_{\mathsf{llbc}}\ s \rightsquigarrow (\mathsf{panic},\ \Omega_1)) \vee \\
&(\exists\, \Omega_1\, \Omega_1^\#\, r \in \{(), \mathsf{return}, \mathsf{break}\, i, \mathsf{continue}\, i\}, \\
&\ \Omega \vdash_{\mathsf{llbc}}\ s \rightsquigarrow (r,\ \Omega_1)\ \wedge\ \Omega_1 \leq \Omega_1^\#\ \wedge\ (r,\ \Omega_1^\#) \in S^\#)
\end{aligned}
$$

This property should be understood as follows. We assume that all functions appearing in the environment have been borrow-checked to match their signature, as represented by the predicate $\mathsf{borrow\_checks}\ f$; in practice, this can be done independently for each function, relying on the modularity of type-checking (see Section 11.3.3 below). Then for all states Ω in LLBC, $\Omega^\#$ in LLBC$^\#$ initially in relation, for all evaluations of a program s starting from $\Omega^\#$ and returning a set of abstract states $S^\#$, there exists a related execution of s in LLBC that either diverges, panics, or yields a result related to one of the states in $S^\#$. We note that, as in our case empty environments are trivially in relation, we get the standard typing result, that is: given some entry point to the program, say a $\mathsf{fn\ main}()$ function, an evaluation of the program starting in the empty state is safe.

11.3.2 Local Transformations

Similarly to the previous section, this proof will rely on the methodology outlined in Section 11.1. We describe below its different components. The first step in our methodology is to define a set of local transformations, whose reflexive transitive closure will allow turning a concrete LLBC state into its abstract LLBC$^{\#}$ counterpart; we present the rules in Figure 11.8. Note that some rules of Figure 11.8 are a bit too primitive; we will sometimes refer to rules of Figure 11.9, which can be trivially derived from those primitive rules. Similarly to how transformations between HLPL and LLBC led to the HLPL^{+} language, transformations between LLBC and LLBC$^{\#}$ commonly span hybrid states that do not belong to either language. We dub this hybrid, union language LLBC^{+}, and define transformations as operating on LLBC^{+} states. The semantics of LLBC^{+} is almost exactly the union of LLBC and LLBC$^{\#}$. To ensure that LLBC remains a stable subset of LLBC^{+}, a key ingredient of our approach, we exclude E-Call-Symbolic, the only rule introducing symbolic values and region abstractions; similarly to our handling of loans and borrows in HLPL^{+}, they will instead be introduced through the state transformations. We of course need to generalize some of the rules so that they work for hybrid states (Figure 11.7). We now detail several of the rules that induce the $\leq$ relation on LLBC^{+} states. Contrary to HLPL^{+} where we explained the transformations from right to left, in the case of LLBC^{+} it is more natural to go from left to right, from concrete to abstract; we adopt this convention in the rest of the section.

Le-ToSymbolic is one of the simplest transformation rules. If we have a plain, concrete value (i.e., that does not contain loans, borrows, or $\bot$), then we can lose information, and transform it into a fresh symbolic value. Le-MoveValue implements a move: so long as no one else relies on v (as captured by the premises), v can be moved out into an anonymous variable. Le-ToAbs captures the core rewriting to go from LLBC to LLBC$^{\#}$: it allows abstracting away parts of the borrow graph by moving borrows and loans associated to anonymous variables into fresh region abstractions. To do so, it relies on $\bowtie$, which we explain below, and on the $\prec^{\text{to-abs}}$ judgement, which transforms a (possibly complex) value v into a set of fresh region abstractions $\overrightarrow{A}$.

The $\prec^{\text{to-abs}}$ judgment relies on $\cup$, which is plain set union. ToAbs-MutLoan simply transfers the loan to a fresh region abstraction. Some values, such as pairs, may contain several loan identifiers, and as such, give rise to several, independent region abstractions (ToAbs-Pair). ToAbs-MutBorrow converts the inner borrowed value into a set of region abstractions, computes their union, then adds the outer mutable borrow to the result. It is particularly useful to abstract away reborrow patterns. Let us go back to

$$\textsc{Reorg-End-MutBorrow}\#\quad \frac{\text{hole of } \Omega[\mathsf{loan}^m\,\ell,\ .] \text{ not inside a borrowed value or a region abstraction} \qquad \text{no loan, } \mathsf{borrow}^r \in v}{\Omega[\mathsf{loan}^m\,\ell,\ \mathsf{borrow}^m\,\ell\ v] \hookrightarrow \Omega[v, \bot]}$$

$$\textsc{Reorg-End-SharedReservedBorrow}\#\quad \frac{\text{hole of } \Omega[.] \text{ not inside a borrowed value or a region abstraction}}{\Omega[\mathsf{borrow}^{s,r}\,\ell] \hookrightarrow \Omega[\bot]}$$

$$\textsc{Reorg-End-Abstraction}\quad \frac{\text{no borrows, loans} \in \overrightarrow{v},\ \overrightarrow{v'} \qquad \overrightarrow{\sigma} \text{ fresh}}{\Omega,\ A\,\{\ \overrightarrow{v},\ \overrightarrow{\mathsf{borrow}^s\,\ell},\ \overrightarrow{\mathsf{borrow}^m\,\ell'\ (v' : \tau)}\ \} \hookrightarrow \Omega,\ \overrightarrow{\_ \to \mathsf{borrow}^s\,\ell},\ \overrightarrow{\_ \to \mathsf{borrow}^m\,\ell'\ (\sigma : \tau)}}$$

$$\textsc{E-IfThenElse-Symbolic}\quad \frac{\Omega \vdash op \Downarrow (v,\ \Omega') \qquad v = (\sigma : \mathsf{bool}) \vee v = \mathsf{loan}^s\,\ell\,(\sigma : \mathsf{bool}) \qquad \Omega_0 = \Omega'[\mathsf{true}/\sigma] \qquad \Omega_1 = \Omega'[\mathsf{false}/\sigma] \qquad \Omega_0 \vdash s_0 \rightsquigarrow S_0^\# \qquad \Omega_1 \vdash s_1 \rightsquigarrow S_1^\#}{\Omega \vdash \mathsf{if}\ op\ \mathsf{then}\ s_0\ \mathsf{else}\ s_1 \rightsquigarrow S_0^\# \cup S_1^\#}$$

$$\textsc{E-Match-Symbolic}\quad \frac{\vdash \Omega(p) \overset{\mathsf{imm}}{\Rightarrow} v \qquad v = (\sigma : \tau_0 + \tau_1) \vee v = \mathsf{loan}^s\,\ell\,(\sigma : \tau_0 + \tau_1) \qquad \sigma_0,\ \sigma_1 \text{ fresh} \qquad \Omega_0 = \Omega'[\mathsf{Left}\,(\sigma_0 : \tau_0)/\sigma] \qquad \Omega_1 = \Omega'[\mathsf{Right}\,(\sigma_0 : \tau_0)/\sigma] \qquad \Omega_0 \vdash s_0 \rightsquigarrow S_0^\# \qquad \Omega_1 \vdash s_1 \rightsquigarrow S_1^\#}{\Omega \vdash (\mathsf{match}\,p\,\mathsf{with} \mid \mathsf{Left}\ \Rightarrow s_0 \mid \mathsf{Right}\ \Rightarrow s_1) \rightsquigarrow S_0^\# \cup S_1^\#}$$

$$\textsc{E-Seq-Symbolic}\quad \frac{\Omega \vdash s_0 \rightsquigarrow \{((), \Omega_i)\} \cup S^\# \qquad \forall\, r \in S^\#,\ \forall\, \Omega,\ r \neq ((), \Omega) \qquad \forall\, i,\ \Omega_i \vdash s_1 \rightsquigarrow S_i^\#}{\Omega \vdash s_0;\ s_1 \rightsquigarrow S^\# \cup (\bigcup_i S_i^\#)}$$

$$\textsc{Copy-Symbolic}\quad \frac{\sigma' \text{ fresh}}{\vdash \mathsf{copy}\,\sigma = \sigma'} \qquad \textsc{Reorg-SymbolicBox}\quad \frac{\sigma' \text{ fresh}}{\Omega[\sigma : \mathsf{Box}\,\tau] \hookrightarrow \Omega[\mathsf{Box}\,\sigma']} \qquad \textsc{Reorg-SymbolicPair}\quad \frac{\sigma_0,\ \sigma_1 \text{ fresh}}{\Omega[\sigma : (\tau_0,\ \tau_1)] \hookrightarrow \Omega[(\sigma_0,\ \sigma_1)]}$$

Figure 11.7: Operational Semantics for LLBC$^\#$

$$\textsc{Le-ToSymbolic}\ \frac{\text{borrows, loans, } \bot \notin v \qquad \sigma \text{ fresh}}{\Omega[v] \leq \Omega[\sigma]} \qquad \textsc{Le-ToAbs}\ \frac{\vdash v \prec^{\text{to-abs}} \overrightarrow{A}}{\Omega,\ \_ \rightarrow v \leq \Omega,\ \overrightarrow{A}}$$

$$\textsc{Le-MoveValue}\ \frac{\text{no outer loans in } v \qquad \text{hole of } \Omega[.] \text{ not inside a shared loan or a region abstraction}}{\Omega[v] \leq \Omega[\bot],\ \_ \mapsto v} \qquad \textsc{Le-ClearAbs}\ \frac{}{\Omega,\ A\ \{\} \leq \Omega}$$

$$\textsc{Le-MergeAbs}\ \frac{\vdash A_0 \bowtie A_1 = A}{\Omega,\ A_0,\ A_1 \leq \Omega,\ A} \qquad \textsc{Le-Fresh-MutLoan}\ \frac{\ell \text{ fresh}}{\Omega[v] \leq \Omega[\mathsf{loan}^m\ \ell],\ \_ \rightarrow \mathsf{borrow}^m\ \ell\ v} \qquad \textsc{Le-Fresh-SharedLoan}\ \frac{\ell \text{ fresh}}{\Omega[v] \leq \Omega[\mathsf{loan}^s\ \ell\ v]}$$

$$\textsc{Le-Reborrow-MutBorrow}\ \frac{\ell_1 \text{ fresh}}{\Omega[\mathsf{borrow}^m\ \ell_0\ v] \leq \Omega[\mathsf{borrow}^m\ \ell_1\ v],\ \_ \rightarrow \mathsf{borrow}^m\ \ell_0\ (\mathsf{loan}^m\ \ell_1)} \qquad \textsc{Le-Abs-ClearValue}\ \frac{\text{no borrows, loans } \in v}{\Omega,\ A \cup \{\ v\ \} \leq \Omega,\ A}$$

$$\textsc{Le-Fresh-SharedBorrow}\ \frac{}{\Omega[\mathsf{loan}^s\ \ell\ v] \leq \Omega[\mathsf{loan}^s\ \ell\ v],\ \_ \rightarrow \mathsf{borrow}^s\ \ell}$$

$$\textsc{Le-Reborrow-SharedLoan}\ \frac{A,\ \ell_1,\ \sigma \text{ fresh} \qquad \bot, \mathsf{loan}^m, \mathsf{borrow}^{s,r,m} \notin v}{\Omega[\mathsf{loan}^s\ \ell_0\ v,\ \overrightarrow{\mathsf{borrow}^s\ \ell_0}] \leq \Omega[\mathsf{loan}^s\ \ell_1\ \sigma,\ \overrightarrow{\mathsf{borrow}^s\ \ell_1}],\ A\ \{\ \mathsf{borrow}^s\ \ell_1,\ \mathsf{loan}^s\ \ell_0\ v\ \}}$$

$$\textsc{Le-Abs-End-SharedLoan}\ \frac{\Omega = \Omega',\ A[\mathsf{loan}^s\ \ell\ v] \qquad \text{no } \mathsf{borrow}^{s,r}\ \ell \in \Omega}{\Omega \leq \Omega',\ A[v]}$$

$$\textsc{Le-Abs-End-DupSharedBorrow}\ \frac{}{\Omega,\ A \cup \{\ \mathsf{borrow}^s\ \ell,\ \mathsf{borrow}^s\ \ell\ \} \leq \Omega,\ A \cup \{\ \mathsf{borrow}^s\ \ell\ \}}$$

$$\textsc{Le-Reborrow-SharedBorrow}\ \frac{\bot, \mathsf{loan}^m, \mathsf{borrow}^{s,r,m} \notin v \qquad \mathsf{loan}^s\ \ell_0\ v \in \Omega[\overrightarrow{\mathsf{borrow}^s\ \ell_0}] \qquad \ell_1,\ \sigma,\ A \text{ fresh}}{\Omega[\overrightarrow{\mathsf{borrow}^s\ \ell_0}] \leq \Omega[\overrightarrow{\mathsf{borrow}^s\ \ell_1}],\ A\ \{\mathsf{borrow}^s\ \ell_0,\ \mathsf{loan}^s\ \ell_1\ \sigma\ \}}$$

$$\textsc{Le-Abs-DeconstructPair}\ \frac{}{\Omega,\ A \cup \{\ (v_0,\ v_1)\ \} \leq \Omega,\ A \cup \{\ v_0,\ v_1\ \}} \qquad \textsc{Le-Abs-DeconstructSum}\ \frac{C \in \{\mathsf{Left},\ \mathsf{Right}\}}{\Omega,\ A \cup \{\ C\ v\ \} \leq \Omega,\ A \cup \{\ v\ \}}$$

$$\textsc{Le-AnonValue}\ \frac{\text{no symbolic values, borrows, loans} \in v}{\Omega \leq \Omega,\ \_ \rightarrow \bot}$$

Figure 11.8: The Relation $\leq$ about LLBC$^+$ States

$$\textsc{Le-Reborrow-MutBorrow-Abs}\ \frac{\ell_1,\ A \text{ fresh}}{\Omega[\mathsf{borrow}^m\ \ell_0\ v] \leq \Omega[\mathsf{borrow}^m\ \ell_1\ v],\ A\,\{\,\mathsf{borrow}^m\ \ell_0\ \_,\ \mathsf{loan}^m\ \ell_1\,\}}$$

$$\textsc{Le-Reborrow-MutLoan-Abs}\ \frac{\ell_1,\ A \text{ fresh}}{\Omega[\mathsf{loan}^m\ \ell_0] \leq \Omega[\mathsf{loan}^m\ \ell_1],\ A\,\{\,\mathsf{borrow}^m\ \ell_1\ \_,\ \mathsf{loan}^m\ \ell_0\,\}}$$

$$\textsc{Le-Fresh-MutLoan-Abs}\ \frac{\ell,\ A \text{ fresh}}{\Omega[v] \leq \Omega[\mathsf{loan}^m\ \ell],\ A\,\{\,\mathsf{borrow}^m\ \ell\ v\,\}}$$

$$\textsc{Le-DeconstructSharedLoans}\ \frac{\sigma \text{ fresh symbolic value}}{\Omega, A \cup \{\ V_0[\mathsf{loan}^s\ \ell_0\ (V_1[\mathsf{loan}^s\ \ell_1\ v])]\ \} \leq \Omega, A \cup \{\ V_0[\mathsf{loan}^s\ \ell_0\ (V_1[\sigma])],\ \mathsf{loan}^s\ \ell_1\ v\ \}}$$

Figure 11.9: Derived Rules for the $\leq$ Relation About LLBC$^+$ States

$$\textsc{ToAbs-Empty}\ \frac{\text{no borrows, loans} \in v}{\vdash v \prec^{\text{to-abs}} \emptyset} \qquad \textsc{ToAbs-Sum}\ \frac{v = \mathsf{Left}\ v' \vee v = \mathsf{Right}\ v' \qquad \vdash v' \prec^{\text{to-abs}} \overrightarrow{A}}{\vdash v \prec^{\text{to-abs}} \overrightarrow{A}}$$

$$\textsc{ToAbs-Box}\ \frac{\vdash v \prec^{\text{to-abs}} \overrightarrow{A}}{\vdash \mathsf{Box}\ v \prec^{\text{to-abs}} \overrightarrow{A}} \qquad \textsc{ToAbs-SharedBorrow}\ \frac{A \text{ fresh}}{\vdash \mathsf{borrow}^s\ \ell \prec^{\text{to-abs}} A\{\mathsf{borrow}^s\ \ell\}}$$

$$\textsc{ToAbs-SharedLoan}\ \frac{A \text{ fresh}}{\vdash \mathsf{loan}^s\ \overrightarrow{\ell}\ v \prec^{\text{to-abs}} A\{\mathsf{loan}^s\ \overrightarrow{\ell}\ v\}} \qquad \textsc{ToAbs-MutBorrow}\ \frac{\text{no borrows}, \bot \in v \qquad v \prec^{\text{to-abs}} \overrightarrow{A}}{\vdash \mathsf{borrow}^m\ \ell\ v \prec^{\text{to-abs}} (\cup \overrightarrow{A}) \cup \{\mathsf{borrow}^m\ \ell\ \_\}}$$

$$\textsc{ToAbs-MutLoan}\ \frac{A \text{ fresh}}{\vdash \mathsf{loan}^m\ \ell \prec^{\text{to-abs}} A\{\mathsf{loan}^m\ \ell\}} \qquad \textsc{ToAbs-Pair}\ \frac{\vdash v_l \prec^{\text{to-abs}} \overrightarrow{A}_l \qquad \vdash v_r \prec^{\text{to-abs}} \overrightarrow{A}_r}{\vdash (v_l, v_r) \prec^{\text{to-abs}} \overrightarrow{A}_l,\ \overrightarrow{A}_r}$$

Figure 11.10: Rules to Transform Values to Region Abstractions.

$$\frac{}{\vdash A_0 \bowtie A_1 = A_0 \cup A_1}\ \textsc{MergeAbs-Union} \qquad \frac{\vdash A_0 \bowtie A_1 = A}{\vdash (A_0 \cup \{\mathsf{loan}^m\ \ell\}) \bowtie (A_1 \cup \{\mathsf{borrow}^m\ \ell\ \_\}) = A}\ \textsc{MergeAbs-Mut}$$

$$\frac{\vdash (A_0 \cup \{\mathsf{loan}^s\ \ell\ v\}) \bowtie A_1 = A}{\vdash (A_0 \cup \{\mathsf{loan}^s\ \ell\ v\}) \bowtie (A_1 \cup \{\mathsf{borrow}^s\ \ell\}) = A}\ \textsc{MergeAbs-Shared}$$

Figure 11.11: Rules to Merge Region Abstractions.

the reborrowing example of Section 11.2.3:

```
1  x  = 0;            // x ↦ 0
2  px = &mut x;       // x ↦ loan^m ℓ0,   px ↦ borrow^m ℓ0 0
3  px = &mut (*px);   // x ↦ loan^m ℓ0,   _ ↦ borrow^m ℓ0 (loan^m ℓ1),   px ↦ borrow^m ℓ1 0
4  assert!(x = 0);    // x ↦ 0,   _ ↦ ⊥,   px ↦ ⊥
```

After the reborrow (line 3) we have the state: $x \mapsto \mathsf{loan}^m\ \ell_0$, $\_ \mapsto \mathsf{borrow}^m\ \ell_0\ (\mathsf{loan}^m\ \ell_1)$, $px \mapsto \mathsf{borrow}^m\ \ell_1\ 0$. We can abstract away the link between px and x by using Le-ToAbs, resulting in the state: $x \mapsto \mathsf{loan}^m\ \ell_0$, $A\,\{\,\mathsf{borrow}^m\ \ell_0\ \_,\ \mathsf{loan}^m\ \ell_1\,\}$, $px \mapsto \mathsf{borrow}^m\ \ell_1\ 0$.

The non-deterministic $\bowtie$ operator *merges* two different region abstractions using semantic criteria. When interpreting Rust programs, a region abstraction can be understood as a set of values associated to a given lifetime. Merging two region abstractions therefore corresponds to a notion of lifetime weakening: if we have two distinct lifetimes, Rust allows adding lifetime constraints to guarantee that borrows associated to both lifetimes will be ended at the same time. This pattern frequently occurs when typechecking a function body against a more restrictive lifetime signature.

A naive, but sound interpretation of merging regions A_0 and A_1 consists of taking the union of all values in both regions using rule MergeAbs-Union. However, we can also perform more precise transformations, for instance by removing a mutable loan and its associated borrow (MergeAbs-Mut). Intuitively, this rule allows hiding them in the internals of the region abstraction: ending the merged abstraction amounts, in the original state, to ending the first abstraction, all the borrows and loans that were hidden by means of MergeAbs-Mut, then the second abstraction; in the state resulting from the merge, we simply abstract all those steps away.

In practice, we devise an algorithm to judiciously apply the $\bowtie$ rules. Indeed, greedily applying MergeAbs-Union instead of leveraging MergeAbs-Mut creates an abstraction which contains both a loan and its associated borrow; we can never end such abstractions because of a cyclic dependency between ending the abstraction and ending the borrow, eventually leading the symbolic evaluation to get stuck. We

$$
\begin{array}{l}
\mathsf{borrow\_checks}\ (\mathsf{fn}\ \langle \overrightarrow{\rho} \rangle\ (\overrightarrow{x} : \overrightarrow{\tau})\ (\overrightarrow{y} : \overrightarrow{\tau'})\ (x_{\mathsf{ret}} : \tau_{\mathsf{ret}})\ \{\ s\ \}) := \\
\quad \mathsf{let}\ \vec{v},\ \overrightarrow{A_{\mathsf{in}}(\rho)} = \mathsf{init}(\vec{\rho}, \vec{\tau}) \\
\quad \mathsf{let}\ \Omega_0^{\#} = \overrightarrow{A_{\mathsf{in}}(\rho)},\ \overrightarrow{x \to v},\ \overrightarrow{y \to \bot},\ x_{\mathsf{ret}} \to \bot \\
\quad \mathsf{let}\ v_{\mathsf{out}},\ \overrightarrow{A_{\mathsf{sig}}(\rho)} = \mathsf{final}(\vec{\rho}, \vec{\tau}, \tau_{\mathsf{ret}}) \\
\quad \mathsf{let}\ \Omega_1^{\#} = \overrightarrow{A_{\mathsf{sig}}(\rho)},\ \overrightarrow{x \to \bot},\ \overrightarrow{y \to \bot},\ x_{\mathsf{ret}} \to v_{\mathsf{out}} \\
\quad \exists\ S^{\#},\ \Omega_0^{\#} \vdash_{\mathsf{llbc\#}} s \rightsquigarrow S^{\#} \wedge \\
\qquad \forall\ res \in S^{\#},\ \exists\ \Omega^{\#},\ res = (\mathsf{panic},\ \Omega^{\#})\ \vee\ (res = (\mathsf{return},\ \Omega^{\#})\ \wedge\ \Omega^{\#} \leq \Omega_1^{\#})
\end{array}
$$

Figure 11.12: The Borrow Checking Predicate For Functions

emphasize that $\bowtie$ is not symmetric: MergeAbs-Mut is directed, and there is no version of it with a borrow on the left, and a loan on the right. This is necessary for soundness. Consider environments $A_0 = \{\mathsf{borrow}^m\ \ell_2\ v,\ \mathsf{loan}^m\ \ell_0,\ \mathsf{borrow}^m\ \ell_1\ v_1\}$ and $A_1 = \{\mathsf{loan}^m\ \ell_1,\ \mathsf{borrow}^m\ \ell_0\ v_0\}$. This symbolic state features a cyclic dependency (perhaps, because of poorly chosen uses of Le-MergeAbs) – it is thus crucial, for our proof of forward simulation, to make sure that such a symbolic state remains stuck. The directionality of MergeAbs-Mut guarantees just that: it allows terminating ℓ_0, but not ℓ_1, and rightfully prevents borrow-checking from succeeding.

Building on the transformations previously presented, we now focus on the simulation proof itself. As outlined in Section 11.1, the proof can be broken down in three steps: we need to establish a forward simulation on LLBC$^+$, prove that LLBC is a stable subset of LLBC$^+$, and conclude by reasoning on the remaining rules in LLBC$^\#$ that were excluded from LLBC$^+$.

The second point can be easily obtained by induction on an LLBC$^+$ evaluation. The first point is tedious, but straightforward and similar to the proof between HLPL and LLBC. The core idea is that the local transformations turn a state into a more abstract version, thus allowing fewer execution steps. Compared to HLPL$^+$, the novel part of the proof is the third point, which requires relating an abstract, modular execution of a function call (E-Call-Symbolic, Figure 11.7) to its concrete counterpart which enters the function body (E-Step-Call, Figure 11.6). Before doing so, we focus briefly on how exactly symbolic execution works, and show how to modularly borrow-check a function in LLBC$^\#$.

11.3.3 Borrow-Checking a Program

The actual setup of the symbolic execution is performed by borrow_checks (Figure 11.12). The init function (Figure 11.13), given the types of the arguments $\vec{\tau}$, initializes a set of

$$\textsc{Init}\quad \frac{\forall\, i,\ \mathsf{init\_extern}\ \vec{\rho}\ \tau_i = \overrightarrow{A_i^{\text{ext}}(\rho)} \qquad \forall\, i,\ \mathsf{proj\_output}\ \vec{\rho}\ \tau_i = (v_i,\ \overrightarrow{A_i^{\text{out}}(\rho)}) \qquad \forall\, \rho,\ A(\rho) = \bigcup_i \left(A_i^{\text{ext}}(\rho) \cup A_i^{\text{out}}(\rho)\right)}{\mathsf{init}\ (\vec{\rho},\ \vec{\tau}) = (\overrightarrow{v_i},\ \overrightarrow{A(\rho)})}$$

$$\textsc{Final}\quad \frac{\forall\, i,\ \mathsf{init\_extern}\ \vec{\rho}\ \tau_i = \overrightarrow{A_i^{\text{ext}}(\rho)} \qquad \mathsf{proj\_output}\ \vec{\rho}\ \tau = (v,\ \overrightarrow{A^{\text{out}}(\rho)}) \qquad \forall\, \rho,\ A(\rho) = A^{\text{out}}(\rho) \cup \left(\bigcup_i A_i^{\text{ext}}(\rho)\right)}{\mathsf{final}\ (\vec{\rho},\ \vec{\tau},\ \tau) = (v,\ \overrightarrow{A(\rho)})}$$

Figure 11.13: Init and Final LLBC# States for Borrow-Checking

$$\textsc{InitExtern-Symbolic}\quad \frac{\text{no borrows} \in \tau \qquad \forall\, \rho,\ A_\rho = \{\}}{\mathsf{init\_extern}\ \vec{\rho}\ \tau = \overrightarrow{A(\rho)}}$$

$$\textsc{InitExtern-Pair}\quad \frac{\mathsf{init\_extern}\ \vec{\rho}\ \tau_0 = \overrightarrow{A_0(\rho)} \qquad \mathsf{init\_extern}\ \vec{\rho}\ \tau_1 = \overrightarrow{A_1(\rho)} \qquad \forall\, \rho,\ A(\rho) = A_0(\rho) \cup A_1(\rho)}{\mathsf{init\_extern}\ \vec{\rho}\ (\tau_0,\ \tau_1) = \overrightarrow{A(\rho)}}$$

$$\textsc{InitExtern-Shared}\quad \frac{\text{no borrows} \in \tau \qquad \ell \text{ fresh} \qquad A(\rho) = \{\ \mathsf{borrow}^s\ \ell\ \} \qquad \forall\, \rho' \neq \rho,\ A(\rho') = \{\}}{\mathsf{init\_extern}\ \vec{\rho}\ (\&\rho\, \tau) = \overrightarrow{A(\rho)}}$$

$$\textsc{InitExtern-Mut}\quad \frac{\text{no borrows} \in \tau \qquad \ell \text{ fresh} \qquad A(\rho) = \{\ \mathsf{borrow}^m\ \ell\ \_\ \} \qquad \forall\, \rho' \neq \rho,\ A(\rho') = \{\}}{\mathsf{init\_extern}\ \vec{\rho}\ (\&\rho\, \mathsf{mut}\, \tau) = \overrightarrow{A(\rho)}}$$

Figure 11.14: Generating the External Borrows for State Initialization

input values $\vec{v}$ by allocating fresh symbolic values and borrows, along with an initial set of region abstractions $\overrightarrow{A_{\mathsf{in}}(\rho)}$ which contain their associated loans, and materialize the signature and its lifetimes; this then serves to create the initial symbolic environment $\Omega_0^\#$, where the input parameters are initialized with $\vec{v}$, and where the remaining local variables and the special return variable are uninitialized. Symmetrically, $\Omega_1^\#$ captures the expected region abstractions upon exiting the function, and states that all that matters is that the return value points to a symbolic variable. The function then borrow-checks if executing the body s in $\Omega_0^\#$ is safe, and leads to states which either panic or are in relation with $\Omega_1^\#$.

We illustrate borrow-checking on the choose function, below. The init function computes an initial state that corresponds to the function signature, lines 2-3. The function has no local variables, so the $\vec{y}$ from borrow_checks are absent. The return value is uninitialized (line 3).

In this state, the variable b contains a symbolic value σ_b, while x and y borrow some other symbolic values, their associated loans being placed in a region abstraction A_{in} which holds all the loans of lifetime α; as choose has only one lifetime, the initial state holds a unique region abstraction. Importantly, A_{in} also contains some borrows $\ell_x^{(0)}$ and $\ell_y^{(0)}$, whose corresponding loans are in an *unspecified* place (i.e., not in the current state). As such, this initial state is a partial state that can be composed with other partial states to form a complete state, where in particular all borrows have an associated loan; we will use this in the proof of the forward simulation, by applying framing lemmas in the same spirit as the frame rule in separation logic [410]. In this context, A_{in} really acts as an abstraction barrier between the local state (the callee) and some external state (the caller); intuitively, $\ell_x^{(0)}$ should be exactly ℓ_x while $\ell_y^{(0)}$ should be exactly ℓ_y, but we abstracted this information away.

```
1  fn choose<'a, T>(b : bool, x : &'a mut T, y : &'a mut T) -> &'a mut T {
2    // A_in { borrow^m l_x^(0) _, borrow^m l_y^(0) _, loan^m l_x, loan^m l_y },
3    // b ↦ σ_b,   x ↦ borrow^m l_x σ_x,   y ↦ borrow^m l_y σ_y,   x_ret ↦ ⊥
4    if b { ret := move x;
5      // A_in { borrow^m l_x^(0) _, borrow^m l_y^(0) _ loan^m l_x, loan^m l_y },
6      // b ↦ true,   x ↦ ⊥,   y ↦ borrow^m l_y σ_y,   x_ret ↦ borrow^m l_x σ_x
7      return;
8      // ≤ A_out { borrow^m l_x^(0) _, borrow^m l_y^(0) _, loan^m l },
9      //    b ↦ ⊥,   x ↦ ⊥,   y ↦ ⊥,   x_ret ↦ borrow^m l σ
10   } else { ret := move y; return; } }
```

Upon reaching the return, symbolic execution yields the state at lines 5-6, where the special return value variable ret now contains the borrow coming from x. We show at lines 8-9 the target output state, computed via final (Figure 11.13): the goal is now to

establish that the final environment (as computed by the symbolic execution) refines the output environment (as determined by the function signature).

We first reorganize the context by ending all the outer loans (loans which are themselves not inside borrows) which we see in local variables. We then repeatedly compare the two states while applying $\leq$ rules until they match: we move the values contained by the local variables (except ret) into fresh region abstractions (Le-MoveValue), transform the values contained by the return variable ret into symbolic values (Le-ToSymbolic), and merge region abstractions together (Le-MergeAbs). We end up in the target state of lines 8-9; we then do the same for the else branch (omitted), which allows us to conclude that choose satisfies its signature. In other words, choose borrow-checks.

11.3.4 Forward Simulation Between LLBC^{+} and LLBC$^{\#}$

We now resume the presentation of the proof of the simulation between LLBC and LLBC$^{\#}$: there remains to show that we can replace E-Call with E-Call-Symbolic. There are several crucial points. First, evaluation rules are local, which means the evaluation relation is also defined for partial states in which borrows don't necessarily have an associated loan; equipped with partial states, we define and prove a framing lemma in the spirit of separation logic, which states that if state $\Omega_0^{\#}$ evaluates to $\Omega_1^{\#}$, we can compose $\Omega_0^{\#}$ with a disjoint frame $\Omega_f^{\#}$ which doesn't get modified during the evaluation. As the $\leq$ rules are also local, we define a similar framing property for the $\leq$ relation. Finally, we carefully designed E-Call, E-Call-Symbolic and the borrow_checks predicate so that: 1. we can always turn a part of the concrete input state into a more abstract partial state that is in relation with $\Omega_0^{\#}$; 2. the conclusion of borrow_checks gives us exactly the partial state we need to get the state resulting from E-Call-Symbolic by applying the frame rules. There are some other technicalities; we refer the interested reader to the Appendix.

11.3.5 Backward Simulation From LLBC to PL

The forward simulation from LLBC$^{\#}$ to LLBC allows us to conclude about the soundness of LLBC$^{\#}$ as a borrow-checker. However, it also guarantees the existence of a backward simulation between LLBC and PL for programs that successfully borrow-check; we formalize this in Theorem 8. This theorem states that, assuming that all functions in program P borrow-check, and that the state Ω^{llbc} is in relation with the initial borrow-checking state $\Omega_0^{\#}$ for function g, then any PL execution of g starting from a related state Ω^{pl} has a related LLBC execution. Combined with the forward simulation

from LLBC to PL proven in Section 11.2, this provides the main result of our paper: we have a bisimulation relation between LLBC and PL for programs that borrow-check.

Theorem 8 (Backward Simulation Between PL and LLBC)**.** *For all Ω^{pl} PL state and Ω^{llbc} LLBC state, for all function $g\langle\vec{\rho}, \vec{\tau}\rangle$, we have:*

$$\mathsf{let}\ \vec{v},\ \overrightarrow{A_{\mathsf{in}}(\rho)} = \mathsf{init}(\vec{\rho}, \vec{\tau});\ \ \mathsf{let}\ \Omega_0^{\#} = \overrightarrow{A_{\mathsf{in}}(\rho)},\ \overrightarrow{x \to v},\ \overrightarrow{y \to \bot},\ x_{\mathsf{ret}} \to \bot;$$
$$(\forall\ f \in \mathcal{P},\ \mathsf{borrow\_checks}\ f) \Rightarrow \Omega^{pl} \leq \Omega^{llbc} \Rightarrow \Omega^{llbc} \leq \Omega_0^{\#} \Rightarrow \forall\ res,\ \Omega^{pl} \vdash g.\mathsf{body} \rightsquigarrow res \Rightarrow$$
$$\exists\ res',\ \Omega^{llbc} \vdash g.\mathsf{body} \rightsquigarrow res' \wedge$$
$$(res = res' = \infty)\ \vee\ (\exists\ r\ \Omega_1^{pl}\ \Omega_1^{llbc},\ res = (r,\ \Omega_1^{pl})\ \wedge\ res' = (r,\ \Omega_1^{llbc})\ \wedge\ \Omega_1^{pl} \leq \Omega_1^{llbc})$$

Building on the previous sections, the proof is straightforward: if all functions in program $\mathcal{P}$ borrow-check, so does g, which by definition means that there exists an LLBC$^{\#}$ evaluation of $g.\mathsf{body}$ starting from $\Omega_0^{\#}$. By forward simulation, this implies the existence of an LLBC evaluation starting from Ω^{llbc} and returning a result res_{llbc}, which in turn implies the existence of a related PL evaluation starting from Ω^{pl} and returning res_{pl}. As PL is deterministic, we derive $res = res_{pl}$ for any PL execution starting from Ω^{pl}, and conclude by exhibiting the witness $res' = res_{llbc}$.

Chapter 12

Joins and Loops

In the previous chapter we proved that the symbolic execution defined by LLBC$^\#$ can be used to provably implement a borrow-checker. We are now ready to lift the restrictions we imposed on LLBC$^\#$ for loops and branchings (Chapter 10).

More precisely, equipped with a framework in which to reason about the soundness of LLBC$^\#$ with regards to LLBC, we define and prove correct a new operation that was previously missing from LLBC$^\#$: the join operation, which can reconcile two branches of control-flow (Section 12.1). Inspired by joins in abstract interpretation (and specifically, joins in shape analysis), this new operation gives us a symbolic interpreter (i.e., a borrow-checker) that can handle loops (Section 12.2), and does not exhibit pathological complexity on conditionals. Furthermore, our support naturally extends to the functional translation, meaning that Aeneas now supports verification of loops. We leave a discussion of the correctness of the Aeneas functional translation to future work.

We evaluate the effectiveness of our join operation (Chapter 14), specifically when used to compute shape fixed-points for loops, over a series of small examples and case studies. We find that, in spite of being (naturally) incomplete, our join operation handles a wide range of examples. Our interpretation is that because Rust imposes so many invariants, a join procedure can leverage this structure and fare better than, say, a general-purpose join operation for analyzing C programs.

As we saw in the previous section, LLBC$^\#$ offers a sound, modular borrow checker for the LLBC semantics. However, its approach based on a symbolic collecting semantics struggles with scalability when considering disjunctive control flow. Consider the rule E-IfThenElse-Symbolic, which defines the LLBC$^\#$ semantics for evaluating common if-then-else constructs. This rule evaluates both branches independently, yielding sets of tagged states $S_0^\#$ and $S_1^\#$, and finally returns their union; each state in the resulting set is then considered as a starting point to evaluate the rest of the program. When a program contains many branching constructs, this leads to a combinatorial explosion

known to symbolic execution practitioners as the path explosion problem. This issue becomes even worse when considering loops, as the number of paths to analyze can be infinite.

To circumvent this issue, one possible solution is to merge control flow paths at different program points, typically after the end of a branching statement. Doing so requires soundly merging environments or, borrowing terminology from abstract interpretation, "computing a join" [171]. In this section, we show how to define such an operator for abstract, borrow-centric environments. Leveraging the forward simulation from LLBC$^\#$ to LLBC, we then prove its soundness with respect to the LLBC semantics.

12.1 Joining Environments

We present in Figure 12.1 and Figure 12.2 the rules for our join operator. To make the presentation easier to follow, we will rely on the following small example, which is a standard Rust pattern after desugaring. We annotate this program with the LLBC$^\#$ environments at relevant program points.

```
// b ↦ (σ_0 : bool)
x = 0;
y = 1;
px = &mut x;
py = &mut y;
// b ↦ (σ_0 : bool), x ↦ loan^m ℓ_0, y ↦ loan^m ℓ_1,
// px ↦ borrow^m ℓ_0 0, py ↦ borrow^m ℓ_1 1
if b {
  p = move px;
  // b ↦ true, x ↦ loan^m ℓ_0, y ↦ loan^m ℓ_1,
  // px ↦ ⊥, py ↦ borrow^m ℓ_1 1, p ↦ borrow^m ℓ_0 0
} else {
  p = move py;
  // b ↦ false, x ↦ loan^m ℓ_0, y ↦ loan^m ℓ_1,
  // px ↦ borrow^m ℓ_0 0, py ↦ ⊥, p ↦ borrow^m ℓ_1 1
}
```

Our goal is to compute the join of the environments after evaluating both branches. Formally, we write $\Omega,\ \Omega' \vdash \mathsf{join}_\Omega\ \Omega_0\ \Omega_1 \rightsquigarrow \Omega_2$ to denote that the join of environments Ω_0 and Ω_1 yields a new environment Ω_2. We define the join operator inductively on the environments. The states Ω and Ω' on the left of the turnstile correspond to the top-level environments being joined, which might differ from Ω_0 and Ω_1 inside recursive derivations. A few specific rules in our complete presentation require access to them, but they can be safely ignored in this section.

$$\textsc{Join-Same}\quad \frac{}{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ v\ v \Downarrow v \mid \emptyset}$$

$$\textsc{Join-Symbolic}\quad \frac{\text{no borrows, loans, } \bot \in v_0,\ v_1 \qquad \sigma \text{ fresh}}{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ v_0\ v_1 \Downarrow \sigma \mid \emptyset}$$

$$\textsc{Join-Bottom-Other}\quad \frac{\text{no outer loan} \in v \qquad \vdash v \prec^{\text{to-abs}} \overrightarrow{A}}{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ \bot\ v \Downarrow \bot \mid \overrightarrow{\boxed{A}}}$$

$$\textsc{Join-Other-Bottom}\quad \frac{\text{no outer loan} \in v \qquad \vdash v \prec^{\text{to-abs}} \overrightarrow{A}}{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ v\ \bot \Downarrow \bot \mid \overrightarrow{\boxed{A}}}$$

$$\textsc{Join-MutBorrows}\quad \frac{\ell_2,\ A' \text{ fresh} \qquad \Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ v_0\ v_1 \Downarrow v_2 \mid \overrightarrow{A}}{\Omega_0, \Omega_1 \vdash \mathsf{join}_v\ (\mathsf{borrow}^m\ \ell_0\ v_0)\ (\mathsf{borrow}^m\ \ell_1\ v_1) \Downarrow \mathsf{borrow}^m\ \ell_2\ v_2 \mid A'\,\{\boxed{\mathsf{borrow}^m\ \ell_0\ \_},\ \boxed{\mathsf{borrow}^m\ \ell_1\ \_},\ \mathsf{loan}^m\ \ell_2\,\},\ \overrightarrow{A}}$$

$$\textsc{Join-SharedBorrows}\quad \frac{\ell_2,\ \sigma,\ A \text{ fresh} \qquad \text{no } \mathsf{loan}^m,\ \text{borrow},\ \bot \in v_0,\ v_1 \qquad \mathsf{loan}^s\ \ell_0\ v_0 \in \Omega_0 \qquad \mathsf{loan}^s\ \ell_1\ v_1 \in \Omega_1}{\Omega_0, \Omega_1 \vdash \mathsf{join}_v\ (\mathsf{borrow}^s\ \ell_0)\ (\mathsf{borrow}^s\ \ell_1) \Downarrow \mathsf{borrow}^s\ \ell_2 \mid A\,\{\boxed{\mathsf{borrow}^s\ \ell_0},\ \boxed{\mathsf{borrow}^s\ \ell_1},\ \mathsf{loan}^s\ \ell_2\ \sigma\,\}}$$

$$\textsc{Join-MutLoans}\quad \frac{\ell_2 \text{ fresh}}{\Omega_0, \Omega_1 \vdash \mathsf{join}_v\ (\mathsf{loan}^m\ \ell_0)\ (\mathsf{loan}^m\ \ell_1) \Downarrow \mathsf{loan}^m\ \ell_2 \mid A\,\{\,\mathsf{borrow}^m\ \ell_2\ \_,\ \boxed{\mathsf{loan}^m\ \ell_0},\ \boxed{\mathsf{loan}^m\ \ell_1}\}}$$

$$\textsc{Join-SharedLoans}\quad \frac{\text{no mut loan, borrow, } \bot \in v_0,\ v_1 \qquad \ell_2,\ \sigma,\ A \text{ fresh}}{\Omega_0, \Omega_1 \vdash \mathsf{join}_v\ (\mathsf{loan}^s\ \ell_0\ v_0)\ (\mathsf{loan}^s\ \ell_1\ v_1) \Downarrow \mathsf{loan}^s\ \ell_2\ \sigma \mid A\,\{\,\mathsf{borrow}^s\ \ell_2,\ \boxed{\mathsf{loan}^s\ \ell_0\ v_0},\ \boxed{\mathsf{loan}^s\ \ell_1\ v_1}\}}$$

$$\textsc{Join-MutLoan-Other}\quad \frac{\ell' \text{ fresh} \qquad \vdash \mathsf{borrow}^m\ \ell'\ v \prec^{\text{to-abs}} \overrightarrow{A} \qquad \Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ (\mathsf{loan}^m\ \ell)\ (\mathsf{loan}^m\ \ell') \Downarrow v' \mid \overrightarrow{A'}}{\Omega_0, \Omega_1 \vdash \mathsf{join}_v\ (\mathsf{loan}^m\ \ell)\ v \Downarrow v' \mid \overrightarrow{\boxed{A}},\ \overrightarrow{A'}}$$

$$\textsc{Join-Other-MutLoan}\quad \frac{\ell' \text{ fresh} \qquad \vdash \mathsf{borrow}^m\ \ell'\ v \prec^{\text{to-abs}} \overrightarrow{A} \qquad \Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ (\mathsf{loan}^m\ \ell')\ (\mathsf{loan}^m\ \ell) \Downarrow v' \mid \overrightarrow{A'}}{\Omega_0, \Omega_1 \vdash \mathsf{join}_v\ v\ (\mathsf{loan}^m\ \ell) \Downarrow v' \mid \overrightarrow{\boxed{A}},\ \overrightarrow{A'}}$$

$$\textsc{Join-SharedLoan-Other}\quad \frac{\ell' \text{ fresh} \qquad \Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ (\mathsf{loan}^s\ \ell\ v_0)\ (\mathsf{loan}^s\ \ell'\ v_1) \Downarrow v_2 \mid \overrightarrow{A}}{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ (\mathsf{loan}^s\ \ell\ v_0)\ v_1 \Downarrow v_2 \mid \overrightarrow{A}}$$

$$\textsc{Join-Other-SharedLoan}\quad \frac{\ell' \text{ fresh} \qquad \Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ (\mathsf{loan}^s\ \ell'\ v_0)\ (\mathsf{loan}^s\ \ell\ v_1) \Downarrow v_2 \mid \overrightarrow{A}}{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ v_0\ (\mathsf{loan}^s\ \ell\ v_1) \Downarrow v_2 \mid \overrightarrow{A}}$$

$$\textsc{Join-Tuple}\quad \frac{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ v_0\ v_1 \Downarrow v \mid \overrightarrow{A_0} \qquad \Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ w_0\ w_1 \Downarrow w \mid \overrightarrow{A_1}}{\Omega_0, \Omega_1 \vdash \mathsf{join}_v\ (v_0, w_0)\ (v_1, w_1) \Downarrow (v, w) \mid \overrightarrow{A}_0,\ \overrightarrow{A}_1}$$

$$\textsc{Join-Sum}\quad \frac{\Omega_0, \Omega_1 \vdash \mathsf{join}_v\ v_0\ v_1 \Downarrow v \mid \overrightarrow{A} \qquad C = \mathsf{Left} \vee C = \mathsf{Right}}{\Omega_0, \Omega_1 \vdash \mathsf{join}_v\ (C\ v_0)\ (C\ v_1) \Downarrow C\ v \mid \overrightarrow{A}}$$

$$\textsc{Join-Box}\quad \frac{\Omega_0, \Omega_1 \vdash \mathsf{join}_v\ v_0\ v_1 \Downarrow v \mid \overrightarrow{A}}{\Omega_0, \Omega_1 \vdash \mathsf{join}_v\ (\mathsf{Box}\ v_0)\ (\mathsf{Box}\ v_1) \Downarrow \mathsf{Box}\ v \mid \overrightarrow{A}}$$

$$\textsc{Join-Same-MutBorrow}\quad \frac{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ v_0\ v_1 \Downarrow v_2 \mid \overrightarrow{A}}{\Omega_0, \Omega_1 \vdash \mathsf{join}_v\ (\mathsf{borrow}^m\ \ell\ v_0)\ (\mathsf{borrow}^m\ \ell\ v_1) \Downarrow \mathsf{borrow}^m\ \ell\ v_2 \mid \overrightarrow{A}}$$

$$\textsc{Join-Same-SharedLoan}\quad \frac{\text{no } \bot \in v_0,\ v_1 \qquad \Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ v_0\ v_1 \Downarrow v_2 \mid \overrightarrow{A}}{\Omega_0, \Omega_1 \vdash \mathsf{join}_v\ (\mathsf{loan}^s\ \ell\ v_0)\ (\mathsf{loan}^s\ \ell\ v_1) \Downarrow \mathsf{loan}^s\ \ell\ v_2 \mid \overrightarrow{A}}$$

Figure 12.1: Join (Values)

$$\textsc{Join-Same-Abs}\ \frac{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_\Omega\ \Omega_0'\ \Omega_1' \rightsquigarrow \Omega_2}{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_\Omega\ (A,\ \Omega_0')\ (A,\ \Omega_1') \rightsquigarrow A,\ \Omega_2} \qquad \textsc{Join-AbsLeft}\ \frac{A \notin \Omega_1 \qquad \Omega_0,\ \Omega_1 \vdash \mathsf{join}_\Omega\ \Omega_0'\ \Omega_1' \rightsquigarrow \Omega_2}{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_\Omega\ (A,\ \Omega_0')\ \Omega_1' \rightsquigarrow \boxed{A},\ \Omega_2}$$

$$\textsc{Join-Same-Anon}\ \frac{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_\Omega\ \Omega_0'\ \Omega_1' \rightsquigarrow \Omega_2}{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_\Omega\ (\_ \to v,\ \Omega_0')\ (\_ \to v,\ \Omega_1') \rightsquigarrow \_ \to v,\ \Omega_2}$$

$$\textsc{Join-AbsRight}\ \frac{A \notin \Omega_0 \qquad \Omega_0,\ \Omega_1 \vdash \mathsf{join}_\Omega\ \Omega_0'\ \Omega_1' \rightsquigarrow \Omega_2}{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_\Omega\ \Omega_0'\ (A,\ \Omega_1') \rightsquigarrow \boxed{\boxed{A}},\ \Omega_2}$$

$$\textsc{Join-Var}\ \frac{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ v_0\ v_1 \Downarrow v_2 \mid \overrightarrow{A} \qquad \Omega_0,\ \Omega_1 \vdash \mathsf{join}_\Omega\ \Omega_0'\ \Omega_1' \rightsquigarrow \Omega_2}{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_\Omega\ (x \to v_0,\ \Omega_0')\ (x \to v_1,\ \Omega_1') \rightsquigarrow x \to v_2,\ \overrightarrow{A},\ \Omega_2} \qquad \textsc{Join-Empty}\ \frac{}{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_\Omega\ \emptyset\ \emptyset \rightsquigarrow \emptyset}$$

Figure 12.2: Join (Environments)

$$\textsc{Collapse-Merge-Abs}\ \frac{\vdash A_0 \bowtie A_1 = A}{\Omega,\ A_0,\ A_1 \searrow \Omega,\ A} \qquad \textsc{Collapse-Dup-MutBorrow}\ \frac{}{\Omega,\ A \cup \{\boxed{\mathsf{borrow}^m\ \ell\ \_},\ \boxed{\boxed{\mathsf{borrow}^m\ \ell\ \_}}\} \searrow \Omega,\ A \cup \{\mathsf{borrow}^m\ \ell\ \_\}}$$

$$\textsc{Collapse-Dup-MutLoan}\ \frac{}{\Omega,\ A \cup \{\boxed{\mathsf{loan}^m\ \ell},\ \boxed{\boxed{\mathsf{loan}^m\ \ell}}\} \searrow \Omega,\ A \cup \{\mathsf{loan}^m\ \ell\}} \qquad \textsc{Collapse-Dup-SharedBorrow}\ \frac{}{\Omega,\ A \cup \{\boxed{\mathsf{borrow}^m\ \ell},\ \boxed{\boxed{\mathsf{borrow}^m\ \ell}}\} \searrow \Omega,\ A \cup \{\mathsf{borrow}^m\ \ell\}}$$

$$\textsc{Collapse-Dup-SharedLoan}\ \frac{\text{no borrows, loans, } \bot\ \in v_0,\ v_1 \qquad v_2 = \begin{cases} v_0 & \text{if } v_1 = v_0 \\ \sigma & \text{where } \sigma \text{ fresh otherwise} \end{cases}}{\Omega,\ A \cup \{\boxed{\mathsf{loan}^s\ \ell\ v_0},\ \boxed{\boxed{\mathsf{loan}^s\ \ell\ v_1}}\} \searrow \Omega,\ A \cup \{\mathsf{loan}^s\ \ell\ v_2\}}$$

$$\textsc{MergeAbs-Mut-MarkedLeft}\ \frac{\vdash A_0 \bowtie A_1 = A}{\vdash (A_0 \cup \{\boxed{\mathsf{loan}^m\ \ell}\}) \bowtie (A_1 \cup \{\boxed{\mathsf{borrow}^m\ \ell\ \_}\}) = A}$$

$$\textsc{MergeAbs-Shared-MarkedLeft}\ \frac{\vdash (A_0 \cup \{\mathsf{loan}^s\ \ell\ v\}) \bowtie A_1 = A}{\vdash (A_0 \cup \{\boxed{\mathsf{loan}^s\ \ell}\ v\}) \bowtie (A_1 \cup \{\boxed{\mathsf{borrow}^s\ \ell}\}) = A}$$

$$\textsc{MergeAbs-Mut-MarkedRight}\ \frac{\vdash A_0 \bowtie A_1 = A}{\vdash (A_0 \cup \{\boxed{\mathsf{loan}^m\ \ell}\}) \bowtie (A_1 \cup \{\boxed{\mathsf{borrow}^m\ \ell\ \_}\}) = A}$$

$$\textsc{MergeAbs-Shared-MarkedRight}\ \frac{\vdash (A_0 \cup \{\mathsf{loan}^s\ \ell\ v\}) \bowtie A_1 = A}{\vdash (A_0 \cup \{\boxed{\mathsf{loan}^s\ \ell}\ v\}) \bowtie (A_1 \cup \{\boxed{\mathsf{borrow}^s\ \ell}\}) = A}$$

Figure 12.3: Collapse

12.1.1 Joining Values

Joins are computed pointwise: if a variable x is present in both environments, we perform a join on its associated value (Join-Var). Value joins are formally defined using the judgement $\Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ v_0\ v_1 \Downarrow v \mid \overrightarrow{A}$. This judgment is similar to the environment join above: it states that merging values v_0 and v_1 yields a new value v, and creates a set of region abstractions $\overrightarrow{A}$ to be added to the current environment. When values are the same, for instance x and y in our example above, the join is the identity (Join-Same). When values differ but do not contain borrows, loans, or $\bot$, e.g., variable b, a fresh symbolic value is returned (Join-Symbolic).

The more interesting cases occur when borrows or loans are involved. In the example above, let us look at the variable p, which corresponds to a mutable borrow in both branches, although associated to loan identifiers ℓ_0, ℓ_1 and values $0, 1$ respectively. A naive way to join these borrows would be to create a new borrow associated to a fresh loan identifier ℓ_2 and a fresh symbolic value σ, and to constrain ℓ_2 to end whenever trying to end either ℓ_0 or ℓ_1; this can be done by creating a fresh region abstraction containing $\mathsf{borrow}^m\ \ell_0\ 0$, $\mathsf{borrow}^m\ \ell_1\ 1$, and $\mathsf{loan}^m\ \ell_2$. This attempt unfortunately does not work in practice, as it might lead to invalid states containing duplicated mutable borrows: coming back to our example, the left environment still contains $\mathsf{borrow}^m\ \ell_1\ 1$ associated to y. To fix this issue, we additionally keep track of the origin of values (Join-MutBorrows). For presentation purposes, we will denote a value v coming from the left (resp. right) environment as $\boxed{v}$ (resp. $\underline{\bar{v}}$). We follow a similar approach to join a value with $\bot$, e.g., for variables px and py (Join-Bottom-Other, Join-Other-Bottom), ultimately leading to the joined environment below.

$$
\begin{array}{l}
x \mapsto \mathsf{loan}^m\ \ell_0, \qquad y \mapsto \mathsf{loan}^m\ \ell_1, \\
px \mapsto \bot, \qquad A_0\ \{\underline{\mathsf{borrow}^m\ \ell_0\ \_}\}, \qquad py \mapsto \bot, \qquad A_1\ \{\boxed{\mathsf{borrow}^m\ \ell_1\ \_}\}, \\
p \mapsto \mathsf{borrow}^m\ \ell_2\ \sigma, \qquad A_2\ \{\boxed{\mathsf{borrow}^m\ \ell_0\ \_}, \underline{\mathsf{borrow}^m\ \ell_1\ \_}, \mathsf{loan}^m\ \ell_2\}
\end{array}
$$

12.1.2 Collapsing Environments

While keeping track of the origin of values avoids inconsistencies due to duplicated borrows, it gives rise to new environments that do not belong to LLBC$^\#$. Instead of extending the semantics, we propose a set of local transformations that gradually turns an environment with marked values (e.g., $\boxed{v}$) back into an LLBC$^\#$ state. We dub their transitive closure the *collapse* operator, which we denote $\searrow$, and show its rules in Figure 12.3.

Coming back to our running example, we aim to collapse the joined environment to remove all markers. To do so, we first apply Collapse-Merge-Abs twice to merge

abstractions A_0, A_1, and A_2, using the merge rules seen in Section 11.3.1. We finally use Collapse-Dup-MutBorrow twice to simplify $\boxed{\mathsf{borrow}^m\ \ell_0\ \_}$ and $\boxed{\mathsf{borrow}^m\ \ell_0\ \_}$ into $\mathsf{borrow}^m\ \ell_0\ \_$, then $\boxed{\mathsf{borrow}^m\ \ell_1\ \_}$ and $\boxed{\mathsf{borrow}^m\ \ell_1\ \_}$ into $\mathsf{borrow}^m\ \ell_1\ \_$ to obtain the following LLBC$^\#$ environment.

$$x \mapsto \mathsf{loan}^m\ \ell_0, \quad y \mapsto \mathsf{loan}^m\ \ell_1, \quad px \mapsto \bot, \quad py \mapsto \bot,$$
$$p \mapsto \mathsf{borrow}^m\ \ell_2\ \sigma, \quad A_3\ \{\mathsf{borrow}^m\ \ell_0\ \_, \mathsf{borrow}^m\ \ell_1\ \_, \mathsf{loan}^m\ \ell_2\}$$

12.1.3 Soundness

We now set our sights on proving the soundness of the join and collapse operators. Formally, we aim to prove the following theorem, which states that for any LLBC$^\#$ states Ω_l, Ω_r, if the composition of join and collapse yields an LLBC$^\#$ state Ω_c, that is, a state with no marked values, then this state is related to both Ω_l and Ω_r. We can then resume the evaluation with the joined state, instead of the set of states resulting from the two branches.

Theorem 9 (Join-Collapse-Le)**.** *For all Ω_l, Ω_r, Ω_j, Ω_c we have:*

$$\Omega_l,\ \Omega_r \vdash \mathsf{join}_\Omega\ \Omega_l\ \Omega_r \rightsquigarrow \Omega_j \Rightarrow$$
$$\vdash \Omega_j \searrow \Omega_c \Rightarrow$$
$$\textit{no marked value in } \Omega_c \Rightarrow$$
$$\forall\ m \in \{l, r\},\ \Omega_m \leq \Omega_c$$

The proof is in Appendix G and relies on an induction on the reductions for join and collapse. To explain the intuition behind the proof, we will consider state projections keeping marked values from only one side. For presentation purposes, we will focus on the environment on the left. The state projection is then defined as discarding all values from the right side (e.g., $\lfloor \bar{v} \rfloor$) and removing the left markers (e.g., replacing $\boxed{v}$ by v). Then, the intuition is that the left environment will always be in relation with the left projection of the join. Using this notion of projection, there is almost a one-to-one mapping between the rules defining join and collapse on one side, and the rules defining the $\leq$ relation on the other. For instance, applying the left projection to the conclusion of Join-MutBorrows yields exactly Le-Reborrow-MutBorrow-Abs.

One important point of this soundness theorem is that it requires that the result of collapse does not contain any marked value; in our implementation of these rules, we rely on several heuristics to find a derivation satisfying this condition, and raise an error when unsuccessful. As we will see in Chapter 14, while incomplete, these heuristics are

sufficient to cover a large subset of Rust.

12.2 Extending Support to Loops

The join and collapse operator we presented allows us to handle disjunctive control flow without leading to an explosion in the number of states to consider. While the presentation focused on simple branching, i.e., merging two environments after an if-then-else statement, this approach also applies to more complex constructs, such as loops.

12.2.1 A Toy Example

As a first example, consider the toy program below which iteratively increments variable x through its mutable borrow p. While this program is purposedly simple, its reborrow inside a loop is a characteristic pattern when iterating over recursive data structures in Rust; we provide a more realistic example in the next section (Section 12.2.2).

```
x  = 0; p = &mut x; // x ↦ loan^m ℓ0, p ↦ borrow^m ℓ0 0
loop {
  p = &mut (*p); // x ↦ loan^m ℓ0, _ ↦ borrow^m ℓ0 (loan^m ℓ1), p ↦ borrow^m ℓ1 0
  *p += 1; // x ↦ loan^m ℓ0, _ ↦ borrow^m ℓ0 (loan^m ℓ1), p ↦ borrow^m ℓ1 1
  continue; }
```

To borrow-check this loop, our goal is to derive a state general enough to encompass all possible states upon entering the loop; this is known as computing a fixed-point in program analysis. Unfortunately, our example shows that using state inclusion to determine if a given state is a fixed-point is not sufficient. Starting from a hypothetical fixed-point, executing the loop body will create a new loan identifier and an anonymous mapping during the reborrow, which after joining with the initial environment will yield new region abstractions. Our main observation is that it is actually sufficient to consider a fixed-point up to loan and region identifier substitution. The intuition is that, seeing LLBC$^\#$ states as shape graphs [411, 412], the substitutions correspond to graph isomorphisms, which preserve the semantics of a program.

Additionally, compared to arbitrary joins, loops follow a generic pattern. The loop body possibly introduces fresh borrows and anonymous mappings, before being merged with an earlier snapshot. To compute a fixed-point according to the LLBC$^\#$ semantics, we therefore rely on several heuristics. First, we convert all fresh anonymous mappings to region abstractions using Le-ToAbs. Second, we flatten the environment by merging freshly introduced region abstractions that contain related borrows and loans, that

is, values associated to the same loan identifier. We finally apply the join operator presented in the previous section to the resulting state and the initial environment, and repeat this approach until we find a fixed-point. In our example, we get the environment below, which remains the same after executing the loop body, up to substitution of ℓ_2 and A_2 by fresh identifiers.

$$x \mapsto \mathsf{loan}^m\, \ell_0,\ A_2\,\{\,\mathsf{borrow}^m\, \ell_0\ \_,\ \mathsf{loan}^m\, \ell_2\,\},\ p \mapsto \mathsf{borrow}^m\, \ell_2\ \sigma$$

Note that, while conceptually similar to widening operators in abstract interpretation, we do not claim that our approach terminates. Our implementation of LLBC$^\#$ fails if the fixed-point computation does not converge after a fixed number of steps. In practice, we however observe that these heuristics are sufficient to handle a wide range of examples, as we demonstrate in the next section; in those examples the computation actually converges in one step.

12.2.2 An Example with a Recursive Data Structure

We now describe a more realistic example which uses a recursive data-structure. In this example, we recursively dive into a list so as to get a mutable borrow to its last element. We could then use this borrow to append another list, for instance. We define the type list as: $\mathsf{List}\,\tau := \mu X.\,() + (\tau,\ \mathsf{Box}\,X)$. We also pose $\mathsf{Nil} := \mathsf{Left}\,()$ and $\mathsf{Cons}\,x\,t := \mathsf{Right}\,(x,\ \mathsf{Box}\,t)$ as syntactic shortcuts. We start in an environment where `l0` maps to a symbolic value $\sigma_0 : \mathsf{List}\,\tau$.

```
// l0 ↦ (σ0 : List τ)
l = &mut l0;
// l0 ↦ loan^m ℓ0,
// l ↦ borrow^m ℓ0 σ0
loop {
  match *l {
    Nil => {
      // l0 ↦ loan^m ℓ0,
      // l ↦ borrow^m ℓ0 Nil
      break; // No need to compute a join here
    }
    Cons => {
      // l0 ↦ loan^m ℓ0,
      // l ↦ borrow^m ℓ0 (Cons σ1 σ2)
      l = &mut (*(*l as Cons).1);
      // l0 ↦ loan^m ℓ0,
      // _ ↦ borrow^m ℓ0 (Cons σ1 loan^m ℓ1),
      // l ↦ borrow^m ℓ1 σ2
```

```
        continue; // We join with the environment at the entry of the loop
    } } }
```

At the continue, we convert the anonymous value to a region abstraction (with Le-ToAbs, ToAbs-MutBorrow, ToAbs-Sum, ToAbs-Box, ToAbs-MutLoan). We join the environment at the entry of the loop with the environment at the continue. There is nothing to do for l_0 because its value is unchanged (we use . Join-Same). For A_0 we use Join-AbsRight. Finally, for l we use Join-MutBorrows. We get:

$$
\begin{aligned}
&l_0 \mapsto \mathsf{loan}^m\ \ell_0,\\
&A_0\ \{\boxed{\mathsf{borrow}^m\ \ell_0\ \_}^{\,\cdots},\ \boxed{\mathsf{loan}^m\ \ell_1}^{\,\cdots}\},\\
&l \mapsto \mathsf{borrow}^m\ \ell_2\ \sigma_3\\
&A_1\ \{\boxed{\mathsf{borrow}^m\ \ell_0\ \_},\ \boxed{\mathsf{borrow}^m\ \ell_1\ \_}^{\,\cdots},\ \mathsf{loan}^m\ \ell_2\ \}
\end{aligned}
$$

We collapse the environment by merging A_0 and A_1. By MergeAbs-Mut-MarkedLeft we simplify $\boxed{\mathsf{loan}^m\ \ell_1}^{\,\cdots}$ and $\boxed{\mathsf{borrow}^m\ \ell_1\ \_}^{\,\cdots}$. We then use Collapse-Dup-MutBorrow to simplify $\boxed{\mathsf{borrow}^m\ \ell_0\ \_}^{\,\cdots}$ and $\boxed{\mathsf{borrow}^m\ \ell_0\ \_}$ into $\mathsf{borrow}^m\ \ell_0\ \_$. We get the following environment, which is a fixed-point.

$$
\begin{aligned}
&l_0 \mapsto \mathsf{loan}^m\ \ell_0,\\
&A_2\ \{\ \mathsf{borrow}^m\ \ell_0\ \_,\ \mathsf{loan}^m\ \ell_2\ \}\\
&l \mapsto \mathsf{borrow}^m\ \ell_2\ \sigma_3
\end{aligned}
$$

Chapter 13

From Symbolic Semantics to Functional Code

At last, we explain how Aeneas, using the symbolic semantics, generates a pure translation of the original LLBC program. We first show examples, then present the rules.

13.1 Translation Example: call_choose

As a starting example, we consider the translation of the call_choose function below, now presented in LLBC syntax with explicit writes and moves, along with a fully-explicit return variable x_ret.

```
1 fn call_choose(mut p : (u32, u32)) -> u32 {
2   let px = &mut p.0;
3   let py = &mut p.1;
4   let pz = choose(true, move px, move py);
5   *pz = *pz + 1;
6   x_ret = move p.0;
7   return;
8 }
```

The translation of the forward function is carried out by performing a symbolic execution on call_choose, and synthesizing an AST in parallel to reflect the effect of applying the symbolic execution rules.

A key insight about our synthesis rules is that they are exclusively concerned with *symbolic values*; the actual variables from the source program ($x \mapsto \ldots$) are mere book-keeping devices and have no relevance to the translated program. Symbolic values

σ, however, cannot be determined statically; they thus compute at run-time, and as such are let-bound in the target program.

We start the translation by initializing an environment (Synth), where p maps to a symbolic value σ_0; as the function does not contain regions, we do not introduce region abstractions. In parallel, we synthesize the function as an AST with a hole to be progressively filled as we make progress throughout the translation. We present both environment and translation side-by-side to reflect the fact that they both make progress in parallel. We point, whenever possible, to the specific rules that apply; these pointers can be skipped upon a first reading, but might prove useful later, once the reader has encountered the formal definition of the translation.

$p \mapsto (\sigma_0 : (\mathsf{u32}, \mathsf{u32}))$

```
def call_choose (s0 : (U32 × U32)) :
  Result U32 := do
  [.]
```

At line 2, accessing field 0 requires us to expand σ_0; we first do so, which leads to the introduction of a let-binding in the translation (Synth-Reorg-SymbolicPair).

$p \mapsto (\sigma_1 : \mathsf{u32}, \sigma_2 : \mathsf{u32})$

```
def call_choose (s0 : (U32 × U32)) :
  Result U32 := do
  let (s1, s2) := s0
  [.]
```

The expansion above allows us to evaluate the two mutable borrows on lines 2-3 via E-MutBorrow. Importantly, borrows and assignments just lead to bookkeeping in the environment: the synthesized translation is left unchanged. We get the following:

$p \mapsto (\mathsf{loan}^m\ \ell_1, \mathsf{loan}^m\ \ell_2)$
$px \mapsto \mathsf{borrow}^m\ \ell_1\ (\sigma_1 : \mathsf{u32})$
$py \mapsto \mathsf{borrow}^m\ \ell_2\ (\sigma_2 : \mathsf{u32})$

```
def call_choose (s0 : (U32 × U32)) :
  Result U32 := do
  let (s1, s2) := s0
  [.]
```

We then reach the function call at line 4. To account for the call, we do several things (Synth-E-Call). As before (Section 10.1), we introduce a region abstraction to account for α, transfer ownership of the effective arguments to the abstraction, then introduce a fresh mutably borrowed value $\mathsf{borrow}^m\ \ell_3\ (\sigma_3 : u32)$ to account for the returned value stored in pz. We also attach a *continuation* to the region abstraction $A(\alpha)$ (inside the $[\![...]\!]$, in the environment) that we will use in the translation upon ending $A(\alpha)$. This continuation is simply the backward function of choose; we use a special notation to indicate that it consumes the value retrieved upon ending ℓ_3 (i.e., $\lambda\, \ell_3 \Rightarrow ...$) and produces the values to give back to ℓ_1 and ℓ_2 (i.e., $(\ell_1, \ell_2) : ...$).

In parallel, we introduce a call to choose in the synthesized translation. As previously mentioned, the borrow types are translated to the identity; the input arguments of

choose are thus simply s1 and s2, while choose simply outputs s3, together with the backward function back. Our translation is monadic, meaning we obtain:

$p \mapsto (\mathsf{loan}^m\ \ell_1, \mathsf{loan}^m\ \ell_2)$
$px \mapsto \bot$
$py \mapsto \bot$
$A(\alpha)\{$
$\quad \_,$
$\quad \mathsf{borrow}^m\ \ell_1\ (\sigma_1 : \mathsf{u32}),$
$\quad \mathsf{borrow}^m\ \ell_2\ (\sigma_2 : \mathsf{u32}),$
$\quad \mathsf{loan}^m\ \ell_3,$
$\}[\![\ (\ell_1, \ell_2) :\ \lambda\, \ell_3 \Rightarrow \mathsf{back}\ \ell_3\ ]\!]$
$pz \mapsto \mathsf{borrow}^m\ \ell_3\ (\sigma_3 : \mathsf{u32})$

```
def call_choose (s0 : (U32 × U32)) :
  Result U32 := do
  let (s1, s2) := s0
  let (s3, back) ← choose true s1 s2
  [.]
```

We can now symbolically execute the increment, which merely introduces an addition and generates a fresh variable σ_4:

$p \mapsto (\mathsf{loan}^m\ \ell_1, \mathsf{loan}^m\ \ell_2)$
$px \mapsto \bot$
$py \mapsto \bot$
$A(\alpha)\{$
$\quad \_,$
$\quad \mathsf{borrow}^m\ \ell_1\ (\sigma_1 : \mathsf{u32}),$
$\quad \mathsf{borrow}^m\ \ell_2\ (\sigma_2 : \mathsf{u32}),$
$\quad \mathsf{loan}^m\ \ell_3,$
$\}[\![\ (\ell_1, \ell_2) :\ \lambda\, \ell_3 \Rightarrow \mathsf{back}\ \ell_3\ ]\!]$
$pz \mapsto \mathsf{borrow}^m\ \ell_3\ (\sigma_4 : \mathsf{u32})$

```
def call_choose (s0 : (U32 × U32)) :
  Result U32 := do
  let (s1, s2) = s0
  let (s3, back) ← choose true s1 s2
  let s4 ← s3 + 1
  [.]
```

Finally, the move at line 6 requires retrieving the ownership of p.0. Doing so requires ending the region abstraction $A(\alpha)$ (introduced by the call to choose), which in turns requires ending the loan inside the abstraction (Synth-Reorg-End-MutBorrow-Abs). Accordingly, we first end ℓ_3 which leads to the environment below. Importantly, we update the continuation associated with $A(\alpha)$ to account for the fact that it consumes the value given back upon ending ℓ_3. Ending a loan (or a borrow, depending of the point of view) leaves the synthesized code unchanged.

$p \mapsto (\mathsf{loan}^m\ \ell_1, \mathsf{loan}^m\ \ell_2)$
$px \mapsto \bot$
$py \mapsto \bot$
$A(\alpha)\{$
  $\_,$
  $\mathsf{borrow}^m\ \ell_1\ (\sigma_1 : \mathsf{u32}),$
  $\mathsf{borrow}^m\ \ell_2\ (\sigma_2 : \mathsf{u32}),$
  $(\sigma_4 : \mathsf{u32})$
$\}[\![\ (\ell_1,\ \ell_2) :\ \mathsf{back}\ \sigma_4\ ]\!]$
$pz \mapsto \bot$

```
def call_choose (s0 : (U32 × U32)) :
  Result U32 := do
  let (s1, s2) := s0 in
  let (s3, back) ← choose true s1 s2
  let s4 := s3
  let s5 ← s4 + 1
  [.]
```

Then, we actually end the region abstraction $A(\alpha)$ by moving back the borrows ℓ_1 and ℓ_2 in the environment, with fresh symbolic values σ_5 and σ_6 (Synth-Reorg-End-Abstraction). Those are the values given back *by* choose. On the side of the translated code, we materialize the end of $A(\alpha)$ by introducing a call to its continuation. As noted above, this continuation consumes the value given back to the loan ℓ_3 upon ending it (that is, σ_4); it also outputs the values given back to ℓ_1 and ℓ_2 (that is, the pair: $(\sigma_5,\ \sigma_6)$). We get:

$p \mapsto (\mathsf{loan}^m\ \ell_1, \mathsf{loan}^m\ \ell_2)$
$px \mapsto \bot$
$py \mapsto \bot$
$\_ \mapsto \mathsf{borrow}^m\ \ell_1\ (\sigma_5 : \mathsf{u32})$
$\_ \mapsto \mathsf{borrow}^m\ \ell_2\ (\sigma_6 : \mathsf{u32})$
$pz \mapsto \bot$

```
def call_choose (s0 : (U32 × U32)) :
  Result U32 := do
  let (s1, s2) := s0
  let (s3, back) ← choose true s1 s2
  let s4 ← s3 + 1
  let (s5, s6) ← back s4
  [.]
```

We can finally end the borrow ℓ_1 and evaluate the return, which ends the translation (Synth-E-Return). As we save meta-information about the assignments to generate suitable names for the variables, Aeneas actually generates the following function:

```
def call_choose (p : (U32 × U32)) : Result U32 := do
  let (px, py) := p
  let (pz, back) ← choose true px py
  let pz0 ← pz + 1
  let (px0, _) ← back pz0
  ok px0
```

13.2 Translation Example: choose

We now proceed with the synthesis of the choose, which requires synthesizing a backward function. We recall its definition here for the sake of clarity; like in the previous example,

we make all the statements explicit.

```
1  fn choose<'a, T>(b : bool, x : &'a mut T, y : &'a mut T) -> &'a mut T {
2    if b {
3      x_ret = move x;
4      return;
5    }
6    else {
7      x_ret = move y;
8      return;
9    }
10 }
```

Unlike call_choose, the choose function takes borrows as input parameters. We thus need to track their provenance, from the point of view of the callee. As a consequence, we initialize the environment by introducing an abstraction containing loans so as to model the values owned by the caller and *loaned* to the function for as long as α lives (Synth). This is the dual of the caller's point of view: the abstraction contains two mutable borrows $\ell_x^{(0)}$ and $\ell_y^{(0)}$, whose loans are not in the environment; they stand for the values we consumed upon calling the function. The abstraction also contains two mutable loans ℓ_x and ℓ_y, associated to the borrows of the input values x and y. Importantly, we attach a continuation that we will use for the synthesis, and which is currently the identity; intuitively, this means that $\ell_x^{(0)}$ and ℓ_x are actually the same (and similarly for $\ell_y^{(0)}$ and ℓ_y). The link between borrows and loans inside the input region abstraction will become more complex as we proceed through the synthesis.

$A_{\text{input}}(\alpha)\{$
  $\text{borrow}^m\ \ell_x^{(0)}\ \_,$
  $\text{borrow}^m\ \ell_y^{(0)}\ \_,$
  $\text{loan}^m\ \ell_x,$
  $\text{loan}^m\ \ell_y,$
$\}[\![\ (\ell_x^{(0)}, \ell_y^{(0)}) :\ \lambda\ \ell_x\ \ell_y \Rightarrow (\ell_x, \ell_y)\ ]\!],$

$b \mapsto \sigma_b$
$x \mapsto \text{borrow}^m\ \ell_x\ (\sigma_x : \text{T})$
$y \mapsto \text{borrow}^m\ \ell_y\ (\sigma_y : \text{T})$

```
def choose
  {T : Type} (b : Bool) (x y : T) :
  Result (T × (T → (T × T))) := do
  [.]
```

We then evaluate the if at line 2 (Synth-E-IfThenElse-NoJoin). This requires branching over the symbolic value σ_b. We duplicate the environment and substitute σ_b with true for the first branch, and false for the second branch. Of course, this introduces a branching in the synthesized translation. Below, we show the environment for the first branch of the `if`:

$A_{\mathsf{input}}(\alpha)\{$
$\quad \mathsf{borrow}^m\ \ell_x^{(0)}\ \_,$
$\quad \mathsf{borrow}^m\ \ell_y^{(0)}\ \_,$
$\quad \mathsf{loan}^m\ \ell_x,$
$\quad \mathsf{loan}^m\ \ell_y,$
$\}[\![\ (\ell_x^{(0)}, \ell_y^{(0)}) :\ \lambda\ \ell_x\ \ell_y \Rightarrow (\ell_x, \ell_y)\ ]\!],$

$b \mapsto \mathsf{true}$
$x \mapsto \mathsf{borrow}^m\ \ell_x\ (\sigma_x : \mathsf{T})$
$y \mapsto \mathsf{borrow}^m\ \ell_y\ (\sigma_y : \mathsf{T})$

```
def choose
  {T : Type} (b : Bool) (x y : T) :
  Result (T × (T → (T × T)) := do
  if b then [.]
  else [.]
```

We proceed with the evaluation of the first branch. Upon reaching line 3, we move x to the special return variable; the synthesized code is left unchanged.

$A_{\mathsf{input}}(\alpha)\{$
$\quad \mathsf{borrow}^m\ \ell_x^{(0)}\ \_,$
$\quad \mathsf{borrow}^m\ \ell_y^{(0)}\ \_,$
$\quad \mathsf{loan}^m\ \ell_x,$
$\quad \mathsf{loan}^m\ \ell_y,$
$\}[\![\ (\ell_x^{(0)}, \ell_y^{(0)}) :\ \lambda\ \ell_x\ \ell_y \Rightarrow (\ell_x, \ell_y)\ ]\!],$

$b \mapsto \mathsf{true}$
$x \mapsto \bot$
$y \mapsto \mathsf{borrow}^m\ \ell_y\ (\sigma_y : \mathsf{T})$
$x_{\mathsf{ret}} \mapsto \mathsf{borrow}^m\ \ell_x\ (\sigma_x : \mathsf{T})$

We then reach the return on line 4. This is the crucial part of the translation. We need to transform the environment, by using reorganization rules then the $\leq$ relation, so that it matches a target environment given by the signature of choose; this target environment is the environment used by Synth-E-Call to model a call to choose. In the process, we will introduce and merge region abstractions, thus progressively synthesizing the backward function for the first branch of the `if`.

We first use Synth-Le-MoveValue to move the values of b and y to anonymous values, and use Synth-Le-AnonValue to eliminate the value of b as it doesn't contain any borrows; this has no effect on the synthesis. We then apply Synth-Le-ToAbs to transform the borrow moved from y into a region abstraction A_0. This region abstraction has no inputs, as it doesn't contain mutable loans, and a single output, the value to give back for the mutable borrow ℓ_y; we thus attach a continuation stating that upon ending A_0 we retrieve a value for ℓ_y, which is actually σ_y.

$A_{\mathsf{input}}(\alpha)\{$
  $\mathsf{borrow}^m\ \ell_x^{(0)}\ \_,$
  $\mathsf{borrow}^m\ \ell_y^{(0)}\ \_,$
  $\mathsf{loan}^m\ \ell_x,$
  $\mathsf{loan}^m\ \ell_y,$
$\}[\![\ (\ell_x^{(0)}, \ell_y^{(0)}) :\ \lambda\ \ell_x\ \ell_y \Rightarrow (\ell_x, \ell_y)\ ]\!],$

$b \mapsto \bot$
$x \mapsto \bot$
$y \mapsto \bot$
$x_{\mathsf{ret}} \mapsto \mathsf{borrow}^m\ \ell_x\ (\sigma_x : \mathsf{T})$

$A_0\{\quad \mathsf{borrow}^m\ \ell_y\ \_\quad \}[\![\ \ell_y :\ \sigma_y\ ]\!]$

We perform one last step by merging $A_{\mathsf{input}}(\alpha)$ and A_0 together (Synth-Le-MergeAbs). Merging the content of region abstractions is done as before; in particular the borrow and the loan for ℓ_y cancel out. Merging two region abstractions also requires composing their continuations. Importantly, the fact that a borrow and a loan cancel out is mirrored by the fact that the continuation of $A_{\mathsf{input}}(\alpha)$ consumes a borrow (ℓ_y) which is actually an output of A_0. The resulting continuation of $A_{\mathsf{input}}(\alpha)$ thus only has one input, the value consumed upon ending ℓ_x.

This yields an environment which matches our target environment: x_{ret} maps to a valid borrow, all the other local variables map to $\bot$, and we have a single region abstraction for α which acts as an interface between the caller and the callee. We can thus end the translation of the first branch by using Synth-E-Return; the returned value is the value inside x_{ret} (i.e., σ_x) while the backward function is given by the continuation attached to $A_{\mathsf{input}}(\alpha)$.

$A_{\mathsf{input}}(\alpha)\{$
  $\mathsf{borrow}^m\ \ell_x^{(0)}\ \_,$
  $\mathsf{borrow}^m\ \ell_y^{(0)}\ \_,$
  $\mathsf{loan}^m\ \ell_x,$
$\}[\![\ (\ell_x^{(0)}, \ell_y^{(0)}) :\ \lambda\ \ell_x \Rightarrow (\ell_x, \sigma_y)\ ]\!],$

$b \mapsto \bot$
$x \mapsto \bot$
$y \mapsto \bot$
$x_{\mathsf{ret}} \mapsto \mathsf{borrow}^m\ \ell_x\ (\sigma_x : \mathsf{T})$

```
def choose
  {T : Type} (b : Bool) (x y : T) :
  Result (T × (T → (T × T))) := do
  if b then ok (x, fun x0 => (x0, y))
  else [.]
```

We omit the translation of the second branch, which is similar.

13.3 Translation Example: Join

We now look at a translation example which involves a join operation. The `if` in the function below is choose inlined; we will see that the translation also leads to the translation of choose being inlined.

```
1  fn inline_choose(b : bool, mut x : u32, mut y : u32) -> u32 {
2    let px = &mut x;
3    let py = &mut y;
4    let pz;
5    if b {
6      pz = move px;
7    } else {
8      pz = move py;
9    }
10   *pz = *pz + 1;
11   x_ret = move x;
12   return;
13 }
```

We directly jump to the interesting part, that is what happens when we join the environment resulting from the evaluation of the different branches (lines 6 and 8). Below, we show the environment at the end of the left branch, the environment at the end of the right branch, and the result of the synthesis.

$b \mapsto \mathsf{true}$
$x \mapsto \mathsf{loan}^m\ \ell_x$
$y \mapsto \mathsf{loan}^m\ \ell_y$
$px \mapsto \bot$
$py \mapsto \mathsf{borrow}^m\ \ell_y\ \sigma_y$
$pz \mapsto \mathsf{borrow}^m\ \ell_x\ \sigma_x$

$b \mapsto \mathsf{false}$
$x \mapsto \mathsf{loan}^m\ \ell_x$
$y \mapsto \mathsf{loan}^m\ \ell_y$
$px \mapsto \mathsf{borrow}^m\ \ell_x\ \sigma_x$
$py \mapsto \bot$
$pz \mapsto \mathsf{borrow}^m\ \ell_y\ \sigma_y$

```
def inline_choose
  {T : Type} (b : Bool) (x y : T) :
  Result T := do
  let [.] ←
    if b then [.]
    else [.]
```

We join the two environments, resulting in the following state *before* the collapse. The join introduces two fresh symbolic values: σ_b when joining the values of b (true and false) and σ_z when joining the values of pz (more specifically, σ_x and σ_y). In the synthesized translation, those symbolic values are output by the `if then else` expression; importantly, we leave some holes for the continuation(s) that we will have to (potentially) synthesize for the region abstractions introduced by the join. We also note that the value b0 synthesized in the translation is not useful: it is an artifact resulting from the fact that, upon evaluating the branching, we substituted the original (symbolic) value of b with true and false in the first and second branch, respectively, and that upon exiting the branches we joined those values back into a fresh symbolic value.

We could update the rules to get rid of this artifact; instead we prefer to keep the rules simple and implement a micro-pass to detect and eliminate this pattern *afterwards*.

$b \mapsto \sigma_b$
$x \mapsto \mathsf{loan}^m\ \ell_x$
$y \mapsto \mathsf{loan}^m\ \ell_y$
$px \mapsto \bot$
$A_0\{\ \boxed{\mathsf{borrow}^m\ \ell_x\ \_}\ \}[\![\ \boxed{\ell_x} : \boxed{\sigma_x}\ ]\!]$
$py \mapsto \bot$
$A_1\{\ \boxed{\mathsf{borrow}^m\ \ell_y\ \_}\ \}[\![\ \boxed{\ell_y} : \boxed{\sigma_y}\ ]\!]$
$pz \mapsto \mathsf{borrow}^m\ \ell_z\ \sigma_z$
$A_2\{$
$\quad \boxed{\mathsf{borrow}^m\ \ell_x\ \_},$
$\quad \boxed{\mathsf{borrow}^m\ \ell_y\ \_},$
$\quad \mathsf{loan}^m\ \ell_z$
$\}[\![\ \lambda\ \ell_z \Rightarrow (\boxed{\ell_x}, \boxed{\ell_y}) : (\boxed{\ell_z}, \boxed{\ell_z})\ ]\!]$

```
def inline_choose
  {T : Type} (b : Bool) (x y : T) :
  Result T := do
  let (b0, z, [.]) ←
    if b then ok (true, x, [.])
    else (false, y, [.])
```

Similarly to the example in Section 12.1, we also introduce three region abstractions: A_0 results from the join of the values of px ($\bot$ in the first branch and $\mathsf{borrow}^m\ \ell_x\ \sigma_x$ in the second branch); A_1 from the join of py (either $\mathsf{borrow}^m\ \ell_y\ \sigma_y$ or $\bot$); A_2 from the join of pz (either $\mathsf{borrow}^m\ \ell_x\ \sigma_x$ or $\mathsf{borrow}^m\ \ell_y\ \sigma_y$). Those abstractions are also annotated with continuations which this time contain markers; those markers indicate the fact that the continuations may consume or output values only if we took a specific branch. For instance, the continuation for A_2 is: $\lambda\ \ell_z \Rightarrow (\boxed{\ell_x}, \boxed{\ell_y}) : (\boxed{\ell_z}, \boxed{\ell_z})$. It consumes the value given back upon ending ℓ_z and outputs: either the value to give back to ℓ_x, in case we took the first branch, or the value to give back to ℓ_y, in case we took the second branch. It may be clearer to "project" this continuation, similarly to what we did in Section 12.1, to only keep the left or right part. If we project it to the left (i.e., if we took the first branch), we get: $\lambda\ \ell_z \Rightarrow \ell_x : \ell_z$ (this region abstraction consumes a value for ℓ_z and outputs the value to give back to ℓ_x). Similarly, if we project to the right (i.e., if we took the second branch) we get: $\lambda\ \ell_z \Rightarrow \ell_y : \ell_z$ (A_2 consumes a value for ℓ_z and outputs the value to give back to ℓ_y).

We now collapse the environment to remove the markers; this has the effect of merging the region abstractions A_0, A_1 and A_2 into a single one. The important part is related to the composition of the continuations. For instance, both A_0 and A_2 output some values for ℓ_x, however A_0 outputs a value only in the case we took the right branch, while A_2 does it only if we took the left branch. As a consequence, the resulting region

abstraction A_3 outputs a value for ℓ_x in *all cases*, but this value depends on whether we took the left or right branch. More precisely, this value must be the value consumed upon ending ℓ_z if we took the left branch (because of A_2), and σ_x otherwise (because of A_0); we note this as: $\ell_z \oplus \sigma_x$. We show the resulting environment below.

$b \mapsto \sigma_b$
$x \mapsto \mathsf{loan}^m\ \ell_x$
$y \mapsto \mathsf{loan}^m\ \ell_y$
$px \mapsto \bot$
$py \mapsto \bot$
$pz \mapsto \mathsf{borrow}^m\ \ell_z\ \sigma_z$
$A_4\{$
  $\mathsf{borrow}^m\ \ell_x\ \_,$
  $\mathsf{borrow}^m\ \ell_y\ \_,$
  $\mathsf{loan}^m\ \ell_z$
$\}[\![\ \lambda\ \ell_z \Rightarrow (\ell_x,\ \ell_y) :\ (\ell_z \oplus \sigma_x,\ \sigma_y \oplus \ell_z)\ ]\!]$

```
def inline_choose
  {T : Type} (b : Bool) (x y : T) :
  Result T := do
  let (b, z, [.]) ←
    if b then ok (true, x, [.])
    else (false, y, [.])
```

We are now ready to finish the synthesis of the branching: we introduce one continuation per region abstraction (here, A_4), and synthesize those continuations simply by "projecting" them. We also replace the continuation in A_4 so that it refers to a fresh name (back), which is the name used in the synthesis; this way, upon ending A_4 later, we simply insert a call to back

$b \mapsto \sigma_b$
$x \mapsto \mathsf{loan}^m\ \ell_x$
$y \mapsto \mathsf{loan}^m\ \ell_y$
$px \mapsto \bot$
$py \mapsto \bot$
$pz \mapsto \mathsf{borrow}^m\ \ell_z\ \sigma_z$
$A_4\{$
  $\mathsf{borrow}^m\ \ell_x\ \_,$
  $\mathsf{borrow}^m\ \ell_y\ \_,$
  $\mathsf{loan}^m\ \ell_z$
$\}[\![\ \lambda\ \ell_z \Rightarrow (\ell_x,\ \ell_y) :\ \mathsf{back}\ \ell_z\ ]\!]$

```
def inline_choose
  {T : Type} (b : Bool) (x y : T) :
  Result T := do
  let (b0, z, back) ←
    if b then ok (true, x, fun z => (z, y))
    else (false, y, fun z => (x, z))
```

We skip the end and show the result of the synthesis below:

```
def inline_choose
  {T : Type} (b : Bool) (x y : T) :
  Result T := do
```

```
  let (b0, z, back) ←
    if b then ok (true, x, fun z => (z, y))
    else (false, y, fun z => (x, z))
  let z0 ← z + 1
  let (x0, _) = back z0
  ok x0
```

Finally, we show the result of the translation after running the micro-passes, which in particular eliminate the useless b0:

```
def inline_choose
  {T : Type} (b : Bool) (x y : T) :
  Result T := do
  let (z, back) ←
    if b then ok (x, fun z => (z, y))
    else (y, fun z => (x, z))
  let z0 ← z + 1
  let (x0, _) = back z0
  ok x0
```

13.4 Translation Example: Loop

We conclude this series of examples with the loop below.

```
1  // x ↦ 0,   y ↦ σ_y
2  loop {
3    if copy x < copy y {
4      x += 1;
5      continue;
6    }
7    else {
8      break;
9    }
10 }
```

We directly jump to the interesting part and omit the computation of the fixed-point at the entry of the loop, which is: $x \mapsto \sigma_x,\ y \mapsto \sigma_y$. We now have two things to translate: the body of the loop, and the snippet of code around the loop; for the body we generate a separate, top-level recursive function, which we then use for the translation of the snippet of code above. We show the translation of the loop below. First, we note that the environment contains two symbolic values: the translation of the loop should thus take one input per symbolic value[1]. Of course, this often generates

[1]It should also take as input one continuation per region abstraction; there is none here.

loops with useless parameters; a micro-pass later detects and eliminates those, in order to generate code which is as tight as possible.

We also compute the join of all the environments we get upon reaching a break in the loop; we dub the resulting environment the *output* environment of the loop. We will use this output environment to resume the execution *after* the loop, and use it at present to compute the output type of the translation. This environment is actually $x \mapsto \sigma_x,\ y \mapsto \sigma_y$; as it contains two symbolic values of type u32, the loop outputs a pair of u32:

$x \mapsto \sigma_x$
$y \mapsto \sigma_y$

```
def loop (x y : U32) :
  Result (U32 × U32) := do
  [.]
```

We elide the treatment of the `if` and the addition, and focus on the continue; upon reaching the continue we have:

$x \mapsto \sigma'_x$
$y \mapsto \sigma_y$

```
def loop (x y : U32) :
  Result (U32 × U32) := do
  if x < y then do
    let x ← x + 1
    [.]
  else
    [.]
```

We introduce a recursive call to the (translation of) the loop. In order to so, we match the current environment (after eventually transforming it following $\leq$) with the fixed-point environment to compute a mapping from the symbolic values in the fixed-point environment to values in the current environment; this allows us to find the inputs of the loop. In the present case, we get: $\sigma_x \mapsto \sigma'_x$ and $\sigma_y \mapsto \sigma_y$. We get:

```
def loop (x y : U32) :
  Result (U32 × U32) := do
  if x < y then do
    let x1 ← x + 1
    loop x1 y
  else
    [.]
```

We then proceed to the `else` branch, and have to evaluate the break. Similarly to the case of the continue, we match the current environment with the *output* environment to compute a mapping from the symbolic values in this output environment to values of the current environment. From this, we deduce that we should return the pair `(x1, y)`:

```
def loop (x y : U32) :
  Result (U32 × U32) := do
  if x < y then do
    let x1 ← x + 1
    loop x1 y
  else
    ok (x1, y)
```

We now switch back to the snippet of code containing the loop. Upon entering the loop we have environment $x \mapsto 0, \quad y \mapsto \sigma_y$, that we match against the fixed-point environment $x \mapsto \sigma_x,\ y \mapsto \sigma_y$. We get the following mapping: $\sigma_x \mapsto 0$ and $\sigma_y \mapsto \sigma_y$; as a consequence we insert a call to loop 0 y. We also continue the execution with the output environment. We get:

```
let (x1, y1) ← loop 0 y
[.]
```

Finally, now that the translation is complete, we note that y is threaded unchanged through the recursive calls of the loop. As a consequence, it does not need to be an output; we detect and fix this in a micro-pass. We get:

```
def loop (x y : U32) :
  Result U32 := do
  if x < y then do
    let x1 ← x + 1
    loop x1 y
  else
    ok x1

-- The snippet of code around the loop:
let x1 ← loop 0 y
[.]
```

13.5 Synthesis Rules

The rules are in figures 13.1, 13.2, 13.3, 13.4, 13.5, 13.6, and 13.7. They describe a process in which we traverse the source program in a forward fashion, simultaneously updating our symbolic environment and generating pure λ-terms. The translation is currently trusted, and its correctness left as future work.

The rules are slightly modified versions of the semantics of LLBC$^\#$. We introduce a new judgement $\Omega \vdash s \rightsquigarrow S^\# \Downarrow_s e$, which means that statement s evaluates to set of tagged states $S^\#$ while compiling down to pure expression e. For statements that are

in terminal position, we modify the semantics so that they evaluate to an empty set of tagged states, meaning we can stop the synthesis there. For instance, when reaching the statement panic we simply synthesize the expression fail.

But for statements that are not in terminal position (i.e., on the left-hand side of a semicolon), we are faced with the usual mismatch between statement-based languages and let/expression-based languages. We solve the issue by allowing expressions to contain holes which receive a continuation – we write $E[\cdot]$ (or $E[\vec{\cdot}]$ when there may be several holes). In practice, our implementation uses continuation-passing style to keep things readable, as opposed to an AST definition with holes for our target language. We spare the reader the grammar of expressions e, which is a standard lambda-calculus; suffices to say that we use $\leftarrow$ to denote the monadic bind operator, as in Chapter 8; successful computations evaluate to ok, while failing (i.e., panicking) computations evaluate to fail. The synthesis of values is captured by the rules Pure-* (Figure 13.1); as we alluded to earlier, our translation never encounters, nor produces, a source variable x; rather, they structurally visit a symbolic value and map source *symbolic* values to variables in the target λ-calculus (Pure-Symb). Naturally, in practice, we use heuristics to pick sensible names for the symbolic variables, thus guaranteeing that the output of our translation is readable. Conversion of types from source to target is almost the identity, except for $\mathsf{Box}\,\tau$, $\&^{\rho}\mathsf{mut}\,\tau$ and $\&^{\tau}$ which become τ, consistently with Pure-Box, Pure-Mut-Borrow and Pure-Shared-Borrow.

Another insight about our rules: in order to make progress, we may synthesize fresh bindings at any time via a reorganization. For instance, if a symbolic value with a tuple type is refined into the tuple of its components, we need to mirror this fact in the generated program (Synth-Reorg-SymbolicPair). That is, if we use a reorganization phase to refine the symbolic value $\sigma : (\tau_0,\ \tau_1)$ into the pair $(\sigma : \tau_0,\ \sigma : \tau_1)$, then we generate the following expression in the translation: $\mathsf{let}\ (\sigma_0,\ \sigma_1) = \sigma\ \mathsf{in}\ \ldots$.

The rest of the rules leverage our earlier concepts of region abstractions, projections, and symbolic environments to precisely capture the relationship between a function body and its parameters (callee), or a function application and its arguments (caller).

We adopt the perspective of the caller and begin with function calls. In Synth-E-Call, we follow the procedure outlined in our earlier examples. First, we allocate fresh symbolic values and borrows for the output value v_{out}; those must be output by the call to the function in the synthesized code. We also introduce one region abstraction per region in the function type. For the purpose of the synthesis, region abstractions are annotated by continuations; those are also output by the call in the translation.

Matches and `if else` are delicate, and come in several flavors. We remark that our matches are made up of non-nested, constructor patterns; by the time we examine

Rust's internal MIR language, nested patterns have already been desugared.

If there is enough static knowledge to determine which branch we should take, we do not bother with generating a trivial match or `if then else` expression and simply generate code for the corresponding branch (Synth-E-IfThenElse-True).

If there is not enough static knowledge, because for instance we are matching over a symbolic value, then we synthesize a branching in the translation. We have two possibilities. We either execute the two branches *without* performing a join afterwards, leading to a disjunction of the control-flow (Synth-E-IfThenElse-NoJoin); because we are currently using an error monad in the translation, which only models successful executions and panics, we currently leverage this possibility when one of the branches contains a return as it forces us to break the control-flow. Otherwise, we join the environments together (Synth-E-IfThenElse-Join). Because the join introduces fresh symbolic values, and fresh symbolic values need to be bound by let-bindings in the translation, we update the rules for the join so that they also synthesize code for the left and right branches. For instance, Synth-Join-Symbolic, which states that we can join two different values (if they don't contain loans, borrows, or $\bot$), into a fresh symbolic value σ, binds σ in the code synthesized for the two branches.

When reaching a return (Synth-E-Return), we have to exhibit a series of transformations (according to the rules of $\leq$) to turn the current state into a target state which derives from the function signature. By doing those transformations, we actually derive the continuations which will be used for the backward functions; for instance, we may merge two region abstractions together, as shown in Section 13.3, which has the effect of composing their respective continuations. For this reason, we update the rules of $\leq$ to also synthesize code. For instance, when using Synth-Le-ToSymbolic to replace a value v with a fresh symbolic value σ, we let-bind σ in the synthesized code. Finally, when synthesizing a function (Synth), we simply apply the synthesis on its body, starting with a properly initialized state.

$$\textsc{Pure-Mut-Borrow}\ \frac{\Omega \vdash v \Downarrow_s e}{\Omega \vdash \mathsf{borrow}^m\ \ell\ v \Downarrow_s e} \qquad \textsc{Pure-Const}\ \frac{}{\Omega \vdash n_{\mathsf{i32}} \Downarrow_s n_{\mathsf{i32}}} \qquad \textsc{Pure-Pair}\ \frac{\Omega \vdash v_l \Downarrow_s \vec{e_l} \quad \Omega \vdash v_r \Downarrow_s \vec{e_r}}{\Omega \vdash (v_l, v_r) \Downarrow_s (e_l, e_r)} \qquad \textsc{Pure-Left}\ \frac{\Omega \vdash v \Downarrow_s \vec{e}}{\Omega \vdash \mathsf{Left}\, v \Downarrow_s \mathsf{Left}\, e}$$

$$\textsc{Pure-Right}\ \frac{\Omega \vdash v \Downarrow_s \vec{e}}{\Omega \vdash \mathsf{Right}\, v \Downarrow_s \mathsf{Right}\, e} \qquad \textsc{Pure-Symb}\ \frac{}{\Omega \vdash \sigma \Downarrow_s \sigma} \qquad \textsc{Pure-Box}\ \frac{\Omega \vdash v \Downarrow_s e}{\Omega \vdash \mathsf{Box}\, v \Downarrow_s e} \qquad \textsc{Pure-Shared-Borrow}\ \frac{\mathsf{loan}^s\ \vec{\ell}\, v \in \Omega \qquad \ell \in \vec{\ell} \quad \Omega \vdash v \Downarrow_s e}{\Omega \vdash \mathsf{borrow}^s\ \ell \Downarrow_s e}$$

Figure 13.1: Synthesis Rules: Values

$$\textsc{Synth-Reorg-End-MutBorrow}\ \frac{\begin{array}{c}\text{first hole of } \Omega[.,.] \text{ not inside a region abstraction}\\ \text{second hole of } \Omega[.,.] \text{ not inside a borrowed value or a region abstraction}\\ \text{no loan, } \mathsf{borrow}^r \in v\end{array}}{\Omega[\mathsf{loan}^m\ \ell,\ \mathsf{borrow}^m\ \ell\ v] \hookrightarrow \Omega[v, \bot] \Downarrow_s [.]}$$

$$\textsc{Synth-Reorg-End-MutBorrow-Abs}\ \frac{\begin{array}{c}\text{hole of } \Omega[.] \text{ not inside a borrowed value or a region abstraction}\\ \text{no loan, } \mathsf{borrow}^r \in v\\ \Omega[\bot] \vdash v \Downarrow_s e\end{array}}{\begin{array}{c}\Omega[\mathsf{borrow}^m\ \ell_k^{\mathsf{in}}\ v],\ A\ \{\ \mathsf{loan}^m\ \ell_k^{\mathsf{in}}\ \}[\![\ (\overrightarrow{\ell_i^{\mathsf{out}}}) :\ \lambda\ \overrightarrow{\ell^{\mathsf{in}}} \Rightarrow\ f\overrightarrow{\ell_j^{\mathsf{in}}}\ ]\!] \hookrightarrow\\ \Omega[v, \bot],\ A\{v\}[\![\ (\overrightarrow{\ell_i^{\mathsf{out}}}) :\ \lambda\ \overrightarrow{\ell_{j\neq k}^{\mathsf{in}}} \Rightarrow\ \mathsf{let}\ \ell_k^{\mathsf{in}} = e\ \mathsf{in}\ f\overrightarrow{\ell_j^{\mathsf{in}}}\ ]\!], \Downarrow_s [.]\end{array}}$$

$$\textsc{Synth-Reorg-End-SharedReservedBorrow}\ \frac{\text{hole of } \Omega[.] \text{ not inside a borrowed value or a region abstraction}}{\Omega[\mathsf{borrow}^{s,r}\ \ell] \hookrightarrow \Omega[\bot] \Downarrow_s [.]}$$

$$\textsc{Synth-Reorg-End-SharedLoan}\ \frac{\mathsf{borrow}^{s,r}\ \ell \notin \Omega[\mathsf{loan}^s\ \ell\ v]}{\Omega[\mathsf{loan}^s\ \ell\ v] \hookrightarrow \Omega[v] \Downarrow_s [.]}$$

$$\textsc{Synth-Reorg-Activate-Reserved}\ \frac{\begin{array}{c}\mathsf{loan}, \mathsf{borrow}^r \notin v\\ \text{hole of } \Omega[., \mathsf{borrow}^r\ \ell] \text{ not inside a shared value}\\ \ell \notin \Omega[.,.], v\end{array}}{\begin{array}{c}\Omega[\mathsf{loan}^s\ \ell\ v, \mathsf{borrow}^r\ \ell] \hookrightarrow\\ \Omega[\mathsf{loan}^m\ \ell, \mathsf{borrow}^m\ \ell\ v] \Downarrow_s [.]\end{array}}$$

$$\textsc{Synth-Reorg-Seq}\ \frac{\Omega_0 \hookrightarrow \Omega_1 \Downarrow_s E_1[.] \qquad \Omega_2 \hookrightarrow \Omega_2 \Downarrow_s E_2[.]}{\Omega_0 \hookrightarrow \Omega_2 \Downarrow_s E_1[E_2[.]]}$$

$$\textsc{Synth-Reorg-None}\ \frac{}{\Omega \hookrightarrow \Omega \Downarrow_s [.]}$$

$$\textsc{Synth-Reorg-End-Abstraction}\ \frac{\begin{array}{c}\text{no borrows, loans} \in \overrightarrow{v},\ \overrightarrow{v'} \qquad \overrightarrow{\sigma} \text{ fresh}\end{array}}{\begin{array}{c}\Omega,\ A\ \{\ \overrightarrow{v},\ \overrightarrow{\mathsf{borrow}^s\ \ell},\ \overrightarrow{\mathsf{borrow}^m\ \ell^{\mathsf{out}}\ (v' : \tau)}\ \}[\![\ \overrightarrow{\ell_i^{\mathsf{out}}} = \overrightarrow{e}\ ]\!] \hookrightarrow\\ \Omega,\ \overrightarrow{\_ \to \mathsf{borrow}^s\ \ell},\ \overrightarrow{\_ \to \mathsf{borrow}^m\ \ell^{\mathsf{out}}\ (\sigma : \tau)} \Downarrow_s \mathsf{let}\ \overrightarrow{\sigma} = \overrightarrow{e}\ \mathsf{in}\ [.]\end{array}}$$

$$\textsc{Synth-Reorg-SymbolicBox}\ \frac{\sigma' \text{ fresh}}{\Omega[\sigma : \mathsf{Box}\,\tau] \hookrightarrow \Omega[\mathsf{Box}\,\sigma'] \Downarrow_s \mathsf{let}\ \sigma' = \sigma\ \mathsf{in}\ [.]}$$

$$\textsc{Synth-Reorg-SymbolicPair}\ \frac{\sigma_0,\ \sigma_1 \text{ fresh}}{\Omega[\sigma : (\tau_0,\ \tau_1)] \hookrightarrow \Omega[(\sigma_0,\ \sigma_1)] \Downarrow_s \mathsf{let}\ (\sigma_0, \sigma_1) = \sigma\ \mathsf{in}\ [.]}$$

Figure 13.2: Synthesis Rules: Reorganization

$$\frac{\text{borrows, loans, } \bot \notin v \quad \sigma \text{ fresh} \quad \Omega \vdash v \Downarrow_s v'}{\Omega[v] \leq \Omega[\sigma] \Downarrow_s \mathsf{let}\ \sigma = v'\ \mathsf{in}\ [.]}\ \textsc{Synth-Le-ToSymbolic} \qquad \frac{\Omega \vdash v \prec^{\text{to-abs}} e,\ \overrightarrow{A}}{\Omega,\ \_ \to v \leq \Omega,\ \overrightarrow{A} \Downarrow_s [.]}\ \textsc{Synth-Le-ToAbs}$$

$$\frac{\text{no outer loans in } v \quad \text{hole of } \Omega[.] \text{ not inside a shared loan or a region abstraction}}{\Omega[v] \leq \Omega[\bot],\ \_ \mapsto v \Downarrow_s [.]}\ \textsc{Synth-Le-MoveValue} \qquad \frac{}{\Omega,\ A\ \{\} \leq \Omega \Downarrow_s [.]}\ \textsc{Synth-Le-ClearAbs}$$

$$\frac{\begin{array}{c}\vdash A_0[\![\ (\overrightarrow{\ell_0^{\mathsf{out}}}) :\ \lambda\ \overrightarrow{\ell_0^{\mathsf{in}}} \Rightarrow f_0\ \overrightarrow{\ell_0^{\mathsf{in}}}\ ]\!] \bowtie A_1[\![\ (\overrightarrow{\ell_1^{\mathsf{out}}}) :\ \lambda\ \overrightarrow{\ell_1^{\mathsf{in}}} \Rightarrow f_1\ \overrightarrow{\ell_1^{\mathsf{in}}}\ ]\!] = A \\ A = \{\ ...,\ \overrightarrow{\mathsf{borrow}^m\ \ell^{\mathsf{out}}\ (v' : \tau)},\ \overrightarrow{\mathsf{loan}^m\ \ell^{\mathsf{in}}}\ \}\end{array}}{\Omega,\ A_0,\ A_1 \leq \Omega,\ A[\![\ (\overrightarrow{\ell^{\mathsf{out}}}) :\ \lambda\ \overrightarrow{\ell^{\mathsf{in}}} \Rightarrow (\mathsf{let}\ \overrightarrow{\ell_1^{\mathsf{out}}} = f_1\ \overrightarrow{\ell_1^{\mathsf{in}}}\ \mathsf{in}\ \mathsf{let}\ \overrightarrow{\ell_0^{\mathsf{out}}} = f_0\ \overrightarrow{\ell_0^{\mathsf{in}}}\ \mathsf{in}\ \overrightarrow{\ell^{\mathsf{out}}})\ ]\!]}\ \textsc{Synth-Le-MergeAbs}$$

$$\frac{\ell \text{ fresh}}{\Omega[v] \leq \Omega[\mathsf{loan}^m\ \ell],\ \_ \to \mathsf{borrow}^m\ \ell\ v \Downarrow_s [.]}\ \textsc{Synth-Le-Fresh-MutLoan} \qquad \frac{\ell \text{ fresh}}{\Omega[v] \leq \Omega[\mathsf{loan}^s\ \ell\ v] \Downarrow_s [.]}\ \textsc{Synth-Le-Fresh-SharedLoan}$$

$$\frac{\ell_1 \text{ fresh}}{\Omega[\mathsf{borrow}^m\ \ell_0\ v] \leq \Omega[\mathsf{borrow}^m\ \ell_1\ v],\ \_ \to \mathsf{borrow}^m\ \ell_0\ (\mathsf{loan}^m\ \ell_1) \Downarrow_s [.]}\ \textsc{Synth-Le-Reborrow-MutBorrow}$$

$$\frac{\text{no symbolic values, borrows, loans} \in v}{\Omega \leq \Omega,\ \_ \to \bot \Downarrow_s [.]}\ \textsc{Synth-Le-AnonValue}$$

Figure 13.3: Synthesis Rules: the $\leq$ Relation (Selected Rules)

$$\textsc{ToAbs-Empty}\ \frac{\text{no borrows, loans} \in v\Omega \vdash v \Downarrow_s e}{\Omega \vdash v \prec^{\text{to-abs}} e,\ \emptyset}$$

$$\textsc{ToAbs-Left}\ \frac{\Omega \vdash v \prec^{\text{to-abs}} e,\ \overrightarrow{A}}{\Omega \vdash \mathsf{Left}\, v \prec^{\text{to-abs}} \mathsf{Left}\, e,\ \overrightarrow{A}}$$

$$\textsc{ToAbs-Right}\ \frac{\Omega \vdash v \prec^{\text{to-abs}} e,\ \overrightarrow{A}}{\Omega \vdash \mathsf{Right}\, v \prec^{\text{to-abs}} \mathsf{Right}\, e,\ \overrightarrow{A}}$$

$$\textsc{ToAbs-Box}\ \frac{\Omega \vdash v \prec^{\text{to-abs}} e,\ \overrightarrow{A}}{\Omega \vdash \mathsf{Box}\, v \prec^{\text{to-abs}} e\ \overrightarrow{A}}$$

$$\textsc{ToAbs-SharedBorrow}\ \frac{A \text{ fresh} \qquad \mathsf{loan}^s\, \ell\, v \in \Omega \qquad \Omega \vdash v \Downarrow_s e}{\Omega \vdash \mathsf{borrow}^s\, \ell \prec^{\text{to-abs}} e,\ A\{\mathsf{borrow}^s\, \ell\}}$$

$$\textsc{ToAbs-SharedLoan}\ \frac{A \text{ fresh} \qquad \Omega \vdash v \Downarrow_s e}{\Omega \vdash \mathsf{loan}^s\ \overrightarrow{\ell}\ v \prec^{\text{to-abs}} e,\ A\{\mathsf{loan}^s\ \overrightarrow{\ell}\ v\}}$$

$$\textsc{ToAbs-MutBorrow}\ \frac{\text{no borrows}, \bot \in v \qquad \Omega \vdash v \prec^{\text{to-abs}} e,\ \overrightarrow{A_i[\![(\overrightarrow{\ell_i^{\mathsf{out}}}) :\ \lambda\, \overrightarrow{\ell_i^{\mathsf{in}}} \Rightarrow f_i\, \overrightarrow{\ell_i^{\mathsf{in}}}]\!]}}{\Omega \vdash \mathsf{borrow}^m\, \ell\, v \prec^{\text{to-abs}} e,\ (\cup \overrightarrow{A}) \cup \{\mathsf{borrow}^m\, \ell\ \_\}[\![\ (\ell,\ ...,\ \overrightarrow{\ell_i^{\mathsf{out}}},\ ...) :\ \lambda\, \overrightarrow{\ell_i^{\mathsf{in}}} \Rightarrow (e,\ ...,\ f_i\, \overrightarrow{\ell_i^{\mathsf{in}}},\ ...)\ ]\!]}$$

$$\textsc{ToAbs-MutLoan}\ \frac{A \text{ fresh}}{\vdash \mathsf{loan}^m\, \ell \prec^{\text{to-abs}} \ell,\ A\{\mathsf{loan}^m\, \ell\}[\![\ () :\ \lambda\, \ell \Rightarrow ()\ ]\!]}$$

$$\textsc{ToAbs-Pair}\ \frac{\vdash v_l \prec^{\text{to-abs}} \overrightarrow{e}_l,\ A_l \qquad \vdash v_r \prec^{\text{to-abs}} \overrightarrow{e}_r,\ A_r}{\vdash (v_l, v_r) \prec^{\text{to-abs}} \overrightarrow{(}\, e_l,\ e_r),\ A_l,\ \overrightarrow{A}_r}$$

Figure 13.4: Synthesis Rules: Transforming Values to Region Abstractions

SYNTH-E-RETURN

$$\frac{f\langle\vec{\rho},\,\vec{\tau}\rangle = \mathsf{fn}\ \langle\vec{\rho}\rangle\ (\overrightarrow{x} : \overrightarrow{\tau})\ (\overrightarrow{y} : \overrightarrow{\tau'})\ (x_{\mathsf{ret}} : \tau_{\mathsf{ret}})\ \{\ s\ \} \text{ is the function being synthesized} \quad \mathsf{final}\ (\vec{\rho},\,\vec{\tau},\,\tau) = (v,\,\overrightarrow{A(\rho)}) \quad \Omega' = x_{\mathsf{ret}} \mapsto v,\ \overrightarrow{x} \mapsto \overrightarrow{\bot},\ \overrightarrow{A(\rho)[\![\ (\ell_\rho^{\mathsf{out}}) :\ \lambda\,\ell_\rho^{\mathsf{in}} \Rightarrow f_\rho\ \overrightarrow{\ell_\rho^{\mathsf{in}}}\ ]\!]} \quad \Omega \leq \Omega' \Downarrow_s E[.] \quad \Omega' \vdash v \Downarrow_s e}{\Omega \vdash \mathsf{return} \rightsquigarrow \emptyset \Downarrow_s E[\mathsf{return}\ (e,\ \overrightarrow{f_\rho})]}$$

SYNTH-E-PANIC

$$\frac{}{\Omega \vdash \mathsf{panic} \rightsquigarrow \emptyset \Downarrow_s \mathsf{fail}}$$

SYNTH-SEQ-SYMBOLIC

$$\frac{\Omega \vdash s_0 \rightsquigarrow \{((), \Omega_i)\} \Downarrow_s E[\,\vec{.}\,] \quad \forall\, i,\ \Omega_i \vdash s_1 \rightsquigarrow S_i^{\#} \Downarrow_s E_i[\,\vec{.}\,]}{\Omega \vdash s_0;\ s_1 \rightsquigarrow \cup\,(\bigcup_i S_i^{\#}) \Downarrow_s E[\ldots,\ E_i[\,\vec{.}\,], \ldots]}$$

SYNTH-E-IFTHENELSE-TRUE

$$\frac{\Omega \vdash op \Downarrow (v,\,\Omega') \quad v = \mathsf{true} \vee v = \mathsf{loan}^s\,\ell\,\mathsf{true} \quad \Omega \vdash s_0 \rightsquigarrow S^{\#} \Downarrow_s E[\,\vec{.}\,]}{\Omega \vdash \mathsf{if}\ op\ \mathsf{then}\ s_0\ \mathsf{else}\ s_1 \rightsquigarrow S^{\#} \Downarrow_s E[\,\vec{.}\,]}$$

SYNTH-E-IFTHENELSE-NOJOIN

$$\frac{\Omega \vdash op \Downarrow (v,\,\Omega') \quad v = (\sigma : \mathsf{bool}) \vee v = \mathsf{loan}^s\,\ell\,(\sigma : \mathsf{bool}) \quad \Omega_0 = \Omega'[\mathsf{true}/\sigma] \quad \Omega_1 = \Omega'[\mathsf{false}/\sigma] \quad \Omega_0 \vdash s_0 \rightsquigarrow S_0^{\#} \Downarrow_s E_0[\,\vec{.}\,] \quad \Omega_1 \vdash s_1 \rightsquigarrow S_1^{\#} \Downarrow_s E_1[\,\vec{.}\,]}{\Omega \vdash \mathsf{if}\ op\ \mathsf{then}\ s_0\ \mathsf{else}\ s_1 \rightsquigarrow S_0^{\#} \cup S_1^{\#} \Downarrow_s \mathsf{if}\ \sigma\ \mathsf{then}\ E_0[\,\vec{.}\,]\ \mathsf{else}\ E_1[\,\vec{.}\,]}$$

SYNTH-E-IFTHENELSE-JOIN

$$\frac{\Omega \vdash op \Downarrow (v,\,\Omega') \quad v = (\sigma : \mathsf{bool}) \vee v = \mathsf{loan}^s\,\ell\,(\sigma : \mathsf{bool}) \quad \Omega_0 = \Omega'[\mathsf{true}/\sigma] \quad \Omega_1 = \Omega'[\mathsf{false}/\sigma] \quad \Omega_0 \vdash s_0 \rightsquigarrow \{(), \Omega_0'\} \Downarrow_s E_0[.] \quad \Omega_1 \vdash s_1 \rightsquigarrow \{(), \Omega_1'\} \Downarrow_s E_1[.] \quad \Omega_0',\ \Omega_1' \vdash \mathsf{join}_\Omega\ \Omega_0'\ \Omega_1' \rightsquigarrow \Omega_j \mid E_0^j[.] \mid E_1^j[.] \quad \Omega_j \searrow \Omega_c \quad \Omega_1 \vdash s_1 \rightsquigarrow S_1^{\#} \Downarrow_s E_1[\,\vec{.}\,] \quad \overrightarrow{\sigma} \text{ and } \overrightarrow{A[\![\ \lambda\,\overrightarrow{\ell^{\mathsf{out}}} \Rightarrow \overrightarrow{\ell^{\mathsf{in}}} :\ f\ \overrightarrow{\ell^{\mathsf{in}}}\ ]\!]} \text{ are the fresh symbolic values and abs. of } \Omega_c}{\Omega \vdash \mathsf{if}\ op\ \mathsf{then}\ s_0\ \mathsf{else}\ s_1 \rightsquigarrow ((),\ \Omega_c) \Downarrow_s \mathsf{let}\ (\overrightarrow{\sigma},\ \overrightarrow{f}) \leftarrow \Big(\mathsf{if}\ \sigma\ \mathsf{then}\ E_0[\ E_0^j[\ \mathsf{ok}\ (\overrightarrow{\sigma},\ \overrightarrow{\mathsf{proj\_left}\ f})\ ]\ ]\ \mathsf{else}\ E_1[\ E_1^j[\ \mathsf{ok}\ (\overrightarrow{\sigma},\ \overrightarrow{\mathsf{proj\_right}\ f})\ ]\ ]\Big);\ [.]}$$

SYNTH-E-CALL

$$\frac{f\langle\vec{\rho},\,\vec{\tau}\rangle = \mathsf{fn}\ \langle\vec{\rho}\rangle\ (\overrightarrow{x} : \overrightarrow{\tau})\ (\overrightarrow{y} : \overrightarrow{\tau'})\ (x_{\mathsf{ret}} : \tau_{\mathsf{ret}})\ \{\ s\ \} \quad \Omega_j \vdash op_j \Downarrow (v_j,\,\Omega_{j+1}) \quad \overrightarrow{\rho}\ \text{fresh} \quad \overrightarrow{A_{\mathsf{sig}}(\rho)},\ v_{\mathsf{out}} = \mathsf{inst\_sig}(\Omega_n, \vec{\rho}, \vec{v}, \tau_{\mathrm{ret}}) \quad \forall\rho,\ A_\rho = \{\ \ldots,\ \overrightarrow{\mathsf{borrow}^m\,\ell_\rho^{\mathsf{out}}\ \_},\ \overrightarrow{\mathsf{loan}^m\,\ell_\rho^{\mathsf{in}}}, \ldots\ \} \quad \overrightarrow{f_\rho^{\mathsf{back}}}\ \text{fresh} \quad \Omega' = \Omega_n,\ \overrightarrow{A_{\mathsf{sig}}(\rho)[\![\ (\ell_\rho^{\mathsf{out}}) :\ \lambda\,\ell_\rho^{\mathsf{in}} \Rightarrow f_\rho^{\mathsf{back}}\ \ell_\rho^{\mathsf{in}}\ ]\!]} \quad \forall j,\ \Omega_{j+1} \vdash v_j \Downarrow_s e_j \quad \Omega' \vdash v_{\mathsf{out}} \Downarrow_s e \quad \Omega' \vdash p := v_{\mathsf{out}} \rightsquigarrow ((), \Omega'') \Downarrow_s E[.]}{\Omega_0 \vdash p := f\langle\_,\,\vec{\tau}\rangle(\overrightarrow{op}) \rightsquigarrow ((), \Omega'') \Downarrow_s \mathsf{let}\ (e,\ \overrightarrow{f_\rho^{\mathsf{back}}}) = f\ \overrightarrow{e_j}\ \mathsf{in}\ E[.]}$$

SYNTH

$$\frac{f\langle\vec{\rho},\,\vec{\tau}\rangle = \mathsf{fn}\ \langle\vec{\rho}\rangle\ (\overrightarrow{x} : \overrightarrow{\tau})\ (\overrightarrow{y} : \overrightarrow{\tau'})\ (x_{\mathsf{ret}} : \tau_{\mathsf{ret}})\ \{\ s\ \} \quad \mathsf{init}\ (\vec{\rho},\,\vec{\tau}) = (\overrightarrow{v_i},\,\overrightarrow{A(\rho)}) \quad \Omega = \overrightarrow{A(\rho)[\![\mathsf{id}]\!]},\ \overrightarrow{x} \mapsto \overrightarrow{v},\ \overrightarrow{y} \mapsto \overrightarrow{\bot}\ x_{\mathsf{ret}} \mapsto \bot \quad \Omega \vdash s \rightsquigarrow \emptyset \Downarrow_s e_f \quad \Omega \vdash \overrightarrow{v} \Downarrow_s \overrightarrow{e}}{f \Downarrow_s \lambda\,\overrightarrow{e} \Rightarrow e_f}$$

Figure 13.5: Synthesis Rules: Evaluation Rules (Selected Rules)

$$\textsc{Synth-Join-Same}\quad \frac{}{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ v\ v \Downarrow v \mid \emptyset \mid [.] \mid [.]}$$

$$\textsc{Synth-Join-Symbolic}\quad \frac{\text{no borrows, loans, } \bot \in v_0,\ v_1 \qquad \sigma \text{ fresh} \qquad \Omega_0 \vdash v_0 \Downarrow_s v_0^{(p)} \qquad \Omega_1 \vdash v_1 \Downarrow_s v_1^{(p)}}{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ v_0\ v_1 \Downarrow \sigma \mid \emptyset \mid \mathsf{let}\ \sigma = v_0^{(p)}\ \mathsf{in}\ [.] \mid \mathsf{let}\ \sigma = v_1^{(p)}\ \mathsf{in}\ [.]}$$

$$\textsc{Synth-Join-MutBorrows}\quad \frac{\ell_2,\ A' \text{ fresh} \qquad \Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ v_0\ v_1 \Downarrow v_2 \mid \overrightarrow{A} \mid E_0[.] \mid E_1[.]}{\begin{array}{c}\Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ (\mathsf{borrow}^m\ \ell_0\ v_0)\ (\mathsf{borrow}^m\ \ell_1\ v_1) \Downarrow \mathsf{borrow}^m\ \ell_2\ v_2 \mid \\ A' \{ \boxed{\mathsf{borrow}^m\ \ell_0\ \_}, \boxed{\mathsf{borrow}^m\ \ell_1\ \_}, \mathsf{loan}^m\ \ell_2 \} [\![\lambda\ \ell_2 \Rightarrow (\boxed{\ell_0}, \boxed{\ell_1}) : (\boxed{\ell_2}, \boxed{\ell_2})]\!],\ \overrightarrow{A} \\ \mid E_0[.] \mid E_1[.]\end{array}}$$

$$\textsc{Synth-Join-SharedBorrows}\quad \frac{\ell_2,\ \sigma,\ A \text{ fresh} \qquad \text{no } \mathsf{loan}^m \text{, borrow, } \bot \in v_0,\ v_1 \qquad \mathsf{loan}^s\ \ell_0\ v_0 \in \Omega_0 \qquad \mathsf{loan}^s\ \ell_1\ v_1 \in \Omega_1}{\begin{array}{c}\Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ (\mathsf{borrow}^s\ \ell_0)\ (\mathsf{borrow}^s\ \ell_1) \Downarrow \mathsf{borrow}^s\ \ell_2 \mid \\ A \{ \boxed{\mathsf{borrow}^s\ \ell_0}, \boxed{\mathsf{borrow}^s\ \ell_1}, \mathsf{loan}^s\ \ell_2\ \sigma \} [\![\lambda() \Rightarrow () :\ ()]\!] \mid E_0[.] \mid E_1[.]\end{array}}$$

Figure 13.6: Synthesis Rules: Joining Values (Selected Rules)

$$\mathsf{proj\_left}\ \left(\overrightarrow{\ell^{\mathsf{in}}} :\ \lambda\ \overrightarrow{\ell^{\mathsf{in}}} \Rightarrow = (..., e_l \oplus e_r, ...)\right) = \left(\overrightarrow{\ell^{\mathsf{in}}} :\ \lambda\ \overrightarrow{\ell^{\mathsf{in}}} \Rightarrow = (..., e_l, ...)\right)$$

$$\mathsf{proj\_right}\ \left(\overrightarrow{\ell^{\mathsf{in}}} :\ \lambda\ \overrightarrow{\ell^{\mathsf{in}}} \Rightarrow = (..., e_l \oplus e_r, ...)\right) = \left(\overrightarrow{\ell^{\mathsf{in}}} :\ \lambda\ \overrightarrow{\ell^{\mathsf{in}}} \Rightarrow = (..., e_r, ...)\right)$$

Figure 13.7: Synthesis Rules: Projecting Continuations for the Join

Chapter 14

Evaluation

14.1 Implementing Aeneas

Our implementation is written in a mixture of Rust and OCaml. A first tool, dubbed Charon, performs the translation from Rust's MIR internal representation to LLBC. Concretely, Charon is a Rust compiler plugin that performs a large amount of mundane, tedious tasks, such as: computing a dependency graph, reordering definitions, grouping mutually recursive definitions together, reconstructing data type creation, and generally getting rid of the idioms that are definitely too low-level for LLBC (Section 9.2). Once this is done, Charon dumps a JSON file to disk containing the LLBC AST. We plan to switch to a more efficient binary format in the future. As of today, Charon totals 16 kLoC (lines of code, excluding whitespace and comments). Charon lives as a separate project because we believe it has an existence of its own outside of Aeneas. We are actually using it for other projects such as the Eurydice compiler [413] from Rust to C, that we use for the purpose of writing all new implementations in Rust while still providing code for legacy projects which require C. We carefully designed Charon to make it a reusable library; this allowed in particular the Kani Rust Verifier [414] to implement an experimental backend which reuses the micro-passes implemented by Charon [415].

Aeneas picks up the Charon-generated AST, and implements the transformations described in Chapter 9 and Chapter 13. Practically speaking, we have a single interpreter that runs in two modes, either concrete or symbolic. The former produces a final value, if running a closed term; the latter produces a translated program. We currently extract to F*, Coq, HOL4 and Lean. Effectively, our symbolic interpreter acts as a borrow checker for Rust programs using our semantic notion of borrows We have written Aeneas in OCaml, a language much better suited to the manipulation of ASTs than

Rust. The implementation of Aeneas totals 35kLoC.

The implementation is, naturally, trusted. However, we have taken extraordinary care to ensure that it is trustworthy. Notably, when running the code in CI, after every application of one of the rules, we verify a large amount of invariants, such as: the environment is well-typed; borrows are consistent; shared values don't contain a mutable loan, etc. In practice, those invariants are extremely tight, and have led to the great level of detail that our rules exhibit. Should one turn off those invariant checks, the whole Charon + Aeneas invocation becomes almost instantaneous, as opposed to a few seconds per file when constantly checking invariants. In addition to those 35 kLoC, we have 13kLoC of comments; our implementation is truly written with great care.

14.2 Borrow-Checking

14.2.1 Evaluation

We now evaluate how Aeneas performs as a borrow-checker. In the previous chapters, we focused on establishing simulations to demonstrate the soundness of LLBC$^\#$ with respect to PL, a low-level, heap-manipulating semantics. Our theorems establish that, for any execution in LLBC$^\#$, there exists a related execution in LLBC, and hence in PL by composing simulations. As PL is deterministic, it gives us that all PL states in relation with the initial LLBC$^\#$ state safely execute. One key question remains however: seeing LLBC$^\#$ as a borrow-checker for LLBC, are we able to construct LLBC$^\#$ derivations in order to apply our theoretical results? To this end, we ran Aeneas on a selection of programs.

Our test suite consists of two categories of programs. First, we implement a collection of micro-tests totalling 2000 LoCs, without blanks and comments, spanning various Rust patterns. In particular, we implemented 22 micro-tests (totalling 300 LoCs) to test patterns based on loops, such as incrementing counters, updating values in a vector (for instance, to reinitialize it), summing the elements in an array or a slice, reversing a list in place, or retrieving a shared or mutable borrow to the n-th element of a list. Second, we evaluate our approach on a hash-table, an AVL tree and a b-ϵ tree, which make heavy use of loops and consist of 1068 LoCs in total.

The LLBC$^\#$ interpreter successfully borrow-checks all the examples, requiring less than 1s for the whole test suite. Interestingly, we note that we actually manage to borrow-check (and translate) functions which are not accepted by Rust's current borrow checker.

14.2.2 A Limitation of Rust's Borrow-Checker

In the process of testing the b-ϵ tree we mentioned above, we bumped into a limitation of the current Rust borrow checking algorithm; interestingly, this limitation does not appear with Aeneas, owing to our semantic checking of borrows.

The get_suffix_at_x function, below, looks for an element in a list, and returns a mutable borrow to the suffix of this list starting at this element, for in-place modifications in a C-style fashion. The current Rust borrow-checker is too coarse to notice that the Cons branch is valid. More specifically, it considers that the reborrows performed through hd and tl should last until the end of the lexical scope, that is, until the end of the Cons branch. We inserted the error message printed by Rust in the comments.

Instead, one may notice that is it possible to end those borrows earlier, after evaluating the conditional, in order to retrieve full ownership of the value borrowed by ls in the first branch of the if, and make the example borrow-check. The ongoing replacement of the borrow-checker in Rust, named Polonius, implements a more refined lifetime analysis, and accepts this program.

More interestingly, our semantic approach of borrows makes this program borrow-check without issues; since our discipline is based on symbolic execution and a semantic approach to loans, we accept the example without troubles, and are resilient to further syntactic tweaks of the program.

```
fn get_suffix_at_x<'a>(ls: &'a mut List<u32>, x: u32) -> &'a mut List<u32> {
  match ls {
    Nil => { ls }
    Cons(hd, tl) => { // error: first mutable borrow occurs here
      if *hd == x { ls // second mutable borrow occurs here
      } else { get_suffix_at_x(tl, x) } } } }
```

14.2.3 Precise Reborrows

Due to its precise management of borrows, we explained above that Aeneas can handle programs which are supported by Polonius, and not by the current implementation of the borrow checker. Because of the way we handle reborrows, there are actually cases of programs deemed invalid even by Polonius, but supported by Aeneas.

For instance, in the example below, we first create a shared borrow that we store in pp at line 2, then a reborrow of a subvalue borrowed by pp that we store in px at line 5. Upon evaluating the assignment at line 9, px simply maps to a shared borrow of the first component of p (environment at lines 6-8). Importantly, even though px reborrows part of pp, there are no links between px and pp: our semantics does not track the

hierarchy between borrows and their subsequent reborrows. In other words, line 5 is equivalent to `let px = &p.0`, where we borrow directly from p without resorting to pp. This implies that, upon ending the borrow ℓ_p stored in pp at line 9, we do not need to end ℓ_x stored in px, which in turn allows us to legally evaluate the assertion at line 13.

```
let mut p = (0, 1);
let pp = &p;
// p  ↦ loan^s ℓ_p (0, 1)
// pp ↦ borrow^s ℓ_p
let px = &(*pp.0);
// p  ↦ loan^s ℓ_p (loan^s ℓ_x 0, 1)
// pp ↦ borrow^s ℓ_p
// px ↦ borrow^s ℓ_x
p.1 = 2;
// p  ↦ (loan^s ℓ_x 0, 2)
// pp ↦ ⊥
// px ↦ borrow^s ℓ_x
assert!(*px == 1);
```

When we attempt to borrow check this program (with Polonius or the current implementation of the borrow checker), the Rust borrow checker considers that px reborrows pp, and thus needs to end before pp ends. It consequently fails with the following error message:

```
cannot assign to p.1 because it is borrowed
   |
 2 |     let pp = &p;
   |              -- borrow of p.1 occurs here
 5 |     let px = &(pp.0);
 9 |     p.1 = 2;
   |     ^^^^^^^ assignment to borrowed p.1 occurs here
13 |     assert!(*px == 1);
   |             --- borrow later used here
```

The code snippet below illustrates a similar example with mutable borrows. We create a borrow px1 of (the value of) x at line 2, then reborrow this value through px2 at line 3. At line 8, we then update px1 to borrow y. At this point, px2 still borrows x. The important point to notice is that upon performing this update, we remember the old value of px1 in an anonymous variable to not lose information about the borrow graph (environment at lines 9-13). Similarly to the previous example with shared borrows, the resulting environment doesn't track the fact that px2 was created by reborrowing the value initially borrowed by px1: there are no links between those two variables.

Consequently, upon ending borrow ℓ_y (stored in px1) to access y at line 16, we don't need to end ℓ_2 (stored in px2). This in return allows us to legally dereference px2 at line 22.

```
let mut x = 0;
let mut px1 = &mut x;
let px2 = &mut (*px1); // Reborrow: px2 now borrows (the value of) x
// x   ↦ loan^m ℓ1
// px1 ↦ borrow^m ℓ1 (loan^m ℓ2)
// px2 ↦ borrow^m ℓ2 0
let mut y = 1;
px1 = &mut y; // Update px1 to borrow y instead of x
// x   ↦ loan^m ℓ1
// _   ↦ borrow^m ℓ1 (loan^m ℓ2)
// px2 ↦ borrow^m ℓ2 0
// y   ↦ loan^m ℓy
// px1 ↦ borrow^m ℓy 1
assert!(*px1 == 1);
assert!(*px2 == 0);
assert!(y == 1); // End the borrow of y through px1 (shouldn't impact px2!)
// x   ↦ loan^m ℓ1
// _   ↦ borrow^m ℓ1 (loan^m ℓ2)
// px2 ↦ borrow^m ℓ2 0
// y   ↦ 1
// px1 ↦ ⊥
assert!(*px2 == 0); // Considered invalid by rustc, but accepted by Aeneas
```

When attempting to borrow check this code snippet, we get the following error:

```
cannot use y because it was mutably borrowed
   |
 8 |     px1 = &mut y; // Update px1 to borrow y instead of x
   |           ------ borrow of y occurs here
...
16 |     assert!(y == 1); // End the borrow of y through px1 (shouldn't impact px2!)
   |             ^ use of borrowed y
22 |     assert!(*px2 == 0); // Considered invalid by rustc, but accepted by Aeneas
   |             ---- borrow later used here
```

The two examples above exemplify cases where both the Rust borrow checker and Polonius deem a program as invalid, while Aeneas accepts it. We do not claim that this is a strong limitation of the Rust borrow checker: these use cases seem quite anecdotal and are probably useless in practice. However, we believe the ability of Aeneas to precisely capture the behavior of such use cases supports our claim that our semantics really captures the essence of the borrow mechanism.

14.3 Backends

Aeneas' pure translation provides the benefit of abstracting away memory reasonings for a large class of programs; this fact alone leads to an important productivity gain for the proof engineer. Yet, it doesn't remove the need to perform complex functional correctness reasonings, making the backends a crucial element of the proof experience. As a consequence, we invested an important amount of work in experimenting with potential backends and in implementing features and tools which are essential for the verification effort. We now review the backends currently supported by Aeneas.

We support extraction to F*, Coq, HOL4 and Lean. The F* backend was our initial backend, and allowed us to carry out preliminary case studies (Section 14.4); we then added backends for HOL4, Coq, and later Lean. HOL4 allowed us to experiment with a powerful rewriting engine, which has been a key feature of the LCF provers since their early days, and which offered a stark contrast with F*'s SMT. We also leveraged HOL4's powerful meta-programming capabilities to implement custom elaboration and automation. As a HOL4 proof is essentially an SML file, we can indeed leverage the full expressivity of SML to write custom automation and, in some way, elaboration. This allowed us to experiment with several important features, namely: a custom encoding of recursive function which leverages Knaster-Tarski's fixed-point theorem to encode partial functions (Section 14.3.1); automatic, forward instantiation of lemmas to design custom automation, in particular for the purpose of reasoning about arithmetic proof obligations (Section 14.3.2); a progress tactic to automatically lookup lemmas written with pre- and post-conditions so as to write proofs in a Hoare-logic style (Section 14.3.3). The HOL4 standard library for Aeneas is currently made of more than 4.7k LoCs; we however decided to switch to a different prover as we encountered several shortcoming: proofs can be hard to structure in HOL4, and the lack of support for dependent types was an important limitation in some situations.

We then experimented with Coq, and quickly moved to Lean; the main reason for this choice was Lean's meta-programming capabilities which allowed us to re-implement better versions of the features we had implemented in HOL4. We currently focus on the Lean backend which is as of today the most complete; the standard library we have implemented for Aeneas currently comprises more than 7.2k LoCs. We now review the different features we implemented.

14.3.1 Recursive, Partial Functions

As idiomatic Rust programs contain recursive functions and, most importantly, loops, Aeneas must provide good support for (mutually) recursive functions. A key issue stems

from the fact that, for soundness purposes, theorem provers only support terminating functions. In particular, HOL4 and Coq only support *structurally* terminating functions, which is very restrictive in practice. Consider for instance a function which iterates over the elements of an array starting at index 0; if this is a very common pattern in Rust, this function is not structurally terminating and can thus not be directly encoded in those provers. F⋆ and Lean on their side have native support for non-structurally terminating functions. In the case of F⋆, the user can supply a measure of termination with a `decreases` clause; this works smoothly combined with the possibility of writing preconditions. In the case of Lean, the user can annotate definitions with termination_by and decreasing_by clauses, to provide a measure of termination and a proof that the measure indeed terminates at each recursive call, respectively; under the hood, Lean encodes such functions with a special fixed-point operator. However, those don't work well in the context of Aeneas, as the pure model is automatically generated, making it cumbersome to add such annotations. A solution would be to allow supplying hand-written annotations in the original Rust code, which would then be inserted in the generated model, but as of today we do not have support for those annotations, while designing and implementing a specification language requires an important amount of work. We experimented with permitting the user to provide, e.g., `decreases` clauses, in a separate file. While this allowed to make some examples work (see Section 14.4), it did not scale very well. Worse, it is possible in Rust to write partial functions which may indefinitely loop for some inputs, and we need to support those.

One last important observation is that the proof of termination required from the user when defining a function is often redundant once the user writes and prove a correctness theorem. Let us consider the example below, written in Lean syntax, and which models a function summing the elements over a list. This function is very similar to the model which would be generated by Aeneas for a loop iterating over the elements of an array. Importantly, making this function type-check in Lean requires annotating it with termination_by and decreasing_by clauses.

```
def sum (l : List Int) (i : Nat) : Int :=
  if i > l.length then 0
  else l.index i + sum l (i + 1)
termination_by l.length - i -- termination measure
decreasing_by ... -- proof that the measure decreases (omitted)
```

The proof of termination of this function is simple: the quantity l.length - i (rounded to 0 if i is greater than l.length) decrements at each recursive call. One may want to prove the following property over sum, which states that using foldl to sum the elements of the list leads to the same result:

```
theorem sum_eq (l : List Int) :
  sum l 0 = l.foldl (fun s x => s + x) 0 :=
  ... -- proof omitted
```

The proof requires generalizing the theorem for any index i. More importantly, it requires performing an induction on l.length − i. We note that when performing this induction to prove sum_eq, we actually exhibit the termination measure of the sum function itself. As a consequence, requiring a proof of termination for sum is not only cumbersome, it is actually redundant; imposing it to the user is a bad design choice.

For those reasons, we decided to resort to a custom elaboration which allows defining partial functions by means of a custom fixed-point operator; this allows us to defer the proof of termination from definition time to proof of correctness time; we actually state and prove theorems of *total* correctness. The technique we present here is adapted from Bertot et al. [416], which uses Knaster-Tarski's fixed point theorem to encode partial functions in Coq. Their approach requires proving that function bodies are monotonous and continuous, which is tedious to do by hand. Our contribution is to make this process automatic by implementing a custom elaboration and remarking that those proofs are essentially guided by the syntax. We now present how we implemented this in Lean; the implementation in HOL4 was slightly different and suffered from several limitations as we could not leverage dependent types.

A Simple Example

We will use the following definition as a running example. The function id is the identity if its input is positive, and diverges otherwise; because it is not structurally terminating, it doesn't type-check in Lean.

```
def id (x : Int) : Result Int :=
  if x = 0 then ok 0
  else do
    let x ← id (x - 1)
    ok (x + 1)
```

A common technique to turn a non-terminating function into a terminating one is to use a notion of fuel [417]: the function receives as additional parameter a natural number which decreases at each recursive call. If the fuel reaches 0, the function evaluates to an error value; as a consequence, one must supply a fuel which is big enough upon calling the function. By construction, this makes the function structurally terminating in the fuel, and thus accepted by the theorem prover. We apply this technique to the id function, leading to id' below:

```
def id' (n : Nat) (x : Int) : Result Int :=
  match n with
  | 0 => div -- No more fuel!
  | succ n' => -- There remains fuel: continue normally
    if x = 0 then ok 0
    else do
      let x ← id' n' (x - 1) -- The recursive call uses the decreased fuel
      ok (x + 1)
```

If this technique allows us to make Lean accept the definition, it has the important drawback of forcing us to manipulate a cumbersome fuel parameter in the proofs. But the situation is not lost, especially if we notice that what we really want is not id', but the limit of id' when the fuel goes to infinity. Fortunately, we can easily do so by using Hilbert's epsilon operator.

The epsilon operator takes as input a predicate P over a non-empty type a. It evaluates to an element of a which satisfies P if such an element exists, and to an arbitrary element of a otherwise. Equipped with this operator, we can easily define the least upper bound of a predicate P operating over natural numbers; we just ask for a number satisfying P, and such that no smaller number satisfies P:

```
def least (P : Nat → Prop) : Nat :=
  epsilon (P n ∧ (∀ m, m < n → ¬ P m))
```

By using this least operator, we can now re-define id in terms of id', so that it uses the smallest fuel (if it exists) such that id' doesn't evaluate to div:

```
def id (x : Int) : Result Int :=
  id' (least (λ n => id' n x ≠ div)) x
```

We managed to remove the fuel parameter from id, but its definition has now become very involved: it is unclear how to reason about this auxiliarly id' definition in the presence of this least predicate. Our final trick is that we can actually prove the following unfolding lemma:

```
theorem id_unfold (x : Int) :
  id x =
    if x = 0 then ok 0
    else do
      let x ← id (x - 1)
      ok (x + 1)
```

This unfolding lemma states that if id is not *definitionally* equal to its (expected) unfolding, it is actually *provably* equal. If we save this unfolding lemma in Lean so

that it gets applied when one needs to unfold id, the user will not even need to be aware of the details of the encoding of id; the function will appear to have the expected definition, except it is actually partial.

We might then want to prove the following property about id, namely that it is equal to the identity if i is positive. The proof is performed by induction on |i|; importantly, by proving that id_eq evaluates to a result which is different from div, we actually prove at the same time that the function terminates for all positive inputs:

```
theorem id_is_id (x : Int) (h : x ≥ 0) : id x = ok x := ... -- proof omitted
```

By stating this theorem in a slightly different manner, we could also prove a *partial* correctness property:

```
theorem id_is_id (x : Int) (h : x ≥ 0) : ∀ y, id x = ok y → y = x := ...
```

A General Fixed-Point

We now generalize what we have shown above. A crucial element which makes our approach applicable is that all the functions generated by Aeneas live in an error monad; i.e., they use Result in their output type. We show the definition of the type Result below:

```
inductive Result (a : Type) :=
  | ok (v: a) | fail | div
```

The type Result has three cases. The first two cases, ok for successful computations, and fail for executions which fail (i.e., panic), do not deserve scrutinee. The important case is div, which we use for diverging executions. We can easily implement a monad for Result, which propagates the fail and div cases (omitted here). Because we want to factor definitions out, we define once and for all a fixed-point operator fix_fuel below:

```
def fix_fuel {a: Type} {b : a → Type} (n : Nat)
  (f : ((x:a) → Result (b x)) → (x:a) → Result (b x)) (x : a) : Result (b x) :=
  match n with
  | 0 => div
  | n + 1 => f (fix_fuel n f) x
```

The operator fix_fuel takes as inputs a fuel n and a *functional* f. The parameter f stands for the body of the function we wish to encode, and receives as input a continuation that it uses for the recursive calls; as such it is not itself recursive. The fix_fuel operator then recursively calls f until it runs out of fuel. Importantly, we defined the output type of f (b) to depend on its input (a) in order to support polymorphic

functions; because the use of this fixed-point operator requires us to currify functions, we need to pack the types with the input arguments, meaning the output type can be polymorphic only if it is dependent in the input[1]. We can finally define a fix operator, which takes the limit of fix_fuel when the fuel goes to infinity:

```
def fix (f : ((x:a) → Result (b x)) → (x:a) → Result (b x)) (x : a) :
  Result (b x) :=
  fix_fuel (least (λ n => fix_fuel n f x ≠ div)) f x
```

Equipped with fix, we can easily define the id function as follows:

```
def id_body (k : Int → Result Int) (x : Int) : Result Int :=
  if x = 0 then ok 0
     else do
      let x ← k (x - 1)
      ok (x + 1)

def id (x : Int) : Result Int := fix id_body
```

There now remains to prove the unfolding equation. Here, we use Knaster-Tarski's fixed point theorem: if f is monotonous and continuous, we get the fixed-point equation: fix f = f (fix f); this trivially gives us the unfolding theorem we want.

More precisely, we define a validity predicate over function bodies as follows.

```
def result_le {a : Type} (x1 x2 : Result a) : Prop :=
    match x1 with
    | div => True
    | fail _ => x2 = x1
    | ok _ => x2 = x1

def arrow_le {a} (k1 k2 : (x:a) → Result (b x)) : Prop :=
  ∀ x, result_rel (k1 x) (k2 x)

def is_mono {a} (f : ((x:a) → Result (b x)) → (x:a) → Result (b x)) : Prop :=
  ∀ k1 k2, arrow_le k1 k2 → arrow_le (f k1) (f k2)

def is_cont {a} (f : ((x:a) → Result (b x)) → (x:a) → Result (b x)) : Prop :=
  ∀ x, (∀ n, f (fix_fuel n f) x = div) → f (fix f) x = div

def is_valid {a} f := is_mono f ∧ is_cont f
```

[1]The real definition of fix_fuel is actually slightly different to make the encoding of polymorphic functions and mutually recursive definitions more straightforward; we omit those details here and refer the interested reader to the implementation.

The predicate result_le states that two values of type Result are in relation if they are equal, or the first one is equal to div; it defines a partial order over elements of type Result. The predicate arrow_rel lifts this partial order to functions. Finally, the monotonicity predicate is_mono is satisfied for all functions which are increasing for this order. We did not define the continuity predicate is_cont the standard way, but rather in a manner more amenable to the proofs we needed to perform; more specifically, it allows us to take the limit of fix_fuel n f in f (fix_fuel n f) x, which is the difficult part in the proof of the fixed-point equation. We finally state that a function body is valid if it is both monotonous and continuous (predicate is_valid), and prove the target fixed-point theorem (the proof requires the use of the excluded-middle):

```
theorem fix_eq (f : ((x:a) → Result (b x)) → (x:a) → Result (b x))
  (Hvalid : is_valid f) :
  fix f = f (fix f)
```

Proving Monotonicity and Continuity

As stated above, the unfolding equation for id immediately derives from this fixed-point theorem. However, applying it requires proving that the definition id_body is valid, i.e., is monotonous and continuous. We finally observe that the property of being monotonous and continuous is mostly *syntactic*; intuitively, it should be satisfied as long as we do not "open" the error monad instead of using the monadic binding we defined, something which the functions output by Aeneas never do. As a consequence, we can make such proofs straightforward by stating and proving monotonicity and continuity theorems for the *combinators* of our language (e.g., bind). In order to pave the way for those theorems, we now define monotonicity and continuity predicates for *expressions* rather than functions.

```
def exp_is_mono {a b c} (e : ((x:a) → Result (b x)) → Result c) : Prop :=
  ∀ k1 k2, karrow_rel k1 k2 → result_rel (e k1) (e k2)

def exp_is_cont {a b c} (k : ((x:a) → Result (b x)) → (x:a) → Result (b x))
  (e : ((x:a) → Result (b x)) → Result c) : Prop :=
  (Hc : ∀ n, e (fix_fuel n k) = .div) →
  e (fix k) = .div

def exp_is_valid {a b c} (k : ((x:a) → Result (b x)) → (x:a) → Result (b x))
  (e : ((x:a) → Result (b x)) → Result c) : Prop :=
  exp_is_mono e ∧
  (is_mono k → exp_is_cont k e)
```

The predicate exp_is_mono is the same as is_mono except that we slightly tweak the type of the input (e). The predicate exp_is_cont is more interesting. As it must operate over a sub-expression of a function body, it receives two inputs: the function k, which stands for the current function body, and the function e, which stands for the sub-expression currently under scrutiny and which itself receives as input a continuation for the recursive calls. The predicate exp_is_valid is also slightly subtle, in that it requires the expression under scrutiny to be continuous only if k is monotonous.

Those auxiliary predicates allow us to prove predicates which operate over the primitive combinators of our language. For instance, we can easily prove a theorem about the monadic bind:

```
theorem expr_is_valid_bind {a b c}
  {k : ((x:a) → Result (b x)) → (x:a) → Result (b x)}
  {g : ((x:a) → Result (b x)) → Result c}
  {h : c → ((x:a) → Result (b x)) → Result d}
  (Hgvalid : is_valid_p k g)
  (Hhvalid : ∀ y, is_valid_p k (h y)) :
  is_valid_p k (λ k => do let y ← g k; h y k)
```

This theorem simply states that a bind expression is valid (i.e., monotonous and continuous) if its sub-expressions are valid. Similarly, an `if then else` expression is valid if its branches are valid, etc. Proving that a function body is valid then simply requires recursing over its sub-expressions to properly combine lemmas like expr_is_valid_bind.

Custom Elaboration

The last step is to automate the elaboration and the proofs of validity. To do so, we defined a keyword divergent which, if present, triggers a custom elaboration. This elaboration triggers after parsing and type-checking, but before the definition is sent to the Lean kernel; doing so is relatively straightforward because, at the exception of its kernel, Lean is implemented in Lean itself, meaning we can easily hook ourselves into the compiler. We can then modify the definition to rewrite it in terms of the fix operator we defined above, prove on the fly the validity theorem we need, and apply the fixed-point equation to get and save the unfolding theorem. Thanks to this custom elaboration, all the steps we described above are automatically performed if we simply write the definition below:

```
divergent def id (x : Int) : Result Int :=
  if x = 0 then ok 0
```

```
  else do
    let x ← id (x - 1)
    ok (x + 1)
```

Limitations and Future Work

Our elaborator currently supports a wide range of functions, including n-ary, mutually recursive, and non-uniform polymorphic functions. The version of Knaster-Tarski's fixed-point theorem we use above however imposes restrictions on the class of functions we can support; in particular we need their body to be continuous. We could lift this limitation and only require their bodies to be monotonous if we used a different version, which requires operating over a complete lattice; the Mathlib library actually has a fixed-point theorem for complete lattices. For instance, recent work studying the problem of encoding partial functions in Coq uses a similar encoding, which doesn't require the continuity property [417]. Updating the development to use a different fixed-point would only require minimal modifications. More specifically, as the elaboration and the proof of the validity theorems for the function bodies is guided by the syntax, the elaboration itself would not need to change; we would only need to update the proof of the primitive validity theorems about the language combinators.

We also currently have limited support for higher-order functions. For instance, one might want to define the following identity function over a type Tree. Importantly, tree_id calls itself recursively through the map function (which is the obvious monadic map function):

```
inductive Tree (a : Type) :=
| leaf (x : a)
| node (tl : List (Tree a))

divergent def tree_id {a} (t : Tree a) : Result (Tree a) :=
  match t with
  | leaf x => ok (leaf x)
  | node tl => do
    let tl ← map tree_id tl
    ok (node tl)
```

Proving that tree_id satisfies the validity property in the presence of this recursive call is non-trivial, and requires a generic validity theorem for map. As of today, we solved the issue by proving validity lemmas by hand over functions such as map, and leveraging Lean's extensible state and attribute system to store them, so that our custom elaboration above can use them. As a consequence, this only works if the map

function above is hand-written in our standard library, and not generated by Aeneas; we intend to make this mechanism more general in the future. More specifically, we proved the following (slightly simplified) lemma for the map function:

```
@[divergent]
theorem map_is_valid {a b c d}
  {f : (a → Result (b a)) → d → Result c}
  (k : (a → Result (b a)) → a → Result (b a))
  (HfValid : ∀ x1, exp_is_valid k (fun kk1 => f kk1 x1))
  (ls : List d) :
  is_valid_p k (λ k => map (f k) ls)
```

By annotating map_is_valid with the divergent attribute, which we implemented for this purpose, we save this lemma in a database. During the custom elaboration and when proving a validity property, whenever we reach an application we check if there exists an applicable lemma from this database; for instance, when elaborating tree_id above, we lookup the lemma map_is_valid when reaching the sub-expression map tree_id tl. We then recursively prove the premises of this lemma (here, HfValid), which allows us to accept the definition tree_id above.

14.3.2 Forward Instantiations and scalar_tac

We now describe some preliminary work aiming at building basic blocks to implement custom automation. In particular, we describe a procedure which automatically instantiates and introduces lemmas by using the variables and assumptions available in the context. Automatically instantiating lemmas is not novel; for instance, the auto tactic in Coq and Isabelle/HOL performs similar instantiations, and we actually borrowed ideas from the aesop tactic [418] for our implementation. The aim of this section is to report how we leveraged Lean's features to implement a flexible system, and how we used it to implement the scalar_tac tactic for linear arithmetic proofs.

When implementing automated procedures, one often needs to automatically instantiate lemmas. A convenient mechanism for this purpose is given by SMT patterns [419]. For instance, in F⋆, one can annotate a lemma with a pattern: the lemma gets automatically instantiated whenever Z3 finds a matching term in the context. Implementing a similar mechanism is straightforward in Lean, and was actually done by aesop[2]. In our case, we leverage the fact that Lean has an extensible state and a system of attributes to define a saturate attribute, which receives a name and a pattern. The pattern controls

[2]We are currently not using aesop's implementation because of minor performance issues, but intend to use it in the future.

how the lemma gets applied, while the name allows us to group patterns together. For instance, we can annotate the theorem below, which states that the length of the concatenation of two lists is the sum of their lengths:

```
@[saturate (set := ListSet) (pattern := l0 ++ l1)]
theorem length_append {a} (l0 l1 : List) :
  length (l0 ++ l1) = length l0 + length l1 := ...
```

The theorem gets saved in a map from (rule set) names to discrimination trees, for efficient matchings. In the present case, we save length_append in the discrimination tree for the set ListSet, by introducing a mapping from the expression l0 ++ l1 to length_append, where l0 and l1 are variables (which can thus match any term).

We can now use the saturate tactic[3] to automatically apply this lemma. For instance, let's assume the expression l0 ++ (l1 ++ l2) appears in the context. By using saturate [ListSet] we can apply all the lemmas for the rule set ListSet, introducing two assumptions in the context: length l0 (l1 ++ l2) = length l0 + length (l1 ++ l2), and length (l1 ++ l2) = length l1 + length l2. As Lean supports adding attributes *locally*, so that an attribute is only applied for the span of a module, we can mark lemmas with the local saturate ... attribute, so that they get automatically applied for the proofs in a single file, but not elsewhere; this kind of features is very useful to tweak the automation as we will see later.

We now turn to the scalar_tac tactic, which is a tactic we implemented to automatically discharge arithmetic goals. Arithmetic proof obligations are numerous when doing program verification; for instance, every addition of two machine integers requires checking for overflows, while every array access requires checking that the index is in bounds. As a consequence, having good solvers for linear arithmetic goals is paramount, and we observed that the ability of SMT solvers to efficiently solve linear arithmetic goals is one of the reasons why they are so powerful when doing program verification. In the context of implementing custom automation for interactive theorem provers like Lean, we posit we can design reasonably efficient decision procedures with simple techniques. Under the hood, scalar_tac massages the goal by applying a set of well-chosen simplification lemmas, uses the saturate tactic to do some forward reasoning, then calls omega, the Lean tactic to reason about linear-arithmetic goals. The scalar_tac tactic is in effect simple, yet quite powerful, especially because we can extend the set of lemmas that automatically get instantiated through saturate. We defined the scalar_tac attribute for this purpose, which is just some syntactic sugar. For instance, we annotated the lemma below, which allows us to automatically introduce scalar bounds in the context, so that scalar_tac can reason about machine scalars:

[3]The name is taken from aesop.

```
@[scalar_tac x]
theorem Scalar.bounds {ty : ScalarTy} (x : Scalar ty) :
  Scalar.min ty ≤ x.val ∧ x.val ≤ Scalar.max ty := ...
```

We also annotated a set of lemmas to reason about lists, like length_append above. More importantly, we can leverage the local attributes to temporarily extend the capabilities of scalar_tac. For instance, the notorious instability of Z3 when reasoning about non-linear arithmetic regularly led to breakages in the HACL⋆ library. In particular, in HACL⋆ we often need to prove that the product of two positive numbers is positive; such proof obligation invariably fails in CI at some point, eventually forcing us to deactivate the non-linear arithmetic heuristics in the broken proofs while manually inserting calls to the proper lemma. This proved to be extremely cumbersome in practice, and particularly frustrating as such proof obligations are extremely simple. In the present case, we can easily provide the required lemma to scalar_tac:

```
@[local scalar_tac x * y]
theorem pos_mul_pos_is_pos (x y : Int) :
  (x < 0) ∨ (y < 0) ∨ (0 ≤ x * y)
```

The lemma is written in a disjunctive form to make it always applicable for pattern $x * y$, leveraging the fact that omega can reason about disjunctions; we could write it as an implication $0 \leq x \to 0 \leq y \to 0 \leq x * y$, but its application would become brittle as it would require the exact assumptions $0 \leq x$ and $0 \leq y$ to be in the context. Of course, we may not want to always use it as some other proofs might require more general reasonings; hence the use of the local keyword, which allows circumscribing its application to the proofs of the current module. The possibility of locally extending the sets of lemmas used by scalar_tac proved very useful in the use-cases we study below (Section 14.4).

14.3.3 Hoare-Logic Style Proofs and the progress Tactic

We reviewed some basic decision procedures in the previous section; let us now turn to the problem of actually verifying programs. A natural way of doing program verification is to reason in a Hoare-Logic style, by specifying functions as having pre- and post-conditions. In our case, we want the proof experience to feel like a debugging session, by which proving a theorem about a function requires interactively stepping through the function calls in its body and, for every call, proving its pre-condition while having the context available for inspection, before continuing with a context augmented with the proper post-conditions. We want to automate as much administrative work as

possible as well as the mundane proof obligations, so that the user can focus on the interesting parts of the proofs. For this purpose we implemented the progress tactic, which inspects the goal to identify the next function call to consider, automatically looks up and instantiates the relevant lemma, attempts to prove its pre-condition, and finally updates the context. Implementing such a tactic to automatically lookup theorems is not novel [420]. However, the progress tactic we implemented for Aeneas has specific features; we now illustrate how it helps in the proofs.

For instance, let's assume we are given the following goal:

```
1  x : U32
2  y : U32
3  h : ↑x + ↑y ≤ U32.max
4  ----------------------
5  (do -- the function body
6     let z ← x + y
7     let v ← f z
8     ...) -- more function calls (omitted)
9    = ... -- the post-condition (omitted)
```

The context contains two variables x and y of type U32 (lines 1-2), and the assumption that their addition doesn't overflow (3). This addition operates over ↑x and ↑y, which are x and y seen as *unbounded* mathematical integers; this means in particular that ↑x + ↑y can not overflow. The expression below the horizontal line (lines 5-9) is the goal we want to prove. Importantly, proving this goal requires reasoning about the scalar addition x + y (line 6), which operates over machine integers and *can* overflow. In order to progress in the proof we can use the following lemma, which states that if the addition of x and y, seen as unbounded mathematical integers, doesn't exceed U32.max, then the addition succeeds with the expected result:

```
theorem U32.add_spec {x y : U32} (h : ↑x + ↑y ≤ U32.max) :
  ∃ z, x + y = ok z ∧ ↑z = ↑x + ↑y
```

We can instantiate this lemma for x and y, prove its precondition, decompose it and simplify the goal: x + y gets substituted with ok z, the expression let z ← ok z simplifies, etc. This requires writing the following proof script:

```
1  have ⟨ x1, hEq, hPost ⟩ := @U32.add_spec x x (by scalar_tac)
2  simp [hEq]
```

We get the following goal:

```
x : U32
y : U32
h : ↑x + ↑y ≤ U32.max
h1 : x + y = ok z
h2 : ↑z = ↑x + ↑y ≤ U32.max
---------------------------
(do -- the function body
   let v ← f z
   ...) -- more function calls (omitted)
  = ... -- the post-condition (omitted)
```

Doing those operations manually for every function call in the proof quickly becomes cumbersome. Instead, we implemented the progress tactic, which inspects the goal, notices that the next function call to consider is x + y, automatically looks up the lemma U32.add_spec and instantiates it, proves its precondition, and updates the goal by introducing the variable z : U32 together with the assumption h : ↑z = ↑x + ↑y in the context; dedicated syntax allows the user to choose the name of the variables and assumptions introduced by the tactic.

Of course, whenever the user proves a correctness theorem, they naturally want to allow progress to automatically look it up in the subsequent proofs. We allow doing this by annotating theorems with the progress attribute. In fact, this is the way we handle theorem U32.add_spec shown above to progress: by annotating it with the progress attribute, we insert a mapping from expression x + y (where x and y are variables) to theorem U32.add_spec in the database of theorems.

The progress tactic can easily handle several patterns commonly found in the proofs. When proving a theorem about a recursive function, progress is aware of the theorem we are currently proving; this allows it to recursively use the *theorem* to handle recursive *function calls* appearing in the goal. This works very well in combination with Lean's termination_by and decreasing_by clauses which allow independently handling the problem of termination, instead of preemptively generalizing the theorem and applying the proper induction principle upon starting the proof.

The progress tactic also uses a heuristics which is simple, yet quite efficient in practice, to automatically instantiate variables. For instance, let's suppose we implement a binary tree in Rust and want to prove that the insert function is correct; in particular, we want to interpret the binary tree as a set. There are several ways of writing the specification of such function; we show a possibility below, which is written mostly for the purpose of illustration:

```
theorem insert_spec {α : Type} (ordInst: Ord α)
  [ordSpec : OrdSpec ord] -- specification for the order
```

```
3   (x : α) (t : BTree α) (s : Set α) :
4   inv s → -- an invariant about s
5   isSet t s → -- t can be interpreted as the set s
6   ∃ t', insert ordInst x t = ok t' ∧ -- the call succeeds
7   inv s ∧
8   isSet t (s ∪ {x}) := ...
```

The theorem above states that, if s satisfies some invariant (which, say, gives a bound on its size), and if the binary tree t can be interpreted as a set s through the predicate isSet, then inserting x in t successfully evaluates to a new tree t', which satisfies the invariant inv and can be interpreted as the set s ∪ {x}. After proving this theorem, we might want progress to use it in a subsequent proof, to reason about a call to insert. Importantly, when applying this theorem, progress will use the term insert x t to infer how to instantiate the variables x and t. However, figuring out how to properly instantiate s is more difficult. A simple solution is to instantiate s by matching the pre-conditions (here, inv s and isSet t s) against assumptions in the context; for instance, if it finds assumption inv s0, it will instantiate s with s0. However, this can lead to spurious instantiations: for instance, there might be two assumptions h0 : inv s0 and h1 : inv s1, about different sets; we need to pick the proper one. Because of this, we use a slightly different heuristics: we instantiate a variable by matching a pre-condition against the assumptions in the context only if there is a single matching assumption. In the present case, we would ignore the pre-condition inv s and move to the next one, isSet t s, which is more likely to unambiguously give us an instantiation for s; as a last resort we simply introduce a fresh meta-variable for s. In practice this simple heuristics provides convenient automation while avoiding spurious instantiations, and proved very useful for the use cases we describe later. In the future, we also intend to allow the user to have more precise control over the instantiations, in case the heuristics do not lead to the expected result.

As of today, the solvers used by progress to discharge pre-conditions are hardcoded; importantly, we use the scalar_tac tactic that we described previously. In practice, it already leads to an interesting proof experience; in the future, we intend to make this set of solvers customizable by the user.

14.4 Case Studies

We applied the Aeneas framework to the verification of several (moderate) case studies, each of which comprising a few hundreds lines of code.

14.4.1 Hash Table, Backend Comparison

The most interesting use case is a resizing hash table written in 222 lines of Rust code, and verified in F⋆, HOL4, and Lean. If this case study is small compared to, e.g., the Noise⋆ project, it already provides interesting insights about the proof experience we provide, as well as about the differences between the backends we support.

The resizing hash table is equipped with insert, get (immutable lookup), get_mut (mutable lookup) and remove. Each bucket is a linked list; insert replaces the existing binding, if any; resizing is automatic once a certain threshold is reached. The functional property we prove is that the hash table functionally behaves like a map. In this regard, the interesting function is insert. Interestingly, such a function leverages many low-level features of Rust: we have to be wary of arithmetic overflows when computing the new size, we mutably borrow the slots vectors for in-place updates, and move elements so as not to perform reallocations, etc.

By virtue of working with a (translated) pure program, we were able to focus on the *functional* behavior of the hash table and the important proof obligations, such as the absence of arithmetic overflows, rather than memory reasoning. This provided a stark contrast with our previous experience working with Low⋆. In practice, the proofs of the hash table did not cause any fundamental difficulty and were straightforward. We however noted interesting differences between the different backends.

F⋆ proved good at automating the mundane proof obligations, in particular those related to arithmetic reasonings; SMT automation was in particular very good at unfolding invariants, combining the relevant facts together, and finally finishing the proof by using its linear arithmetic solver. As Z3's heuristics for non-linear arithmetic proved too unstable, we resorted to writing manual proof scripts for the non-linear arithmetic proofs, which included in particular reasoning about the length computation for the resizing operation. Quite surprisingly, the context of Z3 grew quicker than expected. In order to keep the proof time low and the solver reactivity high we did the proof in two steps: we first proved that the implementation of the hash table refines a pure specification, then proved that the pure specification implements an ideal map. We posit that a better encoding of the proof obligations to Z3 and that a better control of the context by means of opaque and reveal instructions, as is usually done in Dafny, might mitigate this issue.

HOL4 compensates for the lack of SMT automation with its powerful simplification mechanism and the possibility of writing custom automation, allowing us to write a progress tactic which provided a smooth proof experience. As we noted, one of SMT automation's strength is its ability to automatically unfold definitions and combine

relevant facts. In the case of HOL4, we had to specify which definitions (for instance, invariants) to unfold by using the simplifier, before calling more powerful decision procedures (for arithmetic for instance); we note that manually unfolding an invariant is similar to using a reveal instruction in Dafny. Being an LCF prover, which in particular does not require building proof terms, the HOL4 tactics are also extremely fast; we expect the same from other provers like Isabelle/HOL. The proofs involving non-linear arithmetic were also easily done, this time leveraging the interaction with the proof assistant, in particular because it allows inspecting the context. One issue arose however from the fact that we could not control the context as much as needed, as for instance HOL4 does not allow naming assumptions. Rather than referring to assumptions and variables in a precise manner like in Coq, HOL4 enforces a style by which the user refers to assumptions by using patterns, which are supposedly more resilient to proof changes. This works well in many situations, but we sometimes wished for more control and as a result, similarly to F*, we did the proofs in two steps so as to keep the context small.

Lean also provides a powerful simplification mechanism, which is actually more flexible than the one provided by HOL4 when it comes to conditional rewriting, i.e., rewriting with a theorem which has a premise. As we noted in the previous sections, Lean's meta-programming facilities gave us a lot of flexibility. The progress tactic, that we co-developed while doing the use-cases, proved very powerful. We note once again that program verification requires solving a lot of mundane proof obligations, which are often discharged by simple decision procedures such as linear arithmetic solvers; in the present case, this allowed progress to automatically discharge many pre-conditions. We also crucially leveraged the extensibility of scalar_tac to reason about non-linear arithmetic and to automatically introduce facts deriving from the invariant. For instance, we introduced the following local rule, which allows scalar_tac to derive from the invariant, without unfolding it, the fact that the number of slots is strictly positive; such rules allow finely controlling the context by only introducing relevant facts:

```
@[local scalar_tac h]
theorem inv_scalar_facts {hm : HashMap α} (h : hm.inv) :
  0 < hm.slots.length ∧ -- the number of slots is > 0
  ... -- omitted facts
```

The hash table revealed some limitations of the current version of the Lean backend, in particular with regards to the lack of automation. A procedure to automatically instantiate universally quantified assumptions appearing in the context would have made the proofs smoother. We also often had to do case disjunctions by hand; for instance, over the fact that two keys may (or not) be equal. We note that HOL4 suffered from

the same issues, as probably does any other interactive proof assistant. By contrast SMT solvers can automatically instantiate universal quantifiers, though it requires careful control of the context, and are good at automatically doing case disjunctions and eventually back-tracking; as a consequence, such reasoning was (mostly) automatic in F*, though we sometimes had to introduce assertions of a specific shape to guide the instantation. We leave the implementation of such decision procedures in Lean, potentially specialized for some class of problems, as future work.

Finally, despite the fact that the Lean backend is in a preliminary state, in particular with regards to the level of automation which is as of today very primitive, we note that it already provides a rather smooth proof experience. More specifically, it provides a high level of control over the context, can automatically discharge many boring proof obligations, and allows resorting to more manual reasonings to finish the rest of the proofs. An interesting consequence is that we were able to comfortably do the proofs of the hash table in one step, while in HOL4 and F* we felt the need to introduce an intermediate refinement. This has made Lean our reference backend for Aeneas.

14.4.2 I/O and External Dependencies

We now discuss some features permitted by Aeneas' translation. Real-world applications rely on external libraries and often need to interact with the external world through I/O or sockets. We elegantly model interaction with the outside environment using opaque modules, and a state type that combines memory, IO and the outside world.

In other words, Aeneas allows reasoning about such applications by lifting the generated code into a combined-state + error monad, and relying on module signatures to model the interaction with external functions.

When we designate a module as opaque, Aeneas treats all the definitions coming from this module and reachable from the root module as opaque, and requests the user to provide models for those definitions. The user is then free to provide models for those declarations, or simply state assumptions by means of assumed lemmas. (In practice, we simply import a module that the user is required to write by hand, and optionally initialize this module with axioms.) This feature illustrates why a modular, type-directed translation like Aeneas' is important: the user doesn't need to reveal any information about the function's definition (or model its behavior using a specification language); rather, the user can work post-translation in the comfort of their favorite theorem prover. Moreover, while it is possible to add annotations to function signatures in a local crate, this possibility falls short when it comes to dealing with external

dependencies, over which the user has no control! Handling this oftentimes requires tediously wrapping such dependencies in properly annotated modules. In contrast, this work, and Charon and Aeneas, simply treat the external dependencies as opaque by looking up the types and functions that are needed in the (non-opaque modules of the) local crate, and generating corresponding declarations.

Finally, we give the possibility of using a state-error monad to introduce stateful reasoning when this is really needed, for instance when the code uses I/O functions. The state type, which models the external world, is also an opaque type for which the user is free to provide a model or write assumptions, in a fashion similar to opaque modules. In practice, this gives us a lightweight effect system.

Let us illustrate those possibilities with the following example.

We set out to serialize our earlier hash table to the disk. To account for this, we author serialize and deserialize functions in a separate opaque module outside of the scope of verification. We mark the module as opaque, meaning Aeneas generates the following declarations.

First, insert_on_disk, below, simply loads the map from the disk, inserts a new entry, and stores the updated table back on disk.

```
fn insert_on_disk(key: Key, value: u64) {
  let mut hm = deserialize();
  hm.insert(key, value);
  serialize(hm);
}
```

Aeneas requests the following declarations to model the disk state and the serialization and deserialization functions.

```
def State : Type := ... -- to be filled by the user
def deserialize (s : State) : Result (State × HashMap U64) := ...
def serialize (hm : HashMap U64) (s : State) : Result State := ...
```

Those definitions generate the following translation of insert_on_disk:

```
def insert_on_disk (key : Usize) (value : U64) (s : State) : Result State := do
  let (hm0, s1) ← deserialize s
  let hm1 ← insert U64 hm key value
  serialize hm s1
```

Given those declarations, the user is free to write models by filling their bodies, or simply assume them as axioms, together with properties to reason about them.

Chapter 15

Related Work

Software Verification. There exists plenty of work on software verification; as the field is extremely large, we refer the interested reader to surveys and SoKs [421–424], and focus below on work direcly related to Aeneas. We also refer the reader to Part II for work related to the verification of cryptographic implementations.

Electrolysis. Electrolysis [242] most resembles Aeneas in that it translates a (quite impressive) subset of Rust programs to pure models extracted in the Lean 3 [425] proof assistant. It relies on lenses to model mutable borrows, and as such comes with restrictions; for instance, functions may only return borrows to their first argument. Electrolysis does not come with a formal model, and thus does not make a case for semantic correctness. As such, it resembles a very pragmatic "transpiler" rather than a compiler; for instance, traits map to type classes, because they, at a high-level, work in a similar fashion.

Hacspec, hax. Hacspec [426] is an attempt at using a pure subset of Rust to write succinct, executable, formal specifications for cryptographic components. The goal is to allow developers, cryptographers and proof engineers to share a specification language which is non-ambiguous and readable by non-experts. Hacspec formally defines a subset of Rust, its semantics, and a type-system; as this subset is designed to be (mostly) pure it does not support creating (shared or mutable) borrows, but includes variable re-assignements and `for` loops iterating over integers. Specifications written in the hacspec subset can be compiled to pure specifications in F*, for the purpose of being used in proofs about low-level implementations written in, e.g., Low* [322]; hacpsec has been applied to the formalization of a number of cryptographic primitives [427] and of the TLS 1.3 protocol [428]. As the compiler attempts to generate code which is close to what the programmer writes, it consumes the Rust surface AST, while Aeneas operates on MIR; as a consequence, hacspec must redo part of rustc's work such as type inference.

The hacspec subset being essentially pure, the compilation to F⋆ specifications is also almost a one to one translation.

The hax framework [429] is hacspec's successor, and attempts both to be a specification language, in the spirit of hacspec, and a verification framework which works by generating pure models of Rust implementations, in the spirit of Electrolysis and Aeneas. The hax compiler resembles a pragmatic compiler very much like Electrolysis; as such its translation is trusted. Hax also consumes the THIR ("Typed High-level IR") output by rustc, while Aeneas consumes the lower-level MIR ("Mid-level IR"). These have different trade-offs; on our side we decided to operate on the MIR as it has fewer constructs, yielding a simpler formalism. Charon and hax actually share part of their frontend in order to factor out the burden of interacting with rustc; for instance, they share the same code to interact with rustc's trait solver. Hax allows doing proofs of functional correctness by extracting pure models to F⋆ and Coq, but also security proofs by generating models for ProVerif. It allows writing annotations directly in the Rust code, which are then extracted to pre- and post-conditions, assertions and lemma applications in the generated F⋆ code. Hax supports the use of shared borrows but, similarly to Electrolysis, its treatment of mutable borrows is not general and rather relies on the detection of some common, yet expressive patterns. For instance, it does not generally allow functions to return mutable borrows, but supports an interesting class of loops manipulating (mutable) iterators, that are not yet supported by Aeneas. The hax framework has been applied to several cryptographic applications, including in particular a verified implementation of ML-KEM [430].

RustHorn. RustHorn [243] operates on the Calculus of Ownership and Reference (COR), a Rust-like core calculus inspired by λ_{Rust}. Given a COR program, RustHorn uses prophecy variables to compute a first-order logical encoding that can then serve as a basis for reasoning upon the COR program. The encoding by means of prophecy variables served as a source of inspiration for the treatment of mutable borrows in LLBC. The RustHorn paper provides a proof of soundness and completeness of the encoding. Specifically, the authors establish a proof of bisimulation between COR, and the execution of a custom resolution procedure (dubbed SDLC) that mimics program execution when executed over the logical encoding.

The main difference with our work is that COR already takes for granted the ownership discipline of Rust, and materializes lifetimes within its program syntax: COR features instructions for creating or ending a lifetime, and asserting that a lifetime outlives another. One consequence is that COR cannot exhibit more behaviors than SDLC, hence why the bisimulation can be established.

In contrast, our low-level language, PL, does not feature lifetimes, and as such

exhibits more behaviors than LLBC. This means we can target the underlying execution model that Rust programs run on, rather than taking lifetimes as an immutable, granted analysis that we have to trust. A drawback is that we can only establish a forward simulation in the general case. Second, we do not commit to lifetimes, nor to any other particular implementation strategy of borrow-checking (e.g., Polonius [398]). Instead, we declaratively state what ownership-related operations may be performed in LLBC. It is then up to a particular borrow-checker implementation (e.g., Aeneas) to be proven sound with regards to this semantics. Notably, LLBC is not deterministic, and several executions may be valid simultaneously, e.g., by terminating borrows at different points (eagerly or lazily).

To summarize, RustHorn focuses on establishing the soundness of a logical encoding with regards to a model of the Rust semantics that assumes borrow-checking has been performed and can be trusted; here, we establish that LLBC is a correct model of execution for Rust programs, that LLBC$^{\#}$ is a valid borrow-checker for the LLBC semantics, and that LLBC$^{\#}$'s borrow-checking does indeed guarantee soundness of execution for LLBC programs.

RustBelt. In a similar fashion, RustBelt [393] is built atop λ_{Rust}, a core calculus that is already annotated with operations to create and end lifetimes. The operational semantics of λ_{Rust} itself is given by translation; the lifetime operations are assumed to be given by the Rust compiler. This impressive project focuses on a proof of semantic typing, with two chief goals: first, prove the lifetime-based type system sound with regards to the (lifetime-annotated) core language; second, use the semantic typing relation to establish that pieces of unsafe code do satisfy the type they export.

Our work differs from RustBelt in several ways. From the technical standpoint, RustBelt assumes lifetimes are given. Whether the lifetimes annotations are correct, and whether they lead to a successful execution is irrelevant – if the input program features improper lifetime annotations, this is outside of RustBelt's purview. In contrast, we attempt to determine what needs to be established from a semantic perspective in order to borrow-check a Rust program, and prove that successful borrow-checking entails safety of execution. From a goals standpoint, RustBelt attempts to understand the expected behavior of a Rust program that features unsafe blocks, using semantic typing. We do not consider unsafe code, but we intend to tackle this in future work.

RustHornBelt. The combination of RustHorn and RustBelt, RustHornBelt [397], aims to establish that the logical encoding of RustHorn is sound with regards to λ_{Rust}. RustHornBelt extends the methodology of RustBelt; at a very high-level, RustHornBelt proves the encoding of RustHorn, but with λ_{Rust} instead of COR, and with a machine-

checked proof instead of pen-and-paper. For the same reasons as above, we see this endeavor as addressing a different problem than ours: RustHornBelt is concerned with a logical encoding that leverages lifetimes as a central piece of information, rather than giving a functional, semantic account of the borrow-checking and execution of Rust programs.

SMT-Based Tools. Creusot [244] is a tool that follows the RustHorn approach to generate proof obligations encoded by using prophecy variables, and that can then be discharged to SMT. Their design chooses *automated*, *intrinsic* proofs: they introduce an annotation language for specifications, wrap a large part of the standard library in it, then rely on requires/ensures clauses and annotations to perform the proofs. This style emphasizes a logical encoding as opposed to an executable specification, one advantage being that they can easily require annotations for, e.g., loop invariants. They support a large subset of Rust code, allowing them for instance to reason about iterators [431]. It builds on RustHornBelt approach, and as such, benefits from its formalization and mechanization. As mentioned by Matsushita et al. [397], there remain some discrepancies, namely that RustHornBelt operates on a core language (instead of surface Rust), and that Creusot does not use the predicate transformers RustHornBelt relies on. Creusot has been used to verify several use cases such as: a SAT solver [432] and an SMT solver [433].

Verus' design choices [389] are very similar to Creusot's. They generate pure proof obligations that are sent to the Z3 SMT solver, though by using a simpler encoding technique than prophecy variables, making their use of mutable borrows slightly more restrictive; they however support the verification of unsafe code, which is not possible directly within Creusot. Their pure encoding goes directly to the SMT solver and is highly optimized to make the proof time smaller, where Creusot first generates a WhyML program. With regards to the semantics of Rust, Verus contains a pen-and-paper formalization, not about Rust itself, lifetimes, or borrow-checking, but rather about the soundness and termination of their approach to specifications relying on ghost permissions. As such, the proof for Verus answers a different question than our proof of soundness for LLBC$^{\#}$, namely, whether their design on top of existing Rust is sound. The proof remains of limited scope, only taking into account two possible lifetimes. Remarkably, Verus has already been applied to the verification of several realistic projects, including Kubernetes controllers [434], and a security module for confidential VMs [435].

Prusti's frontend [388] is very similar to Creusot's and Verus's, but their encoding of proof obligations is different. The tool uses Rust's type system to guide the application of rules in Viper [391], which means they rely on the Rust borrow checker for lifetime

inference but do not need to trust its results. Doing so, they automate the application of memory reasoning rules and thus avoid general-purpose memory proof search. Creusot and Verus, however, by virtue of their dedicated encodings that directly leverage lifetime information, appear to offer better verification performance than the Prusti frontend for the general-purpose Viper tool [244, §5.3]. Prusti also translates Rust programs into Viper's core logic; the soundness of verifying Rust code then depends on the soundness of Viper, and of the translation itself. To the best of our knowledge, no formal argument exists as to the soundness of the translation. Prusti has been used to verify that a WebAssembly sandboxing runtime correctly enforces memory and resource isolation [436].

Flux [437] uses liquid types [438] to reason about safe Rust programs. Unlike the work mentioned above and which also leverages the automation provided by SMT solvers, Flux does not attempt to verify *deep* functional correctness specifications, but rather constrains the class of properties they can verify to enable a more lightweight approach to verification. They currently have limited support for Rust patterns like functions manipulating mutable borrows, but leverage the use of liquid types to efficiently synthesize a large class of loop invariants, allowing the user to omit loop annotations.

The Move Prover (MVP) [439] is a formal verifier for smart contracts written in the Move programming language [440], and which has been used to verify an implementation of the Diem blockchain. Move uses a notion of (mutable) references which is very similar to Rust's borrows; in particular, there can't be two live mutable aliases of the same location at the same time. MVP leverages those aliasing constraints to generate pure proof obligations from user-annotated code, which are then sent to an SMT solver. Unlike other SMT-based tools we mentioned above, the generation of the verification conditions is based on a translation mechanism which generates a pure, functional model of the Move code. Interestingly, in the presence of functions (or expressions) returning a mutable borrows, a pattern they refer to as "dynamic mutable references", they generate a model which looks almost like Aeneas's backward functions. Designing this translation also required them to implement the equivalent of a borrow-checker for Move. Their handling of references however has some limitations compared to Aeneas's handling of mutable borrows: for instance, they track at most one mutable lifetime at a time, and they do not handle recursive functions returning mutable borrows. They also do not have a formal proof that their checker indeed provides memory safety.

Gillian-Rust. Gillian-Rust [441] is a verifier built on top of the Gillian symbolic analysis platform [442]. It aims at verifying the type safety and functional correctness of unsafe code. Interestingly, it is designed to be compatible with Creusot so as to allow a hybrid verification approach: when necessary, Gillian-Rust can be used to reason about

unsafe code, while Creusot's pure encoding can be leveraged to reason more efficiently about safe code.

RefinedC, RefinedRust. Our presentation of a low-level language equipped with a system of permissions, followed by an embedding into a theorem prover, is reminiscent of RefinedC [443]. RefinedC relies on magic wands to make up for the lack of borrows; wands, by virtue of being very general, require the use of heuristics. RefinedC, however, focuses on the subset of C code that obeys its permission discipline; and it relies on a memory model in Coq rather than a functional translation. RefinedC is foundational, and requires user annotations in an intrinsic style, while Aeneas generates a trusted, pure translation for extrinsic proofs. Crucially, both RefinedC and Aeneas rely on the fact that they never need to backtrack after applying rules. For RefinedC, this is made possible by restricting the user annotations to a carefully crafted, yet extensible, fragment of separation logic, along with alias types in the style of Mezzo's permissions [444]. For Aeneas, we never backtrack because we leverage the way borrows work to lazily terminate borrows. RefinedRust [445] follows the same approach as RefinedC but to reason about unsafe code.

KRust and RustSEM. KRust [446] and RustSEM [447] introduce executable, core semantics for Rust, formalized in the K framework. They both explicitly model lifetimes, and explicitly model the memory for the purpose of supporting unsafe code.

Featherweight Rust. Featherweight Rust [448] is a formalization of a core subset of Rust. It is inspired by Featherweight Java in that it emphasizes simplicity over exhaustivity. As such, it only supports a small subset of Rust which for instance doesn't include loops or function calls. This formalization includes a type system together with a proof of progress and preservation.

Other functional translations. [449] studies a problem similar to Aeneas by introducing an imperative calculus together with a meaning-preserving translation of programs written in this calculus into purely functional ones. The translation comes with a proof of soundness. It differs from Aeneas in that it is store-passing, though it can be quite fine-grained by relying on multiple store fragments. On our side, we manage to completely remove all references to memory for a large class of programs by means of our backward functions.

Mezzo. The Mezzo programming language [450] blends type system, ownership and shape analysis. In a fashion similar to Rust's borrow discipline, Mezzo draws a distinction between immutable, shareable data and mutable, uniquely owned data. It is expressive enough to support patterns such as in-place, tail-recursive concatenation

operations over lists. However, it does not easily support the equivalent of functions returning mutable borrows, which is the challenging case for LLBC$^\#$; with Mezzo, such functions have to be implemented either in an indirect manner [403, §2.4], or by using dynamic ownership tests [403, §2.5]. Mezzo is equipped with a syntactic proof of type soundness [403], but for an operational semantics à la ML. The "merge operation" [444] is akin to our join operation, though for a quite different type system which includes singleton types, substructural typing, and multiple types for a given variable x (top, "dynamic", "singleton x", and other degrees of folding/unfolding of a substructural type), while on our side we focus on a linear type system with a mechanism of borrows. The Mezzo merge algorithm is more sophisticated: it interleaves backtracking, quantifier instantiation strategies, and folding of existential predicates. We perform backtracking to reorganize environments when computing joins, but none of the others. We also crucially leverage our join operation to symbolically execute loops; Mezzo, on its side, being inspired by ML, does not have support for loops.

Shape Analysis. Perhaps more closely connected to this work is the field of shape analysis [411, 451–453]. Very active in the 2000s, the goal was to design abstract domains that would be able to infer shape predicates for pointer languages. Using familiar notions of concrete and symbolic executions, the analysis would then be able to identify bugs in programs via abstract interpretation. This has led to industrial tools such as Meta (née Facebook)'s Infer [454].

We differ from these works in several ways. First, we operate in a much more structured language than, say, C; these works traditionally operate over pointer languages, with NULL pointers, and untagged unions (anonymous sums). In our setting, we can enforce much more discipline onto the original language, and benefit from a lot more structure than languages like C may exhibit. However, this requires reasoning about and proving the correctness of a non-standard, borrow-centric semantics, and developing novel borrow-centric shape analyses, i.e., LLBC$^\#$. Second, our analysis does not exactly fit within the static analysis framework, and is merely inspired by it. We exhibit similarities in the design of our join operation, which just like in shape analysis involves reconciling competing shapes, folding inductive predicates, and abstracting over differing concrete values [411, 455].

Heapster. Heapster [456], just like Aeneas, attempts to extract pure code from low-level programs, and also comes with a soundness proof [457]. In the case of Heapster, the input is LLVM internal code, and the output is logical (non-executable) specifications that are extracted to Coq. The user must guide extraction by adding type annotations and loop invariants to their programs. Heapster also supports a notion of lifetimes,

which is however only partially covered by the soundness proof.

Kani. Kani is a bounded model-checker for Rust, which relies underneath on the C Bounded Model Checker [169]. Being a bounded model-checker, it can't be used to verify properties about programs with an arbitrary number of loop iterations or recursive function calls. On the other hand, it can reason about unsafe code, and is designed to be easy to use by regular engineers, and easy to integrate in CI.

Cogent. Cogent [240, 241] is a domain-specific language equipped with a linear type system. The Cogent compiler produces: C code; a high-level Isabelle/HOL specification; and a proof of refinement from the former to the latter. By virtue of producing an Isabelle/HOL specification, Cogent seamlessly composes with existing developments in that language, and can thus be integrated into a larger project, something Aeneas also enables. However, unlike Aeneas, the Cogent compiler does not need to be trusted since it produces a proof of translation correctness for each compilation run. We also remark that the linear type system of Cogent is significantly less expressive than Rust's; notably, Cogent does not seem to allow an equivalent of mutable borrows.

Stacked Borrows and Tree Borrows. Stacked Borrows [395] give a semantics to the notion of borrows in Rust, but sets out to achieve different goals than Aeneas: namely, to provide a set of rules that Rust developers can follow and validate their code against when writing unsafe code. The work comes with an extensive evaluation, which establishes both that the tool can detect incorrect uses (bugs were found), and that it can prove that some optimizations written using unsafe code are correct. This work adopts a very low-level view of memory, and it is unclear whether it can be used productively at the scale that we envision for Aeneas. The value of the work, however, lies in its precise, memory-based semantics of borrows; we are evaluating the feasibility of proving our semantics against it. Tree Borrows [396], the successor of Stacked Borrows [395], attempts to provide a semantics for correct borrow handling in the presence of unsafe code. This allows detecting, at run-time, violations of the contract (undefined behavior). Tree Borrows, unlike this work, operates at runtime by tracking permissions at the level of memory cells. We operate statically, and focus on safe code, proposing a new notion of borrow-checking that we prove to be semantically sound.

Oxide. The unpublished work-in-progress Oxide [458] attempts to formalize the type system of Rust. It targets a language which is close to Rust's surface AST, while on our side we target the lower-level MIR. It also attempts to give a more traditional interpretation of borrow-checking as a type system, while on our side we rely on a symbolic execution.

Chapter 16

Conclusion

The work presented in this thesis relies on two axes.

We started by exploring the practical limits of the verification of realistic programs through three use cases: the Noise⋆, zero-cost functors, and Dafny-in-Dafny projects. The Noise⋆ project allowed us, in the context of verifying cryptographic code, to move higher-up the stack of verified software by producing the first comprehensive verification result for a protocol compiler that targets C code, while providing a secure high-level API handling state machine transitions, peer and session management, state serialization and deserialization. With the zero-cost functors project we explored the problem of implementing low-level, efficient generic code, by establishing techniques which proved critical in scaling the popular HACL⋆ cryptographic library past 100,000 lines of verified source code. Finally, the Dafny-in-Dafny project allowed us to tackle the verification of a different class of programs, namely compilers, while studying the problem of proof stability.

Building on the practical experience that we acquired through these projects, and in particular the knowledge of the limitations of the existing toolchains, we then decided to create a new verification tool, the Aeneas framework. This tool targets Rust programs and crucially leverages the Rust type system to implement a lightweight functional translation targeting various theorem provers, that we leverage to implement custom, extensible automation.

The initial idea of this work was, perhaps naively, to implement some sort of a transpiler from Rust code to pure models, in the spirit of what Electrolysis and (in some sense) hacspec had already done, but for a larger class of programs. The idea of using backward functions to handle the complex case of functions returning mutable borrows came naturally, and manual case studies seemed to indicate that it applied to a large class of programs; surely such a translation could be done, and once implemented it would allow us to focus on the task of verifying interesting software, while abstracting

away a large class of boring, low-level details which had been crippling our verification efforts so far. We had already set our sights on targets such as WireGuard and OpenMLS, which would allow us to move beyond Noise⋆ and were then clearly out of reach with our existing toolchains. The task of making this translation systematic proved, however, a lot more difficult than expected; to our greatest surprise, it required us to take the extremely long detour of working on the semantics of borrow-checking. This detour had the benefit of producing several incidental, but welcome, side products: in addition to the translation, we designed the Low-Level Borrow-Calculus, a semantics for safe Rust which focuses on Rust's linear type system and mechanism of borrows, as well as a symbolic execution for LLBC, which in effect implements a borrow-checker. For the purpose of proving that Aeneas' translation is sound and extending it to better support disjunctions in the control-flow as well as loops, we set out to formalize it. The consequence is that, while we actually did not have enough time to finish the proof of the translation itself, we ended up with yet another side product: a proof that the symbolic execution implemented by Aeneas gives us memory safety, i.e., Aeneas provably implements a borrow-checker.

If there remains a lot of work to do on the toolchain, Aeneas is already a usable tool which supports an interesting subset of Rust and provides a relatively smooth proof experience. Outside the topic of improving Aeneas as a verification toolchain, the work we present in this manuscript also opened research possibilities which are independent of program verification. We now review the future work that lays ahead.

A first obvious task is to extend the subset supported by the symbolic execution and the translation, to include in particular nested borrows and ADTs containing borrows; those are crucial to support some common patterns like iterators. Properly handling these requires a more general treatment of symbolic values and region abstractions, in particular to model symbolic values which may contain an arbitrary number of borrows (because of, e.g., recursive data-structures). We actually introduced the ingredients to do so in the first Aeneas paper [399] with a notion of *loan* projector, by which we modeled the loans associated to the borrows which are inside a *symbolic* value; whenever revealing a borrow through a symbolic expansion (e.g., when refining a symbolic pair into a pair of symbolic values) we would insert the corresponding loan in the proper region abstraction. Put aside minor details, the more recent version of the semantics that we presented in the subsequent 2024 paper [400], improved all aspects of LLBC and LLBC$^{\#}$ *but* our handling of loans inside of region abstractions, that we simplified for the purpose of doing the proofs. In particular, we removed this notion of loan projectors, that we will have to reintroduce into the semantics. The implementation of the symbolic execution actually already contains experimental support for nested

borrows and ADTs containing borrows, while extending the translation is ongoing work but, as far as it stands, looks mostly straightforward. The main challenges lies in updating the soundness proofs to properly account for these more general symbolic values.

Going beyond nested borrows and ADTs containing borrows, we are keen on exploring how we could expand the symbolic execution performed by Aeneas to support all of Rust's idiosyncrasies; i.e., to turn Aeneas into a realistic borrow-checker. Put aside welcome extensions to the translation, we conjecture that doing so would also shed lights on the essence of borrow-checking, and we have been exploring potential relaxations of the rules to support interesting programs that are currently rejected by both the current Rust borrow-checker and Polonius. This line of work would also ask the question of whether it would be pertinent to use borrow-checkers based on a symbolic execution on real languages *in practice*. As of today and on the subset we support, we observe that a symbolic execution in the style of LLBC$^\#$ is a quite natural way of borrow-checking programs which do not contain loops. In the presence of loops, the answer is however more subtle: as our approach relies on heuristics, it is currently unclear what class of Rust programs it can ultimately validate, and more research and experimentation are needed on the topic.

We currently have a pen and paper proof that our symbolic execution soundly implements a borrow-checker; we can improve upon this results in two important ways. First, it would be valuable to mechanize this proof to make sure that we didn't miss any detail. This would be especially useful whenever we update the semantics later on, to make sure we do not forget to update parts of the proofs. The mechanization is currently ongoing. Second, there still remains the problem of proving that the translation is correct. In this regard, the high-level idea is currently to extend the LLBC$^\#$ environments with information (i.e., predicates) about the symbolic values, in the same spirit as what is traditionally done with standard symbolic executions [70]. For instance, the rule Le-ToSymbolic states that it is possible to abstract away a value v by transforming it into a symbolic value σ; we would update this rule so that the environment contains a predicate holding about σ (e.g., it is actually equal to v). Those predicates could be quite general; by constraining their shape, we would turn the symbolic execution into a synthesis mechanism.

The decision of implementing a translation to functional code, rather than relying on an encoding like the one permitted by prophecy variables [243, 244], is tightly linked to the choice of targeting backends such as interactive theorem provers; we wanted the translation to be natural, smooth to work with, and easy to link to the original Rust code. If we spent a substantial amount of time experimenting with potential target

provers, most of our work has been spent on designing and implementing the translation itself. As a consequence we did not have much time to develop the proof experience we target and that we described in the introduction. This proof experience would rely on using interactive theorem provers with good meta-programming facilities to implement enough automation to make the mundane proof obligations sufficiently smooth to work with, while providing escape hatches when the automation falls short, and allowing to extend the prover with custom automation whenever needs be. If the Lean backend, which has become our reference, already has promising foundations for a good proof experience, we are currently very far from this vision, whose feasibility remains a conjecture. Future work is also needed to demonstrate the applicability of Aeneas on realistic use cases. In this regard, we however note that previous work has successfully applied interactive theorem provers based on tactics to the verification of realistic, low-level software such as micro-kernels [188, 312] or cryptographic primitives [192]. Rust poses several challenges of its own, for instance through its heavy use of traits which make most programs higher-order. But in the context of those past verification efforts and under the condition that the translation works on the examples we wish to verify, we posit that the successful application of our methodology to realistic software should not come as a surprise. A more important open question is rather how far we can lower the burden of writing and maintaining proofs by working on the backend automation.

Another difficulty stems from the problem of embedding pure programs into the logics of interactive theorem provers, which are not as permissive as we would need. We mentioned the problem of encoding recursive and partial functions in Rust, that we solved in a lightweight manner by using a custom fixed-point operator together with a custom elaboration. Similar issues stem from Rust's traits, which we currently encode as structures (which stand for typeclasses). The fact that traits can make arbitrary references to other traits, and in particular be mutually recursive in a very liberal manner, poses interesting challenges.

Finally, the key point of Aeneas' translation is that it allows abstracting away memory on a large class of Rust programs. However, real-world Rust programs also commonly use features that Aeneas currently doesn't support well, such as I/O operations, interior mutability, concurrency, and unsafe code. We mentioned that we support I/O operations and some class of interior mutability through the use of a state-error monad. Future work will require designing a verification framework to comfortably reason about those. A strong possibility we envision is to connect our pure translation to a separation logic framework; this would allow reasoning about concurrency as well. Proofs would then involve a mixture of reasonings about pure

```
fn choose<'a, T>(
  b: bool,
  x: &'a mut T, y: &'a mut T)
  -> &'a mut T {
  if b { x } else { y }
}
```

$$
\begin{array}{l}
\forall\ b\ x\ y\ x_v\ y_v, \\
\ \{x \mapsto x_v * y \mapsto y_v\} \\
\ \ \mathsf{choose}(b,\ x,\ y) \\
\ \{\lambda z, \mathsf{let}\ (z_v,\ \mathsf{back}) = \mathsf{choose}\ b\ x_v\ y_v\ \mathsf{in} \\
\ \ z \mapsto z_v\ * \\
\ \ (\forall z'_v,\ z \mapsto z'_v \mathrel{-\!\!*} \\
\ \ \ (\mathsf{let}\ (x'_v,\ y'_v) = \mathsf{back}\ z'_v\ \mathsf{in}\ x \mapsto x'_v * y \mapsto y'_v))\}
\end{array}
$$

Figure 16.1: The choose Function and its Specification using Separation Logic

code and reasonings with separation logic. For instance, we could use separation logic to model calls to some mutex lock and unlock functions, while leveraging the (easier to work with) pure translation to model what happens between the two. Interestingly, our backward functions have a natural interpretation in terms of separation logic; for instance, we show in Figure 16.1 a Hoare triple that is (intuitively) satisfied by the choose function. Going further, we could connect high-level and low-level separation logics as has been done in prior work [459, 460] to, for instance, use a low-level logic in the spirit of RustBelt [393] to reason about unsafe code, a high-level logic to reason about I/O, coarse-grained concurrency and interior mutability, and 'pure' reasonings for the rest.

Bibliography

[1] CrowdStrike Editorial Team. *VENOM Vulnerability: Community Patching and Mitigation Update).* https://www.crowdstrike.com/blog/venom-vulnerability-community-patching-and-mitigation-update/, (2015).

[2] CrowdStrike Editorial Team. *CrowdStrike's work with the Democratic National Committee: Setting the record straight.* https://www.crowdstrike.com/blog/bears-midst-intrusion-democratic-national-committee/, (2020).

[3] R. Iordache, R. Goswami, J. Reid, A. Capoot, and A. K. Constantino, *Microsoft-CrowdStrike issue causes 'largest IT outage in history'*, CNBC .

[4] H. Ott, *Microsoft outages caused by CrowdStrike software glitch paralyze airlines, other businesses. Here's what to know.*, CNBC .

[5] National Vulnerability Database. *Heartbleed bug.* CVE-2014-0160 http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160, (2014).

[6] A. Edelman, *The Mathematics of the Pentium Division Bug*, Society for Industrial and Applied Mathematics (SIAM) Review **39**, 54–67 (1997).

[7] J.-L. Lions, L. Lübeck, J.-L. Fauquembergue, G. Kahn, W. Kubbat, S. Levedag, L. Mazzini, D. Merle, and C. O'Halloran. *Ariane 5 flight 501 failure report by the inquiry board.* European space agency Paris, (1996).

[8] N. Leveson and C. Turner, *An investigation of the Therac-25 accidents*, Computer **26**, 18–41 (1993).

[9] Akashi et al. *Investigation of an Accidental Exposure of Radiotherapy Patients in Panama.* International Atomic Energy Agency, (2001).

[10] P. Naur and B. Randell, *Software engineering: Report of a conference sponsored by the nato science committee, garmisch, germany, 7th-11th october 1968*, (1969).

[11] E. W. Dijkstra, *The Humble Programmer*, Communications of the ACM **15**, 859–866 (1972).

[12] M. Roser, H. Ritchie, and E. Mathieu, *What is Moore's Law?*, Our World in Data (2023).

[13] S. Wills, *Google Is 2 Billion Lines of Code—And It's All in One Place*, CNN (2015).

[14] S. McConnell, *Code Complete: A Practical Handbook of Software Construction, Second Edition*, Microsoft Press (2004).

[15] *ACM SIGSOFT Software Engineering Notes, vol. 6, no. 2*, SIGSOFT Softw. Eng. Notes **6** (1981).

[16] *Mars Climate Orbiter Mishap Investigation Board - Phase I Report.* NASA Headquarters, (1999).

[17] T. Lee, *Gangnam Style got so many views that it nearly broke YouTube*, Vox (2014).

[18] S. Lavington, *A history of Manchester computers*, The British Computer Society (1998).

[19] S. Lubar, *"Do Not Fold, Spindle or Mutilate": A Cultural History of the Punch Card*, Journal of American Culture, Volume 15, Issue 4 , 70–72 (1992).

[20] A. Turing. *Checking a large routine*, page 70–72. MIT Press, Cambridge, MA, USA, (1989).

[21] J. McCarthy. *A basis for a mathematical theory of computation, preliminary report.* In *Papers presented at the May 9-11, 1961, western joint IRE-AIEE-ACM computer conference*, pages 225–238, (1961).

[22] J. McCarthy. *Towards a Mathematical Science of Computation.* In *IFIP Congress*, (1962).

[23] J. McCarthy and J. A. Painter. *Correctness of a compiler for arithmetic expressions.* (1966).

[24] P. Naur, *Proof of algorithms by general snapshots*, BIT Numerical Mathematics **6**, 310–316 (1966).

[25] E. W. Dijkstra, *A constructive approach to the problem of program correctness*, BIT **8**, 174–186 (1968).

[26] Z. Manna, *The correctness of programs*, Journal of Computer and System Sciences **3**, 119–127 (1969).

[27] R. W. Floyd. *Assigning meaning to programs.* (1967).

[28] C. A. R. Hoare, *An axiomatic basis for computer programming*, Communications of the ACM **12**, 576–580 (1969).

[29] C. A. R. Hoare, *Proof of a program: FIND*, Commun. ACM **14**, 39–45 (1971).

[30] C. A. R. Hoare. *Procedures and parameters: An axiomatic approach.* In E. Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, pages 102–116, Berlin, Heidelberg, (1971). Springer Berlin Heidelberg.

[31] C. A. R. Hoare. *Parallel programming: an axiomatic approach.* In *Language Hierarchies and Interfaces*, pages 11–42. Springer, (1976).

[32] M. Foley and C. A. R. Hoare, *Proof of a recursive program: Quicksort*, The Computer Journal **14**, 391–395 (1971).

[33] M. Clint and C. Hoare, *Program proving: Jumps and functions*, Acta informatica **1**, 214–224 (1972).

[34] E. W. Dijkstra, *Guarded commands, nondeterminacy and formal derivation of programs*, Commun. ACM **18**, 453–457 (1975).

[35] W. D. Maurer. *Proving the correctness of a flight-director program for an airborne minicomputer.* In *Proceedings of the ACM SIGMINI/SIGPLAN Interface Meeting on Programming Systems in the Small Processor Environment*, SIGMINI '76, page 103–108, New York, NY, USA, (1976). Association for Computing Machinery.

[36] R. A. De Millo, R. J. Lipton, and A. J. Perlis, *Social Processes and Proofs of Theorems and Programs*, Communications of the ACM **22**, 271–280 (1979).

[37] ACM FORUM, *Comments on social processes and proofs*, Communications of the ACM **22** (1979).

[38] D. MacKenzie, *Mechanizing proof: computing, risk, and trust*, MIT Press, Cambridge, MA, USA (2001).

[39] J. Harrison, J. Urban, and F. Wiedijk. *History of Interactive Theorem Proving.* In *Computational Logic*, (2014).

[40] M. Davis, *The Early History of Automated Deduction*, Handbook of Automated Reasoning **1** (2002).

[41] A. Newell and H. Simon, *The logic theory machine–A complex information processing system*, IRE Transactions on Information Theory **2**, 61–79 (1956).

[42] A. N. Whitehead and B. Russel, *Principia Mathematica (3 vols)*, Cambridge University Press (1910).

[43] B. Russel, *Autobiography (2nd edition)*, Rootledge (1998).

[44] M. J. Beeson. *The Mechanization of Mathematics*, pages 77–134. Springer Berlin Heidelberg, Berlin, Heidelberg, (2004).

[45] J. A. Robinson, *Abraham Robinson. Proving a theorem (as done by man, logician, or machine). Summaries of talks presented at the Summer Institute for Symbolic Logic, Cornell University, 1957, 2nd edn., Communications Research Division, Institute for Defense Analyses, Princeton, N.J., 1960, pp. 350–352.*, Journal of Symbolic Logic **32**, 522–522 (1968).

[46] P. C. Gilmore, *A Proof Method for Quantification Theory: Its Justification and Realization*, IBM Journal of Research and Development **4**, 28–35 (1960).

[47] D. Prawitz, *An Improved Proof Procedure*, Theoria **26**, 102–139 (1960).

[48] M. Davis and H. Putnam, *Feasible computational methods in the propositional calculus*, Rensselaer Polytechnic Institute, Research Division (1958).

[49] M. Davis and H. Putnam, *A Computing Procedure for Quantification Theory*, J. ACM **7**, 201–215 (1960).

[50] M. Davis, G. Logemann, and D. Loveland, *A machine program for theorem-proving*, Commun. ACM **5**, 394–397 (1962).

[51] B. Dunham and J. North. *Theorem testing by computer.* In *Proceedings of the Symposium on Mathematical Theory of Automata*, pages 173–177, (1962).

[52] D. W. Loveland, *Mechanical Theorem-Proving by Model Elimination*, J. ACM **15**, 236–251 (1968).

[53] T. Chinlund, M. Davis, P. Hinman, and M. McIlroy. *Theorem proving by matching.* Bell Laboratories.

[54] J. A. Robinson, *A Machine-Oriented Logic Based on the Resolution Principle*, J. ACM **12**, 23–41 (1965).

[55] L. Wos, *J. A. Robinson. Automatic deduction with hyper-resolution. International journal of computer mathematics, vol. 1 no. 3 (1965), pp. 227–234.*, Journal of Symbolic Logic **39**, 189–190 (1974).

[56] L. Wos, D. Carson, and G. Robinson. *The unit preference strategy in theorem proving.* In *Proceedings of the October 27-29, 1964, Fall Joint Computer Conference, Part I*, AFIPS '64 (Fall, part I), page 615–621, New York, NY, USA, (1964). Association for Computing Machinery.

[57] L. Wos, G. A. Robinson, and D. F. Carson, *Efficiency and Completeness of the Set of Support Strategy in Theorem Proving*, J. ACM **12**, 536–541 (1965).

[58] G. Nelson and D. C. Oppen, *Simplification by Cooperating Decision Procedures*, ACM Trans. Program. Lang. Syst. **1**, 245–257 (1979).

[59] R. E. Shostak, *Deciding Combinations of Theories*, J. ACM **31**, 1–12 (1984).

[60] A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of satisfiability*, IOS Press (2009).

[61] J. K. Fichte, D. L. Berre, M. Hecher, and S. Szeider, *The Silent (R)evolution of SAT*, Commun. ACM **66**, 64–72 (2023).

[62] J. Marques-Silva and K. Sakallah, *GRASP: a search algorithm for propositional satisfiability*, IEEE Transactions on Computers **48**, 506–521 (1999).

[63] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. *Chaff: engineering an efficient SAT solver.* In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 530–535, (2001).

[64] S. Malik and L. Zhang, *Boolean satisfiability from theoretical hardness to practical success*, Commun. ACM **52**, 76–82 (2009).

[65] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. *DPLL(T): Fast Decision Procedures.* In R. Alur and D. A. Peled, editors, *Computer Aided Verification*, pages 175–188, Berlin, Heidelberg, (2004). Springer Berlin Heidelberg.

[66] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. *CVC4*. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, (2011).

[67] L. De Moura and N. Bjørner. *Z3: an efficient SMT solver*. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, (2008). Springer-Verlag.

[68] J. C. King. *A Program Verifier*. PhD thesis, Carnegie-Mellon University, (1970).

[69] J. C. King, *Symbolic execution and program testing*, Commun. ACM **19**, 385–394 (1976).

[70] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, *A Survey of Symbolic Execution Techniques*, ACM Comput. Surv. **51** (2018).

[71] R. L. Constable and S. D. Johnson. *A PL/CV precis*. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, page 7–20, New York, NY, USA, (1979). Association for Computing Machinery.

[72] D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. *Stanford Pascal Verifier user manual*. Technical report, Stanford, CA, USA, (1979).

[73] The Idris Development Team. *Idris: A Language for Type-Driven Development*. https://www.idris-lang.org.

[74] The Dafny Development Team. *The Dafny Programming and Verification Language*. https://dafny.org.

[75] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin. *Dependent Types and Multi-Monadic Effects in F**. In *ACM Symposium on Principles of Programming Languages*, pages 256–270, (2016).

[76] The FStar Development Team. *FStar: A Proof-oriented Programming Language*. https://github.com/FStarLang/FStar.

[77] J. McCarthy. *Computer programs for checking mathematical proofs*. (1962).

[78] H. Wang, *Toward mechanical mathematics*, IBM J. Res. Dev. **4**, 2–22 (1960).

[79] J. Avigad and J. Harrison, *Formally Verified Mathematics*, Communications of the ACM **57**, 66–75 (2014).

[80] Mizar Development Team. *The Mizar Mathematical Library*. https://mizar.uwb.edu.pl/library/.

[81] Mathlib Development Team. *mathlib4*. https://github.com/leanprover-community/mathlib4.

[82] W. McCune, *Solution of the Robbins Problem*, Journal of Automated Reasoning **19**, 263–276 (1997).

[83] A. Solovyev and T. C. Hales. *Efficient Formal Verification of Bounds of Linear Programs*. In J. H. Davenport, W. M. Farmer, J. Urban, and F. Rabe, editors, *Intelligent Computer Mathematics*, pages 123–132, Berlin, Heidelberg, (2011). Springer Berlin Heidelberg.

[84] G. Gonthier. *A computer-checked proof of the Four Colour Theorem*. (2005).

[85] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. L. Roux, A. Mahboubi, R. O'Connor, S. O. Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. *A Machine-Checked Proof of the Odd Order Theorem*. In *International Conference on Interactive Theorem Proving*, (2013).

[86] R. Milner. *Implementation and applications of Scott's logic for computable functions*. In *Proceedings of ACM Conference on Proving Assertions about Programs*, page 1–6, New York, NY, USA, (1972). Association for Computing Machinery.

[87] R. S. Boyer and J. S. Moore, *Proving Theorems about LISP Functions*, J. ACM **22**, 129–144 (1975).

[88] M. Kaufmann, P. Manolios, and J. S. Moore, *Computer-Aided Reasoning: An Approach*, Springer Science & Business Media (2000).

[89] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. https://coq.inria.fr/doc/V8.19.2/refman/, (2024).

[90] T. Coquand and C. Paulin. *Inductively defined types*. In *Proceedings of the International Conference on Computer Logic*, COLOG '88, page 50–66, Berlin, Heidelberg, (1988). Springer-Verlag.

[91] The Lean Development Team. *The Lean Programming Language and Theorem Prover*. https://lean-lang.org.

[92] M. Berger. *An Interview with Robin Milner*. http://users.sussex.ac.uk/~mfb21/interviews/milner/, (2003).

[93] P. Abrahams. *Machine verification of mathematical proofs,*. PhD thesis, MIT, (1963).

[94] J. R. Guard, F. C. Oglesby, J. H. Bennett, and L. G. Settle, *Semi-Automated Mathematics*, J. ACM **16**, 49–62 (1969).

[95] F. Wiedijk, *The Seventeen Provers of the World, volume 3600 of Lecture Notes in Computer Science*, Springer-Verlag (2006).

[96] N. de Bruijn. *The Mathematical Language Automath, its Usage, and Some of its Extensions**Reprinted from: Laudet, M., Lacombe, D. and Schuetzenberger, M., eds., Symposium on Automatic Demonstration, p. 29-61, by courtesy of Springer-Verlag, Heidelberg.* In R. Nederpelt, J. Geuvers, and R. de Vrijer, editors, *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*, pages 73–100. Elsevier, (1994).

[97] H. Geuvers and E. Barendsen, *Some logical and syntactical observations concerning the first-order dependent type system λP*, Mathematical. Structures in Comp. Sci. **9**, 335–359 (1999).

[98] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith, *Implementing mathematics with the Nuprl proof development system*, Prentice-Hall, Inc., USA (1986).

[99] The Adga Development Team. *Agda's Documentation*. https://agda.readthedocs.io/en/v2.6.1/index.html.

[100] F. Pfenning and C. Schürmann. *System Description: Twelf - A Meta-Logical Framework for Deductive Systems*. In *Proceedings of the 16th International Conference on Automated Deduction: Automated Deduction*, CADE-16, page 202–206, Berlin, Heidelberg, (1999). Springer-Verlag.

[101] L. de Moura and S. Ullrich. *The Lean 4 Theorem Prover and Programming Language*. In *2021 Conference on Automated Deduction*, pages 625–635. Springer, Cham, (2021).

[102] D. S. Scott, *A type-theoretical alternative to ISWIM, CUCH, OWHY*, Theoretical Computer Science **121**, 411–440 (1993).

[103] M. J. C. Gordon, R. Milner, C. P. Wadsworth, and P. T. Christopher, *Edinburgh lcf: a mechanized logic of computation*, Lecture Notes in Computer Science (1978).

[104] G. Cousineau and M. Mauny, *The Functional Approach to Programming*, Cambridge University Press (1998).

[105] P. Weis and X. Leroy, *Le langage Caml*, InterEditions (1993).

[106] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML (Revised)*, MIT Press (1997).

[107] K. Slind and M. Norrish. *A Brief Overview of HOL4.* In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 28–32, Berlin, Heidelberg, (2008). Springer Berlin Heidelberg.

[108] J. Harrison. *HOL Light: An Overview.* In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 60–66, Berlin, Heidelberg, (2009). Springer Berlin Heidelberg.

[109] L. C. Paulson, *Isabelle: The Next 700 Theorem Provers*, CoRR **cs.LO/9301106** (1993).

[110] L. C. Paulson, editor, *Isabelle: A Generic Theorem Prover*, Springer Berlin Heidelberg, Berlin, Heidelberg (1994).

[111] R. S. Boyer and J. S. Moore, *A Computational Logic*, ACM Monograph Series. Academic Press (1979).

[112] M. Kaufmann. *An interactive enhancement to the Boyer-Moore theorem prover.* In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 735–736, Berlin, Heidelberg, (1988). Springer Berlin Heidelberg.

[113] W. R. Bevier, W. A. Hunt, J. S. Moore, and W. D. Young, *An approach to systems verification*, Journal of Automated Reasoning **5**, 411–428 (1989).

[114] W. R. Bevier, *Kit and the short stack*, Journal of Automated Reasoning **5**, 519–530 (1989).

[115] W. A. Hunt. *FM8501: a verified microprocessor.* PhD thesis, University of Texas at Austin, (1985).

[116] M. Kaufmann, J. S. Moore, and P. Manolios, *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers, USA (2000).

[117] The ACL2 Development Team. *ACL2 Website*. https://www.cs.utexas.edu/~moore/acl2/.

[118] W. A. Hunt, M. Kaufmann, R. B. Krug, J. S. Moore, and E. W. Smith. *Meta Reasoning in ACL2*. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics*, pages 163–178, Berlin, Heidelberg, (2005). Springer Berlin Heidelberg.

[119] M. Kaufmann, P. Manolios, and J. Moore, *Computer-Aided Reasoning: ACL2 Case Studies* (2000).

[120] M. Melliar-Smith and J. Rushby, *The enhanced HDM system for specification and verification*, SIGSOFT Softw. Eng. Notes **10**, 41–43 (1985).

[121] S. Owre, J. M. Rushby, and N. Shankar. *PVS: A prototype verification system*. In D. Kapur, editor, *Automated Deduction—CADE-11*, pages 748–752, Berlin, Heidelberg, (1992). Springer Berlin Heidelberg.

[122] H. Ruess and N. Shankar. *Deconstructing Shostak*. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 19–28, (2001).

[123] J. Rushby and F. von Henke, *Formal verification of algorithms for critical systems*, SIGSOFT Softw. Eng. Notes **16**, 1–15 (1991).

[124] L. Lamport and P. M. Melliar-Smith, *Synchronizing clocks in the presence of faults*, J. ACM **32**, 52–78 (1985).

[125] The PVS Development Team. *PVS Website*. https://pvs.csl.sri.com.

[126] S. Owre, J. Rushby, N. Shankar, and F. von Henke, *Formal verification for fault-tolerant architectures: prolegomena to the design of PVS*, IEEE Transactions on Software Engineering **21**, 107–125 (1995).

[127] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. *HACL*: A Verified Modern Cryptographic Library*. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, (2017).

[128] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, et al. *Evercrypt: A fast,*

verified, cross-platform cryptographic provider. In *IEEE Symposium on Security and Privacy*, pages 634–653, (2019).

[129] J. Wensley, L. Lamport, J. Goldberg, M. Green, K. Levitt, P. Melliar-Smith, R. Shostak, and C. Weinstock, *SIFT: Design and analysis of a fault-tolerant computer for aircraft control*, Proceedings of the IEEE **66**, 1240–1255 (1978).

[130] *Peer review of a formal verification/design proof methodology*. NASA Conference Publication 2377, (1983).

[131] A. Cohn. *A Proof of Correctness of the Viper Microprocessor: The First Level*, pages 27–71. Springer US, Boston, MA, (1988).

[132] M. J. C. Gordon. *HOL: A Proof Generating System for Higher-Order Logic*, pages 73–128. Springer US, Boston, MA, (1988).

[133] J. S. Moore, T. W. Lynch, and M. Kaufmann, *A Mechanically Checked Proof of the AMD5K86TM Floating-Point Division Program*, IEEE Trans. Comput. **47**, 913–926 (1998).

[134] E. A. Emerson and E. M. Clarke. *Characterizing correctness properties of parallel programs using fixpoints*. In J. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming*, pages 169–181, Berlin, Heidelberg, (1980). Springer Berlin Heidelberg.

[135] J. P. Queille and J. Sifakis. *Specification and verification of concurrent systems in CESAR*. In M. Dezani-Ciancaglini and U. Montanari, editors, *International Symposium on Programming*, pages 337–351, Berlin, Heidelberg, (1982). Springer Berlin Heidelberg.

[136] E. M. Clarke. *The Birth of Model Checking*, pages 1–26. Springer Berlin Heidelberg, Berlin, Heidelberg, (2008).

[137] W. H. Hesselink, *Wait-free linearization with a mechanical proof*, Distributed Computing **9**, 21–36 (1995).

[138] G. Guiho and C. Hennebert. *SACEM software validation*. In *[1990] Proceedings. 12th International Conference on Software Engineering*, pages 186–191, (1990).

[139] D. Craigen, S. Gerhart, and T. Ralston, *Case study: Darlington nuclear generating station [software-driven shutdown systems]*, IEEE Software **11**, 30–32 (1994).

[140] D. Craigen, S. Gerhart, and T. Ralston, *Formal methods reality check: industrial usage*, IEEE Transactions on Software Engineering **21**, 90–98 (1995).

[141] J. Spivey, *The Z Notation: A Reference Manual*, Prentice Hall (1989).

[142] V. S. Alagar and K. Periyasamy. *Vienna Development Method*, pages 219–279. Springer New York, New York, NY, (1998).

[143] J.-R. Abrial, *The B-book: assigning programs to meanings*, Cambridge University Press, USA (1996).

[144] M. Phillips. *CICS/ESA 3.1 Experiences*. In J. E. Nicholls, editor, *Z User Workshop*, pages 179–185, London, (1990). Springer London.

[145] I. Houston and S. King. *CICS project report experiences and results from the use of Z in IBM*. In S. Prehn and W. J. Toetenel, editors, *VDM'91 Formal Software Development Methods*, pages 588–596, Berlin, Heidelberg, (1991). Springer Berlin Heidelberg.

[146] J. Bowen and V. Stavridou, *Safety-Critical Systems, Formal Methods and Standards*, Software Engineering Journal **8**, 189–209 (1993).

[147] Oxford University Computing Laboratory. *The Queen's Award for Technological Achievement 1992*. https://web.archive.org/web/20081202044350/http://web2.comlab.ox.ac.uk/oucl/about/qata92.html.

[148] J. Hill and P. Robinson. *The development of high reliability software-RRA's experience for safety critical systems*. In *IEE Colloquium on Software Requirements for High Integrity Systems*, pages 1/1–1/7, (1988).

[149] J. Hill, P. Robinson, and P. Stokes. *Safety critical software in control systems-a project view*. In *1989 A First International Conference on the Use of Programmable Electronic Systems in Safety Related Applications Computers and Safety*, pages 92–96. IET, (1989).

[150] D. May. *Use of formal methods by a silicon manufacturer*, page 107–129. Addison-Wesley Longman Publishing Co., Inc., USA, (1991).

[151] Y. Moy, A. Wallenburg, and B. BA. *Tokeneer: Beyond Formal Program Verification*, (2010).

[152] C. B. Jones and M. Thomas, *The Development and Deployment of Formal Methods in the UK*, Form. Asp. Comput. **34** (2022).

[153] D. J. Pavey and L. A. Winsborrow, *Demonstrating Equivalence of Source Code and PROM Contents*, The Computer Journal **36**, 654–667 (1993).

[154] I. O'Neill, D. Clutterbuck, P. Farrow, P. Summers, and W. Dolman, *The Formal Verification of Safety-critical Assembly Code*, IFAC Proceedings Volumes **21**, 115–120 (1988).

[155] B. Ladeau and C. Freeman, *Using formal specification for product development*, Hewlett-Packard Journal , 62–66 (1991).

[156] P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier, *Météor: A successful application of B in a large project*, , 369–387 (1999).

[157] R.-J. Back, *Correctness preserving program refinements: Proof theory and applications*, (No Title) (1980).

[158] C. Morgan, *Programming from specifications*, Prentice-Hall, Inc., USA (1990).

[159] J.-R. Abrial. *Spécification, construction et vérification de programmes : le parcours d'une pensée scientifique sur une quarantaine d'années, Séminaire du Collège de France.* https://www.college-de-france.fr/fr/agenda/seminaire/prouver-les-programmes-pourquoi-quand-comment/specification-construction-et-verification-de-programmes-le-parcours-une-pense (2015).

[160] A. Hall, *Seven myths of formal methods*, IEEE Software **7**, 11–19 (1990).

[161] J. Bowen and M. Hinchey, *Seven More Myths of Formal Methods*, IEEE Software **12**, 34–41 (1995).

[162] K. Finney and N. Fenton, *Evaluating the effectiveness of Z: The claims made about CICS and where we go from here*, Journal of Systems and Software **35**, 209–216 (1996).

[163] S. Pfleeger and L. Hatton, *Investigating the influence of formal methods*, Computer **30**, 33–43 (1997).

[164] T. Ball, B. Cook, V. Levin, and S. Rajamani. *SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft.* volume 2999, pages 1–20, (2004).

[165] C. Calcagno and D. Distefano. *Infer: an automatic program verifier for memory safety of C programs*. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, page 459–465, Berlin, Heidelberg, (2011). Springer-Verlag.

[166] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. *Go Huge or Go Home: POPL'19 Most Influential Paper Retrospective*. https://blog.sigplan.org/2020/03/03/go-huge-or-go-home-popl19-most-influential-paper-retrospective/, (2020).

[167] Infer. *About Infer*. https://fbinfer.com/docs/about-Infer/, (2024).

[168] P. O'Hearn, *Separation logic*, Commun. ACM **62**, 86–95 (2019).

[169] E. Clarke, D. Kroening, and F. Lerda. *A Tool for Checking ANSI-C Programs*. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, Berlin, Heidelberg, (2004). Springer Berlin Heidelberg.

[170] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, *Bounded Model Checking Using Satisfiability Solving*, Form. Methods Syst. Des. **19**, 7–34 (2001).

[171] P. Cousot and R. Cousot. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, (1977). Association for Computing Machinery.

[172] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. *A static analyzer for large safety-critical software*. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, page 196–207, New York, NY, USA, (2003). Association for Computing Machinery.

[173] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. *The ASTREÉ Analyzer*. In M. Sagiv, editor, *Programming Languages and Systems*, pages 21–30, Berlin, Heidelberg, (2005). Springer Berlin Heidelberg.

[174] X. Leroy. *Formal certification of a compiler back-end, or: programming a compiler with a proof assistant*. In *33rd ACM symposium on Principles of Programming Languages*, pages 42–54. ACM Press, (2006).

[175] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. *seL4: formal verification of an OS kernel*. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, (2009). Association for Computing Machinery.

[176] ACM. *ACM Software System Award*. https://awards.acm.org/software-system, (2024).

[177] AbsInt. *Formally verified compilation*. https://www.absint.com/compcert/index.htm, (2024).

[178] J. Souyris. *Industrial use of CompCert on a safety-critical software product*. https://projects.laas.fr/IFSE/FMF/J3/slides/P05_Jean_Souyiris.pdf, (2014).

[179] D. Kästner, J. Barrho, U. Wünsche, M. Schlickling, B. Schommer, M. Schmidt, C. Ferdinand, X. Leroy, and S. Blazy. *CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler*. In *ERTS2 2018 - 9th European Congress Embedded Real-Time Software and Systems*, pages 1–9, Toulouse, France, (2018). 3AF, SEE, SIE.

[180] G. Klein, J. Andronick, M. Fernandez, I. Kuz, T. Murray, and G. Heiser, *Formally verified software in the real world*, Commun. ACM **61**, 68–77 (2018).

[181] K. R. M. Leino. *Dafny: An automatic program verifier for functional correctness*. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, (2010).

[182] K. R. M. Leino, P. Müller, and J. Smans. *Verification of Concurrent Programs with Chalice*, pages 195–222. Springer Berlin Heidelberg, Berlin, Heidelberg, (2009).

[183] J.-C. Filliâtre and A. Paskevich. *Why3 – Where Programs Meet Provers*. In *ESOP'13 22nd European Symposium on Programming*, volume 7792 of *LNCS*, Rome, Italy, (2013). Springer.

[184] P. Müller, M. Schwerhoff, and A. J. Summers. *Viper: A Verification Infrastructure for Permission-Based Reasoning*. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 41–62, Berlin, Heidelberg, (2016). Springer Berlin Heidelberg.

[185] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. *HACL*: A Verified Modern Cryptographic Library.* In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1789–1806, New York, NY, USA, (2017). Association for Computing Machinery.

[186] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. *IronFleet: proving practical distributed systems correct.* In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 1–17, New York, NY, USA, (2015). Association for Computing Machinery.

[187] T. Hance, A. Lattuada, C. Hawblitzel, J. Howell, R. Johnson, and B. Parno. *Storage Systems are Distributed Systems (So Verify Them That Way!).* In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 99–115. USENIX Association, (2020).

[188] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. *CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels.* In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, Savannah, GA, (2016). USENIX Association.

[189] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. *CakeML: a verified implementation of ML.* In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 179–191, New York, NY, USA, (2014). Association for Computing Machinery.

[190] T. Bourke, L. Brun, P.-E. Dagand, X. Leroy, M. Pouzet, and L. Rieg. *A Formally Verified Compiler for Lustre.* In *PLDI 2017 - 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Barcelone, Spain, (2017). ACM.

[191] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub. *Implementing TLS with Verified Cryptographic Security.* In *2013 IEEE Symposium on Security and Privacy*, pages 445–459, (2013).

[192] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. *Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises.* In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, (2019).

[193] *DeepSpec*. https://deepspec.org/main, (2020).

[194] X. Yang, Y. Chen, E. Eide, and J. Regehr. *Finding and understanding bugs in C compilers*. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, (2011). Association for Computing Machinery.

[195] K. Fisher, J. Launchbury, and R. Richards, *The HACMS program: Using formal methods to eliminate exploitable bugs*, Philosophical Transactions of The Royal Society A Mathematical Physical and Engineering Sciences **375**, 20150401 (2017).

[196] M. Dodds, *Formally Verifying Industry Cryptography*, IEEE Security & Privacy **20**, 2–7 (2022).

[197] K. Carter, A. Foltzer, J. Hendrix, B. Huffman, and A. Tomb. *SAW: the software analysis workbench*. In *Proceedings of the 2013 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '13, page 15–18, New York, NY, USA, (2013). Association for Computing Machinery.

[198] P. He, E. Westbrook, B. Carmer, C. Phifer, V. Robert, K. Smeltzer, A. Ştefănescu, A. Tomb, A. Wick, M. Yacavone, and S. Zdancewic, *A type system for extracting functional specifications from memory-safe imperative programs*, Proc. ACM Program. Lang. **5** (2021).

[199] Q. Carbonneaux, N. Zilberstein, C. Klee, P. W. O'Hearn, and F. Zappa Nardelli. *Applying formal verification to microkernel IPC at meta*. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, page 116–129, New York, NY, USA, (2022). Association for Computing Machinery.

[200] B. Cook. *Formal Reasoning About the Security of Amazon Web Services*. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification*, pages 38–47, Cham, (2018). Springer International Publishing.

[201] C. Disselkoen, A. Eline, S. He, K. Headley, M. Hicks, K. Hietala, J. Kastner, A. Mamat, M. McCutchen, N. Rungta, B. Shah, E. Torlak, and A. Wells. *How We Built Cedar: A Verification-Guided Approach*. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 351–357, New York, NY, USA, (2024). Association for Computing Machinery.

[202] *ProvenRun Website*. https://provenrun.com, (2024).

[203] The GCC Development Team. *GCC Git Repository*. https://gcc.gnu.org/git.html, (2024).

[204] D. Monniaux and S. Boulmé. *The Trusted Computing Base of the CompCert Verified Compiler*. In I. Sergey, editor, *Programming Languages and Systems*, pages 204–233, Cham, (2022). Springer International Publishing.

[205] Trustworthy Systems. *Multicore seL4 Verification*. https://trustworthy.systems/projects/OLD/multicore-sel4, (2024).

[206] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, *Comprehensive formal verification of an OS microkernel*, ACM Transactions on Computer Systems **32** (2014).

[207] R. Bornat. *Proving Pointer Programs in Hoare Logic*. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction*, pages 102–126, Berlin, Heidelberg, (2000). Springer Berlin Heidelberg.

[208] I. T. Kassios. *Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions*. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods*, pages 268–283, Berlin, Heidelberg, (2006). Springer Berlin Heidelberg.

[209] J. Smans, B. Jacobs, and F. Piessens, *Implicit dynamic frames*, ACM Trans. Program. Lang. Syst. **34** (2012).

[210] J. Reynolds. *Separation logic: a logic for shared mutable data structures*. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, (2002).

[211] Y. Zhou, J. Bosamiya, Y. Takashima, J. Li, M. Heule, and B. Parno. *Mariposa: Measuring SMT Instability in Automated Program Verification*. In *2023 Formal Methods in Computer-Aided Design (FMCAD)*, pages 178–188. IEEE, (2023).

[212] The FStar Development Team. *Proof-Oriented Programming in F*, Under the hood, Understanding how F* uses Z3*. https://fstar-lang.org/tutorial/book/under_the_hood/uth_smt.html, (2024).

[213] L. de Moura and G. O. Passmore. *The Strategy Challenge in SMT Solving*, pages 15–44. Springer Berlin Heidelberg, Berlin, Heidelberg, (2013).

[214] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. *Ironclad Apps:{End-to-End} Security via Automated {Full-System} Verification.* In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 165–181, (2014).

[215] *Enhancing Proof Stability Session, Dafny Workshop, POPL 2024.* https://popl24.sigplan.org/details/dafny-2024-papers/14/Enhancing-Proof-Stability, (2024).

[216] The Matryoshka Team. *Matryoshka, Fast Interactive Verification through Strong Higher-Order Automation.* https://matryoshka-project.github.io, (2017).

[217] L. de Moura. *Lost in translation: how easy (automated reasoning) problems become hard due to bad encodings.* https://leodemoura.github.io/files/Vampire2015.pdf, (2015).

[218] G. Martínez, D. Ahman, V. Dumitrescu, N. Giannarakis, C. Hawblitzel, C. Hritcu, M. Narasimhamurthy, Z. Paraskevopoulou, C. Pit-Claudel, J. Protzenko, T. Ramananandro, A. Rastogi, and N. Swamy. *Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms.* In *European Symposium on Programming*, pages 30–59. Springer, (2019).

[219] G. Melquiond and R. Rieu-Helft. *A Why3 Framework for Reflection Proofs and Its Application to GMP's Algorithms.* In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *Automated Reasoning*, pages 178–193, Cham, (2018). Springer International Publishing.

[220] G. Gonthier, A. Mahboubi, and E. Tassi. *A Small Scale Reflection Extension for the Coq system.* Research Report RR-6455, Inria Saclay Ile de France, (2016).

[221] A. Chlipala, *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*, The MIT Press (2013).

[222] D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. Anderson. *Planning for change in a formal verification of the raft consensus protocol.* In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, page 154–165, New York, NY, USA, (2016). Association for Computing Machinery.

[223] R. Y. Lewis and P.-N. Madelaine. *Simplifying Casts and Coercions*, (2020).

[224] T. Ringer, K. Palmskog, I. Sergey, M. Gligoric, and Z. Tatlock, *QED at Large: A Survey of Engineering of Formally Verified Software*, Foundations and Trends® in Programming Languages **5**, 102–281 (2019).

[225] J. C. Blanchette, S. Böhme, and L. C. Paulson. *Extending Sledgehammer with SMT Solvers*. In N. Bjørner and V. Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, pages 116–130, Berlin, Heidelberg, (2011). Springer Berlin Heidelberg.

[226] M. S. Nawaz, M. Z. Nawaz, O. Hasan, P. Fournier-Viger, and M. Sun, *Proof searching and prediction in HOL4 with evolutionary/heuristic and deep learning techniques*, Applied Intelligence **51**, 1580–1601 (2021).

[227] C. Kaliszyk and J. Urban, *Learning-assisted Automated Reasoning with Flyspeck*, CoRR **abs/1211.7012** (2012).

[228] K. Yang, A. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. Prenger, and A. Anandkumar. *LeanDojo: Theorem Proving with Retrieval-Augmented Language Models*. In *Neural Information Processing Systems (NeurIPS)*, (2023).

[229] G. Ebner, S. Ullrich, J. Roesch, J. Avigad, and L. de Moura, *A metaprogramming framework for formal verification*, Proc. ACM Program. Lang. **1** (2017).

[230] S. Dailler, C. Marché, and Y. Moy, *Lightweight Interactive Proving inside an Automatic Program Verifier*, Electronic Proceedings in Theoretical Computer Science **284**, 1–15 (2018).

[231] J. Blanchette, D. Greenaway, C. Kaliszyk, D. Kühlwein, and J. Urban, *A Learning-Based Fact Selector for Isabelle/HOL*, Journal of Automated Reasoning **57** (2016).

[232] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue. *A Messy State of the Union: Taming the Composite State Machines of TLS*. In *IEEE Symposium on Security and Privacy*, pages 535–552, (2015).

[233] M. Polubelova, K. Bhargavan, J. Protzenko, B. Beurdouche, A. Fromherz, N. Kulatova, and S. Zanella-Béguelin. *HACLxN: Verified Generic SIMD Crypto (for All Your Favourite Platforms)*. In *ACM Conference on Computer and Communications Security*, page 899–918, (2020).

[234] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Beguelin, K. Bhargavan, J. Pan, and J. K. Zinzindohoue. *Implementing and Proving the TLS 1.3 Record Layer*. In *IEEE Symposium on Security and Privacy*, pages 463–482, (2017).

[235] N. Swamy, A. Rastogi, A. Fromherz, D. Merigoux, D. Ahman, and G. Martínez, *SteelCore: an extensible concurrent separation logic for effectful dependently typed programs*, Proc. ACM Program. Lang. **4** (2020).

[236] A. Fromherz, A. Rastogi, N. Swamy, S. Gibson, G. Martínez, D. Merigoux, and T. Ramananandro, *Steel: proof-oriented programming in a dependently typed concurrent separation logic*, Proceedings of the ACM on Programming Languages **5**, 1–30 (2021).

[237] A. Reitz, A. Fromherz, and J. Protzenko. *StarMalloc: A Formally Verified, Concurrent, Performant, and Security-Oriented Memory Allocator*, (2024).

[238] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. *CakeML: a verified implementation of ML*. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, (2014).

[239] S. Ho, O. Abrahamsson, R. Kumar, M. O. Myreen, Y. K. Tan, and M. Norrish. *Proof-Producing Synthesis of CakeML with I/O and Local State from Monadic HOL Functions*. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *Automated Reasoning*, pages 646–662, Cham, (2018). Springer International Publishing.

[240] L. O'Connor, C. Rizkallah, Z. Chen, S. Amani, J. Lim, Y. Nagashima, T. Sewell, A. Hixon, G. Keller, T. Murray, and G. Klein. *Cogent: Certified Compilation for a Functional Systems Language*, (2016). arXiv:1601.05520 [cs].

[241] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, et al., *Cogent: Verifying high-assurance file system implementations*, ACM SIGARCH Computer Architecture News **44**, 175–188 (2016).

[242] S. Ullrich, *Simple verification of rust programs via functional purification*, Master's Thesis, Karlsruher Institut für Technologie (KIT) (2016).

[243] Y. Matsushita, T. Tsukada, and N. Kobayashi. *RustHorn: CHC-Based Verification for Rust Programs.* In *ESOP*, pages 484–514, (2020).

[244] X. Denis, J.-H. Jourdan, and C. Marché. *Creusot: a foundry for the deductive verification of rust programs*. In *International Conference on Formal Engineering Methods (ICFEM)*, pages 90–105. Springer, (2022).

[245] K. Cohn-Gordon, C. J. F. Cremers, and L. Garratt. *On Post-compromise Security*. In *IEEE Computer Security Foundations Symposium*, pages 164–178, (2016).

[246] B. Blanchet. *Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif*. In *Found. Trends Priv. Secur.*, volume 1, pages 1–135, (2016).

[247] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin. *The TAMARIN Prover for the Symbolic Analysis of Security Protocols*. In *Computer Aided Verification*, volume 8044 of *LNCS*, pages 696–701, (2013).

[248] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe. *A Comprehensive Symbolic Analysis of TLS 1.3*. In *ACM Conference on Computer and Communications Security*, page 1773–1788, (2017).

[249] K. Bhargavan, B. Blanchet, and N. Kobeissi. *Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate*. In *IEEE Symposium on Security and Privacy*, pages 483–502, (2017).

[250] N. Kobeissi, K. Bhargavan, and B. Blanchet. *Automated Verification for Secure Messaging Protocols and their Implementations: A Symbolic and Computational Approach*. In *IEEE European Symposium on Security and Privacy*, pages 435–450, (2017).

[251] B. Blanchet, *A Computationally Sound Mechanized Prover for Security Protocols*, IEEE Trans. Dependable Secur. Comput. **5**, 193–207 (2008).

[252] C. Fournet, M. Kohlweiss, and P.-Y. Strub. *Modular Code-Based Cryptographic Verification*. In *ACM Conference on Computer and Communications Security*, page 341–350, (2011).

[253] B. Lipp, B. Blanchet, and K. Bhargavan. *A Mechanised Cryptographic Proof of the WireGuard Virtual Private Network Protocol*. In *IEEE European Symposium on Security and Privacy*, pages 231–246, (2019).

[254] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. *Implementing TLS with Verified Cryptographic Security*. In *IEEE Symposium on Security and Privacy*, pages 445–459, (2013).

[255] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno. *SoK: Computer-Aided Cryptography*. In *IEEE Symposium on Security and Privacy*, pages 777–795, (2021).

[256] T. Ramananandro, A. Delignat-Lavaud, C. Fournet, N. Swamy, T. Chajed, N. Kobeissi, and J. Protzenko. *Everparse: verified secure zero-copy parsers for authenticated message formats*. In *USENIX Security Symposium*, pages 1465–1482, (2019).

[257] D. Cadé and B. Blanchet, *Proved generation of implementations from computationally secure protocol specifications*, J. Comput. Secur. **23**, 331–402 (2015).

[258] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan. *Formally Verified Cryptographic Web Applications in WebAssembly*. In *IEEE Symposium on Security and Privacy*, pages 1256–1274, (2019).

[259] R. Küsters, T. Truderung, and J. Graf. *A Framework for the Cryptographic Verification of Java-like Programs*. In *IEEE Computer Security Foundations Symposium*, pages 198–212, (2012).

[260] T. Perrin. *The Noise Protocol Framework*. http://noiseprotocol.org/noise.html, (2018).

[261] G. Girol, L. Hirschi, R. Sasse, D. Jackson, C. Cremers, and D. Basin. *A Spectral Analysis of Noise: A Comprehensive, Automated, Formal Analysis of Diffie-Hellman Protocols*. In *USENIX Security Symposium*, (2020).

[262] N. Kobeissi, G. Nicolas, and K. Bhargavan. *Noise Explorer: Fully Automated Modeling and Verification for Arbitrary Noise Protocols*. In *IEEE European Symposium on Security and Privacy*, pages 356–370, (2019).

[263] B. Dowling, P. Rösler, and J. Schwenk. *Flexible Authenticated and Confidential Channel Establishment (fACCE): Analyzing the Noise Protocol Framework*. In *Public-Key Cryptography*, volume 12110, pages 341–373, (2020).

[264] J. Protzenko, J. K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Z. Béguelin, A. Delignat-Lavaud, C. Hritcu, K. Bhargavan, C. Fournet, and N. Swamy, *Verified low-level programming embedded in F**, Proceedings of the ACM on Programming Languages **1**, 17:1–17:29 (2017).

[265] K. Bhargavan, A. Bichhawat, Q. H. Do, P. Hosseyni, R. Küsters, G. Schmitz, and T. Würtele. *DY* : A Modular Symbolic Verification Framework for Executable*

Cryptographic Protocol Code. In *IEEE European Symposium on Security and Privacy*, Virtual, Austria, (2021).

[266] S. Ho, J. Protzenko, A. Bichhawat, and K. Bhargavan. *Noise*: A Library of Verified High-Performance Secure Channel Protocol Implementations*. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, (2022).

[267] D. J. Bernstein, T. Lange, and P. Schwabe. *The security impact of a new cryptographic library*. In *International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*, pages 159–176. Springer, (2012).

[268] R. Barnes and K. Bhargavan. *Hybrid Public Key Encryption*. IRTF Internet-Draft draft-irtf-cfrg-hpke-02, (2019).

[269] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Pan, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Béguelin, and J. K. Zinzindohoué. *Implementing and Proving the TLS 1.3 Record Layer*. In *IEEE Symposium on Security and Privacy*, (2017).

[270] A. Delignat-Lavaud, C. Fournet, B. Parno, J. Protzenko, T. Ramananandro, J. Bosamiya, J. Lallemand, I. Rakotonirina, and Y. Zhou. *A Security Model and Fully Verified Implementation for the IETF QUIC Record Layer*. In *IEEE Symposium on Security and Privacy*, pages 1162–1178, (2021).

[271] *Verified implementations for the Noise family of protocols*. https://github.com/Inria-Prosecco/noise-star, (2022).

[272] D. Dolev and A. Yao. *On the Security of Public Key Protocols*. In *IEEE Trans. Inf. Theor.*, volume 29, page 198–208, (2006).

[273] *DY* Source Code Repository*. https://github.com/REPROSEC/dolev-yao-star.

[274] T. Y. C. Woo and S. S. Lam. *Authentication for Distributed Systems*. In *Computer*, volume 25, page 39–52, (1992).

[275] B. Dowling and K. G. Paterson. *A Cryptographic Analysis of the WireGuard Protocol*. In *Applied Cryptography and Network Security*, pages 3–21, (2018).

[276] A. Pironti and R. Sisto, *Provably Correct Java Implementations of Spi Calculus Security Protocols Specifications*, Journal of Computer Security **29**, 302–314 (2010).

[277] D. Cadé and B. Blanchet, *Proved generation of implementations from computationally secure protocol specifications*, Journal of Computer Security **23**, 331–402 (2015).

[278] J. A. McCarthy, S. Krishnamurthi, J. D. Guttman, and J. D. Ramsdell. *Compiling Cryptographic Protocols for Deployment on the Web.* In *International Conference on World Wide Web*, page 687–696, (2007).

[279] R. Corin, P.-M. Deniélou, C. Fournet, K. Bhargavan, and J. Leifer, *A Secure Compiler for Session Abstractions*, Journal of Computer Security **16**, 573–636 (2008).

[280] C. Fournet, G. L. Guernic, and T. Rezk. *A security-preserving compiler for distributed programs: from information-flow policies to cryptographic mechanisms.* In *ACM Conference on Computer and Communications Security*, pages 432–441. ACM, (2009).

[281] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic. *SoK: General Purpose Compilers for Secure Multi-Party Computation.* In *IEEE Symposium on Security and Privacy*, pages 1220–1237, (2019).

[282] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis, *Refinement types for secure implementations*, ACM Transactions on Programming Languages and Systems **33**, 8:1–8:45 (2011).

[283] K. Bhargavan, C. Fournet, and A. D. Gordon. *Modular verification of security protocol code by typing.* In *ACM Principles of Programming Languages*, pages 445–456, (2010).

[284] M. Backes, C. Hritcu, and M. Maffei, *Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations*, Journal of Computer Security **22**, 301–353 (2014).

[285] G. Barthe, C. Fournet, B. Grégoire, P. Strub, N. Swamy, and S. Z. Béguelin. *Probabilistic relational verification for cryptographic implementations.* In *ACM Principles of Programming Languages*, pages 193–206, (2014).

[286] M. Avalle, A. Pironti, D. Pozza, and R. Sisto, *JavaSPI: A Framework for Security Protocol Implementation*, International Journal of Secure Software Engineering **2**, 34–48 (2011).

[287] F. Dupressoir, A. D. Gordon, J. Jürjens, and D. A. Naumann, *Guiding a general-purpose C verifier to prove cryptographic protocols*, Journal of Computer Security **22**, 823–866 (2014).

[288] M. Aizatulin, A. D. Gordon, and J. Jürjens. *Extracting and verifying cryptographic models from C protocol code by symbolic execution.* In *ACM Conference on Computer and Communications Security*, pages 331–340, (2011).

[289] S. Chaki and A. Datta. *ASPIER: An Automated Framework for Verifying Security Protocol Implementations.* In *IEEE Computer Security Foundations Symposium*, pages 172–185, (2009).

[290] T. Wall Street Journal. *Provable Security for Modern Applications.* https://partners.wsj.com/aws/reinventing-with-the-cloud/provable-security-for-modern-applications/ Retrieved February 2023, (2023).

[291] H. S. Warren, *Hacker's delight*, Pearson Education (2013).

[292] S. Gueron. *Intel® Advanced Encryption Standard (AES) New Instructions Set.* https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf, (2012).

[293] T. Oliveira, J. López, H. Hışıl, A. Faz-Hernández, and F. Rodríguez-Henríquez. *How to (pre-)compute a ladder: Improving the Performance of X25519 and X448.* In *Proceedings of Selected Areas in Cryptography (SAC)*, (2017).

[294] MITRE. *CVE-2017-5715. Systems with microprocessors utilizing speculative execution and indirect branch prediction may allow unauthorized disclosure of information to an attacker with local user access via a side-channel analysis.* http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5715, (2018).

[295] MITRE. *CVE-2017-5753. Systems with microprocessors utilizing speculative execution and branch prediction may allow unauthorized disclosure of information to an attacker with local user access via a side-channel analysis.* http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5753, (2018).

[296] MITRE. *CVE-2014-0160.* http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160, (2013).

[297] D. Benjamin. *poly1305-x86.pl produces incorrect output.* https://mta.openssl.org/pipermail/openssl-dev/2016-March/006161, (2016).

[298] H. Böck. *Wrong results with Poly1305 functions*. https://mta.openssl.org/pipermail/openssl-dev/2016-March/006413, (2016).

[299] Bugzilla. *Crash in Hacl Chacha20Poly1305_128 aead_encrypt & Hacl Chacha20Poly1305_128 aead_decrypt*. https://bugzilla.mozilla.org/show_bug.cgi?id=1605369, (2019).

[300] N. Mouha, M. S. Raunak, D. R. Kuhn, and R. Kacker, *Finding bugs in cryptographic hash function implementations*, IEEE transactions on reliability **67**, 870–884 (2018).

[301] N. Mouha. *SHA-3 Buffer Overflow*. https://mouha.be/sha-3-buffer-overflow/, (2022).

[302] J. A. Donenfeld. *new 25519 measurements of formally verified implementations*. http://moderncrypto.org/mail-archive/curves/2018/000972.html, (2018).

[303] F. Voight. *CVE-2012-2459 (block merkle calculation exploit)*. https://bitcointalk.org/?topic=102395, (2012).

[304] S. Gueron and V. Krasnov. *The Fragility of AES-GCM Authentication Algorithm*. In *Proceedings of the Conference on Information Technology: New Generations*, (2014).

[305] OpenSSL. *Don't break carry chains*. GitHub commit 4b8736a22e758c371bc2f8b3534dc0c274acf42c, (2016).

[306] OpenSSL. *Don't loose [sic] 59-th bit*. GitHub commit bbe9769ba66ab2512678a87b0d9b266ba970db05, (2016).

[307] OpenSSL. *Chase overflow bit on x86 and ARM platforms*. GitHub commit dc3c5067cd90f3f2159e5d53c57b92730c687d7e, (2016).

[308] C. L. Biffle. *NaCl/x86 appears to leave return addresses unaligned when returning through the springboard*. https://bugs.chromium.org/p/nativeclient/issues/detail?id=245, (2010).

[309] N. Swamy, T. Ramananandro, A. Rastogi, I. Spiridonova, H. Ni, D. Malloy, J. Vazquez, M. Tang, O. Cardona, and A. Gupta. *Hardening Attack Surfaces with Formally Proven Binary Format Parsers*. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22)*, (2022).

[310] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S.-C. Weng, H. Zhang, and Y. Guo. *Deep Specifications and Certified Abstraction Layers*. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, (2015).

[311] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo. *CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels*. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 653–669, Berkeley, CA, USA, (2016). USENIX Association.

[312] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, T. Sewell, H. Tuch, and S. Winwood. *seL4: Formal Verification of an OS Kernel*. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, (2009).

[313] T. Mathlib Community. *The Lean Mathematical Library*. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 367–381, New York, NY, USA, (2020). Association for Computing Machinery.

[314] T. A. L. Sewell, M. O. Myreen, and G. Klein. *Translation Validation for a Verified OS Kernel*. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, (2013).

[315] P. Futz. *C Preprocessor tricks, tips and idioms*. https://github.com/pfultz2/Cloak/wiki/C-Preprocessor-tricks,-tips,-and-idioms, (2015).

[316] S. Ho, A. Fromherz, and J. Protzenko, *Modularity, Code Specialization, and Zero-Cost Abstractions for Program Verification*, Proc. ACM Program. Lang. **7** (2023).

[317] R. Milner, *A theory of type polymorphism in programming*, Journal of computer and system sciences **17**, 348–375 (1978).

[318] A. Rastogi, G. Martínez, A. Fromherz, T. Ramananandro, and N. Swamy. *Programming and Proving with Indexed Effects*. Technical report, (2021).

[319] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin. *Dependent Types and Multi-Monadic Effects in F**. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, (2016).

[320] D. Ahman, C. Hriţcu, K. Maillard, G. Martínez, G. Plotkin, J. Protzenko, A. Rastogi, and N. Swamy. *Dijkstra Monads for Free*. In *ACM Symposium on Principles of Programming Languages (POPL)*, (2017).

[321] L. de Moura and N. Bjørner. *Z3: An efficient SMT solver*. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, (2008).

[322] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţcu, K. Bhargavan, C. Fournet, and N. Swamy, *Verified low-level programming embedded in F**, Proc. ACM Program. Lang. **1** (2017).

[323] P. Letouzey. *A new extraction for Coq*. In *International Workshop on Types for Proofs and Programs*, pages 200–219. Springer, (2002).

[324] A. Rossberg, C. Russo, and D. Dreyer, *F-ing modules*, Journal of functional programming **24**, 529–607 (2014).

[325] D. B. MacQueen. *Using dependent types to express modular structure*. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 277–286, (1986).

[326] S. Weeks. *Whole-program compilation in MLton*. In *Proceedings of the 2006 Workshop on ML*, volume 6, pages 1–1, (2006).

[327] K. Bhargavan, B. Bond, A. Delignat-Lavaud, C. Fournet, C. Hawblitzel, C. Hritcu, S. Ishtiaq, M. Kohlweiss, R. Leino, J. Lorch, K. Maillard, J. Pang, B. Parno, J. Protzenko, T. Ramananandro, A. Rane, A. Rastogi, N. Swamy, L. Thompson, P. Wang, S. Zanella-Beguelin, and J.-K. Zinzindohoué. *Everest: Towards a Verified Drop-In Replacement of HTTPS*. In *Proceedings of the Summit on Advances in Programming Languages (SNAPL)*, (2017).

[328] *A call-graph rewriting procedure for F**. https://github.com/hacl-star/hacl-star/blob/main/code/meta/Meta.Interface.fst, (2022).

[329] L. de Moura, S. Kong, J. Avigad, F. Van Doorn, and J. von Raumer. *The Lean theorem prover (system description)*. In *International Conference on Automated Deduction*, pages 378–388. Springer, (2015).

[330] E. Brady, *Idris, a general-purpose dependently typed programming language: Design and implementation*, Journal of functional programming **23**, 552–593 (2013).

[331] *Implementation in Meta-F* of the call-graph rewriting function*. https://github.com/hacl-star/hacl-star/blob/main/code/meta/Meta.Interface.fst, (2024).

[332] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand. *CompCert – A Formally Verified Optimizing Compiler*. In *Embedded Real Time Software and Systems (ERTS)*. SEE, (2016).

[333] C. Pit-Claudel, P. Wang, B. Delaware, J. Gross, and A. Chlipala. *Extensible Extraction of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and Proofs*. In N. Peltier and V. Sofronie-Stokkermans, editors, *Automated Reasoning: 10th International Joint Conference, IJCAR 2020, Paris, France, July 1–4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 119–137. Springer International Publishing, (2020).

[334] C. Pit-Claudel, J. Philipoom, D. Jamner, A. Erbsen, and A. Chlipala. *Relational compilation for performance-critical applications: extensible proof-producing translation of functional models into low-level code*. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 918–933, San Diego CA USA, (2022). ACM.

[335] P. Lammich, *Refinement to imperative HOL*, Journal of Automated Reasoning **62**, 481–503 (2019).

[336] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson. *Vale: Verifying High-Performance Cryptographic Assembly Code*. In *Proceedings of the USENIX Security Symposium*, (2017).

[337] A. Fromherz, N. Giannarakis, C. Hawblitzel, B. Parno, A. Rastogi, and N. Swamy, *A Verified, Efficient Embedding of a Verifiable Assembly Language*, Proc. ACM Program. Lang. **3** (2019).

[338] D. J. Bernstein. *Curve25519: New Diffie-Hellman Speed Records*. In *Proceedings of the IACR Conference on Practice and Theory of Public Key Cryptography (PKC)*, (2006).

[339] *HACL⋆ Github Repository*. https://github.com/hacl-star/hacl-star/tree/main, (2024).

[340] *Federal Information Processing Standards Publication 180-4: Secure hash standard (SHS)*, (2012). NIST.

[341] M. J. Dworkin. *SHA-3 standard: Permutation-based hash and extendable-output functions*. Technical report, (2015).

[342] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein. *BLAKE2: Simpler, Smaller, Fast as MD5*. In *Applied Cryptography and Network Security*, pages 119–135, (2013).

[343] M.-J. Saarinen and J.-P. Aumasson. *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)*. IETF RFC 7693, (2015).

[344] L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel. *Verified Correctness and Security of OpenSSL HMAC*. (2015).

[345] D. J. Bernstein. *The Poly1305-AES message-authentication code*. In *Proceedings of Fast Software Encryption*, (2005).

[346] D. A. McGrew and J. Viega. *The Security and Performance of the Galois/Counter Mode of Operation*. In *Proceedings of the International Conference on Cryptology in India (INDOCRYPT)*, (2004).

[347] I. T. Kassios. *Dynamic frames: Support for framing, dependencies and sharing without restrictions*. In *Proceedings of the 14th International Symposium on Formal Methods (FM)*, (2006).

[348] G. Vranken. *CryptoFuzz*. https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=14822#c3, (2019).

[349] J.-O. Kaiser, B. Ziliani, R. Krebbers, Y. Régis-Gianas, and D. Dreyer, *Mtac2: typed tactics for backward reasoning in Coq*, Proceedings of the ACM on Programming Languages **2**, 1–31 (2018).

[350] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub. *Jasmin: High-Assurance and High-Speed Cryptography*. (2017).

[351] J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, V. Laporte, T. Oliveira, and P. Strub. *The Last Mile: High-Assurance and High-Speed Cryptographic Implementations*. In *IEEE Symposium on Security and Privacy*, pages 965–982, (2020).

[352] A. Anand, A. Appel, G. Morrisett, Z. Paraskevopoulou, R. Pollack, O. S. Belanger, M. Sozeau, and M. Weaver. *CertiCoq: A verified compiler for Coq*. In *3rd International Workshop on Coq for Programming Languages (CoqPL)*, (2017).

[353] A. Chlipala. *The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier*. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, (2013).

[354] A. Erbsen, S. Gruetter, J. Choi, C. Wood, and A. Chlipala. *Integration verification across software and hardware for a simple embedded system*. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 604–619, Virtual Canada, (2021). ACM.

[355] A. W. Appel, *Verification of a cryptographic primitive: SHA-256*, ACM Transactions on Programming Languages and Systems **37**, 7 (2015).

[356] A. W. Appel. *Verified Software Toolchain*. In *Proceedings of the European Conference on Programming Languages and Systems (ESOP/ETAPS)*, (2011).

[357] Haskell-crypto. *cryptonite*. https://github.com/haskell-crypto/cryptonite, (2022).

[358] S. Ho and C. Pit-Claudel, *Incremental Proof Development in Dafny with Module-Based Induction*, ArXiv **abs/2401.16233** (2024).

[359] *Dafny Compiler Bootstrap: a work-in-progress reimplementation of Dafny's compilers, in Dafny*. https://github.com/dafny-lang/compiler-bootstrap, (2023).

[360] *Examples of Dafny induction principles*. https://zenodo.org/records/10553207, (2022).

[361] D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. Anderson. *Planning for Change in a Formal Verification of the Raft Consensus Protocol*. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, page 154–165, New York, NY, USA, (2016). Association for Computing Machinery.

[362] J. Chrząszcz. *Implementing Modules in the Coq System.* In D. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics*, pages 270–286, Berlin, Heidelberg, (2003). Springer Berlin Heidelberg.

[363] M. Sozeau and N. Oury. *First-Class Type Classes.* In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 278–293, Berlin, Heidelberg, (2008). Springer Berlin Heidelberg.

[364] K. R. M. Leino. *Automating Induction with an SMT Solver.* In *International Conference on Verification, Model Checking and Abstract Interpretation*, (2012).

[365] D. Miller, *Robert S. Boyer and J Strother Moore. A computational logic. ACM monograph series. Academic Press, New York etc. 1979, xiv 397 pp. - Robert S. Boyer and J Strother Moore. A computational logic handbook. Perspectives in computing, vol. 23. Academic Press, Boston etc. 1988, xvi 408 pp.*, The Journal of Symbolic Logic **55**, 1302–1304 (1990).

[366] M. Kaufmann, J. S. Moore, and P. Manolios, *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers, USA (2000).

[367] M. A. Dave, *Review of Rippling: Meta-Level Guidance for Mathematical Reasoning Cambridge Tracks in Theoretical Computer Science 56 by Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland*, SIGACT News **42**, 21–23 (2011).

[368] M. Johansson, L. Dixon, and A. Bundy. *Case-Analysis for Rippling and Inductive Proof.* volume 6172, pages 291–306, (2010).

[369] W. Sonnex, S. Drossopoulou, and S. Eisenbach. *Zeno : A tool for the automatic verification of algebraic properties of functional programs.* (2010).

[370] J.-F. Monin. *Proof Trick: Small Inversions.* In Y. Bertot, editor, *Second Coq Workshop*, Edinburgh, United Kingdom, (2010). Yves Bertot.

[371] J.-F. Monin and X. Shi. *Handcrafted Inversions Made Operational on Operational Semantics.* In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving*, pages 338–353, Berlin, Heidelberg, (2013). Springer Berlin Heidelberg.

[372] S. Dailler, C. Marché, and Y. Moy, *Lightweight Interactive Proving inside an Automatic Program Verifier*, Electronic Proceedings in Theoretical Computer Science **284**, 1–15 (2018).

[373] G. Melquiond and R. Rieu-Helft. *A Why3 Framework for Reflection Proofs and its Application to GMP's Algorithms*. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *9th International Joint Conference on Automated Reasoning*, number 10900 in Lecture Notes in Computer Science, pages 178–193, Oxford, United Kingdom, (2018).

[374] A. Chlipala, *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*, The MIT Press (2013).

[375] L. Paulson and J. Blanchette. *Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers*. (2015).

[376] Łukasz Czajka and C. Kaliszyk, *Hammer for Coq: Automation for Dependent Type Theory*, Journal of Automated Reasoning **61**, 423 – 453 (2018).

[377] C. Kaliszyk and J. Urban, *Learning-Assisted Automated Reasoning with Flyspeck (vol 53, pg 173, 2014)*, Journal of Automated Reasoning **54**, 99–99 (2015).

[378] K. R. M. Leino and C. Pit-Claudel. *Trigger Selection Strategies to Stabilize Program Verifiers*. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 361–381. Springer International Publishing, (2016).

[379] F. Bobot, J.-C. Filliâtre, C. Marché, G. Melquiond, and A. Paskevich. *Preserving User Proofs across Specification Changes*. In E. Cohen and A. Rybalchenko, editors, *Verified Software: Theories, Tools, Experiments*, pages 191–201, Berlin, Heidelberg, (2014). Springer Berlin Heidelberg.

[380] T. Ringer, N. Yazdani, J. Leo, and D. Grossman. *Adapting Proof Automation to Adapt Proofs*. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, page 115–129, New York, NY, USA, (2018). Association for Computing Machinery.

[381] T. Ringer, R. Porter, N. Yazdani, J. Leo, and D. Grossman. *Proof Repair across Type Equivalences*. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 112–127, New York, NY, USA, (2021). Association for Computing Machinery.

[382] N. Jeannerod, C. Marché, and R. Treinen. *A Formally Verified Interpreter for a Shell-like Programming Language*. In *VSTTE 2017 - 9th Working Conference*

on Verified Software: Theories, Tools, and Experiments, volume 10712 of *Lecture Notes in Computer Science*, Heidelberg, Germany, (2017).

[383] M. Clochard, J.-C. Filliâtre, C. Marché, and A. Paskevich. *Formalizing Semantics with an Automatic Program Verifier*. In D. Giannakopoulou and D. Kroening, editors, *6th Working Conference on Verified Software: Theories, Tools and Experiments (VSTTE)*, volume 8471 of *Lecture Notes in Computer Science*, Vienna, Austria, (2014). Springer.

[384] B. Becker and C. Marché. *Ghost Code in Action: Automated Verification of a Symbolic Interpreter*. In S. Chakraborty and J. A.Navas, editors, *VSTTE 2019 - 11th Working Conference on Verified Software: Tools, Techniques and Experiments*, volume 12031 of *Lecture Notes in Computer Science*, New York, United States, (2019).

[385] StackOverflow. *2023 Developer Survey*. https://survey.stackoverflow.co/2023/#section-admired-and-desired-programming-scripting-and-markup-languages, (2023).

[386] National Security Agency. *Software Memory Safety*. https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF, (2022).

[387] The Register. *In Rust We Trust: Microsoft Azure CTO shuns C and C++*. https://www.theregister.com/2022/09/20/rust_microsoft_c/, (2022).

[388] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. *Leveraging Rust Types for Modular Specification and Verification*. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 3, pages 147:1–147:30. ACM, (2019).

[389] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel. *Verus: Verifying Rust Programs using Linear Ghost Types*. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, (2023).

[390] F. Kiefer and L. Franceschino. *Introducing hax*. https://hacspec.org/blog/posts/hax-v0-1/, (2023).

[391] U. Juhasz, I. T. Kassios, P. Müller, M. Novacek, M. Schwerhoff, and A. J. Summers. *Viper: A verification infrastructure for permission-based reasoning*. Technical report, ETH Zurich, (2014).

[392] B. Blanchet. *An Efficient Cryptographic Protocol Verifier Based on Prolog Rules*. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, (2001). IEEE Computer Society. This paper received a **test of time award** at the CSF'23 conference.

[393] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, *RustBelt: Securing the foundations of the Rust programming language*, Proceedings of the ACM on Programming Languages **2**, 1–34 (2017).

[394] The Miri Team. *Miri*. https://github.com/rust-lang/miri/, (2021).

[395] R. Jung, H.-H. Dang, J. Kang, and D. Dreyer, *Stacked borrows: an aliasing model for Rust*, Proceedings of the ACM on Programming Languages **4**, 1–32 (2019).

[396] N. Villani. *Tree Borrows, a new aliasing model for Rust*. https://perso.crans.org/vanille/treebor/, (2023).

[397] Y. Matsushita, X. Denis, J.-H. Jourdan, and D. Dreyer. *RustHorn-Belt: A semantic foundation for functional verification of Rust programs with unsafe code*. In *Proceedings of the 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (2022).

[398] The Rust Compiler Team. *The Polonius Book*. https://rust-lang.github.io/polonius/, (2021).

[399] S. Ho and J. Protzenko, *Aeneas: Rust verification by functional translation*, Proc. ACM Program. Lang. **6** (2022).

[400] S. Ho, A. Fromherz, and J. Protzenko, *Sound Borrow-Checking for Rust via Symbolic Semantics*, Proc. ACM Program. Lang. **8** (2024).

[401] N. Matsakis. *Regions are Sets of Loans*. http://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/, (2018).

[402] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. *Boomerang: resourceful lenses for string data*. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 407–419, (2008).

[403] T. Balabonski, F. Pottier, and J. Protzenko, *The design and formalization of Mezzo, a permission-based programming language*, ACM Transactions on Programming Languages and Systems (TOPLAS) **38**, 1–94 (2016).

[404] The Rust Compiler Team. *Guide to rustc development.* https://rustc-dev-guide.rust-lang.org/borrow_check/two_phase_borrows.html, (2022).

[405] X. Leroy, *Formal verification of a realistic compiler*, Communications of the ACM (CACM) **52**, 107–115 (2009).

[406] X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. *The CompCert Memory Model, Version 2.* Research report RR-7987, INRIA, (2012).

[407] X. Leroy. *Coinductive big-step operational semantics.* In *Proceedings of the European Symposium on Programming (ESOP)*, pages 54–68. Springer, (2006).

[408] K. Nakata and T. Uustalu. *Trace-based coinductive operational semantics for While: big-step and small-step, relational and functional styles.* In *International Conference on Theorem Proving in Higher Order Logics*, pages 375–390. Springer, (2009).

[409] S. Owens, M. O. Myreen, R. Kumar, and Y. K. Tan. *Functional big-step semantics.* In *Proceedings of the European Symposium on Programming*, pages 589–615. Springer, (2016).

[410] P. O'Hearn, J. Reynolds, and H. Yang. *Local reasoning about programs that alter data structures.* In *International Workshop on Computer Science Logic (CSL)*, pages 1–19. Springer, (2001).

[411] B.-Y. E. Chang and X. Rival. *Relational inductive shape analysis.* In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, (2008).

[412] V. Laviron, B.-Y. E. Chang, and X. Rival. *Separating shape graphs.* In *Proceedings of the European Symposium on Programming (ESOP)*, pages 387–406. Springer, (2010).

[413] *Eurydice Git Repository.* git@github.com:AeneasVerif/eurydice.git.

[414] *Kani Githuh Repository.* git@github.com:model-checking/kani.git.

[415] *Kani PR for the LLBC backend.* https://github.com/model-checking/kani/pull/3514.

[416] Y. Bertot and V. Komendantsky. *Fixed point semantics and partial recursion in Coq.* In *Proceedings of the 10th International ACM SIGPLAN Conference on*

Principles and Practice of Declarative Programming, PPDP '08, page 89–96, New York, NY, USA, (2008). Association for Computing Machinery.

[417] D. Nowak and V. Rusu, *While Loops in Coq*, Electronic Proceedings in Theoretical Computer Science **389**, 96–109 (2023).

[418] J. Limperg and A. H. From. *Aesop: White-Box Best-First Proof Search for Lean.* In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2023, page 253–266, New York, NY, USA, (2023). Association for Computing Machinery.

[419] L. de Moura and N. Bjørner. *Efficient E-Matching for SMT Solvers.* In F. Pfenning, editor, *Automated Deduction – CADE-21*, pages 183–198, Berlin, Heidelberg, (2007). Springer Berlin Heidelberg.

[420] A. Guéneau, M. O. Myreen, R. Kumar, and M. Norrish. *Verified Characteristic Formulae for CakeML.* In *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings*, page 584–610, Berlin, Heidelberg, (2017). Springer-Verlag.

[421] V. D'Silva, D. Kroening, and G. Weissenbacher, *A Survey of Automated Techniques for Formal Software Verification*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **27**, 1165–1178 (2008).

[422] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno. *SoK: Computer-Aided Cryptography.* Cryptology ePrint Archive, Paper 2019/1393, (2019).

[423] T. Kulik, B. Dongol, P. G. Larsen, H. D. Macedo, S. Schneider, P. W. V. Tran-Jørgensen, and J. Woodcock, *A Survey of Practical Formal Methods for Security*, Form. Asp. Comput. **34** (2022).

[424] N. A. Davis, T. E. Berger, A. McDonald, J. B. Ingram, J. D. Foster, and K. Sanchez. *Software Verification Toolkit (SVT): Survey on Available Software Verification Tools and Future Direction.* Technical report, Sandia National Lab. (SNL-NM), Albuquerque, NM (United States), (2022).

[425] L. d. Moura, S. Kong, J. Avigad, F. v. Doorn, and J. v. Raumer. *The Lean theorem prover (system description).* In *International Conference on Automated Deduction*, pages 378–388. Springer, (2015).

[426] D. Merigoux, F. Kiefer, and K. Bhargavan. *Hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust*. Technical report, Inria, (2021).

[427] *Hacspec Git Repository*. https://github.com/hacspec/hacspec.

[428] *Bertie Github Repository - Hacspec Implementation of TLS*. https://github.com/cryspen/bertie.

[429] T. H. D. Team. *Hax*. https://github.com/hacspec/hax, (2024).

[430] *Hax Git Repository - ML KEM Implementation*. https://github.com/cryspen/libcrux/tree/main/libcrux-ml-kem.

[431] X. Denis and J.-H. Jourdan. *Specifying and Verifying Higher-order Rust Iterators*. In S. Sankaranarayanan and N. Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–110, Cham, (2023). Springer Nature Switzerland.

[432] S. Høverstad Skotåm. *CreuSAT, Using Rust and Creusot to create the world's fastest deductively verified SAT solver*. PhD thesis, University of Oslo, (2022).

[433] X. Denis. *Deductive Verification of Rust Programs*. Theses, Université Paris-Saclay, (2023).

[434] X. Sun, W. Ma, J. T. Gu, Z. Ma, T. Chajed, J. Howell, A. Lattuada, O. Padon, L. Suresh, A. Szekeres, and T. Xu. *Anvil: Verifying Liveness of Cluster Management Controllers*. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 649–666, Santa Clara, CA, (2024). USENIX Association.

[435] Z. Zhou, Anjali, W. Chen, S. Gong, C. Hawblitzel, and W. Cui. *VeriSMo: A Verified Security Module for Confidential VMs*. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 599–614, Santa Clara, CA, (2024). USENIX Association.

[436] E. Johnson, E. Laufer, Z. Zhao, D. Gohman, S. Narayan, S. Savage, D. Stefan, and F. Brown. *WaVe: a verifiably secure WebAssembly sandboxing runtime*. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2940–2955, (2023).

[437] N. Lehmann, A. Geller, N. Vazou, and R. Jhala. *Flux: Liquid Types for Rust*, (2022).

[438] P. M. Rondon, M. Kawaguci, and R. Jhala, *Liquid types*, SIGPLAN Not. **43**, 159–169 (2008).

[439] D. Dill, W. Grieskamp, J. Park, S. Qadeer, M. Xu, and E. Zhong. *Fast and Reliable Formal Verification of Smart Contracts with the Move Prover*. In D. Fisman and G. Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 183–200, Cham, (2022). Springer International Publishing.

[440] S. Blackshear, E. Cheng, D. L. Dill, V. Gao, B. Maurer, T. Nowacki, A. Pott, S. Qadeer, D. Rain, S. Russi, T. Sezer, R. Zakian, and Zhou. *Move: A Language With Programmable Resources*. (2019).

[441] S. Élie Ayoun, X. Denis, P. Maksimović, and P. Gardner. *A hybrid approach to semi-automated Rust verification*, (2024).

[442] J. Fragoso Santos, P. Maksimovic, S. Ayoun, and P. Gardner. *Gillian, Part I: A Multi-language Platform for Symbolic Execution*. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*, (2020).

[443] M. Sammler, R. Lepigre, R. Krebbers, K. Memarian, D. Dreyer, and D. Garg. *RefinedC: automating the foundational verification of C code with refined ownership types*. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 158–174, (2021).

[444] J. Protzenko. *Mezzo: a typed language for safe effectful concurrent programs*. PhD thesis, Université Paris Diderot-Paris 7, (2014).

[445] L. Gäher, M. Sammler, R. Jung, R. Krebbers, and D. Dreyer, *RefinedRust: A Type System for High-Assurance Verification of Rust Programs*, Proc. ACM Program. Lang. **8** (2024).

[446] F. Wang, F. Song, M. Zhang, X. Zhu, and J. Zhang. *KRust: A Formal Executable Semantics of Rust*. In *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 44–51, (2018).

[447] S. Kan, Z. Chen, D. Sanan, S.-W. Lin, and Y. Liu. *An Executable Operational Semantics for Rust with the Formalization of Ownership and Borrowing*, (2020).

[448] D. J. Pearce, *A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust*, ACM Trans. Program. Lang. Syst. **43** (2021).

[449] A. Charguéraud and F. Pottier. *Functional translation of a calculus of capabilities.* In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, pages 213–224, (2008).

[450] F. Pottier and J. Protzenko, *Programming with permissions in Mezzo*, ACM SIGPLAN Notices **48**, 173–184 (2013).

[451] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. *Compositional shape analysis by means of bi-abduction.* In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 289–300, (2009).

[452] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'hearn, T. Wies, and H. Yang. *Shape analysis for composite data structures.* In *Computer Aided Verification: 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007. Proceedings 19*, pages 178–192. Springer, (2007).

[453] D. Distefano, P. W. O'hearn, and H. Yang. *A local shape analysis based on separation logic.* In *Tools and Algorithms for the Construction and Analysis of Systems: 12th International Conference, TACAS 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25-April 2, 2006. Proceedings 12*, pages 287–302. Springer, (2006).

[454] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. *Moving fast with software verification.* In *Proceedings of the NASA Formal Methods Symposium (NFM)*, pages 3–11. Springer, (2015).

[455] H. Illous, M. Lemerre, and X. Rival. *A relational shape abstract domain.* In *Proceedings of the NASA Formal Methods Symposium (NFM)*, pages 212–229. Springer, (2017).

[456] P. He, E. Westbrook, B. Carmer, C. Phifer, V. Robert, K. Smeltzer, A. Ştefănescu, A. Tomb, A. Wick, M. Yacavone, and S. Zdancewic, *A Type System for Extracting Functional Specifications from Memory-Safe Imperative Programs*, Proc. ACM Program. Lang. **5** (2021).

[457] P. He. *Rely-Guarantee Semantics for Separation-Logic-Based Specification Extraction.* PhD thesis, University of Pennsylvania, (2024).

[458] A. Weiss, O. Gierczak, D. Patterson, and A. Ahmed. *Oxide: The Essence of Rust*, (2021).

[459] G. Mével, J.-H. Jourdan, and F. Pottier, *Cosmo: a concurrent separation logic for multicore OCaml*, Proc. ACM Program. Lang. **4** (2020).

[460] A. Moine, S. Westrick, and S. Balzer, *DisLog: A Separation Logic for Disentanglement*, Proc. ACM Program. Lang. **8** (2024).

[461] J. Lee, Y. Kim, Y. Song, C.-K. Hur, S. Das, D. Majnemer, J. Regehr, and N. P. Lopes, *Taming undefined behavior in LLVM*, SIGPLAN Not. **52**, 633–647 (2017).

Appendix A

Authentication and Confidentiality Levels for 59 Noise Protocols

Prot. Name	Message Sequence	Payload Security Properties ← A	← C	→ A	→ C
N	(premessages) → e, es $[d_0]$ → $[d_1, d_2, ...]$	 - -	 - -	 A0 A0	 C2 C2
K	(premessages) → e, es, ss $[d_0]$ → $[d_1, d_2, ...]$	 - -	 - -	 A1 A1	 C2 C2
X	(premessages) → e, es, s, ss $[d_0]$ → $[d_1, d_2, ...]$	 - -	 - -	 A1 A1	 C2 C2
NN	→ e $[d_0]$ ← e, ee $[d_1]$ ↔ $[d_2, d_3, ...]$	A0 A0 A0	C0 C1 C1	A0 A0 A0	C0 C0 C1
KN	(premessages) → e $[d_0]$ ← e, ee, se $[d_1]$ → $[d_2]$ ↔ $[d_3, d_4, ...]$	 A0 A0 A0 A0	 C0 C3 C3 C5	 A0 A0 A2 A2	 C0 C0 C1 C1
NK	(premessages) → e, es $[d_0]$ ← e, ee $[d_1]$ ↔ $[d_2, d_3, ...]$	 A0 A2 A2	 C0 C1 C1	 A0 A0 A0	 C2 C2 C5
KK	(premessages) → e, es, ss $[d_0]$ ← e, ee, se $[d_1]$ → $[d_2]$ ↔ $[d_3, d_4, ...]$	 A0 A2 A2 A2	 C0 C4 C4 C5	 A1 A1 A2 A2	 C2 C2 C5 C5
NX	→ e $[d_0]$ ← e, ee, s, es $[d_1]$ ↔ $[d_2, d_3, ...]$	A0 A2 A2	C0 C1 C1	A0 A0 A0	C0 C0 C5
KX	(premessages) → e $[d_0]$ ← e, ee, se, s, es $[d_1]$ → $[d_2]$ ↔ $[d_3, d_4, ...]$	 A0 A2 A2 A2	 C0 C3 C3 C5	 A0 A0 A2 A2	 C0 C0 C5 C5
XN	→ e $[d_0]$ ← e, ee $[d_1]$ → s, se $[d_2]$ ↔ $[d_3, d_4, ...]$	A0 A0 A0 A0	C0 C1 C1 C5	A0 A0 A2 A2	C0 C0 C1 C1
IN	→ e, s $[d_0]$ ← e, ee, se $[d_1]$ → $[d_2]$ ↔ $[d_3, d_4, ...]$	A0 A0 A0 A0	C0 C3 C3 C5	A0 A0 A2 A2	C0 C0 C1 C1

Prot. Name	Message Sequence	← A	← C	→ A	→ C
XK	(premessages) → e, es $[d_0]$ ← e, ee $[d_1]$ → s, se $[d_2]$ ↔ $[d_3, d_4, ...]$	 A0 A2 A2 A2	 C0 C1 C1 C5	 A0 A0 A2 A2	 C2 C2 C5 C5
IK	(premessages) → e, es, s, ss $[d_0]$ ← e, ee, se $[d_1]$ → $[d_2]$ ↔ $[d_3, d_4, ...]$	 A0 A2 A2 A2	 C0 C4 C4 C5	 A1 A1 A2 A2	 C2 C2 C5 C5
XX	→ e $[d_0]$ ← e, ee, s, es $[d_1]$ → s, se $[d_2]$ ↔ $[d_3, d_4, ...]$	A0 A2 A2 A2	C0 C1 C1 C5	A0 A0 A2 A2	C0 C0 C5 C5
IX	→ e, s $[d_0]$ ← e, ee, se, s, es $[d_1]$ → $[d_2]$ ↔ $[d_3, d_4, ...]$	A0 A2 A2 A2	C0 C3 C3 C5	A0 A0 A2 A2	C0 C0 C5 C5
Npsk0	(premessages) → psk, e, es $[d_0]$ → $[d_1, d_2, ...]$	 - -	 - -	 A1 A1	 C2 C2
Kpsk0	(premessages) → psk, e, es, ss $[d_0]$ → $[d_1, d_2, ...]$	 - -	 - -	 A1 A1	 C2 C2
Xpsk1	(premessages) → e, es, s, ss, psk $[d_0]$ → $[d_1, d_2, ...]$	 - -	 - -	 A1 A1	 C2 C2
NNpsk0	→ psk, e $[d_0]$ ← e, ee $[d_1]$ ↔ $[d_2, d_3, ...]$	A0 A1 A1	C0 C1 C1	A1 A1 A1	C0 C0 C1
NNpsk2	→ e $[d_0]$ ← e, ee, psk $[d_1]$ ↔ $[d_2, d_3, ...]$	A0 A1 A1	C0 C1 C1	A0 A0 A1	C0 C0 C1
NKpsk0	(premessages) → psk, e, es $[d_0]$ ← e, ee $[d_1]$ ↔ $[d_2, d_3, ...]$	 A0 A2 A2	 C0 C1 C1	 A1 A1 A1	 C2 C2 C5
NKpsk2	(premessages) → e, es $[d_0]$ ← e, ee, psk $[d_1]$ ↔ $[d_2, d_3, ...]$	 A0 A2 A2	 C0 C1 C1	 A0 A0 A1	 C2 C2 C5
NXpsk2	→ e $[d_0]$ ← e, ee, s, es, psk $[d_1]$ ↔ $[d_2, d_3, ...]$	A0 A2 A2	C0 C1 C1	A0 A0 A1	C0 C0 C5

Prot. Name	Message Sequence	← A	← C	→ A	→ C
XNpsk3	→ e $[d_0]$	A0	C0	A0	C0
	← e, ee $[d_1]$	A0	C1	A0	C0
	→ s, se, psk $[d_2]$	A0	C1	A2	C1
	↔ $[d_3, d_4, \ldots]$	A1	C5	A2	C1
XKpsk3	(premessages)				
	→ e, es $[d_0]$	A0	C0	A0	C2
	← e, ee $[d_1]$	A2	C1	A0	C2
	→ s, se, psk $[d_2]$	A2	C1	A2	C5
	↔ $[d_3, d_4, \ldots]$	A2	C5	A2	C5
XXpsk3	→ e $[d_0]$	A0	C0	A0	C0
	← e, ee, s, es $[d_1]$	A2	C1	A0	C0
	→ s, se, psk $[d_2]$	A2	C1	A2	C5
	↔ $[d_3, d_4, \ldots]$	A2	C5	A2	C5
KNpsk0	(premessages)				
	→ psk, e $[d_0]$	A0	C0	A1	C0
	← e, ee, se $[d_1]$	A1	C3	A1	C0
	→ $[d_2]$	A1	C3	A2	C1
	↔ $[d_3, d_4, \ldots]$	A1	C5	A2	C1
KNpsk2	(premessages)				
	→ e $[d_0]$	A0	C0	A0	C0
	← e, ee, se, psk $[d_1]$	A1	C3	A0	C0
	→ $[d_2]$	A1	C3	A2	C1
	↔ $[d_3, d_4, \ldots]$	A1	C5	A2	C1
KKpsk0	(premessages)				
	→ psk, e, es, ss $[d_0]$	A0	C0	A1	C2
	← e, ee, se $[d_1]$	A2	C4	A1	C2
	→ $[d_2]$	A2	C4	A2	C5
	↔ $[d_3, d_4, \ldots]$	A2	C5	A2	C5
KKpsk2	(premessages)				
	→ e, es, ss $[d_0]$	A0	C0	A1	C2
	← e, ee, se, psk $[d_1]$	A2	C4	A1	C2
	→ $[d_2]$	A2	C4	A2	C5
	↔ $[d_3, d_4, \ldots]$	A2	C5	A2	C5
KXpsk2	(premessages)				
	→ e $[d_0]$	A0	C0	A0	C0
	← e, ee, se, s, es, psk $[d_1]$	A2	C3	A0	C0
	→ $[d_2]$	A2	C3	A2	C5
	↔ $[d_3, d_4, \ldots]$	A2	C5	A2	C5
INpsk1	→ e, s, psk $[d_0]$	A0	C0	A1	C0
	← e, ee, se $[d_1]$	A1	C3	A1	C0
	→ $[d_2]$	A1	C3	A2	C1
	↔ $[d_3, d_4, \ldots]$	A1	C5	A2	C1
INpsk2	→ e, s $[d_0]$	A0	C0	A0	C0
	← e, ee, se, psk $[d_1]$	A1	C3	A0	C0
	→ $[d_2]$	A1	C3	A2	C1
	↔ $[d_3, d_4, \ldots]$	A1	C5	A2	C1
IKpsk1	(premessages)				
	→ e, es, s, ss, psk $[d_0]$	A0	C0	A1	C2
	← e, ee, se $[d_1]$	A2	C4	A1	C2
	→ $[d_2]$	A2	C4	A2	C5
	↔ $[d_3, d_4, \ldots]$	A2	C5	A2	C5
IKpsk2	(premessages)				
	→ e, es, s, ss $[d_0]$	A0	C0	A1	C2
	← e, ee, se, psk $[d_1]$	A2	C4	A1	C2
	→ $[d_2]$	A2	C4	A2	C5
	↔ $[d_3, d_4, \ldots]$	A2	C5	A2	C5
IXpsk2	→ e, s $[d_0]$	A0	C0	A0	C0
	← e, ee, se, s, es, psk $[d_1]$	A2	C3	A0	C0
	→ $[d_2]$	A2	C3	A2	C5
	↔ $[d_3, d_4, \ldots]$	A2	C5	A2	C5
NK1	(premessages)				
	→ e $[d_0]$	A0	C0	A0	C0
	← e, ee, es $[d_1]$	A2	C1	A0	C0
	↔ $[d_2, d_3, \ldots]$	A2	C1	A0	C5
NX1	→ e $[d_0]$	A0	C0	A0	C0
	← e, ee, s $[d_1]$	A0	C1	A0	C0
	→ es $[d_2]$	A0	C1	A0	C3
	← $[d_3]$	A2	C1	A0	C3
	↔ $[d_4, d_5, \ldots]$	A2	C1	A0	C5

Prot. Name	Message Sequence	← A	← C	→ A	→ C
X1N	→ e $[d_0]$	A0	C0	A0	C0
	← e, ee $[d_1]$	A0	C1	A0	C0
	→ s $[d_2]$	A0	C1	A0	C1
	← se $[d_3]$	A0	C3	A0	C1
	→ $[d_4]$	A0	C3	A2	C1
	↔ $[d_5, d_6, \ldots]$	A0	C5	A2	C1
X1K	(premessages)				
	→ e, es $[d_0]$	A0	C0	A0	C2
	← e, ee $[d_1]$	A2	C1	A0	C2
	→ s $[d_2]$	A2	C1	A0	C5
	← se $[d_3]$	A2	C3	A0	C5
	→ $[d_4]$	A2	C3	A2	C5
	↔ $[d_5, d_6, \ldots]$	A2	C5	A2	C5
XK1	(premessages)				
	→ e $[d_0]$	A0	C0	A0	C0
	← e, ee, es $[d_1]$	A2	C1	A0	C0
	→ s, se $[d_2]$	A2	C1	A2	C5
	↔ $[d_3, d_4, \ldots]$	A2	C5	A2	C5
X1K1	(premessages)				
	→ e $[d_0]$	A0	C0	A0	C0
	← e, ee, es $[d_1]$	A2	C1	A0	C0
	→ s $[d_2]$	A2	C1	A0	C5
	← se $[d_3]$	A2	C3	A0	C5
	→ $[d_4]$	A2	C3	A2	C5
	↔ $[d_5, d_6, \ldots]$	A2	C5	A2	C5
X1X	→ e $[d_0]$	A0	C0	A0	C0
	← e, ee, s, es $[d_1]$	A2	C1	A0	C0
	→ s $[d_2]$	A2	C1	A0	C5
	← se $[d_3]$	A2	C3	A0	C5
	→ $[d_4]$	A2	C3	A2	C5
	↔ $[d_5, d_6, \ldots]$	A2	C5	A2	C5
XX1	→ e $[d_0]$	A0	C0	A0	C0
	← e, ee, s $[d_1]$	A0	C1	A0	C0
	→ es, s, se $[d_2]$	A0	C1	A2	C3
	← $[d_3]$	A2	C5	A2	C3
	↔ $[d_4, d_5, \ldots]$	A2	C5	A2	C5
X1X1	→ e $[d_0]$	A0	C0	A0	C0
	← e, ee, s $[d_1]$	A0	C1	A0	C0
	→ es, s $[d_2]$	A0	C1	A0	C3
	← se $[d_3]$	A2	C3	A0	C3
	→ $[d_4]$	A2	C3	A2	C5
	↔ $[d_5, d_6, \ldots]$	A2	C5	A2	C5
K1N	(premessages)				
	→ e $[d_0]$	A0	C0	A0	C0
	← e, ee $[d_1]$	A0	C1	A0	C0
	→ se $[d_2]$	A0	C1	A2	C1
	↔ $[d_3, d_4, \ldots]$	A0	C5	A2	C1
K1K	(premessages)				
	→ e, es $[d_0]$	A0	C0	A0	C2
	← e, ee $[d_1]$	A2	C1	A0	C2
	→ se $[d_2]$	A2	C1	A2	C5
	↔ $[d_3, d_4, \ldots]$	A2	C5	A2	C5
KK1	(premessages)				
	→ e $[d_0]$	A0	C0	A0	C0
	← e, ee, se, es $[d_1]$	A2	C3	A0	C0
	→ $[d_2]$	A2	C3	A2	C5
	↔ $[d_3, d_4, \ldots]$	A2	C5	A2	C5
K1K1	(premessages)				
	→ e $[d_0]$	A0	C0	A0	C0
	← e, ee, es $[d_1]$	A2	C1	A0	C0
	→ se $[d_2]$	A2	C1	A2	C5
	↔ $[d_3, d_4, \ldots]$	A2	C5	A2	C5
K1X	(premessages)				
	→ e $[d_0]$	A0	C0	A0	C0
	← e, ee, s, es $[d_1]$	A2	C1	A0	C0
	→ se $[d_2]$	A2	C1	A2	C5
	↔ $[d_3, d_4, \ldots]$	A2	C5	A2	C5

Prot. Name	Message Sequence	←		→	
		A	C	A	C
KX1	(premessages)				
	→ e $[d_0]$	A0	C0	A0	C0
	← e, ee, se, s $[d_1]$	A0	C3	A0	C0
	→ es $[d_2]$	A0	C3	A2	C3
	← $[d_3]$	A2	C5	A2	C3
	↔ $[d_4, d_5, ...]$	A2	C5	A2	C5
K1X1	(premessages)				
	→ e $[d_0]$	A0	C0	A0	C0
	← e, ee, s $[d_1]$	A0	C1	A0	C0
	→ se, es $[d_2]$	A0	C1	A2	C3
	← $[d_3]$	A2	C5	A2	C3
	↔ $[d_4, d_5, ...]$	A2	C5	A2	C5
I1N	→ e, s $[d_0]$	A0	C0	A0	C0
	← e, ee $[d_1]$	A0	C1	A0	C0
	→ se $[d_2]$	A0	C1	A2	C1
	↔ $[d_3, d_4, ...]$	A0	C5	A2	C1
I1K	(premessages)				
	→ e, es, s $[d_0]$	A0	C0	A0	C2
	← e, ee $[d_1]$	A2	C1	A0	C2
	→ se $[d_2]$	A2	C1	A2	C5
	↔ $[d_3, d_4, ...]$	A2	C5	A2	C5
IK1	(premessages)				
	→ e, s $[d_0]$	A0	C0	A0	C0
	← e, ee, se, es $[d_1]$	A2	C3	A0	C0
	→ $[d_2]$	A2	C3	A2	C5
	↔ $[d_3, d_4, ...]$	A2	C5	A2	C5
I1K1	(premessages)				
	→ e, s $[d_0]$	A0	C0	A0	C0
	← e, ee, es $[d_1]$	A2	C1	A0	C0
	→ se $[d_2]$	A2	C1	A2	C5
	↔ $[d_3, d_4, ...]$	A2	C5	A2	C5
I1X	→ e, s $[d_0]$	A0	C0	A0	C0
	← e, ee, s, es $[d_1]$	A2	C1	A0	C0
	→ se $[d_2]$	A2	C1	A2	C5
	↔ $[d_3, d_4, ...]$	A2	C5	A2	C5
IX1	→ e, s $[d_0]$	A0	C0	A0	C0
	← e, ee, se, s $[d_1]$	A0	C3	A0	C0
	→ es $[d_2]$	A0	C3	A2	C3
	← $[d_3]$	A2	C5	A2	C3
	↔ $[d_4, d_5, ...]$	A2	C5	A2	C5
I1X1	→ e, s $[d_0]$	A0	C0	A0	C0
	← e, ee, s $[d_1]$	A0	C1	A0	C0
	→ se, es $[d_2]$	A0	C1	A2	C3
	← $[d_3]$	A2	C5	A2	C3
	↔ $[d_4, d_5, ...]$	A2	C5	A2	C5

Appendix B

Authentication and Confidentiality Target Security Labels for 59 Noise Protocols

Prot.	Message Seq.	Stage	Initiator Handshake State Label l_i	$l_i^{\leftarrow}$	Responder Handshake State Label l_r	$l_r^{\rightarrow}$	Payload Security Properties ← Auth	← Conf	→ Auth	→ Conf
NN	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	0	1	0	0
	↔ $[d_2, d_3, ...]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	0	1	0	1
KN	→ s	pre								
	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, se $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[1]$	0	3	0	0
	→ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	0	3	2	1
	↔ $[d_3, d_4, ...]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	0	5	2	1
NK	← s	pre								
	→ e, es $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	-	(CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	0	0	0	2
	← e, ee $[d_1]$	2	$l_i[1]$ ⊓ (CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[1]$ ⊓ (CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	2
	↔ $[d_2, d_3, ...]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	1	0	5

Prot.	Message Seq.	Stage	Initiator Handshake State Label l_i	$l_i^{\leftarrow}$	Responder Handshake State Label l_r	$l_r^{\rightarrow}$	Payload Security Properties ← Auth	← Conf	→ Auth	→ Conf
KK	→ s	pre								
	← s	pre								
	→ e, es, ss $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) ⊓ (CanRead [P idx_i.p; P idx_i.peer])	-	(CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	0	0	1	2
	← e, ee, se $[d_1]$	2	$l_i[1]$ ⊓ (CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[1]$ ⊓ (CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[1]$	2	4	1	2
	→ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	4	2	5
	↔ $[d_3, d_4, ...]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	5	2	5
NX	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, s, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	0
	↔ $[d_2, d_3, ...]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	1	0	5
KX	→ s	pre								
	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, se, s, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	2	3	0	0
	→ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	3	2	5
	↔ $[d_3, d_4, ...]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	5	2	5
XN	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	0	1	0	0
	→ s, se $[d_2]$	3	$l_i[2]$ ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[2]$ ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	0	1	2	1
	↔ $[d_3, d_4, ...]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	0	5	2	1
IN	→ e, s $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, se $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[1]$	0	3	0	0
	→ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	0	3	2	1
	↔ $[d_3, d_4, ...]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	0	5	2	1
XK	← s	pre								
	→ e, es $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	-	(CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	0	0	0	2
	← e, ee $[d_1]$	2	$l_i[1]$ ⊓ (CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[1]$ ⊓ (CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	2
	→ s, se $[d_2]$	3	$l_i[2]$ ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[2]$ ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	2	1	2	5
	↔ $[d_3, d_4, ...]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
IK	← s	pre								
	→ e, es, s, ss $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) ⊓ (CanRead [P idx_i.p; P idx_i.peer])	-	(CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	0	0	1	2
	← e, ee, se $[d_1]$	2	$l_i[1]$ ⊓ (CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[1]$ ⊓ (CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[1]$	2	4	1	2
	→ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	4	2	5
	↔ $[d_3, d_4, ...]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	5	2	5
XX	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, s, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	0
	→ s, se $[d_2]$	3	$l_i[2]$ ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[2]$ ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	2	1	2	5
	↔ $[d_3, d_4, ...]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5

Prot.	Message Seq.	Stage	Initiator Handshake State Label l_i	$l_i^{\leftarrow}$	Responder Handshake State Label l_r	$l_r^{\rightarrow}$	Payload Security Properties ← Auth	← Conf	→ Auth	→ Conf
IX	→ e, s $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, se, s, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	2	3	0	0
	→ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	3	2	5
	↔ $[d_3, d_4, ...]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	5	2	5
N	← s	pre								
	→ e, es $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	-	(CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	-	-	0	2
	→ $[d_1, d_2, ...]$	2	$= l_i[1]$	$= l_i[1]$	$= l_r[1]$	$= l_r[1]$	-	-	0	2
K	→ s	pre								
	← s	pre								
	→ e, es, ss $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) ⊓ (CanRead [P idx_i.p; P idx_i.peer])	-	(CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	-	-	1	2
	→ $[d_1, d_2, ...]$	2	$= l_i[1]$	$= l_i[1]$	$= l_r[1]$	$= l_r[1]$	-	-	1	2
X	← s	pre								
	→ e, es, s, ss $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) ⊓ (CanRead [P idx_i.p; P idx_i.peer])	-	(CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	-	-	1	2
	→ $[d_1, d_2, ...]$	2	$= l_i[1]$	$= l_i[1]$	$= l_r[1]$	$= l_r[1]$	-	-	1	2
NNpsk0	→ psk, e $[d_0]$	1	(CanRead [P idx_i.p; P idx_i.peer])	-	(CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	0	0	1	0
	← e, ee $[d_1]$	2	$l_i[1]$ ⊓ (CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[1]$ ⊓ (CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	1	1	1	0
	↔ $[d_2, d_3, ...]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	1	1	1	1
NNpsk2	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, psk $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	1	1	0	0
	↔ $[d_2, d_3, ...]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	1	1	1	1
NKpsk0	← s	pre								
	→ psk, e, es $[d_0]$	1	(CanRead [P idx_i.p; P idx_i.peer]) ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	-	(CanRead [P idx_r.p; P idx_r.peer]) ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	0	0	1	2
	← e, ee $[d_1]$	2	$l_i[1]$ ⊓ (CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[1]$ ⊓ (CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	2	1	1	2
	↔ $[d_2, d_3, ...]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	1	1	5
NKpsk2	← s	pre								
	→ e, es $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	-	(CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	0	0	0	2
	← e, ee, psk $[d_1]$	2	$l_i[1]$ ⊓ (CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p; P idx_i.peer])	$= l_i[2]$	$l_r[1]$ ⊓ (CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	2	1	0	2
	↔ $[d_2, d_3, ...]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	1	1	5
NXpsk2	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, s, es, psk $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) ⊓ (CanRead [P idx_i.p; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	2	1	0	0
	↔ $[d_2, d_3, ...]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	1	1	5
XNpsk3	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	0	1	0	0
	→ s, se, psk $[d_2]$	3	$l_i[2]$ ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p; P idx_i.peer])	$= l_i[2]$	$l_r[2]$ ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) ⊓ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[3]$	0	1	2	1
	↔ $[d_3, d_4, ...]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	1	5	2	1

Prot.	Message Seq.	Stage	Initiator Handshake State Label l_i	$l_i^{\leftarrow}$	Responder Handshake State Label l_r	$l_r^{\rightarrow}$	Payload Security Properties ← Auth	← Conf	→ Auth	→ Conf
XKpsk3	← s	pre								
	→ e, es $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	-	(CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	0	0	0	2
	← e, ee $[d_1]$	2	$l_i[1]$ ⊓ (CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[1]$ ⊓ (CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	2
	→ s, se, psk $[d_2]$	3	$l_i[2]$ ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p; P idx_i.peer])	$= l_i[2]$	$l_r[2]$ ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) ⊓ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[3]$	2	1	2	5
	↔ $[d_3, d_4, ...]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
XXpsk3	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, s, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	0
	→ s, se, psk $[d_2]$	3	$l_i[2]$ ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p; P idx_i.peer])	$= l_i[2]$	$l_r[2]$ ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) ⊓ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[3]$	2	1	2	5
	↔ $[d_3, d_4, ...]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
KNpsk0	→ s	pre								
	→ psk, e $[d_0]$	1	(CanRead [P idx_i.p; P idx_i.peer])	-	(CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	0	0	1	0
	← e, ee, se $[d_1]$	2	$l_i[1]$ ⊓ (CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[1]$ ⊓ (CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[1]$	1	3	1	0
	→ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	1	3	2	1
	↔ $[d_3, d_4, ...]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	1	5	2	1
KNpsk2	→ s	pre								
	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, se, psk $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) ⊓ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	1	3	0	0
	→ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	1	3	2	1
	↔ $[d_3, d_4, ...]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	1	5	2	1
KKpsk0	→ s	pre								
	← s	pre								
	→ psk, e, es, ss $[d_0]$	1	(CanRead [P idx_i.p; P idx_i.peer]) ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) ⊓ (CanRead [P idx_i.p; P idx_i.peer])	-	(CanRead [P idx_r.p; P idx_r.peer]) ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	0	0	1	2
	← e, ee, se $[d_1]$	2	$l_i[1]$ ⊓ (CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[1]$ ⊓ (CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[1]$	2	4	1	2
	→ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	4	2	5
	↔ $[d_3, d_4, ...]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	5	2	5
KKpsk2	→ s	pre								
	← s	pre								
	→ e, es, ss $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) ⊓ (CanRead [P idx_i.p; P idx_i.peer])	-	(CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	0	0	1	2
	← e, ee, se, psk $[d_1]$	2	$l_i[1]$ ⊓ (CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p; P idx_i.peer])	$= l_i[2]$	$l_r[1]$ ⊓ (CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) ⊓ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	2	4	1	2
	→ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	4	2	5
	↔ $[d_3, d_4, ...]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	5	2	5

Prot.	Message Seq.	Stage	Initiator Handshake State Label l_i	$l_i^{\leftarrow}$	Responder Handshake State Label l_r	$l_r^{\rightarrow}$	Payload Security Properties ← Auth	← Conf	→ Auth	→ Conf
KXpsk2	→ s	pre								
	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, se, s, es, psk $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) ⊓ (CanRead [P idx_i.p; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	2	3	0	0
	→ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	3	2	5
	↔ $[d_3, d_4, ...]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	5	2	5
INpsk1	→ e, s, psk $[d_0]$	1	(CanRead [P idx_i.p; P idx_i.peer])	-	(CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	0	0	1	0
	← e, ee, se $[d_1]$	2	$l_i[1]$ ⊓ (CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[1]$ ⊓ (CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[1]$	1	3	1	0
	→ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	1	3	2	1
	↔ $[d_3, d_4, ...]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	1	5	2	1
INpsk2	→ e, s $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, se, psk $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) ⊓ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	1	3	0	0
	→ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	1	3	2	1
	↔ $[d_3, d_4, ...]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	1	5	2	1
IKpsk1	← s	pre								
	→ e, es, s, ss, psk $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) ⊓ (CanRead [P idx_i.p; P idx_i.peer]) ⊓ (CanRead [P idx_i.p; P idx_i.peer])	-	(CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p; P idx_r.peer]) ⊓ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	0	0	1	2
	← e, ee, se $[d_1]$	2	$l_i[1]$ ⊓ (CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[1]$ ⊓ (CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[1]$	2	4	1	2
	→ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	4	2	5
	↔ $[d_3, d_4, ...]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	5	2	5
IKpsk2	← s	pre								
	→ e, es, s, ss $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) ⊓ (CanRead [P idx_i.p; P idx_i.peer])	-	(CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	0	0	1	2
	← e, ee, se, psk $[d_1]$	2	$l_i[1]$ ⊓ (CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p; P idx_i.peer])	$= l_i[2]$	$l_r[1]$ ⊓ (CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) ⊓ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	2	4	1	2
	→ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	4	2	5
	↔ $[d_3, d_4, ...]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	5	2	5
IXpsk2	→ e, s $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, se, s, es, psk $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) ⊓ (CanRead [P idx_i.p; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	2	3	0	0
	→ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	3	2	5
	↔ $[d_3, d_4, ...]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	5	2	5
Npsk0	← s	pre								
	→ psk, e, es $[d_0]$	1	(CanRead [P idx_i.p; P idx_i.peer]) ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	-	(CanRead [P idx_r.p; P idx_r.peer]) ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	-	-	1	2
	→ $[d_1, d_2, ...]$	2	$= l_i[1]$	$= l_i[1]$	$= l_r[1]$	$= l_r[1]$	-	-	1	2

Prot.	Message Seq.	Stage	Initiator Handshake State Label l_i	$l_i^{\leftarrow}$	Responder Handshake State Label l_r	$l_r^{\rightarrow}$	Payload Security Properties ← Auth	← Conf	→ Auth	→ Conf
Kpsk0	→ s	pre								
	← s	pre								
	→ psk, e, es, ss $[d_0]$	1	(CanRead [P idx_i.p; P idx_i.peer]) ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) ⊓ (CanRead [P idx_i.p; P idx_i.peer])	-	(CanRead [P idx_r.p; P idx_r.peer]) ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	-	-	1	2
	→ $[d_1, d_2, ...]$	2	$= l_i[1]$	$= l_i[1]$	$= l_r[1]$	$= l_r[1]$	-	-	1	2
Xpsk1	← s	pre								
	→ e, es, s, ss, psk $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) ⊓ (CanRead [P idx_i.p; P idx_i.peer]) ⊓ (CanRead [P idx_i.p; P idx_i.peer])	-	(CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p; P idx_r.peer]) ⊓ (CanRead [P idx_r.p; P idx_r.peer])	$= l_r[1]$	-	-	1	2
	→ $[d_1, d_2, ...]$	2	$= l_i[1]$	$= l_i[1]$	$= l_r[1]$	$= l_r[1]$	-	-	1	2
NK1	← s	pre								
	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	0
	↔ $[d_2, d_3, ...]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	1	0	5
NX1	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, s $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	0	1	0	0
	→ es $[d_2]$	3	$l_i[2]$ ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	$l_r[2]$ ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[3]$	0	1	0	3
	← $[d_3]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	1	0	3
	↔ $[d_4, d_5, ...]$	5	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	1	0	5
X1N	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	0	1	0	0
	→ s $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[3]$	0	1	0	1
	← se $[d_3]$	4	$l_i[3]$ ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[4]$	$l_r[3]$ ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	0	3	0	1
	→ $[d_4]$	5	$= l_i[4]$	$= l_i[4]$	$= l_r[4]$	$= l_r[4]$	0	3	2	1
	↔ $[d_5, d_6, ...]$	6	$= l_i[4]$	$= l_i[4]$	$= l_r[4]$	$= l_r[4]$	0	5	2	1
X1K	← s	pre								
	→ e, es $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	-	(CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	0	0	0	2
	← e, ee $[d_1]$	2	$l_i[1]$ ⊓ (CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[1]$ ⊓ (CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	2
	→ s $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[3]$	2	1	0	5
	← se $[d_3]$	4	$l_i[3]$ ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[4]$	$l_r[3]$ ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	2	3	0	5
	→ $[d_4]$	5	$= l_i[4]$	$= l_i[4]$	$= l_r[4]$	$= l_r[4]$	2	3	2	5
	↔ $[d_5, d_6, ...]$	6	$= l_i[4]$	$= l_i[4]$	$= l_r[4]$	$= l_r[4]$	2	5	2	5
XK1	← s	pre								
	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	0
	→ s, se $[d_2]$	3	$l_i[2]$ ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[2]$ ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	2	1	2	5
	↔ $[d_3, d_4, ...]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5

Prot.	Message Seq.	Stage	Initiator Handshake State Label l_i	$l_i^{\leftarrow}$	Responder Handshake State Label l_r	$l_r^{\rightarrow}$	Payload Security Properties ← Auth	← Conf	→ Auth	→ Conf
X1K1	← s	pre								
	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	0
	→ s $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[3]$	2	1	0	5
	← se $[d_3]$	4	$l_i[3]$ ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[4]$	$l_r[3]$ ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	2	3	0	5
	→ $[d_4]$	5	$= l_i[4]$	$= l_i[4]$	$= l_r[4]$	$= l_r[4]$	2	3	2	5
	↔ $[d_5, d_6, \ldots]$	6	$= l_i[4]$	$= l_i[4]$	$= l_r[4]$	$= l_r[4]$	2	5	2	5
X1X	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, s, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	0
	→ s $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[3]$	2	1	0	5
	← se $[d_3]$	4	$l_i[3]$ ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[4]$	$l_r[3]$ ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	2	3	0	5
	→ $[d_4]$	5	$= l_i[4]$	$= l_i[4]$	$= l_r[4]$	$= l_r[4]$	2	3	2	5
	↔ $[d_5, d_6, \ldots]$	6	$= l_i[4]$	$= l_i[4]$	$= l_r[4]$	$= l_r[4]$	2	5	2	5
XX1	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, s $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	0	1	0	0
	→ es, s, se $[d_2]$	3	$l_i[2]$ ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer]) ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[2]$ ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	0	1	2	3
	← $[d_3]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	3
	↔ $[d_4, d_5, \ldots]$	5	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
X1X1	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, s $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	0	1	0	0
	→ es, s $[d_2]$	3	$l_i[2]$ ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	$l_r[2]$ ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[3]$	0	1	0	3
	← se $[d_3]$	4	$l_i[3]$ ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[4]$	$l_r[3]$ ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	2	3	0	3
	→ $[d_4]$	5	$= l_i[4]$	$= l_i[4]$	$= l_r[4]$	$= l_r[4]$	2	3	2	5
	↔ $[d_5, d_6, \ldots]$	6	$= l_i[4]$	$= l_i[4]$	$= l_r[4]$	$= l_r[4]$	2	5	2	5
K1N	→ s	pre								
	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	0	1	0	0
	→ se $[d_2]$	3	$l_i[2]$ ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[2]$ ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	0	1	2	1
	↔ $[d_3, d_4, \ldots]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	0	5	2	1
K1K	→ s	pre								
	← s	pre								
	→ e, es $[d_0]$	1	(CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	-	(CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	0	0	0	2
	← e, ee $[d_1]$	2	$l_i[1]$ ⊓ (CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[1]$ ⊓ (CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	2
	→ se $[d_2]$	3	$l_i[2]$ ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[2]$ ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	2	1	2	5
	↔ $[d_3, d_4, \ldots]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
KK1	→ s	pre								
	← s	pre								
	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, se, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	2	3	0	0
	→ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	3	2	5
	↔ $[d_3, d_4, \ldots]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	5	2	5

Prot.	Message Seq.	Stage	Initiator Handshake State Label l_i	$l_i^{\leftarrow}$	Responder Handshake State Label l_r	$l_r^{\rightarrow}$	Payload Security Properties ← Auth	← Conf	→ Auth	→ Conf
K1K1	→ s	pre								
	← s	pre								
	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	0
	→ se $[d_2]$	3	$l_i[2]$ ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[2]$ ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	2	1	2	5
	↔ $[d_3, d_4, \ldots]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
K1X	→ s	pre								
	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, s, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	0
	→ se $[d_2]$	3	$l_i[2]$ ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[2]$ ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	2	1	2	5
	↔ $[d_3, d_4, \ldots]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
KX1	→ s	pre								
	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, se, s $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[1]$	0	3	0	0
	→ es $[d_2]$	3	$l_i[2]$ ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	$l_r[2]$ ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[3]$	0	3	2	3
	← $[d_3]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	3
	↔ $[d_4, d_5, \ldots]$	5	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
K1X1	→ s	pre								
	→ e $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, s $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	0	1	0	0
	→ se, es $[d_2]$	3	$l_i[2]$ ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	$l_r[2]$ ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[3]$	0	1	2	3
	← $[d_3]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	3
	↔ $[d_4, d_5, \ldots]$	5	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
IK1	← s	pre								
	→ e, s $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, se, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	2	3	0	0
	→ $[d_2]$	3	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	3	2	5
	↔ $[d_3, d_4, \ldots]$	4	$= l_i[2]$	$= l_i[2]$	$= l_r[2]$	$= l_r[2]$	2	5	2	5
I1K1	← s	pre								
	→ e, s $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	0
	→ se $[d_2]$	3	$l_i[2]$ ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[2]$ ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	2	1	2	5
	↔ $[d_3, d_4, \ldots]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
I1X	→ e, s $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	← e, ee, s, es $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] ⊔ idx_i.peer_eph_label) ⊓ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] ⊔ idx_r.peer_eph_label) ⊓ (CanRead [P idx_r.p] ⊔ idx_r.peer_eph_label)	$= l_r[1]$	2	1	0	0
	→ se $[d_2]$	3	$l_i[2]$ ⊓ (CanRead [P idx_i.p] ⊔ idx_i.peer_eph_label)	$= l_i[2]$	$l_r[2]$ ⊓ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[3]$	2	1	2	5
	↔ $[d_3, d_4, \ldots]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5

Prot.	Message Seq.	Stage	Initiator Handshake State Label l_i	$l_i^{\leftarrow}$	Responder Handshake State Label l_r	$l_r^{\rightarrow}$	Payload Security Properties ← Auth	← Conf	→ Auth	→ Conf
IX1	$\rightarrow$ e, s $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, se, s $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label) $\sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer])	$= l_r[1]$	0	3	0	0
	$\rightarrow$ es $[d_2]$	3	$l_i[2] \sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	$l_r[2] \sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[3]$	0	3	2	3
	$\leftarrow$ $[d_3]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	3
	$\leftrightarrow$ $[d_4, d_5, ...]$	5	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5
I1X1	$\rightarrow$ e, s $[d_0]$	1	public	-	public	$= l_r[1]$	0	0	0	0
	$\leftarrow$ e, ee, s $[d_1]$	2	(CanRead [S idx_i.p idx_i.sid] $\sqcup$ idx_i.peer_eph_label)	$= l_i[2]$	(CanRead [S idx_r.p idx_r.sid] $\sqcup$ idx_r.peer_eph_label)	$= l_r[1]$	0	1	0	0
	$\rightarrow$ se, es $[d_2]$	3	$l_i[2] \sqcap$ (CanRead [P idx_i.p] $\sqcup$ idx_i.peer_eph_label) $\sqcap$ (CanRead [S idx_i.p idx_i.sid; P idx_i.peer])	$= l_i[2]$	$l_r[2] \sqcap$ (CanRead [S idx_r.p idx_r.sid; P idx_r.peer]) $\sqcap$ (CanRead [P idx_r.p] $\sqcup$ idx_r.peer_eph_label)	$= l_r[3]$	0	1	2	3
	$\leftarrow$ $[d_3]$	4	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	3
	$\leftrightarrow$ $[d_4, d_5, ...]$	5	$= l_i[3]$	$= l_i[3]$	$= l_r[3]$	$= l_r[3]$	2	5	2	5

Appendix C

Forward Simulation Between HLPL and LLBC

Figure 9.5, Figure 9.6, Figure 9.7, Figure 9.10, Figure 9.3 and Figure 9.4 give the operational semantics of LLBC. This semantics is the big-step semantics without step-indexing; we omit the version with step-indexing, which just consists in adding a step index to all the rules (the index is the same in the conclusion and in the premises, see E-Step-Seq-Unit) at the exception of the rules to evaluate loops and function calls (see E-Step-Call), and finally adding the rule E-Step-Zero. Figure 9.9 introduces the additional rules to evaluate loop statements. Figures Figure 11.3 and Figure 11.5 describe the operational semantics for HLPL and HLPL$^+$. We omit some rules for HLPL because it shares most of its semantics with LLBC. The semantics of HLPL$^+$ is actually not *exactly* a superset of the semantics of HLPL, because we need to replace HLPL-E-Assign with HLPL+-E-Assign, E-Ptr with HLPL+-E-Ptr and HLPL-E-Move with HLPL+-E-Move. For HLPL+-E-Assign, we indeed need to prevent overriding v_p if it contains locations as well as *outer loans*. Without this additional restriction, we can not prove the forward simulation for states related by Le-MutBorrow-To-Ptr or Le-SharedLoan-To-Loc, because they allow turning outer loans into locations (we might get into a situation where we are allowed to overwrite a value in the right state but not the left state). We note that because loan values can only exist in HLPL$^+$ states, HLPL-E-Assign and HLPL+-E-Assign coincide on HLPL states. As a consequence we still get the crucial property we need for the proof of the forward simulation, that is that HLPL is a stable subset of HLPL$^+$. Similar reasonings apply for HLPL+-E-Ptr and HLPL+-E-Move. We need the additional restrictions for the HLPL+-E-Ptr because otherwise we may insert locations where there are already shared loans, which causes issues when combined with Le-SharedLoan-To-Loc.

We now turn to the proof of 2. We need several auxiliary lemmas, to show that

evaluating rvalues (i.e., expressions - Lemma 1), evaluating assignments (Lemma 7) and finally reorganizing states (Lemma 8) preserves the relation $\leq$.

Lemma 1 (Rvalue-Preserves-HLPL+-Rel)**.** *For all Ω_l and Ω_r HLPL$^+$ states and rv right-value we have:*

$$\Omega_l \leq \Omega_r \Rightarrow \forall\, v_r\, \Omega'_r,\ \Omega_r \vdash_{\mathsf{hlpl+}} rv \Downarrow (v_r,\, \Omega'_r) \Rightarrow$$
$$\exists\, v_l\, \Omega'_l,\ \Omega_l \vdash_{\mathsf{hlpl+}} rv \Downarrow (v_l,\, \Omega'_l) \wedge (v_l,\, \Omega'_l) \leq (v_r,\, \Omega'_r)$$

where we define $(v_l,\, \Omega'_l) \leq (v_r,\, \Omega'_r)$ as:

$$(v_l,\, \Omega'_l) \leq (v_r,\, \Omega'_r) := (\Omega'_l,\, \_ \to v_l) \leq (\Omega'_r,\, \_ \to v_r)$$

Proof

We do the proof by induction on $\Omega_r \vdash_{\mathsf{hlpl+}} rv \Downarrow (v_r,\, \Omega'_r)$. Then, in (most of) the subcases we do the proof by induction on $\Omega_l \leq \Omega_r$. The high-level idea is to show that, if Ω_l is related to Ω_r in some specific manner (for instance, Ω_l is Ω_r where we replaced a shared borrow by a pointer by using Le-SharedReserved-To-Ptr) then in most situations they remain related excatly the same way (that is, they remain related by Le-SharedReserved-To-Ptr). For instance, if the left state is the right state where we replaced a shared borrow by a pointer, then after a move operation, the left state is still exactly the right state where we replaced a shared borrow by a pointer; of course, we need to reason about wether the borrow was moved or not, so that we can instantiate Le-SharedReserved-To-Ptr with the proper state with a hole to show that the resulting left state and right state are related.

- Case $\mathsf{copy}\, p$. We have (premises of E-Copy):

$$\vdash \Omega_r(p) \overset{\mathsf{imm}}{\Rightarrow} v_r \wedge$$
$$\bot, \mathsf{loan}^m, \mathsf{borrow}^{m,r} \notin v \wedge$$
$$\vdash \mathsf{copy}\, v_r = v'_r$$

  By induction on $\Omega_l \leq \Omega_r$.

  - Reflexive case. Trivial.
  - Transitive case. Trivial by the induction hypotheses.
  - Case Le-SharedReserved-To-Ptr.

    By the premises of Le-SharedReserved-To-Ptr, there exists $\Omega_1[.]$ such that $\Omega_l = \Omega_1[\mathsf{ptr}\, \ell] \leq \Omega_1[\mathsf{borrow}^s\, \ell] = \Omega_r$ (doing the $\mathsf{borrow}^r\, \ell$ case later).

    We have to reason about whether the hole of $\Omega_1[.]$ is inside the value we read at path p or not, that is: either the hole is not at path p, in which case we read the same value in Ω_l and Ω_r, or the hole is inside, in which case the value we read differs in exactly one place, where we have $\mathsf{borrow}^s\, \ell$ for Ω_r and $\mathsf{ptr}\, \ell$ for Ω_l. We formalize this in the auxiliary lemma 2 below.

Lemma 2. *Auxiliary Lemma*

$$
\begin{aligned}
&\forall\, p\, v_r,\\
&\quad (\vdash \Omega_r(p) \overset{\mathsf{imm}}{\Rightarrow} v_r \Rightarrow)\\
&\quad \Omega_l = \Omega_1[\mathsf{ptr}\,\ell] \wedge \Omega_1[\mathsf{borrow}^s\,\ell] = \Omega_r \Rightarrow\\
&\quad ((\vdash \Omega_l(p) \overset{\mathsf{imm}}{\Rightarrow} v_r) \vee\\
&\qquad (\exists\, V[.],\, (\vdash \Omega_l(p) \overset{\mathsf{imm}}{\Rightarrow} V[\mathsf{ptr}\,\ell]) \;\wedge\; (\vdash \Omega_r(p) \overset{\mathsf{imm}}{\Rightarrow} V[\mathsf{borrow}^s\,\ell])))
\end{aligned}
$$

The proof of this lemma is straightforward by induction on p: we simply have to show that if we can read through one path element on the right (for instance, we can reduce a projection) then we can do the same on the left. The tricky case happens when dereferencing (i.e., $*$). We have to pay attention to two elements.

* We might dereference $\mathsf{borrow}^s\,\ell$ on the right and $\mathsf{ptr}\,\ell$ on the left. In this case we have to use the fact that dereferencing $\mathsf{ptr}\,\ell$ is the same as dereferencing $\mathsf{borrow}^s\,\ell$ (because the read rules are defined so that it is the case; see R-Deref-SharedBorrow and R-Deref-Ptr-SharedLoan).
* Dereferencing a value allows us to "jump" to a value elsewhere in the environment (a shared loan, a location, or an HLPL box). We then have to make a case disjunction on whether the hole is inside the value we jump to or not. We made the theorem statement general enough so that we can handle this case.

Given lemma 2, we instantiate it on p and v_r then do a case disjunction on its conclusion.

* Case 1 (we read the same value on the left and the right). We have:

$$
\begin{aligned}
&\vdash \Omega_r(p) \overset{\mathsf{imm}}{\Rightarrow} v_r \wedge\\
&\vdash \Omega_l(p) \overset{\mathsf{imm}}{\Rightarrow} v_r
\end{aligned}
$$

  Because $(v'_r, \Omega_l) \leq (v'_r, \Omega_r)$ is defined as $(\Omega_l, \_ \to v'_r) \leq (\Omega_r, \_ \to v'_r)$, we can conclude by using Le-SharedReserved-To-Ptr, which gives us: $(v'_r, \Omega_l) \leq (v'_r, \Omega_r)$.
* Case 2 (the hole is inside the value we read). There exists $V[.]$ such that:

$$
\begin{aligned}
&\vdash \Omega_l(p) \overset{\mathsf{imm}}{\Rightarrow} V[\mathsf{ptr}\,\ell] \wedge\\
&\vdash \Omega_r(p) \overset{\mathsf{imm}}{\Rightarrow} V[\mathsf{borrow}^s\,\ell]
\end{aligned}
$$

  By induction on $V[.]$ we prove that the copy differs in exactly one place as well, that is, there exists $V'[.]$ such that: $\vdash \mathsf{copy}\, V[\mathsf{ptr}\,\ell] = V'[\mathsf{ptr}\,\ell]$ and $\vdash \mathsf{copy}\, V[\mathsf{borrow}^s\,\ell] = V'[\mathsf{borrow}^s\,\ell]$.
  This time we have to apply Le-SharedReserved-To-Ptr twice, once for the original borrow, once for the borrow inside the copied value.

$$
\begin{aligned}
\Omega_l, \_ \to V'[\mathsf{ptr}\,\ell] &= \Omega_1[\mathsf{ptr}\,\ell], \_ \to V'[\mathsf{ptr}\,\ell]\\
&\leq \Omega_1[\mathsf{borrow}^s\,\ell], \_ \to V'[\mathsf{ptr}\,\ell]\\
&= \Omega_r, \_ \to V'[\mathsf{ptr}\,\ell]\\
&\leq \Omega_r, \_ \to V'[\mathsf{borrow}^s\,\ell]
\end{aligned}
$$

The case $\mathsf{borrow}^r\ \ell$ is similar, and actually even simpler because: 1. in the proof of 2 we can't dereference a reserved borrow; 2. for the end of the proof we can't copy a reserved borrow.

– Case Le-MutBorrow-To-Ptr. Same as Le-SharedReserved-To-Ptr. We prove an auxiliary lemma which is similar to 2, and use in the proof the fact that pointers and mutable borrows are dereferenced in ways compatible with Le-MutBorrow-To-Ptr (R-Deref-MutBorrow, R-Deref-Ptr-Loc). For the end of the proof, in the case we don't read the same value on the left and on the right, we use the fact that we can't copy mutable borrows or mutable loans.

– Case Le-RemoveAnon. Similar to Le-SharedReserved-To-Ptr but the auxiliary lemma is even simpler because we read the same value on the left and on the right (the presence or absence of an anonymous value *without* borrows or loans doesn't have any impact on evaluation).

– Case Le-Merge-Locs. By the premises of the rule, there exist $\Omega_1[.]$ ℓ_0, ℓ_1 and v such that: $\Omega_r = \Omega_1[\mathsf{loc}\,\ell_0\,(\mathsf{loc}\,\ell_1\,v)]$, $\forall\ v', \mathsf{loc}\,\ell_1\,v' \notin \Omega_1[\ell_0]$, and $\Omega_l = \left[\ell_0 \middle/ \ell_1\right](\Omega_1[\mathsf{loc}\,\ell_0\,v])$.

We prove that, depending on whether the hole of $\Omega_1[.]$ is inside the value we read at p or not, then reading in Ω_l along p is well defined, and the read value is the same as in Ω_r modulo two things: 1. we substitute ℓ_0 for ℓ_1; 2. we may have collapsed the collapsed the locations for ℓ_0 and ℓ_1. Formally, we prove the auxiliary lemma below.

Lemma 3. *Auxiliary Lemma*

$$
\begin{aligned}
&\Omega_r = \Omega_1[\mathsf{loc}\,\ell_0\,(\mathsf{loc}\,\ell_1\,v)] \Rightarrow (\forall\ v', \mathsf{loc}\,\ell_1\,v' \notin \Omega_1[\ell_0]) \Rightarrow \\
&\Omega_l = \left[\ell_0 \middle/ \ell_1\right](\Omega_1[\mathsf{loc}\,\ell_0\,v]) \Rightarrow \forall\ p\,v_r, \vdash \Omega_r(p) \overset{\mathsf{imm}}{\Rightarrow} v_r \Rightarrow \\
&\quad (\vdash \Omega_l(p) \overset{\mathsf{imm}}{\Rightarrow} \left[\ell_0 \middle/ \ell_1\right](v_r) \vee \\
&\quad (\exists\ V[.],\ \vdash \Omega_l(p) \overset{\mathsf{imm}}{\Rightarrow} \left[\ell_0 \middle/ \ell_1\right](V[\mathsf{loc}\,\ell_0\,v]) \wedge v_r = V[\mathsf{loc}\,\ell_0\,(\mathsf{loc}\,\ell_1\,v)]))
\end{aligned}
$$

We do the proof of 3 by induction on $\vdash \Omega_r(p) \overset{\mathsf{imm}}{\Rightarrow} v_r$. A crucial point which makes the proof works is that the substitution applies a *pointwise* transformation. The difficult case is the dereference $(*)$. If we dereference ℓ_0 to read $\mathsf{loc}\,\ell_1\,v$ on the right, we read $\left[\ell_0 \middle/ \ell_1\right] v$ on the left. We then have to make a case disjunction on whether the path is empty (in which case we stop) or not (in which case we dive into the shared loan by R-SharedLoan). Also note that because we do not enforce that states are well-formed, there may be several $\mathsf{loc}\,\ell_0\ldots$ in Ω_r, meaning we don't have to read $\mathsf{loc}\,\ell_1\,v$ (but this doesn't have much impact on the proof, because then a similar $\mathsf{loc}\,\ell_0\ldots$ will appear on the left, and we have, again, to make a case disjunction on whether the hole appears inside the pointed value or not).

Given 3, we can easily conclude the proof (we relate Ω'_l to Ω'_r with Le-Merge-Locs again).

– Case Le-SharedLoan-To-Loc. Same as above. We also use the fact that copying a location (respectively, a shared loan) removes the location (respectively, the shared loan) wrapper (Copy-Loc, Copy-SharedLoan).

– Case Le-Box-To-Loc. We note that $\mathsf{ptr}\,\ell$ and $\mathsf{Box}\,v$ are dereferenced in ways compatible with Le-Box-To-Loc.

By using 2 as model, we prove the auxiliary below, and conclude by using Le-Box-To-Loc.

Lemma 4. *Auxiliary Lemma*

$$\begin{aligned}
&\forall\, p\, v_r, \vdash \Omega_r(p) \overset{\mathsf{imm}}{\Rightarrow} v_r \Rightarrow \\
&\Omega_l = (\Omega_1[\mathsf{ptr}\,\ell],\ _\ell \to v)\ \wedge\ \Omega_1[\mathsf{Box}\,v] = \Omega_r \Rightarrow \\
&((\vdash \Omega_l(p) \overset{\mathsf{imm}}{\Rightarrow} v_r) \vee \\
&\quad (\exists\, V[.],\ (\vdash \Omega_l(p) \overset{\mathsf{imm}}{\Rightarrow} V[\mathsf{ptr}\,\ell])\ \wedge\ (\vdash \Omega_r(p) \overset{\mathsf{imm}}{\Rightarrow} V[\mathsf{Box}\,v])))
\end{aligned}$$

– Case Le-Subst. The read judgement is not affected by the substitution, and we use the fact that $\vdash \mathsf{copy}\, v$ commutes with identifier substitutions.

- Case $\mathsf{move}\, p$.

– Reflexive case. Trivial.

– Transitive case. Trivial by the induction hypotheses.

– Case Le-SharedReserved-To-Ptr.

There exists $\Omega_1[.]$ such that $\Omega_l = \Omega_1[\mathsf{ptr}\,\ell]$ and $\Omega_1[\mathsf{borrow}^{s,r}\,\ell]$. By using the assumption $\vdash \Omega_r(p) \overset{\mathsf{mov}}{\Rightarrow} v_r$ and the fact that the move capability doesn't allow to dereference a shared borrow (R-Deref-SharedBorrow, W-Deref-SharedBorrow) then when reading or updating along path p we don't have to consider the case where we might dereference $\mathsf{borrow}^{s,r}\,\ell$ in the right environment and $\mathsf{ptr}\,\ell$ in the left environment. This allows us to prove by induction on p the following auxiliary lemma (either the hole of Ω_2 is independent of the moved value, or it is inside).

Lemma 5. *Auxiliary Lemma*

$$\begin{aligned}
&\forall\, p\, v_r, \vdash \Omega_r(p) \overset{\mathsf{mov}}{\Rightarrow} v_r \Rightarrow \\
&\Omega_l = \Omega_1[\mathsf{ptr}\,\ell] \wedge \Omega_1[\mathsf{borrow}^{s,r}\,\ell] = \Omega_r \Rightarrow \\
&(\exists\, \Omega_2[.,.],\ \Omega_1[.] = \Omega_2[.,\ v_r]\ \wedge\ (\forall\, v, \vdash \Omega_2[.,\ v](p) \overset{\mathsf{mov}}{\Rightarrow} v) \wedge \\
&\quad (\forall\, v, \vdash \Omega_2[.,.][p \leftarrow v] \overset{\mathsf{mov}}{\Rightarrow} \Omega_2[.,\ v])) \vee \\
&(\exists\, \Omega_2[.]\ V[.], \Omega_1[.] = \Omega_2[V[.]]\ \wedge\ (\forall\, v, \vdash \Omega_2[v](p) \overset{\mathsf{mov}}{\Rightarrow} v) \wedge \\
&\quad (\forall\, v, \vdash \Omega_2[.][p \leftarrow v] \overset{\mathsf{mov}}{\Rightarrow} \Omega_2[v]))
\end{aligned}$$

Given this lemma, we conclude the proof by using Le-SharedReserved-To-Ptr.

– Case Le-MutBorrow-To-Ptr.

There exist ℓ, v, $\Omega_1[.,\ .]$ such that $\Omega_l = \Omega_1[\mathsf{loc}\,\ell\, v]$ and $\Omega_r = \Omega_1[\mathsf{loan}^m\,\ell,\ \mathsf{borrow}^m\,\ell\, v]$. The proof is similar to the case Le-SharedReserved-To-Ptr but this time we have two holes.

Because it is not possible to dive into mutable borrows (e.g., R-Deref-MutBorrow) or mutable loans when moving values, we prove that when reading or updating along path p we can't dereference ℓ and can't go through $\mathsf{loan}^m\,\ell$. Also, following the premise of E-Move, we can't move mutable loans. This allows us to prove the following theorem by induction on p.

Lemma 6. *Auxiliary Lemma*

$$
\begin{array}{l}
\forall\, p\, v_r, \\
\vdash \Omega_r(p) \overset{\mathsf{mov}}{\Rightarrow} v_r \Rightarrow \\
(\exists\, \Omega_2[\,.\,,\,.\,,\,.\,], \\
\quad \Omega_1[\,.\,,\,.\,] = \Omega_2[\,.\,,\,.\,,\,v_r\,] \wedge \\
\quad (\forall\, v, \vdash \Omega_2[\,.\,,\,.\,,\,v\,](p) \overset{\mathsf{mov}}{\Rightarrow} v) \wedge \\
\quad (\forall\, v, \vdash \Omega_2[\,.\,,\,.\,,\,.\,][p \leftarrow v] \overset{\mathsf{mov}}{\Rightarrow} \Omega_2[\,.\,,\,v\,])) \vee \\
(\exists\, \Omega_2[\,.\,,\,.\,]\, V[\,.\,], \\
\quad \Omega_1[\,.\,,\,.\,] = \Omega_2[\,.\,,\,V[\,.\,]\,] \wedge \\
\quad (\forall\, v, \vdash \Omega_2[\,.\,,\,v\,](p) \overset{\mathsf{mov}}{\Rightarrow} v) \wedge \\
\quad (\forall\, v, \vdash \Omega_2[\,.\,,\,.\,][p \leftarrow v] \overset{\mathsf{mov}}{\Rightarrow} \Omega_2[\,.\,,\,v\,]))
\end{array}
$$

We do a case disjunction on the conclusion of the auxiliary lemma.

∗ Case 1. There exist $\Omega_2[\,.\,,\,.\,,\,.\,]$ such that:

$$
\begin{array}{l}
\vdash \Omega_r(p) \overset{\mathsf{mov}}{\Rightarrow} v_r \\
\vdash \Omega_r[p \leftarrow \bot] \overset{\mathsf{mov}}{\Rightarrow} \Omega_2[\,\mathsf{loan}^m\, \ell\,,\, \mathsf{borrow}^m\, \ell\, v\,,\, \bot\,] \\
\vdash \Omega_l(p) \overset{\mathsf{mov}}{\Rightarrow} v_r \\
\vdash \Omega_l[p \leftarrow \bot] \overset{\mathsf{mov}}{\Rightarrow} \Omega_2[\,\mathsf{loc}\, \ell\, v\,,\, \mathsf{ptr}\, \ell\,,\, \bot\,]
\end{array}
$$

Thus:

$$
\begin{array}{l}
\Omega_r(p) \vdash \mathsf{move}\, p \Downarrow (v_r,\, \Omega_2[\,\mathsf{loan}^m\, \ell\,,\, \mathsf{borrow}^m\, \ell\, v\,,\, \bot\,]) \\
\Omega_l(p) \vdash \mathsf{move}\, p \Downarrow (v_r,\, \Omega_2[\,\mathsf{loc}\, \ell\, v\,,\, \mathsf{ptr}\, \ell\,,\, \bot\,])
\end{array}
$$

Posing:

$$
\begin{array}{l}
\Omega'_r := \Omega_2[\,\mathsf{loan}^m\, \ell\,,\, \mathsf{borrow}^m\, \ell\, v\,,\, \bot\,] \\
\Omega'_l := \Omega_2[\,\mathsf{loc}\, \ell\, v\,,\, \mathsf{ptr}\, \ell\,,\, \bot\,]
\end{array}
$$

By instantiating Le-MutBorrow-To-Ptr with the state with holes $\Omega_2[\,.\,,\,.\,,\,\bot\,]$) we get $(v_r,\, \Omega'_l) \leq (v_r,\, \Omega'_r)$.

∗ Case 2. There exist $\Omega_2[\,.\,,\,.\,]$, $V[\,.\,]$ such that:

$$
\begin{array}{l}
\Omega_1[\,.\,,\,.\,] = \Omega_2[\,.\,,\,V[\,.\,]\,] \\
\forall\, v, \vdash \Omega_2[\,.\,,\,v\,](p) \overset{\mathsf{mov}}{\Rightarrow} v \\
\forall\, v, \vdash \Omega_2[\,.\,,\,.\,][p \leftarrow v] \overset{\mathsf{mov}}{\Rightarrow} \Omega_2[\,.\,,\,v\,]
\end{array}
$$

This gives us:

$$\vdash \Omega_r(p) \overset{\mathsf{mov}}{\Rightarrow} V[\,\mathsf{borrow}^m\,\ell\,v\,]$$
$$\vdash \Omega_r[p \leftarrow \bot] \overset{\mathsf{mov}}{\Rightarrow} \Omega_2[\,\mathsf{loan}^m\,\ell\,,\,\bot\,]$$
$$\vdash \Omega_l(p) \overset{\mathsf{mov}}{\Rightarrow} V[\,\mathsf{ptr}\,\ell\,]$$
$$\vdash \Omega_l[p \leftarrow \bot] \overset{\mathsf{mov}}{\Rightarrow} \Omega_2[\,\mathsf{loc}\,2\,v\,,\,\bot\,]$$

By instantiating Le-MutBorrow-To-Ptr with the state with holes $\Omega_2[\,.\,,\,\bot\,]$, $\_ \to V[\,.\,]$ we get: $(v_r,\,\Omega'_l) \leq (v_r,\,\Omega'_r)$.

– Case Le-RemoveAnon. Trivial, for the same arguments as in the $\mathsf{copy}\,p$ case.

– Case Le-Merge-Locs. There exist ℓ_1, ℓ_2, v, $\Omega_1[.,\ .]$ such that $\Omega_l = \left[\ell_1 \middle/ \ell_2\right](\Omega_1[\mathsf{loc}\,\ell_1\,v])$, $\Omega_r = \Omega_1[\mathsf{loc}\,\ell_1\,(\mathsf{loc}\,\ell_2\,v)]$.

The crucial point of the proof is that the read and write rules are such that it is not possible to move the inner location (i.e., $\mathsf{loc}\,\ell_2\,v$) elsewhere (R-Loc requires the immut capability, while here we use the mov capability). If it were the case, we could not apply Le-Merge-Locs to conclude the proof because after the move the locations might not be nested anymore (i.e., we might have lost the fact that ℓ_1 and ℓ_2 point to the same value). This is actually the reason why we forbid moving a value through a pointer which points to a location (while we allow moving a value through a pointer which points to an HLPL box).

– Case Le-SharedLoan-To-Loc. Similar to the case Le-MutBorrow-To-Ptr but simpler because we only have to consider the loan, which can't be moved.

– Case Le-Box-To-Loc. Also similar to the cases above.

– Case Le-Subst. We easily get that the read and write judgements commute with identifier substitutions.

• Case HLPL+-E-Ptr ($\&\,p$, $\&\mathsf{reserved}\,p$, $\&\mathsf{mut}\,p$).

– Reflexive case. Trivial.

– Transitive case. Trivial by the inductive hypotheses.

– Case Le-SharedReserved-To-Ptr.

By the premises of Le-SharedReserved-To-Ptr, there exists $\Omega_1[.]$ such that $\Omega_l = \Omega_1[\mathsf{ptr}\,\ell] \leq \Omega_1[\mathsf{borrow}^s\,\ell] = \Omega_r$

We prove by induction on p the auxiliary lemma.

$$
\begin{array}{l}
\forall\, p\, v_r, \\
\vdash \Omega_r(p) \overset{\mathsf{mov}}{\Rightarrow} v_r \Rightarrow \\
(\exists\, \Omega_2[\,.\,,\,.\,]\, v, \\
\quad \Omega_1[\,.\,] = \Omega_2[\,.\,,\, v\,] \wedge \\
\quad (\forall\, v,\, \Omega_2[\,.\,,\, v\,](p) \overset{\mathsf{mov}}{\Rightarrow} v) \wedge \\
\quad (\forall\, v,\, \vdash \Omega_2[\,.\,,\,.\,][p \leftarrow v] \overset{\mathsf{mov}}{\Rightarrow} v)) \vee \\
(\exists\, \Omega_2[\,.\,]\, V[\,.\,], \\
\quad \Omega_1[\,.\,] = \Omega_2[\,.\,] V[\,.\,] \\
\quad (\forall\, v,\, \Omega_2[\,v\,](p) \overset{\mathsf{mov}}{\Rightarrow} v) \wedge \\
\quad (\forall\, v,\, \vdash \Omega_2[\,.\,][p \leftarrow v] \overset{\mathsf{mov}}{\Rightarrow} v))
\end{array}
$$

We conclude by doing a case disjunction.

- Case Le-MutBorrow-To-Ptr. Similar to above.
- Case Le-RemoveAnon. Similar to above (adding an anonymous value with no borrows or loans doesn't have any effect on the evaluation).
- Case Le-Merge-Locs. Similar to the move case. We note that we can't introduce a location between two shared locations, meaning that we can apply Le-Merge-Locs to conclude.
- Case Le-SharedLoan-To-Loc. Similar. The important case is the case where we transform the shared loan to which we create a new pointer.
- Case Le-Box-To-Loc. Similar to the Le-MutBorrow-To-Ptr case.
- Case Le-Subst. We easily prove by induction that the read and write judgements commute with substitutions. However, we have to consider the cases where the substitution is applied to the (potentially fresh) location that the new pointer points to. If the location is fresh, we can use the substituted identifier to apply HLPL+-E-Ptr on the left, making the two states equal, which allows us to conclude by reflexivity of $\leq$. If it is not fresh, we conclude by Le-Subst.

- Case new. The premises of the rule give us that:

$$
\begin{array}{l}
\Omega_r \vdash op \Downarrow (v'_r,\, \Omega''_r) \\
v_r = \mathsf{ptr}\, \ell \\
\Omega'_r = \Omega''_r,\, \ell \rightarrow v'_r
\end{array}
$$

By the induction hypothesis we get:

$$
\begin{array}{l}
\Omega_l \vdash op \Downarrow (v'_l,\, \Omega''_l) \\
(v'_l,\, \Omega''_l) \leq (v'_r,\, \Omega''_r)
\end{array}
$$

By HLPL-E-Box-New we get:

$$
\Omega_l \vdash \mathsf{new}\, op \Downarrow (\mathsf{ptr}\, \ell,\, \Omega''_l),\, \_ \rightarrow v'_l
$$

All we have to show is that (for some ℓ' fresh, that we choose so that it doesn't appear in the left but also not in the right environment - this makes reasoning about Le-Subst easier, and we can always use Le-Subst to map ℓ to ℓ'):

$$(\mathsf{ptr}\,\ell',\ (\Omega_l'',\ \ell' \to v_l')) \leq (\mathsf{ptr}\,\ell,\ (\Omega_r'',\ \ell \to v_r''))$$

We prove it by induction on $\leq$.

- Case constants. Trivial, because the states are unchanged, and the result of evaluating the constants is independent of the states.
- Case adt constructor. Trivial by the induction hypotheses (we conclude in a manner similar to the case new).
- Case unary/binary operations ($\neg$, $+$, -, etc.). Trivial by the induction hypotheses.

We now prove the following lemma about assignments.

Lemma 7 (Assign-Preserves-HLPL+-Rel)**.** *For all Ω_l and Ω_r HLPL$^+$ states, rv right-value and p place we have:*

$$\Omega_l \leq \Omega_r \Rightarrow \forall\, \Omega_r',\ \Omega_r \vdash_{\mathsf{hlpl+}} p := rv \rightsquigarrow ((),\ \Omega_r') \Rightarrow$$
$$\exists\, \Omega_l',\ \Omega_l \vdash_{\mathsf{hlpl+}} p := rv \rightsquigarrow ((),\ \Omega_l') \wedge \Omega_l' \leq \Omega_r'$$

Proof

Lemma 1 gives us that there exist v_r, Ω_r'', v_l, Ω_l'' such that:

$$\Omega_r \vdash_{\mathsf{hlpl+}} rv \Downarrow (v_r,\ \Omega_r'') \wedge$$
$$\Omega_l \vdash_{\mathsf{hlpl+}} rv \Downarrow (v_l,\ \Omega_l'') \wedge$$
$$(v_l,\ \Omega_l'') \leq (v_r,\ \Omega_r'')$$

We do the proof by induction on $(v_l,\ \Omega_l'') \leq (v_r,\ \Omega_r'')$, then on the path p. The reasoning is very similar to what we saw in the proof of lemma 1. We focus on the important parts of the proofs.

- Reflexive case. Trivial.
- Transitive case. Trivial by the induction hypotheses.
- Case Le-SharedReserved-To-Ptr. We cannot update through a shared or a reserved borrow. We have to reason about three cases: 1. the hole is the right-value and moved to the place at p; 2. the hole is in the overwritten value, in which case we use the fact that it will be moved to an anonymous variable; 3. the hole is neither in the right-value or the overwritten value. In all cases we conclude by using Le-SharedReserved-To-Ptr.
- Case Le-MutBorrow-To-Ptr. Same as Le-SharedReserved-To-Ptr but we have to consider more cases: 1. the right-value might contain the mutable borrow that gets transformed to a pointer; 2. the overwritten value might contain the mutable borrow *and/or* the mutable loan, but they will be moved to an anonymous value. In all cases we conclude by using Le-MutBorrow-To-Ptr.

- Case Le-RemoveAnon. Similar to the cases in 1: trivial by using the fact that an anonymous value with no loans or borrows doesn't have any influence on the evaluation.
- Case Le-Merge-Locs. We have to consider the fact that the two locations may be in the overwritten value. If it is the case, they are both moved to an anonymous variable. In particular, they can't get separated because we can't dive into a location by using the move capability (W-Loc), meaning we can conclude by applying Le-Merge-Locs.
- Case Le-SharedLoan-To-Loc. Similar to above cases. The shared loan might be in the overwritten value, in which case it is moved to an anonymous value.
- Case Le-Box-To-Loc. Similar to above cases. We have to consider the following cases: 1. the box value may be in the right-value; 2. we may move the (inner) boxed value but not the outer box; 3. we may overwrite the box value; 4. we may overwrite the (inner) boxed value but not the outer box value. In all cases, we conclude by using Le-Box-To-Loc.
- Case Le-Subst. Trivial by using the fact that the read and write judgements commute with identifier substitutions.

We now prove that reorganizations preserve the relation between HLPL$^+$ states.

Lemma 8 (Reorg-Preserves-HLPL+-Rel)**.** *For all Ω_l and Ω_r HLPL$^+$ states we have:*

$$\Omega_l \leq \Omega_r \Rightarrow \forall\ \Omega'_r,\ \Omega_r \hookrightarrow \Omega'_r \Rightarrow$$
$$\exists\ \Omega'_l,\ \Omega_l \hookrightarrow \Omega'_l \wedge \Omega'_l \leq \Omega'_r$$

Proof

By induction on $\Omega_r \hookrightarrow \Omega'_r$.

- Case Reorg-None. Trivial.
- Case Reorg-Seq. Trivial by the induction hypotheses.
- Case Reorg-End-MutBorrow. By induction on $\Omega_l \leq \Omega'_r$.
  - Reflexive case. Trivial.
  - Transitive case. Trivial by the induction hypotheses.
  - Case Le-SharedReserved-To-Ptr. We have to make a case disjunction on whether the shared borrow we convert to a pointer is inside the mutably borrowed value or not. We apply Reorg-End-MutBorrow to the left environment and conclude with Le-SharedReserved-To-Ptr.
  - Case Le-MutBorrow-To-Ptr. By using the premises of Le-MutBorrow-To-Ptr we get that there can't be other mutable borrows or loans with the same identifier as the one being transformed (note again that we don't have a well-formedness assumption about the states). We have to consider the following cases: 1. the ended mutable borrow may contain the borrow we convert to a pointer; 2. the ended mutable borrow and the transformed borrow may be the same. In all cases, we end the pointer on the left, then the location. If we are in case 2., we conclude by using reflexivity of $\leq$ (the states on the left and on the right are now the same). Otherwise, we conclude by using Le-MutBorrow-To-Ptr.
  - Case Le-RemoveAnon. Trivial.

  - Case Le-Merge-Locs. Following the premise of Le-Merge-Locs, the borrow we end is necessarily independent of the inner location we get rid of. This allows us apply Reorg-End-MutBorrow to the left state and conclude by using Le-Merge-Locs.
  - Case Le-SharedLoan-To-Loc. The shared loan doesn't have the same loan identifier as the mutable borrow and the mutable loan we end. We apply Reorg-End-MutBorrow to the left state and conclude by Le-SharedLoan-To-Loc.
  - Case Le-Box-To-Loc. Straightforward. We have to reason about the respective places of the box and the borrow and the loan. We apply Reorg-End-MutBorrow to the left state and conclude by Le-Box-To-Loc.
  - Case Le-Subst. We apply Reorg-End-MutBorrow to the left state, potentially on the substituted identifier.

- Case Reorg-End-SharedReservedBorrow. By induction on $\Omega_l \leq \Omega'_r$.
  - Reflexive case. Trivial.
  - Transitive case. Trivial by the induction hypotheses.
  - Case Le-SharedReserved-To-Ptr. The borrow we end may be the one we transform to a pointer. If it is the case, we end the corresponding pointer (Reorg-End-Ptr and we conclude by using reflexivity of $\leq$. Otherwise, we use Le-SharedReserved-To-Ptr.
  - Case Le-MutBorrow-To-Ptr. By using the premise of Le-MutBorrow-To-Ptr we get that the mutable borrow (and its loan) can't have the same loan identifier as the shared borrow we convert to a pointer. This allows us to conclude by using Reorg-End-SharedReservedBorrow on the left environment then Le-MutBorrow-To-Ptr.
  - Case Le-RemoveAnon. Trivial because the anonymous variables can't contain borrows or loans.
  - Case Le-Merge-Locs. The borrow we convert to a pointer may be affected by the substitution. In all cases, we conclude with Reorg-End-SharedReservedBorrow then Le-Merge-Locs.
  - Case Le-SharedLoan-To-Loc. We note that the shared loan we end and the borrow we convert are necessarily independent. We conclude by Reorg-End-SharedReservedBorrow applied to the left environment then Le-SharedLoan-To-Loc.
  - Case Le-Box-To-Loc. We apply Reorg-End-SharedReservedBorrow to the left environment then conclude by Le-Box-To-Loc.
  - Case Le-Subst. We apply Reorg-End-SharedReservedBorrow to the left state, potentially on the substituted identifier.

- Case HLPL+-Reorg-End-SharedLoan. By induction on $\Omega_l \leq \Omega'_r$.
  - Reflexive case. Trivial.
  - Transitive case. Trivial by the induction hypotheses.
  - Case Le-SharedReserved-To-Ptr. By the premise of Reorg-End-SharedLoan, the borrow we convert can't have the same identifier as the shared loan we end. We apply Reorg-End-SharedLoan to the left state and conclude by Le-SharedReserved-To-Ptr.

  - Case Le-MutBorrow-To-Ptr. By the premise of Le-MutBorrow-To-Ptr the borrow (and the loan) we convert can't have the same identifier as the shared loan we end. We apply Reorg-End-SharedLoan to the left environment and conclude by Le-MutBorrow-To-Ptr.
  - Case Le-RemoveAnon. Trivial because the anonymous variable can't contain loans.
  - Case Le-Merge-Locs. By the premise of Le-Merge-Locs, the shared loan we end on the right can't have the same identifier as the inner location that we remove. This means that it is not affected by the substitution, and as a result the premises of Reorg-End-SharedLoan are also satisfied in the left state; we apply this rule and conclude by Le-Merge-Locs.
  - Case Le-SharedLoan-To-Loc. We have to make a case disjunction on whether the shared loan we convert is also the shared loan we end. If it is the case, we apply Reorg-End-Loc to the left state and conclude by reflexivity of $\leq$, otherwise we apply Reorg-End-SharedLoan and conclude by Le-SharedLoan-To-Loc.
  - Case Le-Box-To-Loc. Straightforward. We conclude by Le-Box-To-Loc.
  - Case Le-Subst. We apply Reorg-End-SharedLoan to the left state, potentially on the substituted identifier.

- Case Reorg-End-Ptr. By induction on $\Omega_l \leq \Omega'_r$.
  - Reflexive case. Trivial.
  - Transitive case. Trivial by the induction hypotheses.
  - Case Le-SharedReserved-To-Ptr. We apply Reorg-End-Ptr in the left environment (beware that in the right state we have a *borrow*, so there must be another pointer, possibly with the same identifier, that we can also find in the left state) and conclude by Le-SharedReserved-To-Ptr.
  - Case Le-MutBorrow-To-Ptr. Same as the case Le-SharedReserved-To-Ptr.
  - Case Le-RemoveAnon. Trivial.
  - Case Le-Merge-Locs. The pointer may be affected by the substitution, but we can also end it in the left state and conclude by Le-Merge-Locs.
  - Case Le-SharedLoan-To-Loc. We apply Reorg-End-Ptr in the left state and conclude by Le-SharedLoan-To-Loc.
  - Case Le-Box-To-Loc. We apply Reorg-End-Ptr in the left state and conclude by Le-Box-To-Loc.
  - Case Le-Subst. We apply Reorg-End-Ptr to the left state, potentially on the substituted identifier.

- Case Reorg-End-Loc. By induction on $\Omega_l \leq \Omega'_r$.
  - Reflexive case. Trivial.
  - Transitive case. Trivial by the induction hypotheses.
  - Case Le-SharedReserved-To-Ptr. By the premises of Reorg-End-Loc, the location we end and the borrow we convert don't have the same identifier. We also end the location on the left and conclude by Le-SharedReserved-To-Ptr.
  - Case Le-MutBorrow-To-Ptr. Same as case Le-SharedReserved-To-Ptr.
  - Case Le-RemoveAnon. Trivial.

  - Case Le-Merge-Locs. This one is subtle. Let's name ℓ_0 the location we end, ℓ_1 the outer location and ℓ_2 the inner location we merge with ℓ_1. We might have that $\ell_2 = \ell_0$. If this happens, it means that there are no borrows or loans with identifier ℓ_2, meaning that the substitution applied by Le-Merge-Locs has no effect. In particular, if we end ℓ_2 in the left state by Reorg-End-Loc then we get the same state as on the right and conclude by reflexivity of $\leq$. We might also have that $\ell_1 = \ell_0$. In this case, we also end the outer location ℓ_1 in the left state then conclude by applying Le-Subst to substitute ℓ_2 (the inner location identifier) with ℓ_1. If we are in one of the above cases, we apply Reorg-End-Loc and conclude by Le-Merge-Locs.
  - Case Le-SharedLoan-To-Loc. We apply Reorg-End-Loc to the left state (there must be a location which corresponds to the location we end on the right - the shared loan and this location can not be related) and conclude by Le-SharedLoan-To-Loc.
  - Case Le-Box-To-Loc. Straightforward. We apply Reorg-End-Loc to the left state and conclude by Le-Box-To-Loc.
  - Case Le-Subst. We apply Reorg-End-Loc to the left state, potentially on the substituted identifier, and conclude either by reflexivity or by Le-Subst.

- Case Reorg-Activate-Reserved. By induction on $\Omega_l \leq \Omega'_r$.

  - Reflexive case. Trivial.
  - Transitive case. Trivial by the induction hypotheses.
  - Case Le-SharedReserved-To-Ptr. If the borrow we activate is not the one we convert to a pointer, we apply Reorg-Activate-Reserved on the left state and conclude by Le-SharedReserved-To-Ptr. If it is the same, we do nothing on the left state and conclude by Le-MutBorrow-To-Ptr (the premises of Reorg-Activate-Reserved give us the premises we need for Le-MutBorrow-To-Ptr).
  - Case Le-MutBorrow-To-Ptr. The borrow we activate can not be the same as the borrow we convert to a pointer. This means we can apply Reorg-Activate-Reserved to the left environment and conclude by Le-MutBorrow-To-Ptr.
  - Case Le-RemoveAnon. Trivial.
  - Case Le-Merge-Locs. The locations we merge can't have the same identifier as the borrow we activate. we can apply Reorg-Activate-Reserved to the left environment and conclude by Le-Merge-Locs.
  - Case Le-SharedLoan-To-Loc. Similar to Le-SharedReserved-To-Ptr.
  - Case Le-Box-To-Loc. We apply Reorg-Activate-Reserved to the left state and conclude by Le-Box-To-Loc.
  - Case Le-Subst. We make a case disjunction on whether the borrow we activate is the one on which we apply the substitution to use Reorg-Activate-Reserved on the proper borrow in the left state and conclude by Le-Subst.

We finally turn to the proof of the target theorem, that is 2.

Proof

By induction on $\Omega_r \vdash_{\mathsf{hlpl+}} s \rightsquigarrow (r, \Omega'_r)$.

- Case empty statement. Trivial.
- Case E-Reorg. By Lemma 8.
- Case s_0; s_1. Trivial by the induction hypotheses.
- Case $p :=: rv$. By Lemma 7.
- Case `if then else`. Trivial by the induction hypotheses.
- Case `match`. Trivial by the induction hypotheses.
- Case $\mathsf{free}\, p$. The reasoning is similar to the $\mathsf{move}\, p$ and $p :=$ cases in Lemmas 1 7. The proof is straightforward by induction on $\Omega_l \leq \Omega_r$. The interesting case is the combination of E-Box-Free in the right state and Le-Box-To-Loc as the relation between the two states. By applying Reorg-End-Ptr then HLPL-E-Box-Free to the left state we get equal states and conclude by reflexivity of $\leq$. Importantly, the rule E-Box-Free was written so that only the *content* of the box gets moved to an anonymous value, so that we can relate E-Box-Free to HLPL-E-Box-Free.
- Case return. Trivial.
- Cases panic, $\mathsf{break}\, i$, $\mathsf{continue}\, i$. Trivial.
- Case loop. Trivial by the induction hypotheses.
- Case function call. The rule is heavy but the proof is straightforward. We use the Lemma 1 to relate the states resulting from evaluating the operands given as inputs to the function (trivial induction on the number of inputs). We easily get that the states resulting from $\vdash \mathsf{push\_stack}$ are related (again, induction on the number of inputs then on the number of local variables which are not used as inputs). The induction hypothesis gives us that the states are related after evaluation of *body*. If the tag is panic we are done. If it is return, we prove that the states resulting from $\vdash \mathsf{pop\_stack}$ are related (induction on the number of local variables, and we use 7 for the assignments which "drop" the local variables). Finally, we do a reasoning similar to the one we did in 7 to show that the states are still related after assigning to the destination.

Appendix D

Forward Simulation Between PL and HLPL

The Pointer Language (PL) uses an explicit heap adapted from the CompCert memory model, where the memory is a map from block id to sequence of memory cells, and each memory cell contains a word. We do not fix the architecture (that is, the width of a word) as this is not relevant to the proof. An address is a block id and an offset inside this block. In this model, values are sequences of cells; for instance, a pair is the concatenation of the cells of its sub-values. Similarly to HLPL, PL doesn't make any distinctions between shared borrows and mutables borrows, which are modeled as pointers. Because PL operates over the same AST as LLBC, the evaluation of places is still relatively high-level and requires typing information to compute the proper offsets; for this purpose, memory blocks are typed, and those types are used for the sole purpose of guiding those projections. Also note that the simulation proof relies on the fact that the rules for HLPL and LLBC preserve the types (e.g., see W-Base in Figure 9.4 - we made the types implicit in most of the rules, in particular in the body of the paper, for the purpose of clarity) so it is not possible to update a value with a value of a different type, in order to enforce the fact that the *size* of the values is preserved. Similarly to what CompCert does, every variable has its own memory block, every allocation inserts a fresh memory block in the heap, while deallocation removes the corresponding block. We always deallocate a complete block at once (we do not deallocate sub-blocks). We do not distinguish allocations on the stack and allocations on the heap; however, as we forbid double frees, deallocating a block allocated on the stack (i.e., a variable) leads to the program eventually being stuck as we deallocate variables when popping the stack.

We show the grammar of values and states in Figure D.1. Values are addresses (a block id and an offset), literals (which must fit into a word), or undefined values. We use the symbol ☣ for undefined values to distinguish them from $\bot$ values that we use

v	$::=$		value
		$addr$	address
		$\mathsf{true} \mid \mathsf{false} \mid n_{\mathsf{i32}} \mid n_{\mathsf{u32}} \mid ...$	literal constants
		☣	undefined value
$addr$	$::=$		address
		$(bi,\, n)$	block id and offset
Ω^{LLBC}	$::=$	$\{\ \mathsf{env} : x \underset{\text{partial, inj}}{\longrightarrow} bi,\ \mathsf{heap} : bi \underset{\text{partial}}{\longrightarrow} [v] : \tau,\ \mathsf{stack} : [[x]]\ \}$	state

Figure D.1: Grammar of PL States and Values

$\mathsf{sizeof}\ \tau := 1$ if τ is a literal type

$\mathsf{sizeof}\ (\tau_0, \tau_1) := \mathsf{sizeof}\ \tau_0 + \mathsf{sizeof}\ \tau_1$

$\mathsf{sizeof}\ (\tau_0 + \tau_1) := 1 + \mathsf{max}(\mathsf{sizeof}\ \tau_0,\ \mathsf{sizeof}\ \tau_1)$ (we need an integer for the tag)

$\mathsf{sizeof}\ (\mathsf{Box}\,\tau) := 1$

$\mathsf{sizeof}\ (\&\,\tau) := 1$

$\mathsf{sizeof}\ (\&\mathsf{mut}\,\tau) := 1$

$\mathsf{sizeof}\ (\mu X.\,\tau) := \mathsf{sizeof}\ (\tau[\mu X.\,\tau \big/ X])$

Figure D.2: The sizeof Function

in LLBC. While we use $\bot$ to keep track of non-initialized values and lost permissions such as moved values or ended borrows, in PL ☣ is simply a non-initialized, poison value [461]. Similarly to $\bot$, it causes the program to get stuck if it ever attempts to read them. But at the difference of LLBC, moving a value (PL-E-Move) leaves it unchanged by behaving like a copy, while there is no such thing as ending a borrow or a pointer by replacing it with a ☣.

As with our low-level memory model values are modeled as sequences of words, we need to introduce a notion of type size to properly relate PL values and HLPL values (Figure D.2).

We use the following notations. If s is a sequence, we use the standard notations about subsequences by using intervals. For instance, $(\Omega.\mathsf{mem}\ bi)[n;\, m[$ is the sub-sequence of $\Omega.\mathsf{mem}\ bi$ covering the cells from index n (included) to index m (excluded). We also use the standard index notation: $(\Omega.\mathsf{mem}\, bi)[i]$ is the cell at index i in sequence $\Omega.\mathsf{mem}\ bi$. We define an "update" notation for sequences: $(\Omega.\mathsf{mem}\ bi)[n;\, m[:= [\vec{v}]$ is Ω where the sub-sequence of $\Omega.\mathsf{mem}\ bi$ of indices $[n;\, m[$ has been udpated with $[\vec{v}]$. The notation $[\vec{v}] : \tau$ simply means that the sequence $[\vec{v}]$ has length $\mathsf{sizeof}\,\tau$, and *nothing more*. In particular, we do not check any well-typedness property of those values.

We introduce read and write judgments, like in HLPL and LLBC. Those reading

$$\frac{\Omega.\mathsf{env}\, x = bi}{\vdash \&\Omega(x : \tau) \Rightarrow (bi,\, 0)}\ \textsc{PL-ReadAddr-Var}$$

$$\frac{\vdash \&\Omega(p : \{\&, \&\mathsf{mut}, *\}\tau) \Rightarrow (bi,\, n) \qquad (\Omega.\mathsf{mem}\, bi)[n] = (bi', n')}{\vdash \&\Omega(*p : \tau) \Rightarrow (bi', n')}\ \textsc{PL-ReadAddress-Deref}$$

$$\frac{\vdash \&\Omega(p : (\tau_0, \tau_1)) \Rightarrow (bi,\, n)}{\vdash \&\Omega(p.0 : \tau_0) \Rightarrow (bi,\, n)}\ \textsc{PL-ReadAddress-ProjPairLeft}$$

$$\frac{\vdash \&\Omega(p : (\tau_0, \tau_1)) \Rightarrow (bi,\, n) \qquad n + \mathsf{sizeof}\ \tau_0 < \mathsf{length}\ \Omega.\mathsf{mem}\, bi}{\vdash \&\Omega(p.1 : \tau_1) \Rightarrow (bi,\, n + \mathsf{sizeof}\ \tau_0)}\ \textsc{PL-ReadAddress-ProjPairRight}$$

$$\frac{\begin{array}{c}\vdash \&\Omega(p : \tau_0 + \tau_1) \Rightarrow (bi,\, n) \\ n + 1 < \mathsf{length}\ \Omega.\mathsf{mem}\, bi \\ \tau = \tau_0 \vee \tau = \tau_1\end{array}}{\vdash \&\Omega(p.0 : \tau) \Rightarrow (bi,\, n + 1)}\ \textsc{PL-ReadAddress-ProjSum}$$

$$\frac{\begin{array}{c}\vdash \&\Omega(p : \tau) \Rightarrow (bi,\, n) \\ (\Omega.\mathsf{mem}\, bi)[n;\, \mathsf{sizeof}\ \tau[= [\vec{v}]\end{array}}{\vdash \Omega(p : \tau) \Rightarrow [\vec{v}]}\ \textsc{PL-Read}$$

Figure D.3: Read Judgments for PL

$$\frac{\vdash \&\Omega(p : \tau) \Rightarrow (bi, n) \qquad \Omega' = ((\Omega.\mathsf{mem}\, bi)[n;\, \mathsf{sizeof}\ \tau[:= [\vec{v}])}{\vdash \Omega(p) \leftarrow [\vec{v}] \Rightarrow \Omega'}\ \textsc{PL-Write}$$

Figure D.4: Write Judgment for PL

and writing judgments are guided by types, which we use to compute address offsets when reducing projections, and to compute the length of the sequence of cells we have to read. We use this typing information only for the projections and nothing else. In particular, we do not check any well-typedness property of the states.

For reading in the environment, we introduce two judgments (Figure D.3). The judgment $\vdash \Omega(p : \tau) \Rightarrow [\vec{v}]$ states that reading $\mathsf{sizeof}\ \tau$ cells at place p gives $[\vec{v}]$. The judgment $\vdash \&\Omega(p : \tau) \Rightarrow addr$, states that $addr$ is the address of the (sequence of) value(s) at place p. Note that this judgment does not enforce that there are $\mathsf{sizeof}\ \tau$ cells available at $addr$. Also note that for sum values, we reserve the first cell for the tag (see PL-ReadAddress-ProjSum).

For writing in environments we introduce the judgment $\vdash \Omega(p) \leftarrow [\vec{v}] \Rightarrow \Omega'$ (Figure D.4), which states that updating the cells at place p in Ω with the sequence of values $[\vec{v}]$ yields state Ω'. The full judgment is in Figure D.4.

We define the rules to evaluate expressions in Figure D.5 and the rules to evaluate statements in Figure D.6. The judgment $\Omega \vdash op \Downarrow [\vec{v} : \tau]$ has to be read as: evalu-

$$\text{PL-E-Ptr}\ \frac{\vdash \&\Omega(p:\tau) \Rightarrow addr}{\Omega(p) \vdash \{\&, \&\mathsf{mut}, \&\mathsf{reserved}\}\, p \Downarrow addr : *\tau} \qquad \text{PL-E-Move}\ \frac{\vdash \Omega(p:\tau) \Rightarrow [\vec{v}]}{\Omega(p) \vdash \mathsf{move}\, p \Downarrow [\vec{v}] : \tau}$$

$$\text{PL-E-Copy}\ \frac{\vdash \Omega(p:\tau) \Rightarrow [\vec{v}]}{\Omega(p) \vdash \mathsf{copy}\, p \Downarrow [\vec{v}] : \tau} \qquad \text{PL-E-Constructor-Pair}\ \frac{\Omega \vdash op_0 \Downarrow [\vec{v_0}] : \tau_0 \qquad \Omega \vdash op_1 \Downarrow [\vec{v_1}] : \tau_1}{\Omega \vdash (op_0, op_1) \Downarrow ([\vec{v_0}] \mathbin{+\!\!+} [\vec{v_1}]) : (\tau_0, \tau_1)}$$

$$\text{PL-E-Constructor-Left}\ \frac{\Omega \vdash op \Downarrow [\vec{v}] : \tau_0 \qquad [\vec{v'}] = [0] \mathbin{+\!\!+} [\vec{v}] \mathbin{+\!\!+} [\vec{\bot}] \qquad \mathsf{length}\,[\vec{v'}] = \mathsf{sizeof}\,(\tau_0 + \tau_1)}{\Omega \vdash \mathsf{Left}\, op \rightsquigarrow \vec{v'} : \tau_0 + \tau_1}$$

$$\text{PL-E-Constructor-Right}\ \frac{\Omega \vdash op \Downarrow [\vec{v}] : \tau_1 \qquad [\vec{v'}] = [1] \mathbin{+\!\!+} [\vec{v}] \mathbin{+\!\!+} [\vec{\bot}] \qquad \mathsf{length}\,[\vec{v'}] = \mathsf{sizeof}\,(\tau_0 + \tau_1)}{\Omega \vdash \mathsf{Right}\, op \rightsquigarrow \vec{v'} : \tau_0 + \tau_1}$$

Figure D.5: Evaluating Expressions in PL

ating op in Ω results in the sequence of values $[\vec{v}]$ of length $\mathsf{sizeof}\,\tau$ (the type is only used for the length of the sequence). One has to note that we encode sum values as tagged unions, and the first cell is reserved for the tag (PL-E-Constructor-Left, PL-E-Constructor-Right). The judgment $\Omega \vdash s \rightsquigarrow (r, \Omega')$ has to be read as: evaluating statement s in state Ω leads to result r in state Ω'.

In order to relate the PL states and the HLPL states we introduce a concretization function which turns HLPL states into PL states. The concretization function is parameterized by several auxiliary functions:

- $\mathsf{blockof} : x + \ell^b \underset{\text{partial, inj}}{\longrightarrow} bi : \tau$: partial, injective function from variable or box identifier to typed block (a pair of a block identifier and a type).
- $\mathsf{addrof} : \ell \underset{\text{partial}}{\longrightarrow} addr$: partial function from a loan or a box identifier to an address.

We define the concretization functions $\mathcal{C}_\Omega\ \mathsf{blockof}\ \mathsf{addrof}\ \Omega$ and $\mathcal{C}_v\ \mathsf{blockof}\ \mathsf{addrof}\ v$, for functions and states respectively. We define the concretization function for states below, and the concretization for values in Figure D.7.

$$\begin{aligned}
\mathcal{C}_\Omega\ \mathsf{blockof}\ \mathsf{addrof}\ \Omega^{\text{hlpl}} := \ \{ & \\
& \mathsf{env} := \lambda x\{x \in \Omega^{\text{hlpl}}.\mathsf{env}\}.\ \mathsf{blockof}\, x \\
& \mathsf{mem} := \lambda \mathsf{bi}\{(\mathsf{bi}, \tau) \in \mathsf{image}(\mathsf{blockof})\}, \\
& \qquad \mathcal{C}_v\ \mathsf{blockof}\ \mathsf{addrof}\ (\Omega^{\text{hlpl}}.\mathsf{mem}\ (\mathsf{blockof}^{-1}\, bi) : \tau), \\
& \mathsf{stack} := \Omega^{\text{hlpl}}.\mathsf{stack}\ \}
\end{aligned}$$

$$\text{PL-E-Assign}\ \frac{\Omega \vdash op \rightsquigarrow [\vec{v}] : \tau \qquad \vdash \Omega[p \leftarrow [\vec{v}]] \Rightarrow \Omega'}{\Omega \vdash p := op \rightsquigarrow ((), \Omega')}$$

$$\text{PL-E-IfThenElse-T}\ \frac{\Omega \vdash op \rightsquigarrow [\mathsf{true}] \qquad \Omega \vdash s_0 \rightsquigarrow (r, \Omega')}{\Omega \vdash \mathsf{if}\ op\ \mathsf{then}\ s_0\ \mathsf{else}\ s_1 \rightsquigarrow (r, \Omega')}$$

$$\text{PL-E-IfThenElse-F}\ \frac{\Omega \vdash op \rightsquigarrow [\mathsf{false}] \qquad \Omega \vdash s_1 \rightsquigarrow (r, \Omega')}{\Omega \vdash \mathsf{if}\ op\ \mathsf{then}\ s_0\ \mathsf{else}\ s_1 \rightsquigarrow (r, \Omega')}$$

$$\text{PL-E-Match-Left}\ \frac{\vdash \Omega(p) \Rightarrow [0] \mathbin{+\!\!+} [v] \qquad \Omega \vdash s_0 \rightsquigarrow (r, \Omega')}{\Omega \vdash \mathsf{match}\ p\ \mathsf{with} \mid \mathsf{Left}\ \Rightarrow s_0 \mid \mathsf{Right}\ \Rightarrow s_1 \rightsquigarrow (r, \Omega')}$$

$$\text{PL-E-Match-Right}\ \frac{\vdash \Omega(p) \Rightarrow [1] \mathbin{+\!\!+} [v] \qquad \Omega \vdash s_1 \rightsquigarrow (r, \Omega')}{\Omega \vdash \mathsf{match}\ p\ \mathsf{with} \mid \mathsf{Left}\ \Rightarrow s_0 \mid \mathsf{Right}\ \Rightarrow s_1 \rightsquigarrow (r, \Omega')}$$

$$\text{PL-E-Return}\ \frac{}{\Omega \vdash \mathsf{return} \rightsquigarrow (\mathsf{return}, \Omega)}$$

$$\text{PL-E-Panic}\ \frac{}{\Omega \vdash \mathsf{return} \rightsquigarrow (\mathsf{panic}, \Omega)}$$

$$\text{PL-E-Break}\ \frac{}{\Omega \vdash \mathsf{break}\ i \rightsquigarrow (\mathsf{break}\ i, \Omega)}$$

$$\text{PL-E-Continue}\ \frac{}{\Omega \vdash \mathsf{continue}\ i \rightsquigarrow (\mathsf{continue}\ i, \Omega)}$$

$$\text{PL-E-New}\ \frac{bi\ \text{fresh} \qquad \Omega \vdash op \rightsquigarrow [\vec{v}] : \tau \qquad \Omega' = \{\, \Omega\ \mathsf{with}\ \mathsf{mem} := \Omega.\mathsf{mem} \cup \{\, bi \to [\vec{v}] \,\} \,\}}{\Omega \vdash \mathsf{new}\ op \rightsquigarrow ([(bi, 0)] : \mathsf{Box}\ \tau, \Omega')}$$

$$\text{PL-E-Free}\ \frac{\vdash \Omega(p : \mathsf{Box}\ \tau) \Rightarrow [(bi, 0)] \qquad \Omega' = \{\, \Omega\ \mathsf{with}\ \mathsf{mem} := \Omega.\mathsf{mem} \setminus \{\, bi \,\} \,\}}{\Omega \vdash \mathsf{free}\ p \rightsquigarrow ((), \Omega')}$$

$$\text{PL-E-Loop-Break-Inner}\ \frac{\Omega \vdash s \rightsquigarrow (\mathsf{break}\ 0, \Omega)}{\Omega \vdash \mathsf{loop}\ s \rightsquigarrow ((), \Omega)}$$

$$\text{PL-E-Loop-Break-Outer}\ \frac{\Omega \vdash s \rightsquigarrow (\mathsf{break}\ i + 1, \Omega)}{\Omega \vdash \mathsf{loop}\ s \rightsquigarrow (\mathsf{break}\ i, \Omega)}$$

$$\text{PL-E-Loop-Continue-Inner}\ \frac{\Omega \vdash s \rightsquigarrow (\mathsf{continue}\ 0, \Omega') \qquad \Omega' \vdash \mathsf{loop}\ s \rightsquigarrow (r, \Omega'')}{\Omega \vdash \mathsf{loop}\ s \rightsquigarrow (r, \Omega'')}$$

$$\text{PL-E-Loop-Continue-Outer}\ \frac{\Omega \vdash s \rightsquigarrow (\mathsf{continue}\ i + 1, \Omega)}{\Omega \vdash \mathsf{loop}\ s \rightsquigarrow (\mathsf{continue}\ i, \Omega)}$$

$$\text{PL-E-Loop-Panic}\ \frac{\Omega \vdash s \rightsquigarrow (\mathsf{panic}, \Omega)}{\Omega \vdash \mathsf{loop}\ s \rightsquigarrow (\mathsf{panic}, \Omega)}$$

$$\text{PL-E-Loop-Return}\ \frac{\Omega \vdash s \rightsquigarrow (\mathsf{return}, \Omega)}{\Omega \vdash \mathsf{loop}\ s \rightsquigarrow (\mathsf{return}, \Omega)}$$

$$\text{PL-E-PushStack}\ \frac{\vec{bi}\ \text{fresh} \qquad \Omega' = \{\, \Omega\ \mathsf{with}\ \mathsf{env} = [\overrightarrow{x \to (bi, 0)}] \mathbin{+\!\!+} \Omega.\mathsf{env},\ \mathsf{mem} = [\overrightarrow{bi \to [\vec{v}]}] \mathbin{+\!\!+} \Omega.\mathsf{mem},\ \mathsf{stack} = [\vec{x}] :: \Omega.\mathsf{stack} \,\}}{\vdash \mathsf{pl\_push\_stack}\ [\overrightarrow{x \to \vec{v}}]\ \Omega\ =\ \Omega'}$$

$$\text{PL-E-PopStack}\ \frac{\begin{array}{c}\Omega.\mathsf{stack} = ([x_{\mathrm{ret}}] \mathbin{+\!\!+} [\overrightarrow{x_i}]) :: \mathsf{stack}' \qquad \Omega.env = [x_{\mathrm{ret}} \to bi_{\mathrm{ret}}] \mathbin{+\!\!+} [\overrightarrow{x_i \to [\overrightarrow{bi_i}]}] \mathbin{+\!\!+} \mathsf{env}' \qquad \Omega.\mathsf{mem} = [\overrightarrow{v_{\mathrm{ret}}}] \\ \mathsf{env}' = \Omega.\mathsf{env} \setminus \{bi_{\mathrm{ret}}, \overrightarrow{bi_i}\} \qquad \Omega' = \{\, \Omega\ \mathsf{with}\ \mathsf{stack}\ =\ \mathsf{stack}',\ \mathsf{env}\ =\ \mathsf{env}',\ \mathsf{mem}\ =\ \mathsf{mem}' \,\}\end{array}}{\vdash \mathsf{pl\_pop\_stack}\ \Omega\ =\ ([\overrightarrow{v_{\mathrm{ret}}}],\ \Omega')}$$

$$\text{PL-E-Call}\ \frac{\begin{array}{c} f\langle \vec{\_}, \vec{\tau} \rangle = \mathsf{fn}\ \langle \vec{\_} \rangle\ (\overrightarrow{x_i : \tau_i})\ (\overrightarrow{y_j : \tau_j})\ (x_{\mathsf{ret}} : \tau)\ \{\ s\ \} \qquad \forall\, j, \Omega_j \vdash op_j \Downarrow ([\overrightarrow{v_j}], \Omega_{j+1}) \\ \vdash \mathsf{pl\_push\_stack}\ \left([x_{\mathrm{ret}} \to [\vec{\bot}]] \mathbin{+\!\!+} [\overrightarrow{x_j \to [\overrightarrow{v_j}]}] \mathbin{+\!\!+} [\overrightarrow{y_k \to [\vec{\bot}]}]\right) \Omega_m = \Omega_{\mathrm{begin}} \\ \Omega_{\mathrm{begin}} \vdash body \rightsquigarrow (r, \Omega_{\mathrm{end}}) \\ (r', \Omega_1) = \begin{cases} (\mathsf{panic}, \Omega_{\mathrm{end}}) & \text{if } r = \mathsf{panic} \\ ((), \Omega''_{\mathrm{end}}) & \text{if } r = \mathsf{return} \wedge \vdash \mathsf{pl\_pop\_stack}\ \Omega_{\mathrm{end}}\ =\ ([\overrightarrow{v_{\mathrm{ret}}}],\ \Omega'_{\mathrm{end}}) \wedge \vdash \Omega'_{\mathrm{end}}[p \leftarrow [\overrightarrow{v_{\mathrm{ret}}}]] \overset{\mathsf{mut}}{\Rightarrow} \Omega''_{\mathrm{end}} \end{cases}\end{array}}{\Omega_0 \vdash p := f\langle \vec{\_}, \vec{\tau} \rangle(\overrightarrow{op_j}) \rightsquigarrow (r', \Omega_1)}$$

Figure D.6: Evaluating Statements in PL

$$\textsc{Concrete-Lit}\ \frac{\text{v is a literal value}}{\mathcal{C}_v\ \mathsf{blockof}\ \mathsf{addrof}\ (v : \tau) = [i]}$$

$$\textsc{Concrete-Bot}\ \frac{s = [\overrightarrow{☠}] \qquad \mathsf{length}\ s = \mathsf{sizeof}\ \tau}{\mathcal{C}_v\ \mathsf{blockof}\ \mathsf{addrof}\ (\bot : \tau) = s}$$

$$\textsc{Concrete-Pair}\ \frac{\mathcal{C}_v\ \mathsf{blockof}\ \mathsf{addrof}\ v_0 : \tau_0 = [\vec{v}] \quad \mathcal{C}_v\ \mathsf{blockof}\ \mathsf{addrof}\ v_1 : \tau_1 = [\vec{w}]}{\mathcal{C}_v\ \mathsf{blockof}\ \mathsf{addrof}\ ((v_0, v_1) : (\tau_0, \tau_1)) = [\vec{v}]\ \mathsf{++}\ [\vec{w}]}$$

$$\textsc{Concrete-Sum-Left}\ \frac{\mathcal{C}_v\ \mathsf{blockof}\ \mathsf{addrof}\ v : \tau_0 = [\vec{v}] \quad s = [0]\ \mathsf{++}\ [\vec{v}]\ \mathsf{++}\ [\overrightarrow{☠}] \quad \mathsf{length}\ s = \mathsf{sizeof}\ (\tau_0 + \tau_1)}{\mathcal{C}_v\ \mathsf{blockof}\ \mathsf{addrof}\ (\mathsf{Left}\ v : \tau_0 + \tau_1) = s}$$

$$\textsc{Concrete-Sum-Right}\ \frac{\mathcal{C}_v\ \mathsf{blockof}\ \mathsf{addrof}\ v : \tau_1 = [\vec{v}] \quad s = [0]\ \mathsf{++}\ [\vec{v}]\ \mathsf{++}\ [\overrightarrow{☠}] \quad \mathsf{length}\ s = \mathsf{sizeof}\ (\tau_0 + \tau_1)}{\mathcal{C}_v\ \mathsf{blockof}\ \mathsf{addrof}\ (\mathsf{Right}\ v : \tau_0 + \tau_1) = s}$$

$$\textsc{Concrete-Loc}\ \frac{\mathcal{C}_v\ \mathsf{blockof}\ \mathsf{addrof}\ (v : \tau) = [\vec{v}] : \tau}{\mathcal{C}_v\ \mathsf{blockof}\ \mathsf{addrof}\ \mathsf{loc}\ \ell\ (v : \tau) = [\vec{v}] : \tau}$$

$$\textsc{Concrete-Ptr-Location}\ \frac{\mathsf{addrof}\ \ell^p = addr}{\mathcal{C}_v\ \mathsf{blockof}\ \mathsf{addrof}\ (\mathsf{Right}\ \mathsf{ptr}\ \ell^p) = [addr] : *\tau}$$

$$\textsc{Concrete-Ptr-Box}\ \frac{\mathsf{blockof}\ \ell^b = bi}{\mathcal{C}_v\ \mathsf{blockof}\ \mathsf{addrof}\ (\mathsf{Right}\ \mathsf{ptr}\ \ell^b) = [(bi, 0)] : *\tau}$$

Figure D.7: Concretizing HLPL Values

In order to properly relate the structured values in the HLPL states to a lower-level view with addresses and memory cells, we introduce the judgment $\mathsf{blockof}, \mathsf{addrof} \vdash \Omega^{\text{hlpl}}(addr) \Rightarrow (v : \tau)$ in Figure D.8 which, given some mappings blockof and addrof, defines what it means to read a value *of a given type* at an *address* in the HLPL state Ω^{hlpl}. In order to define this judgment, we also introduce an auxiliary judgment $\vdash (v : \tau) + n \Rightarrow v' : \tau'$ which states that reading the sub-value of $v : \tau$ at offset n yields $v' : \tau'$.

We define a compatibility predicate between an HLPL state Ω^{hlpl} and the auxiliary concretization functions blockof and addrof in Figure D.9. More precisely, the predicate $\mathsf{compatible\ blockof\ addrof}\ \Omega^{\text{hlpl}}$ states that blockof must be defined for all the variables and box identifiers in Ω^{hlpl}, and that addrof must be consistent with the low-level view of Ω^{hlpl} with addresses defined in Figure D.8.

We define a relation $\leq$ between PL states in Figure D.10. We need this relation because of some semantic discrepancies between PL and HLPL. When concretizing a sum value, we might need to pad it with a sequence of ☠ so that it has the proper size. As a result, we might get discrepancies between a concretized PL state in which we

$$\textsc{HLPL-Read-Address}\ \frac{\Omega^{\text{hlpl}}.\mathsf{env}\ (\mathsf{blockof}^{-1}\ ((bi,\ n))) = v : \tau \quad \vdash (v : \tau) + n \Rightarrow v' : \tau}{\mathsf{blockof},\ \mathsf{addrof} \vdash \Omega^{\text{hlpl}}((bi, n)) \Rightarrow v' : \tau}$$

$$\textsc{HLPL-Read-Offset-0}\ \frac{}{\vdash (v : \tau) + 0 \Rightarrow v : \tau}$$

$$\textsc{HLPL-Read-Offset-Pair-Left}\ \frac{n \leq \mathsf{sizeof}\ \tau_0 \quad \vdash (v_0 : \tau_0) + n \Rightarrow v' : \tau'}{\vdash ((v_0,\ v_1) : (\tau_0,\ \tau_1)) + n \Rightarrow v' : \tau'}$$

$$\textsc{HLPL-Read-Offset-Pair-Right}\ \frac{n \geq \mathsf{sizeof}\ \tau_0 \quad \vdash (v_1 : \tau_1) + n - \mathsf{sizeof}\ \tau_0 \Rightarrow v' : \tau'}{\vdash ((v_0,\ v_1) : (\tau_0,\ \tau_1)) + n \Rightarrow v' : \tau'}$$

$$\textsc{HLPL-Read-Offset-Sum-Left}\ \frac{1 \leq n \leq 1 + \mathsf{sizeof}\ \tau_0 \quad \vdash (v_0 : \tau_0) + n - 1 \Rightarrow v' : \tau'}{\vdash (\mathsf{Left}\ v : \tau_0 + \tau_1)) + n \Rightarrow v' : \tau'}$$

$$\textsc{HLPL-Read-Offset-Sum-Right}\ \frac{1 \leq n \leq 1 + \mathsf{sizeof}\ \tau_1 \quad \vdash (v_1 : \tau_1) + n - 1 \Rightarrow v' : \tau'}{\vdash (\mathsf{Right}\ v : \tau_0 + \tau_1)) + n \Rightarrow v' : \tau'}$$

$$\textsc{HLPL-Read-Offset-Loc}\ \frac{\vdash (v : \tau) + n \Rightarrow v' : \tau'}{\vdash \mathsf{loc}\ \ell\ (v : \tau) + n \Rightarrow v' : \tau'}$$

Figure D.8: Reading Addresses in HLPL

$$\begin{aligned}
&\mathsf{compatible\ blockof\ addrof}\ \Omega^{\text{hlpl}} := \\
&\quad \mathsf{domain}\ \Omega^{\text{hlpl}}.\mathsf{env} \subset \mathsf{domain\ blockof} \wedge \\
&\quad (\forall\ \ell^b \in \Omega^{\text{hlpl}},\ \ell^b \in \mathsf{domain\ blockof}) \wedge \\
&\quad (\forall\ addr\ \ell\ v,\ (\mathsf{blockof},\ \mathsf{addrof} \vdash \Omega^{\text{hlpl}}(addr) \Rightarrow \mathsf{loc}\ \ell\ v) \Rightarrow \mathsf{addrof}\ \ell = addr) \wedge \\
&\quad (\forall\ \ell \in \Omega^{\text{hlpl}},\ \exists\ addr\ v,\ \mathsf{blockof},\ \mathsf{addrof} \vdash \Omega^{\text{hlpl}}(addr) \Rightarrow \mathsf{loc}\ \ell\ v)
\end{aligned}$$

Figure D.9: Compatibility Predicate for the Auxiliary Concretization Functions

$$
\begin{array}{lll}
\Omega \leq \Omega' & := & \text{States} \\
 & \Omega.\mathsf{env} = \Omega'.\mathsf{env} \wedge & \\
 & \Omega.\mathsf{stack} = \Omega'.\mathsf{stack} \wedge & \\
 & \Omega.\mathsf{mem} \leq \Omega'.\mathsf{mem} \wedge & \\
m \leq m' & := & \text{Heaps} \\
 & \mathsf{domain}\ m = \mathsf{domain}\ m' \wedge & \\
 & \forall\ bi \in \mathsf{domain}\ m,\ m\ bi \leq m'\ bi & \\
[\vec{v}] \leq [\vec{w}] & := & \text{Sequences of cells} \\
 & \mathsf{length}\ [\overrightarrow{v_i}] = \mathsf{length}\ [\overrightarrow{w_i}] \wedge & \\
 & \forall\ i,\ v_i \leq w_i &
\end{array}
$$

Figure D.10: The $\leq$ Relation For PL States

update the variant of a sum and the concretization of the HLPL state after doing the same update. Also, we get a discrepancy when evaluating the move operator: a move invalidates the value in the HLPL state but not in its concretized PL state. The $\leq$ relation states that $\Omega_0 \leq \Omega_1$ if Ω_0 and Ω_1 are equal everywhere but at the cells where we have ☣ in Ω_1.

We now define a relation $\leq$ between PL states and HLPL states. Contrary to the other relations in this paper and because the states of PL and HLPL are quite difference, $\leq$ is not defined as a series of transformations between states but in a more standard manner.

Definition 3 (Refinement Between PL and HLPL)**.** *For Ω^{pl} a PL state and Ω^{hlpl} an HLPL state, we state that Ω^{pl} and Ω^{hlpl} are in relation, noted $\Omega^{pl} \leq \Omega^{hlpl}$, if there exist* blockof, addrof *such that:*

$$\mathsf{compatible\ blockof\ addrof}\ \Omega^{hlpl}\ \wedge\ \Omega^{pl} \leq \mathcal{C}_\Omega\ \mathsf{blockof\ addrof}\ \Omega^{hlpl}$$

We now turn to the proof of the forward simulation between PL and HLPL. We want to prove the theorem 10.

Theorem 10 (Forward Simulation From HLPL to PL)**.** *For all Ω^{pl} PL state and Ω^{hlpl} HLPL state we have:*

$$
\begin{array}{l}
\Omega^{pl} \leq \Omega^{hlpl} \Rightarrow \\
\forall\ s\ r\ \Omega_1^{hlpl},\ \Omega^{hlpl} \vdash_{\mathsf{hlpl}}\ s \rightsquigarrow (r,\ \Omega_1^{hlpl}) \Rightarrow \\
\exists\ \Omega_1^{pl},\ \Omega^{pl} \vdash_{\mathsf{pl}}\ s \rightsquigarrow (r,\ \Omega_1^{pl})\ \wedge\ \Omega_1^{pl} \leq \Omega_1^{hlpl}
\end{array}
$$

We need a series of auxiliary lemmas.

The following lemma is straightforward to prove, but provides a crucial property to make the proofs work for the expressions and assignments, which are the difficult cases. In particular, it gives us that assignments don't move or duplicate locations, meaning the view of locations as addresses remains consistent between the HLPL state and the PL state.

Lemma 9 (HLPL-Rvalue-NoLoc)**.** *For all* Ω^{hlpl} *HLPL state,* rv*,* v *and* Ω_1^{hlpl} *such that* $\Omega^{hlpl} \vdash_{\mathsf{hlpl}} rv \Downarrow (v, \Omega_1^{hlpl})$*, there are no* loc *values in* v*.*

Proof

By induction on $\Omega^{\text{hlpl}} \vdash_{\mathsf{hlpl}} rv \Downarrow (v, \Omega_1^{\text{hlpl}})$.

- Case copy p. Trivial induction on the value we read. We use the fact that copying a value in HLPL returns the same value but where the locations have been removed (see Copy-Loc in particular).
- Case move p. Trivial by the premises of HLPL-E-Move.
- Case E-Ptr ($\& p$ &reserved p, &mut p). Trivial by the fact that the value is a pointer.
- Case new op. Trivial by the fact that the value is a pointer.
- Case constant. Trivial.
- Case Adt constructor. Trivial.
- Case unary/binary operations (not, neg, +, -, etc.). Trivial.

Lemma 10 (HLPL-PL-Read)**.** *For all* Ω^{pl} *PL state and* Ω^{hlpl} *HLPL state,* blockof*,* addrof *we have:*

$$\begin{aligned}
&\mathsf{compatible\ blockof\ addrof}\ \Omega^{hlpl} \Rightarrow \\
&\Omega^{pl} \leq \mathcal{C}_\Omega\ \mathsf{blockof\ addrof}\ \Omega^{hlpl} \Rightarrow \\
&\forall\, p\, k\, v\, \tau,\ (\vdash \Omega^{hlpl}(p) \overset{\mathsf{k}}{\Rightarrow} v : \tau) \Rightarrow \\
&\exists\, \vec{v},\ \vdash \Omega^{pl}(p : \tau) \overset{\mathsf{k}}{\Rightarrow} [\vec{v}]\ \wedge\ [\vec{v}] \leq \mathcal{C}_v\ \mathsf{blockof\ addrof}\ v
\end{aligned}$$

Proof

Straightforward by induction on p.

Lemma 11 (Rvalue-Preserves-PL-HLPL-Rel)**.** *For all* Ω^{pl} *PL state and* Ω^{hlpl} *HLPL*

state, blockof, addrof *we have:*

$$
\begin{aligned}
&\mathsf{compatible\ blockof\ addrof}\ \Omega^{hlpl} \Rightarrow \\
&\Omega^{pl} \leq \mathcal{C}_\Omega\ \mathsf{blockof\ addrof}\ \Omega^{hlpl} \Rightarrow \\
&\forall\ rv\ v\ \Omega_1^{hlpl},\ \Omega^{hlpl} \vdash_{\mathsf{hlpl}}\ rv \Downarrow (v,\ \Omega_1^{hlpl}) \Rightarrow \\
&\exists\ \mathsf{blockof}_1\ \mathsf{addrof}_1\ \vec{v}, \\
&\quad \Omega^{pl} \vdash_{\mathsf{pl}}\ rv \Downarrow \vec{v} \wedge \\
&\quad \mathsf{compatible\ blockof}_1\ \mathsf{addrof}_1\ \Omega_1^{hlpl} \wedge \\
&\quad \Omega_1^{pl} \leq \mathcal{C}_\Omega\ \mathsf{blockof}_1\ \mathsf{addrof}_1\ \Omega_1^{hlpl} \wedge \\
&\quad \vec{v} \leq \mathcal{C}_v\ \mathsf{blockof}_1\ \mathsf{addrof}_1\ v
\end{aligned}
$$

Proof

By induction on $\Omega^{\text{hlpl}} \vdash_{\mathsf{hlpl}}\ rv \Downarrow (v,\ \Omega_1^{\text{hlpl}})$.

- Case $\mathsf{copy}\, p$. We use Lemma 10 and the fact E-Copy leaves the state unchanged.
- Case $\mathsf{move}\, p$. We use Lemma 10 to relate the values we evaluate to. Relating the updated HLPL state to the PL state is slightly more technical. The goal is implied by the following auxiliary lemma (the proof is straightforward by induction on the path P):

$$
\begin{aligned}
&\forall\ P\ v_0\ v_1\ \tau\ \overrightarrow{v_0}\ \overrightarrow{v_1}\ \Omega_1^{\text{hlpl}}, \\
&\mathsf{compatible\ blockof\ addrof}\ \Omega^{\text{hlpl}} \Rightarrow \\
&\Omega^{\text{hlpl}} \vdash P(v_0) \overset{\mathsf{mov}}{\Rightarrow} v_1 : \tau \Rightarrow \\
&\text{no location} \in v_1 \Rightarrow \\
&\Omega^{\text{pl}} \vdash P([\overrightarrow{v_0}]) \Rightarrow [\overrightarrow{v_1}] : \tau \Rightarrow \\
&[\overrightarrow{v_1}] \leq \mathcal{C}_v\ \mathsf{blockof\ addrof}\ v_1 \Rightarrow \\
&\Omega^{\text{hlpl}} \vdash P(v_0) \leftarrow (\bot : \tau) \overset{\mathsf{mov}}{\Rightarrow} (v_1',\ \Omega_1^{\text{hlpl}}) \Rightarrow \\
&\Omega^{\text{pl}} \leq \mathcal{C}_\Omega\ \mathsf{blockof\ addrof}\ \Omega_1^{\text{hlpl}}\ \wedge\ [\overrightarrow{v_1}] \leq \mathcal{C}_v\ \mathsf{blockof\ addrof}\ v_1'
\end{aligned}
$$

- Case E-Ptr ($\&\, p$ $\&\mathsf{reserved}\, p$, $\&\mathsf{mut}\, p$). Similar to the move case, but if we insert a fresh location, we have to update the addrof function to insert a mapping for the fresh location identifier.
- Case $\mathsf{new}\, op$. We use the induction hypothesis, and need to insert a binding in blockof for the fresh box.
- Case constant. Trivial.
- Case Adt constructor. Straightforward by using the induction hypotheses and the rules to evaluate and concretize Adts.
- Case unary/binary operations ($\neg$, $+$, -, etc.). Trivial.

Lemma 12 (Assign-Preserves-PL-HLPL-Rel)**.** *For all Ω^{pl} PL state and Ω^{hlpl} HLPL state we have:*

$$\begin{aligned}
&\Omega^{pl} \leq \Omega^{hlpl} \Rightarrow \\
&\forall\, p\, rv\, \Omega_1^{hlpl},\ \Omega^{hlpl} \vdash_{\mathsf{hlpl}} p := rv \rightsquigarrow ((), \Omega_1^{hlpl}) \Rightarrow \\
&\exists\, \Omega_1^{pl},\ \Omega^{pl} \vdash_{\mathsf{pl}} p := rv \rightsquigarrow ((), \Omega_1^{pl}) \ \wedge\ \Omega_1^{pl} \leq \Omega_1^{hlpl}
\end{aligned}$$

Proof

We do the proof by induction on p. It is very similar to the move case of Lemma 11. In particular, we use the fact that there are no locations in the value we move (by Lemma 9), and in the value we overwrite and gets saved to an anonymous value (by the premises of HLPL-E-Assign) [1].

Lemma 13 (Reorg-Preserves-PL-HLPL-Rel)**.** *For all Ω^{pl} PL state and Ω^{hlpl} HLPL state we have:*

$$\Omega^{pl} \leq \Omega^{hlpl} \Rightarrow \forall\, \Omega_1^{hlpl},\ \Omega^{pl} \hookrightarrow \Omega^{hlpl} \Rightarrow \Omega^{hlpl} \leq \Omega_1^{hlpl}$$

Proof

By induction on $\Omega^{\text{pl}} \hookrightarrow \Omega^{\text{hlpl}}$. By $\Omega^{\text{pl}} \leq \Omega^{\text{hlpl}}$ there exist $\mathsf{blockof}$, addrof such that:

$$\mathsf{compatible\ blockof\ addrof}\ \Omega^{\text{hlpl}} \ \wedge\ \Omega^{\text{pl}} \leq \mathcal{C}_\Omega\ \mathsf{blockof\ addrof}\ \Omega^{\text{hlpl}}$$

- Case Reorg-None. Trivial.
- Case Reorg-Seq. Trivial by the induction hypotheses.
- Case Reorg-End-Ptr. There exist $\Omega[.]$, ℓ such that:

$$\begin{aligned}
\Omega^{\text{hlpl}} &= \Omega[\mathsf{ptr}\ \ell] \\
\Omega_1^{\text{hlpl}} &= \Omega[\bot]
\end{aligned}$$

  We have to show that for all bi box identifier we have:

$$\Omega^{\text{pl}}.\mathsf{mem}\ bi \leq \mathcal{C}_v\ \mathsf{blockof\ addrof}\ (\Omega_1^{\text{hlpl}}.\mathsf{mem}\ (\mathsf{blockof}^{-1}\ bi))$$

[1] The fact that we need the overwritten value in HLPL to not contain locations to be able to this proof (otherwise we break the relation between the locations in the HLPL state and the addresses in the PL state) is the reason why we ultimately need to forbid overwriting values which contain outer loans in LLBC (see E-Move).

Let's pose:

$$[\vec{v}] := \Omega^{\text{pl}}.\mathsf{mem}\ bi$$
$$v_0 = \Omega_0^{\text{hlpl}}.\mathsf{mem}\ (\mathsf{blockof}^{-1}\ bi)$$
$$v_1 = \Omega_1^{\text{hlpl}}.\mathsf{mem}\ (\mathsf{blockof}^{-1}\ bi)$$

We have $[\vec{v}] \leq \mathcal{C}_v\ \mathsf{blockof}\ \mathsf{addrof}\ v_0$ and we want to show $[\vec{v}] \leq \mathcal{C}_v\ \mathsf{blockof}\ \mathsf{addrof}\ v_1$.

There are two cases, depending on whether the hole is in v_1 or not. More formally, either we have $v_1 = v_0$, in which case the proof is trivial, or there exists $V[.]$ such that $v_0 = V[\mathsf{ptr}\,\ell]$ and $v_1 = V[\bot]$.

The end of the proof is implied by the following auxiliary theorem, which is straightforward to prove by induction on $V[.]$:

$$[\vec{v}] \leq \mathcal{C}_v\ \mathsf{blockof}\ \mathsf{addrof}\ v_0 \Rightarrow$$
$$v_0 = V[\mathsf{ptr}\,\ell] \Rightarrow v_1 = V[\bot] \Rightarrow$$
$$[\vec{v}] \leq \mathcal{C}_v\ \mathsf{blockof}\ \mathsf{addrof}\ v_1$$

- Case Reorg-End-Loc. Similar to above, but this time we have to update addrof to remove the location identifier that we eliminate.

  There exist $\Omega[.]$, ℓ, v_s such that:

$$\Omega^{\text{hlpl}} = \Omega[\mathsf{loan}^s\ \ell\ v_s]$$
$$\Omega_1^{\text{hlpl}} = \Omega[v_s]$$

  Let's pose $\mathsf{addrof}_1 := \mathsf{addrof}_{|(\mathsf{domain}\ \mathsf{addrof})\setminus\{\ell\}}$ the restriction of addrof to its domain from which we removed ℓ. We show that:

$$\Omega^{\text{pl}} \leq \mathcal{C}_\Omega\ \mathsf{blockof}\ \mathsf{addrof}_1\ \Omega_1^{\text{hlpl}} \wedge \tag{1}$$
$$\mathsf{compatible}\ \mathsf{blockof}\ \mathsf{addrof}_1\ \Omega_1^{\text{hlpl}} \tag{2}$$

  We show (1) the same way as in the Reorg-End-Ptr case. For (2), the difficult part is the last two conjuncts:

$$(\forall\ addr\ \ell\ v,\ (\mathsf{blockof}, \mathsf{addrof} \vdash \Omega_1^{\text{hlpl}}(addr) \Rightarrow \mathsf{loc}\,\ell\,v) \Rightarrow \mathsf{addrof}\,\ell = addr) \wedge \tag{3}$$
$$(\forall\ \ell \in \Omega_1^{\text{hlpl}},\ \exists\ addr\ v,\ \mathsf{blockof}, \mathsf{addrof} \vdash \Omega_1^{\text{hlpl}}(addr) \Rightarrow \mathsf{loc}\,\ell\,v) \tag{4}$$

  For (1), we pose (bi, n) an address and do the proof by induction on $\Omega_1^{\text{hlpl}}.\mathsf{mem}\ bi$ (straightforward). For (2), the compatibility assumption for Ω^{hlpl} gives us a candidate address (bi, n), and we conclude the proof also by induction on $\Omega_1^{\text{hlpl}}.\mathsf{mem}\ bi$.

We can finally turn to the proof of the target theorem (10).

- Case empty statement. Trivial.
- Case E-Reorg. By Lemma 13.
- Case $s_0;\ s_1$. Trivial by the induction hypotheses.

- Case $p := rv$. By Lemma 12.
- Case if then else. Trivial by the induction hypotheses.
- Case match. Trivial by the induction hypotheses.
- Case free p. We remove one block from the map and end the corresponding pointer: straightforward.
- Case return. Trivial.
- Cases panic, break i, continue i. Trivial.
- Case loop. Trivial by the induction hypotheses.
- Case function call. Similar to the same case in the proof of 2. We leverage the fact that PL-E-Call and E-Call have a very similar structure. We use the Lemma 11 to relate the states resulting from evaluating the operands given as inputs to the function (trivial induction on the number of inputs). We easily get that the states resulting from $\vdash$ push_stack are related (again, induction on the number of inputs then on the number of local variables which are not used as inputs; we have to take care to extend the map blockof to account for the blocks freshly allocated for the local variables). The induction hypothesis gives us that the states are related after evaluation of *body*. If the tag is panic we are done. If it is return, we prove that the states resulting from $\vdash$ pop_stack are related (induction on the number of local variables, and we use 12 for the assignments which "drop" the local variables; this time we have to remove the blocks which were allocated for the local variables from the map blockof). Finally, we do a reasoning similar to the one we did in 12 to show that the states are still related after assigning to the destination.

Appendix E

Forward Simulation Between LLBC and LLBC$^{\#}$

We show the additional rules for LLBC$^{\#}$ in Figure 10.2. We show the full rules for the $\leq$ relation about LLBC^{+} states in Figure 11.8. Figure 11.10 lists the rules we use to transform values into fresh region abstractions (this judgment is used by Le-ToAbs), while Figure 11.11 describes how to merge two region abstractions into one (used by Le-MergeAbs).

We now turn to the proof that evaluation for LLBC^{+} preserves the relation $\leq$ over LLBC^{+} states. We need several auxiliary lemmas.

Lemma 14 (Rvalue-Preserves-LLBC+-Rel)**.** *For all Ω_l and Ω_r LLBC^{+} states and rv right-value we have:*

$$\Omega_l \leq \Omega_r \Rightarrow \forall\, v_r\, \Omega'_r,\ \Omega_r \vdash_{\mathsf{llbc+}} rv \Downarrow (v_r,\, \Omega'_r) \Rightarrow$$
$$\exists\, v_l\, \Omega'_l,\ \Omega_l \vdash_{\mathsf{llbc+}} rv \Downarrow (v_l,\, \Omega'_l) \wedge (v_l,\, \Omega'_l) \leq (v_r,\, \Omega'_r)$$

where we define $(v_l,\, \Omega'_l) \leq (v_r,\, \Omega'_r)$ as:

$$(v_l,\, \Omega'_l) \leq (v_r,\, \Omega'_r) := (\Omega'_l,\, \_ \to v_l) \leq (\Omega'_r,\, \_ \to v_r)$$

Proof

By induction on $\Omega'_r,\ \Omega_r \vdash_{\mathsf{llbc+}} rv \Downarrow (v_r,\, \Omega'_r)$.

- Case $\mathsf{copy}\, p$. By induction on $\Omega_l \leq \Omega_r$.
  - Case Le-ToSymbolic. The interesting case happens if in the right state we copy the symbolic value we just introduced. If this happens, we use the fact that copying a symbolic value introduces a fresh symbolic value (Copy-Symbolic) and apply Le-ToSymbolic twice to relate the states (once for the original value, once for the copied value).

- Case Le-MoveValue. We note that moving a value can only constrain the places we can directly access in the right state. It also doesn't have any effect on the values we indirectly access through shared borrows (thanks to the premise that the hole is not inside a shared loan). Following this intuition, we prove by induction on p that $\mathsf{copy}\, p$ reduces to the same value in the left state and the right state, and leaves the states unchanged; we conclude by using Le-MoveValue.
- Case Le-Fresh-MutLoan. Similar to Le-MoveValue. We prove by induction on p that $\mathsf{copy}\, p$ reduces to the same value in the left state and the right state, and leaves the states unchanged; we conclude by Le-Fresh-MutLoan.
- Case Le-Fresh-SharedLoan. Similar to above. Here, the copy actually succeeds in the right state if and only if it succeeds in the left state, and it yields the same value.
- Case Le-Reborrow-MutBorrow. Similar to case Le-Fresh-SharedLoan.
- Case Le-Fresh-SharedBorrow. Similar to case Le-Fresh-SharedLoan.
- Case Le-Reborrow-SharedLoan. We might use a shared borrow to copy the fresh symbolic value in the right state and the original value in the left state. Similarly to the case Le-ToSymbolic, we use the fact that Copy-Symbolic introduces a fresh symbolic value and conclude by using Le-ToSymbolic twice if it happens.
- Case Le-Abs-End-SharedLoan. The shared loan in the region abstraction is not accessible from the outside because there are no remanining borrows pointing to this value. The copied value is the same in both states, and we conclude by Le-Abs-End-SharedLoan.
- Case Le-Abs-End-DupSharedBorrow.

  Ending a shared borrow in a region abstraction doesn't have any effect on the $\vdash \mathsf{copy}\, p$ judgement. We conclude by Le-Abs-End-DupSharedBorrow.
- Case Le-Reborrow-SharedBorrow. Similar to the case Le-Reborrow-SharedLoan.
- Case Le-ToAbs.

  We note that the shared loans, which are accessible from the non-anonymous values (and so through the copy) are preserved by the $\prec^{\text{to-abs}}$ rules (in particular, ToAbs-SharedLoan). We prove by induction on the value that: $\forall\, \mathsf{loan}^s\, \ell\, v \in \Omega_l, \mathsf{loan}^s\, \ell\, v \in \Omega_r$. This allows us to prove that the copy operation reduces to the same values in the left and right states, and we conclude by Le-ToAbs.
- Case Le-MergeAbs. Similar to Le-ToAbs: the shared loans are preserved (by induction on the $\bowtie$ derivation).
- Cases Le-ClearAbs, Le-Abs-ClearValue, Le-Abs-DeconstructPair, Le-Abs-DeconstructSum. Similar to Le-ToAbs: the shared loans are preserved.
- Case Le-AnonValue. Similar to above.

- Case $\mathsf{move}\, p$. By induction on $\Omega_l \leq \Omega_r$.
  - Case Le-ToSymbolic. Case disjunction on wether we move the symbolic value or not. We conclude by Le-ToSymbolic.
  - Case Le-MoveValue. By induction on p, we get that move reduces to the same value in both states, and yields two states related by Le-MoveValue.
  - Case Le-Fresh-MutLoan. Similar to Le-MoveValue.
  - Case Le-Fresh-SharedLoan. Similar to Le-MoveValue.

  - Case Le-Reborrow-MutBorrow. Similar to Le-MoveValue.
  - Case Le-Fresh-SharedBorrow. Similar to Le-MoveValue.
  - Case Le-Reborrow-SharedLoan. Similar to Le-MoveValue. We use the fact that we can't move a loaned value (R-SharedLoan, W-SharedLoan) to show that $\mathsf{move}\, p$ reduces to the same value in both states.
  - Case Le-Abs-End-SharedLoan. Similar to the case Le-Reborrow-SharedLoan.
  - Case Le-Abs-End-DupSharedBorrow. Similar to Le-MoveValue (and simpler).
  - Case Le-Reborrow-SharedBorrow. Similar to Le-Reborrow-SharedLoan, but here we use the fact that we can't dereference shared borrows when moving values.
  - Case Le-ToAbs. We note that we can not dereference shared borrows when moving values; this forbids us from jumping to anonymous values or region abstractions.
  - Case Le-MergeAbs. Same as Le-ToAbs.
  - Cases Le-ClearAbs, Le-Abs-ClearValue, Le-Abs-DeconstructPair, Le-Abs-DeconstructSum. Same as Le-ToAbs.
  - Case Le-AnonValue. Similar to above.

- Cases $\&\, p$, $\&\mathsf{reserved}$. By induction on $\Omega_l \leq \Omega_r$.
  - Case Le-ToSymbolic. Case disjunction on wether we borrow the symbolic value or not (we might insert a shared borrow). We conclude by Le-ToSymbolic.
  - Case Le-MoveValue. By induction on p, we get that move reduces to the same value in both states, and yields two states related by Le-MoveValue (we might insert a shared loan in the moved value).
  - Case Le-Fresh-MutLoan. Similar to Le-MoveValue.
  - Case Le-Fresh-SharedLoan. Similar to Le-MoveValue.
  - Case Le-Reborrow-MutBorrow. Similar to Le-MoveValue.
  - Case Le-Fresh-SharedBorrow. Similar to Le-MoveValue.
  - Case Le-Reborrow-SharedLoan. Similar to Le-MoveValue and Le-ToSymbolic (we might borrow the fresh symbolic value or one of the shared values which were moved to the region abstraction; in the first case we have to convert one more shared borrow).
  - Case Le-Abs-End-SharedLoan. The shared loan inside the region abstraction is not accessible as there are no remaining borrows to this value.
  - Case Le-Abs-End-DupSharedBorrow. This has no impact on the evaluation.
  - Case Le-Reborrow-SharedBorrow. Similar to Le-Reborrow-SharedLoan.
  - Case Le-ToAbs. Similar to the $\mathsf{copy}\, p$ case: we use the fact that all the shared loans are preserved in the region abstraction.
  - Case Le-MergeAbs. Same as Le-ToAbs.
  - Cases Le-ClearAbs, Le-Abs-ClearValue, Le-Abs-DeconstructPair, Le-Abs-DeconstructSum. Same as Le-ToAbs.
  - Case Le-AnonValue. Similar to above.

- Case $\&\mathsf{mut}\, p$. By induction on $\Omega_l \leq \Omega_r$. This is similar to the $\mathsf{move}\, p$ case, in particular because we can't mutably borrow shared values.

- Case Le-ToSymbolic. Case disjunction on wether we mutably borrow the symbolic value or not. We conclude by Le-ToSymbolic.
- Case Le-MoveValue. By induction on p, we get that $\&\, p$ reduces to the same value in both states, and yields two states related by Le-MoveValue.
- Case Le-Fresh-MutLoan. Similar to Le-MoveValue.
- Case Le-Fresh-SharedLoan. Similar to Le-MoveValue.
- Case Le-Reborrow-MutBorrow. Similar to Le-MoveValue.
- Case Le-Fresh-SharedBorrow. Similar to the move case.
- Case Le-Reborrow-SharedLoan. Similar to the move case.
- Case Le-Abs-End-SharedLoan. Similar to the move case.
- Case Le-Abs-End-DupSharedBorrow. Similar to the move case.
- Case Le-Reborrow-SharedBorrow. Similar to the move case.
- Case Le-ToAbs. Similar to the move case.
- Case Le-MergeAbs. Similar to the move case.
- Cases Le-ClearAbs, Le-Abs-ClearValue, Le-Abs-DeconstructPair, Le-Abs-DeconstructSum. Similar to the move case.
- Case Le-AnonValue. Similar to above.

- Case new *op*. We use the induction hypothesis.
- Case constant. Trivial: the states have no impact on the reduction of the constants, and are left unchanged.
- Case Adt constructor. We use the induction hypothesis (the proof is similar to the new *op*) case.
- Case unary/binary operations ($\neg$, neg, $+$, -, etc.). Trivial by the induction hypotheses.

We now prove the following lemma about assignments.

Lemma 15 (Assign-Preserves-LLBC+-Rel)**.** *For all Ω_l and Ω_r LLBC$^+$ states, rv right-value and p place we have:*

$$\Omega_l \leq \Omega_r \Rightarrow \forall\, \Omega_r',\ \Omega_r \vdash_{\mathsf{llbc+}} p := rv \rightsquigarrow ((), \Omega_r') \Rightarrow$$
$$\exists\, \Omega_l'\ \ \Omega_l \vdash_{\mathsf{llbc+}} p := rv \rightsquigarrow ((), \Omega_l') \wedge \Omega_l' \leq \Omega_r'$$

Proof

Lemma 14 gives us that there exist v_r, Ω_r'', v_l, Ω_l'' such that:

$$\Omega_r \vdash_{\mathsf{llbc+}} rv \Downarrow (v_r, \Omega_r'') \wedge$$
$$\Omega_l \vdash_{\mathsf{llbc+}} rv \Downarrow (v_l, \Omega_l'') \wedge$$
$$(v_l, \Omega_l'') \leq (v_r, \Omega_r'')$$

We do the proof by induction on $(v_l, \Omega_l'') \leq (v_r, \Omega_r'')$, then on the path p. The reasoning is very similar to what we saw in the proof of lemma 14. Something important to note is that the value that gets overwritten is always saved in a fresh (ghost) anonymous value; this allows us to conclude in most situations.

- Case Le-ToSymbolic. We have to consider several possibilities: 1. the symbolic value is in the value we move; 2. the symbolic value gets moved to an anonymous value because of the assignment. We conclude by Le-ToSymbolic.
- Case Le-MoveValue. Similar to above.
- Case Le-Fresh-MutLoan. Similar to above.
- Case Le-Fresh-SharedLoan. Similar to above.
- Case Le-Reborrow-MutBorrow. Similar to above.
- Case Le-Fresh-SharedBorrow. Similar to above.
- Case Le-Reborrow-SharedLoan. Similar to above.
- Case Le-Abs-End-SharedLoan. Modifications in region abstractions have no impact on writes with the move capability.
- Case Le-Abs-End-DupSharedBorrow. Similar to above.
- Case Le-Reborrow-SharedBorrow. Similar to above.
- Case Le-ToAbs. We can't use a move capability to write to an anonymous value, so we can conclude by Le-ToAbs.
- Case Le-MergeAbs. Similar to above.
- Cases Le-ClearAbs, Le-Abs-ClearValue, Le-Abs-DeconstructPair, Le-Abs-DeconstructSum. Similar to above.
- Case Le-AnonValue. Similar to above.

We now prove that reorganizations preserve the relation between HLPL$^+$ states.

Lemma 16 (Reorg-Preserves-LLBC+-Rel)**.** *For all Ω_l and Ω_r HLPL$^+$ states we have:*

$$\Omega_l \leq \Omega_r \Rightarrow \forall\ \Omega'_r,\ \Omega_r \hookrightarrow \Omega'_r \Rightarrow$$
$$\exists\ \Omega'_l,\ \Omega_l \hookrightarrow \Omega'_l \wedge \Omega'_l \leq \Omega'_r$$

Proof

By induction on $\Omega_r \hookrightarrow \Omega'_r$.

- Case Reorg-None. Trivial.
- Case Reorg-Seq. Trivial by the induction hypotheses.
- Case Reorg-End-MutBorrow. By induction on $\Omega_l \leq \Omega_r$.
  - Case Le-ToSymbolic. Case disjunction on whether the symbolic value is inside the mutably borrowed value or not. We conclude by Le-ToSymbolic.
  - Case Le-MoveValue. Similar. An important point is that the moved value can't come from a region abstraction (this is important because we can't end borrows which are inside region abstractions: we have to end the abstraction first).
  - Case Le-Fresh-MutLoan. Similar.
  - Case Le-Fresh-SharedLoan. Similar.

  - Case Le-Reborrow-MutBorrow. We have to make a case disjunction on whether we end the fresh borrow ℓ_1 or not. If no, we conclude by Le-Reborrow-MutBorrow. If yes, the premises of Reorg-End-MutBorrow give us that the hole is not inside a shared loan, and there are no loans in v. This allows us to conclude by using Le-MoveValue.
  - Case Le-Fresh-SharedBorrow. We conclude by Le-Fresh-SharedBorrow.
  - Case Le-Reborrow-SharedLoan. We conclude by Le-Reborrow-SharedLoan.
  - Case Le-Abs-End-SharedLoan. We conclude by Le-Abs-End-SharedLoan.
  - Case Le-Abs-End-DupSharedBorrow. We conclude by Le-Abs-End-DupSharedBorrow.
  - Case Le-Reborrow-SharedBorrow. We conclude by Le-Abs-End-DupSharedBorrow.
  - Case Le-ToAbs. We can not end borrows in region abstractions with Reorg-End-MutBorrow (we have to use Reorg-End-Abstraction). However, we can end a mutable loan inside a region abstraction. If the mutable loan is not inside the region abstraction we just introduced, we conclude by Le-ToAbs. Otherwise, we prove that ending the same mutable loan on the left (it must be in the anonymous variable we convert to an abstraction) then converting the anonymous variables to a region abstraction by Le-ToAbs yields the same state as on the right (the proof is by induction on the value we convert).
  - Case Le-MergeAbs. Similar to Le-ToAbs. The difficult case is MergeAbs-Mut; the important point to notice is that it makes borrows and loans disappear in the *right* state, meaning there are more borrows and loans in the left state (this is important, because we don't have a well-formedness assumption). In particular, if the mutable loan we end is in the merged abstraction, then it is also present in one of the two initial abstractions in the left state, and the corresponding borrow is not inside a region abstraction.
  - Cases Le-ClearAbs, Le-Abs-ClearValue, Le-Abs-DeconstructPair, Le-Abs-DeconstructSum. Trivial.
  - Case Le-AnonValue. Trivial.

- Case Reorg-End-SharedReservedBorrow. By induction on $\Omega_l \leq \Omega_r$.
  - Case Le-ToSymbolic. Trivial.
  - Case Le-MoveValue. Case disjunction on wether the borrow is moved or not; we conclude by Le-MoveValue. Similarly to the Reorg-End-MutBorrow case, an important point is that the moved value can't come from a region abstraction (this is important because we can't end borrows which are inside region abstractions: we have to end the abstraction first).
  - Case Le-Fresh-MutLoan. The ended borrows is necessarily independent from the fresh mutable loan; we conclude by Le-Fresh-MutLoan.
  - Case Le-Fresh-SharedLoan. We conclude by Le-Fresh-SharedLoan.
  - Case Le-Reborrow-MutBorrow. Similar to above.
  - Case Le-Fresh-SharedBorrow. On the right we might terminate the fresh shared borrow in which case we conclude by Le-AnonValue; otherwise we conclude with Le-Fresh-SharedBorrow.
  - Case Le-Reborrow-SharedLoan. We can not end any of the borrows inside the fresh region abstraction, meaning we can not end the fresh borrow nor the borrows inside the shared value. We conclude by Le-Reborrow-SharedLoan.
  - Case Le-Abs-End-SharedLoan. We conclude by Le-Abs-End-SharedLoan.

- Case Le-Abs-End-DupSharedBorrow. We can not directly end a borrow inside a region abstraction; we conclude by Le-Abs-End-DupSharedBorrow.
- Case Le-Reborrow-SharedBorrow. This one is slightly technical. there exist $\Omega[\,.\,]$, ℓ_0, ℓ_1, v, σ, A_0 such that:

$$\begin{aligned}
&\bot, \mathsf{loan}^m, \mathsf{borrow}^{s,r,m} \notin v \\
&\mathsf{loan}^s\ \ell_0\ v \in \Omega_l \\
&\Omega_l = \Omega[\,\mathsf{borrow}^s\ \ell_0\,] \\
&\Omega_r = \Omega[\,\mathsf{borrow}^s\ \ell_1\,],\ A_0\ \{\,\mathsf{borrow}^s\ \ell_0,\ \mathsf{loan}^s\ \ell_1\ \sigma\,\}
\end{aligned}$$

  The difficult case happens when we end the borrow ℓ_1, so this is the case we focus on (in the other case, we conclude by Le-Reborrow-SharedBorrow). We do no reorganization on the left. We have to prove:

$$\Omega[\,\mathsf{borrow}^s\ \ell_0\,] \leq \Omega[\,\bot\,],\ A_0\ \{\,\mathsf{borrow}^s\ \ell_0,\ \mathsf{loan}^s\ \ell_1\ \sigma\,\}$$

  Because we could end ℓ_1 on the right, necessarily the hole on the left is not inside a shared loan. This means we can move it to an anonymous value by using Le-MoveValue. We get:

$$\Omega[\,\ell_0\,] \leq \Omega[\,\bot\,],\ \_ \to \mathsf{borrow}^s\ \ell_0$$

  We now apply Le-Reborrow-SharedBorrow to $\mathsf{borrow}^s\ \ell_0$. We get:

$$\Omega[\,\ell_0\,] \leq \Omega[\,\bot\,],\ \_ \to \mathsf{borrow}^s\ \ell_1,\ A_1\ \{\,\mathsf{borrow}^s\ \ell_0,\ \mathsf{loan}^s\ \ell_1\ \sigma\,\}$$

  We apply Le-ToAbs to $\mathsf{borrow}^s\ \ell_0$, then merge the two region abstractions with Le-MergeAbs, taking care of using MergeAbs-Shared to get rid of $\mathsf{borrow}^s\ \ell_1$. QED.
- Case Le-ToAbs. We can not end a borrow which was moved to a region abstraction; the borrow we end on the right state was thus not in the anonymous value we converted to an abstraction; we can end it in the left state and conclude by Le-ToAbs.
- Case Le-MergeAbs. Similar to Le-ToAbs.
- Cases Le-ClearAbs, Le-Abs-ClearValue, Le-Abs-DeconstructPair, Le-Abs-DeconstructSum. Trivial.
- Case Le-AnonValue. Trivial.

- Case Reorg-End-SharedLoan. By induction on $\Omega_l \leq \Omega_r$.
  - Case Le-ToSymbolic. Trivial.
  - Case Le-MoveValue. Trivial.
  - Case Le-Fresh-MutLoan. Trivial.
  - Case Le-Fresh-SharedLoan. We might end the fresh shared loan, in which case the right state become equal to the left state; we conclude by reflexivity of $\leq$.
  - Case Le-Reborrow-MutBorrow. Trivial.
  - Case Le-Fresh-SharedBorrow. Trivial (we can not end the shared loan from which we created a new shared borrow).

  - Case Le-Reborrow-SharedLoan. Because there is $\mathsf{borrow}^s\ \ell_1$ in the region abstraction, we can not end $\mathsf{loan}^s\ \ell_1\ \sigma$ on the right. We can end the shared loan ℓ_0, if there are no shared borrows for ℓ_0. If it is the case, we do no reorganization on the left, then use Le-Reborrow-SharedLoan in combination with Le-Abs-End-SharedLoan. In the other situations, we can conclude by Le-Reborrow-SharedLoan.
  - Case Le-Abs-End-SharedLoan. Trivial.
  - Case Le-Abs-End-DupSharedBorrow. Trivial.
  - Case Le-Reborrow-SharedBorrow. We can not end any of the shared loans for ℓ_0 or ℓ_1 on the right; we easily conclude by Le-Reborrow-SharedBorrow.
  - Case Le-ToAbs. If we end one of the shared loans which were moved to the fresh region abstraction, we use Le-ToAbs then Le-Abs-End-SharedLoan. Otherwise we end the corresponding shared loan on the left then conclude by Le-ToAbs.
  - Case Le-MergeAbs. If the shared loan we end doesn't appear inside the merged region abstraction, we end the corresponding loan on the left and conclude by Le-MergeAbs. Otherwise, we simply use Le-MergeAbs then Le-Abs-End-SharedLoan.
  - Cases Le-ClearAbs, Le-Abs-ClearValue, Le-Abs-DeconstructPair, Le-Abs-DeconstructSum. Trivial.
  - Case Le-AnonValue. Trivial.

- Case Reorg-Activate-Reserved. By induction on $\Omega_l \leq \Omega_r$.
  - Case Le-ToSymbolic. Trivial.
  - Case Le-MoveValue. Trivial.
  - Case Le-Fresh-MutLoan. Trivial.
  - Case Le-Fresh-SharedLoan. Trivial.
  - Case Le-Reborrow-MutBorrow. Trivial.
  - Case Le-Fresh-SharedBorrow. The borrow we activate can't have the same identifier as the fresh shared borrow; we can thus activate the same borrow on the left and conclude by Le-Fresh-SharedBorrow.
  - Case Le-Reborrow-SharedLoan. The borrow we activate can not be ℓ_1 because there is a corresponding shared borrow, nor ℓ_0, because the shared loan is in a region abstraction. We can thus activate the same borrow on the left and conclude by Le-Reborrow-SharedLoan.
  - Case Le-Abs-End-SharedLoan. Trivial.
  - Case Le-Abs-End-DupSharedBorrow. Trivial.
  - Case Le-Reborrow-SharedBorrow. Similar to Le-Reborrow-SharedLoan.
  - Case Le-ToAbs. We can not move reserved borrows to region abstractions, and can not activate borrows associated to loans in region abstractions. We can thus activate the same borrow on the left and conclude by Le-ToAbs.
  - Case Le-MergeAbs. Similar to Le-ToAbs.
  - Cases Le-ClearAbs, Le-Abs-ClearValue, Le-Abs-DeconstructPair, Le-Abs-DeconstructSum. Trivial.
  - Case Le-AnonValue. Trivial.

- Case Reorg-End-Abstraction. By induction on $\Omega_l \leq \Omega_r$.
  - Case Le-ToSymbolic. Trivial.
  - Case Le-MoveValue. Trivial (the moved value can't be inside a region abstraction; we can thus end the region abstraction on the left and conclude by Le-MoveValue).
  - Case Le-Fresh-MutLoan. Trivial (we can't end a region abstraction in which there is a loan).
  - Case Le-Fresh-SharedLoan. Trivial (similar to the case Le-Fresh-MutLoan).
  - Case Le-Reborrow-MutBorrow. The mutable borrow might be inside the region abstraction we end, but we can still conclude Le-Reborrow-MutBorrow. We also conclude by Le-Reborrow-MutBorrow in the other cases.
  - Case Le-Fresh-SharedBorrow. Trivial (note that we can't end a region abstraction in which there is a loan).
  - Case Le-Reborrow-SharedLoan. Trivial (note that we can't end a region abstraction in which there is a loan).
  - Case Le-Abs-End-SharedLoan. We might end the region abstraction in which we just ended the loan. In this case, we also end the loan and the region abstraction on the left, and conclude by reflexivity of $\leq$. Otherwise, we end the corresponding region on the left, and conclude by Le-Abs-End-SharedLoan.
  - Case Le-Abs-End-DupSharedBorrow. If we end the region abstraction from which we removed the duplicated borrow, we can end the same region abstraction on the left, and end one of the borrows which were reintroduced into the context. We then need to remove the anonymous variable containing $\bot$ that is left in place of the borrow (there is no primitive rule for $\leq$ to do that, but we can actually achieve the same result by using Le-ToAbs on this value).
  - Case Le-Reborrow-SharedBorrow. Trivial.
  - Case Le-ToAbs. If the abstraction we end is not the one we just introduced, we can end the corresponding abstraction on the left an conclude by Le-ToAbs. If it is the same region abstraction, then this abstraction doesn't contain any loans. This means that the anonymous value we converted to a region abstraction only contains borrows (we get this by a simple induction on the value). Also, it doesn't contain nested borrows (because it doesn't contain shared loans, and because of the premises of ToAbs-MutBorrow). Finally, the mutably borrowed values don't contain $\bot$ (again, because of the premises of ToAbs-MutBorrow). We can thus apply Le-ToSymbolic on all the mutably borrowed value of the anonymous value (note that when ending a region abstraction, we reintroduce the mutable borrows with fresh symbolic values in the context; see Reorg-End-Abstraction). We can then move all the borrows from the anonymous to their own anonymous values (by Le-MoveValue; we showed before that those borrows don't appear in shared loans) and finally get rid of the original anonymous value by using the same technique as in the case Le-Abs-End-DupSharedBorrow.
  - Case Le-MergeAbs. If the abstraction we end is not the merged abstraction, we end the corresponding abstraction on the left and conclude by Le-MergeAbs. If it is the same abstraction, we can end the first merged abstraction on the left, the borrows it reintroduced in the context and whose corresponding loans are in the second abstraction, then the second abstraction. In particular, we note that we could end the merged abstraction only if, for all pairs borrow/loan such that the borrow was in the first abstraction and the loan in the second, we correctly used

MergeAbs-Mut. The borrows consumed during the merge by using MergeAbs-Mut are the ones we need to end on the left, after ending the first abstraction, so that we can end the second. Finally, we conclude by the reflexivity of $\leq$.

- Cases Le-ClearAbs, Le-Abs-ClearValue, Le-Abs-DeconstructPair, Le-Abs-DeconstructSum. Trivial.
- Case Le-AnonValue. Trivial.

We now turn to the proof of the target theorem.

Theorem 11 (Eval-Preserves-LLBC+-Rel)**.** *For all Ω_l and Ω_r LLBC$^+$ states we have:*

$$\Omega_l \leq \Omega_r \Rightarrow \forall\ s\ v_r\ \Omega'_r,\ \Omega_r \vdash_{\mathsf{llbc+}} s \rightsquigarrow (r,\ \Omega'_r) \Rightarrow$$
$$\exists\ \Omega'_l,\ \Omega_l \vdash_{\mathsf{llbc+}} s \Downarrow (r,\ \Omega'_l) \wedge \Omega'_l \leq \Omega'_r$$

Proof

By induction on $\Omega_r \vdash_{\mathsf{llbc+}} s \rightsquigarrow\rightsquigarrow (r,\ \Omega'_r)$.

- Case empty statement. Trivial.
- Case E-Reorg. By Lemma 16.
- Case $s_0;\ s_1$. We use the induction hypotheses, then have to consider all the states resulting from evaluating s_0; this is straightforward.
- Case $p := rv$. By Lemma 15.
- Case if then else. Trivial by the induction hypotheses.
- Case match. Trivial by the induction hypotheses.
- Case free p. The reasoning is similar to the move p and $p :=$ cases in Lemmas 14 and 15. The proof is straightforward by induction on $\Omega_l \leq \Omega_r$.
- Case return. Trivial.
- Cases panic, break i, continue i. Trivial.
- Case loop. Trivial by the induction hypotheses.
- Case function call. Similar to the same case in the proof of 2. Note that we study LLBC$^+$ here, not LLBC$^{\#}$, meaning the rule we need to consider is E-Call, not E-Call-Symbolic. We use the Lemma 14 to relate the states resulting from evaluating the operands given as inputs to the function (trivial induction on the number of inputs). We easily get that the states resulting from $\vdash$ push_stack are related (again, induction on the number of inputs then on the number of local variables which are not used as inputs). The induction hypothesis gives us that the states are related after evaluation of *body*. If the tag is panic we are done. If it is return, we get that the states resulting from $\vdash$ pop_stack are related (induction on the number of local variables, and we use 15 for the assignments which "drop" the local variables). Finally, we do a reasoning similar to what we did in 15 to show that the states are still related after assigning to the destination.

Appendix F

Forward Simulation for LLBC$^+$ and LLBC$^\#$

We present the definitions of init, final, $\mathsf{inst} - \mathsf{sig}$ in figures Figure 10.4, Figure 10.5, Figure 11.14 and Figure 10.3.

The following substitution lemmas are trivial but crucial in several situations.

Lemma 17 (Le-Subst)**.** *For all Ω_l, Ω_r LLBC$^+$ states and* subst *identifier substitution we have (where* $\mathsf{subst}\ \Omega$ *is a pointwise substitution):*

$$\Omega_l \leq \Omega_r \Rightarrow \mathsf{subst}\ \Omega_l \leq \mathsf{subst}\ \Omega_r$$

Proof.

The proof is straightforward by induction on $\Omega \leq \Omega'$.

- Reflexive case. Trivial.
- Transitive case. Trivial by the induction hypothesis.
- Case Le-ToSymbolic. There exist $\Omega[.]$, v and σ fresh such that:

$$\text{borrows, loans}, \bot \notin v \ \wedge\ \Omega_l = \Omega[v] \ \wedge\ \Omega_r = \Omega[\sigma]$$

A trivial induction on v gives us that, as there are no borrows, loans and $\bot$ in v (premise of the rule) then there are no borrows, loans and $\bot$ in $\mathsf{subst}\ v$. Moreover, as subst is injective we have that $\mathsf{subst}\ \sigma$ if fresh for Ω_l. We pose:

$$\Omega'[.] := (\mathsf{subst}\ \Omega)[.]$$
$$v' := \mathsf{subst}\ v'$$
$$\sigma' := \mathsf{subst}\ \sigma$$

We have (trivial induction on $\Omega[.]$, comes from the fact that subst is applied pointwise):

$$\begin{aligned}\mathsf{subst}\ \Omega_l &= \mathsf{subst}\ (\Omega[v]) \\ &= (\mathsf{subst}\ \Omega)[\mathsf{subst}\ v]) \\ &= \Omega'[v']\end{aligned}$$

Similarly, we have: $\mathsf{subst}\ \Omega_r = \Omega'[\sigma']$. Hence: $\mathsf{subst}\ \Omega_l \leq \mathsf{subst}\ \Omega_r$.

- Case Le-MoveValue. Trivial. We use the fact that the substitution is pointwise (like in the Le-ToSymbolic case).
- Case Le-Fresh-MutLoan. Similar to the Le-ToSymbolic case.
- Case Le-Fresh-SharedLoan. Similar to the Le-ToSymbolic case.
- Case Le-Reborrow-MutBorrow. Similar to the Le-ToSymbolic case.
- Case Le-Fresh-SharedBorrow. Similar to the Le-ToSymbolic case.
- Case Le-Reborrow-SharedLoan. Similar to the Le-ToSymbolic case.
- Case Le-Abs-End-SharedLoan. Similar to the Le-ToSymbolic case.
- Case Le-Abs-End-DupSharedBorrow.Similar to the Le-ToSymbolic case.
- Case Le-Reborrow-SharedBorrow. Similar to the Le-ToSymbolic case.
- Case Le-ToAbs. We need the auxiliary lemma:

$$\forall v \vec{A},\ v \prec^{\text{to-abs}} \vec{A} \Rightarrow \mathsf{subst}\ v \prec^{\text{to-abs}} \mathsf{subst}\ (\prec^{\text{to-abs}} \vec{A})$$

  The proof is straightforward by induction on $v \prec^{\text{to-abs}} \vec{A}$.
- Case Le-MergeAbs. We need the auxiliary lemma:

$$\forall A_0\ A_1\ A, \vdash A_0 \bowtie A_1 = A \Rightarrow \vdash \mathsf{subst}\ A_0 \bowtie \mathsf{subst}\ A_1 = \mathsf{subst}\ A$$

  The proof is straightforward by induction on $\vdash A_0 \bowtie A_1 = A$.
- Cases Le-ClearAbs, Le-Abs-ClearValue, Le-Abs-DeconstructPair, Le-Abs-DeconstructSum, Le-AnonValue. Similar to the Le-MoveValue case.

Lemma 18 (Eval-Subst-LLBC$^+$)**.** *For all Ω, Ω' LLBC$^+$ states, s statement, r control-flow tag and* subst *identifier substitution we have:*

$$\begin{aligned}&\Omega \leq \Omega' \Rightarrow \\ &\Omega \vdash s \rightsquigarrow (r, \Omega') \Rightarrow \\ &\mathsf{subst}\ \Omega \vdash s \rightsquigarrow (r, \mathsf{subst}\ \Omega')\end{aligned}$$

Proof. Like with the proof of the simulation for LLBC$^+$, we have to introduce auxiliary lemmas for evaluating rvalues and applying reorganizations. All the proofs are straightforward by induction on the evaluation or reorganization derivations, and are actually very similar to the proof of Lemma 17.

We have a lemma similar to the above one for LLBC$^{\#}$ evaluations (omitted).

We need the following framing lemma for $\leq$. Note that as the transformation rules for $\leq$ are quite local we don't need very strong disjointness conditions. The disjointness condition is given by $\mathsf{le\_framable}$ and states that: 1. the fresh identifiers introduced in Ω' are not present in the frame (this allows us to preserve freshness); 2. the initial state Ω and the frame Ω_f don't use the same identifiers, put aside the fact that we allow some borrows in the partial state Ω to be dangling, by pointing to loans in the frame Ω_f (we will need this condition when handling function calls).

Lemma 19 (Frame Rule for $\leq$ in LLBC^{+})**.** *For all Ω, Ω' LLBC^{+} states, we have:*

$$\Omega \leq \Omega' \Rightarrow \forall\ \Omega_f,\ \mathsf{framable}\ \Omega\ \Omega'\ \Omega_f \Rightarrow \quad \Omega \cup \Omega_f \leq \Omega' \cup \Omega_f$$

where:

$$\begin{aligned}
&\mathsf{framable}\ \Omega\ \Omega'\ \Omega_f :=\\
&\quad (\forall\ \ell \in \Omega',\ \ell \notin \Omega \Rightarrow \ell \notin \Omega_f) \wedge && (1)\\
&\quad (\forall\ \sigma,\ \sigma \in \Omega\ \vee\ \sigma \in \Omega' \Rightarrow \sigma \notin \Omega_f) \wedge && (2)\\
&\quad (\forall\ A,\ A \in \Omega\ \vee\ A \in \Omega' \Rightarrow A \notin \Omega_f) \wedge && (3)\\
&\quad (\forall\ \ell \in \Omega,\ \ell \in \Omega_f \Rightarrow\\
&\qquad (\mathsf{borrow}^m\ \ell\ v \in \Omega\ \wedge\ \mathsf{loan}^m\ \ell \notin \Omega) \vee\\
&\qquad (\mathsf{borrow}^s\ \ell \in \Omega\ \wedge\ \mathsf{borrow}^s\ \ell \in \Omega'\ \wedge\ \mathsf{loan}^s\ \ell\ v \notin \Omega')) && (4)
\end{aligned}$$

Proof. We do the proof by induction on $\Omega \leq \Omega'$.

- Reflexive case. Trivial.
- Transitive case. This is the difficult case: if we want to use the induction hypotheses we have to pay attention to the $\mathsf{framable}$ condition. We take care of this mostly by using the substitution lemma 17.

  More precisely, the facts $\mathsf{framable}\ \Omega\ \Omega''\ \Omega_f$ and $\mathsf{framable}\ \Omega''\ \Omega'\ \Omega_f$ don't necessarily hold. However, we can build subst s.t.:

  $$\begin{aligned}
  &\mathsf{framable}\ \Omega\ \mathsf{subst}\ \Omega''\ \Omega_f \wedge\\
  &\mathsf{framable}\ \mathsf{subst}\ \Omega''\ \Omega'\ \Omega_f \wedge\\
  &\Omega \leq \mathsf{subst}\ \Omega'' \leq \Omega'
  \end{aligned}$$

  For instance, if there exists $\sigma \in \Omega''$, we have to prove that $\sigma \notin \Omega_f$. If $\sigma \in \Omega$ or $\sigma \notin \Omega'$ we simply use (2). However, it may happen that $\sigma \notin \Omega$ and $\sigma \notin \Omega'$, if a transformation (e.g., Le-ToSymbolic)

introduces a fresh σ when transforming Ω to Ω'', then another transformation (e.g., Le-MoveValue then Le-ToAbs) removes it when transforming Ω'' to Ω'. In this situation, we just have to pick σ' such that $\sigma' \notin \Omega,\ \Omega'\ \Omega'',\ \Omega_f$ and pose subst such that subst is the identity but on σ where we define it as: $\mathsf{subst}\ \sigma = \sigma'$. We then use 17 to prove: $\Omega \leq \mathsf{subst}\ \Omega'' \leq \Omega'$. By using this technique, we can make sure (2) and (3) are satisfied for Ω and Ω'', and for Ω'' and Ω'; that is:

$$(\forall\ \sigma,\ \sigma \in \Omega\ \vee\ \sigma \in \Omega'' \Rightarrow \sigma \notin \Omega_f) \wedge$$
$$(\forall\ A,\ A \in \Omega\ \vee\ A \in \Omega'' \Rightarrow A \notin \Omega_f)$$

and:

$$(\forall\ \sigma,\ \sigma \in \Omega''\ \vee\ \sigma \in \Omega' \Rightarrow \sigma \notin \Omega_f) \wedge$$
$$(\forall\ A,\ A \in \Omega''\ \vee\ A \in \Omega' \Rightarrow A \notin \Omega_f)$$

(1) and (4) are more difficult. We can make sure that (1) is satisfied for Ω and Ω'' (same technique). Then, given $\ell \in \Omega'$ such that $\ell \notin \Omega''$, we want to show that $\ell \notin \Omega_f$. If $\ell \notin \Omega$ it is trivial by (1). However, it may happen that $\ell \in \Omega$: it means that ℓ was originally in Ω, that we removed it (because of Le-Abs-End-SharedLoan or MergeAbs-Mut), then reused it in a rule which introduces a fresh loan identifier. But this is actually forbidden by (4), and we use this fact.

By induction on $\Omega \leq \Omega'$ we prove that:

$$\begin{array}{l}(\forall\ \ell \in \Omega'',\ \ell \in \Omega_f \Rightarrow \\ \quad (\mathsf{borrow}^m\ \ell\ v \in \Omega''\ \wedge\ \mathsf{loan}^m\ \ell \notin \Omega'') \vee \\ \quad (\mathsf{borrow}^s\ \ell \in \Omega''\ \wedge\ \mathsf{borrow}^s\ \ell \in \Omega'\ \wedge\ \mathsf{loan}^s\ \ell\ v \notin \Omega'))\end{array}$$

We then reuse this to prove (by contradiction and) by induction on $\Omega'' \leq \Omega'$ that: $\forall\ \ell \in \Omega',\ \ell \notin \Omega'' \Rightarrow \ell \notin \Omega_f$.

This allows us to use the induction hypotheses, and conclude the proof.

- Case Le-ToSymbolic. There exist $\Omega_h[.]$, v and σ fresh for Ω_l such that:

$$\text{borrows, loans}, \bot \notin v\ \wedge\ \Omega = \Omega_h[v]\ \wedge\ \Omega' = \Omega_h[\sigma]$$

  Because of (2) in the $\mathsf{framable}\ \Omega\ \Omega'\ \Omega_f$ assumption we have that σ is fresh for $\Omega \cup \Omega_f$. Posing $\Omega'_h[.] := \Omega_h[.] \cup \Omega_f$ we get:

$$\Omega \cup \Omega_f = \Omega'_h[v]\Omega' \cup \Omega_f = \Omega'_h[\sigma]$$

  Hence: $\Omega \cup \Omega_f \leq \Omega' \cup \Omega_f$.

- Case Le-MoveValue. Trivial (we don't introduce fresh identifiers).
- Case Le-Fresh-MutLoan. Similar to the Le-ToSymbolic case.
- Case Le-Fresh-SharedLoan. Similar to the Le-ToSymbolic case.
- Case Le-Reborrow-MutBorrow. Similar to the Le-ToSymbolic case.

- Case LE-FRESH-SHAREDBORROW. Similar to the LE-TOSYMBOLIC case.
- Case LE-REBORROW-SHAREDLOAN. Similar to the LE-TOSYMBOLIC case.
- Case LE-ABS-END-SHAREDLOAN. We use (4) to prove that there is no $\mathsf{borrow}^{s,r}\,\ell \in \Omega_f$.
- Case LE-ABS-END-DUPSHAREDBORROW. Similar to the LE-TOSYMBOLIC case.
- Case LE-REBORROW-SHAREDBORROW. Similar to the LE-TOSYMBOLIC case.
- Case LE-TOABS. Similar to the LE-TOSYMBOLIC case.
- Case LE-MERGEABS. Trivial.
- Cases LE-CLEARABS, LE-ABS-CLEARVALUE, LE-ABS-DECONSTRUCTPAIR, LE-ABS-DECONSTRUCTSUM, LE-ANONVALUE. Trivial.

We need the framing lemma below for evaluation. We have the same disjointness conditions as for the $\leq$ framing lemma (19).

Lemma 20 (Frame Rule for Evaluation in LLBC$^+$)**.** *For all Ω, Ω' LLBC$^+$ states, we have:*

$$\Omega \vdash s \rightsquigarrow (r,\, \Omega') \Rightarrow \forall\, \Omega_f,\ \mathsf{framable}\ \Omega\ \Omega'\ \Omega_f \Rightarrow \quad \Omega \cup \Omega_f \vdash s \rightsquigarrow (r,\, \Omega' \cup \Omega_f)$$

Proof. The proof is similar to the proof of 19. Like with the simulation proofs, we have to split it into several auxiliarly lemmas for the evaluation of rvalues and for reorganizations. Exactly like with 19, the difficult cases are: 1. the transitive case of reorganization, because it allows us to remove borrows from the context; 2. the cases for statement evaluationwhere we need to use the induction hypotheses (typically, E-SEQ-UNIT and E-REORG). We handle those by using the same strategy as in 19.

We now move on to the proof of Theorem 7. We do the proof by induction on the step index. In order to do so we have to generalize the theorem statement a bit.

Lemma 21. *For all Ω and $\Omega^\#$ LLBC$^\#$ states, step n, statement s, and $S^\#$ set of states with outcomes, we have:*

$$\begin{aligned}
&(\forall\ f \in \mathcal{P},\ \mathsf{borrow_checks}\ f) \Rightarrow \Omega \leq \Omega^\# \Rightarrow \Omega^\# \vdash_{\mathsf{llbc\#}}\ s \rightsquigarrow S^\# \Rightarrow \\
&(\Omega \vdash_{\mathsf{llbc}}\ s \underset{n}{\rightsquigarrow} \infty)\ \vee\ (\exists\, \Omega_1,\ \Omega \vdash_{\mathsf{llbc}}\ s \underset{n}{\rightsquigarrow} (\mathsf{panic},\, \Omega_1)) \vee \\
&(\exists\, r\, \Omega_1\, \Omega_1^\#,\ r \in \{(), \mathsf{return}, \mathsf{break}\, i, \mathsf{continue}\, i\} \wedge \\
&\quad \Omega \vdash_{\mathsf{llbc}}\ s \rightsquigarrow (r,\, \Omega_1)\ \wedge\ \Omega_1 \leq \Omega_1^\#\ \wedge\ (r,\, \Omega_1^\#) \in S^\#)
\end{aligned}$$

Proof. We do the proof by induction on the step index.
Case $n = 0$: trivial.
Case $n = n' + 1$:

We do it by induction on the statement s. Note that because we do not dive into function calls, we can prove the statement by induction on the syntax rather than on the derivation rules.

When the statement does not contain a function call, we can reuse the proofs that there is a forward simulation from LLBC$^\#$ to LLBC$^\#$ (also note that only function calls decrease the step index).

There remains the interesting case, that is a call to some function $\mathsf{fn}\ \langle \overrightarrow{\rho} \rangle\ (\overrightarrow{x_i} : \overrightarrow{\tau_i})\ (\overrightarrow{y_j} : \overrightarrow{\tau_j})\ (x_{\mathsf{ret}} : \tau)\ \{\ s\ \}$). We use the fact that the rule E-Call-Symbolic and the predicate borrow_checks were written so that they "match".

Let us illustrate how we do on an example. We start in the state Ω_0 below and evaluate let z = choose(true, move px, move py):

$$
\begin{aligned}
&\Omega_0 = \\
&\quad x0 \mapsto \mathsf{loan}^m\ \ell_{\ell_0} \\
&\quad y0 \mapsto \mathsf{loan}^m\ \ell_{\ell_1} \\
&\quad px \mapsto \mathsf{borrow}^m\ \ell_{\ell_0}\ 0 \\
&\quad py \mapsto \mathsf{borrow}^m\ \ell_{\ell_1}\ 1
\end{aligned}
$$

After pushing the stack (E-PushStack) we get Ω_1 below (the field stack of the state, which contains the list of all the pushed stack variables, is implicit):

$$
\begin{aligned}
&\Omega_1 = \\
&\quad x0 \mapsto \mathsf{loan}^m\ \ell_0 \\
&\quad y0 \mapsto \mathsf{loan}^m\ \ell_1 \\
&\quad px \mapsto \bot \\
&\quad py \mapsto \bot \\
&\quad x_{\mathsf{ret}} \mapsto \bot \\
&\quad x \mapsto \mathsf{borrow}^m\ \ell_0\ 0 \\
&\quad y \mapsto \mathsf{borrow}^m\ \ell_1\ 1
\end{aligned}
$$

The borrow checking assumption gives us that there *exists* some loan identifiers, symbolic values, states Ω_{init} and Ω_{final}, and a set of states with outcomes $S^\#$ such that:

$$
\begin{aligned}
&\Omega_{\mathsf{init}} \vdash \mathsf{choose.body} \rightsquigarrow S^\# \ \wedge \\
&\forall\ res \in S^\#,\ \exists\ \Omega^\#,\ res = (\mathsf{panic},\ \Omega^\#)\ \vee\ (res = (\mathsf{return},\ \Omega^\#)\ \wedge\ \Omega^\# \leq \Omega^\#_{\mathsf{final}})
\end{aligned}
$$

where:

$$
\begin{aligned}
&\Omega_{\mathsf{init}} = \\
&\quad A\,\{\,\mathsf{borrow}^m\ \ell^0_x\ \_,\ \mathsf{borrow}^m\ \ell^0_y\ \_,\ \mathsf{loan}^m\ \ell_x,\ \mathsf{loan}^m\ \ell_y\,\}, \\
&\quad x_{\mathsf{ret}} \mapsto \bot \\
&\quad x \mapsto \mathsf{borrow}^m\ \ell_x\ \sigma_x, \\
&\quad y \mapsto \mathsf{borrow}^m\ \ell_y\ \sigma_y),
\end{aligned}
$$

and:

$$
\begin{array}{l}
\Omega_{\mathsf{final}} = \\
\quad A\,\{\,\mathsf{borrow}^m\ \ell_x^0\ \_,\ \mathsf{borrow}^m\ \ell_y^0\ \_,\ \mathsf{loan}^m\ \ell_z\,\},\\
\quad x_{\mathsf{ret}} \mapsto \mathsf{borrow}^m\ \ell_z\ \sigma_z\\
\quad x \mapsto \bot,\\
\quad y \mapsto \bot,
\end{array}
$$

We want to transform Ω_1 (by using $\leq$ rules) in order to isolate a local state which is equal to Ω_{init}, then apply the framing rules. One issue is that the borrow checking predicate fixes a choice of loans, symbolic values and region abstractions identifiers (the ℓ_x, ℓ_y, etc. above), which is not necessarily the choice which suits us (the borrow checking predicate gives us a "there exist", while we want a "for all"). Fortunately, we can apply the substitution lemmas to get the identifiers we want (17 and 18).

Another issue is that the "dangling" borrows in environment Ω_{init} are pairwise disjoint. This is not necessarily the case in environment Ω_0 for two reasons: 1. we don't enforce a well-formedness condition on the state in the assumptions of our theorem, meaning there can be duplicated mutable borrows (though in practice it won't happen of course); 2. it is perfectly valid to use several times the same shared borrows. We can take care of this problem by introducing reborrowing abstractions (A_0 and A_1 below, and $\overrightarrow{A_{\mathsf{reborrow}}(\rho)}$ in the proof of the general case afterwards) with rules Le-Reborrow-MutBorrow and Le-Reborrow-SharedBorrow: as those rules introduce fresh loan identifiers, we make sure the external "dangling" borrows are pairwise distinct.

Coming back to the example with choose, by repeatedly applying Le-Reborrow-MutBorrow-Abs, Le-Reborrow-SharedBorrow, Le-MergeAbs and Le-ToSymbolic, we get that $\Omega_1 \leq \Omega_2$, where:

$$
\begin{array}{l}
\Omega_2 = \\
\quad x0 \mapsto \mathsf{loan}^m\ \ell_0,\\
\quad y0 \mapsto \mathsf{loan}^m\ \ell_1,\\
\quad px \mapsto \bot,\\
\quad py \mapsto \bot,\\
\quad A_0\,\{\,\mathsf{borrow}^m\ \ell_0\ \_,\ \mathsf{loan}^m\ \ell_x^0\,\},\\
\quad A_1\,\{\,\mathsf{borrow}^m\ \ell_1\ \_,\ \mathsf{loan}^m\ \ell_y^0\,\},\\
\quad \texttt{// Frame is above, local state is below}\\
\quad A\,\{\,\mathsf{borrow}^m\ \ell_x^0\ \_,\ \mathsf{borrow}^m\ \ell_y^0\ \_,\ \mathsf{loan}^m\ \ell_x,\ \mathsf{loan}^m\ \ell_y\,\},\\
\quad x \mapsto \mathsf{borrow}^m\ \ell_x\ \sigma_x,\\
\quad y \mapsto \mathsf{borrow}^m\ \ell_y\ \sigma_y,\\
\quad x_{\mathsf{ret}} \mapsto \bot
\end{array}
$$

We now apply the frame rule and get that:

$$\Omega_2 \vdash \mathsf{choose.body} \rightsquigarrow S^\# \wedge$$
$$\forall\, res \in S^\#,\ \exists\, \Omega^\#,\ res = (\mathsf{panic},\ \Omega^\#)\ \vee\ (res = (\mathsf{return},\ \Omega^\#)\ \wedge\ \Omega^\# \leq \Omega_3^\#)$$

where:

$$\begin{aligned}
&\Omega_3 = \\
&\quad x0 \mapsto \mathsf{loan}^m\ \ell_0, \\
&\quad y0 \mapsto \mathsf{loan}^m\ \ell_1, \\
&\quad px \mapsto \bot, \\
&\quad py \mapsto \bot, \\
&\quad A_0\ \{\ \mathsf{borrow}^m\ \ell_0\ _,\ \mathsf{loan}^m\ \ell_x^0\ \}, \\
&\quad A_1\ \{\ \mathsf{borrow}^m\ \ell_1\ _,\ \mathsf{loan}^m\ \ell_y^0\ \}, \\
&\quad \texttt{// Frame is above, local state is below} \\
&\quad A_{\mathsf{final}}\ \{\ \mathsf{borrow}^m\ \ell_x^0\ _,\ \mathsf{borrow}^m\ \ell_y^0\ _,\ \mathsf{loan}^m\ \ell_z\ \}, \\
&\quad x \mapsto \bot, \\
&\quad y \mapsto \bot, \\
&\quad x_{\mathsf{ret}} \mapsto \mathsf{borrow}^m\ \ell_z\ \sigma_z
\end{aligned}$$

We pop the stack of Ω_3 (rule E-PopStack) and apply the assignemnt to z, and get Ω_4 (once again, we make the stack field explicit):

$$\begin{aligned}
&\Omega_4 = \\
&\quad x0 \mapsto \mathsf{loan}^m\ \ell_0, \\
&\quad y0 \mapsto \mathsf{loan}^m\ \ell_1, \\
&\quad px \mapsto \bot, \\
&\quad py \mapsto \bot, \\
&\quad A_0\ \{\ \mathsf{borrow}^m\ \ell_0\ _,\ \mathsf{loan}^m\ \ell_x^0\ \}, \\
&\quad A_1\ \{\ \mathsf{borrow}^m\ \ell_1\ _,\ \mathsf{loan}^m\ \ell_y^0\ \}, \\
&\quad A_{\mathsf{final}}\ \{\ \mathsf{borrow}^m\ \ell_x^0\ _,\ \mathsf{borrow}^m\ \ell_y^0\ _,\ \mathsf{loan}^m\ \ell_z\ \}, \\
&\quad z \mapsto \mathsf{borrow}^m\ \ell_z\ \sigma_z
\end{aligned}$$

We finally merge the reborrowing abstractions (A_0 and A_1) into A_{final} and get (this is exactly the

state resulting from applying the rule E-Call-Symbolic):

$$
\begin{aligned}
&\Omega_5 = \\
&\quad x0 \mapsto \mathsf{loan}^m\ \ell_0, \\
&\quad y0 \mapsto \mathsf{loan}^m\ \ell_1, \\
&\quad px \mapsto \bot, \\
&\quad py \mapsto \bot, \\
&\quad A_{\mathsf{final}}\ \{\, \mathsf{borrow}^m\ \ell_0\ \_,\ \mathsf{borrow}^m\ \ell_1\ \_,\ \mathsf{loan}^m\ \ell_z\ \}, \\
&\quad z \mapsto \mathsf{borrow}^m\ \ell_z\ \sigma_z
\end{aligned}
$$

We illustrated the proof on the example of choose; we now come back to the general case. We need some auxiliary lemmas.

Lemma 22 (Auxiliary Lemma: Frame Body - Value)**.**

$$
\begin{aligned}
&\mathsf{init\_extern}\ \vec{\rho}\ \tau = \overrightarrow{A^{\mathsf{ext}}(\rho)} \Rightarrow \mathsf{proj\_output}\ \vec{\rho}\ \tau = (v',\ \overrightarrow{A^{\mathsf{ext\_in}}(\rho)}) \Rightarrow \\
&\big(\forall\ \rho,\ A_{\mathsf{init}}(\rho) = \cup\big(A^{\mathsf{ext}}(\rho) \cup A^{\mathsf{ext\_in}}(\rho)\big)\big) \Rightarrow \mathsf{proj\_output}\ \vec{\rho}\ \tau^{\mathsf{out}} = (v_{\mathsf{out}},\ \overrightarrow{A^{\mathsf{out}}(\rho)}) \Rightarrow \\
&\big(\forall\ \rho,\ A_{\mathsf{final}}(\rho) = A^{\mathsf{out}}(\rho) \cup A^{\mathsf{ext}}(\rho)\big) \Rightarrow \big(\forall\ \rho,\ A^{\mathsf{in}}(\rho) = \{\ \mathsf{proj\_input}\ \rho\ v\ \}\big) \Rightarrow \\
&\Big(\forall\ \mathsf{borrow}^s\ \ell \in v,\ \exists\ v,\ \mathsf{loan}^s\ \ell\ v \in \Omega_0^{\#}\Big) \Rightarrow \big(\forall\ \rho,\ A_{\mathsf{sig}}(\rho) = A^{\mathsf{in}}(\rho) \cup A^{\mathsf{out}}(\rho)\big) \Rightarrow \\
&\exists\ \overrightarrow{A_{\mathsf{reborrow}}(\rho)}, \\
&\mathsf{let}\ \Omega^{\#}_{\mathsf{beg}} = \Omega^{\#}_0,\ x \to V[v:\tau],\ x_{\mathsf{ret}} \to \bot \\
&\mathsf{let}\ \Omega^{\#}_f = \Omega^{\#}_0,\ \overrightarrow{A_{\mathsf{reborrow}}(\rho)} \\
&\mathsf{let}\ \Omega^{\#}_{\mathsf{init\_local}} = \overrightarrow{A_{\mathsf{init}}(\rho)},\ x \to V[v':\tau],\ x_{\mathsf{ret}} \to \bot \\
&\mathsf{let}\ \Omega^{\#}_{\mathsf{init}} = \Omega^{\#}_{\mathsf{init\_local}} \cup \Omega^{\#}_f \\
&\mathsf{let}\ \Omega^{\#}_{\mathsf{final\_local}} = \overrightarrow{A_{\mathsf{final}}(\rho)},\ x \to V[\bot],\ \overrightarrow{y \to \bot},\ x_{\mathsf{ret}} \to v_{\mathsf{out}} \\
&\mathsf{let}\ \Omega^{\#}_{\mathsf{final}} = \Omega^{\#}_{\mathsf{final\_local}} \cup \Omega^{\#}_f \\
&\mathsf{let}\ \Omega^{\#}_{\mathsf{end}} = \Omega^{\#}_0,\ \overrightarrow{A_{\mathsf{sig}}(\rho)},\ x \to V[\bot],\ x_{\mathsf{ret}} \to v_{\mathsf{out}} \\
&\Omega^{\#}_{\mathsf{beg}} \le \Omega^{\#}_{\mathsf{init}}\ \wedge\ \Omega_{\mathsf{final}} \le \Omega^{\#}_{\mathsf{end}}\ \wedge\ \mathsf{framable}\ \Omega^{\#}_{\mathsf{init\_local}}\ \Omega^{\#}_{\mathsf{final\_local}}\ \Omega^{\#}_f
\end{aligned}
$$

Proof. By induction on τ.

- Pair. By induction hypotheses and Le-MergeAbs.
- & τ. We apply Le-Reborrow-SharedBorrow twice: once for the A_{init} abstraction and once for the A_{reborrow} abstraction. We note that only InitExtern-Shared, ProjOutput-Shared and ProjInput-Shared apply for the projections (we do case disjunctions on v, $\mathsf{init\_extern}\ \vec{\rho}\ \tau = \overrightarrow{A^{\mathsf{ext}}(\rho)}$, $\mathsf{proj\_output}\ \vec{\rho}\ \tau = (v',\ \overrightarrow{A^{\mathsf{ext\_in}}(\rho)})$ and $\mathsf{proj\_input}\ \rho\ v$). For framable we leverage the freshness of the identifiers introduced by the rules and the fact that the $A_{\mathsf{init}}(\rho)$ and $A_{\mathsf{final}}(\rho)$ abstractions don't have identifiers which are also used elsewhere, at the exception of dangling borrows pointing to loans in the A_{reborrow} abstractions.

- &mut τ. Similar to the & τ case, but this time with Le-Reborrow-MutBorrow-Abs.
- Sum. We apply Le-ToSymbolic (there can't be borrows in sums).
- Literal. We apply Le-ToSymbolic.

Lemma 23 (Auxiliary Lemma: Frame Body)**.**

$$
\begin{array}{l}
\left(\forall\, i,\ \mathsf{init\_extern}\ \vec{\rho}\ \tau_i = \overrightarrow{A_i^{\mathsf{ext}}(\rho)}\right) \Rightarrow \left(\forall\, i,\ \mathsf{proj\_output}\ \vec{\rho}\ \tau_i = (v'_i,\ \overrightarrow{A_i^{\mathsf{ext\_in}}(\rho)})\right) \Rightarrow \\
\left(\forall\, \rho,\ A_{\mathsf{init}}(\rho) = \bigcup_i \left(A_i^{\mathsf{ext}}(\rho) \cup A_i^{\mathsf{ext\_in}}(\rho)\right)\right) \Rightarrow \mathsf{proj\_output}\ \vec{\rho}\ \tau^{\mathsf{out}} = (v_{\mathsf{out}},\ \overrightarrow{A^{\mathsf{out}}(\rho)}) \Rightarrow \\
\left(\forall\, \rho,\ A_{\mathsf{final}}(\rho) = A^{\mathsf{out}}(\rho) \cup \left(\bigcup_i A_i^{\mathsf{ext}}(\rho)\right)\right) \Rightarrow \left(\forall\, \rho,\ A^{\mathsf{in}}(\rho) = \{\ \mathsf{proj\_input}\ \rho\ v_i\ \}\right) \Rightarrow \\
\left(\forall\, \mathsf{borrow}^s\ \ell \in \overrightarrow{v_i},\ \exists\, v,\ \mathsf{loan}^s\ \ell\ v \in \Omega_0^\#\right) \Rightarrow \left(\forall\, \rho,\ A_{\mathsf{sig}}(\rho) = A^{\mathsf{in}}(\rho) \cup A^{\mathsf{out}}(\rho)\right) \Rightarrow \\
\exists\, \overrightarrow{A_{\mathsf{reborrow}}(\rho)}, \\
\mathsf{let}\ \Omega^\#_{\mathsf{beg}} = \Omega_0^\#,\ \overrightarrow{x_i \to v_i},\ \overrightarrow{y \to \bot},\ x_{\mathsf{ret}} \to \bot \\
\mathsf{let}\ \Omega^\#_f = \Omega_0^\#,\ \overrightarrow{A_{\mathsf{reborrow}}(\rho)} \\
\mathsf{let}\ \Omega^\#_{\mathsf{init\_local}} = \overrightarrow{A_{\mathsf{init}}(\rho)},\ \overrightarrow{x_i \to v'_i},\ \overrightarrow{y \to \bot},\ x_{\mathsf{ret}} \to \bot \\
\mathsf{let}\ \Omega^\#_{\mathsf{init}} = \Omega^\#_{\mathsf{init\_local}} \cup \Omega^\#_f \\
\mathsf{let}\ \Omega^\#_{\mathsf{final\_local}} = \overrightarrow{A_{\mathsf{final}}(\rho)},\ \overrightarrow{x_i \to \bot},\ \overrightarrow{y \to \bot},\ x_{\mathsf{ret}} \to v_{\mathsf{out}} \\
\mathsf{let}\ \Omega^\#_{\mathsf{final}} = \Omega^\#_{\mathsf{final\_local}} \cup \Omega^\#_f \\
\mathsf{let}\ \Omega^\#_{\mathsf{end}} = \Omega_0^\#,\ \overrightarrow{A_{\mathsf{sig}}(\rho)},\ \overrightarrow{x_i \to \bot},\ \overrightarrow{y \to \bot},\ x_{\mathsf{ret}} \to v_{\mathsf{out}} \\
\Omega^\#_{\mathsf{beg}} \leq \Omega^\#_{\mathsf{init}} \ \wedge\ \Omega_{\mathsf{final}} \leq \Omega^\#_{\mathsf{end}} \ \wedge\ \mathsf{framable}\ \Omega^\#_{\mathsf{init\_local}}\ \Omega^\#_{\mathsf{final\_local}}\ \Omega^\#_f
\end{array}
$$

Proof. By induction on $\mathsf{length}\ \overrightarrow{x_i}$. The base case is trivial: we note that $\overrightarrow{A_{\mathsf{init}}(\rho)} = \emptyset$ and $\overrightarrow{A_{\mathsf{final}}(\rho)} = \overrightarrow{A_{\mathsf{sig}}(\rho)}$, and pose $\overrightarrow{A_{\mathsf{reborrow}}(\rho)} = \emptyset$. In the recursive case, the list of local input variables is: $x_n,\ \overrightarrow{x_i}$. We use the induction hypothesis with $\Omega_0^\# := \Omega_0^\#,\ x_n \to v_n$, then apply 22 to introduce reborrowing abstractions for v_n.

Given 23, the rest of the proof of the recursive case of 21 is straightforward.

Theorem 21 implies the target theorem 7.

Appendix G

Proof of Join and Collapse

We show the full rules for the join and collapse operations in Figure 12.1, Figure 12.2, and Figure 12.3. In practice, we also sometimes need some more precise rules to join values, that we show in Figure G.1.

We prove Theorem 9.

We introduce an auxiliary function proj_marked to formalize what it means to project values and states to keep only the values coming from the left state or the values coming from the right state. The term proj_marked l v (respectively, proj_marked r v) is the value v where we discard the values marked as coming from the right (respectively, left) state. We naturally extend this definition to region abstractions and states.

$$
\begin{aligned}
&\mathsf{proj\_marked}\ l\ v = v \\
&\mathsf{proj\_marked}\ l\ \boxed{v} = \mathsf{proj\_marked}\ r\ \lfloor \bar{v} \rfloor = v \\
&\mathsf{proj\_marked}\ l\ \lfloor \bar{v} \rfloor = \mathsf{proj\_marked}\ r\ \boxed{v} = \emptyset \\
&\text{... (Omitting tuples, etc.)} \\
&\forall\, m \in \{l, r\},\ \mathsf{proj\_marked}\ m\ A\,\{\ \overrightarrow{v_i}\ \} = A\,\{\ \overrightarrow{\mathsf{proj\_marked}\ m\ v_i}\ \} \\
&\text{... (Omitting the rules for states)}
\end{aligned}
$$

We introduce the auxiliary predicate sloans_incl for the proofs. We need it because of the premises of Join-SharedBorrows and Join-SharedBorrows-Precise (which in turn come from the premises of Le-Reborrow-SharedBorrow). The environments Ω_0 and Ω_1 on the left of $\Omega_0, \Omega_1 \vdash \mathsf{join}\,(,\ v)\ v_0\ v_1$ are actually needed only so that they can be mentioned by those rules.

$$\frac{\textsc{Join-MutBorrows-Precise}\quad \Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ v_0\ v_1 \Downarrow v_2 \mid \overrightarrow{A} \qquad l'_0,\ l'_1,\ l_2,\ A_0,\ A_1,\ A_2 \text{ fresh}}{\begin{array}{c}\Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ (\mathsf{borrow}^m\ \ell_0\ v_0)\ (\mathsf{borrow}^m\ \ell_1\ v_1) \Downarrow \mathsf{borrow}^m\ \ell_2\ v_2 \mid \\ A_0\ \{\boxed{\mathsf{borrow}^m\ \ell_0\ \_},\ \mathsf{loan}^m\ \ell'_0\},\ A_1\ \{\boxed{\mathsf{borrow}^m\ \ell_1\ \_},\ \mathsf{loan}^m\ \ell'_1\}, \\ A_2\ \{\mathsf{borrow}^m\ \ell'_0\ \_,\ \mathsf{borrow}^m\ \ell'_1\ \_,\ \mathsf{loan}^m\ \ell_2\},\ \overrightarrow{A}\end{array}}$$

$$\frac{\begin{array}{c}\textsc{Join-SharedBorrows-Precise}\quad \mathsf{loan}^s\ \ell_0\ v_0 \in \Omega_0 \qquad \mathsf{loan}^s\ \ell_1\ v_1 \in \Omega_1 \\ \text{no borrows, loans, } \bot \in v_0,\ v_1 \qquad \ell'_0,\ \ell'_1,\ \ell_2,\ s_0,\ s_1,\ s_2,\ A_0,\ A_1,\ A_2 \text{ fresh}\end{array}}{\begin{array}{c}\Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ (\mathsf{borrow}^s\ \ell_0)\ (\mathsf{borrow}^s\ \ell_1) \Downarrow \mathsf{borrow}^s\ \ell_2 \mid \\ A_0\ \{\boxed{\mathsf{borrow}^s\ \ell_0},\ \mathsf{loan}^s\ \ell'_0\ s_0\},\ A_1\ \{\boxed{\mathsf{borrow}^s\ \ell_1},\ \mathsf{loan}^s\ \ell'_1\ s_1\},\ A_2\ \{\mathsf{borrow}^s\ \ell'_0,\ \mathsf{borrow}^s\ \ell'_1,\ \mathsf{loan}^s\ \ell_2\ s_2\}\end{array}}$$

Figure G.1: More Precise Rules to Join Borrows

$$\mathsf{sloans\_incl}\ \Omega\ \Omega' := \forall\ \ell\ v,\ \mathsf{loan}^s\ \ell\ v \in \Omega \Rightarrow \exists\ v',\ \mathsf{loan}^s\ \ell\ v' \in \Omega'$$

The proof requires several lemmas.

Lemma 24 (Join-Values-Le). *For all Ω^0_l, Ω^0_r, $\Omega_l[.]$, $\Omega_r[.]$, v_l, v_r, v_j, $\overrightarrow{A}$ we have:*

$$(\forall\ m \in \{l, r\},\ \Omega^0_m \leq \Omega_m[v_m] \wedge \mathsf{sloans\_incl}\ \Omega^0_m\ \Omega_m \wedge \mathsf{wf\_join\_hole}\ \Omega_m[.]\ v_m) \Rightarrow$$
$$\Omega^0_l,\ \Omega^0_r \vdash \mathsf{join}\ (v_l,\ v_r) \rightsquigarrow v_j \mid \overrightarrow{A} \Rightarrow$$
$$\forall\ m \in \{l, r\},\ \Omega^0_m \leq \Omega_m[v_j], \mathsf{proj\_marked}\ m\ \overrightarrow{A} \wedge \mathsf{sloans\_incl}\ \Omega^0_m\ (\Omega_m[v_j], \mathsf{proj\_marked}\ m\ \overrightarrow{A})$$

where:

$$\begin{array}{l}\mathsf{wf\_join\_hole}\ \Omega[.]\ v := \\ \textit{hole of } \Omega[.] \textit{ inside a shared loan} \Rightarrow \bot \notin v\end{array}$$

Proof:
By induction on $\Omega^0_l,\ \Omega^0_r \vdash \mathsf{join}\ (v_l,\ v_r) \rightsquigarrow v_j \mid \overrightarrow{A}$.

- Case Join-Same: trivial by reflexivity of $\leq$.
- Case Join-Symbolic: we use Le-ToSymbolic.
- Cases Join-Bottom-Other, Join-Other-Bottom: we use Le-MoveValue. We need the assumption $\mathsf{wf\_join\_hole}\ \Omega_m[.]\ v_m$ for the premise that the hole of $\Omega_l[.]$ or $\Omega_r[.]$ (depending on whether $v_l = \bot$ or $v_r = \bot$) is not inside a shared loan.
- Case Join-MutBorrows: we the the induction hypothesis for the inner value (with states

$\Omega_l[\mathsf{borrow}^m\ \ell_0\ [.]]$ and $\Omega_r[\mathsf{borrow}^m\ \ell_1\ [.]]$) then Le-Reborrow-MutBorrow-Abs to introduce the fresh region abstraction with the reborrow.

- Case Join-SharedBorrows: we use Le-Reborrow-SharedBorrow; the premises stating that there are corresponding shared loans in the context are proven by using the premise of Join-SharedBorrows in combination with the assumption $\mathsf{sloans\_incl}\ \Omega^0_m\ \Omega_m$.
- Case Join-MutLoans: we use Le-Reborrow-MutLoan-Abs to insert a fresh mutable loan and move the current loan to a fresh region abstraction; the rules for $\prec^{\text{to-abs}}$ (ToAbs-MutBorrow then ToAbs-MutLoan) allow us to conclude that the fresh region abstraction has the proper shape.
- Case Join-SharedLoans: we use Le-Reborrow-SharedLoan.
- Cases Join-MutLoan-Other, Join-Other-MutLoan: we use Le-Fresh-MutLoan-Abs to introduce a fresh loan on the side which doesn't have one, then use the induction hypothesis.
- Cases Join-SharedLoan-Other, Join-Other-SharedLoan: we use Le-Fresh-SharedLoan to introduce a fresh loan on the side which does't have one, then use the induction hypothesis.
- Cases Join-Tuple, Join-Sum: trivial by the induction hypotheses.
- Case Join-Same-MutBorrow: trivial by the induction hypothesis.
- Case Join-Same-SharedLoan: trivial by the induction hypothesis; we have to use the premise that there are no $\bot$ in the inner values to prove that we can maintain the assumption $\mathsf{sloans\_incl}\ \Omega^0_m\ \Omega_m$.
- Case Join-MutBorrows-Precise: same as for the case Join-MutBorrows-Precise, but we have to introduce additional region abstractions (with Le-Reborrow-MutBorrow-Abs).
- Case Join-SharedBorrows-Precise: same as for the case Join-SharedBorrows-Precise, but we have to introduce additional region abstractions (with Le-Reborrow-SharedBorrow).

Lemma 25 (Join-States-Le). *For all Ω^0_l, Ω^0_r, Ω_{acc}, Ω_l, Ω_r, Ω_j we have:*

$$
\begin{aligned}
&(\forall\ m \in \{l, r\},\ \Omega^0_m \leq ((\mathsf{proj\_marked}\ m\ \Omega_{acc}) \cup \Omega_m) \wedge \\
&\mathsf{sloans\_incl}\ \Omega^0_m\ ((\mathsf{proj\_marked}\ m\ \Omega_{acc}) \cup \Omega_m)) \Rightarrow \\
&\Omega^0_l,\ \Omega^0_r \vdash \mathsf{join}\ (\Omega_l,\ \Omega_r) \rightsquigarrow \Omega_j \Rightarrow \\
&\forall\ m \in \{l, r\},\ \Omega^0_m \leq \mathsf{proj\_marked}\ m\ (\Omega_{acc} \cup \Omega_j)
\end{aligned}
$$

Proof:
By induction on $\Omega^0_l,\ \Omega^0_r \vdash \mathsf{join}\ (\Omega_l,\ \Omega_r) \rightsquigarrow \Omega_j$.

- Case Join-Same-Abs: we pose $\Omega'_{acc} := \Omega_{acc},\ A$ (we add the region abstraction A to Ω_{acc}) and use the induction hypothesis.
- Case Join-Same-Anon: similar to case Join-Same-Abs; we pose $\Omega'_{acc} := \Omega_{acc},\ \_ \rightarrow v$ and use the induction hypothesis.
- Case Join-AbsLeft: we pose $\Omega'_{acc} := \Omega_{acc},\ \boxed{A}$ and use the induction hypothesis.
- Case Join-AbsRight: we pose $\Omega'_{acc} := \Omega_{acc},\ \boxed{\overline{A}}$ and use the induction hypothesis.
- Case Join-Var: we use 24, pose $\Omega'_{acc} := \Omega_{acc},\ \overrightarrow{A}$, use the fact that $\mathsf{sloans\_incl}$ is transitive, and use the induction hypothesis.

Lemma 26 (Collapse-Merge-Abs-Le)**.** *For all* Ω_l, Ω_r, Ω_{acc}, Ω_l, Ω_r, Ω_j *we have:*

$$\vdash A_0 \bowtie A_1 = A \Rightarrow \forall\, m \in \{l, r\},\ \mathsf{proj\_marked}\ m\ A_0 \bowtie \mathsf{proj\_marked}\ m\ A_1 = \mathsf{proj\_marked}\ m\ A$$

Proof:
By induction on $\vdash A_0 \bowtie A_1 = A$.

- Case MergeAbs-Union: by the induction hypothesis.
- Case MergeAbs-Mut: by the induction hypothesis.
- Case MergeAbs-Shared: by the induction hypothesis.
- Case MergeAbs-Mut-MarkedLeft: we use MergeAbs-Mut then the induction hypothesis for the left projection, and directly use the induction hypothesis for the right projection.
- Case MergeAbs-Mut-MarkedRight: symmetric of previous case.
- Case MergeAbs-Shared-MarkedLeft: we use MergeAbs-Shared then the induction hypothesis for the left projection, and directly use the induction hypothesis for the right projection.
- Case MergeAbs-Shared-MarkedRight: symmetric of previous case.

Lemma 27 (Collapse-Le)**.** *For all* Ω, Ω' *we have:*

$$\vdash \Omega \searrow \Omega' \Rightarrow \forall\, m \in \{l, r\},\ \mathsf{proj\_marked}\ m\ \Omega \leq \mathsf{proj\_marked}\ m\ \Omega'$$

Proof:
By induction on $\vdash \Omega \searrow \Omega'$.

- Case Collapse-Merge-Abs. By Collapse-Merge-Abs and Le-MergeAbs.
- Case Collapse-Dup-MutBorrow. We note that, for $m \in \{l, r\}$, we have:

$$\begin{aligned}\mathsf{proj\_marked}\ m\ \left(A \cup \{\ \boxed{\mathsf{borrow}^m\ \ell\ \_}, \boxed{\mathsf{borrow}^m\ \ell\ \_}\ \}\right) &= (\mathsf{proj\_marked}\ m\ A) \cup \{\ \mathsf{borrow}^m\ \ell\ \_\ \} \\ &= \mathsf{proj\_marked}\ m\ (A \cup \{\ \mathsf{borrow}^m\ \ell\ \_\ \})\end{aligned}$$

- Case Collapse-Dup-MutLoan. Similar to previous case.
- Case Collapse-Dup-SharedBorrow. Similar to previous case.
- Case Collapse-Dup-SharedLoan. Similar to previous case, but we use Le-ToSymbolic if we need to introduce a fresh symbolic value.

Finally, the combination of theorems 25 and 27 gives us the target theorem (9).