Proof-oriented domain-specific language design for high-assurance software

Denis Merigoux

PSL
UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à l'ENS Paris et Inria (Prosecco)

Proof-Oriented Domain-Specific Language Design for High-Assurance Software

Soutenue par

Denis MERIGOUX

Le 13 décembre 2021

École doctorale n°386

Sciences Mathématiques de Paris Centre

Spécialité

Informatique

Composition du jury :

Karthikeyan BHARGAVAN Directeur de recherche Inria	*Directeur*
Évelyne CONTEJEAN Directrice de recherche CNRS/Université Paris-Saclay	*Examinatrice*
Derek DREYER Professeur MPI-SWS	*Rapporteur*
Sarah LAWSKY Professeure Northwestern Pritzker School of Law	*Examinatrice*
Xavier LEROY Professeur Collège de France/Inria	*Président du jury*
Jonathan PROTZENKO Chargé de recherche Microsoft Research	*Directeur*
Bas SPITTERS Professeur associé Aarhus University	*Rapporteur*

Inria

ENS | PSL

Remerciements

Ô toi, aventureuse lectrice ou aventureux lecteur qui tourne les pages de ce manuscrit, que ton courage soit récompensé par ces quelques mots qui te permettront de toucher du doigt l'incroyable toile invisible de soutiens qui a rendu cet aride et touffu travail scientifique possible. Si je suis le seul à signer cette thèse, c'est bien grâce à l'aide indéfectible de mes proches, amis et collègues. C'est grâce à elles et à eux que j'ai pu garder la motivation et l'envie d'aller toujours plus loin dans cet apprentissage du métier de chercheur, fait de doutes incessants et de questionnements déstabilisants. C'est aussi à elles et eux que je dédie cette dissertation ; je ne pourrais évidemment pas citer chacune et chacun à sa juste valeur ici, aussi je tiens à remercier du fond du cœur toutes celles et ceux qui se reconnaîtront sans être inclus dans ces lignes.

Tout d'abord, c'est grâce à un formidable environnement de travail que j'ai pu prendre mes marques et effectuer un travail dans des conditions très confortables. Merci tout d'abord au Conseil européen de la recherche qui a financé ma thèse au travers de la bourse CIRCUS, et au formidable contrat doctoral de l'Inria avec sa durée légale de travail de 38 h 30 qui m'a apporté surcroît de rémunération et de RTT. En effet, jusqu'en 2021 le contrat doctoral standard était rémunéré 1 769 € bruts mensuels, soit un net d'environ 150 € au dessus du SMIC. Si la loi de programmation de la recherche récemment adoptée prévoit une revalorisation à 2 300 € du contrat doctoral d'ici 2025 (sic), la trop faible rémunération du doctorat par rapport au niveau de qualification des doctorants et le coût de la vie, spécialement en région parisienne, privent cette formation de trop de candidats qui ne peuvent tout simplement pas se permettre financièrement une telle situation pendant de longues années. Évidemment, de nombreux autres aspects du contrat doctoral seraient à réformer, et les conditions dans lesquelles ces contrats sont attribués sont également améliorables, surtout dans les humanités. J'espère que la lutte associée à ces sujets continuera : l'acquis social que constitue le contrat doctoral unique de plein droit institué en 2009 montre que le progrès sur ces sujets est possible, surtout quand je compare la situation

française au milieu anglo-saxon où les doctorants n'ont ni contrat de travail, ni congés, ni même permanence du salaire tout au long de l'année (césure les mois d'été).

Grâce à ce cadre confortable, j'ai pu dédier ces trois années de doctorat à ma recherche et à ses applications, ayant décidé de ne pas prendre de mission d'enseignement. Ainsi les journées au deuxième étage du centre Inria de Paris (hors confinement COVID), jalonnées par les pauses thé et café, ont été l'occasion d'interminables et très productives discussions avec mes chers collègues de laboratoire : Ben "l'ancien", Marina, Natascha, Carmine, Thibaut, Blipp, Florian, Roberto, Kenji, Ram, Aaron pour la vieille génération ; Son, Théophile, Théo, Paul-Nicolas, Aymeric et Xavier pour la jeune génération qui monte malgré le silence des médias ; Cătălin, Bruno, Vincent, Adrien pour les permanents qui assurent la continuité de Prosecco à travers les âges. En dehors de Prosecco, je voudrais remercier François, Jacques-Henri et Pierre-Évariste dont les conseils plus ponctuels mais non moins précieux m'ont été d'une grande aide. Enfin, *last but not least*, l'inimitable Mathieu qui, non content d'être le meilleur assistant de recherche du monde, a réussi à passer de l'autre côté de la barrière pour commencer lui-même son doctorat.

Évidemment, cette liste ne serait pas complète sans les mentions spéciales réservées à mes deux mentors intellectuels, véritables sherpas rompus aux pistes escarpées de la formalisation. Tout d'abord, celui qui m'a ouvert la porte de la vérification appliquée à l'automne 2017 en me proposant de postuler chez Prosecco en tant qu'ingénieur de recherche, le distingué *Principal Cheese Researcher* Jonathan, auteur du best-seller *XUL* de la célèbre série "Les cahiers du programmeur" chez Eyrolles. Je remercie Jonathan qui, tel un Victor Hugo profitant de son exil contraint pour mieux exercer son génie, contribue au rayonnement de la France en dédiant une partie de son temps à la formation de jeunes et naïfs doctorants avec rigueur et humanité, patience et confiance, sagacité et attention. J'espère que nous pourrons à nouveau gravir les montagnes du Pacifique nord-ouest ensemble, à défaut de l'Everest ! Ensuite et logiquement, je suis infiniment reconnaissant d'avoir pu bénéficier de la direction stratégique et visionnaire de la star de la cryptographie vérifiée et mécène de phalanstère artistique, l'unique et formidable Karthik. Toujours présent aux moments critiques, l'autonomie qu'il m'a confiée m'a surpris et honoré, témoignant de sa confiance et sa générosité ; j'espère que j'en aurais été à la hauteur durant ces trois années. Karthik, j'espère également te recroiser prochainement dans une galerie d'art du VI^{e} arrondissement ou une réception mondaine à Paris ou à Pondichéry, soit partout là où ton caractère

insaisissable te mènera. Enfin, je voudrais terminer sur un remerciement à Dan Gohman, qui a été mon directeur de stage à Mozilla, et grâce auquel j'ai acquis le goût des compilateurs et la connaissance de WebAssembly, connaissance qui a été le déclencheur de mon recrutement à Prosecco.

Poursuivant cette liste à la Prévert, il est temps de s'arrêter sur les personnes qui ont peuplé ma version de l'été 2019, à Seattle et au cœur du réacteur de la recherche en méthodes formelles de Microsoft, bizarrement plus fondamentale que celle menée avec son partenaire public Inria ! Merci donc à Nik, Danel, Aseem, Chris, Tahina et Jonathan (encore une fois) de m'avoir fait vivre un concentré de science vivante et stimulante dans les locaux aux fenêtres trop peu nombreuses mais à la cantine fort goûteuse du bâtiment 99. Les discussions passionnées autour du tableau blanc rempli de contre-exemples invalidant notre modèle avorté de logique de séparation resteront à jamais pour moi un exemple vibrant de science en gestation, bouillonnante et virevoltante. Mais ces journées éprouvantes, qui commençaient à 7 h 30 après 30 minutes de transport dans le bus privé de Microsoft à travers les autoroutes urbaines de Seattle, n'auraient pas été complètes sans les soirées en compagnie de mon colocataire devenu cher ami Aymeric, et les autres jeunes recrues de l'été Sydney, Jay et Haobin, avec qui moult IPA houblonnées furent consommées.

Mais au-delà du petit monde des méthodes formelles, j'ai pu étendre mon horizon des possibles grâce à d'extraordinaires rencontres, fortuites ou provoquées par mes recherches. Je veux ici commencer par remercier Sarah, respectable professeure de droit fiscal qui a su maintenir la flamme de sa conviction scientifique malgré sa fulgurante carrière, et qui m'a accueilli avec respect et attention lorsque je me suis présenté à elle, ignorant presque tout de son domaine d'expertise et prétendant apporter une solution au problème qu'elle soulevait. Respect et attention également que je voudrais souligner et remercier chez Liane, dont j'ai passé avec succès les fourches caudines de son examen de probité pour informaticiens, et qui m'a ouvert les portes d'un monde juridique étranger, mais pourtant rapproché de l'informatique par une véritable invasion technoptimiste, qu'il serait maintenant temps de tempérer pour y distinguer le bon grain de l'ivraie. Merci donc à Liane et ses brillantes étudiantes comme Lilya de m'avoir aidé à vérifier mon implémentation du calcul allocations familiales au cours de séances de *pair programming* qui nous ont laissé perplexe quant à la qualité du législateur français. Merci également à Marie qui a bien voulu nous guider Liane et moi dans la jungle de la transformation numérique de l'administration, nous faisant profiter de son expérience durement acquise et de sa tolérance au positivisme scientifique

exceptionnelle pour une sociologue aussi acérée et lucide qu'elle.

De la théorie à l'application, il en fût question lors de cette aventure insolite au pays de l'administration qu'on dit parfois à tort trop administrante, et dont les représentants que j'ai eu la chance de côtoyer ont retoqué nombre de mes préjugés. Merci donc aux damnés du code M, ces forçats de l'impôt qui, tels Atlas portant le monde, soutiennent la bonne marche de l'État avec un flegme impressionnant. Merci aussi à ceux qui ont soutenu ma folle démarche de transfert de technologie au secteur public, à tous les niveaux de la pyramide hiérarchique de Kéops de la DGFiP : James, Laurent, Christophe, Violaine, Lisa, Éric, Tomasz, Audran et tant d'autres. Sans oublier les retraités en pleine forme comme Dominique dont la connaissance du passé vient éclairer l'avenir, et les pionniers du logiciel libre, de la science ouverte et de la régulation algorithmique : Bastien, Soizic et Simon. Évidemment, tout cela n'aurait pas été possible sans le soutien décisif et indéfectible de Raphaël, qui a bien voulu sortir de son rôle de maître d'armes pour écrire de l'OCaml avec moi. Merci également à l'Inria et à sa Direction générale déléguée à la science, en particulier Alain et Jean-Christophe qui ont cru en mon projet et qui m'ont permis de continuer à le porter après la soutenance de cette thèse.

Face à tant de soutiens et de professeurs qui ont assuré ma formation, j'ai eu à cœur de moi-même contribuer au grand cycle de la passation de la connaissance humaine, en focalisant mes efforts sur une transmission individuelle plus que collective. Aussi, je remercie Nicolas et Alain de m'avoir choisi en tant que tuteur de stage, et j'espère qu'ils ne me jugeront pas avec trop de sévérité, ayant été mes premiers cobayes dans le domaine.

À côté du travail, il y a bien sûr les nombreux amis qui ont supporté mes anecdotes interminables sur le calcul de l'impôt avec bonne grâce. Ceux du phlysle bien sûr : Clara, Antoine (×3), Maxime, Gautier, Jules, Grégoire, Gaëtan, José, Dhruv. Les autres ex-compagnons d'infortune du plateau de Saclay : Goeffrey, Matthieu, Nilo, Thomas, Marc, Basile, Camille, Pierre-Cyril, Robin, Bénédicte et tant d'autres. Mais aussi les escrimeurs invétérés et les anciens des repas du jeudi soir : Raphaël, Marion, Adrien, Flora, Arthur, Philippine, Quentin, Vincent, Charles, Mathilde, Alexandre, Romain, Jean-Pierre, Jean-Baptiste, Lara, Marguerite, Nina, Soizic, Thibaut (×2), et Théo.

Enfin, je souhaite terminer par mes soutiens les plus proches et les plus importants : mes parents, André et Sylvie, ma sœur adorée, Celia, mes grands-parents, Michel et Chantal, qui m'ont accompagné sans retenue et avec beaucoup d'amour dans une thèse dont ils avaient parfois du mal à comprendre les tenants et aboutissants. Enfin, je souhaite dédier ce manuscrit à

celle dont ma relation a trouvé son commencement dans cette merveilleuse période du doctorat, qui m'a accompagné au quotidien dans toutes les épreuves que la thèse comportait et qui a été mon inspiration et mon réconfort dans tant des moments heureux ou plus difficiles : ma très chère et tendre Yoko.

Paris, le 23 novembre 2021

List of Publications

J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan, "Formally verified cryptographic web applications in WebAssembly", in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1256–1274. DOI: 10.1109/SP.2019.00064.

N. Swamy, A. Rastogi, A. Fromherz, D. Merigoux, D. Ahman, and G. Martinez, "Steelcore: an extensible concurrent separation logic for effectful dependently typed programs", *Proc. ACM Program. Lang.*, vol. 4, no. ICFP, Aug. 2020. DOI: 10.1145/3409003. [Online]. Available: https://doi.org/10.1145/3409003.

A. Fromherz, A. Rastogi, N. Swamy, S. Gibson, G. Martinez, D. Merigoux, and T. Ramananandro, "Steel: proof-oriented programming in a dependently typed concurrent separation logic", *Proc. ACM Program. Lang.*, vol. 5, no. ICFP, Aug. 2021. DOI: 10.1145/3473590. [Online]. Available: https://doi.org/10.1145/3473590.

D. Merigoux, F. Kiefer, and K. Bhargavan, "Hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust", Inria, Technical Report, Mar. 2021. [Online]. Available: https://hal.inria.fr/hal-03176482.

D. Merigoux, R. Monat, and J. Protzenko, "A modern compiler for the french tax code", in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2021, Virtual, Republic of Korea: Association for Computing Machinery, 2021, pp. 71–82, ISBN: 9781450383257. DOI: 10.1145/3446804.3446850. [Online]. Available: https://doi.org/10.1145/3446804.3446850.

D. Merigoux, N. Chataing, and J. Protzenko, "Catala: a programming language for the law", *Proc. ACM Program. Lang.*, vol. 5, no. ICFP, Aug. 2021. DOI: 10.1145/3473582. [Online]. Available: https://doi.org/10.1145/3473582.

Summary

1. Connecting Program to Proofs with Nimble Languages

Contents

Abstract

There is no mythical origin story like the Tower of Babel to explain the diversity of programming languages and formal methods tools: instead, the idea that different paradigms and frameworks are needed to solve different problems is intuitive to computer scientists [1]. On the other hand, the need for efficiency and avoiding duplication creates a tension that has led to the creation of common infrastructure for formal methods.

In this chapter, we argue that advancing the state of the art for verification implies creating a diversity of tools and domain-specific languages. This raises two issues: first, making sure that efforts are shared across frameworks; second, making sure that various domain-specific languages remain relevant for their target audience, *i.e.* domain experts.

L'outil simple, pauvre, transparent est un humble serviteur ; l'outil élaboré, complexe, secret est un maître arrogant.

(Ivan Illich, La Convivialité*, 1973)*

An ounce of practice is generally worth more than a ton of theory.

(Ernst Friedrich Schumacher, Small is beautiful*, 1973)*

1.1. The Rise and Fall of the General-Purpose Framework

This section paints a broad picture of the different technical methods of the program verification community, and their implication in terms of tooling infrastructure. More particularly, we point out the rising omnipresence of a general-purpose proof assistant in the program verification community: Coq. While it be considered a success and is generally unavoidable in its fields, we claim that its continuous improvement will not be sufficient to significantly push the state of the art. Indeed, it can be argued that Coq is now reaching the limits imposed by its early technical design decisions. The need for specialized tools is thus rising, as well as techniques to connect these diverse tools together.

1.1.1. Background on program verification

This work, although centered around programming language design, is deeply connected to formal methods and one of its active research areas: program verification. The goal of program verification is to prove some properties about the source code of a program, the result it returns upon execution and the eventual side-effects it can have.

The properties that one can prove about a program are diverse. Some are common to all programs : whether the computation terminates or not, whether the program accesses the memory in a well-behaved fashion (memory safety), etc. Some are very specific and relate to what a program is trying to do: in that case, the property that we are trying to prove about the program is akin to a specification of the program, expressed in a more concise or abstract way than the program itself.

For instance, one might want to prove that the source code of a program P implementing a map data structure as an optimized red-black tree does indeed behave like an abstract map. This property can be expressed by attaching to the state of P a ghost abstract map M, with an invariant I stating that a value is stored in P if and only if it is stored in M. Then, the property holds if we can prove that the invariant I is preserved by all the operations in P.

In this chapter, as well as the rest of the dissertation, we will focus on applications of program verification to real-world examples: programs that run in production inside some device or organization and provide services.

This focus is of course very narrow and does not represent the whole diversity and complexity of the program verification research. This editorial choice is guided by our general inclination towards more applied research in the field.

A Fragmented State of the Union The definition we choose for program verification is intentionally broad so as to encompass the various techniques that belong to it. Indeed, the approaches to encode the properties and the programs into a logical system able to perform proofs are very diverse. We will list here the major schools of thought in that regard. Note that we disregard model checking for cyber-physical systems (*e.g.* UPPAAL [2]) in this subsection and in the rest of the dissertation, as we consider program implementation verification as opposed to specification model checking.

Symbolic Execution. Symbolic execution [3] is the earliest form of program verification. As a recent survey [4] reminds us, the principle of symbolic execution is to replace the concrete program inputs with symbols, and propagate those symbols by following the execution of the program. When encountering a conditional, one generally has to consider in parallel the two possible execution paths corresponding to a true or false condition. The result of the program can then be expressed as a logical expression that depends on the inputs symbols. However, since every branching in the programs leads to a new path to consider, the number of paths quickly explodes with the program size: this is "path explosion". Another challenging problem for symbolic execution is unbounded loops, which require inferring a loop invariant. Some tools work around this problem by unrolling all loops to a fixed number of iterations, in an example of *bounded model checking*. The main real-world application of symbolic execution is fuzzing: crafting relevant tests for corner-cases of the program using a source of randomness to explore the input space. Tools like Klee [5] (built on top of LLVM) and SAGE [6] are used in conjunction with fuzzers to find edge cases in all kinds of low-level, system software. This line of research is still active today [7].

Abstract Interpretation. A solution for automatically inferring loop invariants for unbounded loop came from abstract interpretation [8], [9]. This technique, grounded in a solid theoretical framework that can adapt to virtually any programming language semantics, can be succintly described as partitioning the input space into domains, and mapping the evolution of these domains throughout the execution of the program. By proving that a loop iteration does not change the domain of the variables it mutates, a loop invariant

can be inferred. Abstract interpretation has been successfully applied to real-world aerospace software [10], with the famous Astrée [11] analyzer. The downside of abstract interpretation is precisely its strength: the choice of the input domains (and the corresponding abstraction function) can reveal itself tedious. Too large a domain could make the analysis imprecise, while making the domains too specific can create difficult proofs, and lead to an explosion of the analysis resource consumption. Problematically, the choice of the domains is the only way for the user to interact with the prover. This interaction bottleneck makes it difficult to reason about locally-difficult parts of the proof when the whole process is globally automated.

Hoare Logic and Weakest Preconditions. The two previous areas of program verification both focus on mostly automatic program analysis and proofs, given the source program and an initial user-provided tool configuration. However, it is also possible for the developer to guide the proof more finely by annotating each function of the program with pre- and post-conditions. These conditions effectively define a *contract* for the function. The contract itself is backed by a proof on the body of the function: the conclusion of this proof is that the post-condition holds on the function result while assuming the preconditions. On the other side each caller has to check whether the precondition holds, in which case it can assume the post-condition after the function call. This style of program verification corresponds to Hoare logic [12], and it allows for increased proof modularity. Hoare logic is the foundation of program verification frameworks embedded in many interactive proof assistants such as Coq [13] or Isabelle/HOL [14]. The precision and flexibility of interactive proofs based on Hoare logic have allowed program verification on real-world critical software like the seL4 kernel [15]. Moreover, some automation can be provided for this method thanks to the weakest-precondition transformer [16] that composes sub-contracts together to yield a single property which, if valid, achieves the proof. The resulting weakest-precondition property can be encoded in automated solvers (SAT or SMT) and automatically checked. This alliance between interactive proofs and automation is at the heart of program verification frameworks like Why3 [17] or F* [18]. Lastly, some provers chose a mostly automatic approach to program verification while designing their proof obligations around Hoare logic and separation logic for memory-related properties, like Viper [19] and Verifast [20].

Altogether, the techniques described above have turned program verification into a reliable technique that can be used to increase the assurance of real-world software. But we have to be more precise about what the current state

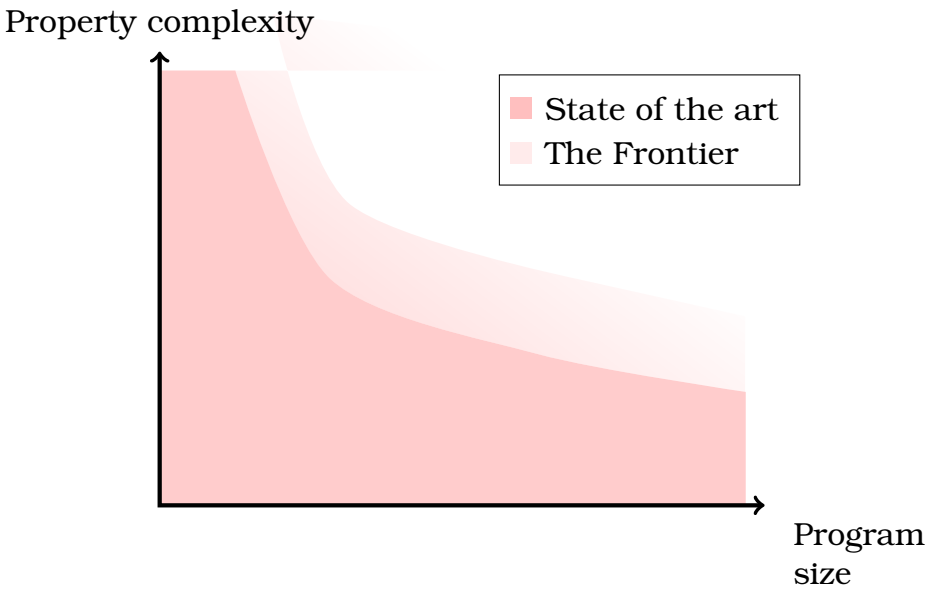

Figure 1.1: Illustration: where is the program verification frontier?

of the art can achieve. For that, consider the illustration of Figure 1.1. We chose to project the state on the art on two axes, program size and property complexity, chosen among many possible others. These axes match our focus on real-world applications of program verification. The gist of this diagram is as follows: we know well how to prove complex properties on small programs, or weak properties on large programs. Let us justify these claims with a few examples.

We will deliberately omit from the following discussions any considerations about the foundational aspect of the technologies, as well as questions about the certification of all their steps and abstractions. As stated before, we are more interested about how these technologies can raise the level of assurance of real-world software. To reach this objective, we are willing to adopt a pragmatic approach to certification requirements and consider the chain of trust in its entirety. As such, the chain of trust is as strong as its weakest link. The weakest link may lie in the absence of certification of a compilation step, but is often not so, as we will argue in Section 1.2.1.

Program Verification Through Error Messages on Steroïds Program verification technology transfer to industry and real-world software historically started using static analysis as its point of entry [21]. The prevalent logic in this

setting is the following: a large organization running critical software wants to improve the safety of its large, legacy codebase. The engineers of this organization do not possess any formal methods skills, but luckily some of them are familiar with typed languages and are used to the compiler error feedback loop. The key to successful transfer, as detailed by this insightful user story on Astrée [22], is to highjack the traditional formal of compiler error messages with warning messages powered by more complex forms of static analysis and program verification. As a well trained engineering team is used to maintain a codebase with the `-Wall` flag enabled, and chase down all warnings, supplying the development effort with more precise and subtle warnings and indications of possible bugs will lead to improvement of the codebase. This trojan horse method has several other industrial-grade proponents, such as Frama-C [23], [24] by the French Atomic Energy Commission (CEA) or Facebook's Infer [25].

While this approach enabled filling the bottom right quadrant of Figure 1.1, it has several drawbacks that limit its scope. First, its usefulness depends greatly on the rate of false negatives and false positives of the underlying static analysis method. Of course, too many false negatives means that the tool is not effective at detecting bugs. But too many false positives can trigger user attrition: when a programmer perceives that the tool is not providing accurate information, she will stop trusting it and will stop caring about warnings. Second, analysis tools can only reason about the bare source code, and often do not have access to the various invariants and logical conditions that underpin the program's behavior. While some tools offer an annotation language that lets users write in a formal fashion, annotating a legacy codebase in such a way is a daunting task beyond the competence and training of a regular software engineer. Hence, static analysis tools that operate on large, legacy codebase are barred from proving complex properties about programs due to this lack of formal global and local context on the source code itself, but also the lack of manual guidance on the difficult parts of the proof.

Program Verification as Applied Mechanized Meta-Theory On the other hand, program verification frameworks embedded in general-purpose proof assistants have allowed to reach the top left quadrant of Figure 1.1. The virtually unbounded expressive power of the proof assistant, combined with the fine control over all the parts of the proof, enable a complete specification and

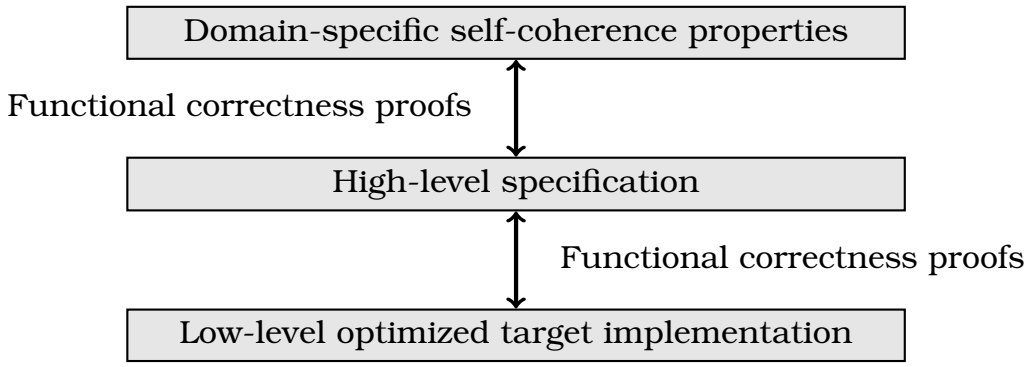

Figure 1.2: High-level schema of a typical program verification architecture inside a general-purpose proof assistant.

proof of a program. The flagship real-world application of this program verification philosophy has traditionally been small to medium-sized low-level systems programming and especially kernels. In their experience report on the verification of the seL4 kernel [26], Klein *et al.* lay out the general ideas of this technique. The verification relies on multiple versions of the same code arranged in a layered infrastructure, from the more abstract to the lower-level: see Figure 1.2. In seL4 as in other applications, the verification starts from a high-level, executable (testable) specification of the program being verified, usually written in the proof assistant's language or in a shallowly embedded language [27]. Then, we write the target, low-level, usually optimized version of the program we wish to verify in the lower layer of this architecture. This target program version is more often deeply embedded than shallowly embedded, since reasoning about the meta-theory of the target programming language can be useful to certify its compilation to a lower-level language.

This architecture enabled a significant progress of the program verification frontier. On the top left quadrant of Figure 1.1, recent work [28] suggest that we can provide a full specification and proof of a small-sized data structure, even under a daunting weak-memory, concurrent computation environment. Then, the crux of the issue turns to pushing the middle section of the verification frontier of Figure 1.1. This section concerns medium-to-large sized programs, and properties like functional correctness or some more complex domain-specific properties. This spot of the state of the art frontier will be the subject of several chapters of this dissertation.

Designing the three components of the stack of Figure 1.2 together has the advantage of minimizing the amount of proofs necessary to glue them. But

each element of this stack is itself so complex, and calls for such a different set of skills, that they often come in separate research artifacts. For instance, inside the Coq proof assistant, Bedrock [29], RefinedC [30] and ClightX [31] provide the bottom part of the stack. The former framework was used as the basis of the CertikOS [32] kernel verification effort. Higher on the stack, Fiat [33] or CoqEAL [34] provide a way to write declarative specifications in Coq that can be used as a basis for program verification.

Lately, devising a successful program verification framework has become a game of building a tangled, high-rising tower of complex abstractions. Moreover, the development effort of this kind of framework follows a law akin to Tsiolkovsky's rocket equation. In the rocket world, the heavier a rocket is, the more fuel you need to lift it. But since fuel is itself heavy, you need more fuel to lift it: the fuel required to lift a rocket to target velocity v depends exponentially on v. In the program verification framework world, the more complex a proof is, the more abstractions you will need to present it in a human-readable format. This proof explosion requires a large amount of advanced engineering, and provides an endless playground for formal methods research. However, we will argue in this dissertation that to further the impact of program verification research, actively involving the end-users of verified programs in industry organizations is required.

1.1.2. Coq, Fiat and Beyond

This draws us back to the main theme of this section, that is to say the rise and fall of the general-purpose proof assistant Coq. While there exist many different general-purpose proof assistants based on different semantics or proof interaction techniques, the one that attracted the biggest community and where most of the proof effort has been spent over the last three decades is Coq, based on the calculus of constructions [35]. The expressive power of this elegant foundation soon raised the hopes for building a mechanically-checked theory of everything, directly following the Bourbakist tradition of French mathematics. Indeed, as mathematicians had begun to accept the validity of computer-aided proofs after the case of the four colors theorem [36], workshops and conferences such as CPP (Certified Programs and Proofs) helped build a community of researchers dedicated to rewriting the foundations of all the formal knowledge in Coq. For instance, the same four colors theorem was proven again in Coq by Gonthier in 2008 [37]. But as the subjects of the proofs continued to complexify, the unity that the community was longing for

began to crumble. Gonthier decided to move his proofs to a dedicated library called SSReflect [38], adding another layer of complexity on top of the original framework.

A recent article [39] emphasizes the threat that this lack of user-friendliness and interoperability poses to research in mathematics. On a more positive note, astounding progress in mathematics formalization is recently coming from the Lean [40] prover community, notably a full proof [41] of a state of the art theorem in analytic geometry [42]. However, this proof is based on an old version of the Lean math library and unification of the libraries is left as future work.

This library diversification phenomenom stems from the same principle that explains why there are multiple programming languages. Different domains require different base assumptions and views of the objects studied that are sometimes irreconcilable, even though they are expressed using the same calculus of inductive constructions. Of course, this common calculus allows for a theoretical possibility of connection between Coq frameworks. The translation of values and proofs between a framework and another can be done via a combination of metaprogramming and tactics, certified or not. When considering the domain of program verification, this line of thought can be pushed towards the logical conclusion presented by Chlipala *et al.* in 2017 [43]. Calling for "the end of history", Chlipala *et al.* condemn the custom implementations of new domain-specific languages in this era where all languages can simply be defined as a meta-theory library inside the Coq proof assistant – using the Fiat [33] framework from the same authors. While the authors lay out unquestionable arguments about their vision, namely the boost in correctness and meta-theoretical interoperability that their proposal brings, some of their arguments can be criticized. For instance, they claim that Coq-embedded domain-specific languages will be easier to learn for newcomers, as their familiarity with Coq's syntax and concepts will provide a strong safety net and basis for language learning, as opposed to a external domain-specific language with a custom implementation and little documentation. In short and pushing the caricature to the extreme, Chlipala *et al.* promote a new world of cosmopolitan Coq proof engineers, spending some time applying their wisdom to a specific domain that has to bend itself to enter the logic of the all-powerful Coq semantics framework.

While deliberately provocative, this last commentary lays out in the open some of the social assumptions behind program verification viewed as a collaborative effort between domain experts and proof engineers. We claim in

this work that program verification is actually an interdisciplinary problem, which comes with all the classical issues of trans-disciplinary collaboration. The recent developments in proof-assistant based program verifications have placed, in our opinion, too much importance in the research for complex tooling for its own sake. Our objective is to push out the program verification frontier as laid out by Figure 1.1 for medium-to-large-sized programs into proving domain-specific properties. To accomplish this objective, we believe that the proof engineering techniques should leave the center of the stage and be put in service of the domain where programs shall be verified.

Looking for the Verification Lingua Franca To conclude this subsection, the hegemony of Coq has to be nuanced. To balance the proof size explosion, interactive proof automation has become an active research domain. This line of works seeks to integrate the automatic theorem provers and SMT solvers used in static analysis tools inside general-purpose proof assistants. Unlike Isabelle/HOL [44] or F* [45] that integrate automation at their heart, reflecting on their design choices, Coq chose early to rely on complex user-provided abstractions to make proof manageable: tactics [46]. This early technical decision made it more difficult to bring proof automation to Coq later on. While some "hammers" have been recently developed [47] to connect Coq to automated provers, their relatively low effectiveness and lack of feedback have slowed their adoption inside the proof engineering community, as reported by a recent survey [48].

Indeed, most automated theorem provers can only work on a restricted logic base, usually first-order. This creates a tension with the higher-order, sometimes dependent-types-heavy style of reasoning that is promoted by the Coq proof assistant. Using the hammer requires translating the higher-order proof obligations and context into the simpler, first-order logic of the prover. This translation is usually lossy with respect to a lot of the proof structure, and requires the prover to rebuild the proof structure, making it harder for it to find the solution with a low ressource consumption. While this problem is acute when considering general-purpose provers and hammers, some of its effects can be mitigated with domain-specific proof assistants. The translation of the higher-order proof obligations to the first-order logic of the prover can be fined-tuned with techniques that are custom to a domain. Hence, the past decade has seen the rise of a number of domain-specific provers such as EasyCrypt [49] and SAW [50] for cryptography, or Why3 [17] for program

verification. This recent trend suggest that domain-specific approaches for proof tooling might be a viable alternative to the construction of more and more massive, unified and abstract proof frameworks in general-purpose proof assistants. The examples provided by this dissertation will later discuss this idea in depth.

1.2. Domain-Specific Languages for Program Verification

After having introduced the general context regarding program verification, it is time to converge on the need for the central concept of this dissertation: domain-specific languages. First, we feature the role such languages can play in repairing the weakest link of the certification chain of trust for high-assurance software. Then, we discuss the strengths and weaknesses of domain-specific languages in general, and make the case for their use in program verification frameworks.

1.2.1. The Specification Problem

In Section 1.1.1, we claimed that the strength of the program verification chain of trust should be asserted by examining its weakest link, and thus that certifying compilation steps may not always be the more efficient way to raise the global level of assurance. We will further discuss this claim here, and raise awareness about an often overlooked link of the verification chain of trust: specification correctness.

In the rest of this section, we will be considering a typical verified programming example involving an interactive proof assistant in which the program to verify is embedded. Examples of that architecture include, as mentioned before, the seL4 [15] kernel where a specification written in Haskell is translated to the Isabelle/HOL proof assistant, then proved equivalent to a C-like implementation embedded inside Isabelle/HOL. This C-like program is then extracted to C, the extraction being justified by a meta-theoretical model of C inside Isabelle/HOL. In this examples as in others, the architecture features a sequence of layers ranging from higher-level and concise to lower-level and optimized. Between the nodes of this sequence, a series of translations act as links. These links effectively form a chain of trust between the top and the bottom of the architecture.

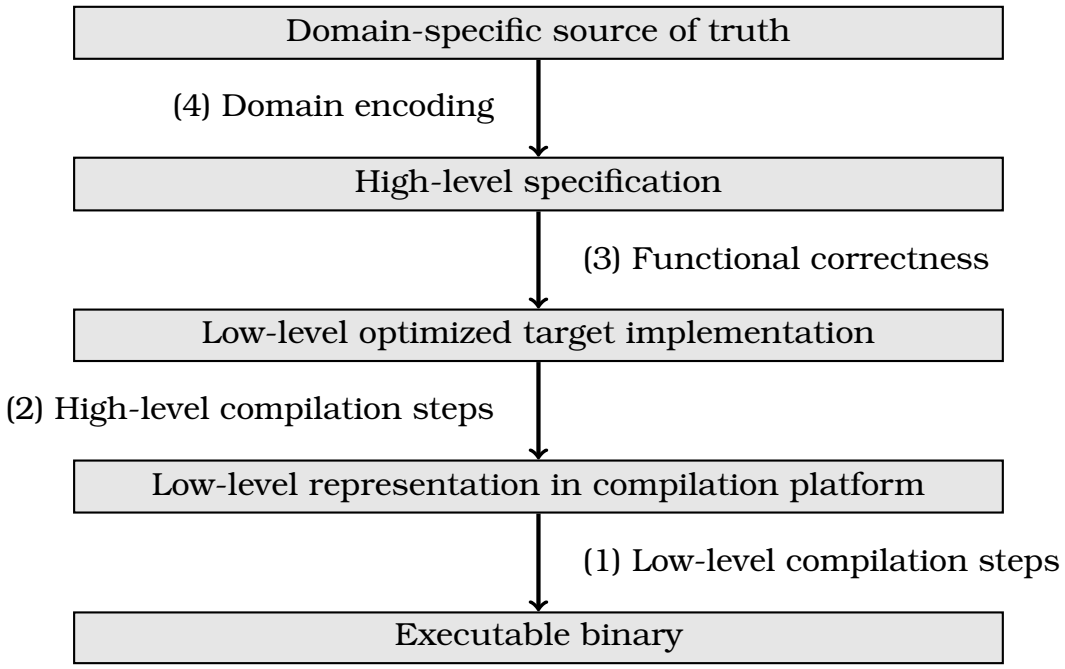

Figure 1.3: High-level schema of a typical program verification chain of trust using a proof assistant.

Tracking Down the Weakest Link Figure 1.3 Illustrates the typical chain of trust. We will discuss the relative vulnerabilities of each link of this chain, starting from the bottom. The diagram starts with the executable binary, but attentive readers know that there is a missing bottom layer to the chain of trust: the hardware. Although recent major hardware attacks like Spectre [51] and Meltdown [52] are reminders that the software assumptions about hardware are frail, we will leave this issue out of the scope of this dissertation.

The link (1) goes from a low-level representation in a compilation platform to an executable binary. In most cases, the low-level representation is C, LLVM IR [53] or WebAssembly [54]. Providing a high level of assurance for (1) is the goal of the field of secure compilation. So far, Compcert [55] remains the only major project capable of delivering C to assembly compilation with a proof of semantics preservation. However, there are many types of properties one might want to prove about a secure compilation; a recent survey [56] provides a classification of such properties, as well as many examples of formal works that provide secure compilations for various language subsets. The major blocking point for the widespread adoption of certified compilers

in real-world software compilation toolchains is the performance of the generated code. Indeed, proving the correctness of compiler optimizations is a daunting task. While some work has been performed to improve CompCert's performance [57]–[59], it is still stuck at the level of an `-O1` optimization flag for GCC or LLVM. Hence, industrial users are often left with the dilemma of choosing between proven certification and performance. Some industrial sectors like aerospace choose the first, while others like cryptography might choose the second due to their performance constraints. However, C compilers like LLVM or GCC are heavily tested and enjoy a relatively high level of assurance, continuously improved by techniques and tools coming from academia, like the CSmith fuzzer [60].

Moving on to link (2) and the low-level optimized target implementation. This implementation, in the case of a verified programming development, is usually written in a shallowly or deeply embedded language inside a proof assistant. For instance, seL4 relies on a deeply embedded model of C inside Isabelle/HOL [61], while Fiat-crypto [62] has a built-in semantics of a domain-specific subset of C. The assurance of link (2) is thus directly related to the semantics model of the low-level language inside the proof assistant. A deeply-embedded low-level language, accompanied by a semantics preservation proof covering the translation from it to the target (usually C), offers the highest level of assurance. The downside of this certified approach is the daunting task of formalizing a big enough subset of C inside a proof assistant, while avoiding the pitfalls of undefined behavior; Clight [63], deeply embedded in Coq, is the reference in that area. So, when can (2) be considered the weakest link of the trust chain? The vulnerability of this translation to bugs increases with the domain covered by the low-level language in which the optimized implementation is written. If optimized implementation depend on aliased pointers, assembly and `goto`s in a weak memory model, then the standard in existing literature requires a deeply embedded model of C (and/or assembly). This might not be the case when the optimized implementation merely requires arithmetic operations and regular functions. The smaller the subset, the smaller the risk of introducing bugs in the translation. But in all cases, the baseline for trust in existing literature is to have a formalization of the languages involved, and at least an external, non-mechanized meta-theoretical argument for the correctness of the translation.

Up next, we examine the link (3) between the high-level specification and the low-level, optimized target implementation. Again, the high-level specification is usually encoded as a shallowly or deeply embedded language inside a proof

assistant. This embedding is crucial because (3) is where lie the bulk of the domain-specific functional correctness proof for the verified development. This functional correctness proof is therefore not the go-to suspect for the weakest link of the trust chain, since it is precisely the step that program verification attempts at making high-assurance. Note that while the high-level specification can be written directly in the proof assistant [62], [64], [65], another approach later developed in this dissertation is to have the specification written in another high-level language, and then translated to the proof assistant (for instance, seL4 and Haskell [15]). In that case, this translation becomes part of the trust chain and needs an argument to justify its correctness.

Last but not least, we raise the question of the provenance of the high-level specification. What is the argument for their correctness? So far, the trusted computing base that can be found in links (1), (2) and (3) is not domain-specific. It concern various translations, and sometimes the proof assistant itself. While foundational approaches like Coq's meta-theory [66] assuredly enjoy a maximum level of assurance, regular testing of a less foundational proof assistant by a community of users yields some assurance by continuous improvements, using the same assurance arguments for mainstream C compilers like LLVM and GCC against CompCert. But link (4) is domain-specific, and usually only receives attention from the domain-specific programmers that participate in the implementation of the verified development. To answer the question of whether it is or not the weakest link, we have to enter the domain-specific matters that the program being verified aims to tackle.

Proving or Trusting Domain-Specific Truth Sometimes, the high-level specifications can themselves be derived from a more high-level, more abstract and mathematical property. As we will see in Chapter 2, the specifications of the cryptographic primitives and protocol enjoy security properties like computational security, confidentiality and authentication, forward security, etc.

Then, an additional layer of proof can be added on top of the high-level specifications to check whether they satisfy those meta-properties. These proofs can either be written inside the main proof assistant, or require an external domain-specific prover. The latter is often true for cryptography, as a recent survey [67] points out. Depending on whether the property we want to prove is expressible in the symbolic or in the computational model

of the cryptographic application, one might chose a different solver. Since the external solver has its own language to express the program and the properties to prove, this step adds a fifth link to the chain of trust, with a translation certification required to obtain the maximum assurance level.

However, not all specifications enjoy abstract mathematical properties that ensure their internal coherence. In Part II of this dissertation, the source of truth for the specifications is the text of the law, which is a social rather than mathematical construction. In those cases, the link (4) of the chain of trust is immaterial, and represents the assurance that the high-level specifications correctly encodes the domain-specific source of truth.

But when the domain-specific source of truth is not formal in the first place, link (4) relies on custom interpretations or decisions made by the writer of the formal specification about what would happen in an situation ambiguous for the domain-specific source of truth. When the writer of the formal specification does not possess the sufficient domain-specific expertise, such disambiguations should be reviewed by a proper domain expert. This reviewing and validation of the specification is crucial to ensure a high-level of assurance of step (4).

How can we have a non-computer scientist domain-expert review and validate a formal specification of the domain-specific truth? If the specification is written in the language of the proof assistant, then the domain expert can be startled by the foreign concepts and syntax of the code presented before her, making the validation difficult if not impossible in practice.

Even in domains like cryptography where specifications derive from some sort of higher mathematical theory, existing verified cryptographic libraries do not always package a proof layer for the specifications, by lack of proof engineering ressources or because the mathematical theory behind the specifications is beyond the verification state of the art. In this situation, the same domain-specific validation problem arises; Chapter 3 explains this problem in detail.

Languages for Domain Specifications Based on the careful examination above, we conclude that link (4) can be the weakest link in verified developments and therefore deserves a greater deal of attention that what can be found in the literature so far. Thus, we need a medium of communication between the domain experts and the formalizers that write the specification. In this dissertation, we argue that domain-specific languages can act as the

medium of communication, when they are designed with this objective in mind. These domain-specific languages can be shallowly embedded in a proof assistant as suggested by Chlipala *et al.* [43], or benefit from a custom frontend implementation, external syntax and tooling.

The important point of our proposal is to design these specification-friendly domain-specific languages with the later proofs in mind. Usually, there is a tradeoff between language complexity and program complexity: the more advanced and domain-specific features a programming language offers, the simpler the programs one can write inside it. Simpler programs can be reviewed more thoroughly and are generally more trustable, as high-level specifications. But making the programming language too complex results in an uneasy formalization, making the connexion with the proof assistants virtually impossible. This tradeoff will be especially explored in Chapter 6.

To sum up the direction of our work, we use the label *proof-oriented domain-specific languages* for domain-specific languages designed with a formal application in mind, usually a verified programming development. The term *proof oriented programming language* is not new, as reminded by Aymeric Fromherz in his recent dissertation [68]:

> "To the best of our knowledge, this term was first used by Hoffmann [69] to describe Lucid, a language that 'uses the same denotation for writing and proving properties of programs, thus is, at the same time, a formal proof system and a programming language'. More recently, Jean-Karim Zinzindohoué [70] used the same term to present F⋆, a general purpose programming language aimed at program verification that we use throughout this thesis."

Examples of such proof-oriented domain-specific programming languages already exist in the literature: Usuba [71], Cryptol [72] and Jasmin [73] are good examples in the domain of cryptography. This dissertation merely makes this concept explicit and proposes a methodology to design and deploy those languages. But before, let us discuss the specificities of domain-specific languages compared to their general-purpose counterparts.

1.2.2. The Domain-Specific Solution

The topic of domain-specific languages has been extensively studied for a long time, and this subsection will merely summarize the main points from existing works [74]–[78].

The frontier between a domain-specific and a general-purpose programming language has always been blurry. Because of its community of programmers and library ecosystem, a general-purpose programming language can specialize over time in one or several domains. For instance, C for embedded systems, Java for business logic, Go for distributed systems, etc. On the other hand, some languages initially thought for a specific domain or use have gained so much popularity that they have *de facto* become general-purpose: Javascript, initially designed in 10 days for animating Web pages [79], or Python, initially suited for "throw-away programming" and "rapid prototyping" [80].

Hence, we need to be more precise as to the scope of the domain-specific languages that are of interest to us. Here, we focus on domain-specific languages whose syntax and semantics is designed with the explicit purpose of improving communication with domain experts, while retaining executability. Performance and optimization are also major features of domain-specific language implementation, but we shall put them second in our priority list, since we are interested in writing high-level specifications that will act as sources of truth for more optimized implementations.

Conciseness as the Key to Correctness The crux of the domain-specific correctness issue boils down to code reviewing by domain experts. And the main factor of external reviewing ineffectiveness is program verbosity. Some verbosity cannot be avoided, when it comes from the domain itself. But some sources of verbosity can be dealt with language design. The simplest technique is to encapsulate domain-specific concepts inside a library that exposes high-level functions taken for granted. Next, syntax is a good candidate for improvement, as defining syntactic sugars can help reduce the amount of boilerplate code required.

But sometimes, verbosity comes from a semantics mismatch between the programming concepts and the source domain concepts. A staple of domain-specific language design is the handling of complicated conditional structures, that can only be expressed poorly with `if`/`then`/`else` statements; but in a clearer form using decision tables [81] or a negation as failure [82]. A paradigm switch, from imperative to declarative, or from object-oriented to functional, can also reduce verbosity. Other fruitful paradigms for domain-specific languages include aspect-oriented programming [83] for enforcing separation of concern in some applications, synchronous programming [84] for cyber-physical systems or functional reactive programming [85] for specifying

interfaces and user interactions.

The traditional way of software engineering to code conciseness, abstraction through functions, classes or modules, can be unadapted to some domains. Business-related software engineering uses the term of *business rules* [86] to describe the way enterprise-ready specifications are described. In this setting, the specifications are viewed as some kind of evolving data (stored in a database) with a particularly complex and intricate structure. State-of-the-art Business rules management systems such as Drools [87] are centered around the database, and require a heavy runtime system to execute while offering a versatile frontend for domain experts that can enter the business rules in the syntax of their choosing.

Domain-specific language design can draw from all of those techniques to cut down verbosity and foster communication with domain experts. However, the complexity of its design comes with the tradeoff of the complexity of its implementation.

Embedded and External Implementations Domain-specific languages are traditionally divided between the *external* and *embedded* (or *internal*) categories. These categories relate to the way the languages are implemented. Embedded domain-specific languages piggyback on a host programming language, from which they can borrow the syntax or semantics, fully or partially. Thus, an embedded domain-specific language can merely consist of a library in the host language with a few macro-defined syntactic sugars. All embedded domain-specific languages reuse at least some part of the tooling of the host language, usually the compiler.

On the other hand, an externally-implemented domain-specific language possesses its own tooling, especially the compiler. An external implementation allows for more flexibility in the syntax and semantics, as the compiler for a domain-specific language can be arbitrarily complex. However, external implementations do have the effect of fragmenting the community as users may have to learn a completely different set of syntax, compiler options, etc. Some of this fragmenting can be mitigated by reusing an existing syntax family (ML, C, Java, etc), or follow interface standards for the tooling. Another downside of external implementations is the big development effort required to produce the tooling that is expected of a modern programming language: good error messages, linting, etc. There again, this effort can be mitigated by choosing for the implementation of the compiler a functional language with a

solid compiler-related library ecosystem like OCaml.

Recently, a third way of domain-specific language implementation has risen in the form of language workbenches [88], [89] or, closer to the formal world, "a programming language for creating new programming languages" [90], Racket [91]. While this approach has yielded promising results in terms of implementation efficiency and expressiveness, several challenges remain for this way to connect elegantly to the world of proof assistants and program verification. As a recent Racket survey [92] points out, tying typechecking rules to language constructs is difficult, leaving users to fallback on a manual external typechecker implementation. On ther other side, semantics framework such as the K framework [93] are currently oblivious of the implementation of the language defined. Future work will maybe see a convergence between those two lines of research.

In this dissertation, we will use a mix of embedded and external domain-specific languages. Our implementations are backed by a formalization of the domain-specific language, at least on paper, if not mechanized inside a proof assistant and accompanied by a meta-theory.

Limitations of Domain-Specific Languages To offset the advantages presented above, we list here the usual limitations of domain-specific language. These limitation stem from their specificity: fragmentation of the user base and community, need for the users to learn a new syntax or new programming paradigm, difficulty and cost of maintaining specific tooling. Domain-specific languages for verification suffer even more harshly from those drawbacks, as the initial size of the verification community is already quite small. In his PhD dissertation, Théo Zimmermann [94] reveals that despite having been created in 1984, Coq only truly opened its development to external contributors in 2015 :

> "Even if the development today is entirely open, transparent, distributed, and online, with the number of contributors steadily increasing and now well over a hundred, it is still the case than the main developers are Inria employees, and that Coq lacks the huge environmental support that designers of a language within a large company can benefit from. Furthermore, this is not just a matter of financial support. Given the complexity of the Coq system, a high level of expertise is needed to contribute, which excludes most software engineers without strong mathematical and computer science

> backgrounds, and means that, like for most scientific software [95], the developers are mainly researchers, or engineers with research experience."

Hence, the biggest proof assistant in the field already suffers from flaws that usually affect domain-specific languages. Consequently, domain-specific languages embedded in Coq are even more heavily impacted by this problem. The F⋆ proof assistant, that will be extensively featured in this dissertation, has similar problems since its user base is smaller than Coq's.

Given the low adoption of program verification by industrial users, we believe we cannot wait until the user base of these tools grows sufficiently to address these problems. So, we advocate here for domain-specific languages whose main public are experts: domain experts and language design experts. The social implications of this statement will be discussed in the next section.

1.3. A Novel Methodology for Pushing the Verification Frontier

Equipped with the context presented in the last two sections, we set out to introduce the thesis of this dissertation. To push the program verification state of the art frontier, we claim that a domain-specific approach can yield significant contributions both in terms of raising the level of assurance on the correctness of the development, and enabling specialized proof tactics and tools. To support this domain-specific approach, we propose a novel methodology that starts from an existing codebase, and uses domain-specific language design to carve out a formal, executable subset. This subset serves a dual purpose. First, it can be used as a medium of communication with domain experts to validate the code, as discussed in Section 1.2.2. Second, the subset should carry proof obligations related to the functional correctness of the program, that can then be fed to off-the-shelf or custom proof backends.

1.3.1. Carving Out Subsets and Connecting to Proofs

In this section, we propose a methodology for incrementally verifying an existing codebase. This endeavor involves several characters who will play a different role: let us introduce them now. The main character, which we will eventually describe as the *language architect*, is responsible for coordinating

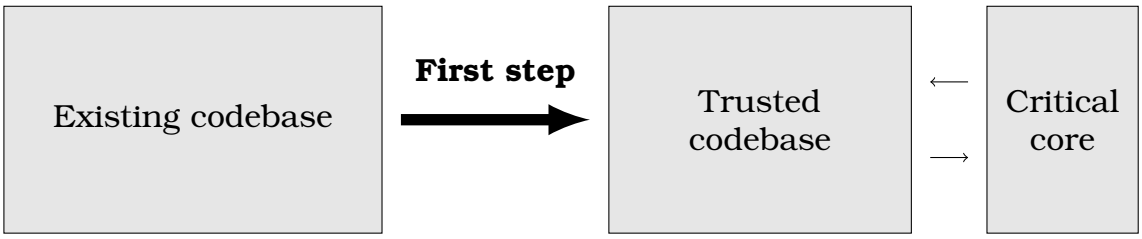

Figure 1.4: First step of the methodology: tidy up and isolate the critical code parts.

all the phases of the work. The original maintainers of the codebase are regular *engineers*, while the people working on the proofs shall be called *proof engineers*. Finally, we use the term *formalizers* to designate the people in the process that possess a training in formal methods, and *domain experts* to designate the people that have a deep understanding of the sometimes informal requirements of the system in the context of its domain-specific use.

The methodology requires a specific scenario we envision for the intervention of the formalizers. The starting point of our program verification journey is an existing real-world codebase, whose level of assurance we want to raise. This codebase is usually written in a mainstream programming language, but the formalizers can leverage a human connection with the engineers that maintain the codebase, who are willing to refactor or replace certain parts of it provided reasonable guarantees. This scenario assumes a collaborative relationship between the formalizers and the domain experts who have control over the codebase. This relationship is key to technology transfer, as the most foundational and proven artefact may not enter a production codebase if its creator does not establish trust with the maintainer of that codebase.

The **first step** (Figure 1.4) of the methodology for the formalizers is to divide up the codebase between the core that will be verified, and the part that shall remain part of the trusted computed base. This first step is usually time-consuming as it requires refactoring the critical core of the codebase into separate modules with clean interfaces. If the critical section of the codebase is heavily optimized and written in an obscure style, then a higher-level, concise and more readable version of the code should be written, that will act as a reference for the optimized implementation. Note that the first step of this methodology coincides perfectly with good software engineering practices. It is also the occasion to take a deeper look at the critical core of the code and fix some bugs before even using verification tools. Actually, it should be

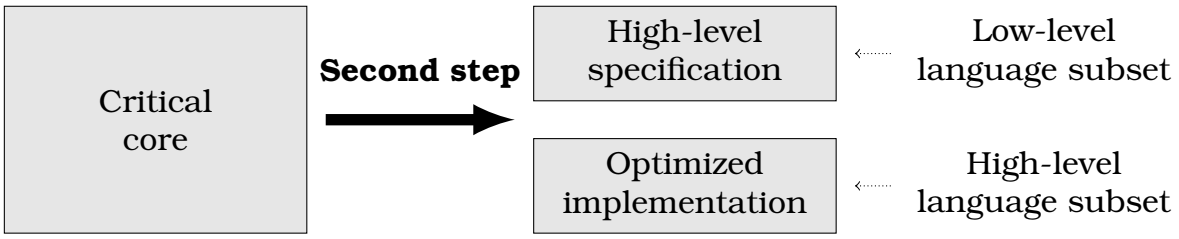

Figure 1.5: Second step of the methodology: isolating the language subset(s) – or domain-specific language(s) – for the core.

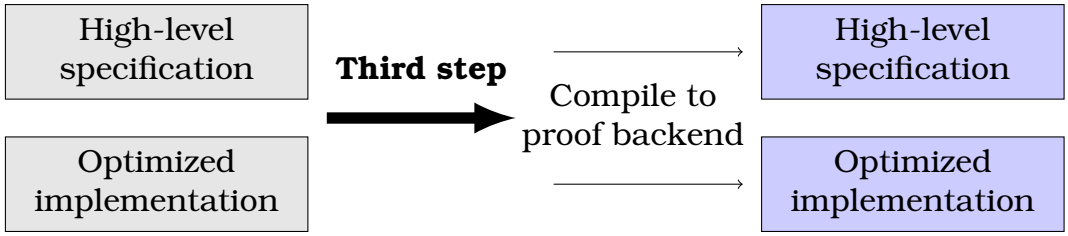

Figure 1.6: Third step of the methodology: translating the core code into the proof backends.

reminded that this first step is sufficient for most projects that do not require a very high level of assurance.

The **second step** (Figure 1.5) of the methodology is to carefully examine the critical part of the codebase that has been put aside. The goal of this examination is to determine and minimize the set of features that the critical code is exercising inside the host programming language. For this step, a tradeoff similar to the one presented in Section 1.2.2 has to be determined between the verbosity and readability of the code, and the complexity of the language features that it uses. The closer the language subset is to lambda calculus or another semantic base, the easier the next steps will be. But the more verbose and ugly the code is – especially the high-level, reference version of it – the harder it will be to get domain experts on board. This step is where the discussion with domain experts is the most intense, since the formalizers have to co-design with them a subset of the host language that will be the target of the verification process. If there are two versions of the critical code, the high-level one and the optimized one, they can live in two different subsets of the host language.

The **third step** (Figure 1.6) of the methodology is to formalize the one or two subsets of the host language identified by the previous step. The formalization can build on previous works, mechanized or not, concerning formalizations

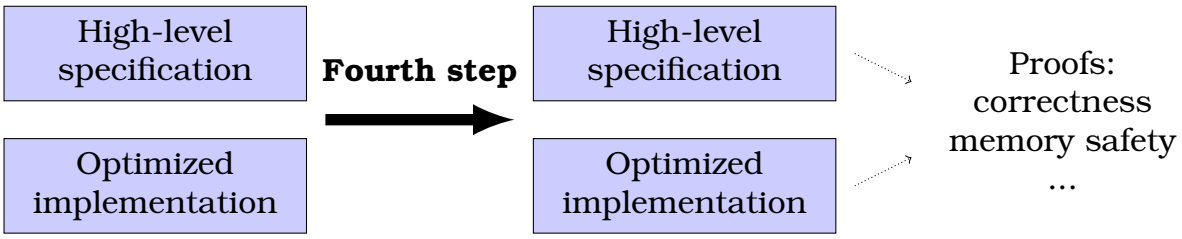

Figure 1.7: Fourth step of the methodology: using proof backends on compiled core code.

of the complete host language. The formalization can then be implemented and tested with a subset-specific compiler, that can also build on existing infrastructure for the host language. At this point, formalizers are doing the heavy lifting, but interactions with domain experts are needed to determine the trusted computed base of the verification endeavor. Some libraries of the host language used in the critical code can be made to fit within the formalized subset, or rather be considered as primitives and made part of the trusted computing base. Then, the subset-specific compiler can be extended with translations to proof backend languages: languages of proof assistants, encoding of proof obligations. These translations can be certified mechanically or just formalized on paper, depending on the target level of assurance and the state of the whole trust chain.

The **fourth step** (Figure 1.7) of the methodology involves most of the actual proofs of the verified programming development. Once the critical code has been translated in a proof backend, proof engineers (who can be distinct from the formalizers) take it from here. Several kind of proofs can be developed: domain-specific specifications proofs to ensure the coherence of the high-level specifications, functional correctness proofs between the high-level and the optimized versions of the same critical code, memory-safety or security-related proofs on the optimized implementation, etc. Critically, not all proofs of this step have to be made with the same proof backend. The ability of the subset-specific compiler to target multiple proof backends can be used to take advantage of the specificities of each prover, and choose the right tool for each different job. Of course, having different provers interact might introduce breaches to the chain of trust, but the process is semantically controlled by the shared formalization of the source subset.

Finally, an optional **fifth step** (Figure 1.8) can be added to this methodology

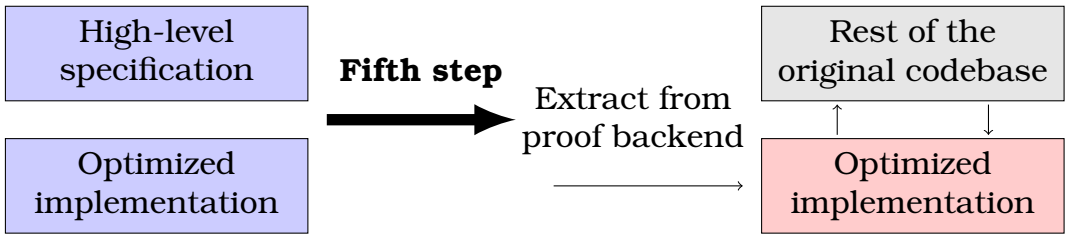

Figure 1.8: Optional fifth step of the methodology: extract verified code and link back.

in order to loop back to the original source code. So far, we have assumed that the proof backends are used in a model-checking fashion. But proof backends can also be used to generate verified code via extraction and compilation techniques. If that is the case, the proof backend can generate an executable, optimized implementation whose functional correctness with respect to the high-level specification has been verified. This generated optimized implementation can then be plugged back in the source critical code via direct source code integration, or via the use of foreign-language interface bindings if there is a mismatch between the host language and the language generated via proof backend extraction.

Figure 1.9 shows the final state of the methodology, after having applied the five steps in the most complete setting with different implementation and specification domain-specific languages. The final diagram for less comprehensive applications of the methodology can be obtained by removing blocks from the picture, while keeping the translation and equivalence links in place when possible.

The important aspect of this methodology is its flexibility. The list of steps is meant to be adapted to each domain, and especially the state of the art of languages, compilers and proof backends available for the domain. The overall goal is to connect programs to proofs with nimble languages, thus the participants of this process should constantly question their choices, whether it is to create a new language or intermediate representations, use existing ones and formalize them, tackle difficult and complex properties or focus on the easy ones that represent the main risk for high-assurance.

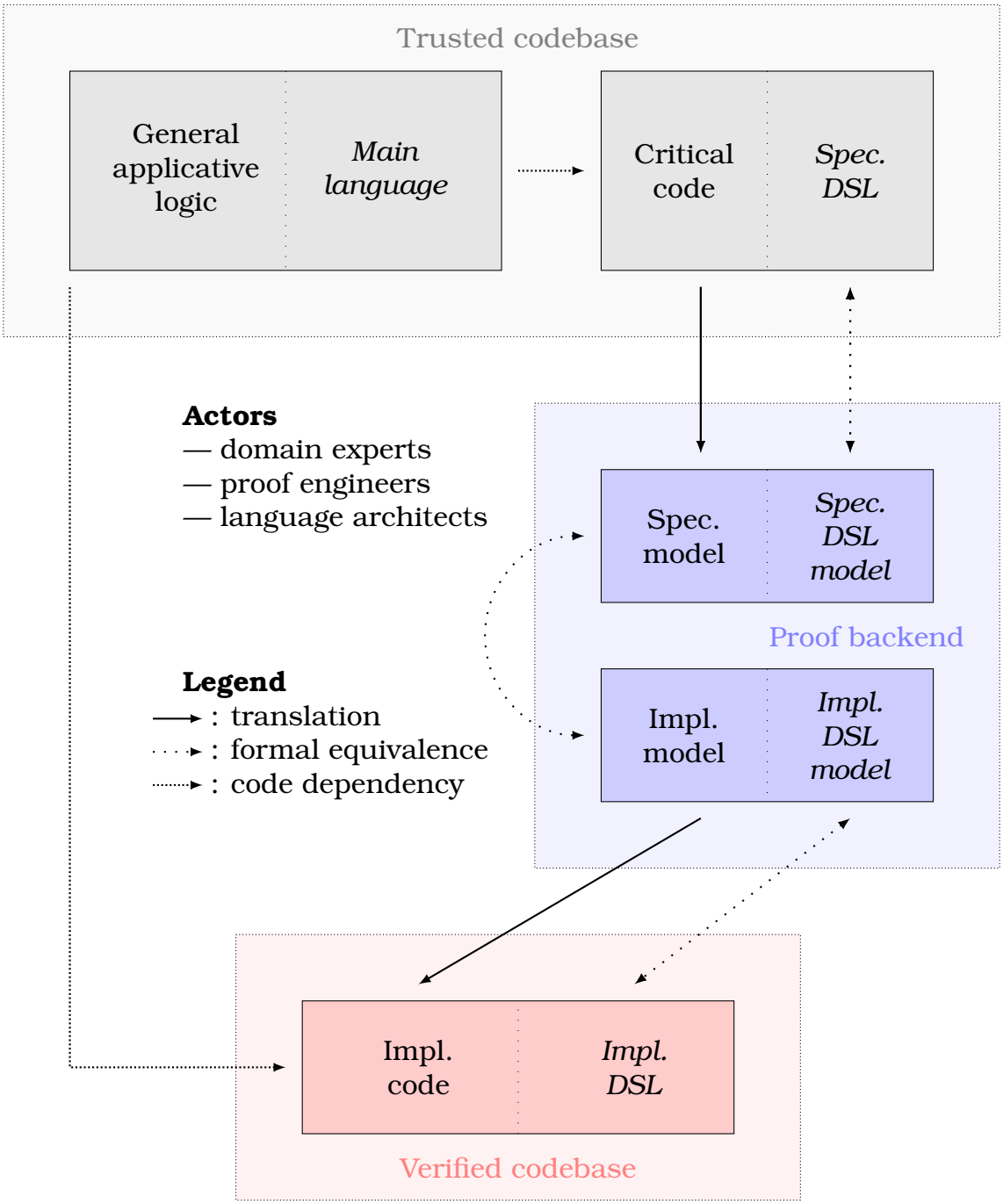

Figure 1.9: Overview illustration of the methodology proposed in this dissertation. Inside each block picturing a chunk of the software, the left part labels the code while the right part labels the language in which the code is written. Translations and equivalence proofs can be done manually or mechanically.

1.3.2. Proof-Oriented Domain-Specific Languages

At the heart of the methodology presented in Section 1.3.1 lies the concept of domain-specific language, discussed in Section 1.2.2. While domain-specific languages are an old concept in formal methods, the contribution of this dissertation is to integrate the needs of later proof developments in their design. Hence, we coin the concept of *proof-oriented* domain-specific language design.

Designing a domain-specific language with later proof developments in mind has several implications. First and foremost, the features of the domain-specific language must remain completely formalized at all times. Nevertheless, this is difficult to accommodate with the need of users and domain experts for customized syntax and language features. To solve this problem, we advocate using a compiler architecture similar to CompCert [55] or Nanopass [96], centered on a restricted, regular and formalized intermediate representation as simple as possible, layered with as much syntactic sugar as needed on top of it.

Second, the language design should include space for proof obligations that will come with the program. These obligations can either come from contracts written by the user (in a contract domain-specific sub-language), or be automatically generated and inserted by the compilation chain depending on syntactic or semantic criteria. To benefit from the specialization of the proof backends, the handling of proof obligations should be as structured as possible, with sources of obligations separated by domains and attached close to their location in the program. With this kind of architecture, the domain-specific compiler can be turned into a proof platform that distributes the workload as several packages, each sent to the right tool that can prove it. In this strategy, crafting a good program verification stack is all about reducing the complexity of the program space to its very essence, and applying a divide and conquer strategy with proof obligations to cut them down into bits that are manageable by off-the-shelf or custom proof backends. The most important aspect of this method is to get rid of all complexity that is not necessary for the programs of the domain.

This line of thought is not entirely new: the division of proof labor has already been experimented in the Viper [19] framework, that features both a custom symbolic execution backend and a backend to the Boogie [97] verifier. Moreover, Viper possesses several frontend exposing verification abilities to subsets of mainstream programming languages: Prusti [98] for

Rust, Gobra [99] for Go, Nagini [100] for Python. The same line of research produced a domain-specific language for verification of concurrent programs, Chalice [101]. However, the philosophy and methodology behind the Viper endeavor, unlike our proposal, goes from language theory and proof automation techniques to examples of programs to verify in the source languages. Hence, the evaluation of the Viper-based verification tools very rarely features real-world programs of more than a thousand lines of code. Moreover, due to the very complex semantics of its internal representation, Viper cannot be connected to an interactive proof assistant for the difficult parts of the proof. Thus, this dissertation sets out from existing work by advocating not to depend completely on a monolithic verification framework.

This dissertation mostly explores the division of proof labor in Chapter 4. However, the domain-specific languages presented in Chapter 3 and Chapter 6 could be developed in future work to achieve this goal. But such achievements cannot be completed by a single researcher: the methodology of Section 1.3.1 encompasses a tall stack of languages, implementations and proofs, and the expertise required to master all steps is beyond the most skilled of researchers or programmers. Hence, the social organization that underlies the contributions of this dissertation is complex, requiring close interactions between the domain experts, the language designers and the proof engineers. At this point in program verification research, we claim that contributions should acknowledge the size of the developments and take the social aspect of the division of labor into account. This means defining clean interfaces to separate layers of the verification stack. These interfaces must be understood by the people on all sides: domain experts, compiler engineers, proof engineers.

Therefore, we propose that proof-oriented domain-specific languages fill the role of these interfaces. In this scheme, the role of the software architect that traditionally defines code interfaces would be matched by the role of language architect. The language architect is in charge of designing the compilation chain, and all intermediate representations and languages that may not exist yet. When software architects use UML diagrams to convey their thoughts, language architects should use semantics and translation formalizations. From that comparison, we can note that UML succeeded as a *lingua franca* precisely because of its ambiguous and flexible nature, despite many attempts to formalize subsets of it [102]–[104]. Hence, rather than advocating for all these formalizations to be constrained in a unifying Coq framework, we recommend for language architects to adapt to the existing technologies of the domain, and use paper proofs and descriptions as a backing *lingua franca*.

The Language Architect Manifest At the end of the introduction for this dissertation, it is time to turn back to Winograd's 1979 paper on the future of programming [105] and quote the author:

> "We cannot turn programmers into native speakers of abstract mathematics, but we can turn our programming formalisms in the direction of natural descriptive forms."

We claim that this task is precisely the job of programming language designers, that ought to occupy a central role in projects involving critical software. Software projects including a full-time language architect position can benefit from productivity increases due to better communication between domain experts and programmers, but also better tooling and reuse of language infrastructure, for optimized compilation or correctness proofs.

The training of such language architects could be very close to existing university-organized formations in theoretical computer science, and should include as a basis: functional programming, compilation theory with a focus on semantic analysis and compiler architecture, programming language theory with semantics of the lambda calculus and its extensions. However, a formal and theoretical training is not enough, as the key skill for the language architect position is to listen and translate faithfully the requirements of the system as stated by the domain experts. This skill may be hindered by the pitfall of being in charge of a wide-ranging structural plan: blindness to local particularities that can lead to oversimplification and destruction. This nefarious behavior is masterfully described by the anthropologist James C. Scott in his piece *Seeing Like a State: How Certain Schemes to Improve the Human Condition have Failed* [106]. Throughout recent history, a number of *high-modernist* State reformators have sought to apply uniformly to vast empires rules of government for administrative and well-intentioned purposes. But sometimes, oversimplified rules simply made no sense in the particular local contexts. For instance, the Russian tsarist administration imposed in the late XIX$^{\text{th}}$ a strict assignation of any piece of land to a unique owner in rural Russia. This completely disrupted the local collective land ownership patterns in place that ensured an equal division of labor and good yield for all the crops. Eventually, the new system led to a decrease in production where it had been applied. More interestingly, some villages kept the old system in place while pretending to apply the new rules when official inspectors came along.

A parallel can be drawn with computer science and the high-modernist views of certain software architects, advocating for a single paradigm/framework for all applications, regarding of the local context. This view is quite prevalent in large organizations, where software urbanists recommend the use of a single programming language for an entire information system (often Java). Instead, we advocate for Scott's solution to the negative effect of over-planification and simplification, which he designates using the Greek *mêtis* ("μῆτις") concept. The *mêtis* is the constant adaptation and local ingenuity that keeps organizations and activities running. It is the individual autonomy of the user that can modify its tools to maximise their fitness for the task at hand. It is also the occasional breaking of the rules when their literate application to the local context does not make any sense.

The high-modernist approach for Computer Science has yielded a number of great pieces of infrastructure, as Section 1.1 reminds. However, we believe that language architects should use a little more *mêtis* and show some adaptation if they want to succeed at raising the level of quality and assurance of real-world software. The first step in this direction is to be more aware of what happens after the program executes: is anybody affected by the decision taken by the program? What are the consequences of an error? Who understands and defines what the program is doing? Who is able to maintain it after the formalizers have left the scene? Answering these questions is hard when the initial training of a language architect merely focuses on technical aspects. Of course, advocating for more individual curiosity is welcome, but including more humanities in the initial training of language architects might be a more structural solution.

In conclusion, we claim that promoting the importance of the language architect position, and the ethos that comes with it, to industrial users of formal methods, will be key to provide more prospects to academic graduates wishing to be employed for their formal abilities.

Contributions of this dissertation

This dissertation presents several variations of the strategy outlined in Section 1.3.1, featuring proof-oriented domain-specific languages in the style of Section 1.3.2. Each chapter presents a separate contribution, corresponding to a peer-reviewed publication or a technical report. The chapters apply the feature methodology of this dissertation to two different domains: cryptogra-

phy and legal expert systems.

Even though they belong to different business universes, these two domains feature critical programs who exercise the limits of the current program verification state of the art. At the same time, the real-world uses of these programs are important enough that it makes economic sense to spend a lot of resources trying to formally verify their correctness and safety.

The first part of the dissertation, consisting of Chapter 2, Chapter 3 and Chapter 4, deals with high-assurance cryptography. The main contribution of Chapter 2 is the demonstration of a complete rundown of the methodology of Section 1.3.1 on the Signal protocol, based on the F* program verification ecosystem state of the art and a new translation to WebAssembly. Chapter 3 solves the specification problem (in the sense of Section 1.2.1) for these verified cryptographic developments, and opens up new connections between specialized provers in the domain. As Chapter 2 emphasizes the limitation of this state of the art, it also and motivates the need for the Steel framework presented in Chapter 4, that brings automated proofs for programs specified using separation logic in F*.

Moving outside the traditional application areas of formal methods, the second part of this dissertation tackles the challenge of correctness of legal expert computer systems that are supposed to follow a legislative specification. Chapter 5 starts from a real-world, critical codebase, the French income tax computation algorithm, and performs a reverse-engineering of the language architecture of the whole system, bringing it up to date and enabling future connections to formal tools. Finally, Chapter 6 examines what can be achieved in this domain when starting from a clean slate, and proposes a high-level domain-specific language designed with lawyers to efficiently and correctly transform law into code. The generated code can either be distributed as libraries in existing information system, or be sent to proof backends for future coherence and safety proofs about the underlying logic of the legal text.

While not all of the main chapter feature a complete run-down of the steps, they all illustrate this adaptability that we claim can help push the program verification frontier.

References

[1] A. Stefik and S. Hanenberg, “The programming language wars: questions and responsibilities for the programming language community”,

in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, 2014, pp. 283–299.

[2] K. Larsen, P. Pettersson, and W. Yi, "Uppaal", *Uppsala University, Sweden and Aalborg University, Denmark.[Online]. Available: http://www. uppaal. com*, 1997.

[3] J. C. King, "Symbolic execution and program testing", *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[4] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques", *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[5] C. Cadar, D. Dunbar, D. R. Engler, *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.", in *OSDI*, vol. 8, 2008, pp. 209–224.

[6] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing", *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[7] J. Fragoso Santos, P. Maksimović, S.-É. Ayoun, and P. Gardner, "Gillian, part I: a multi-language platform for symbolic execution", in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 927–942.

[8] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints", in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, R. M. Graham, M. A. Harrison, and R. Sethi, Eds., ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973. [Online]. Available: https://doi.org/10.1145/512950.512973.

[9] ——, "Abstract interpretation frameworks", *Journal of logic and computation*, vol. 2, no. 4, pp. 511–547, 1992.

[10] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival, "Static analysis and verification of aerospace software by abstract interpretation", in *AIAA Infotech@ Aerospace 2010*, 2010, p. 3385.

[11] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "The Astrée analyzer", in *European Symposium on Programming*, Springer, 2005, pp. 21–30.

[12] C. A. R. Hoare, "An axiomatic basis for computer programming", *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[13] The Coq Development Team, *The Coq Proof Assistant Reference Manual, version 8.7*, Oct. 2017. [Online]. Available: `http://coq.inria.fr`.

[14] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer Science & Business Media, 2002, vol. 2283.

[15] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, *et al.*, "Sel4: formal verification of an OS kernel", in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 207–220.

[16] E. W. Dijkstra, *A discipline of programming*. Prentice-hall.

[17] J.-C. Filliâtre and A. Paskevich, "Why3—where programs meet provers", in *European symposium on programming*, Springer, 2013, pp. 125–128.

[18] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, *et al.*, "Dependent types and multi-monadic effects in F*", in *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 256–270.

[19] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: a verification infrastructure for permission-based reasoning", in *International Conference on Verification, Model Checking, and Abstract Interpretation*, Springer, 2016, pp. 41–62.

[20] B. Jacobs and F. Piessens, "The VeriFast program verifier", Technical Report CW-520, Department of Computer Science, Katholieke . . ., Tech. Rep., 2008.

[21] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world", *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[22] D. Delmas and J. Souyris, "Astrée: from research to industry", in *International Static Analysis Symposium*, Springer, 2007, pp. 437–451.

[23] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, “Frama-C”, in *International conference on software engineering and formal methods*, Springer, 2012, pp. 233–247.

[24] P. Baudin, F. Bobot, D. Bühler, L. Correnson, F. Kirchner, N. Kosmatov, A. Maroneze, V. Perrelle, V. Prevosto, J. Signoles, and N. Williams, “The dogged pursuit of bug-free C programs: the Frama-C software analysis platform”, *Commun. ACM*, vol. 64, no. 8, pp. 56–68, Jul. 2021, ISSN: 0001-0782. DOI: 10.1145/3470569. [Online]. Available: https://doi.org/10.1145/3470569.

[25] D. Harmim, V. Marcin, and O. Pavela, “Scalable static analysis using Facebook Infer”, *Excel@ FIT'19*,

[26] G. Klein, P. Derrin, and K. Elphinstone, “Experience report: seL4: formally verifying a high-performance microkernel”, in *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, 2009, pp. 91–96.

[27] M. Wildmoser and T. Nipkow, “Certifying machine code safety: shallow versus deep embedding”, in *International Conference on Theorem Proving in Higher Order Logics*, Springer, 2004, pp. 305–320.

[28] G. Mével and J.-H. Jourdan, “Formal verification of a concurrent bounded queue in a weak memory model”, in *Proceedings of the 26th ACM SIGPLAN international conference on Functional programming*, 2021.

[29] A. Chlipala, “The Bedrock structured programming system: combining generative metaprogramming and Hoare logic in an extensible program verifier”, in *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, 2013, pp. 391–402.

[30] 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*, 2021, pp. 158–174.

[31] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S.-C. Weng, H. Zhang, and Y. Guo, “Deep specifications and certified abstraction layers”, *ACM SIGPLAN Notices*, vol. 50, no. 1, pp. 595–608, 2015.

[32] 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)*, 2016, pp. 653–669.

[33] B. Delaware, C. Pit-Claudel, J. Gross, and A. Chlipala, "Fiat: deductive synthesis of abstract data types in a proof assistant", *Acm Sigplan Notices*, vol. 50, no. 1, pp. 689–700, 2015.

[34] C. Cohen, M. Dénès, and A. Mörtberg, "Refinements for free!", in *International Conference on Certified Programs and Proofs*, Springer, 2013, pp. 147–162.

[35] T. Coquand and G. Huet, "The calculus of constructions", INRIA, Tech. Rep., 1986.

[36] K. Appel, W. Haken, *et al.*, "Every planar map is four colorable", *Bulletin of the American mathematical Society*, vol. 82, no. 5, pp. 711–712, 1976.

[37] G. Gonthier *et al.*, "Formal proof–the four-color theorem", *Notices of the AMS*, vol. 55, no. 11, pp. 1382–1393, 2008.

[38] G. G.-A. Mahboubi, "A small scale reflection extension for the Coq system", 2007.

[39] A. Bordg, "A replication crisis in mathematics?", *The Mathematical Intelligencer*, pp. 1–5, 2021.

[40] 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*, Springer, 2015, pp. 378–388.

[41] T. L. P. Community, *Liquid tensor experiment*, 2021. [Online]. Available: `https://github.com/leanprover-community/lean-liquid`.

[42] D. Clausen and P. Scholze, *Lectures on analytic geometry*, 2020.

[43] A. Chlipala, B. Delaware, S. Duchovni, J. Gross, C. Pit-Claudel, S. Suriyakarn, P. Wang, and K. Ye, "The end of history? using a proof assistant to replace language design with library design", in *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[44] J. Meng, C. Quigley, and L. C. Paulson, "Automation for interactive proof: first prototype", *Information and computation*, vol. 204, no. 10, pp. 1575–1596, 2006.

[45] G. Martinez, D. Ahman, V. Dumitrescu, N. Giannarakis, C. Hawblitzel, C. Hriţcu, M. Narasimhamurthy, Z. Paraskevopoulou, C. Pit-Claudel, J. Protzenko, *et al.*, "Meta-F*: proof automation with SMT, tactics, and metaprograms", in *European Symposium on Programming*, Springer, Cham, 2019, pp. 30–59.

[46] D. Delahaye, "A tactic language for the system Coq", in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, Springer, 2000, pp. 85–95.

[47] Ł. Czajka and C. Kaliszyk, "Hammer for Coq: automation for dependent type theory", *Journal of automated reasoning*, vol. 61, no. 1, pp. 423–453, 2018.

[48] T. Ringer, K. Palmskog, I. Sergey, M. Gligoric, and Z. Tatlock, "Qed at large: a survey of engineering of formally verified software", *arXiv preprint arXiv:2003.06458*, 2020.

[49] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P.-Y. Strub, "Easycrypt: a tutorial", in *Foundations of security analysis and design vii*, Springer, 2013, pp. 146–166.

[50] 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*, 2013, pp. 15–18.

[51] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, *et al.*, "Spectre attacks: exploiting speculative execution", in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 1–19.

[52] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, *et al.*, "Meltdown: reading kernel memory from user space", in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 973–990.

[53] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation", in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, IEEE, 2004, pp. 75–86.

[54] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly", in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017, pp. 185–200, ISBN: 978-1-4503-4988-8.

[55] X. Leroy, "Formal certification of a compiler back-end or: programming a compiler with a proof assistant", *SIGPLAN Not.*, vol. 41, no. 1, pp. 42–54, Jan. 2006.

[56] M. Patrignani, A. Ahmed, and D. Clarke, "Formal approaches to secure compilation: a survey of fully abstract compilation and related work", *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 1–36, 2019.

[57] E. Mullen, D. Zuniga, Z. Tatlock, and D. Grossman, "Verified peephole optimizations for CompCert", in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 448–461.

[58] D. Monniaux and C. Six, "Simple, light, yet formally verified, global common subexpression elimination and loop-invariant code motion", *arXiv preprint arXiv:2105.01344*, 2021.

[59] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand, "Compcert-a formally verified optimizing compiler", in *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[60] 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*, 2011, pp. 283–294.

[61] H. Tuch, "Formal memory models for verifying c systems code", Ph.D. dissertation, 2008.

[62] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala, "Simple high-level code for cryptographic arithmetic-with proofs, without compromises", in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 1202–1219.

[63] S. Blazy and X. Leroy, "Mechanized semantics for the Clight subset of the C language", *Journal of Automated Reasoning*, vol. 43, no. 3, pp. 263–288, 2009.

[64] B. Delaware, S. Suriyakarn, C. Pit-Claudel, Q. Ye, and A. Chlipala, "Narcissus: correct-by-construction derivation of decoders and encoders from binary formats", *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, pp. 1–29, 2019.

[65] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich, "Using crash Hoare logic for certifying the FSCQ file system", in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 18–37.

[66] M. Sozeau, S. Boulier, Y. Forster, N. Tabareau, and T. Winterhalter, "Coq coq correct! verification of type checking and erasure for Coq, in Coq", *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–28, 2019.

[67] 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 (S&P'21)*, 2021.

[68] A. Fromherz, "A proof-oriented approach to low-level, high-assurance programming", Ph.D. dissertation, Carnegie Mellon University, 2021.

[69] C. M. Hoffmann, "Design and correctness of a compiler for a non-procedural language", *Acta Informatica*, vol. 9, no. 3, pp. 217–241, 1978.

[70] J.-K. Zinzindohoué-Marsaudon, "Secure, fast and verified cryptographic applications: a scalable approach", Ph.D. dissertation, ENS, Inria, 2018.

[71] D. Mercadier and P.-É. Dagand, "Usuba: high-throughput and constant-time ciphers, by construction", in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 157–173.

[72] L. Erkök and J. Matthews, "Pragmatic equivalence and safety checking in Cryptol", in *Proceedings of the 3rd workshop on Programming languages meets program verification*, 2009, pp. 73–82.

[73] 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", in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1807–1823.

[74] A. Van Deursen, P. Klint, and J. Visser, "Domain-specific languages: an annotated bibliography", *ACM Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[75] P. Hudak, "Domain-specific languages", *Handbook of programming languages*, vol. 3, no. 39-60, p. 21, 1997.

[76] D. Spinellis, "Notable design patterns for domain-specific languages", *Journal of systems and software*, vol. 56, no. 1, pp. 91–99, 2001.

[77] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages", *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[78] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[79] C. Severance, "Javscript: designing a language in 10 days", *Computer*, vol. 45, no. 2, pp. 7–8, 2012.

[80] G. Van Rossum, F. L. Drake, *et al.*, *Python reference manual*. iUniverse Indiana, 2000.

[81] T. Biard, A. Le Mauff, M. Bigand, and J.-P. Bourey, "Separation of decision modeling from business process modeling using new "decision model and notation" (DMN) for automating operational decision-making", in *Working Conference on Virtual Enterprises*, Springer, 2015, pp. 489–496.

[82] K. L. Clark, "Negation as failure", in *Logic and data bases*, Springer, 1978, pp. 293–322.

[83] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming", in *European conference on object-oriented programming*, Springer, 1997, pp. 220–242.

[84] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE", *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[85] S. Blackheath, *Functional reactive programming*. Simon and Schuster, 2016.

[86] M. Zoet, "Methods and concepts for business rules management", Ph.D. dissertation, Utrecht University, 2014.

[87] M. Proctor, “Drools: a rule engine for complex event processing”, in *Proceedings of the 4th International Conference on Applications of Graph Transformations with Industrial Relevance*, ser. AGTIVE’11, Budapest, Hungary: Springer-Verlag, 2012, pp. 2–2. DOI: 10.1007/978-3-642-34176-2_2. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-34176-2_2.

[88] S. Erdweg, T. Van Der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, *et al.*, “The state of the art in language workbenches”, in *International Conference on Software Language Engineering*, Springer, 2013, pp. 197–217.

[89] T. van der Storm, “The Rascal language workbench”, *CWI. Software Engineering [SEN]*, vol. 13, p. 14, 2011.

[90] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin-Hochstadt, “The racket manifesto”, in *1st Summit on Advances in Programming Languages (SNAPL 2015)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[91] M. Flatt, “Creating languages in racket”, *Communications of the ACM*, vol. 55, no. 1, pp. 48–56, 2012.

[92] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin-Hochstadt, “A programmable programming language”, *Communications of the ACM*, vol. 61, no. 3, pp. 62–71, 2018.

[93] G. Roșu and T. F. Șerbănuță, “An overview of the K semantic framework”, *The Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[94] T. Zimmermann, “Challenges in the collaborative evolution of a proof language and its ecosystem”, Theses, Université de Paris, Dec. 2019. [Online]. Available: https://hal.inria.fr/tel-02451322.

[95] J. Segal and C. Morris, “Developing scientific software”, *IEEE software*, vol. 25, no. 4, pp. 18–20, 2008.

[96] A. W. Keep and R. K. Dybvig, “A nanopass framework for commercial compiler development”, in *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, 2013, pp. 343–350.

[97] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: a modular reusable verifier for object-oriented programs", in *International Symposium on Formal Methods for Components and Objects*, Springer, 2005, pp. 364–387.

[98] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging Rust types for modular specification and verification", *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. DOI: 10.1145/3360573. [Online]. Available: https://doi.org/10.1145/3360573.

[99] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller, "Gobra: modular specification and verification of go programs", in *International Conference on Computer Aided Verification*, Springer, 2021, pp. 367–379.

[100] M. Eilers and P. Müller, "Nagini: a static verifier for python", in *International Conference on Computer Aided Verification*, Springer, 2018, pp. 596–603.

[101] K. R. M. Leino, P. Müller, and J. Smans, "Verification of concurrent programs with Chalice", in *Foundations of Security Analysis and Design V*, Springer, 2009, pp. 195–222.

[102] W. E. McUmber and B. H. Cheng, "A general framework for formalizing UML with formal languages", in *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, IEEE, 2001, pp. 433–442.

[103] J. E. Smith, M. M. Kokar, and K. Baclawski, "Formal verification of UML diagrams: a first step towards code generation.", in *pUML*, Citeseer, 2001, pp. 224–240.

[104] M. von der Beeck, "Formalization of UML-statecharts", in *UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, M. Gogolla and C. Kobryn, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 406–421, ISBN: 978-3-540-45441-0.

[105] T. Winograd, "Beyond programming languages", *Communications of the ACM*, vol. 22, no. 7, pp. 391–401, 1979.

[106] J. C. Scott, *Seeing Like a State: How Certain Schemes to Improve the Human Condition have Failed*. Yale University Press, 2008.

Part I.

High-Assurance Cryptographic Software

2. LibSignal*: Porting Verified Cryptography to the Web

Contents

Abstract

The introductive chapter of this dissertation exposed the general principles and methodology of our work. Now, we apply those to a first example: the LibSignal* cryptographic protocol implementation. Indeed, as real-world Web Applications embark more and more security-critical components, formally verifying their implementation becomes a more and more sensible option.

To begin our journey in the land of high-assurance cryptographic software, the first two sections of this chapter introduce the F* proof assistant, the Low* program verification domain-specific language, and related work around verified cryptographic developments. Then, we unroll the steps of our signature methodology on LibSignal*, the Web-compatible implementation of the Signal protocol using a novel Low* to WebAssembly toolchain.

Si on ne raisonne pas sur la correction, les bugs arrivent. Le raisonnement sur la correction est l'insecticide de l'informatique.

(Gérard Berry, Leçon inaugurale au collège de France*, 2009)*

I couldn't do the Scheme thing in Javascript, I was under these marching orders to make it look like Java, I had 10 days to prototype it.

(Brendan Eich, 2012)

2.1. The F* Verification Ecosystem

This chapter, as well as Chapter 4, present contributions that build on a particular proof assistant developed since 2016 at Microsoft Research and other academic institutions: F* [1]. Hence, this first section describes the language of this proof assistant, as well as the verification experience it offers.

2.1.1. The proof experience in F*

Since this dissertations focuses on language design, our primer on F* in this subsection will focus on the design elements of F* as a verification tool and the experience they provide, rather than the details of the system and its logical foundations that are already documented in the literature [1]–[5].

Let us begin with a quote from `fstar-lang.org`:

> F* (pronounced F star) is a general-purpose functional programming language with effects aimed at program verification. It puts together the automation of an SMT-backed deductive verification tool with the expressive power of a proof assistant based on dependent types.

The verification philosophy behind F* differs from traditional proof assistants. First, F* is not implemented around a small "kernel" trusted code base that checks all proof correctness, like Coq [6], Isabelle/HOl [7] or Lean [8]. Rather, F* is based on a higher-order dependent type theory that separate the programs from the proof context. For instance, the following definitions defines a tree type and the `nodes` function counting the elements in the tree:

```
type tree (a: Type) : Type =
  | Leaf: tree a
  | Node: v:a -> l:tree a -> r:tree a -> tree a

let rec nodes (#a: Type) (t: tree a) : nat =
  match t with
  | Leaf -> 0
  | Node _ l r -> 1 + nodes l + nodes r
```

The syntax of F* programs is roughly the same as OCaml and ML, with a few additions for the dependently-typed features. The `#` prefix indicates an implicit argument that F* will automatically try to infer by unification at the

call site. The F* programming style encourages the use of type refinements to specify contracts, using the `{...}` syntax. Then, it is possible to define an `insert_left` function that provably adds one node to the tree:

```
let rec insert_left (#a: Type) (t: tree a) (v: a)
    : (t':tree a{nodes t' = nodes t + 1})
  =
  match t with
  | Leaf -> Node v Leaf Leaf
  | Node v' l r -> Node v' (insert_left l v) r
```

When typechecking this piece of code, F* generates multiple guards that stem from its typing rules. Guards can be generated from refinements types, or at the top level from the effect system, e.g. to show that the weakest-precondition of a function body implies the weakest-precondition of its contract in the signature. F* then tries to solve these guards by unification; in case of failures, they are encoded as proof obligation in an SMT query by and discharged to the Z3 [9] SMT solver.

Thus, inside F*, the proofs all happen within the refinements of the different types being manipulated by functions, and the proof context has historically not been directly accessible to the user. So much so that to force F* to check whether a property is true at a given program point, one can write the following:

```
let _ : unit{(* property to prove *)} = () in ...
```

This pattern is how proofs are written in F*. By this, a lemma has the following form:

```
(* The ubiquitous abbreviation for the refined unit *)
type squash (p: Type) : Type0 = x: unit{p}

let foo_lemma ... : squash ((* property to prove *)) =
  (* proof of the lemma *)
```

In idiomatic F* programs, this machinery is hidden by the better looking syntactic sugar:

```
let foo_lemma ... : Lemma ((* property to prove *)) =
  ...
```

The lemma can then be called as s simple unit-returning function inside the program to prove. This call will bring the property of the lemma inside the context, where it will be encoded into SMT and used for the proof. The proof of a lemma can consist in calls to other lemmas, or simply `()`, as this is the value returned by the lemma.

Instead of discussing the specifics of the technical implementation of the F⋆ compiler, we will instead focus of what it entails in terms of proof experience for its users. The preferred development mode for F⋆ uses an Emacs plugin that offers an experience similar to Proof General [10]. In this integrated development environment, the user can type-check her programs incrementally by feeding one top-level definition at a time to the F⋆ type-checker. When the proof associated to the definition's proof obligations goes through with Z3, the user can move forward. But when F⋆'s preliminary type-checking or Z3 fails, an error message is returned to the user.

Before the addition of a tactics system to F⋆ [4], the error messages were the only way for the user to interact with her proof, and the context remained hidden at all times. Moreover, the error messages often do not provide enough information to locate the exact reason why the proof was failing: in the best scenario, they only highlight the location of a failed assertion or precondition. On top of that, F⋆'s encoding of proof obligations into Z3 via the weakest-precondition calculus is such that assertions failures in the code are not reported in order. For instance, if you have two consecutive assertions in a proof and F⋆ reports a failure on the second one, this do not mean that the first assertion succeeded. Because of these limitations, the F⋆ proof experience is dominated by the "sliding admit" verification style.

The "sliding admit" verification style is a work-around that consists in debugging proofs via the use of `admit` calls that will move through the proof over time. `admit : unit -> unit` is a catch-all F⋆ predicate that, once encoded into Z3, will mark as proven all the proof obligations coming from a later point in the F⋆ program. Consider the following program whose type-checking fails:

```
let difficult_function (i: input)
    : o:output{conjunct1 i o /\ conjunct2 io}
  =
  let t1 = function1 i in
  let t2 = function2 i t1 in
  ...
```

```
  let o = tn in
  o
```

Instead of searching through the whole function to determine which pre-condition or part of the post-condition is failing, you can put an `admit` after the first let-binding. This `admit` call will admit the weakest-precondition corresponding to everything that happens after it inside the expression:

```
let difficult_function i =
  let t1 = function1 i in
  admit()
```

If at that point you can already express parts of the post-condition, you can also express them using **`assert`**:

```
let difficult_function i =
  let t1 = function1 i in
  assert(conjunct1 i t1);
  admit()
```

You can move your way forward by sliding the `admit` down through the function, hence the name of this verification style. Another big part of F*'s proof experience is to fiddle with the SMT encoding parameters. The three big parameters that have considerable influence on Z3's ability to prove F*'s proof obligations are the `z3rlimit`, the `fuel` and the `ifuel`. `z3rlimit` is akin to a Z3 timeout, after which the prover stops searching for a proof and returns control back to F* that yields an error message. Actually, `z3rlimit` is a machine-independent resource limit that differs from the wall clock time to account for the different computing power and offer an uniform proof experience across machines; but we'll refer to it as a timeout in the rest of the dissertation. A timeout is necessary for the proof experience because Z3 can spend several minutes looking for a proof of an incorrect statement; in that case, we prefer stopping the search early and return a prompt to the user quickly. However, difficult but correct proofs also take more time for Z3, and this case requires the user to specify an increased timeout; this also indicates to code reviewers the difficult sections of the proof. Another hurdle in F*'s proof experience is Z3's non-deterministic behavior, that hinders replication of proofs with a high `z3rlimit` over time. The only solution to that problem is to

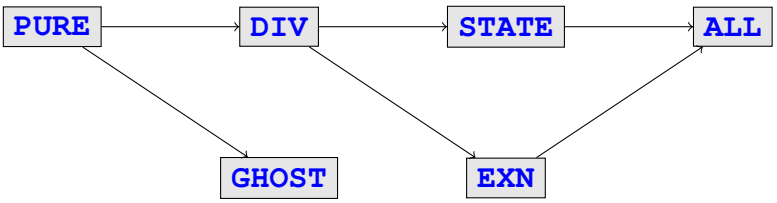

Figure 2.1: Lattice of builtin effects in F⋆, from [1]

give more details about the proof in the source code, by placing intermediate lemma calls or asserting critical properties.

The two remaining parameters, `fuel` and `ifuel`, control respectively the unrolling of recursive predicates and the inversion of inductive types in the Z3 encoding. Indeed, some proofs relying on unrolling one or more times the recursive definition of a function, or by case-matching on one or more levels of the structure of an instance of an inductive datatype. For instance, by setting `fuel` to 2, all recursive functions in the proof context will be unrolled twice in the Z3 encoding. By setting `ifuel` to 1, the Z3 encoding will feature proof branches that depend on the cases of all inductive values in the context. The higher `fuel` and `ifuel`, the broader the proof search, but also the bigger the query and the longer its resolution by Z3. Hence, a tradeoff has to be determined by the user, who can also user finer-grained specification mechanism for controlling unrolling of specific terms and types.

Overall, the F⋆ verification experience relies heavily on an intimate knowledge of Z3's behavior, and a sense about how things are encoded in the SMT query. On the plus side, the automation provided by the prover is a relief for bookkeeping-heavy proofs, and the unity of the syntax for proofs and programs make for a good readability of the code. On the negative side, the imprecision of F⋆'s error messages is puzzling and the "sliding" admit verification style is a poor substitute for a closer and more transparent inspection of the proof context at the problematic program point, which F⋆ cannot provide.

2.1.2. Using Effects for Specifications and Language Design

The previous subsection deliberately omitted a key feature of the F⋆ language: its effect system. This effect system provides a lightweight and direct way of writing and specifying programs that use advanced verification features like state manipulation.

All computations in F⋆ are labeled with a monadic effect that maps the F⋆

surface syntax to the combinators of the effect: `bind`, `return`, `bind`, `subcomp`, `if_then_else`, etc. Figure 2.1 present the builtin effects defined in the F* type-checker, and the lattice that their implicit conversions form. These builtin effects support a weakest-preconditions predicate transformer that can be elaborated to a pre- and post-condition sugar, **Pure**:

```
let pure_post (a: Type) (pre: Type) =
  _: a{pre} -> GTot Type0

effect Pure
  (a: Type)
  (pre: Type0)
  (post: pure_post' a pre)
  =
  PURE a
    (fun (p: pure_post a) -> pre /\ (forall (pure_result: a).
      post pure_result ==> p pure_result))
```

In the above snippet, **Type0** is a refinement of **Type** that belongs to universe zero, and **GTot** is an effect sugar for **GHOST**. Note that effect declarations (`effect`) can be indexed by parameters. These parameters will in general carry the specifications of the computation labeled by the effect. Here, **Pure** is indexed by a pre- and post-condition. These pre- and post-conditions are used in the effect definition to craft the corresponding weakest precondition transformer of **PURE**.

By enabling the user to define effect sugars on top of the builtin effects, and recently, in a completely abstract way [11], F* offers a powerful mechanism for defining proof-oriented domain-specific languages in F*. This approach has been best illustrated by Protzenko *et al.*'s Low* [12] language for low-level programming in F*. Low* effectively defines several effect sugars on top of **State**, like **Stack**, intended for programs that only allocate memory on the stack (and not on the heap).

Functions labeled with the **Stack** effect are specified using pre- and post-conditions located in the indices of the effect. These pre- and post-conditions can refer to the state of the memory before and after the computation. The memory model is structured, inspired by Clight [13], and memory reasoning is performed using the SMT automation of F* via a library of memory locations. In this model, the SMT query for a top-level computation labeled with the

Stack will actually contain a mix of proof obligations concerning memory separation and other proof obligations concerning, for instance, the functional correctness of the computation with regards to a high-level specification written in the **PURE** effect.

Low⋆ is effectively a domain-specific language for low-level programming and not merely a model of such a language, because it is meant to be extracted to C via the KreMLin compiler. This compilation toolchain starts within the F⋆ type-checker, where all the sub-computations of the program labeled by the **GHOST** effect or the **Ghost**.erased type are erased. The resulting program is then dumped and picked by KreMLin which scans it to see whether it fits within the Low⋆ domain-specific language: indeed, not all F⋆ constructions can be extracted to C (especially the higher-order constructions). KreMLin then performs a series of translation of the abstract syntax tree of the program to reach a C abstract syntax tree inspired from Clight, and finally emit a valid C program. This C program can then be compiled with CompCert [14] or established C compilers like GCC or LLVM.

As we will see in the next section, Low⋆ has been used for several high-profile real-world verified cryptography applications. The key feature that enabled technology transfers for KreMLin-compiled artifacts has not been compiler certification, but human readability of the generated code. Indeed, while compiler certification is definitely a way to raise the level of assurance of the trust chain as discussed in Section 1.2.1, the engineers at Firefox [15] or Wireguard [16] who decided to swap their existing software with a KreMLin-compiled artifact were most concerned about their ability to manually review and understand the code they were integrating into their codebase.

For this purpose, KreMLin's authors have written several compiler passes whose sole purpose is to make the generated C code more readable, while keeping the same semantics. Likely, F⋆ offers keywords like inline_for_extraction which lets user control precisely how the code will be extracted, while allowing functions to be sliced up arbitrarily to ease verification and specification tasks. All of these features represent a non-negligible implementation overhead for the maintainers of the toolchain, that likely barred them from moving to adding fancy new features that would have enabled the tackling of a more advanced concepts of C like handling pointers to inside data structures.

Indeed, Low⋆ is currently limited to sequential programs, and its user experience is skewed towards programs that look like cryptographic primitives, its main application so far. Because the Low⋆ subset seeks efficiency on real-

world applications rather than the complexity of its language feature, it fully represents the proof-oriented domain-specific language philosophy that we advocate for in this dissertation.

2.2. Cryptographic Program Verification

After having introduced the context on the F* verification ecosystem, we now present its main real-world applications to verified cryptography. First, we showcase a state of the art of current work on cryptographic formal verification. Then, we focus more specifically on the project Everest collaboration between Microsoft Research and multiple research institutions, including Inria. Its goal was to verify all the components necessary to implement the Transport Layer Security (TLS) protocol that secures the Internet. It notably led to the creation of the EverCrypt cryptographic provider, which is the largest verified cryptographic provider so far.

2.2.1. Related Work on Cryptographic Program Verification

The most recent and exhaustive survey has been completed by Barbosa *et al.* [17] in 2021. This subsection is largely based on this survey, while focusing on functional correctness of verified programming developments of two categories of cryptographic programs: primitives and protocols.

The goal of a cryptographic primitive is to achieve an cryptographic operation with clearly defined inputs and outputs, that is often used as a basic block for larger applications: encrypting a message, decrypting a message, hashing a message, signing a message, etc. Each of these operations correspond to a function signature that identifies the parameters and result of the operation. For instance, `encrypt: plaintext -> key -> ciphertext`. Hence, a program implementing a cryptographic primitive is a function corresponding to the signature of the operation it implements. Moreover, this program must enjoy additional security guarantees: for `encrypt`, it should be virtually impossible to get back the plain-text for the cipher-text in absence of the key used to encrypt it. For `hash`, it should be virtually impossible to get the input of the function given its output (one-way hashing).

On the other hand, cryptographic protocols use cryptographic primitives as their building blocks, to implement each step of a codified communication between two ore more parties. The goal of each step of this communication

is to contribute to global security goals for the protocols: authentication, confidentiality, forward security, etc. To prove these global security properties, one assumes the properties that the cryptographic primitives should enjoy. Hence, there is a clear separation between primitives and protocols concerning cryptographic proofs, that is reflected in mechanized developments.

While some provers focus on proving the security properties for primitives and protocols, using a symbolic model [18]–[20] or a computational model [21]–[24], we will focus in this chapter on the verification of cryptographic implementations as opposed to specifications. This means that we consider a high-level specification of the primitive or protocol as our source of truth, and we assume that this specification successfully provides the security properties mentioned above.

Given this assumption, several works have sought to provide ready-to-use, high-assurance cryptographic implementations that formally enjoy as many as four main properties: functional correctness with respect to a high-level specification, memory safety, state-of-the-art performance and some form of side-channel resistance. These works are all based on a verification framework, and/or domain-specific language with its compiler.

First, the Galois, Inc. has developed over the years a domain-specific language – Cryptol [25] – coupled with a multi-proof backend verification framework – SAW [26]. Using these tools, they released a comprehensive cryptographic library that targets Java ad C [27]. However, SAW is limited to symbolic execution, which is adapted for proving memory safety but makes it hard to verify the equivalence between a high-level specification and the target optimized implementation.

Second, Fiat Crypto [28] is an application of the Fiat [29] Coq proof framework to the synthesis of optimized C implementations of elliptic curve cryptographic primitives from a high-level description in mathematical terms. All the steps of the program synthesis are certified, making the level of assurance of the resulting C library very high. However, this synthesis technique requires manually writing optimization passes tailored for the implementation of each category of primitives to reach state-of-the-art performance for the generated code. Writing these optimizations (and certify them) is very costly, hence Fiat Crypto has so far been only limited to cryptographic primitives based on elliptic curves. A similar approach based on code synthesis trough compilation of high-level specifications is featured by Usuba [30], an optimizing compiler for a domain-specific language describing constant-time implementations of block ciphers using bit-slicing.

Third, EasyCrypt [31] is a specialized theorem prover for cryptography. Its focus is security proofs of primitives using a computational model, but it has been extended a first time [32] with an embedded domain-specific language modeling imperative C-like programs, and a second time with a model of the Jasmin [33] domain-specific language that generates assembly implementations through certified compilation steps. Unlike Fiat Crypto that extracts directly C programs from the reification of Coq term into an abstract syntax tree of a small imperative language, EasyCrypt is mostly used as a proof backend for external compilers that output a model of the code they process.

This non-exhaustive review demonstrates the importance of domain-specific languages and specialized provers in cryptographic program verification. Indeed, proving the equivalence of optimized implementations with respect to high-level mathematical specifications is increasingly harder, as optimized implementations mix C and Assembly, and rely on low-level tricks to gain a few cycles per byte.

All the works cited above enabled the production of high-assurance, ready-to-use cryptographic libraries that have made their way into real-world production environment, like Google's BoringSSL library for Fiat Crypto. In the next subsection, we will discuss the contributions of the largest of these verified libraries in terms of number of algorithms covered: EverCrypt [34].

2.2.2. The EverCrypt Cryptographic Provider

EverCrypt [34] is the combination of several verified cryptography projects built around the F* theorem prover. Presenting its characteristics will be necessary to introduce our later work on LibSignal* (Section 2.3) and Steel (Chapter 4).

The starting point of verified cryptography in F* is the Low* embedded domain-specific language presented in Section 2.1.2. Indeed, Low* perfectly handles the kind of C programs that are found in cryptographic primitives' implementations: mostly stack-based memory allocation, buffers as the most complex data structure, lots of arithmetic and bit-wise manipulation of machine integers. Then, it is possible to write optimized implementations of cryptographic primitives in Low*. These implementations are specified and verified functionally equivalent to high-level specifications (also written in F*) on the one hand, and extracted to human-readable C on the other hand. The collection of all these implementations is called HACL* [35], and features

110,000 lines of hand-written verified Low⋆.

Figure 2.2 presents an example of HACL⋆ code. The same `line` function is implemented twice. A first time as a concise, functional-style specification that is part of the trusted code base and meant to be reviewed for correctness with respect to the official pseudo-code description of the Chacha20 algorithm [36]. And a second time as a Low⋆ implementation using the **`Stack`** effect, indexed by a post-condition ensuring that the contents of the `st` buffer after the computation is equal to the action of the **`Spec`**`.line` function on the content of `st` before the computation. This post-condition is encoded as an SMT query at F⋆ type-checking time, and its verification guarantees the functional correctness of `line`.

Once verified, the function is extracted to human-readable C; Figure 2.3 presents the result of this extraction for `line`. Note that in Low⋆, `line` is marked as `inline_for_extraction`. This means that all occurrences of `line` are inlined in the resulting C code; Figure 2.3 thus shows the `quarter_round` function that originally contains four calls to `line`. As an example of the measures KreMLin implements to preserve human readability of the extracted code, the inlining preserves the original variable names while suffixing them with low numbers (0,1,2, etc.) to preserve the semantics of the original code after inlining.

While HACL⋆'s generated C implementation provide the bulk of the EverCrypt provider, a modern cryptographic library needs some assembly-written parts to be performance-competitive on major architectures. Similarly to the Jasmin mentioned in Section 2.2.1, Vale [37] allows users to write assembly-like implementations of cryptographic primitives in a domain-specific language. These implementations are then checked for functional equivalence with higher-level specifications inside the Dafny [38] prover. Later, the Vale compiler was extended with an F⋆ backend [39], which has been used to verify the correct interoperability between HACL⋆'s Low⋆ code and a certified model of the Vale code in F⋆. With this new connection, EverCrypt was able to integrate the 14,000 lines of hand-written Vale and provide high-speed implementations for major primitives on popular platforms.

Orthogonally to integrating verified C and assembly through mixing and interoperating two domain-specific languages, Low⋆ and Vale, EverCrypt offers on top of the implementations a wide choice of APIs. Thanks to F⋆'s meta-programming abilities [4], agile (choosing between multiple algorithms for the same functionality) and multiplexed (choosing between multiple implementations of the same algorithm) interfaces have been automatically generated

Chacha20's **Spec** F* module

```
let line (a:idx) (b:idx) (d:idx) (s:rotval U32) (m:state)
     : Tot state
  =
  let m = m.[a] <- (m.[a] +. m.[b]) in
  let m = m.[d] <- ((m.[d] ^. m.[a]) <<<. s) in m
```

Chacha20's **Impl** F* module

```
inline_for_extraction
val line:
    st:state
  -> a:index
  -> b:index
  -> d:index
  -> r:rotval U32 ->
  Stack unit
  (requires fun h -> live h st /\ v a <> v d)
  (ensures  fun h0 _ h1 -> modifies (loc st) h0 h1 /\
    as_seq h1 st ==
      Spec.line (v a) (v b) (v d) r (as_seq h0 st))

let line st a b d r =
  let sta = st.(a) in
  let stb = st.(b) in
  let std = st.(d) in
  let sta = sta +. stb in
  let std = std ^. sta in
  let std = rotate_left std r in
  st.(a) <- sta;
  st.(d) <- std
```

Figure 2.2: F* specification and Low* implementation of the `line` function of the Chacha20 cryptographic primitive, as found in HACL*.

Hacl_Chacha20.c

```
static inline void quarter_round(uint32_t *st, uint32_t a,
  uint32_t b, uint32_t c, uint32_t d)
{
  uint32_t sta0 = st[a];
  uint32_t stb0 = st[b];
  uint32_t std0 = st[d];
  uint32_t sta10 = sta0 + stb0;
  uint32_t std10 = std0 ^ sta10;
  uint32_t std20 = std10 << (uint32_t)16U |
    std10 >> (uint32_t)16U;
  uint32_t sta2; uint32_t stb1;
  uint32_t std3; uint32_t sta11;
  uint32_t std11; uint32_t std21;
  uint32_t sta3; uint32_t stb2;
  uint32_t std4; uint32_t sta12;
  uint32_t std12; uint32_t std22;
  uint32_t sta; uint32_t stb;
  uint32_t std; uint32_t sta1;
  uint32_t std1; uint32_t std2;
  st[a] = sta10; st[d] = std20;
  sta2 = st[c];  stb1 = st[d]; std3 = st[b];
  sta11 = sta2 + stb1;
  std11 = std3 ^ sta11;
  std21 = std11 << (uint32_t)12U | std11 >> (uint32_t)20U;
  st[c] = sta11; st[b] = std21;
  sta3 = st[a]; stb2 = st[b]; std4 = st[d];
  sta12 = sta3 + stb2;
  std12 = std4 ^ sta12;
  std22 = std12 << (uint32_t)8U | std12 >> (uint32_t)24U;
  st[a] = sta12; st[d] = std22;
  sta = st[c]; stb = st[d]; std = st[b];
  sta1 = sta + stb;
  std1 = std ^ sta1;
  std2 = std1 << (uint32_t)7U | std1 >> (uint32_t)25U;
  st[c] = sta1; st[b] = std2;
}
```

Figure 2.3: C extraction of the Chacha20 Low* implementation of Figure 2.2. `quarter_round` features four inlined calls to `line`.

to support all the combinations of algorithms and specific implementations for each functionality of the cryptographic libraries. Agility and multiplexing are especially important for applications that run on heterogeneous hardware and platforms; these applications want to choose the more efficient cryptography at runtime depending on system configuration and supported hardware instructions.

Now that we have presented the context of verified cryptography within the F* ecosystem, we will showcase the agility of this domain-specific language-based architecture with a case study about deploying a popular cryptographic protocol on the Web.

2.3. High-Assurance Cryptography on the Web: A Case Study

This section is based upon the following publication:

J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan, "Formally verified cryptographic web applications in WebAssembly", in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1256–1274. DOI: 10.1109/SP.2019.00064

My personal contribution to this publication has been a formalization of the Low to WebAssembly translation, later revised and polished by Jonathan Protzenko for the publication, as well as all the Javascript development work in LibSignal*. I also designed and implemented the secret independence translation validator for the generated WebAssembly code.*

Modern Web applications rely on a variety of cryptographic constructions and protocols to protect sensitive user data from a wide range of attacks. For the most part, applications can rely on standard builtin mechanisms. To protect against network attacks, client-server connections are typically encrypted using the Transport Layer Security (TLS) protocol, available in all Web servers, browsers, and application frameworks like iOS, Android, and Electron. To protect stored data, user devices and server databases are often encrypted by default.

However, many Web applications have specific security requirements that

require custom cryptographic mechanisms. For example, popular password managers like LastPass [41] aim to synchronize a user's passwords across multiple devices and back them up on a server, without revealing these passwords to the server. So, the password database is always stored encrypted, with a key derived from a master passphrase known only to the user. If this design is correctly implemented, even a disgruntled employee or a coercive nation-state with full access to the LastPass server cannot obtain the stored passwords. A similar example is that of a cryptocurrency wallet, which needs to encrypt the wallet contents, as well as sign and verify currency transactions.

Secure messaging applications like WhatsApp and Skype use even more sophisticated mechanisms to provide strong guarantees against subtle attacks. For example, they provide *end-to-end security* between clients, so that a compromised or coerced server cannot read or tamper with messages. They guarantee *forward secrecy*, so that even if one of the devices used in a conversation is compromised, messages sent before the compromise are still secret. They even provide *post-compromise security*, so that a compromised device can recover and continue to participate in a conversation. To obtain these guarantees, many messaging applications today rely on some variant of Signal, a cryptographic protocol designed by Moxie Marlinspike and Trevor Perrin [42], [43].

To provide a seamless experience to users, most Web applications are implemented for multiple platforms; e.g. native apps for iOS and Android, Electron apps that work on most desktop operating systems, installable browser extensions for specific browsers, or a website version accessible from any Web browser. Except for the native apps, these are all written in JavaScript. For example, most Signal-based messaging apps use the official LibSignal library, which has C, Java, and JavaScript versions. The desktop versions of WhatsApp and Skype use the JavaScript version, as depicted in Figure 2.4.

In this section, we are concerned with the question of how we can gain higher assurance in the implementations of such cryptographic Web applications. The key novelty of our work is that we target WebAssembly rather than general JavaScript. We show how to build verified implementations of cryptographic primitives so that they can be deployed both within platform libraries (via a C implementation) and within pure JavaScript apps (via a WebAssembly implementation). We show how to build a verified implementation of the Signal protocol (as a WebAssembly module) and use it to develop a drop-in replacement for LibSignal-JavaScript.

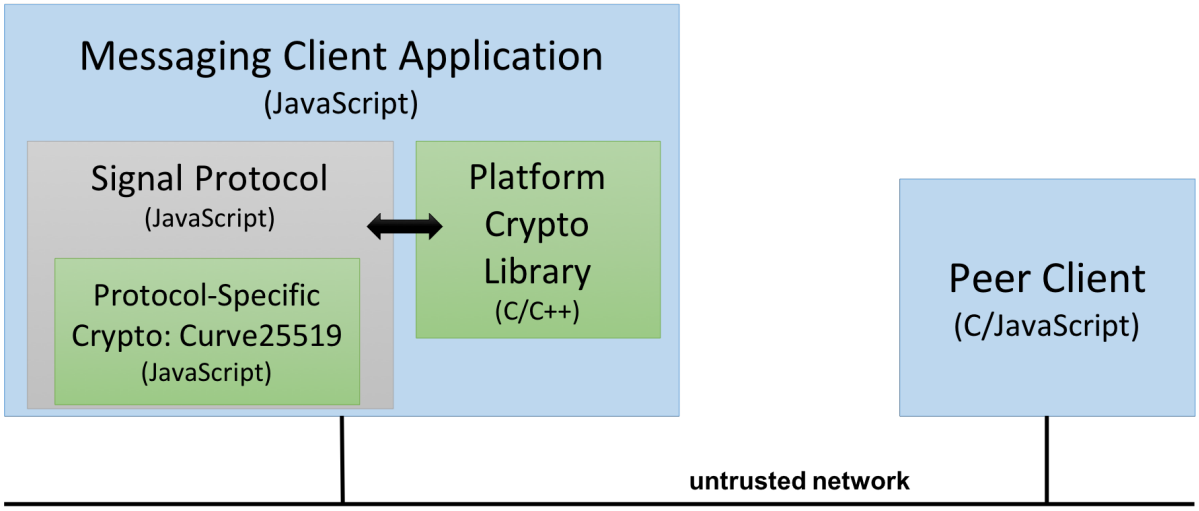

Figure 2.4: Secure Messaging Web App Architecture: The application includes the official LibSignal library, which in turn uses the platform's crypto library, but also provides custom implementations for crypto primitives that are not available on all platforms. The security-critical components that we aim to verify are the core signal protocol and all the crypto code it relies on.

Introduced in 2017, WebAssembly [44] is a portable execution environment supported by all major browsers and Web application frameworks. It is designed to be an alternative to but interoperable with JavaScript. WebAssembly defines a compact, portable instruction set for a stack-based machine. The language is made up of standard arithmetic, control-flow, and memory operators. The language only has four value types: floating-point and signed numbers, both 32-bit and 64-bit. Importantly, WebAssembly is *typed*, meaning that a well-typed WebAssembly program can be safely executed without fear of compromising the host machine (WebAssembly relies on the OS page protection mechanism to trap out-of-memory accesses). This allows applications to run independently and generally deterministically. WebAssembly applications also enjoy superior performance, since WebAssembly instructions can typically be mapped directly to platform-specific assembly. Interaction with the rest of the environment, e.g. the browser or a JavaScript application, is done via an import mechanism, wherein each WebAssembly module declares a set of imports whose symbols are resolved when the compiled WebAssembly code is dynamically loaded into the browser. As such, WebAssembly is completely platform-agnostic (it is portable) but also Web-agnostic (there is no mention of the Document Object Model or the Web in the specification).

Our approach is to compile WebAssembly code from formally verified source code written in Low* [12], a subset of the F* programming language [1]. As far as we know, this is the first verification toolchain for WebAssembly that supports correctness, memory safety, and side-channel resistance.

Programmers, when authoring Web applications, have very few options when it comes to efficient, trustworthy cryptographic libraries. When running within a browser-like environment, the W3C WebCrypto API [45] provides a limited choice of algorithms, while imposing the restriction that all code calling into WebCrypto must be asynchronous via the mandatory use of promises. This entails that WebAssembly code cannot call WebCrypto, since it does not support async functions. When running within a framework like Electron, programmers can use the `crypto` package, which calls OpenSSL under the hood and hence supports more algorithms, but requires trust in a large unverified library.

In both these scenarios, the main restriction is perhaps the lack of novel algorithms: for a new algorithm to be available, the W3C must adopt a new standard, and all browsers must implement it; or, OpenSSL must implement it, issue a release, and binaries must percolate to all target environments. For example, modern cryptographic standards such as Curve25519, Chacha20, Poly1305, SHA-3 or Argon2i are not available in WebCrypto or older versions of OpenSSL.

When an algorithm is not available on all platforms, Web developers rely on hand-written, unverified JavaScript implementations or compile such implementations from unverified C code via Emscripten. In addition to correctness questions, this JavaScript code is often vulnerable to new timing attacks. We aim to address this issue, by providing application authors with a verified crypto library that can be compiled to both C and WebAssembly: therefore, our library is readily available in both native and Web environments.

Complex cryptographic protocols are hard to implement correctly, and correctness flaws (e.g. [46]) or memory-safety bugs (e.g. HeartBleed) in their code can result in devastating vulnerabilities. A number of previous works have shown how to verify cryptographic protocol implementations to prove the absence of some of these kinds of bugs. In particular, implementations of TLS in F# [47], C [48], and JavaScript [49] have been verified for correctness, memory safety, and cryptographic security. An implementation of a non-standard variant of Signal written in a subset of JavaScript was also verified for cryptographic security [50], but not for correctness.

We propose to build and verify a fully interoperable implementation of

Signal in Low* for memory safety and functional correctness with respect to a high-level specification of the protocol in F*. We derive a formal model from this specification and verify its symbolic security using the protocol analyzer ProVerif [18]. We then compile our Low* code to WebAssembly and embed it within a modified version of LibSignal-JavaScript to obtain a drop-in replacement for LibSignal for use in JavaScript Web applications.

Our contributions are twofold. First, we present the first verification and compilation toolchain targeting WebAssembly, along with its formalization and a compact auditable implementation. Second, we present WHACL*, the first high-assurance cryptographic library in WebAssembly, based on the existing HACL* library [35] (presented in Section 2.2.2), and LibSignal*, a novel verified implementation of the Signal protocol, that by virtue of our toolchain, enjoys compilation to both C and WebAssembly, making it a prime choice for application developers.

2.3.1. Compiling Low* to WebAssembly

Before presenting our compilation from Low* to WebAssembly, we begin by motivating the creation of this new toolchain.

A New Toolchain Targeting WebAssembly WebAssembly is the culmination of a series of experiments (NaCl, PNaCl, asm.js) whose goal was to enable Web developers to write high-performance assembly-like code that can be run within a browser. Now with WebAssembly, programmers can target a portable, compact, efficient binary format that is supported by Chrome, Firefox, Safari and Edge. For instance, Emscripten [51], a modified version of LLVM, can generate WebAssembly. The code is then loaded by a browser, JIT'd to machine code, and executed. This means that code written in, say, C, C++ or Rust, can now be run efficiently on the web.

The syntax of WebAssembly (from [44]) is shown in Figure 2.5. We use i for WebAssembly instructions and t for WebAssembly types. WebAssembly is a typed, expression language that reduces using an operand stack; each instruction has a function type that indicates the types of operands it consumes from the stack, and the type of operand it pushes onto the stack. For instance, if ℓ has type i32, then get_local ℓ has type $[] \to$ i32, i.e. it consumes nothing and pushes a 32-bit value on the stack. Similarly, t.store has type i32; $t \to []$, i.e. it consumes a 32-bit address, a value of type t, and pushes

f ::=		function
	func t_f local $\overrightarrow{\ell : t}$ $\overrightarrow{i}$	
i ::=		instruction
	if t_f $\overrightarrow{i}$ else $\overrightarrow{i}$	conditional
	call f	function call
	get_local ℓ	read local variable
	set_local ℓ	set local variable
	t.load	load from memory
	t.store	write to memory
	t.const k	push constant
	drop	drop operand
	loop $\overrightarrow{i}$	loop
	br_if	break-if-true
	t.binop o	binary arithmetic
t ::=		value type
	i32	32-bits integer
	i64	64-bits integer
t_f ::=		function type
	$\overrightarrow{t} \to t$	
o ::=		operator
	add, sub, div, ...	

Figure 2.5: WebAssembly Syntax (selected constructs)

nothing onto the stack.

We omit from this presentation: n-ary return types for functions (currently not supported by any WebAssembly implementation); treatment of packed 8-bit and 16-bit integer arrays (supported by our implementation, elided for clarity).

This human-readable syntax maps onto a compact binary format. The programmer is not expected to directly write programs in WebAssembly; rather, WebAssembly was designed as a compilation target. Indeed, WebAssembly delivers performance: offline compilers generates better code than a JIT; compiling WebAssembly code introduces no runtime-overhead (no GC); the presence of 64-bit values and packed arrays enables more efficient arithmetic and memory locality.

Previous works attempted to protect against the very loose, dynamic nature of JavaScript (extending prototypes, overloading getters, rebinding `this`, etc.) by either defining a "safe" subset [52], [53], or using a hardening compilation scheme [54], [55]. By contrast, none of the JavaScript semantics leak into WebAssembly, meaning that reasoning about a WebAssembly program within a larger context boils down to reasoning about the boundary between WebAssembly and JavaScript.

From a security standpoint, this is a substantial leap forward, but some issues still require attention. First, the boundary between WebAssembly and JavaScript needs to be carefully audited: the JavaScript code is responsible for setting up the WebAssembly memory and loading the WebAssembly modules. This code must use defensive techniques, e.g. make sure that the WebAssembly memory is suitably hidden behind a closure. Second, the whole module loading process needs to be reviewed, wherein one typically assumes that the network content distribution is trusted, and that the WebAssembly API cannot be tampered with (e.g. `Module.instantiate`).

The flagship toolchain for compiling to WebAssembly is Emscripten [51], a compiler from C/C++ to JavaScript that combines LLVM and Binaryen, a WebAssembly-specific optimizer and code emitter. Using Emscripten, several large projects, such as the Unity and Unreal game engines, or the Qt Framework have been ported to WebAssembly. Recently, LLVM gained the ability to directly emit WebAssembly code without going through Binaryen; this has been used successfully by Rust and Mono.

Cryptographic libraries have been successfully ported to WebAssembly using Emscripten. The most popular one is libsodium, which owing to its relatively small size and simplicity (no plugins, no extensibility like OpenSSL) has

successfully been compiled to both JavaScript and WebAssembly.

The core issue with the current toolchain is both the complexity of the tooling involved and its lack of auditability. Trusting libsodium to be a correct cryptographic library for the web involves trusting, in order: that the C code is correct, something notoriously hard to achieve; that LLVM introduces no bugs; that the runtime system of Emscripten does not interfere with the rest of the code; that the Binaryen tool produces correct WebAssembly code; that none of these tools introduce side-channels; that the code is sufficiently protected against attackers.

In short, the trusted computing base (TCB) is very large. The source language, C, is difficult to reason about. Numerous tools intervene, each of which may be flawed in a different way. The final WebAssembly (and JavaScript) code, being subjected to so many transformations and optimizations, can neither be audited or related to the original source code.

Overview of the Toolchain Seeing that WebAssembly represents a compelling compilation target for security-critical code on the web; seeing that F* is a prime language for writing security-critical code; we repurpose the Low*-to-C toolchain and present a verified compilation path from Low* to WebAssembly.

Protzenko *et.al.* [12] model the Low*-to-C compilation in three phases (Figure 2.6). The starting point is Explicitly Monadic F* [2]. First, the erasure of all computationally-irrelevant code yields a first-order program with relatively few constructs, which they model as λow*, a simply-typed lambda calculus with mutable arrays. Second, λow* programs are translated to C*, a statement language with stack frames built into its reduction semantics. Third, C* programs go to CLight, CompCert's internal frontend language for C [14].

Semantics preservation across these three steps is shown using a series of simulations. More importantly, this Low*-to-C pipeline ensures a degree of *side-channel* resistance, via type abstraction. This is achieved through *traces* of execution, which track memory access and branches. The side-channel resistance theorem states that if two programs verify against an abstract secret type; if these two programs only differ in their secret values; if the only functions that operate on secrets have secret-independent traces; then once compiled to Clight, the two programs reduce by producing the same result and emitting the same traces. In other words, if the same program operates on different secrets, the traces of execution are indistinguishable.

We repurpose both the formalization and the implementation, and replace

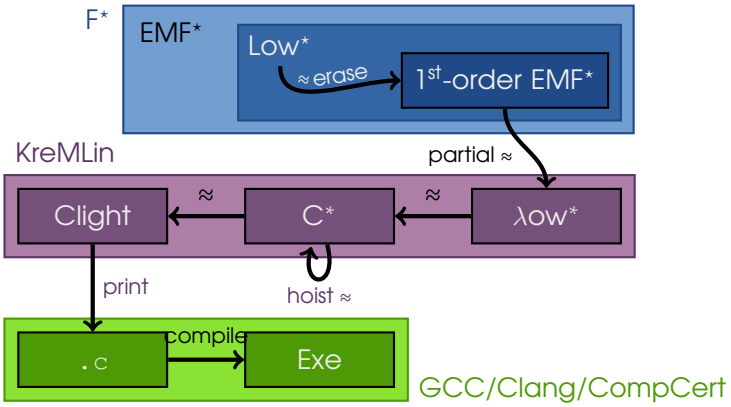

Figure 2.6: The original Low*-to-C translation

the λow* → C* → Clight toolchain with a new λow* → C♭ → WebAssembly translation. We provide a paper formalization in the present section and our implementation is now up and running as a new backend of the KreMLin compiler. (Following [12], we omit the handling of heap allocations, which are not used in our target applications.)

Using off-the-shelf tools, one can already compile Low* to C via KreMLin, then to WebAssembly via Emscripten. As we mentioned earlier, this TCB is substantial, but in addition to the trust issue, there are technical reasons that justify a new pipeline to WebAssembly.

First, C is ill-suited as an intermediary language. C is a statement language, where every local variable is potentially mutable and whose address can be taken; LLVM immediately tries to recover information that was naturally present in Low* but lost in translation to C, such as immutable local variables ("registers"), or an expression-based representation via a control-flow graph. Second, going through C via C* puts a burden on both the formalization and the implementation. On paper, this mandates the use of a nested stack of continuations for the operational semantics of C*. In KreMLin, this requires not only dedicated transformations to go to a statement language, but also forces KreMLin to be aware of C99 scopes and numerous other C details, such as undefined behaviors. In contrast, C♭, the intermediary language we use on the way to WebAssembly, is expression-based, has no C-specific concepts, and targets WebAssembly whose semantics have no undefined-behavior. As such, C♭ could be a natural compilation target for a suitable subset of OCaml, Haskell, or any other expression-based programming language.

$$
\begin{array}{rcl}
\tau & ::= & \mathsf{int32} \mid \mathsf{int64} \mid \mathsf{unit} \mid \{\overrightarrow{f = \tau}\} \mid \mathsf{buf}\ \tau \mid \alpha \\
v & ::= & x \mid g \mid k : \tau \mid () \mid \{\overrightarrow{f = v}\} \\
e & ::= & \mathsf{readbuf}\ e_1\ e_2 \mid \mathsf{writebuf}\ e_1\ e_2\ e_3 \mid \mathsf{newbuf}\ n\ (e_1 : \tau) \\
 & \mid & \mathsf{subbuf}\ e_1\ e_2 \mid e.f \mid v \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\
 & \mid & d\ \overrightarrow{e} \mid \mathsf{let}\ x : \tau = e_1\ \mathsf{in}\ e_2 \mid \{\overrightarrow{f = e}\} \mid e \oplus n \mid \mathsf{for}\ i \in [0; n)\ e \\
P & ::= & \cdot \mid \mathsf{let}\ d = \lambda\overrightarrow{y : \tau}.\ e_1 : \tau_1, P \mid \mathsf{let}\ g : \tau = e, P
\end{array}
$$

Figure 2.7: λow* syntax

Translating λow* to C♭ We explain our translation via an example: the implementation of the `fadd` function for Curve25519. The function takes two arrays of five limbs each, adds up each limb pairwise (using a for-loop) and stores the result in the output array. It comes with the precondition (elided) that the addition must not overflow, and therefore limb addition does not produce any carries. The loop has an invariant (elided) that guarantees that the final result matches the high-level specification of `fadd`.

```
let fadd (dst: felem) (a b: felem): Stack unit ... =
  let invariant = ... in
  C.Loops.for 0ul 5ul invariant
    (fun i -> dst.(i) <- a.(i) + b.(i))
```

This function formally belongs to EMF*, the formal model for F* (Figure 2.6). The first transformation is erasure, which gets rid of the computationally-irrelevant parts of the program: this means removing the pre- and post-condition, as well as any mention of the loop invariant, which is relevant only for proofs. After erasure, this function belongs to λow*.

λow* is presented in Figure 2.7. λow* is a first-order lambda calculus, with recursion. It is equipped with stack-allocated buffers (arrays), which support: writebuf, readbuf, newbuf, and subbuf for pointer arithmetic. These operations take indices, lengths or offsets expressed in *array elements* (not bytes). λow* also supports structures, which can be passed around as values (as in C). Structures may be stored within an array, or may appear within another structure. They remain immutable; to pass a structure by reference, one has to place it within an array of size one. None of: in-place mutation of a field; taking the address of a field; flat (packed) arrays within structures are supported. This accurately matches what is presently implemented in Low* and the KreMLin compiler.

$$
\begin{array}{rcl}
\hat{\tau} & ::= & \mathsf{int32} \mid \mathsf{int64} \mid \mathsf{unit} \mid \mathsf{pointer} \\
\hat{v} & ::= & \ell \mid g \mid k : \hat{\tau} \mid () \\
\hat{e} & ::= & \mathsf{read}_n\ \hat{e} \mid \mathsf{write}_n\ \hat{e}_1\ \hat{e}_2 \mid \mathsf{new}\ \hat{e} \mid \hat{e}_1 \oplus \hat{e}_2 \mid \ell := \hat{e} \mid \hat{v} \mid \hat{e}_1;\ \hat{e}_2 \\
 & \mid & \mathsf{if}\ \hat{e}_1\ \mathsf{then}\ \hat{e}_2\ \mathsf{else}\ \hat{e}_3 : \hat{\tau} \mid \mathsf{for}\ \ell \in [0; n)\ \hat{e} \mid \hat{e}_1 \times \hat{e}_2 \mid \hat{e}_1 + \hat{e}_2 \mid d\ \overrightarrow{\hat{e}} \\
\hat{P} & ::= & \cdot \mid \mathsf{let}\ d = \lambda \overrightarrow{\ell : \hat{\tau}}. \overrightarrow{\ell : \hat{\tau}}, \hat{e} : \hat{\tau}, \hat{P} \mid \mathsf{let}\ g : \hat{\tau} = \hat{e}, \hat{P}
\end{array}
$$

Figure 2.8: C♭ syntax

Base types are 32-bit and 64-bit integers; integer constants are annotated with their types. The type α stands for a secret type, which we discuss in the next section. For simplicity, the scope of a stack allocation is always the enclosing function declaration.

Looking at the `fadd` example above, the function belongs to Low⋆ (after erasure) because: its signature is in the **`Stack`** effect, i.e. it verifies against the C-like memory model; it uses imperative mutable updates over pointers, i.e. the `felem` types and the `<-` operator; it uses the C loops library. As such, `fadd` can be successfully interpreted as the following λow⋆ term:

$$
\begin{array}{l}
\mathsf{let}\ \mathit{fadd} = \lambda(\mathit{dst} : \mathsf{buf}\ \mathsf{int64})(a : \mathsf{buf}\ \mathsf{int64})(b : \mathsf{buf}\ \mathsf{int64}). \\
\quad \mathsf{for}\ i \in [0; 5).\ \mathsf{writebuf}\ \mathit{dst}\ i\ (\mathsf{readbuf}\ a\ i + \mathsf{readbuf}\ b\ i)
\end{array}
$$

λow⋆ enjoys typing preservation, but not subject reduction. Indeed, λow⋆ programs are only guaranteed to terminate if they result from a well-typed F⋆ program that performed verification in order to guarantee spatial and temporal safety. In the example above, the type system of λow⋆ does *not* guarantee that the memory accesses are within bounds; this is only true because verification was performed over the original EMF⋆ program.

The differences here compared to the original presentation [12] are as follows. First, we impose no syntactic constraints on λow⋆, i.e. we do not need to anticipate on the statement language by requiring that all writebuf operations be immediately under a let. Second, we do not model in-place mutable structures, something that remains, at the time of writing, unimplemented by the Low⋆/KreMLin toolchain. Third, we add a raw pointer addition $e \oplus n$ that appears only as a temporary technical device during the structure allocation transformation (below).

C♭ (Figure 2.8) resembles λow⋆, but: i) eliminates structures altogether, ii) only retains a generic pointer type, iii) expresses all memory operations (pointer addition, write, reads, allocation) in terms of byte addresses, offsets and sizes, and iv) trades lexical scoping in favor of local names. As in WebAssembly,

$$
\begin{array}{lcl}
\mathsf{let}\ d = \lambda y : \tau_1.\ e : \tau_2 & \rightsquigarrow & \mathsf{let}\ d = \lambda y : \mathsf{buf}\ \tau_1.\ [\mathsf{readbuf}\ y\ 0/y]e : \tau_2 \\
 & & \quad \text{if } \tau_1 \text{ is a struct type} \\
\mathsf{let}\ d = \lambda y : \tau_1.\ e : \tau_2 & \rightsquigarrow & \mathsf{let}\ d = \lambda y : \tau_1.\ \lambda r : \mathsf{buf}\ \tau_2.\ \mathsf{let}\ x : \tau_2 = e\ \mathsf{in}\ \mathsf{writebuf}\ r\ 0\ x : \mathsf{unit} \\
 & & \quad \text{if } \tau_2 \text{ is a struct type} \\
 & & \\
f\ (e : \tau) & \rightsquigarrow & \mathsf{let}\ x : \mathsf{buf}\ \tau = \mathsf{newbuf}\ 1\ e\ \mathsf{in}\ f\ x \\
 & & \quad \text{if } \tau \text{ is a struct type} \\
(f\ e) : \tau & \rightsquigarrow & \mathsf{let}\ x : \mathsf{buf}\ \tau = \mathsf{newbuf}\ 1\ (\_ : \tau)\ \mathsf{in}\ f\ e\ x;\ \mathsf{readbuf}\ x\ 0 \\
 & & \quad \text{if } \tau \text{ is a struct type} \\
 & & \\
\mathsf{let}\ x : \tau = e_1\ \mathsf{in}\ e_2 & \rightsquigarrow & \mathsf{let}\ x : \mathsf{buf}\ \tau = \mathsf{take\_addr}\ e_1\ \mathsf{in}\ [\mathsf{readbuf}\ x\ 0/x]e_2 \\
 & & \quad \text{if } \tau \text{ is a struct type} \\
\{\overrightarrow{f = e}\} \quad \text{(not under newbuf)} & \rightsquigarrow & \mathsf{let}\ x : \mathsf{buf}\ \{\overrightarrow{f = \tau}\} = \mathsf{newbuf}\ 1\ \{\overrightarrow{f = e}\}\ \mathsf{in}\ \mathsf{readbuf}\ x\ 0 \\
 & & \quad \text{if } \tau \text{ is a struct type} \\
 & & \\
\mathsf{take\_addr}(\mathsf{readbuf}\ e\ n) & \rightsquigarrow & \mathsf{subbuf}\ e\ n \\
\mathsf{take\_addr}((e : \overrightarrow{f : \tau}).f) & \rightsquigarrow & \mathsf{take\_addr}(e) \oplus \mathsf{offset}(\overrightarrow{f : \tau}, f) \\
\mathsf{take\_addr}(\mathsf{let}\ x : \tau = e_1\ \mathsf{in}\ e_2) & \rightsquigarrow & \mathsf{let}\ x : \tau = e_1\ \mathsf{in}\ \mathsf{take\_addr}\ e_2 \\
\mathsf{take\_addr}(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3) & \rightsquigarrow & \mathsf{if}\ e_1\ \mathsf{then}\ \mathsf{take\_addr}\ e_2\ \mathsf{else}\ \mathsf{take\_addr}\ e_3
\end{array}
$$

Figure 2.9: Ensuring all structures have an address

functions in C♭ declare the set of local mutable variables they introduce, including their parameters.

Translating from λow* to C♭ involves three key steps: ensuring that all structures have an address in memory; converting let-bindings into local variable assignments; laying out structures in memory.

1) Desugaring structure values. Structures are values in λow* but not in C♭. In order to compile these, we make sure every structure is allocated in memory, and enforce that only pointers to such structures are passed around. This is achieved via a mundane type-directed λow*-to-λow* transformation detailed in Figure 2.9. The first two rules change the calling-convention of functions to take pointers instead of structures; and to take a destination address instead of returning a structure. The next two rules enact the calling-convention changes at call-site, introducing an uninitialized buffer as a placeholder for the return value of f. The next rule ensures that let-bindings have pointer types instead of structure types. The last rule actually implements the allocation of structure literals in memory.

$$
\begin{array}{lcll}
\mathsf{size\ int32} & = & 4 & \\
\mathsf{size\ unit} & = & 4 & \\
\mathsf{size\ int64} & = & 8 & \\
\mathsf{size\ buf}\ \tau & = & 4 & \\
\mathsf{size}\ \overrightarrow{f:\tau} & = & \mathsf{offset}\ (\overrightarrow{f:\tau}, f_n) + \mathsf{size}\ \tau_n & \\
 & & & \\
\mathsf{offset}\ (\overrightarrow{f:\tau}, f_0) & = & 0 & \\
\mathsf{offset}\ (\overrightarrow{f:\tau}, f_{i+1}) & = & \mathsf{align}(\mathsf{offset}\ (\overrightarrow{f:\tau}, f_i) + \mathsf{size}\ \tau_i, & \\
 & & \qquad \mathsf{alignment}\ \tau_{i+1}) & \\
 & & & \\
\mathsf{alignment}(\overrightarrow{f:\tau}) & = & 8 & \\
\mathsf{alignment}(\tau) & = & \mathsf{size}\ \tau & \text{otherwise} \\
 & & & \\
\mathsf{align}(k, n) & = & k & \text{if } k \bmod n = 0 \\
\mathsf{align}(k, n) & = & k + n - (k \bmod n) & \text{otherwise}
\end{array}
$$

Figure 2.10: Structure layout algorithm

The auxiliary take_addr function propagates the address-taking operation down the control flow. When taking the address of sub-fields, a raw pointer addition, in bytes, is generated. Unspecified cases are ruled out either by typing or by the previous transformations.

This phase, after introducing suitable let-bindings (elided), establishes the following invariants: i) the only subexpressions that have structure types are of the form $\{\overrightarrow{f = e}\}$ or readbuf $e\ n$ and ii) $\{\overrightarrow{f = e}\}$ appears exclusively as an argument to newbuf.

2) Assigning local variables. The desugaring of structure values was performed within λow*. We now present the translation rules from λow* to C♭ (Figure 2.11 and Figure 2.12). Our translation judgements from λow* to C♭ are of the form $G; V \vdash e : \tau \Rightarrow e' : \tau' \dashv V'$. The translation takes G, a (fixed) map from λow* globals to C♭ globals; V, a mapping from λow* variables to C♭ locals; and $e : \tau$, a λow* expression. It returns $\hat{e} : \hat{\tau}$, the translated C♭ expression, and V', which extends V with the variable mappings allocated while translating e.

We leave the discussion of the WRITE* rules to the next paragraph, and now focus on the general translation mechanism and the handling of variables.

Since λow* is a lambda-calculus with a true notion of *value*, let-bound variables cannot be mutated, meaning that they can be trivially translated as C♭ local variables. We thus compile a λow* let-binding let $x = e_1$ to a C♭ assignment $\ell := \hat{e}_1$ (rule LET). We chain the V environment throughout the

$$\textsc{Let}\quad \frac{\ell \text{ fresh} \qquad \begin{array}{l} G; V \vdash e_1 : \tau_1 \Rrightarrow \hat{e}_1 : \hat{\tau}_1 \dashv V' \\ G; (x \mapsto \ell, \hat{\tau}_1) \cdot V' \vdash e_2 : \tau_2 \Rrightarrow \hat{e}_2 : \hat{\tau}_2 \dashv V'' \end{array}}{G; V \vdash \mathsf{let}\ x : \tau_1 = e_1\ \mathsf{in}\ e_2 : \tau_2 \Rrightarrow \ell := \hat{e}_1; \hat{e}_2 : \hat{\tau}_2 \dashv V''}$$

$$\textsc{FunDecl}\quad \frac{G; \overrightarrow{y \mapsto \ell, \hat{\tau}} \vdash e_1 : \tau_1 \Rrightarrow \hat{e}_1 : \hat{\tau}_1 \dashv \overrightarrow{x \mapsto \ell', \hat{\tau}'} \cdot \overrightarrow{y \mapsto \ell, \hat{\tau}}}{G \vdash \mathsf{let}\ d = \lambda \overrightarrow{y : \tau}.\ e_1 : \tau_1 \Rrightarrow \mathsf{let}\ d = \lambda \overrightarrow{\ell : \hat{\tau}}. \overrightarrow{\ell' : \hat{\tau}'}, \hat{e_1} : \hat{\tau}_1}$$

$$\textsc{Var}\quad \frac{V(x) = \ell, \tau}{G; V \vdash x \Rrightarrow \ell : \tau \dashv V}$$

$$\textsc{BufWrite}\quad \frac{G; V \vdash \mathsf{writeB}\ (e_1 + e_2 \times \mathsf{size}\ \tau_1)\ e_3 \Rrightarrow \hat{e} \dashv V'}{G; V \vdash \mathsf{writebuf}\ (e_1 : \tau_1)\ e_2\ e_3 \Rrightarrow \hat{e} : \mathsf{unit} \dashv V'}$$

$$\textsc{WriteInt32}\quad \frac{G; V \vdash e_1 \Rrightarrow \hat{e}_1 \dashv V' \qquad G; V' \vdash e_2 \Rrightarrow \hat{e}_2 \dashv V''}{G; V \vdash \mathsf{writeB}\ e_1\ (e_2 : \mathsf{int32}) \Rrightarrow \mathsf{write}_4\ \hat{e}_1\ \hat{e}_2 \dashv V''}$$

$$\textsc{WriteLiteral}\quad \frac{G; V_i \vdash \mathsf{writeB}\ (e + \mathsf{offset}\ (\overrightarrow{f : \tau}, f_i))\ e_i \Rrightarrow \hat{e}_i \dashv V_{i+1}}{G; V_0 \vdash \mathsf{writeB}\ e\ (\{\overrightarrow{f = e : \tau}\}) \Rrightarrow \hat{e}_0; \ldots; \hat{e}_{n-1} \dashv V_n}$$

$$\textsc{WriteDeRef}\quad \frac{\ell \text{ fresh} \qquad V' = \ell, \mathsf{int32} \cdot V \qquad G; V \vdash v_i \Rrightarrow \hat{v}_i \dashv V \qquad \mathsf{memcpy}\ v_1\ v_2\ n = \mathsf{for}\ \ell \in [0; n)\ \mathsf{write}_1\ (v_1 + \ell)\ (\mathsf{read}_1\ (v_2 + \ell)\ 1)}{G; V \vdash \mathsf{writeB}\ v_1\ (\mathsf{readbuf}\ (v_2 : \tau_2)\ 0) \Rrightarrow \mathsf{memcpy}\ v_1\ v_2\ (\mathsf{size}\ \tau_2) \dashv V'}$$

$$\textsc{BufNew}\quad \frac{\ell, \ell' \text{ fresh} \qquad G; x \mapsto (\ell, \mathsf{int32}) \cdot y \mapsto (\ell', \mathsf{int32}) \cdot V \vdash \mathsf{writeB}\ (x + \mathsf{size}\ \tau \times y)\ v_1 \Rrightarrow \hat{e} \dashv V'}{G; V \vdash \mathsf{newbuf}\ n\ (v : \tau) \Rrightarrow \ell := \mathsf{new}\ (n \times \mathsf{size}\ \tau);\ \mathsf{for}\ \ell' \in [0; n)\ \hat{e};\ \ell \dashv V'}$$

Figure 2.11: Translating from λow* to C♭ (selected rules)

IfThenElse

$$\frac{G; V \vdash e_1 : \mathsf{bool} \Rrightarrow \hat{e}_1 : \mathsf{bool} \dashv V' \quad G; V' \vdash e_2 : \tau \Rrightarrow \hat{e}_2 : \hat{\tau} \dashv V'' \quad G; V'' \vdash e_3 : \tau \Rrightarrow \hat{e}_3 : \hat{\tau} \dashv V'''}{G; V \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau \Rrightarrow \mathsf{if}\ \hat{e}_1\ \mathsf{then}\ \hat{e}_2\ \mathsf{else}\ \hat{e}_3 : \hat{\tau} \dashv V'''}$$

BufRead

$$\frac{G; V \vdash e_1 : \mathsf{buf}\ \tau \Rrightarrow \hat{e}_1 : \mathsf{pointer} \dashv V' \quad G; V' \vdash e_2 : \mathsf{int32} \Rrightarrow \hat{e}_2 : \mathsf{int32} \dashv V'' \quad \mathsf{size}(\tau) = n}{G; V \vdash \mathsf{readbuf}\ e_1\ e_2 : \tau \Rrightarrow \mathsf{read}_n(\hat{e}_1 + n \times \hat{e}_2) : \hat{\tau} \dashv V''}$$

BufSub

$$\frac{G; V \vdash e_1 : \mathsf{buf}\ \tau \Rrightarrow \hat{e}_1 : \mathsf{pointer} \dashv V' \quad G; V' \vdash e_2 : \mathsf{int32} \Rrightarrow \hat{e}_2 : \mathsf{int32} \dashv V'' \quad \mathsf{size}(\tau) = n}{G; V \vdash \mathsf{subbuf}\ e_1\ e_2 : \mathsf{buf}\ \tau \Rrightarrow \hat{e}_1 + n \times \hat{e}_2 : \mathsf{pointer} \dashv V''}$$

Field

$$\frac{G; V \vdash e : \mathsf{buf}\ \tau \Rrightarrow \hat{e} : \mathsf{pointer} \dashv V' \quad \mathsf{offset}(\tau, f) = k \qquad \mathsf{size}(\tau_f) = n}{G; V \vdash (\mathsf{readbuf}\ e\ 0).f : \tau_f \Rrightarrow \mathsf{read}_n(\hat{e} + k) : \hat{\tau}_f \dashv V'}$$

PointerAdd

$$\frac{G; V \vdash e : \mathsf{buf}\ \tau \Rrightarrow \hat{e} : \mathsf{pointer} \dashv V'}{G; V \vdash e \oplus n : \mathsf{buf}\ \tau \Rrightarrow \hat{e} + n : \mathsf{pointer} \dashv V'}$$

FunCall

$$\frac{G; V \vdash e : \tau_1 \Rrightarrow \hat{e} : \hat{\tau}_1 \dashv V'}{G; V \vdash d\ e : \tau_2 \Rrightarrow d\ \hat{e} : \hat{\tau}_2 \dashv V'}$$

Unit

$$\frac{}{G; V \vdash () : \mathsf{unit} \Rrightarrow () : \mathsf{unit} \dashv V}$$

Constant

$$\frac{}{G; V \vdash k : \tau \Rrightarrow k : \hat{\tau} \dashv V}$$

Global

$$\frac{g \in G}{G; V \vdash g : \tau \Rrightarrow g : \hat{\tau} \dashv V}$$

ForLoop

$$\frac{G; (i \mapsto \ell, \mathsf{int32}) \cdot V \vdash e : \mathsf{unit} \Rrightarrow \hat{e} : \mathsf{unit} \dashv V' \qquad \ell\ \textbf{fresh}}{G; V \vdash \mathsf{for}\ i \in [0; n)\ e : \mathsf{unit} \Rrightarrow \mathsf{for}\ \ell \in [0; n)\ \hat{e} : \mathsf{unit} \dashv V'}$$

Figure 2.12: Translating from λow* to C♭ (remaining rules). Some notes: Field: the type τ_f can only be a non-struct type per our invariant.

premises, meaning that the rule produces an extended V'' that contains the additional $x \mapsto \ell, \hat{\tau}$ mapping. Translating a variable then boils down to a lookup in V (rule VAR).

The translation of top-level functions (rule FUNDECL) calls into the translation of expressions. The input variable map is pre-populated with bindings for the function parameters, and the output variable map generates extra bindings $\overrightarrow{y}$ for the locals that are now needed by that function.

3) Performing struct layout. Going from λow^* to C♭, BUFWRITE and BUFNEW (Figure 2.11) call into an auxiliary writeB function, defined inductively via the rules WRITE*. This function performs the layout of structures in memory, relying on a set of mutually-defined functions (Figure 2.10): size computes the number of bytes occupied in memory by an element of a given type, and offset computes the offset in bytes of a field within a given structure. Fields within a structure are aligned on 64-bit boundaries (for nested structures) or on their intrinsic size (for integers), which WebAssembly can later leverage.

We use writeB as follows. From BUFWRITE and BUFNEW, we convert a pair of a base pointer and an index into a byte address using size, then call writeB e_1 e_2 to issue a series of writes that will lay out e_2 at address e_1. Writing a base type is trivial (rule WRITEINT32). Recall that from the earlier desugaring, only two forms can appear as arguments to writebuf: writing a structure located at another address boils down to a memcpy operation (rule WRITEDEREF), while writing a literal involves recursively writing the individual fields at their respective offsets (rule WRITELITERAL).

The allocation of a buffer whose initial value is a struct type is desugared into the allocation of uninitialized memory followed by a series of writes in a loop (rule BUFNEW).

After translation to C♭, the earlier `fadd` function now features four locals: three of type pointer for the function arguments, and one for the loop index; buffer operations take byte addresses and widths.

$$
\begin{array}{l}
\mathsf{let}\ \mathit{fadd} = \lambda(\ell_0, \ell_1, \ell_2 : \mathsf{pointer})(\ell_3 : \mathsf{int32}). \\
\quad \mathsf{for}\ \ell_3 \in [0; 5). \\
\quad\quad \mathsf{write}_8\ (\ell_0 + i \times 8)\ (\mathsf{read}_8\ (\ell_1 + i \times 8) + \mathsf{read}_8(\ell_2 + i \times 8))
\end{array}
$$

Translating C♭ to WebAssembly The C♭ to WebAssembly translation appears in Figure 2.13). A C♭ expression $\hat{e}$ compiles to a series of WebAssembly instructions $\overrightarrow{i}$.

WRITE32 compiles a 4-byte write to WebAssembly. WebAssembly is a stack-

WRITE32

$$\frac{\hat{e}_1 \Rrightarrow \overrightarrow{i_1} \qquad \hat{e}_2 \Rrightarrow \overrightarrow{i_2}}{\mathsf{write}_4\ \hat{e}_1\ \hat{e}_2 \Rrightarrow \overrightarrow{i_1};\ \overrightarrow{i_2};\ \mathsf{i32.store};\ \mathsf{i32.const}\ 0}$$

NEW

$$\frac{\hat{e} \Rrightarrow \overrightarrow{i}}{\mathsf{new}\ \hat{e} \Rrightarrow \overrightarrow{i};\ \mathsf{call\ grow\_stack}}$$

FOR

$$\frac{\hat{e} \Rrightarrow \overrightarrow{i}}{\begin{array}{l}\mathsf{for}\ \ell \in [0; n)\ \hat{e} \Rrightarrow \\ \mathsf{loop}(\overrightarrow{i}; \mathsf{drop}; \\ \quad \mathsf{get\_local}\ \ell; \mathsf{i32.const}\ 1; \mathsf{i32.op+}; \mathsf{tee\_local}\ \ell; \\ \quad \mathsf{i32.const}\ n; \mathsf{i32.op}\ =; \mathsf{br\_if}); \mathsf{i32.const}\ 0\end{array}}$$

FUNC

$$\frac{\hat{e} \Rrightarrow \overrightarrow{i} \qquad \hat{\tau}_i \Rrightarrow t_i}{\begin{array}{l}\mathsf{let}\ d = \lambda\overrightarrow{\ell_1 : \hat{\tau}_1}.\overrightarrow{\ell_2 : \hat{\tau}_2}, \hat{e} : \hat{\tau} \Rrightarrow \\ d = \mathsf{func}\ \overrightarrow{t_1} \to t\ \mathsf{local}\ \overrightarrow{\ell_1 : t_1 \cdot \ell_2 : t_2} \cdot \ell : t. \\ \qquad \mathsf{call\ get\_stack};\ \overrightarrow{i};\ \mathsf{store\_local}\ \ell;\ \mathsf{call\ set\_stack};\ \mathsf{get\_local}\ \ell\end{array}}$$

Figure 2.13: Translating from C♭ to WebAssembly (selected rules)

based language, meaning we accumulate the arguments to a function on the operand stack before issuing a call instruction: the sequence $\overrightarrow{i_1};\ \overrightarrow{i_2}$ pushes two arguments on the operand stack, one for the 32-bit address, and one for the 32-bit value. The store instruction then consumes these two arguments.

By virtue of typing, this expression has type unit; for the translation to be valid, we must push a unit value on the operand stack, compiled as i32.const 0. A similar mechanism operates in FOR, where we drop the unit value pushed by the loop body on the operand stack (a loop growing the operand stack would be ill-typed in WebAssembly), and push it back after the loop has finished.

WebAssembly only offers a flat view of memory, but Low* programs are written against a memory stack where array allocations take place. We thus need to implement run-time memory management, the only non-trivial bit of our translation. Our implementation strategy is as follows. At address 0, the memory always contains the address of the top of the stack, which is initially 1. We provide three functions for run-time memory stack management.

$$
\begin{array}{lcl}
\mathsf{get\_stack} & = & \mathsf{func}\ [] \to \mathsf{i32}\ \mathsf{local}\ [] \\
 & & \mathsf{i32.const}\ 0;\ \mathsf{i32.load} \\
\mathsf{set\_stack} & = & \mathsf{func}\ \mathsf{i32} \to [] \ \mathsf{local}\ \overrightarrow{\ell : \mathsf{i32}} \\
 & & \mathsf{i32.const}\ 0;\ \mathsf{get\_local}\ \ell;\ \mathsf{i32.store} \\
\mathsf{grow\_stack} & = & \mathsf{func}\ \mathsf{i32} \to \mathsf{i32}\ \mathsf{local}\ \overrightarrow{\ell : \mathsf{i32}} \\
 & & \mathsf{call\ get\_stack};\ \mathsf{get\_local}\ \ell;\ \mathsf{i32.op+}; \\
 & & \mathsf{call\ set\_stack};\ \mathsf{call\ get\_stack}
\end{array}
$$

Thus, allocating uninitialized memory on the memory stack merely amounts to a call to grow_stack (rule NEW). Functions save the top of the memory stack on top of the operand stack, then restore it before returning their value (rule FUNC).

Combining all these rules, the earlier `fadd` is compiled as shown in Figure 2.14.

This formalization serves as a succinct description of our compiler as well as a strong foundation for future theoretical developments, while subsequent sections demonstrate the applicability and usefulness of our approach. This is, we hope, only one of many future papers connecting state-of-the-art verification tools to WebAssembly. As such, the present paper leaves many areas to be explored. In particular, we leave proofs for these translations to future work. The original formalization only provides paper proofs in the appendix [12]; since we target simpler and cleaner semantics (WebAssembly instead of C), we believe the next ambitious result should be to perform a mechanical proof of our translation, leveraging recent formalizations of the WebAssembly semantics [56].

Secret Independence in WebAssembly When compiling verified source code in high-level programming language like F⋆ (or C) to a low-level machine language like WebAssembly (or x86 assembly), a natural concern is whether the compiler preserves the security guarantees proved about source code. Verifying the compiler itself provides the strongest guarantees but is an ambitious project [14].

Manual review of the generated code and comprehensive testing can provide some assurance, and so indeed we extensively audit and test the WebAssembly generated from our compiler. However, testing can only find memory errors and correctness bugs. For cryptographic code, we are also concerned that some compiler optimizations may well introduce side-channel leaks even if

```
fadd = func [int32; int32; int32] → []
  local [ℓ0, ℓ1, ℓ2 : int32; ℓ3 : int32; ℓ : int32].
  call get_stack; loop(
    // Push dst + 8*i on the stack
    get_local ℓ0; get_local ℓ3; i32.const 8; i32.binop*; i32.binop+
    // Load a + 8*i on the stack
    get_local ℓ1; get_local ℓ3; i32.const 8; i32.binop*; i32.binop+
    i64.load
    // Load b + 8*i on the stack (elided, same as above)
    // Add a.[i] and b.[i], store into dst.[i]
    i64.binop+; i64.store
    // Per the rules, return unit
    i32.const 0; drop
    // Increment i; break if i == 5
    get_local ℓ3; i32.const 1; i32.binop+; tee_local ℓ3
    i32.const 5; i32.op =; br_if
  ); i32.const 0
  store_local ℓ ; call set_stack; get_local ℓ
```

Figure 2.14: Compilation of the `fadd` example to WebAssembly

they were not present in the source.

We illustrate the problem with a real-world example taken from the Curve-25519 code in LibSignal-JavaScript, which is compiled using Emscripten from C to JavaScript (*not* to WebAssembly). The source code includes an `fadd` function in C very similar to the one we showed page 79. At the heart of this function is 64-bit integer addition, which a C compiler translates to some constant-time addition instruction on any modern platform.

Recall, however, that JavaScript has a single numeric type, IEEE-754 double precision floats, which can accurately represent 32-bit values but not 64-bit values. As such, JavaScript is a 32-bit target, so to compile `fadd`, Emscripten generates and uses the following 64-bit addition function in JavaScript:

```
function _i64Add(a, b, c, d) {
    // x = a + b*2^32 ; y = c + d*2^32 ; result = l + h*2^32
    a = a|0; b = b|0; c = c|0; d = d|0;
    var l = 0, h = 0;
    l = (a + c)>>>0;
    // Add carry from low word to high word on overflow.
```

```
    h = (b + d + (((l>>>0) < (a>>>0))|0))>>>0;
    return ((tempRet0 = h,l|0)|0);
}
```

This function now has a potential side-channel leak, because of the `(l>>>0) < (a>>>0)` subterm, a direct comparison between `l` and `a`, one or both of which could be secret. Depending on how the JavaScript runtime executes this comparison, it may take different amounts of time for different inputs, hence leaking these secret values. These kinds of timing attacks are an actual concern for LibSignal-JavaScript, in that an attacker who can measure fine-grained running time (say from another JavaScript program running in parallel) may be able to obtain the long-term identity keys of the participants.

This exact timing leak does not occur in the WebAssembly output of Emscripten, since 64-bit addition is available in WebAssembly, but how do we know that other side-channels are not introduced by one of the many optimizations? This is a problem not just for Emscripten but for all optimizing compilers, and the state-of-the-art for side-channel analysis of cryptographic code is to check that the generated machine code preserves so-called "constant-time" behaviour [57], [58].

We propose to build a validation pass on the WebAssembly code generated from KreMLin to ensure that it preserves the side-channel guarantees proved for the Low* source code. To ensure that these guarantees are preserved all the way to machine code, we hope to eventually connect our toolchain to CT-Wasm [59], a new proposal that advocates for a notion of secrets directly built into the WebAssembly semantics.

HACL* code manipulates arrays of machine integers of various sizes and by default, HACL* treats all these machine integers as secret, representing them by an abstract type (which we model as α in λow*) defined in a secret integer library. The only public integer values in HACL* code are array lengths and indices.

The secret integer library offers a controlled subset of integer operations known to be constant-time, e.g. the library rules out division or direct comparisons on secret integers. Secret integers cannot be converted to public integers (although the reverse is allowed), and hence we cannot print a secret integer, or use it as an index into an array, or compare its value with another integer. This programming discipline guarantees a form of timing side-channel resistance called *secret independence* at the level of the Low* source [12].

Carrying this type-based information all the way to WebAssembly, we de-

$$\textsc{Classify}\ \frac{C \vdash i : \pi}{C \vdash i : \sigma} \qquad \textsc{BinOpPub}\ \frac{o \text{ is constant-time}}{C \vdash t.\mathsf{binop}\ o : m\ m \to m} \qquad \textsc{BinOpPriv}\ \frac{o \text{ is not constant-time}}{C \vdash t.\mathsf{binop}\ o : \pi\ \pi \to \pi}$$

$$\textsc{Load}\ \frac{}{C \vdash t.\mathsf{load} : *\sigma\ \pi \to \sigma} \qquad \textsc{Local}\ \frac{C(\ell) = m}{C \vdash \mathsf{get\_local}\ \ell : [\,] \to m}$$

$$\textsc{Cond}\ \frac{C \vdash \overrightarrow{i_1} : \overrightarrow{m} \to \pi \qquad C \vdash \overrightarrow{i_{\{2,3\}}} : \overrightarrow{m} \to \overrightarrow{m}}{C \vdash \mathsf{if}\ \overrightarrow{i_1}\ \mathsf{then}\ \overrightarrow{i_2}\ \mathsf{else}\ \overrightarrow{i_3} : \overrightarrow{m}\ \pi \to \overrightarrow{m}}$$

Figure 2.15: Secret Independence Checker (selected rules)

velop a checker that analyzes the generated WebAssembly code to ensure that secret independence is preserved, even though Low* secret integers are compiled to regular integers in WebAssembly. We observe that adding such a checker is only made possible by having a custom toolchain that allows us to propagate secrecy information from the source code to the generated WebAssembly. It would likely be much harder to apply the same analysis to arbitrary optimized WebAssembly generated by Emscripten.

We ran our analysis on the entire WHACL* library; the checker validated all of the generated WebAssembly code. We experimented with introducing deliberate bugs at various points throughout the toolchain, and were able to confirm that the checker declined to validate the resulting code.

The rules for our secret independence checker are presented in Figure 2.15. We mimic the typing rules from the original WebAssembly presentation [44]: just like the typing judgement captures the effect of an instruction on the operand stack via a judgement $C \vdash i : \overrightarrow{t} \to \overrightarrow{t}$, our judgement $C \vdash i : \overrightarrow{m} \to \overrightarrow{m}$ captures the information-flow effect of an instruction on the operand stack.

The *context* C maps each local variable to either π (public) or σ (secret). The *mode* m is one of π, σ or $*\sigma$. The $*\sigma$ mode indicates a pointer to secret data, and embodies our hypothesis that all pointers point to secret data. (This assumption holds for the HACL* codebase, but we plan to include a more fine-grained memory analysis in future work.)

For brevity, Figure 2.15 omits administrative rules regarding sequential composition; empty sequences; and equivalence between $\overrightarrow{m}\ \overrightarrow{m_1} \to \overrightarrow{m}\ \overrightarrow{m_2}$ and

$\overrightarrow{m_1} \rightarrow \overrightarrow{m_2}$. The mode of local variables is determined by the context C (rule LOCAL). Constant time operations accept any mode m for their operands (rule BINOPPUB); if needed, one can always classify data (rule CLASSIFY) to ensure that the operands to BINOPPUB are homogeneous. For binary operations that are not constant-time (e.g. equality, division), the rules require that the operands be public. Conditionals always take a public value for the condition (rule COND). For memory loads, the requirement is that the address be a pointer to secret data (always true of all addresses), and that the index be public data (rule LOAD).

In order to successfully validate a program, the checker needs to construct a context C that assigns modes to variables. For function arguments, this is done by examining the original λow* type for occurrences of α, i.e. secret types. For function locals, we use a simple bidirectional inference mechanism, which exploits the fact that i) our compilation scheme never re-uses a local variable slot for different modes and ii) classifications are explicit, i.e. the programmer needs to explicitly cast public integers to secret in HACL*.

2.3.2. LibSignal*, when F* and HACL* meets WebAssembly

We now describe the first application of our toolchain: WHACL*, a WebAssembly version of the (previously existing) verified HACL* crypto library [35]. Compiling such a large body of code demonstrates the viability of our toolchain approach. Then, we extend this artifact to a Web-compatible implementation of the Signal protocol. To keep things brief, this subsection will only sum up contributions by Benjamin Beurdouche, mentioned in his own PhD dissertation [60] (Chapter 4). Please refer to his work or the related article [40] for more details.

WHACL*, HACL* for the Browser We successfully compiled all the algorithms above to WebAssembly using KreMLin, along with their respective test suites, and dub the resulting library WHACL*, for Web-HACL*, a novel contribution. All test vectors pass when the resulting WebAssembly code is run in a browser or in node.js, which serves as experimental validation for our compiler.

Once compiled to WebAssembly, there are several ways clients can leverage WHACL*. In a closed-world setting, the whole application can be written in Low*, meaning one compiles the entire client program with KreMLin in a single pass. In this scenario, JavaScript only serves as an entry point, and the

rest of the program execution happens solely within WebAssembly. KreMLin automatically generates boilerplate code to: load the WebAssembly modules; link them together, relying on JavaScript for only a few library functions (e.g. for debugging).

In an open-world setting, clients will want to use WHACL* from JavaScript. We rely on the KreMLin compiler to ensure that only the top-level API of WHACL* is exposed (via the `exports` mechanism of WebAssembly) to JavaScript. These top-level entry points abide by the restrictions of the WebAssembly-JavaScript FFI, and only use 32-bit integers (64-bit integers are not representable in JavaScript). Next, we automatically generate a small amount of glue code; this code is aware of the KreMLin compilation scheme, and takes JavaScript `ArrayBuffer`s as input, copies their contents into the WebAssembly memory, calls the top-level entry point, and marshals back the data from the WebAssembly memory into a JavaScript value. We package the resulting code as a portable node.js module for easy distribution.

We then evaluate WHACL* against versions of HACL* and libsodium compiled to WebAssembly using Emscripten. While the Emscripten-compiled versions of HACL* and libsodium exhibit somewhat similar performance, WHACL* is 1,5 to 3 times slower. This performance hit is consistent with the fact that our custom toolchain targeting WebAssembly does not enjoy all the fine-tuned optimizations that Emscripten performs, through its use of LLVM. However, there are several low-hanging optimizations that could be implemented in KreMLin for the kind of code used by WHACL*. We believe they could reduce the performance gap to an acceptable level for production deployments, but leave them as future work.

LibSignal*: Verified LibSignal in WebAssembly As our main case study, we rewrite and verify the core protocol code of LibSignal in F*. We compile our implementation to WebAssembly and embed the resulting code back within LibSignal-JavaScript to obtain a high-assurance drop-in replacement for this popular library, which is currently used in the desktop versions of WhatsApp, Skype, and Signal.

Our Signal implementation is likely the first cryptographic protocol implementation to be compiled to WebAssembly, and is certainly the first to be verified for correctness, memory safety, and side-channel resistance. In particular, we carefully design a defensive API between our verified WebAssembly code and the outer LibSignal JavaScript code so that we can try to preserve

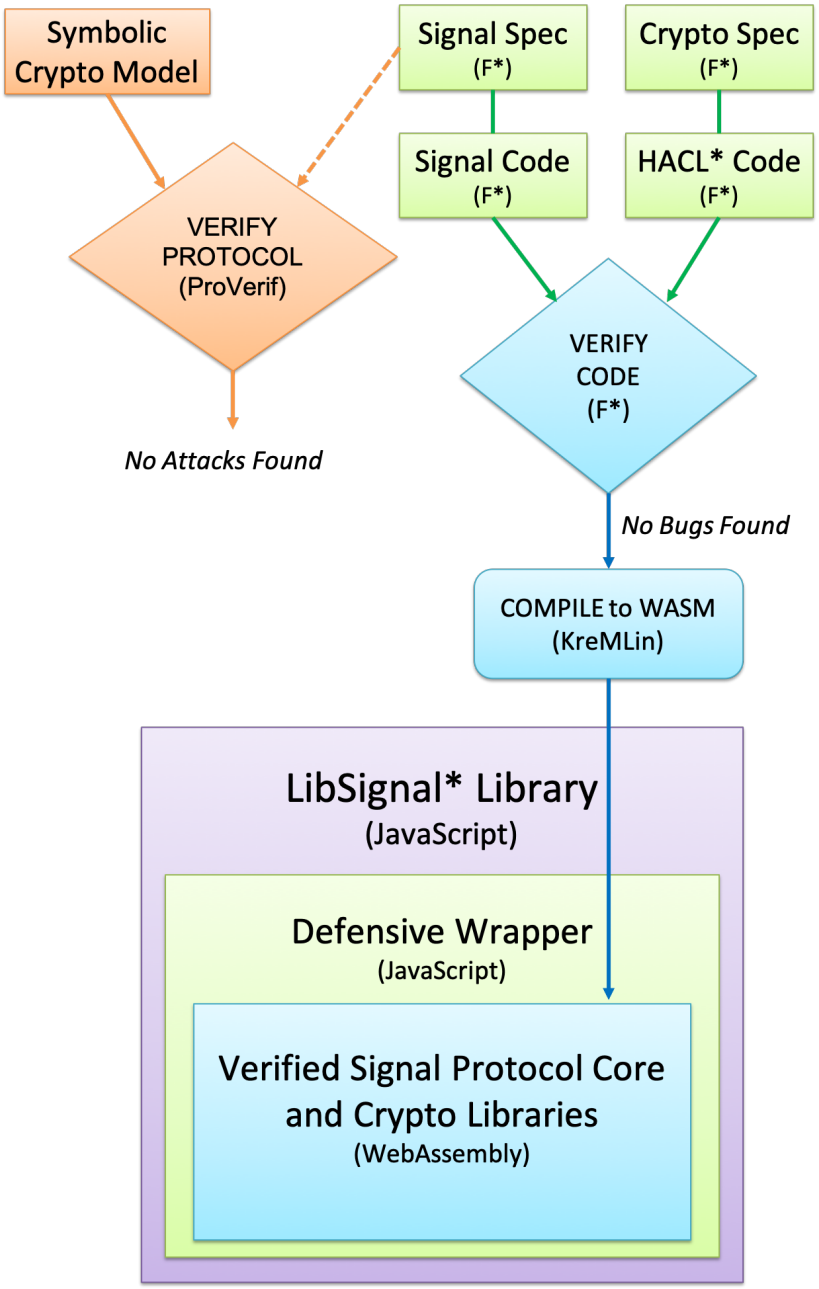

Figure 2.16: LibSignal*: We write an F* specification for the Signal protocol and verify its security by transcribing it into a ProVerif model. We then write a Low* implementation of Signal and verify it against the spec using F*. We compile the code to WebAssembly, link it with WHACL* and embed both modules within a defensive JavaScript wrapper in LibSignal-JavaScript.

some of the strong security guarantees of the Signal protocol, even against bugs in LibSignal. LibSignal* uses WHACL* cryptography for all the primitives that it needs, thus reaping the benefits of the previous application. Figure 2.16 offers an overview of the language and proof architecture of this verified artifact. The size of the F* development corresponding to the specification and implementation of the protocol, excluding the cryptographic primitives it relies on, is about 3,000 lines of code.

To craft LibSignal*, we followed all the steps of the methodology presented in Section 1.3.1. First, we cleaned up the Javascript codebase of the official Signal implementation, separating the critical core of the protocol aside from the glue code and key storage system. Second and third, we used the existing F* language and Low* domain-specific language to encode two versions of the protocol core: a high-level specification, manually translated to pure F* from the Javascript implementation, and a low-level implementation in Low*, bound to be extracted to WebAssembly with our novel KreMLin toolchain. Fourth, we prove functional equivalence between the two versions, as well as security of the protocol in the symbolic model (using Proverif [18]) and secret-independence for the implementation, which we carried down to WebAssembly using a translation validator implemented in KreMLin. Fifth, we plugged back the extracted WebAssembly to the original Signal codebase, using the Javascript-to-WebAssembly interoperability layer provided by Web browsers.

LibSignal* was evaluated on the official LibSignal test suite, and our new version is fully interoperable down to the byte with other implementations of the Signal protocol. We measured the performance of LibSignal* in the browser and found it to be on par with LibSignal; indeed, the performance gains of using WebAssembly seem to be offset by the performance loss of using WHACL*'s Ed25519 instead of libsodium's Ed25519 compiled from C to WebAssembly via Emscripten.

Conclusion

The layout of this chapter, starting from related work to detail our contribution to the F*/Low* ecosystem and finally LibSignal*, might leave the impression that LibSignal* is merely a pretext to showcase the agility of the F* verification ecosystem over other cryptographic verification provers. At the time of the completion of this work, in 2019, the methodology of Section 1.3.1 had not been clearly laid out and the primary motivation was indeed to extend HACL*

to cryptographic protocols; Signal was a desirable target in terms of impact due to its large real-world usage. However, we did not stay in the closed world of F* and did not build another proof framework to prove everything about Signal in the same theorem prover. Rather, we made use of the existing specialized prover Proverif [18] and repurposed in a lightweight way the KreMLin compiler to target an environment that critically lacked a high level of assurance: the Web. Hence, we emphasize the size and reach of the final artifact created, which attracted the attention of industrial users at the time, though we could not complete the technology transfer due to a lack of time and personal investment: establishing trust with the end-users and the engineers usually requires one or more internships where the formalizer can carefully study the codebase and perform the surgical operation of swapping existing code with a new, verified artifact.

Although we believe the main reason why LibSignal* was not adopted is social, the resulting language architecture of the artifact is composite and many steps are still uncertified or manual translations. This weakens the chain of trust; if a bug is found in the unverified parts of the chain, and has serious security consequences, this could hurt the credibility of formal verification as a whole and delay its adoption by industrial end-users. To improve these weaknesses, several measures could be taken: formalize and implement an automatic translation from specialized provers like Proverif and Cryptoverif [61] to F*, mechanize the semantics of WebAssembly and F* to mechanically certify the translation, etc. We leave those as future work.

Indeed, after LibSignal*, we chose to focus on a different improvement point. As argued in Section 1.2.1, the chain of trust has a topmost link: the correctness of the specifications used as the basis of the verified development. In LibSignal*, those specifications amount to only 500 lines of code only for the protocol part, excluding the specifications of the cryptographic primitives involved. While conciseness of specifications is key to reducing the trusted code base, we claim that ensuring that they are actually reviewed correctly by domain experts is equally important. Hence, the next chapter will focus on improving this review by cryptographers.

References

[1] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, *et al.*,

"Dependent types and multi-monadic effects in F*", in *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 256–270.

[2] D. Ahman, C. Hritcu, K. Maillard, G. Martínez, G. Plotkin, J. Protzenko, A. Rastogi, and N. Swamy, "Dijkstra monads for free", in *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, ACM, Jan. 2017, pp. 515–529. DOI: 10.1145/3009837.3009878. [Online]. Available: https://www.fstar-lang.org/papers/dm4free/.

[3] D. Ahman, C. Fournet, C. Hritcu, K. Maillard, A. Rastogi, and N. Swamy, "Recalling a witness: foundations and applications of monotonic state", *PACMPL*, vol. 2, no. POPL, 65:1–65:30, Jan. 2018. [Online]. Available: https://arxiv.org/abs/1707.02466.

[4] G. Martinez, D. Ahman, V. Dumitrescu, N. Giannarakis, C. Hawblitzel, C. Hriţcu, M. Narasimhamurthy, Z. Paraskevopoulou, C. Pit-Claudel, J. Protzenko, *et al.*, "Meta-F*: proof automation with SMT, tactics, and metaprograms", in *European Symposium on Programming*, Springer, Cham, 2019, pp. 30–59.

[5] K. Maillard, C. Hritcu, E. Rivas, and A. V. Muylder, *The next 700 relational program logics*, arXiv:1907.05244, Jul. 2019. [Online]. Available: https://arxiv.org/abs/1907.05244.

[6] The Coq Development Team, *The Coq Proof Assistant Reference Manual, version 8.7*, Oct. 2017. [Online]. Available: http://coq.inria.fr.

[7] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer Science & Business Media, 2002, vol. 2283.

[8] 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*, Springer, 2015, pp. 378–388.

[9] L. De Moura and N. Bjørner, "Z3: an efficient SMT solver", in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337–340.

[10] D. Aspinall, "Proof general: a generic tool for proof development", in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2000, pp. 38–43.

[11] A. Rastogi, G. Martínez, A. Fromherz, T. Ramananandro, and N. Swamy, *Programming and proving with indexed effects*, In submission, Jul. 2021. [Online]. Available: fstar-lang.org/papers/indexedeffects.

[12] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hritcu, K. Bhargavan, C. Fournet, and N. Swamy, "Verified low-level programming embedded in F*", *PACMPL*, vol. 1, no. ICFP, 17:1–17:29, Sep. 2017. DOI: 10.1145/3110261. [Online]. Available: http://arxiv.org/abs/1703.00053.

[13] S. Blazy and X. Leroy, "Mechanized semantics for the Clight subset of the C language", *Journal of Automated Reasoning*, vol. 43, no. 3, pp. 263–288, 2009.

[14] X. Leroy, "Formal certification of a compiler back-end or: programming a compiler with a proof assistant", *SIGPLAN Not.*, vol. 41, no. 1, pp. 42–54, Jan. 2006.

[15] B. Beurdouche, F. Kiefer, and T. Taubert, *Verified cryptography for Firefox 57*, https://blog.mozilla.org/security/2017/09/13/verified-cryptography-firefox-57/, 2017.

[16] J. Donenfeld, *Wireguard – formal verification*, https://www.wireguard.com/formal-verification/, 2020.

[17] 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 (S&P'21)*, 2021.

[18] B. Blanchet, "Modeling and verifying security protocols with the applied pi-calculus and ProVerif", *Foundations and Trends® in Privacy and Security*, vol. 1, no. 1-2, pp. 1–135, 2016.

[19] C. J. Cremers, "The Scyther tool: verification, falsification, and analysis of security protocols", in *International conference on computer aided verification*, Springer, 2008, pp. 414–418.

[20] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The Tamarin prover for the symbolic analysis of security protocols", in *International Conference on Computer Aided Verification*, Springer, 2013, pp. 696–701.

[21] G. Barthe, B. Grégoire, and S. Zanella Béguelin, "Formal certification of code-based cryptographic proofs", in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2009, pp. 90–101.

[22] D. A. Basin, A. Lochbihler, and S. R. Sefidgar, "CryptHOL: game-based proofs in higher-order logic", *Journal of Cryptology*, vol. 33, no. 2, pp. 494–566, 2020.

[23] B. Blanchet, "A computationally sound mechanized prover for security protocols", *IEEE Transactions on Dependable and Secure Computing*, vol. 5, no. 4, pp. 193–207, 2008.

[24] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, "Computer-aided security proofs for the working cryptographer", in *Annual Cryptology Conference*, Springer, 2011, pp. 71–90.

[25] L. Erkök and J. Matthews, "Pragmatic equivalence and safety checking in Cryptol", in *Proceedings of the 3rd workshop on Programming languages meets program verification*, 2009, pp. 73–82.

[26] 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*, 2013, pp. 15–18.

[27] A. Tomb, "Automated verification of real-world cryptographic implementations", *IEEE Security & Privacy*, vol. 14, no. 6, pp. 26–33, 2016.

[28] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala, "Simple high-level code for cryptographic arithmetic-with proofs, without compromises", in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 1202–1219.

[29] B. Delaware, C. Pit-Claudel, J. Gross, and A. Chlipala, "Fiat: deductive synthesis of abstract data types in a proof assistant", *Acm Sigplan Notices*, vol. 50, no. 1, pp. 689–700, 2015.

[30] D. Mercadier and P.-É. Dagand, "Usuba: high-throughput and constant-time ciphers, by construction", in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 157–173.

[31] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P.-Y. Strub, "Easycrypt: a tutorial", in *Foundations of security analysis and design vii*, Springer, 2013, pp. 146–166.

[32] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir, "Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations", in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 1217–1230.

[33] 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", in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1807–1823.

[34] 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 (SP)*, 2020, pp. 634–653.

[35] 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*, 2017, pp. 1789–1806.

[36] Y. Nir and A. Langley, "Chacha20 and poly1305 for IETF protocols", *RFC*, vol. 8439, pp. 1–46, 2018. DOI: 10.17487/RFC8439. [Online]. Available: https://doi.org/10.17487/RFC8439.

[37] 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 *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 917–934.

[38] K. R. M. Leino, "Dafny: an automatic program verifier for functional correctness", in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, Springer, 2010, pp. 348–370.

[39] A. Fromherz, N. Giannarakis, C. Hawblitzel, B. Parno, A. Rastogi, and N. Swamy, "A verified, efficient embedding of a verifiable assembly language", *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, 2019.

[40] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan, "Formally verified cryptographic web applications in WebAssembly", in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1256–1274. DOI: 10.1109/SP.2019.00064.

[41] *The lastpass password manager*. [Online]. Available: https://www.lastpass.com/how-lastpass-works.

[42] M. Marlinspike and T. Perrin, *The x3dh key agreement protocol*, https://signal.org/docs/specifications/x3dh/, 2016.

[43] T. Perrin and M. Marlinspike, *The double ratchet algorithm*, https://signal.org/docs/specifications/doubleratchet/, 2016.

[44] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly", in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017, pp. 185–200, ISBN: 978-1-4503-4988-8.

[45] *Web cryptography api*. [Online]. Available: https://www.w3.org/TR/WebCryptoAPI.

[46] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. 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 (Oakland)*, 2015, pp. 535–552.

[47] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub, "Implementing TLS with verified cryptographic security", in *IEEE Symposium on Security and Privacy (Oakland)*, 2013, pp. 445–459.

[48] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Z. Béguelin, K. Bhargavan, J. Pan, and J. K. Zinzindohoue, "Implementing and proving the TLS 1.3 record layer", in *IEEE Symposium on Security and Privacy (Oakland)*, 2017, pp. 463–482.

[49] 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 (Oakland)*, 2017, pp. 483–502.

[50] N. Kobeissi, K. Bhargavan, and B. Blanchet, "Automated verification for secure messaging protocols and their implementations: a symbolic and computational approach", in *2nd IEEE European Symposium on Security and Privacy (EuroSP)*, 2017, pp. 435–450.

[51] A. Zakai, "Emscripten: an LLVM-to-Javscript compiler", in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, 2011, pp. 301–312.

[52] A. Taly, Ú. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra, "Automated analysis of security-critical JavaScript APIs", in *IEEE Symposium on Security and Privacy (Oakland)*, 2011, pp. 363–378.

[53] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis, "Defensive JavaScript", in *Foundations of Security Analysis and Design VII*, Springer, 2014, pp. 88–123.

[54] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits, "Fully abstract compilation to JavaScript", in *ACM SIGPLAN Notices*, ACM, vol. 48, 2013, pp. 371–384.

[55] N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P. Strub, and G. M. Bierman, "Gradual typing embedded securely in JavaScript", in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2014, pp. 425–438.

[56] C. Watt, "Mechanising and verifying the Webassembly specification", in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ACM, 2018, pp. 53–65.

[57] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations", in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, T. Holz and S. Savage, Eds., USENIX Association, 2016, pp. 53–70.

[58] G. Barthe, B. Grégoire, and V. Laporte, "Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time"", in *IEEE Computer Security Foundations Symposium (CSF)*, 2018, pp. 328–343.

[59] J. Renner, S. Cauligi, and D. Stefan, "Constant-time WebAssembly", 2018, `https://cseweb.ucsd.edu/~dstefan/pubs/renner:2018:ct-wasm.pdf`.

[60] B. Beurdouche, “Formal verification for high assurance security software in F*”, Ph.D. dissertation, Paris Science et Lettres, 2020.

[61] B. Blanchet, “Cryptoverif: computationally sound mechanized prover for cryptographic protocols”, in *Dagstuhl seminar “Formal Protocol Verification Applied*, vol. 117, 2007, p. 156.

3. hacspec: High-Assurance Cryptographic Specifications

Contents

Abstract

The verified cryptography developments of Chapter 2, while mechanized inside the F* proof assistant, still suffer from the specification problem detailed in Section 1.2.1. Indeed, cryptographers usually write reference implementations of the primitives and protocols they design in low-level languages like C, making it hard to retro-engineer a high-level specification.

Recently, the cryptographers have begun moving to Rust, and the codebase rewrites this change triggered made a good occasion for nudging cryptographers into writing higher-level specifications. In this chapter, we propose a domain-specific language, hacspec, embedded inside Rust. We claim that hacspec is the right tool for cryptographers to write succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust. Moreover, it helps bridging the gap between formalizers and domain experts.

Parler avec lui, surtout en ligne, vous apprenait au moins deux choses : compacité et anticipation. Laconique à l'extrême, il l'était, par goût, par ascèse aussi. C'était son élégance de codeur : entre deux formules, chercher l'optimale : celle qui contenait la plus forte quantité d'infos en utilisant le minimum de caractères.

(Alain Damasio, Les furtifs*, 2019)*

Turing is neither a mortal nor a god. He is Antaeus. That he bridges the mathematical and physical worlds is his strength and his weakness.

(Neal Stephenson, Cryptonomicon*, 1999)*

This chapter is based upon the following publication:

D. Merigoux, F. Kiefer, and K. Bhargavan, "Hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust", Inria, Technical Report, Mar. 2021. [Online]. Available: https://hal.inria.fr/hal-03176482

My personal contribution to this publication has been the design, formalization and implementation of the hacspec domain-specific language.

3.1. Motivating a New Domain-specific Language

Modern Web applications use sophisticated cryptographic constructions and protocols to protect sensitive user data that may be sent over the wire or stored at rest. However, the additional design complexity and performance cost of cryptography is only justified if it is implemented and used correctly. To prevent common software bugs like buffer overflows [2] without compromising on performance, developers of security-oriented applications, like the Zcash and Libra blockchains, are increasingly turning to strongly typed languages like Rust. However, these type systems cannot prevent deeper security flaws. Any side-channel leak [3] or mathematical bug [4] in the cryptographic library, any parsing bug [5] or state machine flaw [6] in the protocol code, or any misused cryptographic API [7] may allow an attacker to steal sensitive user data, bypassing all the cryptographic protections.

The problem is that these kinds of deep bugs often appear only in rarely-used corner-cases that are hard to find by random testing, but can be easily exploited by attackers who know of their existence. Furthermore, since cryptographic computations often constitute a performance bottleneck in high-speed network implementations, the code for these security-critical components is typically written in low-level C and assembly and makes use of subtle mathematical optimizations, making it hard to audit and test for developers who are not domain experts.

3.1.1. Bridging Rust and Verified Cryptography

In recent years, formal methods for software verification have emerged as effective tools for systematically preventing entire classes of bugs in crypto-

graphic software (see [8] for a survey). For example, verification frameworks like F⋆ [9], EasyCrypt [10], and Coq [11] are used to verify high-performance cryptographic code written in C [12], [13] and assembly [14], [15]. Cryptographic analysis tools like ProVerif [16] and CryptoVerif [17] are used to verify protocols for security properties against sophisticated attackers [18], [19]. Languages like F⋆ have been used to write verified parsers [20] and protocol code [21], [22].

By focusing on specific security-critical components, these tools have been able to make practical contributions towards increasing the assurance of widely-used cryptographic software. For example, verified C code from the HACL⋆ cryptographic library [12] is currently deployed within the Firefox web browser and Linux kernel.

Conversely, the use of domain-specific tools also has its drawbacks. Each verification tool has its own formal specification language tailored for a particular class of analysis techniques. This fragmentation means that we cannot easily compose a component verified against an F⋆ specification with another verified using EasyCrypt. More worryingly, these formal languages are unfamiliar to developers, which can lead to misunderstandings of the verified guarantees and the unintended misuse of the components. For example, an application developer who incorrectly assumes that an elliptic curve implementation validates the peer's public key may decide to skip this check and become vulnerable to an attack [23].

A Better Way to Safely Integrate Verified Code Without sufficient care, the interface between unverified application code and verified components can become a point of vulnerability. Applications may misuse verified components or compose them incorrectly. Even worse, a memory safety bug or a side-channel leak in unverified application code may reveal cryptographic secrets to the adversary, even if all the cryptographic code is formally verified. A classic example is HeartBleed [5], a memory safety bug in the OpenSSL implementation of an obscure protocol feature, which allowed remote attackers to learn the private keys for thousands of web servers. A similar bug anywhere in the millions of lines of C++ code in the Firefox web browser would invalidate all the verification guarantees of the embedded HACL⋆ cryptographic code.

To address these concerns, we propose a hybrid framework (depicted in Figure 3.1) that allows programmers to safely integrate application code written in a strongly typed programming language with clearly specified

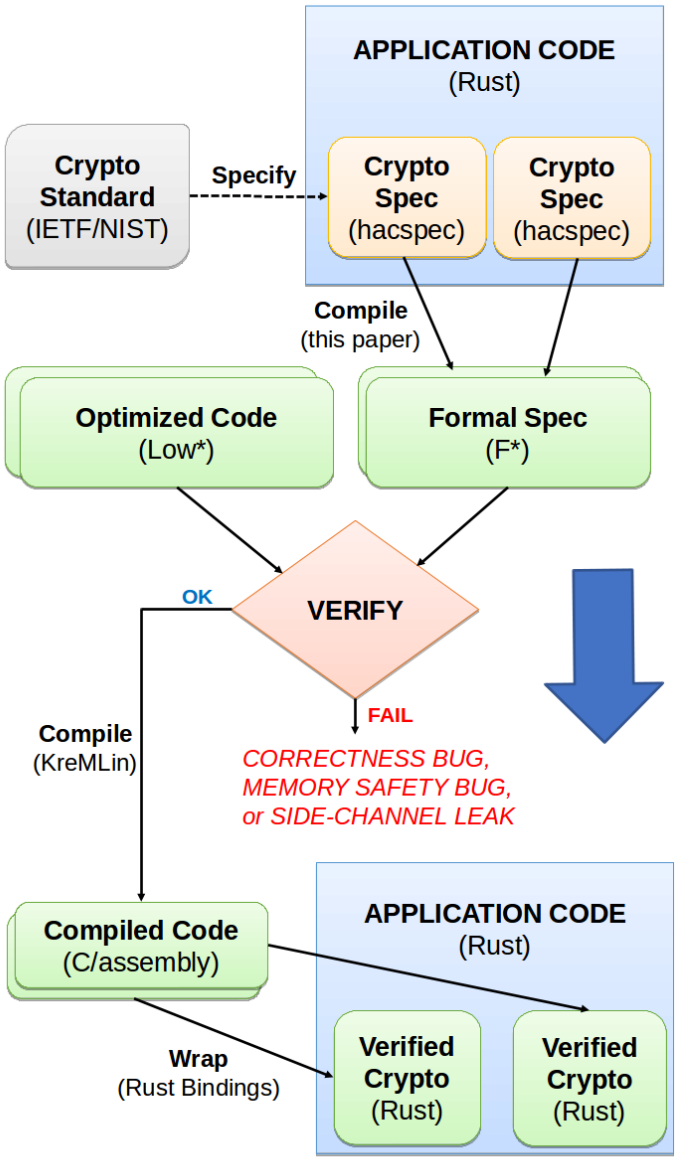

Figure 3.1: **hacspec programming and verification workflow.** The Rust programmer writes executable specifications for cryptographic components in hacspec and compiles them to formal specifications (in F*). The proof engineer implements the cryptographic components (in Low*) and proves that they meet their specifications. The verified code is compiled to high-performance C or assembly code, which is finally wrapped within Rust modules (using foreign function interfaces) that can safely replace the original hacspec specifications.

security-critical components that are verified using domain-specific tools. We choose Rust as the application programming, and rely on the Rust type system to guarantee memory safety. In addition, we provide a library of secret integers that enforces a *secret independent* coding discipline to eliminate source-level timing side-channels. Hence, well-typed code that only uses secret integers does not break the security guarantees of embedded cryptographic components. Of course, this claim only works if the application does not include unverified code written in a low-level, not memory-safe language like C (or unsafe Rust).

hacspec: Cryptographic Specifications In Rust At the heart of our framework is hacspec, a specification language for cryptographic components with several notable features: (1) Syntactically, hacspec is a subset of Rust, and hence is familiar to developers, who can use the standard Rust development toolchain to read and write specifications of cryptographic algorithms and constructions. (2) Specifications written in hacspec are executable and so they can be tested for correctness and interoperability, and they can be used as prototype implementations of cryptography when testing the rest of the application. (3) The hacspec library includes high-level abstractions for commonly used mathematical constructs like prime fields, modular arithmetic, and arrays, allowing the developer to write succint specifications that correspond closely with the pseudocode descriptions of cryptographic algorithms in published standards. (4) hacspec is a purely functional language without side-effects, equipped with a clean formal semantics that makes specifications easy to reason about and easy to translate to other formal languages like F*.

These features set hacspec apart from other crypto-oriented specification languages. The closest prior work that inspired the design of hacspec is hacspec-python [24], a cryptographic specification language embedded in Python that could also be compiled to languages like F*. However, unlike Rust, Python is typically not used to build cryptographic software, so specifications written in hacspec-python stand apart from the normal developer workflow and serve more as documentation than as useful software components. Furthermore, since Python is untyped, hacspec-python relies on a custom type-checker, but building and maintaining a Python type-checker that provides intuitive error messages is a challenging engineering task. Because of these usability challenges, hacspec-python has fallen into disuse, but we believe that our approach of integrating hacspec specifications into

the normal Rust development workflow offers greater concrete benefits to application developers, which increases its chances of adoption.

Contributions and Outline We propose a new framework for safely integrating verified cryptographic components into Rust applications. As depicted in Figure 3.1, the developer starts with a hacspec specification of a cryptographic component that serves as a prototype implementation for testing. Then, via a series of compilation and verification steps, we obtain a Rust module that meets the hacspec specification and can be used to replace the hacspec module before the application is deployed. Hence, the programmer can incrementally swap in verified components while retaining full control over the specifications of each component.

Our main contribution is hacspec, a new specification language for security-critical components that seeks to be accessible to Rust programmers, cryptographers, and verification experts. We present the formal syntax and semantics of hacspec, which is the first formalization of a purely functional subset of Rust, to our knowledge. We demonstrate the use of hacspec on a series of popular cryptographic algorithms, but we believe that hacspec can be used more generally to write functional specifications for Rust code. Our second contribution is a set of tools for hacspec, including a compiler from hacspec to F* that enable the safe integration of verified C and assembly code HACL* in Rust applications. Our third contribution is a set of libraries that any Rust application may use independent of hacspec, including a secret integer library that enforces a constant-time coding discipline, and Rust bindings for the full EverCrypt cryptographic provider [25]. The source code for all these contributions can be found on GitHub[1].

The main technical limitation of our work is that not all the steps are formally verified. To use verified C code from HACL* for example, the programmer needs to trust the compiler from hacspec to F*, the Rust binding code that calls the C code from Rust, and the Rust and C compilers. We carefully document each of these steps and intend to formalize and verify some of these elements in the future. In this work, however, we focus on building a pragmatic toolchain that solves a pressing software engineering problem in real-world cryptographic software.

First, we discuss related work in section Section 3.1.2. Section Section 3.2.1 starts by presenting the hacspec language and its implementation. Then, we

[1] `https://github/hacspec/hacspec`

introduce our security-oriented Rust libraries in section Section 3.2.2, that can be used in conjunction with the hacspec language or in a standalone context. Section Section 3.3.1 deals with the connection between the Rust-based hacspec tooling and multiple verification frameworks having a track record in cryptographic proofs. Finally, we evaluate the language on high-assurance cryptographic primitives in Section 3.3.2.

3.1.2. The Emerging Rust Verification Scene

hacspec is not the first domain-specific language targeting cryptographic specifications. The most notable works in this domain include Cryptol [26], Jasmin [15], and Usuba [27]. There are two main differences between hacspec and these languages.

The first difference is the embedded nature of hacspec's language. By leveraging the existing Rust ecosystem, we believe it is easier for programmers to use our language, compared to the effort of learning a new domain-specific language with its different syntax. We extend this claim to the tooling of the domain-specific language, which is written completely in the host language (Rust) and therefore does not require installing any extra dependency with whom the developer might be unfamiliar (like an entire OCaml or Haskell stack).

The second difference is the target of the domain-specific language. Cryptol targets C and VHDL, Jasmin targets assembly and Usuba targets C. Cryptol and Jasmin each are closely integrated with their respective proof assistants/verification backend: SAW [28] and Easycrypt [10]. The code written in those domain-specific languages is closer to an implementation than a specification, as it is directly compiled to a performant target and is sometimes proven correct against a more high-level specification (like libjc [29]). Instead, hacspec can target different verification toolchains and acts as a bridge between those projects.

Rust-based Verification Tools hacspec is not the first attempt at a Rust frontend for verification toolchains; we provide a summary of existing work in Figure 3.2. The "input" column of the comparison table refers to the entry point of the verification frameworks withing the Rust compiler architecture. The unusual choice of hacspec for AST will be discussed later in Section 3.2.1.

We also classify the previous work according to the extent of formalization

Frontend	Target(s)	Input	Formal
KLEE [30]	KLEE [31]	MIR	○
crux-mir [32]	SAW [28]	MIR	○
Prusti [33]	Viper [34]	MIR	◑
Electrolysis [35]	Lean [36]	MIR	◑
SMACK [37]	SMACK [38]	LLVM IR	◑
RustHorn [39]	CHC [40]	MIR	●
μMIR [41]	WhyML [42]	MIR	●
hacspec	F* [9], EasyCrypt* [10], Coq* [11]	AST	◐

○ = DSL and translation to target not formalized
◑ = DSL defined by its translation to formalized target
◐ = DSL formalized but not the translation to target(s)
● = DSL and translation to target formalized

Figure 3.2: Rust frontends for verification toolchains.
* These backends are experimental, see Section 3.3.1.

that they contain. Indeed, multiple things can be formalized when creating an embedded domain-specific language. First, the domain-specific language can be defined as a subset of a formalization of the host language. In the case of Rust, only RustBelt [43] currently provides a full formalization. But no existing tool that uses RustBelt can extract to another target. On the other hand, the domain-specific language can be defined intrinsically in terms of its encoding in the formalized target (◑). Finally, the domain-specific language itself can be formalized (◐), as well as its translation to the formalized target (●).

We intend for hacspec to belong to the last category, corresponding to the ● case. However, we chose to prioritize the interoperability of hacspec by targeting multiple backends, which increases the workload of translation formalization. Hence, we leave the migration from ◐ to ● as future work.

The main difference of hacspec compared to previous Rust frontend is the scope of the subset it intends to capture. Indeed, hacspec does not deal with memory manipulations and mutable borrows, which is the heart of Rust. On the contrary, hacspec explicitly forbids these as its aim is to capture the functional, pure subset of Rust. Of course, this makes it unsuitable for verifying any kind of performance-oriented programs. Instead, we believe such programs should be dealt by writing this optimized implementation inside an adapted DSL like Jasmin or Low*, and then prove that implementation functionally correct to a specification derived from a hacspec program trans-

lated to the relevant proof assistant (Figure 3.1). hacspec provides to Rust programmers an entry point into the verification world, inside the ecosystem that they are familiar with.

3.2. The hacspec Embedded Language

hacspec is a domain-specific language embedded in the Rust programming language and targeted towards cryptographic specifications. It serves several purposes. Firstly, it acts as a frontend for verification toolchains. We provide a formal description of the syntax, semantics and type system of hacspec as a reference. Secondly, hacspec aims to be a shared language that can foster communication between cryptographers, Rust programmers and proof engineers.

As a motivating example, consider the ChaCha20 encryption algorithm standardized in RFC 8439 [44]. The RFC includes pseudocode for the ChaCha20 block function in 20 lines of informal syntax (see Figure 3.4). However, this pseudocode is not executable and hence cannot be tested for bugs. Indeed, an earlier version of this RFC has several errors in pseudocode. Furthermore, pseudocode lacks a formal semantics and cannot be used as a formal specification for software verification tools.

Figure 3.3 shows code for the same function written in hacspec. It has 23 lines of code, and matches the RFC pseudocode almost line-by-line. The code is concise and high-level, but at the same time is well-typed Rust code that can be executed and debugged with standard programming tools. Finally, this code has a well-defined formal semantics and can be seen as a reference for formal verification.

We believe that hacspec programs straddle the fine line between pseudocode, formal specification, and prototype implementation and are useful both as documentation and as software artifacts. In this section, we detail the syntax, semantics and type system of hacspec, show how we embed it in Rust, and describe our main design decisions.

3.2.1. Syntax, semantics, typing

hacspec is a typed subset of Rust, and hence all hacspec programs are valid Rust programs. However, the expressiveness of hacspec is deliberately limited, compared to the full Rust language. We believe that a side-effect-free purely-

```
fn inner_block(state: State) -> State {
    let state = quarter_round(0, 4, 8, 12, state);
    let state = quarter_round(1, 5, 9, 13, state);
    let state = quarter_round(2, 6, 10, 14, state);
    let state = quarter_round(3, 7, 11, 15, state);
    let state = quarter_round(0, 5, 10, 15, state);
    let state = quarter_round(1, 6, 11, 12, state);
    let state = quarter_round(2, 7, 8, 13, state);
    quarter_round(3, 4, 9, 14, state)
}

fn block(key: Key, ctr: U32, iv: IV) -> StateBytes {
    let mut state = State::from_seq(&constants_init()
        .concat(&key_to_u32s(key))
        .concat(&ctr_to_seq(ctr))
        .concat(&iv_to_u32s(iv)));
    let mut working_state = state;
    for _i in 0..10 {
        working_state = chacha_double_round(state);
    }
    state = state + working_state;
    state_to_bytes(state)
}
```

Figure 3.3: hacspec's version of Chacha20 Block

```
inner_block (state):
   Qround(state, 0, 4, 8,12)
   Qround(state, 1, 5, 9,13)
   Qround(state, 2, 6,10,14)
   Qround(state, 3, 7,11,15)
   Qround(state, 0, 5,10,15)
   Qround(state, 1, 6,11,12)
   Qround(state, 2, 7, 8,13)
   Qround(state, 3, 4, 9,14)
   end

chacha20_block(key, counter, nonce):
   state = constants | key | counter | nonce
   working_state = state
   for i=1 upto 10
      inner_block(working_state)
      end
   state += working_state
   return serialize(state)
   end
```

Figure 3.4: The ChaCha20 Block Function in Pseudocode (RFC7532, 2.3.1)

functional style is best suited for concise and understandable specifications. Extensive use of mutable state, as in C and Rust programs, obscures the flow of data and forces programmers to think about memory allocation and state invariants, which detracts from the goal of writing "obviously correct" specifications. Hence, we restrict hacspec to forbid mutable borrows, and we limit immutable borrows to function arguments.

The usual way of writing side-effect-free code Rust is to use `.clone()` to duplicate values. Figuring out where to insert `.clone()` calls is notably difficult for new Rust users – in spite of good quality Rust compiler error messages. We intentionally strived to reduce the need for `.clone()` calls in hacspec. We leverage the `Copy` trait for all the values that can be represented by an array of machine integers whose length is known at compile-time. This holds true for all kinds of machine integers, but also for the value types defined in the libraries (later discussed in Section 3.2.2).

hacspec benefits from the strong type system of Rust both to avoid specification bugs and to cleanly separate logically different values using types. For instance, separately declared array types like `Key` and `StateBytes` are disjoint, which forces the user to explicitly cast between them and avoids, for instance, the inadvertent mixing of cryptographic keys with internal state.

We describe in detail a simplified version of hacspec. The main simplification lies in the values of the language. We present our formalization with a dummy integer type, but the full hacspec language features all kinds of machine integers, as well as modular natural integers. The manipulation of these other values, detailed in Section 3.2.2, does not involve new syntax or unusual semantics rules, so we omit them from our presentation. Essentially, the rules for binary and unary operators over each kind of machine and natural integers are similar to the rules for our dummy integers; all other operators are modeled as functions and therefore governed by the standard rules about functions.

Syntax The syntax of hacspec (Figure 3.5) is a strict subset of Rust's surface syntax, with all the standard control flow operators, values, and functions. However, hacspec source files are also expected to import a standard library that defines several macros like `array!`. These macros add abstractions like arrays an natural integers to hacspec. The other notable syntactic feature of hacspec is the restriction on borrowing: hacspec only allows immutable borrowings in function arguments. This is the key mechanism by which we

are able to greatly simplify the general semantics of safe Rust, compared to existing work like Oxide [45]. The attentive reader might have noticed an apparent contradiction with our earlier claim that hacspec does not deal with mutable state, since the syntax of Figure 3.5 includes `mut` let-bindings declaring variables that can be reassigned later in the code. Indeed, mutable variable constitute a form of mutable state, and hacspec is not completely side-effect-free. The operational semantics of the language presented later includes a context storing the values of each mutable variable. However, this very restricted form of mutable state can be encoded as a instance of a state monad in a completely side-effect-free language, as Section 3.3.1 will show. Hence, we justify our claim of functional pureness for hacspec despite the looks of `mut`ability.

Another tricky point of the syntax concern the array types in hacspec. In Rust, the go-to type for a collection of elements is `Vec`. hacspec does not feature this type, nor the native Rust array types `[T, n]`. Instead, hacspec offers two different kinds of arrays whose size is constant and fixed at creation time: `Seq` and `array!`. `Seq` is designed for arrays whose length is not statically known at compilation; usually messages (plaintext or ciphered) passed around in cryptographic specifications. Specialized array types of fixed lengths created with `array!` are also useful for cryptography, where specifications define a number of "block" arrays that always have the same size. Defining a special types for those arrays help prevent the mixing of, for instance, a key byte array with a hash digest byte array.

Semantics The structured operational semantics for hacspec corresponds to a simple first-order, imperative, call-by-value programming language. To demonstrate the simplicity of these semantics, we will present them in full here.

The first list of Figure 3.6 presents the values of the language: booleans, integers, arrays and tuples. The evaluation context is an unordered map from variable identifiers to values. Here are the different evaluation judgments that we will present:

The second list of Figure 3.6 shows the evaluation judgments for the various syntactic kinds of hacspec. The big-step evaluation judgment for expressions $p; \Omega \vdash e \Downarrow v$, reads as: "in program p (containing the function bodies to evaluate function calls) and evaluation context Ω, the expression e evaluates to value v". The other evaluation judgments read in a similar way. The last evaluation

p	::=	$[i]^*$	program items
i	::=	`array!` `(` t`,` μ`,` $n \in \mathbb{N}$ `)`	array type declaration
	|	`fn` f `(` $[d]^+$ `)` `->` μ b	function declaration
d	::=	x `:` τ	function argument
μ	::=	`unit` | `bool` | `int`	base types
	|	`Seq<` μ `>`	sequence
	|	t	type variable
	|	`(` $[\mu]^+$ `)`	tuple
τ	::=	μ	plain type
	|	`&`μ	immutable reference
b	::=	`{` $[s;]^+$ `}`	block
s	::=	`let` x `:` τ `=` e	let binding
	|	x `=` e	variable reassignment
	|	`if` e `then` b (`else` b)	conditional statements
	|	`for` x `in` e `..` e b	for loop (integers only)
	|	x`[` e `]` `=` e	array update
	|	e	return expression
	|	b	statement block
e	::=	`()` | `true` | `false`	unit and boolean literals
	|	$n \in \mathbb{N}$	integer literal
	|	x	variable
	|	f `(` $[a]^+$ `)`	function call
	|	$e \odot e$	binary operations
	|	$\oslash\ e$	unary operations
	|	`(` $[e]^+$ `)`	tuple constructor
	|	e`.`$(n \in \mathbb{N})$	tuple field access
	|	x`[` e `]`	array or seq index
a	::=	e	linear argument
	|	`&`e	call-site borrowing
$\odot$	::=	`+` | `-` | `*`	
	|	`/` | `&&` | `||`	
	|	`==` | `!=` | `>` | `<`	
$\oslash$	::=	`-` | `~`	

Figure 3.5: Syntax of hacspec

Value	v	::=	$() \mid \texttt{true} \mid \texttt{false}$
		$\mid$	$n \in \mathbb{Z}$
		$\mid$	$[\ [v]^*\ ]$
		$\mid$	$(\ [v]^*\ )$
Evaluation context	Ω	::=	$\varnothing$
(unordered map)		$\mid$	$x \mapsto v, \Omega$

Expression evaluation	$p; \Omega \vdash e \Downarrow v$
Function argument evaluation	$p; \Omega \vdash a \Downarrow v$
Statement evaluation	$p; \Omega \vdash s \Downarrow v \Rrightarrow \Omega$
Block evaluation	$p; \Omega \vdash b \Downarrow v \Rrightarrow \Omega$
Function evaluation	$p \vdash f\ (\ v_1, \ldots, v_n\ ) \Downarrow v$

Figure 3.6: Values and evaluation judgments of hacspec

judgment is the top-level function evaluation, which is meant as the entry point of the evaluation of a hacspec program.

First, let us examine the simplest rules for values and variable evaluation:

$$\textsc{EvalUnit}\ \frac{}{p; \Omega \vdash () \Downarrow ()} \qquad \textsc{EvalBool}\ \frac{b \in \{\texttt{true}, \texttt{false}\}}{p; \Omega \vdash b \Downarrow b} \qquad \textsc{EvalInt}\ \frac{n \in \mathbb{Z}}{p; \Omega \vdash n \Downarrow n} \qquad \textsc{EvalVar}\ \frac{x \mapsto v \in \Omega}{p; \Omega \vdash x \Downarrow v}$$

We can now move to the rules for function calls evaluation. As shown in the syntax, some immutable borrowing is authorized for function calls arguments. This borrowing is basically transparent for our evaluation semantics, as the following rules show:

$$\textsc{EvalFuncArg}\ \frac{p; \Omega \vdash e \Downarrow v}{p; \Omega \vdash e \Downarrow v} \qquad \textsc{EvalBorrowedFuncArg}\ \frac{p; \Omega \vdash e \Downarrow v}{p; \Omega \vdash \&e \Downarrow v}$$

All values inside the evaluation context Ω are assumed to be duplicable at will. Of course, an interpreter following these rules will be considerably slower compared to the original Rust memory sharing discipline, because it will have to copy a lot of values around. But we argue that this simpler evaluation semantics yields the same results as the original Rust, for our very restricted subset.

We can now proceed to the function call evaluation rule, which looks up the program p for the body of the function called.

$$\textsc{EvalFuncCall}\quad \frac{\texttt{fn}\ f\ (\ x_1\ :\ \tau_1, \ldots, x_n\ :\ \tau_n\ )\ \texttt{->}\ \mu\ b \in p \qquad \forall i \in [\![1, n]\!],\ p; \Omega \vdash a_i \Downarrow v_i \qquad p; x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \vdash b \Downarrow v}{p; \Omega \vdash f\ (\ a_1, \ldots, a_n\ ) \Downarrow v}$$

Next, the evaluation rules for binary and unary operators. These rules are completely standard and assume that the operator has been formally defined on the values of hacspec it can operate on. The dummy arithmetic operators that we have defined in our syntax are assumed to operate only on integers and have the usual integer arithmetic behavior.

$$\textsc{EvalBinaryOp}\quad \frac{p; \Omega \vdash e_1 \Downarrow v_1 \qquad p; \Omega \vdash e_2 \Downarrow v_2}{p; \Omega \vdash e_1 \odot e_2 \Downarrow v_1 \odot v_2} \qquad \textsc{EvalUnaryOp}\quad \frac{p; \Omega \vdash e \Downarrow v}{p; \Omega \vdash \oslash e \Downarrow \oslash v}$$

The rules governing tuples are also very standard. Here, we chose to include only tuple access $e.n$ in our semantics but one can derive similar rules for a tuple destructuring of the form $\texttt{let}\ (\ x_1, \ldots, x_n\ )\ :\ \tau = e$ (which can also be viewed as a syntactic sugar for multiple tuple accesses).

$$\textsc{EvalTuple}\quad \frac{\forall i \in [\![1, n]\!],\ p; \Omega \vdash e_i \Downarrow v_i}{p; \Omega \vdash (\ e_1, \ldots, e_n\ ) \Downarrow (\ v_1, \ldots, v_n\ )}$$

$$\textsc{EvalTupleAccess}\quad \frac{p; \Omega \vdash e \Downarrow (\ v_1, \ldots, v_m\ ) \qquad n \in [\![1, m]\!]}{p; \Omega \vdash e.n \Downarrow v_n}$$

Array accesses are handled similarly to tuple accesses. Note that while the Rust syntax for array access is only a syntactic sugar that calls the `.index()` function of the `Index` trait, we view it as a primitive of the language. By giving a dedicated evaluation rule to this construct, we are able to hide the immutable borrowing performed by the `.index()` function.

$$\textsc{EvalArrayAccess}\quad \frac{p; \Omega \vdash x \Downarrow [\ v_0, \ldots, v_m\ ] \qquad p; \Omega \vdash e \Downarrow n \qquad n \in [\![0, m]\!]}{p; \Omega \vdash x\ [\ e\ ] \Downarrow v_n}$$

We have completely described the evaluation of expressions, let us now move to statements. The next two rules are similar but correspond to two very different variable assignments. The first rule is a traditional, expression-based let binding that creates a new scope for the variable x. The second rule deals with variable reassignment: x has to be created first with a let-binding before reassigning it. We omit here the difference that Rust does between immutable and mutable variables (`mut`). Rust has an immutable-by-default policy that helps programmers better spot where they incorporate mutable state, but here we just assume that all variables are mutable for the sake of simplicity. As mentioned earlier, this apparent `mut`ability does not contradict our claim of functional pureness for hacspec since all this mutable state will later be turned into pure code through a state-passing style transformation (Section 3.3.1). Both statements have a unit return type, to match Rust's behavior.

$$\textsc{EvalLet}\quad \frac{x \notin \Omega \qquad p; \Omega \vdash e \Downarrow v}{p; \Omega \vdash \texttt{let}\ x : \tau = e \Downarrow () \Rrightarrow x \mapsto v, \Omega}$$

$$\textsc{EvalReassign}\quad \frac{p; x \mapsto v, \Omega \vdash e \Downarrow v'}{p; x \mapsto v, \Omega \vdash x = e \Downarrow () \Rrightarrow x \mapsto v', \Omega}$$

The rules for conditional statements are standard. We do not currently include inline conditional expressions in our syntax, although they are legal in Rust and compatible with hacspec. This restricts the return type of conditional blocks to unit.

$$\textsc{EvalIfThenTrue}\quad \frac{p; \Omega \vdash e_1 \Downarrow \texttt{true} \qquad p; \Omega \vdash b \Downarrow () \Rrightarrow \Omega'}{p; \Omega \vdash \texttt{if}\ e_1\ b \Downarrow () \Rrightarrow \Omega'}$$

$$\textsc{EvalIfThenFalse}\quad \frac{p; \Omega \vdash e_1 \Downarrow \texttt{false}}{p; \Omega \vdash \texttt{if}\ e_1\ b \Downarrow () \Rrightarrow \Omega}$$

EvalIfThenElseTrue

$$\frac{p; \Omega \vdash e_1 \Downarrow \texttt{true} \qquad p; \Omega \vdash b \Downarrow () \Rrightarrow \Omega'}{p; \Omega \vdash \texttt{if}\ e_1\ b\ \texttt{else}\ b' \Downarrow () \Rrightarrow \Omega'}$$

EvalIfThenElseFalse

$$\frac{p; \Omega \vdash e_1 \Downarrow \texttt{false} \qquad p; \Omega \vdash b' \Downarrow () \Rrightarrow \Omega'}{p; \Omega \vdash \texttt{if}\ e_1\ b\ \texttt{else}\ b' \Downarrow () \Rrightarrow \Omega'}$$

Looping is very restricted in hacspec, since we only allow for loops ranging over an integer index. This restriction is purposeful, since general while loops can be difficult to reason about in proof assistants. One could also easily add a construct looping over each element of an array, which Rust already supports. However, we chose not to include the idiomatic `.iter().map()` calls to avoid closures.

EvalForLoop

$$\frac{\begin{array}{c} p; \Omega \vdash e_1 \Downarrow n \quad p; \Omega \vdash e_2 \Downarrow m \\ \Omega_n = \Omega \quad \forall i \in [\![n, m-1]\!],\ p; x \mapsto i, \Omega_i \vdash b \Downarrow () \Rrightarrow \Omega_{i+1} \end{array}}{p; \Omega \vdash \texttt{for}\ x\ \texttt{in}\ e_1\ \texttt{..}\ e_2\ b \Downarrow () \Rrightarrow \Omega_m}$$

The array update statement semantics are standard. Here, we require that e_1 evaluates to an in-bounds index. We chose to omit the error case where the index falls outside the range of the array, to avoid including a classic propagating error in our semantics. In Rust, the default behavior of out-of-bounds indexing is to raise a `panic!()` that cannot be caught. Like array indexing, array updating is treated by Rust as a syntactic sugar to a `.index_mut()` call, which mutably borrows its argument. We treat this syntactic sugar as a first-class syntactic construct in hacspec to carefully specify the behavior of the underlying mutable borrow of the array.

EvalArrayUpd

$$\frac{p; x \mapsto [\ v_0, \ldots, v_n\ ], \Omega \vdash e_1 \Downarrow m \qquad m \in [\![0, n]\!] \qquad p; x \mapsto [\ v_0, \ldots, v_n\ ], \Omega \vdash e_2 \Downarrow v}{\begin{array}{c} p; x \mapsto [\ v_0, \ldots, v_n\ ], \Omega \vdash x\,[\ e_1\ ] = e_2 \Downarrow () \Rrightarrow \\ x \mapsto [\ v_0, \ldots, v_{m-1}, v, v_{m+1}, \ldots, v_n\ ], \Omega \end{array}}$$

The next two rules replicate the special case of the last statement of a block in Rust. Indeed, the **`return`** keyword in Rust is optional for the last statement

of a function, because the value returned by a function is assumed to be the result of the expression contained in the last statement. In fact, our syntax does not include the `return` keyword at all to avoid control-flow-breaking effects.

$$\textsc{EvalExprStmt}\ \frac{p; \Omega \vdash e \Downarrow v}{p; \Omega \vdash e \Downarrow v \Rrightarrow \Omega}$$

Blocks in Rust are a list of statements, which also act as a scoping unit: a variable defined inside a block cannot escape it. This behavior is captured by the intersection of the two contexts at then end of EvalBlockAsStatement: we keep all the values Ω' that were already defined in Ω.

$$\textsc{EvalBlock}\ \frac{p; \Omega \vdash s_1 \Downarrow () \Rrightarrow \Omega' \qquad p; \Omega' \vdash \{\ s_2; \ldots; s_n\ \} \Downarrow v \Rrightarrow \Omega''}{p; \Omega \vdash \{\ s_1; \ldots; s_n\ \} \Downarrow v \Rrightarrow \Omega''}$$

$$\textsc{EvalBlockOne}\ \frac{p; \Omega \vdash s \Downarrow v \Rrightarrow \Omega'}{p; \Omega \vdash \{\ s\ \} \Downarrow v \Rrightarrow \Omega'}$$

$$\textsc{EvalBlockAsStatement}\ \frac{p; \Omega \vdash b \Downarrow v \Rrightarrow \Omega'}{p; \Omega \vdash b \Downarrow v \Rrightarrow \Omega' \cap \Omega}$$

Finally, we can define the top-level rule that specifies the execution of a function, given the values of its arguments.

$$\textsc{EvalFunc}\ \frac{\texttt{fn}\ f\ (\ x_1 : \tau_1, \ldots, x_n : \tau_n\ )\ \texttt{->}\ \mu\ b \in p \qquad p; x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \vdash b \Downarrow v}{p \vdash f\ (\ v_1, \ldots, v_n\ ) \Downarrow v}$$

Typing While the operational semantics of hacspec are simple, its typing judgment is trickier. This judgment has to replicate Rust's typechecking on our restricted subset, including borrow-checking. The other complicated part of Rust typechecking, trait resolution, is currently out of hacspec's scope, even though some polymorphism could be introduced in future work to hacspec via a limited use of Rust traits.

The typing environnement of hacspec is fairly standard. We need a type dictionary to enforce the named type discipline of Rust that covers the types

declared by the `array!` macro. Please note that the contexts Γ and Δ are considered as unordered maps rather than ordered associative lists. As such, the rules have the relevant elements appear at the end or the beginning of those contexts without loss of generality.

Typing context	Γ	::=	$\varnothing$
(unordered map)		$\mid$	$x : \tau, \Gamma$
		$\mid$	$f : (\ [\tau]^+\ ) \to \mu, \Gamma$
Type dictionary	Δ	::=	$\varnothing$
(unordered map)		$\mid$	$t \to [\ \mu; n \in \mathbb{N}\ ], \Delta$

The restrictions on borrowing lead to severe limitations on how we can manipulate values of linear type in our language, rendering it quite useless at first sight. Indeed, when you receive a reference as a function argument, you can only use it in expressions and perform identity let bindings with it. You cannot store it in memory or in a tuple and pass it around indirectly in your program. This behavior is well-suited for input and output buffers in cryptographic code.

Linearity is at the heart of the Rust type system, and idiomatic Rust code include a number of explicit `.clone()` indicating where we need to duplicate values in the context. However, Rust also introduces an escape hatch from linearity under the form of the `Copy` trait implementation. This trait, that is primitive to the Rust language, is used to distinguish the values that are "cheap" to copy. In hacspec, the `Copy` trait is implemented for all the reference-free μ types except `Seq`, whose size is not known at compilation time (and thus can be arbitrarily large). Hence, `array!` types, enjoy the `Copy` trait and do not need to be cloned, they are passed around as pure functional values. Indeed, because the length of `array!` types is known at compilation time, the code generation backend of Rust (LLVM) can optimize the representation of the array in memory, especially if the size is small.

Implementing the `Copy` trait $\quad \Delta \vdash \tau : \texttt{Copy}$

$$\textsc{CopyUnit}\ \frac{}{\Delta \vdash \texttt{unit} : \texttt{Copy}} \qquad \textsc{CopyBool}\ \frac{}{\Delta \vdash \texttt{bool} : \texttt{Copy}} \qquad \textsc{CopyInt}\ \frac{}{\Delta \vdash \texttt{int} : \texttt{Copy}}$$

$$\textsc{CopyTuple}\ \frac{\Delta \vdash \tau_1 : \texttt{Copy} \quad \cdots \quad \Delta \vdash \tau_n : \texttt{Copy}}{\Delta \vdash (\ \tau_1, \ldots, \tau_n\ ) : \texttt{Copy}} \qquad \textsc{CopyArray}\ \frac{\Delta \vdash \mu : \texttt{Copy}}{t \to [\ \mu; n\ ], \Delta \vdash t : \texttt{Copy}}$$

Because Rust has an affine type system, hacspec also enjoys an affine typing context with associated splitting rules (SplitLinear). Please note that immutable references values can be duplicated freely in the context (SplitDuplicable). During an elaboration phase inside the Rust compiler, the linearity of the type system gets circumvented for `Copy` values with the insertion of `clone()` functions call that perform a copy of the value wherever the linear type system forces a copy of the value to be made. We formalize this behavior here by allowing `Copy` types duplication in the typing context, like immutable references (SplitCopy). Lastly, functions are always duplicable in the context (SplitFunction). In the following, Γ behaves like an unordered map from variables to their types.

Context splitting $\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2$

$$\textsc{SplitEmpty}\ \frac{}{\Delta \vdash \varnothing = \varnothing \circ \varnothing} \qquad \textsc{SplitLinear1}\ \frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2}{\Delta \vdash x : \tau, \Gamma = (\ x : \tau, \Gamma_1\ ) \circ \Gamma_2} \qquad \textsc{SplitLinear2}\ \frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2}{\Delta \vdash x : \tau, \Gamma = \Gamma_1 \circ (\ x : \tau, \Gamma_2\ )}$$

$$\textsc{SplitDuplicable}\ \frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2}{\Delta \vdash x : \&\tau, \Gamma = (\ x : \&\tau, \Gamma_1\ ) \circ (\ x : \&\tau, \Gamma_2\ )} \qquad \textsc{SplitCopy}\ \frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Delta \vdash \tau : \texttt{Copy}}{\Delta \vdash x : \tau, \Gamma = (\ x : \tau, \Gamma_1\ ) \circ (\ x : \tau, \Gamma_2\ )}$$

$$\textsc{SplitFunction}\ \frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2}{\Delta \vdash f : (\ \tau_1, \ldots, \tau_n\ ) \to \mu, \Gamma = (\ f : (\ \tau_1, \ldots, \tau_n\ ) \to \mu, \Gamma_1\ ) \circ (\ f : (\ \tau_1, \ldots, \tau_n\ ) \to \mu, \Gamma_2\ )}$$

We can now proceed to the main typing judgments. TypVarLinear and TypVarDup reflect the variable typing present in the context. TypTupleCons only allows non-reference values inside a tuple, with a linear context split-

```
foo(&(x,x))
      - ^ value used here after move
      |
      value moved here
```

Figure 3.7: Rust error message for function argument borrowing

ting to check each term of the tuple. TypArrayAccess, TypSeqAccess and TypSeqRefAccess specify the array indexing syntax, which is overloaded to work with both `array!`, `Seq` and `&Seq`. This corresponds to the implementing of the `Index` trait in Rust.

The function call rule, TypFuncCall, is the most complex rule of the typing judgment, because it contains the restricted borrowing form allowed in hacspec. First, note that the context is split for typechecking the arguments of the function, because a linear value cannot be used in two arguments.

While TypFunArg and TypFunArgBorrow are transparent, TypFunArgBorrowVar is the trickiest rule to explain. Let us imagine you are calling the function `foo(&(x,x))` where `x` is a non-borrowed, non-copyable value. The Rust compiler will give you the error message of Figure 3.7.

This means that even though the argument of the function is borrowed, the borrowing happens after the typechecking of the borrowed term using regular rules and linearity. However, if you were to typecheck `foo(&x, &x)` Rust's typechecker would not complain because the borrowing directly affects `x`, and not an object of which `x` is a part of.

This unintuitive feature of the type system becomes more regular when looking at a desugared representation of the code like MIR. But in our type system, we have to include several rules like TypFunArgBorrowVar to reflect it. Since we deal with the surface syntax of Rust, these special rules for borrows are necessary. Although hacspec does not currently include structs, we expect special rules to be added to deal with borrowing struct fields. Note that the syntax of hacspec only allows array indexing to be done via $x[\ e\ ]$ instead of the more general $e_1[\ e_2\ ]$, with the objective of keeping rules simple, since indexing implies borrowing in Rust.

The last rules for binary and unary operations are standard.

Value typing	$\Gamma; \Delta \vdash v : \mu$
Expression typing	$\Gamma; \Delta \vdash e : \tau \Rrightarrow \Gamma'$
Function argument typing	$\Gamma; \Delta \vdash a \sim \tau \Rrightarrow \Gamma'$

TypUnit
$$\frac{}{\Gamma; \Delta \vdash () : \texttt{unit}}$$

TypBool
$$\frac{b \in \{\texttt{true}, \texttt{false}\}}{\Gamma; \Delta \vdash b : \texttt{bool}}$$

TypInt
$$\frac{n \in \mathbb{N}}{\Gamma; \Delta \vdash n : \texttt{int}}$$

TypSeqValue
$$\frac{\forall i \in [\![1, n]\!],\ \Gamma; \Delta \vdash v_i : \mu}{\Gamma; \Delta \vdash [\ v_1, \ldots, v_n\ ] : \texttt{Seq<}\ \mu\ \texttt{>}}$$

TypArrayValue
$$\frac{\forall i \in [\![1, n]\!],\ \Gamma; \Delta \vdash v_i : \mu}{\Gamma; \Delta \vdash [\ v_1, \ldots, v_n\ ] : [\ \mu; n\ ]}$$

TypValueAsExpr
$$\frac{\Gamma; \Delta \vdash v : \mu}{\Gamma; \Delta \vdash v : \mu \Rrightarrow \Gamma}$$

TypVarLinear
$$\frac{}{x : \tau, \Gamma; \Delta \vdash x : \tau \Rrightarrow \Gamma}$$

TypVarDup
$$\frac{x : \&\mu \in \Gamma}{\Gamma; \Delta \vdash x : \&\mu \Rrightarrow \Gamma}$$

TypTupleElim
$$\frac{\Gamma; \Delta \vdash e : (\ \mu_1, \ldots, \mu_m\ ) \Rrightarrow \Gamma' \qquad n \in [\![1, m]\!]}{\Gamma; \Delta \vdash e.n : \mu_n \Rrightarrow \Gamma'}$$

TypTupleRefElim
$$\frac{\Gamma; \Delta \vdash e : \&(\ \mu_1, \ldots, \mu_m\ ) \Rrightarrow \Gamma' \qquad n \in [\![1, m]\!]}{\Gamma; \Delta \vdash e.n : \&\mu_n \Rrightarrow \Gamma'}$$

TypTupleIntro
$$\frac{\Delta \vdash \Gamma = \Gamma_1 \circ \cdots \circ \Gamma_n \qquad \forall i \in [\![1, n]\!],\ \Gamma_i; \Delta \vdash e_i : \mu_i \Rrightarrow \Gamma_i' \qquad \Delta \vdash \Gamma' = \Gamma_1' \circ \cdots \circ \Gamma_n'}{\Gamma; \Delta \vdash (\ e_1, \ldots, e_n\ ) : (\ \mu_1, \ldots, \mu_n\ ) \Rrightarrow \Gamma'}$$

TypArrayAccess
$$\frac{t \to [\ \mu; n\ ] \in \Delta \qquad \Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Gamma_1; \Delta \vdash x : t \Rrightarrow \Gamma_1' \qquad \Gamma_2; \Delta \vdash e : \texttt{int} \Rrightarrow \Gamma_2' \qquad \Delta \vdash \Gamma' = \Gamma_1' \circ \Gamma_2'}{\Gamma; \Delta \vdash x[\ e\ ] : \mu \Rrightarrow \Gamma'}$$

TypSeqAccess
$$\frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Gamma_1; \Delta \vdash x : \texttt{Seq<}\ \mu\ \texttt{>} \Rrightarrow \Gamma_1' \qquad \Gamma_2; \Delta \vdash e : \texttt{int} \Rrightarrow \Gamma_2' \qquad \Delta \vdash \Gamma' = \Gamma_1' \circ \Gamma_2'}{\Gamma; \Delta \vdash x[\ e\ ] : \mu \Rrightarrow \Gamma'}$$

TypSeqRefAccess

$$\frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Gamma_1; \Delta \vdash x : \&\texttt{Seq<}\ \mu\ \texttt{>} \Rrightarrow \Gamma_1' \qquad \Gamma_2; \Delta \vdash e : \texttt{int} \Rrightarrow \Gamma_2' \qquad \Delta \vdash \Gamma' = \Gamma_1' \circ \Gamma_2'}{\Gamma; \Delta \vdash x\ [\ e\ ] : \mu \Rrightarrow \Gamma'}$$

TypFuncCall

$$\frac{f : (\ \mu_1, \ldots, \mu_n\ ) \to \tau \in \Gamma \qquad \Delta \vdash \Gamma = \Gamma_1 \circ \cdots \circ \Gamma_n \qquad \forall i \in [\![1, n]\!],\ \Gamma_i; \Delta \vdash a_i \rightsquigarrow \tau_i \Rrightarrow \Gamma_i' \qquad \Delta \vdash \Gamma' = \Gamma_1' \circ \cdots \circ \Gamma_n'}{\Gamma; \Delta \vdash f\ (\ a_1, \ldots, a_n\ ) : \mu \Rrightarrow \Gamma'}$$

TypFunArg

$$\frac{\Gamma; \Delta \vdash e : \mu \Rrightarrow \Gamma'}{\Gamma; \Delta \vdash e \rightsquigarrow \mu \Rrightarrow \Gamma'}$$

TypFunArgBorrow

$$\frac{\Gamma; \Delta \vdash e : \mu \Rrightarrow \Gamma'}{\Gamma; \Delta \vdash \&e \rightsquigarrow \&\mu \Rrightarrow \Gamma'}$$

TypFunArgBorrowVar

$$\frac{\Gamma; \Delta \vdash x : \mu \Rrightarrow \_}{\Gamma; \Delta \vdash \&x \rightsquigarrow \&\mu \Rrightarrow \Gamma}$$

TypBinopInt

$$\frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Gamma_1; \Delta \vdash e_1 : \texttt{int} \Rrightarrow \Gamma_1' \qquad \Gamma_2; \Delta \vdash e_2 : \texttt{int} \Rrightarrow \Gamma_2' \qquad \odot \in \{\ +\ ,\ -\ ,\ *\ ,\ /\ \} \qquad \Delta \vdash \Gamma' = \Gamma_1' \circ \Gamma_2'}{\Gamma; \Delta \vdash e_1\ \odot\ e_2 : \texttt{int} \Rrightarrow \Gamma'}$$

TypBinopBool

$$\frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Gamma_1; \Delta \vdash e_1 : \texttt{bool} \Rrightarrow \Gamma_1' \qquad \Gamma_2; \Delta \vdash e_2 : \texttt{bool} \Rrightarrow \Gamma_2' \qquad \odot \in \{\ \&\&\ ,\ ||\ \} \qquad \Delta \vdash \Gamma' = \Gamma_1' \circ \Gamma_2'}{\Gamma; \Delta \vdash e_1\ \odot\ e_2 : \texttt{bool} \Rrightarrow \Gamma'}$$

TypBinopComp

$$\frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Gamma_1; \Delta \vdash e_1 : \texttt{int} \Rrightarrow \Gamma_1' \qquad \Gamma_2; \Delta \vdash e_2 : \texttt{int} \Rrightarrow \Gamma_2' \qquad \odot \in \{\ ==\ ,\ !=\ ,\ >\ ,\ <\ \} \qquad \Delta \vdash \Gamma' = \Gamma_1' \circ \Gamma_2'}{\Gamma; \Delta \vdash e_1\ \odot\ e_2 : \texttt{bool} \Rrightarrow \Gamma'}$$

TypUnopInt

$$\frac{\Gamma; \Delta \vdash e : \texttt{int} \Rrightarrow \Gamma' \qquad \oslash \in \{\ -\ \}}{\Gamma; \Delta \vdash \oslash\ e : \texttt{int} \Rrightarrow \Gamma'}$$

TypUnopBool

$$\frac{\Gamma; \Delta \vdash e : \texttt{bool} \Rrightarrow \Gamma' \qquad \oslash \in \{\ \sim\ \}}{\Gamma; \Delta \vdash \oslash\ e : \texttt{bool} \Rrightarrow \Gamma'}$$

Let's now move to the statement typing. We've chosen statements here rather than nested expressions because of the Rust behavior of the `if` statement and the `for` loop. A list of statement corresponds to a block, introduced in Rust by $\{\ \cdots\ \}$. The single statement typing judgment produces a new Γ' because of variable definitions inside a block. Single statements also yield back a type, because the last statement of the function is also the return value of the function. All statement type except the last one should be `unit`.

The rule TypLet introduces a new mutable local variable, that can later be reassigned (TypReassign) in the program. In Rust, the `mut` indicates that

the local variable is mutable, in its absence, variable reassignments are prohibited. In this formalization, all variables are mutable for simplification. The main use of mutable local variables is for variables that are mutated inside a `for` loop. Indeed, because `for` loops are restricted to integer range iteration, we cannot express what would normally be a fold without these mutable variables. Because the mutable variables are local to a block, we do not need to formalize a full-fledged heap for the operational semantics. Rather, we will model them as a limited piece of state that gets passed around during execution.

Next, TypArrayAssign and TypSeqAssign define the overloading of the array update syntax that works for both `array!` and `Seq`. Note that TypIfThenElse use the same context Γ for the two branches of the conditional.

Statement typing	$\Gamma; \Delta \vdash s : \tau \Rrightarrow \Gamma'$
Block typing	$\Gamma; \Delta \vdash b : \tau \Rrightarrow \Gamma'$

TypLet

$$\frac{x \notin \Gamma \qquad \Gamma; \Delta \vdash e : \tau \Rrightarrow \Gamma'}{\Gamma; \Delta \vdash \texttt{let}\ x : \tau = e : \texttt{unit} \Rrightarrow \Gamma', x : \tau}$$

TypReassign

$$\frac{x : \tau \in \Gamma \qquad \Gamma; \Delta \vdash e : \tau \Rrightarrow \Gamma'}{\Gamma; \Delta \vdash x = e : \texttt{unit} \Rrightarrow \Gamma', x : \tau}$$

TypArrayAssign

$$\frac{x : t \in \Gamma \qquad t \to [\ \mu; n\ ] \in \Delta \qquad \Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \qquad x : t \in \Gamma_2 \qquad \Gamma_1; \Delta \vdash e_1 : \texttt{int} \Rrightarrow \Gamma_1' \qquad \Gamma_2; \Delta \vdash e_2 : \mu \Rrightarrow \Gamma_2' \qquad \Delta \vdash \Gamma' = \Gamma_1' \circ \Gamma_2'}{\Gamma; \Delta \vdash x\,[\ e_1\ ] = e_2 : \texttt{unit} \Rrightarrow x : t, \Gamma'}$$

TypSeqAssign

$$\frac{x : \texttt{Seq<}\ \mu\ \texttt{>} \in \Gamma \qquad \Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Gamma; \Delta \vdash e_1 : \texttt{int} \Rrightarrow \Gamma_1' \qquad \Gamma; \Delta \vdash e_2 : \mu \Rrightarrow \Gamma_2' \qquad \Delta \vdash \Gamma' = \Gamma_1' \circ \Gamma_2'}{\Gamma; \Delta \vdash x\,[\ e_1\ ] = e_2 : \texttt{unit} \Rrightarrow x : \texttt{Seq<}\ \mu\ \texttt{>}, \Gamma'}$$

TypIfThen

$$\frac{\Gamma; \Delta \vdash e : \texttt{bool} \Rrightarrow \Gamma' \qquad \Gamma'; \Delta \vdash b : \texttt{unit} \Rrightarrow \Gamma''}{\Gamma; \Delta \vdash \texttt{if}\ e\ \texttt{then}\ b : \texttt{unit} \Rrightarrow \Gamma''}$$

TypIfThenElse

$$\frac{\Gamma; \Delta \vdash e : \texttt{bool} \Rrightarrow \Gamma_c \qquad \Gamma_c; \Delta \vdash b : \texttt{unit} \Rrightarrow \Gamma_t \qquad \Gamma_c; \Delta \vdash b' : \texttt{unit} \Rrightarrow \Gamma_f \qquad \Gamma' = \Gamma \cap \Gamma_f \cap \Gamma_t}{\Gamma; \Delta \vdash \texttt{if}\ e\ \texttt{then}\ b\ \texttt{else}\ b' : \texttt{unit} \Rrightarrow \Gamma'}$$

TypForLoop

$$\frac{\Gamma; \Delta \vdash e_1 : \texttt{int} \Rrightarrow \Gamma_1 \quad \Gamma_1; \Delta \vdash e_2 : \texttt{int} \Rrightarrow \Gamma_2 \quad \Gamma_2, x : \texttt{int}; \Delta \vdash b : \texttt{unit} \Rrightarrow \Gamma_b \quad \Gamma_2 \subset \Gamma_b \quad \Gamma' = \Gamma \cap \Gamma_b}{\Gamma; \Delta \vdash \texttt{for}\ x\ \texttt{in}\ e_1\ \texttt{..}\ e_2\ b : \texttt{unit} \Rrightarrow \Gamma'}$$

TypBlock

$$\frac{\Gamma; \Delta \vdash s_1 : \texttt{unit} \Rrightarrow \Gamma' \quad \Gamma'; \Delta \vdash \{\ s_2\texttt{;}\ \ldots\texttt{;}\ s_n\ \} : \tau}{\Gamma; \Delta \vdash \{\ s_1\texttt{;}\ \ldots\texttt{;}\ s_n\ \} : \tau}$$

TypBlockOne

$$\frac{\Gamma; \Delta \vdash s_1 : \tau \Rrightarrow \Gamma'}{\Gamma; \Delta \vdash \{\ s_1\ \} : \tau}$$

TypBlockAsStatement

$$\frac{\Gamma; \Delta \vdash b : \tau}{\Gamma; \Delta \vdash b : \tau \Rrightarrow \Gamma}$$

TypExpToStmt

$$\frac{\Gamma; \Delta \vdash e : \tau}{\Gamma; \Delta \vdash e : \tau \Rrightarrow \Gamma}$$

A hacspec program is a list of items i. Their typing judgment produces both a new Γ and Δ, because an item introduces either a new function or a new named type. Please note, as mentionned before, that the return type of functions is restricted to μ, as returning a reference is forbidden. The TypFnDecl also means that recursion is forbidden in hacspec, since f is not passed in its typing context.

Item typing $\Gamma; \Delta \vdash i \Rrightarrow \Gamma'; \Delta'$

TypArrayDecl

$$\frac{}{\Gamma; \Delta \vdash \texttt{array!}\ (\ t,\ \mu,\ n\ ) \Rrightarrow \Gamma; \Delta, t \to [\ \mu; n\ ]}$$

TypFnDecl

$$\frac{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n; \Delta \vdash b : \mu}{\Gamma; \Delta \vdash \texttt{fn}\ f\ (\ x_1 : \tau_1, \ldots, x_n : \tau_n\ )\ \texttt{->}\ \mu\ b \Rrightarrow \quad \Gamma, f : (\ \tau_1, \ldots, \tau_n\ ) \to \mu; \Delta}$$

Language extensions From this semantic base, we can add classical language extensions following the same pattern. In particular, we added support for algebraic data types (**enum**s) whose reduction and typing rules are standatrd.

A second, more unusual extension is the the `?` operator. Indeed, `?` is the idiomatic Rust way to perform error handling. Since the language does not feature exceptions, errors flow through a `Result<T,U>` parametrized two-cases **enum** where `T` is the type of the `Ok` case, and `U` is the type for the `Err` case. Most Rust functions in idiomatic code then return a `Result` type, indicating that they can fail. But then, composing such functions together requires a lot of boilerplate **match** code, as shown by the `bar_vanilla` function of

```
fn foo(x: bool) -> Result<u32, String> {
  if x {
    Ok(42u32)
  } else {
    Err("wrong argument!".to_string())
  }
}

fn bar_question() -> Result<u64, String> {
  let y = foo(true)?;
  Ok(y as u64 + 1)
}

fn bar_vanilla() -> Result<u64, String> {
  let y = match foo(true) {
    Ok(x) => x,
    Err(s) => return Err(s)
  };
  Ok(y as u64 + 1)
}
```

Figure 3.8: Example of using the `?` operator in Rust. `bar_vanilla` is semantically equivalent to `bar_qestion`.

Figure 3.8. `?` comes in as a syntactic sugar that perform the **`match`**, retrieving the value of the `Ok` case and leaving the enclosing function early with the value of the `Err` case when there is an error. Semantically, `?` can be explained completely with parametrized algebraic datatypes (of which `Result` is an instance) and a semantics addition for early returns inside functions.

3.2.2. Compiler, Libraries and Domain-specific Integration

We implement the syntax and type system presented above in the hacspec typechecker. Programmers need tooling that help them stay within the bounds of the language, and it is precisely the role of the hacspec typechecker, which kicks in after the regular Rust typechecker.

Compiler Architecture The hacspec typechecker is completely implemented in Rust, integrated into the regular Rust ecosystem of `cargo` and the Rust compiler. As such, programmers that already use Rust need not to install complex dependencies to use hacspec. Concretely, the hacspec typechecker uses the `rustc_driver` [2] crate, offering direct access to the Rust compiler API. The Rust compiler is architected as a series of translations between several intermediate representations:

$$\text{AST} \xrightarrow{\text{desugaring}} \text{HIR} \xrightarrow{\text{to CFG}} \text{MIR} \xrightarrow{\text{to LLVM}} \text{LLVM IR}$$

The borrow checking and typechecking are bound to be performed exclusively on MIR, making it the richest and most interesting IR to target as a verification toolchain input. However, MIR's structure is quite far from the original Rust AST. For instance, MIR is control-flow-graph-based (CFG) and its control flow is destructured, making it hard for a deductive verification toolchain to recover the structure needed to host loop invariants. For hacspec, we chose to take the Rust AST as our input, for two reasons.

First, choosing the AST moves the formalization of hacspec closer to the Rust source code, making it easier to understand. Second, it enables a compilation to the target verification toolchains that preserves the overall look of the code, easing the transition from Rust to the verified backends, and the communication between a cryptography expert and a proof engineer. This

[2] `https://doc.rust-lang.org/nightly/nightly-rustc/rustc_driver/`

```
error[hacspec]: type not allowed in hacspec
  --> hacspec-poly1305/src/poly1305.rs:61:17
   |
61 | pub fn poly(m: &[U8], key: KeyPoly) -> Tag {
   |                 ^^^^
```

Figure 3.9: hacspec error message

choice makes hacspec quite different from the other Rust-based verification tools discussed in Section 3.1.2.

Three-tier Typechecking The hacspec typechecker operates in three phases.

In the first phase, the program goes through its regular flow inside the Rust compiler, up to regular Rust typechecking. Second, the Rust surface AST is translated into a smaller AST, matching the formal syntax of hacspec. Third, a typechecking phase following the formal typechecking rules of hacspec is run on this restricted AST.

The second phase is the most helpful for the developers, as it will yield useful error messages whenever the program does not fall withing the hacspec subset of Rust. The error messages look like regular error messages emitted by the Rust compiler, as shown in figure Figure 3.9.

Here, the error message points at the native Rust slice being used, which is forbidden since native Rust slices are not part of hacspec (the corresponding type is `ByteSeq`). These error messages are a key component of the security-oriented aspect of hacspec, since they enforce the subset limits at compile-time rather than at execution time. The goal of this tooling is to make it impossible for the programmer to unknowingly break the abstraction of the subset.

The third phase, corresponding to hacspec's typechecking, should never yield any error since the program has at this point already be type-checked by Rust. If the program had a typing problem, it should have been caught by the Rust typechecker first and never come to this third phase. However, the hacspec typechecker still catches some restrictions of hacspec subset that are not purely syntactic, and therefore pass the second phase.

Interacting With The Rust typechecker The first kind of errors caught by the third phase concerns external function calls. Indeed, one should only call in hacspec functions that are within the bounds of hacspec. But this rule

suffer some exceptions, the main one being the primitive functions of the hacspec library that operate on base types such as `Seq`, whose behavior is part of the trusted computing base. Another issue is the import of functions defined outside of the current crate. The import of the hacspec library is recognized by the hacspec typechecker and given special treatment, but we also allow importing regular crates. This allows hacspec programs to enjoy some kind of modularity, as programs can be distributed over several crates. The hacspec typechecker uses the Rust compiler's "crate metadata reader" to import function definitions from external crates, and scan their signatures. If the signature of an imported function typechecks in hacspec, then its use is valid and does not yield an error. This behavior, while practical, leaves a gap opened for breaking the abstraction of the subset. Indeed, one could import a function whose signature is in hacspec, but its body is not, thereby increasing the trusted computing base. A solution to this problem would be to define an allow-list of valid hacspec functions via Rust's attribute system, but technical constraints on the Rust compiler (custom attributes are erased from crate metadata) makes this solution inoperable. A better system to close this potential abstraction leak is left as future work.

The second kind of errors yielded by the third phase of hacspec program processing relates to type inference. The hacspec typechecking procedure does not support any kind of type inference, unlike regular Rust typechecking. Nevertheless, idiomatic Rust programs often rely on type inference for things like integer literal types, or methods operating on parametrized types. Hence, hacspec forces programmers to explicitly annotate their integer literals with their type, using regular Rust syntax like `1u32`. This issue also arises with methods. hacspec's syntax does not include methods because a method call is the same as calling a function whose first argument is the `self`. However, this assumes that method resolution has already been performed. This is not the case when taking the Rust AST as an input, which means that the hacspec typechecker has to replicate the Rust method resolution algorithm. The algorithm is very simple: it typechecks the first `self` argument, then looks into a map from types to functions if the type contains the correct function being called. This behavior is more complicated with parametric types such as `Seq`, introducing a nested map for the type parameter, but keeps the same principle. Hence, and because of the lack of type inference, the programmer has to explicitly annotate the type parameter of methods concerning parametric types, using regular Rust syntax like `Seq::<U8>::new(16)`.

The hacspec language alone is not sufficient to write meaningful programs.

Indeed, cryptographic and security-oriented programs manipulate a wide range of values that need to be represented in hacspec. To that purpose, we provide several Rust standalone libraries that implement security-related abstractions. Although these libraries are tightly integrated in hacspec, they can be used individually in regular Rust applications.

Secret Integers Timing side-channels are some of the most important threats to cryptographic software, with many CVEs for popular libraries appearing every year. To mitigate these attacks, cryptographic code is usually written to be *secret independent*: the code never branches on a secret value and it never uses a secret index to access an array. This discipline is sometimes called *constant time coding* and many verification tools try to enforce these rules at compile time or in the generated assembly [46], [47].

We follow the approach of HACL⋆ [12] to enforce secret indendence at the source-code level using the standard Rust typechecker. We build a library of secret machine integers that contains wrapped versions of all machine integer types, signed and unsigned, supported by Rust. These wrappers define a new type `U8`, `U32`,... corresponding to each integer type **`u8`**, **`u32`**,... in Rust. To define our secret integer types, we make use of Rust's **`struct`** declaration semantics and nominal typing, which differs from a simple type alias introduced with **`type`**.

For each integer type, the library defines only the operations that are known to be constant-time, *i.e.* operations that are likely to be compiled to constant-time instructions in most assembly architectures. Hence, secret integers support addition, subtraction, multiplication (but not division), shifting and rotation (for public shift values), bitwise operators, conversions to and from big- and little-endian bytearrays, and constant-time masked comparison.

Secret integers can be converted to and from regular Rust integers, but only by calling an explicit operation. The library provides the functions `classify` and `declassify`, to make visible in the code the points where there is an information flow between public and secret. Hence, a Rust program that uses secret integers but never calls `declassify` is guaranteed to be secret independent at the source-code level. We have carefully curated the integer operations to ensure so that the Rust compiler should preserve this guarantee down to assembly, but this is not a formal guarantee, and the generated assembly should still be verified using other tools. However, our methodology provides early feedback to the developer and allows them to eliminate an

entire class of sofware side-channels.

Secret integers are extensively used in hacspec programs to check that our code does not inadvertently leak secrets. However, these libraries can also be used outside hacspec to add protections to any Rust program. The developer simply replaces the types all potentially secret values (keys, messages, internal cryptographic state) from `u8`, `u32`, ... to `U8`, `U32`, ... and uses the Rust typechecker to point out typing failures that may indicate timing leaks. The developer can then selectively add `declassify` in cases that they carefully audit and deem safe. To the best of our knowledge, this is the first static analysis technique that can analyze Rust cryptographic code for secret independence.

Fixed-length Arrays Formal specifications of software components often need to use arrays and sequences, but for clarity of specification and ease of translation to purely functional specifications in languages like Coq and F*, we believe that hacspec programs should only use arrays in simple ways. Any sophisticated use of mutable or extensible arrays can quickly make a specification muddled and bug-prone and its invariants harder to understand. Consequently, in hacspec, we only allow fixed-length arrays that cannot be extended (unlike the variable-size `Vec` type provided by Rust).

The hacspec library includes two kinds of fixed-length arrays. The go-to type is `Seq`, which models a fixed-length sequence of any type. `Seq` supports a large number of operations such as indexing, slicing and chunking. The Rust typechecker ensures that the contents of the `Seq` have the correct type, and C-like array pointer casting is forbidden. This forces the user to properly cast the contents of the array rather than casting the array pointer itself, which can be a source of bugs.

However, `Seq` has a blind spot triggered by a lot of usual cryptographic specifications bugs: array bounds checking. `Seq`, like `Vec`, will trigger a dynamic error if one tries to access an index outside its bounds. This Rust dynamic error is better than the C undefined behavior (usually resulting in a segfault), but is not enough for our security-oriented goals. A full proof that array accesses are always within bounds typically requires the use of a fully-fledged proof assistant, as we'll see in Section 3.3.1. But the hacspec libraries offers a mechanism to bake into Rust's typechecking part of this array bound proof.

This mechanism is offered by the `array!` macro. `array!(State, U32, 16)` is the declaration of a named Rust type that will correspond to an array of

16 secret 32-bit (secret) integers. The length of 16 is baked into Rust's type-system via the underlying use of the `[U32;16]` Rust native array type. Using such a native known-length array type compared to a regular `Seq` has multiple advantages.

First, the `array!(State, U32, 16)` macro call defines all the methods of the `State` type on-the-spot, and uses its length to provide debug assertions and dynamic error messages that help the user with the extra information that `State`'s length should always be 16, whereas a `Seq` can have any length. Moreover, the `State` constructor expects a literal Rust array of 16 integers, whose length can be directly checked by Rust's typechecker.

Second, the use of `array!(State, U32, 16)` acts as a length hint to the verification backends of hacspec. Known-length arrays act as control points in the proof that help inference of array lengths during the array bounds proof.

While the two first advantages help increase the correctness of the hacspec program, they often come as a burden to programmers which have to annotate their program with explicit casts between `Seq` and `array!`-declared types. However, a third advantage of the underlying native known-length array type compensates the annoyance. Indeed, `Seq`, as `Vec`, does not implement the `Copy` trait and therefore has to be explicitly `.clone()` every time it is used multiple times without borrowing. As mutable borrowing is forbidden in hacspec, this would lead to a high-number of `.clone()` calls for `Seq` values mutated and used in the programs. But since `array!` uses Rust's native known-length array that implement `Copy`, their manipulation does not require any explicit `.clone()` call. This feature is especially helpful for cryptographic code, which usually manipulates small-sized chunks (represented using `array!`) coming from a few big messages (represented using `Seq`).

Modular Natural Integers Many cryptographic algorithms rely on mathematical constructs like finite fields, groups, rings, and elliptic curves. Our goal is to provide libraries for all these constructions in hacspec, so that the programmer can specify cryptographic components using high-level concepts, without worrying about how to implement them.

For example, one of the most common needs for cryptographic code is modular arithmetic, that is the field of natural numbers between 0 and n with all arithmetic operations performed modulo n. For example, by setting n to a power of 2, we can build large integer types like `u256`; by setting it to a prime,

```
public_nat_mod!(
    type_name: FieldElement,
    type_of_canvas: FieldCanvas,
    bit_size_of_field: 131,
    modulo_value:
      "03ffffffffffffffffffffffffffffffb"
);
```

Figure 3.10: Declaration of a (public) modular natural number in hacspec

we obtain a prime field; by choosing a product of primes, we get an RSA field etc.

We provide a dedicated library for arbitrary-precision modular natural integers, that can be manipulated by Rust programs just like machine integer values, without worrying about any allocation or deallocation. Figure 3.10 shows what a finite field declaration looks like, taken from the Poly1305 specification. The `public_nat_mod!` macro call defines a fresh Rust type, `FieldElement`, along with multiple methods corresponding to operations on these natural integers. The two next arguments of the macro call concern the underlying representation of the natural integer.

Our implementation of modular arithmetic relies on Rust's `BigInt` crate[3]. But `BigInt` does not implement the `Copy` trait and is therefore cumbersome to use, requiring the insertion of numerous `.clone()` calls. To bypass this limitation, hacspec's modular integers use a concrete representation as a big-endian, fixed-length array of bytes. The length of this array is determined using the `bit_size_of_field` argument of the macro call. The methods of `FieldElement` constantly switch back and forth between the `Copy`able array representation and its `BigInt` equivalent to get the computations right.

The `modulo_value` argument contains the value for the modulus n as a hex string because the value can be arbitrarily large and often cannot fit inside a `u128` literal. The `type_of_canvas: FieldCanvas` argument is required merely because of Rust's macro system fundamental limitation of forcing the user to explicitly provide the identifier for all the types declared by the macro. Indeed, the macro defines two types: `FieldCanvas` is the type for the underlying array-of-bytes representation of the bounded natural integers, that enjoys the `Copy` trait. `FieldElement` is a wrapper around `FieldCanvas` that

[3] https://crates.io/crates/bigint

takes the modulo value into account for all its operations.

hacspec's natural modular integers also come in two versions, public and secret. The secret version can only be converted to arrays of secret integers, ensuring the continuity of information flow checking across machine and natural integers. However, the underlying modular arithmetic arithmetic operations themselves are not constant-time, so this inter-conversion serves primarily to document information flow, not to enforce secret independence.

The seamless interoperability provided by hacspec between machine integers and modular natural integers allows programmers to mix in different styles of specifications, ranging from high-level math-like to low-level implementation-like code. hacspec allows programmers to write and test code at both levels and bridge the gap between them, by allowing them to interoperate, and also though formal verification, as we'll see in Section 3.3.1.

3.3. hacspec as a Proof Frontend

hacspec is a security-oriented domain-specific language embedded in Rust, along with a set of useful libraries for cryptographic specifications. However, as strong as Rust type system is, it is not sufficient to catch common errors like array indexing problems at compile time. To increase the level of assurance on the correctness of specifications written in hacspec, we implement one backends from hacspec to state-of-the art verification frameworks: F* [9] (Section 3.3.1). By using this backend, that could easily be extended to other frameworks like Easycrypt [10] and Coq [11], the hacspec programmer can further debug specifications, find non-trivial errors which may have escaped manual audits, and formally connect the specification to an optimized verified implementation We demonstrate this approach on a library of classic cryptographic primitives (Section 3.3.2).

3.3.1. Translating hacspec to F*

The functional semantics of the hacspec language makes it easy to compile it to the F* language. To illustrate this claim, Figure 3.11 and Figure 3.12 show what the same function, the main loop of the Poly1305 specification, looks like both in hacspec and after its translation to F*.

The translation is very regular, as each hacspec assignment is translated to an F* let-binding. Variable names are suffixed with indexes coming from a

```
1  pub fn poly(m: &ByteSeq, key: KeyPoly) -> Tag {
2    let r = le_bytes_to_num(&key.slice(0, BLOCKSIZE));
3    let r = clamp(r);
4    let s = le_bytes_to_num(
5      &key.slice(BLOCKSIZE, BLOCKSIZE));
6    let s = FieldElement::from_secret_literal(s);
7    let mut a = FieldElement::from_literal(0u128);
8    for i in 0..m.num_chunks(BLOCKSIZE) {
9      let (len, block) =
10       m.get_chunk(BLOCKSIZE, i);
11     let block_uint = le_bytes_to_num(&block);
12     let n = encode(block_uint, len);
13     a = a + n;
14     a = r * a;
15   }
16   poly_finish(a, s)
17 }
```

Figure 3.11: Poly1305 main loop in hacspec

name resolution pass happening in the hacspec typechecker, to ensure that the scoping semantics are preserved by the compilation. Apart from syntax changes, the bulk of this translation relates to the functional purification of the mutable variables involved in loop and conditional statements. This approach is similar to the earlier work of Electrolysis [35], although much simpler because we do not deal with mutable references.

However, hacspec features mutable plain variables which can be reassigned throughout the program. These reassignments are always translated into functional let-bindings, but since statements are translated into expressions during the compilation to F*, the assignment side-effects have to be hoisted upwards in the let-binding corresponding to conditional or loop statements. For example, see the translation of the loop statement lines 8 to 15 of Figure 3.11, to the loop expression lines 15 to 27 of Figure 3.12.

While conditional and loop statements constitute the main structural changes of the translation, most of the compiler implementation work goes into connecting the libraries handling arithmetic and sequences in F*. Fortunately, this task is simplified by having access to the rich typing environment provided by the hacspec typechecker, which allows us to insert annotations and hints into the generated F*, significantly easing the out-of-the box typechecking of the translated programs. For instance, `FieldElement::from_s`

```
let poly (m_15: byte_seq) (key_16: key_poly) : tag =
  let r_17 = le_bytes_to_num
    (array_slice (key_16) (usize 0) (blocksize))
  in
  let r_18 = clamp (r_17) in
  let s_19 = le_bytes_to_num
    (array_slice (key_16) (blocksize) (blocksize))
  in
  let s_20 = nat_from_secret_literal
    (0x03fffffffffffffffffffffffffffffffb) (s_19)
  in
  let a_21 = nat_from_literal
    (0x03fffffffffffffffffffffffffffffffb) (pub_u128 0x0)
  in
  let (a_21) = foldi
    (usize 0) (seq_num_chunks (m_15) (blocksize))
    (fun i_22 (a_21) ->
      let (len_23, block_24) =
        seq_get_chunk (m_15) (blocksize) (i_22)
      in
      let block_uint_25 = le_bytes_to_num (block_24) in
      let n_26 = encode (block_uint_25) (len_23) in
      let a_21 = (a_21) +% (n_26) in
      let a_21 = (r_18) *% (a_21) in
      (a_21))
    (a_21)
  in
  poly_finish (a_21) (s_20)
```

Figure 3.12: Poly1305 main loop in F*, compiled from Figure 3.11

`ecret_literal(s)` (line 6 of Figure 3.11) is translated to the F⋆ expression `nat_from_secret_literal (0x03ff[...]ffb) (s_19)` (lines 9-10 of Figure 3.12). The modulo value of the `FieldElement` integer type has been automatically added during the translation, as a hint to F⋆ typechecking.

Overall, these annotations added automatically during the translation enable smooth typechecking and verification inside F⋆. The only manual proof annotations still needed in F⋆ concern logical pre-conditions and loop invariants that cannot be expressed using the Rust type system.

The Benefits Of Using Theorem Provers Once embedded in F⋆, various properties out of reach of the Rust type system can be proven correct about the specification. The most obvious of these properties is the correctness of array indexing.

RFC 8439 [44] is the second version of the ChaCha20Poly1305 RFC, written after a number of errata were reported on the previous version. However, the second version still contains a specification flaw that illustrates the need for debugging specifications with proof assistants. RFC 8439 defines the core loop in Poly1305 in a way that overruns the message if its length is not a multiple of 16. Figure 3.13 shows the RFC8439 pseudocode, as well as its corresponding snippets in hacspec and F⋆.

In our hacspec code, the `get_chunk` function always provides the correct chunk length, preventing the bug. However, if we tried to precisely mimic the RFC pseudocode, we would have introduced the bug, which would not have been caught by the Rust or hacspec typecheckers. The issue could have been uncovered by careful testing, but we note that the standard test vectors did not prevent the introduction of this bug in the RFC.

We claim that using F⋆ to typecheck specifications can help optimize the specification development workflow by catching this kind of bugs early. The bug of RFC 8439 is indeed detected by F⋆ with a helpful error message. hacspec offers an integrated experience for cryptographic specification development: a cryptographer can write a Rust-embdded hacspec program that looks like pseudocode, and do a first round of debugging with testing since hacspec programs are executable. Then, the cryptographer can translate the specification to F⋆ where a proof engineer can prove array-indexing and other properties correct.

Once the specification has been fixed and proven correct, an optimized implementation is required for the cryptographic primitive or protocol to be

```
for i=1 upto ceil(msg length in bytes / 16)
  n = le_bytes_to_num(
    msg[((i-1)*16)..(i*16)] | [0x01])
  a += n
  a = (r * a) % p
end
```

```
let block_uint = le_bytes_to_num(
  &m.slice(BLOCKSIZE * i, BLOCKSIZE));
```

```
let block_uint_1876 = le_bytes_to_num (
  seq_slice (m_1868)
    ((blocksize) * (i_1875))
    (blocksize))
in
```

```
(Error 19) Subtyping check failed;
expected type len: uint_size{
  blocksize * i_22 + len <= Lib.Seq.length m_15
};
got type uint_size; The SMT solver could not prove
the query, try to spell your proof in more detail
or increase fuel/ifuel
```

Figure 3.13: RFC 8439: first, the main Poly1305 loop pseudocode containing the bug. Then in second and third, corresponding buggy snippets in hacspec and F* (line numbers of Figure 3.11 and Figure 3.12). Fourth: F* error message catching the bug

embedded in a real-world application. There is often a wide gap between the specification and the optimized code, and this spot is where formal methods have proven their usefulness in the past. Proof assistants like F* enable proving the functional equivalence between a specification and an optimized implementation. hacspec fits nicely into this process, since it directly provides the specification in the language of the proof assistant (here F*), based on hacspec code that can be audited by cryptographers.

A Platform To Connect Theorem Provers While proof assistants are powerful, they are often specialized. In the cryptography space, Fiat-crypto uses Coq and generate C implementations for elliptic curves [48], and Jasmin covers more kinds of primitives using EasyCrypt [10] but only targets assembly [15]. HACL* [12] and Vale [14], both using the F* proof assistant, are currently the only instance of proven-correct interoperability between C and assembly [49].

In this context, hacspec is a way to break the integration silo imposed by proof assistant frameworks which typically cannot interoperate. The simplicity of the hacspec language semantics, close to a first-order functional language, makes it easy to translate in the specification languages of most proof assistants. A Coq backend for hacspec was thus written by Mikkel Milo from Aarhus University[4], and we wrote a proof of concept for an Easycrypt backend. The bulk of the work for creating a new backend comes from the port of the hacspec Rust libraries into the target language, since some of them are part of the trusted computing base and are not translated by the compiler. By centralizing the source of truth for cryptographic specifications inside multiple verified developments, hacspec will raise the level of assurance of all of them. Of course, the weakness of a centralized system is the single point of failure, and here a specification bug in the hacspec source will be reflected in all the verified developments that depend on it. This is why hacspec source code reviewing and debugging by domain experts (cryptographers) is crucial.

3.3.2. Evaluation on real-world software

To evaluate the hacspec language, libraries, typechecker and compiler, we build a library of popular cryptographic algorithms in hacspec, presented in Figure 3.14. For each primitive, we wrote a hacspec specification that typechecks in Rust and with the hacspec typechecker. Then, we translated

[4] https://github.com/hacspec/hacspec/pull/123

Primitive / Lines of code	hacspec	HACL⋆
ChaCha20	121	191
Poly1305	107	77
Chacha20Poly1305	52	89
NTRU-Prime	95	–
SHA256	148	219*
SHA3	173	227
P256	172	246
Curve25519	107	124
GF128	74	94
AES	366	425
AES128-GCM	130	–
ECDSA-P256-SHA256	52	60
HMAC	54	42
BLS12-381	540	–
Gimli	256	–
HKDF	56	–

Figure 3.14: Cryptographic Primitives written in hacspec, lines of code count. The HACL⋆ column is included as reference.
* The HACL⋆ SHA2 specification covers all versions of SHA2, not just 256.

some of these primitives to F⋆ and typechecked the result with the proof assistant.

Some annotations (maximum 2-3 per primitive) are needed to typecheck the specifications in the proof assistants. These annotations concern bounds for array indexes passed as function parameters, as well as some loop invariants. In the future, we may extend the Rust-embedded syntax of hacspec to include design-by-contracts annotations. This functionality is already provided by crates like `contracts`[5], where the annotations are incorporated into Rust `debug_assert!` assertions.

These hacspec specifications have been extensively tested and can be used as prototype cryptographic implementations when building and debuging larger Rust applications. Further, they form the basis for our verification workflow.

[5] `https://crates.io/crates/contracts`

Going Beyond Cryptography While cryptography is the main target of the hacspec domain-specific language, early results suggest that it could target other domains that rely on the same language feature. Critical sections of kernel code often fit inside the same language subset as cryptographic code. While kernel code is generally executed inside a parallel computing environment, which is outside the current scope of hacspec, some kernel features have to run sequentially. This sequential setting is true for two case studies from the RIOT [50] open-source: `riot-bootloader` and `riot-runqueue`.

First, `riot-bootloader` is the hacspec reimplementation of RIOT's first interactions with the machine on which it is booted. The bootloader's mission is to scan the memory of the boot partition, looking for headers whose signature matches the parameters expected by RIOT. The corresponding hacspec program takes as input a list of available headers parsed from the memory contents, and determines whether each header is valid (in the sense that it correctly points to a bootable RIOT implementation). The validation process involves a cryptographic-like operation that computes the Fletcher checksum [51] of the header. The whole process can be specified in 110 lines of self-contained hacspec code.

Second, `riot-runqueue` concern a critical component of the operating system: the scheduler, which allocates dynamically which thread to run on the available processing cores. This allocation is decided based on threads' priorities, and how long they have been waiting to run. The implementation of the schedular has to enjoy both low-level properties (memory safety and hard performance constraints) and high-level properties (fairness, absence of deadlocks). This makes the scheduler code a good target for verification. We implemented the core data structure of the RIOT scheduler, a bounded priority runqueue, in hacspec. This case study amounts to 160 lines of code, and fully models the data structure and its operations, based on bitwise manipulation of the contents of two arrays storing the threads and the priority queues. Scheduler verification has been a research area for a long time; notable, the Bossa framework [52] introduces a domain-specific language for scheduling algorithm specifications. Future work should investigate whether hacspec is on par with prior scheduling literature, and examine what contribution it could bring to this ecosystem.

Conclusion

In this chapter, our contribution to program verification in the traditional acceptation of the term is not direct. The hacspec domain-specific language does not have a novel semantics, and its compilation toolchain does not feature novel techniques. However, hacspec completely illustrates the methodology proposed in this dissertation for pushing the frontier of program verification. First, we carve out a formal, executable subset of Rust, a language that cryptographers use to implement their latest creations. Then, we build a proof-oriented toolchain around this domain-specific language, whose goal is twofold. First, fix the specification problem existing in many verified cryptographic developments. Second, provide a gateway for domain-specific experts using Rust to push their developments to various proof assistants, therefore building a bridge with program verification experts.

A lot of work remains to be done on the hacspec language. Adding more backends and useful language extensions will improve the languages' expressivity, but that should not bar the way to a future certification of the compilation steps of this toolchain. Particularly, the question whether hacspec is a faithful susbset of Rust should be proved within the Rustbelt [43] framework; translations to proof asssistants should also be formalized and mechanically certified (at least partially). However, the danger of discovering toolchain bugs can be mitigated by the simplicity and standardness of the language features, and a compiler design that restricts ambiguous programs by default.

Overall, hacspec's philosophy can be viewed as complementary to the traditional philosophy of program verification frameworks. In recent program verification works like this verified extraction framework for SQL-like queriers [53], the domain-specific part of the study is merely anecdotal, and serves as a playground to showcase the strenghts of a general-purpose framework that pushes the frontier of program verification techniques. On the other hand, this dissertation argues for a methodology that starts from the domain, and then looks how the domain could be embedded into existing program verification frameworks. We hope that combining the two approaches will yield a more complete chain of trust.

That problem being fixed, remains the issue of performing functional correctness and security proofs for cryptographic programs. More generally, the current program verification frontier is located on programs that have complex memory manipulation patterns. For instance, the protocol code found

in LibSignal* (Chapter 2) exercises the limits of the Low* domain-specific language, especially concerning its way to reason about memory. The verification of the protocol code required painful proof organization and fine-tuning of the Z3 encoding parameters of F* (see Section 2.1.1). It became apparent that a new approach was necessary for verifying programs that manipulated the memory in more complex ways than buffers of data. The next chapter will focus on these issues that concern cryptographic programs but not only, thus widening our application range.

References

[1] D. Merigoux, F. Kiefer, and K. Bhargavan, "Hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust", Inria, Technical Report, Mar. 2021. [Online]. Available: https://hal.inria.fr/hal-03176482.

[2] OpenSSL, *ChaCha20/Poly1305 heap-buffer-overflow*, CVE-2016-7054, 2016.

[3] N. J. Al Fardan and K. G. Paterson, "Lucky thirteen: breaking the tls and dtls record protocols", in *IEEE Symposium on Security and Privacy*, 2013, pp. 526–540.

[4] B. B. Brumley, M. Barbosa, D. Page, and F. Vercauteren, "Practical realisation and elimination of an ecc-related software bug attack", in *Topics in Cryptology–CT-RSA 2012*, Springer, 2012, pp. 171–186.

[5] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, *et al.*, "The matter of heartbleed", in *Proceedings of the 2014 conference on internet measurement conference*, 2014, pp. 475–488.

[6] 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 *2015 IEEE Symposium on Security and Privacy*, IEEE, 2015, pp. 535–552.

[7] H. Böck, A. Zauner, S. Devlin, J. Somorovsky, and P. Jovanovic, "Nonce-disrespecting adversaries: practical forgery attacks on GCM in TLS", in *USENIX Workshop on Offensive Technologies (WOOT)*, 2016.

[8] 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 (S&P'21)*, 2021.

[9] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, *et al.*, "Dependent types and multi-monadic effects in F*", in *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 256–270.

[10] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P.-Y. Strub, "Easycrypt: a tutorial", in *Foundations of security analysis and design vii*, Springer, 2013, pp. 146–166.

[11] The Coq Development Team, *The Coq Proof Assistant Reference Manual, version 8.7*, Oct. 2017. [Online]. Available: `http://coq.inria.fr`.

[12] 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*, 2017, pp. 1789–1806.

[13] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala, "Simple high-level code for cryptographic arithmetic-with proofs, without compromises", in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 1202–1219.

[14] 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 *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 917–934.

[15] 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", in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1807–1823.

[16] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules", in *Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001.*, IEEE, 2001, pp. 82–96.

[17] B. Blanchet, "Cryptoverif: computationally sound mechanized prover for cryptographic protocols", in *Dagstuhl seminar "Formal Protocol Verification Applied*, vol. 117, 2007, p. 156.

[18] K. Bhargavan, N. Kobeissi, and B. Blanchet, "Proscript TLS: building a TLS 1.3 implementation with a verifiable protocol model", in *TRON Workshop-TLS 1.3, Ready Or Not*, 2016.

[19] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate", in *2017 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2017, pp. 483–502.

[20] 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 *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1465–1482.

[21] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Béguelin, K. Bhargavan, J. Pan, and J. K. Zinzindohoue, "Implementing and proving the TLS 1.3 record layer", in *2017 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2017, pp. 463–482.

[22] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan, "Formally verified cryptographic web applications in WebAssembly", in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1256–1274. DOI: 10.1109/SP.2019.00064.

[23] C. Cremers and D. Jackson, "Prime, order please! revisiting small subgroup and invalid curve attacks on protocols using Diffie-Hellman", in *IEEE Computer Security Foundations Symposium (CSF)*, 2019, pp. 78–93.

[24] K. Bhargavan, F. Kiefer, and P.-Y. Strub, "Hacspec: towards verifiable crypto standards", in *Security Standardisation Research*, C. Cremers and A. Lehmann, Eds., Cham: Springer International Publishing, 2018, pp. 1–20, ISBN: 978-3-030-04762-7.

[25] 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 (SP)*, 2020, pp. 634–653.

[26] L. Erkök and J. Matthews, "Pragmatic equivalence and safety checking in Cryptol", in *Proceedings of the 3rd workshop on Programming languages meets program verification*, 2009, pp. 73–82.

[27] D. Mercadier and P.-É. Dagand, "Usuba: high-throughput and constant-time ciphers, by construction", in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 157–173.

[28] R. Dockins, A. Foltzer, J. Hendrix, B. Huffman, D. McNamee, and A. Tomb, "Constructing semantic models of programs with the software analysis workbench", in *Working Conference on Verified Software: Theories, Tools, and Experiments*, Springer, 2016, pp. 56–72.

[29] J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, V. Laporte, T. Oliveira, and P.-Y. Strub, "The last mile: high-assurance and high-speed cryptographic implementations", in *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2020, pp. 965–982.

[30] M. Lindner, J. Aparicius, and P. Lindgren, "No panic! Verification of rust programs by symbolic execution", in *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*, 2018, pp. 108–114.

[31] C. Cadar, D. Dunbar, D. R. Engler, *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.", in *OSDI*, vol. 8, 2008, pp. 209–224.

[32] Galois, Inc., *Crux-mir*, Oct. 26, 2020. [Online]. Available: `https://github.com/Galoisinc/mir-verifier`.

[33] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging Rust types for modular specification and verification", *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. DOI: `10.1145/3360573`. [Online]. Available: `https://doi.org/10.1145/3360573`.

[34] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: a verification infrastructure for permission-based reasoning", in *International Conference on Verification, Model Checking, and Abstract Interpretation*, Springer, 2016, pp. 41–62.

[35] S. Ullrich, *Simple verification of Rust programs via functional purification*, 2016.

[36] 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*, Springer, 2015, pp. 378–388.

[37] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamaric, and L. Ryzhyk, "System programming in rust: beyond safety", in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, May 8-10, 2017*, A. Fedorova, A. Warfield, I. Beschastnikh, and R. Agarwal, Eds., ACM, 2017, pp. 156–161. DOI: 10.1145/3102980.3103006. [Online]. Available: https://doi.org/10.1145/3102980.3103006.

[38] Z. Rakamaric and M. Emmi, "SMACK: decoupling source language details from verifier implementations", in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, A. Biere and R. Bloem, Eds., ser. Lecture Notes in Computer Science, vol. 8559, Springer, 2014, pp. 106–113. DOI: 10.1007/978-3-319-08867-9_7. [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9%5C_7.

[39] Y. Matsushita, T. Tsukada, and N. Kobayashi, "RustHorn: CHC-based verification for rust programs", in *European Symposium on Programming*, Springer, Cham, 2020, pp. 484–514.

[40] N. Bjørner, A. Gurfinkel, K. McMillan, and A. Rybalchenko, "Horn clause solvers for program verification", in *Fields of Logic and Computation II: Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, L. D. Beklemishev, A. Blass, N. Dershowitz, B. Finkbeiner, and W. Schulte, Eds. Cham: Springer International Publishing, 2015, pp. 24–51, ISBN: 978-3-319-23534-9. DOI: 10.1007/978-3-319-23534-9_2. [Online]. Available: https://doi.org/10.1007/978-3-319-23534-9_2.

[41] X. Denis, "Deductive program verification for a language with a Rust-like typing discipline", Université de Paris, Intership report, Sep. 2020. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02962804.

[42] J.-C. Filliâtre and A. Paskevich, "Why3—where programs meet provers", in *European symposium on programming*, Springer, 2013, pp. 125–128.

[43] 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*, vol. 2, no. POPL, pp. 1–34, 2017.

[44] Y. Nir and A. Langley, "Chacha20 and poly1305 for IETF protocols", *RFC*, vol. 8439, pp. 1–46, 2018. DOI: 10.17487/RFC8439. [Online]. Available: https://doi.org/10.17487/RFC8439.

[45] A. Weiss, D. Patterson, N. D. Matsakis, and A. Ahmed, "Oxide: the essence of rust", *CoRR*, vol. abs/1903.00982, 2019. arXiv: 1903.00982. [Online]. Available: http://arxiv.org/abs/1903.00982.

[46] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. Trieu, "Formal verification of a constant-time preserving C compiler", *Proc. ACM Program. Lang.*, vol. 4, no. POPL, 7:1–7:30, 2020.

[47] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations", in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, T. Holz and S. Savage, Eds., USENIX Association, 2016, pp. 53–70.

[48] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala, "Systematic generation of fast elliptic curve cryptography implementations", MIT, Tech. Rep., 2018.

[49] A. Fromherz, N. Giannarakis, C. Hawblitzel, B. Parno, A. Rastogi, and N. Swamy, "A verified, efficient embedding of a verifiable assembly language", *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, 2019.

[50] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "Riot: an open source operating system for low-end embedded devices in the IoT", *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, 2018.

[51] J. Fletcher, "An arithmetic checksum for serial transmissions", *IEEE transactions on Communications*, vol. 30, no. 1, pp. 247–252, 1982.

[52] G. Muller, J. L. Lawall, and H. Duchesne, "A framework for simplifying the development of kernel schedulers: design and performance evaluation", in *Ninth IEEE International Symposium on High-Assurance Systems Engineering (HASE'05)*, IEEE, 2005, pp. 56–65.

[53] 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 *International Joint Conference on Automated Reasoning*, Springer, 2020, pp. 119–137.

4. Steel: Divide and Conquer Proof Obligations

Contents

Abstract

Outside of the closed garden of cryptographic primitives, real-world software usually manipulates the memory in intricate ways that involve aliasing and complex pointer graphs. Verifying such programs has long remained a daunting task, until Reynolds came up with separation logic in 2002 [1].

However, separation logic imposes a strict format on Hoare-style specifications, and has traditionally limited automation through a tradeoff with expressiveness. Building on the concepts of the recent Iris [2] framework, we propose Steel, a new embedded domain-specific language for stateful program verification in F⋆, that mixes tactics and SMT for automating proofs about memory and everything else, while retaining the full expressive power of separation logic.

Ah ! que les charpentes
d'acier se multiplient donc,
dressent donc des édifices
utiles des villes heureuses,
des ponts pour franchir les
fleuves et les vallées et que
des rails jaillissent toujours
des laminoirs, allongent sans
fin les voies ferrées,
abolissent les frontières,
rapprochent les peuples,
conquièrent le monde entier
à la civilisation fraternelle de
demain !

(Émile Zola, Travail*, 1901)*

— It's too brittle when pure,
Spencer said, but if we alloy
it just a bit we'll have an
extremely light and strong
metal.
— Martian steel, Nadia said.
— Better than that.

(Kim Stanley Robinson,
Red Mars*, 1992)*

```
void reverse_message(char* msg, int len) {
  char* beginning = msg;
  char* end = msg + len - 1;
  // beginning and end both alias msg,
  // pointing at different locations
  for (int i = 0; i < len / 2; i++) {
    // Swapping beginning and end contents
    char tmp = *beginning;
    *beginning = *end;
    *end = tmp;
    // Moving to the middle
    beginning += 1;
    end -= 1;
  }
};
```

Figure 4.1: Illustrations of pointers-as-data and aliasing in C

4.1. From Implicit Dynamic Frames to Separation Logic

With LibSignal*, Chapter 2 explains how the methods for verifying cryptographic primitives can be scaled up to verify cryptographic protocols. However, the Low* domain-specific language suffers from structural weaknesses that place a limit to how complex memory manipulation can be. In this section, we describe these weaknesses, and present a state of the art related program verification framework that solves some of these weaknesses.

4.1.1. The Difficulties of Stateful Program Verification

A major source of difficulty for program verification is the presence of memory manipulation in the source program. There are many kinds of memory manipulation, and some can be formalized with lightweight constructs, such as a state monad with a limited number of memory locations that only store data. However, most low-level real-world software extensively uses a memory manipulation style made popular by the C programming language, featuring pointers-as-data and aliasing.

Figure 4.1 presents an example of C program that is both very idiomatic and very hard to verify. The `reverse_message` function simply reverses the order of the bytes stored in the array `msg` of length `len`. The implementation

```
let rec reverse_message (msg: Char.t list) : Char.t list =
  match msg with
  | [] -> []
  | hd::tl -> (reverse_message tl)@[hd]
```

Figure 4.2: Functional specification of Figure 4.1 in OCaml

uses two pointers that alias `msg` and range over its contents starting from both end and progressing to the middle, while swapping contents on the way.

Actually, the behavior of this function can be specified by the shorter functional-style program of Figure 4.2. But this style is not at all idiomatic in C, simply because it would lead to very poor performance at execution. This is a classic example of the tension between correctness and performance for low-level, real-world programs. Program verification is all about relieving that tension by bridging the gap between Figure 4.1 and Figure 4.2 through a functional correctness proof.

But the contents of this proof for the `reverse_message` involves modeling the aliasing of `beginning` and `end`, and their evolution inside the loop. There are several ways to model pointers and their aliasing. Here, Low⋆ [3] (presented in Section 2.1.2) provides a good example of a modeling based on implicit dynamic frames [4].

Low⋆ features a structured memory model *à la* Clight [5] with memory references indexed by natural numbers. Each of these memory references can hold any type of data, whose size can be arbitrary. Memory references are first-class values in Low⋆ programs, and can be passed around functions. However, references are limited to types in universe zero. The universes mentioned here refer to the classic notion of type universes in constructive type theory [6], [7], which has a strict equivalent in F⋆'s dependent type theory.

The crux of the modeling of memory manipulation is how it handles modularity, and proof composition. By default, every function call could modify the contents of every memory location. But this overly conservative assumption breaks any attempt at splitting the program into modular computation units such as functions. Hence, the specification of each function should state which part of the memory it modifies, the rest being left untouched. This observation is the essence of the frame rule in Reynold's separation logic [1]. But Low⋆ uses implicit dynamic frames, which is more amenable to encoding into first-order logic predicates (for automation). Hence, in Low⋆,

the frame rule is replaced by a library of lemmas governing the behavior of the `modifies: location -> mem -> mem -> prop` predicate, which can be read as follows: "`modifies l h0 h1` if and only if the only changes to memory contents between memories `h0` and `h1` happen in the references contained in location `l`". A memory location `l` is simply a set of memory references, that can be built be through successive unions of references. Another crucial predicate interacting with `modifies` is `disjoint: location -> location -> prop`, that indicates a separation between two memory locations. If `disjoint l l'` and `modifies l h0 h1`, then every memory reference in `l'` is left untouched between `h0` and `h1`.

In this framework, the critical lemma replacing the frame rule is:

```
val modifies_reference_elim (#a: Type) (r: reference a)
  (l: location) (h h': mem)
    : Lemma
      (requires (
        disjoint (loc_of_reference r) l /\ modifies l h h'))
      (ensures (reference_preserved r h h'))
```

Then, the heart of memory proofs in Low⋆ is all about applying inclusion, associativity and commutativity lemmas to locations inside `disjoint` and `modifies` predicates such that the proof context contains the right instances to call `modifies_reference_elim` and obtain the desired outcome, proving that a specific memory reference has been left untouched by a function call. Unlike separation logic which promotes a style where pointers are separated by default, Low⋆ considers pointers aliased by default, and it is up to the user to separate them.

For instance, consider the following function:

```
let function_to_verify (a b c: reference U32.t) =
  (* a <> c /\ b <> c *)
  let h0 = get () in
  step_1 a; (* modifies (loc_of_reference a) h0 h1 *)
  let h1 = get () in
  step_2 b; (* modifies (loc_of_reference b) h1 h2 *)
  let h2 = get () in
  assert(reference_preserved c h0 h2)
```

The proof goal `reference_preserved c h0 h2` can only be solved by a combination of lemmas about `disjoint` and `modifies`. Figure 4.3 shows all

```
let function_to_verify (a b c: reference U32.t) =
  (* a <> c /\ b <> c *)
  let h0 = get () in
  step_1 a; (* modifies (loc_of_reference a) h0 h1 *)
  let h1 = get () in
  step_2 b; (* modifies (loc_of_reference b) h1 h2 *)
  let h2 = get () in
  modifies_trans (loc_of_reference a) h0 h1
    (loc_of_reference b) h2;
  (* modifies
       (loc_union (loc_of_reference a) (loc_of_reference b))
       h0 h2 *)
  loc_disjoint_adresses c a;
  (* disjoint (loc_of_reference c) (loc_of_reference a) *)
  loc_disjoint_adresses c b;
  (* disjoint (loc_of_reference c) (loc_of_reference b) *)
  loc_disjoint_union_r c a b;
  (* disjoint
      (loc_of_reference c)
      (loc_union (loc_of_reference a) (loc_of_reference b)) *)
  modifies_reference_elim c
    (loc_union (loc_of_reference a) (loc_of_reference b))
    h0 h2;
  assert(reference_preserved c h0 h2)
```

Figure 4.3: Full F* proof for a reference contents preservation result.

the lemma steps that are necessary to reach the goal, there are 5 of them needed for this simple result. Imposing such a proof burden on Low* proof engineers is unthinkable more real-world programs, so F* provides a form of proof automation.

One method for automating these tedious proof would be to implement a tactic to pattern match on usual shapes of memory proofs and call the correct lemmas. But Low* was designed before a tactics system [8] was added to F*, hence the authors of the domain-specific language defaulted to the only system of proof automation available in F*: SMT patterns.

SMT patterns are lemma annotations representing a syntactic pattern. Here is what the SMT pattern looks like for our version of the frame rule:

```
val modifies_reference_elim (#a: Type) (r: reference a)
```

```
(l: location) (h h': mem)
  : Lemma
    (requires (
      disjoint (loc_of_reference r) l /\ modifies l h h'))
    (ensures (reference_preserved r h h'))
    [SMTPat (reference_preserved r h h');
      SMTPat (modifies l h h')]
```

At SMT encoding time, F⋆ translates at all the SMT patterns present in lemmas found in opened modules into e-matching directives of Z3 [9]. E-matching triggers when a program fragment matches the syntactic pattern of an pattern, and instantiate the pattern with the corresponding arguments, resulting in a call to the lemma behind the pattern. For instance, for `modifies_reference_elim`, a call to the lemma will be triggered every time expressions of the form `reference_preserved r h h'` and `modifies l h h'` are found at the same time in the encoding.

Importantly, SMT pattern triggering is recursive: if an SMT pattern inserts a lemma call that matches a pattern from another SMT pattern, this other pattern is triggered too. Hence, one can build SMT pattern systems that build whole proof trees ahead of time, based on syntactic structures of data or programs. In the generated proof trees, not all the branches yield a valid proofs, since the SMT patterns trigger regardless of whether the preconditions of lemmas are satisfied. This is what happens within Low⋆; the lemma libraries for `disjoint` and `modifies` contain a carefully crafted set of SMT patterns whose goal is to silently build proof trees that will automatically solve the most common kind of memory-related goals, using the SMT encoding of F⋆ and e-matching directives of Z3.

Alas, this convenient automation solution also comes with severe pitfalls that limit its practicality. The root of the problem is precisely the recursiveness of SMT pattern triggering that makes its strength. Indeed, the proof search strategy induced by an SMT patterns systems is akin to a Datalog query, where the SMT encoding fills the context with every possible branch of the proof tree, leaving to Z3 the task of checking whether there is at least a path in the tree that yields a correct proof, respecting all the preconditions of the lemmas involved. But contrary to Datalog where it is easy to pattern match on the desired result once the context is filled, Z3's task becomes harder as the size of the encoding generated by F⋆ grows.

For instance, the `loc_union` predicate in Low⋆ is associative and commu-

tative, and the associated lemmas have SMT patterns that trigger for every occurrence of `loc_union`. Hence, an expression as simple as `loc_union (loc_union a b) (loc_union c d)` will trigger dozens of SMT patterns that will generate all the possible associative-commutative rewritings of this union expression. These rewritings are sometimes needed for a particular application of the frame rule, but most of them are useless and pollute Z3's proof context, leading to a waste of the provers' resources. There are other documented patterns to avoid when relying on SMT for proofs in F*: a set of patterns that trigger too often (other than the associate-commutative rewriting we just mentioned), and non-linear arithmetic which is a known weak point of SMT solvers [10].

This waste of resources is particularly visible when programming in Low*, when the `z3rlimit` parameters (see Section 2.1.1) has to be set to very high value, meaning that the proof of a single top-level function can take up to several minutes for Z3 to complete. In Low* programs, a `z3rlimit` of 100 or above usually indicates a proof where the Z3 context explosion induced by the SMT pattern system has reached a critical state. Those proofs are empirically more prone to experience another of Z3's corner cases pitfall: proof flakiness. Indeed, the SMT solving heuristics of Z3 are non-deterministic and as such, proving a theorem once with Z3 does not mean that Z3 will manage to prove it again, even on the same machine and with the same condition. And the harder the proof, the flakier Z3 will behave. Note that proof flakiness is not limited to Low* but is rather a general problem with F* proofs that are not carefully crafted to avoid SMT context explosion. Hence, Low* programmers are frequently forced to go back to proofs that once passed and try some mitigation techniques to restore it.

The go-to mitigation technique is to add more details to the proof, by asserting the key intermediate steps and explicitly calling the critical lemmas. Nevertheless, this technique does not reduce the size of Z3's proof context; it merely give Z3 a more straightforward path through it. To reduce the proof context, it is possible to configure a single SMT query to only include context from a list of modules via the `--using_facts_from` option. SMT patterns in modules outside the provided whitelist will not trigger, and this allows for disabling a general proof infrastructure locally for a goal that is not related to the usual proof structure of a module.

But sometimes, a given function is simply too big (in terms of proof context generated) for Z3, and has to be split into smaller pieces. The process of splitting a function into smaller pieces is quite painful, since it involves cutting

the proof by writing manually the intermediate invariants and results as pre- and post-conditions of the resulting pieces. And real-world programming in Low* involves a lot of splitting: in LibSignal* (Section 2.3.2), each function handled 5 or 6 different memory references, which led to the SMT pattern triggering to flood Z3's proof context with only one or two stateful calls per body.

Overall, Low*'s memory-related lemma libraries and proof automation techniques have been able to scale to real-world applications like HACL*, LibSignal* or EverParse [11]. However, these projects necessitated a lot of F* expertise to craft proof contexts that didn't explode, and the resulting proof structure is not maintainable enough to scale to the next stage of applications. This observation motivated the search for a replacement to Low*, that would be built on the best practices of stateful program verification state of the art: separation logic.

4.1.2. Separation Logic in the Program Verification Literature

Since the discovery of separation logic [1], [12] in 2001-2002, many program verification frameworks have used it for their internal Hoare logic to specify program states. The separating conjunction connective $\star$ is the staple of the logic, which also introduces other specific connectives like the separating implication (magic wand) $\mathrel{-\!\!*}$. Several program verification frameworks have adopted separation logic in the next decades.

Viper [13] (already mentioned in Section 1.3.2) is based on an intermediate verification language that features a permission-based modeling closely inspired by separation logic. The strength of Viper is the efficient encoding of the verification conditions expressed in that intermediate language into SMT or Boogie [14] queries; notably Viper has a special encoding of $\mathrel{-\!\!*}$ [15], exposed through special `inhale` and `exhale` predicates in the intermediate verification language.

VeriFast [16] is an older program verification framework whose input language is a subset of C extended with contract annotations on functions, as well as proof-related predicates. As its names states, VeriFast aims at providing verification answers in real time. The typechecker has a custom verifier implementation that handles automatically the solving of common separation logic proof goals, and falls back to encoding obligations about program data to an SMT solver. The reliance on a custom implementation for separation-logic-related proof goals avoid the trap of Low*'s SMT encoding,

and performs associative-commutative rewriting of separation logic terms efficiently. Moreover, the speedy termination of the SMT calls is guaranteed by a restriction on the rolling and unrolling of recursive predicates and inductive data types. These rolling and unrolling must be specified manually in the source program for the proofs to complete. In that regard, the SMT encoding of VeriFast is less powerful than F⋆'s with its `--fuel` and `--ifuel` options – but F⋆'s Z3 calls are not guaranteed to terminate quickly.

It is worth noting that some stateful program verification purposely avoid separation logic, instead opting for imposing restrictions on pointer aliasing that enable more amenable decision procedures for memory-related proofs. This is the case of Why3 [17] and its pragmatic method of static aliasing [18]. But so far, the main applications of this framework have been limited to classic data structures [19], [20].

The main innovation in program verification by separation logic in the last decade has been the Iris [2] framework, embedded inside Coq. The notable difference with previous works is that Iris is a separation-logic-powered program verification semantics that can be parametrized by any source programming language semantics on which the verification is performed. To enable that parametricity, the Iris authors meticulously carved out and combined the minimal abstractions that were required to make concurrent separation logic work. In their first paper [21], Jung *et al.* discover an "orthogonal basis for concurrent reasoning" : partial commutative monoids and invariants.

Separation logic is not exclusive to computer memory, and can be applied to any resource represented by a partial commutative monoid: a set governed by a composition law that is associative, commutative, and has an absorbing and a distinct unit element in the set. In the case of memory, elements of the set are fragments of memory, and the composition law is the union of those fragments. The unit element is the empty fragment of memory, and the absorbant is a special undefined fragment. But the monoidal structure can also apply to ghost state containing memory invariants about the program.

This monoidal structure allows reasoning over different threads that each own a fragment of memory. But concurrent reasoning is all about shared memory, and for that Iris introduces the notion of invariants. Invariants encapsulate a separation logic term about a shared memory fragment, and restrict its usage to atomic programs via strict open and close rules. The combination of partial commutative monoids and invariants allow for a very expressive resulting separation logic, parametrized over a programming language semantics, that yields concurrent program verification frameworks.

In their second paper, Jung *et al.* extend their original work to allow for storing higher-order separation logic predicates inside ghost state. This restricted form of self-referencing needed a significant theoretical work to be proven sound, but enabled a critical leap in expressiveness needed for real-world programs. For instance, the original version of Iris could not handle programs that dynamically choose their communication channels between main and worker threads. Finally, the theoretical foundations of Iris have been rationalized and mechanically formalized in Coq in a third paper [22].

In parallel to these theoretical developments, Iris was successfully applied to enable groundbreaking applications in concurrent program verification, thanks to the Iris proof mode [23], which eased the proof engineering with Iris thanks to custom syntax, tactics for the common cases and proof context display. The first main real-world application of Iris has been the proof of memory safety of critical `unsafe` parts of the Rust standard library [24]. Then, applications of Iris have included a concurrent and crash-safe mail server written in a subset of Go [25], a concurrent journaling system also written in Go [26] a fixpoint solver for monotonic equations performing memoization [27], a formalized weak memory semantics for multicore OCaml [28], and a multiple-producer multiple-consumer concurrent queue in a weak memory setting [29].

While these first-class applications demonstrate the expressive power of Iris on complex, critical, concurrent programs, its ability to scale up to programs of several thousand lines of code (the middle of Figure 1.1 in Section 1.1.1) is still unknown. Even with a good library of tactics, we believe that some form of automation using SMT solvers is necessary to ease the proof burden. This ambitious goal was our primary motivation to attempt to construction of our own separation logic framework in F*: Steel.

4.2. Bringing Separation Logic to F*

While this section and the next focus on explaining the Steel framework for concurrent separation logic program verification in F*, we start here by describing two attempts at building the foundations of Steel. The first attempt, ultimately failed and unpublished, provides an interesting look at the motivation of program verification tool builders and the constraints that make this task hard. The second attempt, more successful, was largely inspired by the theoretical insights of Iris while bringing novel aspects.

4.2.1. Breaking with the Low* legacy

This subsection is based upon the following collaboration:

From August 2019 to October 2019, I visited the RiSE group of Microsoft Research, as part of a collaborative effort to improve low-level programming in F: Steel. This subsection is a* post-mortem *of the failed first attempt at founding Steel over Low*. The first attempt involved Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Danel Ahman, Guido Martinez, Tahina Ramananandro, Jonathan Protzenko and me, and was based on previous work also involving Zoe Paraskevopoulou. This succinct account of what happened presented here is exclusive and my personal contribution. I also contributed a presentation based on this work at the ADSL 2020 workshop.*

Since this dissertation is about language design, we believe that the description of this failed attempt provides multiple insights about negative patterns to avoid when evolving a program verification framework.

The main motivation for starting this project was Low*'s known weakness for scaling up memory-safety-related proof, documented in Section 4.1.1. To solve the SMT context explosion caused by the SMT patterns system, we began by looking for ways to abstract and encapsulate memory predicates for objects. Indeed, the memory safety for a linked list in Low* involved the proof of no less that 5 specialized lemmas relating the footprint of the list, that way is was modified and how its head was disjoint from its tail while unrolled [30]. The missing key property for this kind of proofs was a more modular version of the frame-rule-like `modifies_reference_elim` predicate from Section 4.1.1 that could be used to prove memory non-intereference for whole structures at a time, rather than at the level of each reference.

The first attempt at Steel was thus to provide a methodology for building component-wide frame-rule-like lemmas for user-defined data structures, all of this as a proof layer on top of Low*. The inspiration for this attempt was Parkinson and Summer's work [31] of building a full separation logic on top of implicit dynamic frames [4], which is the logical foundation of Low*.

After a previous failed attempt using the category theoretical concept of lenses, we settled on a design related to the Views [32] framework for representing physical ressources. In this attempt, our ressources bundled three components :

1. the footprint of the resource, expressed in terms of Low*'s `location` type (a set of memory references);

2. the invariant of the resource, a higher-order predicate that depends on a memory where the footprint was live (*i.e* allocated and owned);

3. the view of the resource, another higher-order predicate depending on a memory where the footprint is live, but whose goal is to express a high-level specification of the content of the resource in memory (*i.e* a functional list of a Low* linked list).

Each of these components must satisfy a lemma that state their consistency with the frame rule. For instance, for the footprint, one of the properties to check is:

```
let footprint_frame_rule
  (fp: mem -> GTot location)
  (inv: mem -> prop)
    : prop
  =
  forall (h0 h1: mem) (loc: location).
    (loc_disjoint (fp h0) loc /\ modifies loc h0 h1 /\ inv h0)
    ==> fp h0 == fp h1
```

This states that all memory transformations that have a footprint (`loc`) disjoint with the resources' footprint (`fp`), should leave the resources' footprint intact. This kind of property must be proved once and for all when a data structure is turned into a resource. Then, the resource's implementation can be hidden behind an abstraction layer (in F*, `.fsti` interface files) so that Low* programs using it do not see their proof context flooded with the data structure's internals.

This abstraction layer should also expose all the mid-level functions necessary to interact with the data structure: allocation, deallocation, read and write commands, sharing and gathering (permission-wise if the structure has permissions). Above the abstraction layer, Low* programs should use a new **`RST`** abbreviation effect (see Section 2.1.2), equipped with five indexes instead of three (for the regular **`Stack`** effect): the first one for the return type of the computation, the last two for the pre- and post- conditions, and two new intermediate ones describing the pre- and post-resources. The pre- and

post-resource describe the state of memory before and after the computation; indeed, resources can be composed together using a $\star$ operator similar to the separating conjunction of separation logic.

The motivation for this dual specification baked in the effect (pre- and post-resources as well as pre- and post-conditions) is the lack of expressivity and flexibility of the resources. They cannot be easily changed since determined once and for all for their invariant, footprint, views and basic operations are determined once and for all for the whole data structure when building the `.fsti` abstraction layer. Hence, a finer and more versatile specification using pre- and post-condition was needed.

While this new effect enabled promising developments and uncovered the need for more Steel proof infrastructure (that will appear in Section 4.3), its foundations appeared to be flawed when trying to create a simple array resource similar to Low*'s buffers. The flaw was stemming from the underlying memory model of Low* and the `modifies` and `disjoint` predicates for memory locations. Indeed, it was impossible to prove that allocation and deallocation of arrays complied with the frame rule, in the sense of properties like `footprint_frame_rule`. The culprit was a design flaw in the abstract set theory underlying memory locations (viewed as a set of references). This set theory did not support intersection nor complement, but only unions. We made an attempt at retrofitting the set theory to add the features we needed, but the new version broke the compatibility with the buffer libraries of Low* due to the retrocompatibility with memory regions. Memory regions were a feature initially present in Low* and used in the miTLS [33] codebase, but considered as deprecated in HACL*. Hence, the set theory needs to cope with intertwined sets of references and sets of regions, making for a difficult implementation.

Because of this incompatibility, the goal of making Steel programs verifiably interoperable with Low* programs by building the new theory on top of the old one became inaccessible. Therefore, we wondered if building Steel on top of Low* was a good idea at all. What this deep dive into the foundations of Low* showed us was that the foundations of Low* were not powerful enough to enable the full expressive power of a complete separation logic. Moreover, we wanted Steel to support concurrency out of the box to extend the field of our applications. We thus decide to break with the Low* legacy, go back to the drawing board and design Steel from the ground up on more principled foundations.

4.2.2. SteelCore: A Foundational Core for Steel

This subsection is based upon the following publication:

N. Swamy, A. Rastogi, A. Fromherz, D. Merigoux, D. Ahman, and G. Martinez, "Steelcore: an extensible concurrent separation logic for effectful dependently typed programs", *Proc. ACM Program. Lang.*, vol. 4, no. ICFP, Aug. 2020. DOI: 10.1145/3409003. [Online]. Available: https://doi.org/10.1145/3409003

My personal contribution was a design, implementation, and proof of an intermediate version of the memory model. The version presented in the published paper is substantially different.

Proof assistants based on type theory can be a programmers' delight, allowing one to build modular abstractions coupled with strong specifications that ensure program correctness. Their expressive power also allows one to develop new program logics within the same framework as the programs themselves. A notable case in point is the Iris framework [2] embedded in Coq [35], which provides an impredicative, higher-order, concurrent separation logic (CSL) [1], [12], [36] within which to specify and prove programs.

Iris has been used to model various languages and constructs, and to verify many interesting programs [25], [37], [38]. However, Iris is not in itself a programming language: it must instead be instantiated with a *deeply embedded* representation and semantics of one provided by the user. For instance, several Iris-based papers work with a mini ML-like language deeply embedded in Coq [23].

Taking a different approach, FCSL [39]–[41] embeds a predicative CSL in Coq enabling proofs of Coq programs (rather than embedded-language programs) within a semantics that accounts for effects like state and concurrency. This allows programmers to use the full power of type theory not just for proving, but also for programming, e.g., building dependently typed programs and metaprograms over inductive datatypes, with typeclasses, a module system, and other features of a full-fledged language. However, Nanevski et al.'s program logics are inherently predicative, which makes it difficult to express constructs like dynamically allocated invariants and locks, which are natural in impredicative logics like Iris.

In this subsection, we present the highlights of a new framework called SteelCore that aims to provide the benefits of Nanevski et al.'s shallow embed-

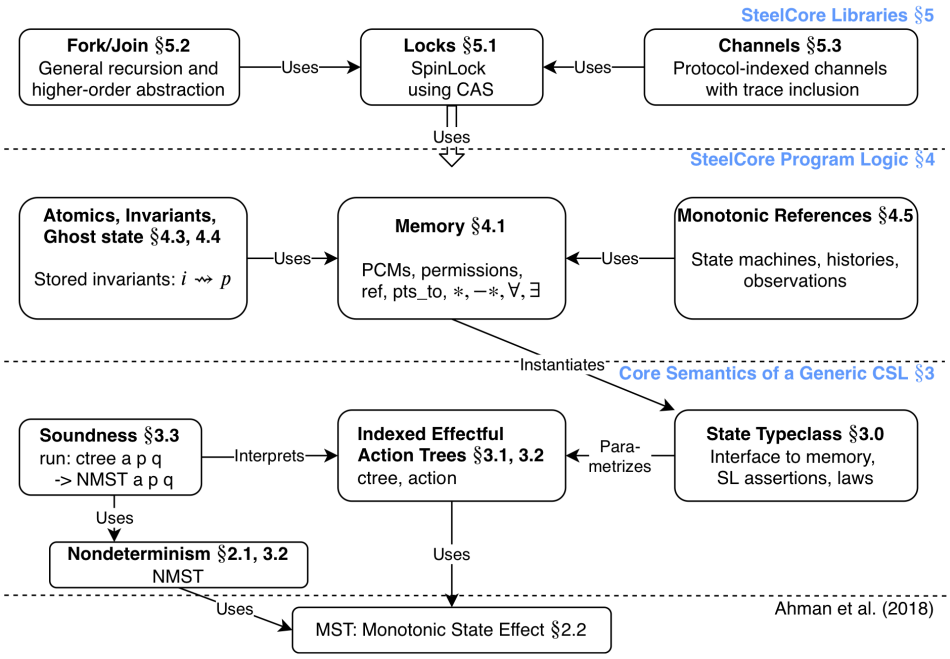

Figure 4.4: An overview of SteelCore, section numbers from [34]

dings, while also supporting dynamically allocated invariants and1 locks in the flavor of Iris. Specifically, we develop SteelCore in the *effectful* type theory provided by the F* proof assistant [42]. One of our main insights is that an effectful type theory is not only useful for programming; it can also be leveraged to build new program logics for effectful program features like concurrency (see also Section 2.1.2). Building on prior work [43] that models the effect of monotonic state in F*, we develop a semantics for concurrent F* programs while simultaneously deriving a CSL to reason about F* programs using the effect of concurrency. The use of monotonic state enables us to account for invariants and atomic actions entirely within SteelCore. The net result is that we can program higher-order, dependently-typed, generally recursive, shared-memory and message-passing concurrent F* programs and prove their partial correctness using SteelCore. Full explanations and evaluation of the framework can be found in the related publication [34].

SteelCore is the core semantics of Steel, a DSL in F* for programming and proving concurrent programs. In this subsection, we focus primarily on the semantics, leaving a treatment of other aspects of the Steel framework to the next section. Building on the monotonic state effect, we prove sound a generic program logic for concurrency, parametric in a memory model and a

```
type ctree (st:state) :
     a:Type -> pre:st.slprop ->
     post:(a -> st.slprop) -> Type
   =
   | Ret : x:a -> ctree st a (post x) post
   | Act : action pre post -> ctree st a pre post
   | Par : ctree st a p q -> ctree st a' p' q' ->
     ctree st (a & a') (p `st.star` p')
       (fun (x, x') -> q x `st.star` q' x')
   | Bind : ctree st a p q ->
     ((x:a) -> Dv (ctree st b (q x) r)) ->
     ctree st b p r
```

Figure 4.5: SteelCore action trees as an F* datatype

separation logic. We instantiate this semantics with a separation logic based on partial commutative monoids, stored invariants, and state machines. Using this logic, we program verified, dependently typed, higher-order libraries for various kinds of concurrency constructs, culminating in a library for message-passing on typed channels. We present an overview of several novel elements of our contributions, next.

For starters, we need to extend F* with concurrency. To do this, we follow the well-known approach of encoding computational effects as definitional interpreters over free monads [44]–[47]. That is, we can represent computations as a datatype of (infinitely branching) trees of atomic actions. When providing a computational interpretation for action trees, one can pick an execution strategy (e.g., an interleaving semantics) and build an interpreter to run programs. The first main novelty of our work is that we provide an intrinsically typed definitional interpreter [48] that both provides a semantics for concurrency while also deriving a CSL in which to reason about concurrent programs. Enabling this development is a new notion of indexed action trees, which we describe next.

Indexed action trees for structured parallelism We represent concurrent computations as an instance of the datatype `ctree st a pre post`, shown in Figure 4.5. The `ctree` type is a tree of atomic computational actions, composed sequentially or in parallel.

The type `ctree st a pre post` is parameterized by an instance `st` of the

`state` typeclass, which provides a generic interface to memories, including `st.slprop`, the type of separation logic assertions, and `st.star`, the separating conjunction. The index `a` is the result type of the computation, while `pre` and `post` are separation logic assertions. The **`Act`** nodes hold stateful atomic actions; **`Par`** nodes combine trees in parallel; while **`Bind`** nodes sequentially compose a computation with a potentially divergent continuation, as signified by the **`Dv`** effect label. Divergent computations are primitively expressible within F*, and are soundly isolated from its logical core of total functions by the effect system.

Interpreting action trees in the effects of nondeterminism and monotonic state We interpret a term `(e : ctree st a pre post)` as both a computation `e` as well as a proof of its own partial correctness Hoare triple `{pre} e : a {post}`. To prove this sound, we define an interpreter that non-deterministically interleaves atomic actions run in parallel. The interpreter is itself an effectful F* function with the following (simplified) type, capturing our main soundness theorem:

```
val run (e:ctree st a p q) : NMST a st.evolves
  (fun m -> st.interp p m)
  (fun _ x m' -> st.interp (q x) m')
```

where **`NMST`** is the effect of monotonic stateful computations extended with nondeterminism. Here, we use it to represent abstract, stateful computations whose states are constrained to evolve according to the preorder `st.evolves`, and which when run in an initial state `m` satisfying the interpretation of the precondition `p`, produce a result `x` and final state `m'` satisfying the postcondition `q x`. As such, using the Hoare types of **`NMST`**, the type of `run` validates the Hoare rules of CSL given by the indexing structure on `ctree`. In doing so, we avoid the indirection of traces in Brooke and Stephen's original proof of CSL [49] as well as in the work of Nanevski *et al.* [40].

Atomics and Invariants: Breaking circularities with monotonic state Although most widely used concurrent programming frameworks, e.g., the POSIX pthread API, support dynamically allocated locks, few existing CSL frameworks actually support them, with some notable exceptions [2], [50]–[53]. The main challenge is to avoid circularities that arise from storing locks that are associated with assertions about the memory in the memory itself. Iris, with

its step-indexed model of impredicativity, can express this. However, other existing state of the art logics, including FCSL, cannot. We show in details in the related publication [34] how to leverage the underlying model of monotonic state to allocate a stored invariant, and to open and close it safely within an atomic command, without introducing step indexing.

PCMs, ghost state, state machines, and implicit dynamic frames We base our memory model on partial commutative monoids (PCMs), allowing the user to associate a PCM of their choosing with each allocation unit. Relying on F*'s existing support for computationally irrelevant erased types, we can easily model *ghost state* by allocating values of erased types in the heap, and manipulating these values only using atomic ghost actions – all of which are erased during compilation. PCMs in SteelCore are orthogonal from ghost state: they can be used both to separate and manage access permissions to both concrete and ghost state – in practice, we use fractional permissions to control read and write access to references. Further, SteelCore includes a notion of *monotonic* references, which when coupled with F*'s existing support for ghost values and invariants, allow programmers to code up various forms of *state machines* to control the use and evolution of shared resources. Demonstrating the flexibility of our semantics, we extend it to allow augmenting CSL assertions with frameable heap predicates, a style that combines CSL with implicit dynamic frames [4] within the same mechanized framework.

Putting it to work We present several examples showing SteelCore at work, aiming to illustrate the flexibility and extensibility of the logic and its smooth interaction with dependently typed programming in F*. Starting with an atomic compare-and-set (CAS) instruction, we program verified libraries for spin-locks, for fork/join parallelism, and finally for protocol-indexed channel types. Our channel-types library showcases dependent types at work with SteelCore: its core construct is a type of channels, `chan p`, where `p` is itself a free-monad-like computation structure "one-level up" describing an infinite state machine on types. We prove, once and for all, that programs using a `c:chan p` exchange a trace of messages on `c` accepted by the state machine `p`.

Summary of the paper's contributions SteelCore is a fully mechanized CSL embedded in F*, and applicable to F* itself. The code and proofs are available from `https://fstar-lang.org/papers/steelcore`. In summary, the new design decisions behind SteelCore allowed us to go past the failed attempt of Section 4.2.1, and build solid semantic foundations for a new separation-logic-based, concurrent program verification framework in F*: Steel. The contributions of these semantic foundations include the following:

- A new construction of indexed, effectful action trees, mixing data and effectful computations to represent concurrent, stateful and potentially divergent computations, with an indexing structure capturing the proof rules of a generic CSL.
- An intrinsically typed definitional interpreter that interprets our effectful action trees into another effect, namely the effect of nondeterminism layered on the effect of monotonic state. This provides both a new style of soundness proof for CSL, as well as providing a reference executable semantics for our host language F* extended with concurrency.
- An instantiation of our semantics with a modern CSL inspired by recent logics like Iris, with a core memory model based on partial commutative monoids and support for dynamically allocated invariants. Relying on the underlying semantic model of monotonic state is a key element, allowing us to internalize the step-indexing that is necessary in Iris for dealing soundly with invariants.
- We use our logic to build several verified libraries, programmed in and usable by dependently typed, effectful host-language programs, validating our goal of providing an Iris-style logic applicable to a shallow rather than a deeply embedded programming language.

Summary of personal contributions My personal contributions to SteelCore revolve around the memory model, whose final form is presented in Section 4.1 of the SteelCore paper. In particular, I added support for arrays within references, and designed and implemented a library of atomic memory actions compatible with the separation logic semantics and monotonic preorders.

I authored the pull request #1898 that contained an implementation (complete with proofs) of a memory model for SteelCore including permissions and atomic actions on references containing arrays. Later, I added support for

monotonicity to this memory model by equipping each array cell inside each memory cell with a preorder; the combination of all of these individual preorders would form the heap-wide preorder, whose monotonicity is guaranteed by the Steel effect. Finally, other SteelCore authors reshaped this memory model to be based on the more generic notion of PCM, rather than on the notion of cells containing arrays.

4.3. Separating Memory Proofs from Functional Correctness

This section is based upon the following publication:

A. Fromherz, A. Rastogi, N. Swamy, S. Gibson, G. Martinez, D. Merigoux, and T. Ramananandro, "Steel: proof-oriented programming in a dependently typed concurrent separation logic", *Proc. ACM Program. Lang.*, vol. 5, no. ICFP, Aug. 2021. DOI: 10.1145/3473590. [Online]. Available: https://doi.org/10.1145/3473590

My personal contributions for this paper are less substantive than for the SteelCore paper. Concretely, I proposed a design and implementation for selectors and framing tactic in an early version of the Steel framework, on which I worked from August to October 2019. The version of these items presented in the published paper is substantially different.

Structuring programs with proofs in mind is a promising way to reduce the effort of building high-assurance software. There are many benefits: the program structure can simplify proofs, while proofs can simplify programming too by, for example, eliminating checks and unnecessary cases. Programming languages of many different flavors embrace this view and we use the phrase *proof-oriented programming* to describe the paradigm of co-developing proofs and programs.

Dependently typed programming languages, like Agda and Idris, are great examples of languages that promote proof-oriented programming. As Brady [55] argues, the iterative "type-define-refine" style of type-driven development allows programs to follow the structure of their dependently typed specifications, simplifying both programming and proving. From a different community, languages like Dafny [56], Chalice [57], and Viper [13] enrich imperative

languages with first-order program logics for program correctness, driving program development from Floyd-Hoare triples, with proof automation using SMT solvers. Hoare Type Theory [39] combines these approaches, embedding Hoare logic in dependently typed Coq with tactic-based proofs, while F* [42] follows a similar approach but, like Dafny, Chalice and Viper, uses an SMT solver for proof automation.

In this section, we aim to present some aspect of a proof-oriented programming language based on SteelCore [34] (see Section 4.2.2), the recent concurrent separation logic (CSL) [1], [36] for dependently typed programs formalized in F*. Our goal is to integrate the expressive power of the SteelCore logic within a higher-order, dependently typed programming language with shared-memory and message-passing concurrency, with proof automation approaching what is offered by Dafny, Chalice, and Viper, but with soundness ensured by construction upon the foundations of SteelCore.

We have our work cut out: SteelCore, despite providing many features as a logic, including an impredicative, dependently typed CSL for partial-correctness proofs, with a user-defined partial commutative monoid (PCM)-based memory model, monotonic state, atomic and ghost computations, and dynamically allocated invariants, all of whose soundness is derived from a proof-oriented, intrinsically typed definitional interpreter in F*, is far from being usable directly to build correct programs.

SteelCore's main typing construct is a Hoare type, `SteelCore a p q`, describing potentially divergent, concurrent, stateful computations returning a value of type `a` and whose pre- and postconditions are `p:slprop` and `q:a -> slprop`, where `slprop` is the type of separation logic propositions. Figure 4.6 shows, at the top, a SteelCore program that swaps the content of two references. Calls to `frame` wrapping each action combined with rearrangements of `slprop`s with `commute_star` overwhelm the program – the pain is perceptible.

Through several steps, we offer instead Steel, an embedded domain-specific language (DSL) within F* based on SteelCore, which enables writing the swap program at the bottom of Figure 4.6. We briefly call out some of its salient features. First, we introduce a new "quintuple" computation type, shown below:

```
Steel a (p:slprop) (q:a -> slprop)
  (requires (r:pre p))
  (ensures (s:post p a q))
```

```
let swap (#v1 #v2:ghost int) (r1 r2:ref int)
  : SteelCore unit
      (pts_to r1 #v1 * pts_to r2 #v2)
      (fun _ -> pts_to r1 #v2 * pts_to r2 #v1)
  = let x1 = frame (read r1) (pts_to r2 #v2) in
    commute_star (pts_to r1 #v1) (pts_to r2 #v2);
    let x2 = frame (read r2) (pts_to r1 #v1) in
    frame (write v2 x1) (pts_to r1 #x2);
    commute_star (pts_to r2 #x1) (pts_to r1 #v1);
    frame (write r1 x2) (pts_to r2 #v2);

let swap (r1 r2:ref int) : Steel unit
   (ptr r1 * ptr r2) (fun _ -> ptr r1 * ptr r2)
   (requires fun _ -> True)
   (ensures fun s _ s' ->
      s'.[r1] = s.[r2] /\ s'.[r2] = s.[r1])
  = let x1 = read r1 in
    let x2 = read r2 in
    write r2 x1;
    write r1 x2
```

Figure 4.6: SteelCore implementation of swap (top); Steel version (bottom)

The additional indices `r` and `s` are *selector predicates*, that depend only on the `p`-fragment of the initial memory and `q`-fragment of the final memory, i.e., they are self-framing, in the terminology of Parkinson and Summers [31]. These predicates are SMT encodeable and allow a useful interplay between tactic-based proofs on `slprops` and SMT reasoning on the content of memory.

In swap, these predicates also remove the need for existentially quantified ghost variables to reason about the values stored in the two references (i.e., the function arguments `v1` and `v2`). Next, through the use of F*'s effect system, we encode a syntax-directed algorithm to automatically insert applications of the frame rule at the leaves of a computation, while a tactic integrated with the DSL solves for frames using a variant of AC-matching [58].

Freed from the burden of framing, the program's intent is evident once more. The `swap` program in Steel is perhaps close to what one would expect in Chalice or Viper, but we emphasize that Steel is a shallow embedding in dependently typed F* and the full SteelCore logic is available within Steel. So, while Viper-style program proofs are possible and encouraged, richer, dependently typed idioms are also possible and enjoy many of the same benefits, e.g., automated framing and partial automation via SMT. Indeed, our approach seeks only to automate the most mundane aspects of proofs, focusing primarily on framing. For the rest, including introducing and eliminating quantifiers, rolling and unrolling recursive predicates, writing invariants, and manipulating ghost state, the programmer can develop lemmas in F*'s underlying type theory and invoke those lemmas at strategic points in their code – the Steel library provides many generic building blocks for such lemmas. The result is a style that Leino and Moskal [59] have called *auto-active* verification, a mixture of automated and interactive proof that has been successful in several other languages, including in other large F* developments, but now applied to SteelCore's expressive CSL.

Steel is entirely implemented in the F* proof assistant, with proofs fully mechanized upon the SteelCore program logic. All of our code and proofs are open-source, and publicly available online[1].

4.3.1. From Hoare Triplets to Quintuples

Like any separation logic, SteelCore has rules for framing, sequential composition, and consequence, shown below in their first, most simple forms,

[1] `https://zenodo.org/record/4768854`

where the type `stc a p q` represents a Hoare type with `a:Type`, `p:slprop`, and `q:a -> slprop`. These proof rules are implemented in SteelCore as combinators with the following signatures:

```
let stc a p q = unit -> SteelCore a p q
  (* represents { p } x:a { q x } *)
val frame (_:stc a p q) : stc a (p ⋆ f) (fun x -> q x ⋆ f)
val bind (_:stc a_1 p q')
  (_: (x:a_1 -> stc a_2 (q' x) r))
    : stc a_2 p r
val conseq (_:stc a p' q') (_:squash (p -⋆ p' /\ q' -⋆ q))
    : stc a p q
```

Our goal is to shallowly embed Steel as a DSL[2] in F⋆, whereby Steel user programs are constructed by repeated applications of combinators like `frame`, `bind` and `conseq`. The result is a program whose inferred type is a judgment in the SteelCore logic, subject to verification conditions (VCs) that must be discharged, e.g., the second argument of `conseq`, `squash (p -⋆ p' /\ q' -⋆ q)`, is a proof obligation.

For this process to work, we need to make the elaboration of a Steel program into the underlying combinator language algorithmic, resolving the inherent nondeterminism in rules like Bind and Consequence by deciding the following: first, where exactly should Bind and Consequence be applied; second, how should existentially bound variables in the rules be chosen, notably the frame `f`; and, finally, how should the proof obligations be discharged.

Verification Condition Generation for Separation Logic The standard approach to this problem is to define a form of weakest precondition (WP) calculus for separation logic that strategically integrates the use of frame and consequence into the other rules in the system. Starting with Ishtiaq and O'Hearn's [60] "backwards" rules, weakest precondition readings of separation logic have been customary. Hobord and Villard [61] propose a ramified frame rule that integrates the rule of consequence with framing, while Iris' [2] "Texan triples" combine both ideas, integrating a form of ramified framing in the

[2]A note on terminology: From one perspective, Steel is not domain-specific – it is a general-purpose, Turing complete language, with many kinds of computational effects. But, from the perspective of its host language F⋆, Steel is a domain-specific language for proof-oriented stateful and concurrent programming.

WP-Wand rule of its WP calculus. In the setting of interactive proofs, Texan triples are convenient in that every command is always specified with respect to a parametric postcondition, enabling it to be easily applied to a framed and weakened (if necessary) postcondition.

Prior attempts at encoding separation logic in F* [8] followed a similar approach, whereby a Dijkstra monad [62] for separation logic computes weakest preconditions while automatically inserting frames around every function call or primitive action. However, Martínez *et al.* [8] have not scaled their prototype to verify larger programs and we have, to date, failed to scale their WP-based approach to a mostly-automated verifier for Steel.

The main difficulty is that a WP-calculus for separation logic computes a single (often quite large) VC for a program in, naturally, separation logic. F* aims to encode such VCs to an SMT solver. However, encoding a separation logic VC to an SMT solver is non-trivial. SMT solvers like Z3 [63] do not handle separation logic well, in part because `slprops` are equivalent up to Associativity-Commutativity (AC) rewriting of $\star$, and AC-rewriting is hard to encode efficiently in SMT. Besides, WP-based VCs heavily use magic wand and computing frames involves solving for existential quantifiers over AC terms, which again is hard to automate in SMT. Viper (the underlying engine of Chalice) does provide an SMT-encoding for a permission system with implicit dynamic frames that is equivalent to a fragment of separation logic [31], however, we do not have such an encoding for SteelCore's more expressive logic. While some other off-the-shelf solvers for various fragments of separation logic exist [64], [65], using them for a logic like SteelCore's dependently typed, impredicative CSL is an open challenge.

Martínez *et al.* [8] confront this problem and develop tactics to process a separation logic VC computed by their Dijkstra monad, AC-rewriting terms and solving for frame variables, and finally feeding a first-order logic goal to an SMT solver. However, this scales poorly even on their simpler logic, with the verification time of a program dominated by the tactic simply discovering fragments of a VC that involve non-trivial separation logic reasoning, introducing existentially bound variables for frames, solving them and rewriting the remainder of the VC iteratively.

Our solution addresses these difficulties by developing a verification condition generator for quintuples, and automatically discharging the computation of frames using a combination of AC-matching tactics and SMT solving, while requiring the programmer to write invariants and to provide lemmas in the form of imperative ghost procedures. We now present our elaboration and VC

constant	T	::=	unit | () | Type | prop | slprop | ...
term	e, t	::=	$x \mid T \mid \lambda x{:}t.\ e \mid e_1\ e_2 \mid x{:}t \to C \mid$ ret $e \mid$ bind $e_1\ x.e_2$
		|	$e_1\ *\ e_2 \mid e_1 \mathrel{-\!\!*}\ e_2 \mid e_1\ \wedge\ e_2 \mid \forall x.e \mid \ldots$
computation type	C	::=	Tot $t \mid \{\ P \mid R\ \}\ y{:}t\ \{\ Q \mid S\ \}$
program	d	::=	val $f\ (x : t)\ :\ C\ =\ e$

Figure 4.7: Simplified syntax for Steel

generation strategy for Steel as a small idealized calculus.

We transcribe the rules omitting some side conditions (e.g., on the well-typedness of some terms) when they add clutter – such conditions are all captured formally in our mechanization. As such, these rules are implemented as combinators in F⋆'s effect system and mechanically proven sound against SteelCore's logic in F⋆. See Section 4 of [54], for a study of the metatheory of the system from the perspective of completeness and the solvability of the constraints it produces.

A Type-and-Effect System for Separation Logic Quintuples Figure 4.7 presents the syntax of a subset of the internal desugared, monadic language of Steel in F⋆. Our implementation supports the full F⋆ language, including full dependent types, inductive types, pattern matching, recursive definitions, local let bindings, universes, implicit arguments, a module system, typeclasses, etc. This is the advantage of a shallow embedding: Steel inherits the full type system of F⋆. For the purposes of our minimalistic presentation, the main constructs of interest are `slprop`s and computation types, though an essential preliminary notion is a memory, which we describe first.

A memory `mem` represents the mutable heap of a program and SteelCore provides an abstract memory model of mutable higher-order typed references, where each memory cell stores an element in the carrier type of a user-chosen PCM. For the specifics of the memory model and case studies, see Section 5 of the related publication [54]. For now, it suffices to note that `mem` supports two main operations:

- `disjoint (m0 m1: mem) : prop`, indicating that the domains of the two memory maps are disjoint;
- `join (m0:mem) (m1:mem{disjoint m0 m1}) : mem`, the disjoint union

of two memories.

An `slprop` is a typeclass that encapsulates two things: an interpretation as a separation logic proposition and a self-framing memory representation called a *selector*. This encapsulation is inspired by the findings of our first Steel attempt (see Section 4.2.1). Specifically, it supports the following operations:

- An interpretation as an affine predicate on memories, namely `interp (_: slprop) : mem -> prop` such that `interp p m /\ disjoint m m' ==> interp p (join m m')`. We write `fpmem (p:slprop)` for a memory validating `p`, i.e., `m:mem { interp p m }`.
- A selector type, `type_of (p:slprop) : Type`.
- A selector, `sel (p:slprop) (m:fpmem p) : type_of p`, with the property that `sel` depends only on the `p` fragment of `m`: `forall (m0:fpmem p) m1. disjoint m0 m1 ==> sel p m0 = sel p (join m0 m1)`.
- `slprop`s have all the usual connectives, including `*`, `-*`, $\wedge$, $\vee$, $\forall$, $\exists$ etc. We observe that the selectors provide a form of linear logic over memory fragments as resources. For instance, the selector type for `p * q` corresponds to a linear pair `type_of p * type_of q`, while the selector type for `p -* q` is a map from memories validating `p `star` (p -* q)` to the type of `q`. However, we do not yet exploit this connection deeply, except to build typeclass instances for `*` and `-*` and to derive from the double implication `p *--* q` a bidirectional coercion on selectors.

It is trivial to give a degenerate selector for any `slprop` simply by picking the selector type to be `unit`. But, more interesting instances can be provided by the programmer depending on their needs. For example, given a reference `r:ref a`, the interpretation of `ptr r : slprop` could be that `r` is present in a given memory; `type_of (ptr r) = a`; and `sel (ptr r) m : a` could return the value of the reference `r` in `m`.

The type `Tot t` is the type of total computations and is not particularly interesting. The main computation type is the quintuple $\{\ P \mid R\ \}\ x{:}t\ \{\ Q \mid S\ \}$, where:

- `P : slprop` is a separation logic precondition;
- `R : fppred P`, where `R` is a predicate on `P`'s selector, i.e. `fppred p = type_of p -> prop`, where the predicate is applied to `sel p` on the underlying memory;

- `x : t` binds the name `x` to the `t`-typed return value of the computation;
- **`Q`** `: slprop` is a postcondition, with `x:t` in scope;
- **`S`**`: fppost` **`P Q`** is an additional postcondition, relating the selector of **`P`** in the initial memory, to the result and the selector of **`Q`** in the final memory. It also has `x:t` in scope: `fppost (p:slprop) (q:slprop) = type_of p -> type_of q -> prop`.

SteelCore's logic also provides support for a form of quintuples, but with one major difference: instead of operating on selectors, SteelCore uses memory predicates with proof obligations that they depend only on the appropriate part of memory. Steel's quintuples with selectors are proven sound in the model of SteelCore's "raw" quintuples, and the abstraction they provide yields useful algebraic structure while freeing the user from proof-obligations on the framing of memory predicates – proof-oriented programming at work!

4.3.2. A Case Study in Proof Backend Specialization

Now that we have presented the syntax of our quintuples, we move to type-checking them. Beyond the inference rules, we discuss why this scheme makes for a good splitting of verification conditions between tactics and SMT.

VC Generation for Steel Figure 4.8 presents selected rules for typechecking Steel programs. There are 3 main ideas in the structure of the rules.

First, there are two kinds of judgments $\vdash$ and $\vdash_F$. The $\vdash$ judgment applies to terms on which no top-level occurrence of framing has been applied. The $\vdash_F$ judgment marks terms that have been framed. We use this modality to ensure that frames are applied at the leaves, to effectful function calls only, and nowhere else. The application of framing introduces metavariables to be solved and introduces equalities among framed selector terms.

Second, the rule of consequence together with a form of framing is folded into sequential composition. Both consequence and framing can also be triggered by a user annotation in a **`val`**. Although Steel's separation logic is affine, Steel aims at representing and modeling a variety of concurrent programs, including programs implemented in a language with manual memory management, such as C. To this end, we need to ensure that separation logic predicates do not implicitly disappear. As such, our VC generator uses equivalence $\dashv\vdash$ where otherwise a reader might expect to see implications ($\rightarrow$). Programmers are

$$\textsc{App}\quad \frac{\Gamma \vdash e : \mathsf{Tot}\ t \qquad \Gamma \vdash f : x{:}t \to C}{\Gamma \vdash f\ e : C[e/x]}$$

let ***fpost*** $?F\ S = \lambda(s_{p_0}, s_{f_0})\ (s_{q_1}, s_{f_1}).S\ s_{p_0}\ s_{q_1} \wedge$ ***seleq*** $?F\ s_{f_0}\ s_{f_1}$

$$\textsc{Frame}\quad \frac{\Gamma \vdash e : \{\ P \mid R\ \}\ y{:}t\ \{\ Q \mid S\ \}}{\Gamma \vdash_F e : \{\ P * ?F \mid \lambda(s_{p_0}, s_{f_0}).R\ s_{p_0}\ \}\ y{:}t\ \{\ Q * ?F \mid \textbf{\textit{fpost}}\ ?F\ S\ \}}$$

let ***pre*** $\chi\ R_1\ S_1\ R_2\ ?a\ ?b = \lambda s_{p_1}.\ R_1\ s_{p_1} \wedge \forall x\ s_{p_2}.\ ?a \wedge (S_1[x/y]\ s_{p_1}\ (\chi\ s_{p_2}) \Longrightarrow R_2\ s_{p_2} \wedge \forall z.\ ?b)$
let ***post*** $\chi_1\ \chi_2\ S_1\ S_2 = \lambda s_{p_1}\ s_q.\ \exists x\ s_{p_2}.\ S_1\ s_{p_1}\ s_{p_2} \wedge S_2\ (\chi_1\ s_{p_2})\ (\chi_2\ s_q)$

$$\textsc{Bind}\quad \frac{\begin{array}{c}\Gamma \vdash_F e_1 : \{\ P_1 \mid R_1\ \}\ y{:}t_1\ \{\ Q_1 \mid S_1\ \} \qquad \Gamma, x{:}t_1 \vdash_F e_2 : \{\ P_2 \mid R_2\ \}\ z{:}t_2\ \{\ Q_2 \mid S_2\ \} \\ \Gamma, x{:}t_1, ?a \vDash_{tac} Q_1[x/y] \mathrel{*\!\!-\!\!*} P_2 : \chi_1 \qquad \Gamma, x{:}t_1, z{:}t_2, ?b \vDash_{tac} Q_2 \mathrel{*\!\!-\!\!*} ?Q : \chi_2 \qquad x \notin FV(t_2, ?Q)\end{array}}{\Gamma \vdash_F \mathsf{bind}\ e_1\ x.e_2 : \{\ P_1 \mid \textbf{\textit{pre}}\ \chi_1\ R_1\ S_1\ R_2\ ?a\ ?b\ \}\ z{:}t_2\ \{\ ?Q \mid \textbf{\textit{post}}\ \chi_1\ \chi_2\ S_1\ S_2\ \}}$$

$$\textsc{Val}\quad \frac{\begin{array}{c}\Gamma, x : t_1 \vdash_F e : \{\ P' \mid R'\ \}\ y{:}t_2\ \{\ Q' \mid S'\ \} \\ \Gamma, x{:}t_1, ?a \vDash_{tac} P \mathrel{*\!\!-\!\!*} P' : \chi_p \qquad \Gamma, x{:}t_1, y{:}t_2, ?b \vDash_{tac} Q' \mathrel{*\!\!-\!\!*} Q : \chi_q \\ \Gamma \vDash_{smt} \forall x\ s_p.\ (R\ s_p \Longrightarrow ?a \wedge R'\ (\chi_p\ s_p)) \wedge (\forall y\ s_q.\ S'\ (\chi_p\ s_p)\ (\chi_q\ s_q) \Longrightarrow ?b \wedge S\ s_p\ s_q)\end{array}}{\Gamma \vdash \mathsf{val}\ f\ (x : t_1)\ :\ \{\ P \mid R\ \}\ y{:}t_2\ \{\ Q \mid S\ \}\ =\ e}$$

Figure 4.8: Core rules of Steel's type and effect system

expected to explicitly *drop* separation logic predicates by either freeing memory or calling ghost functions to drop ghost resources.

Finally, the proof obligations corresponding to the VCs in the rules appear in the premises in two forms, $\vDash_{tac}$ and $\vDash_{smt}$. The former involves solving separation logic goals using a tactic, which can produce auxiliary propositional goals to interact with SMT. The latter are SMT-encodeable goals – all non-separation logic reasoning is collected in the other rules eventually dispatched to SMT at the use of consequence triggered by a user annotation.

We now describe each of the rules in turn.

App This is a straightforward dependent function application rule. F* internal syntax is already desugared into a monadic form, so we need only consider the case where both the function f and the argument e are total terms. Of course, the application may itself have an effect, depending on C. The important aspect of this rule is that it is a $\vdash$ judgment, indicating that this is a raw application – no frame has been added.

Frame This rule introduces a frame. Its premise requires a $\vdash$ judgment to ensure that no repeated frames are added, while the conclusion is, of course, in $\vdash_F$, since a frame has just been applied. The rule involves picking a fresh metavariable $?F$ and framing it across the pre- and postconditions. The effect of framing of the memory postcondition S is particularly interesting: we strengthen the postcondition with $seleq\ ?F\ s_{f_0}\ s_{f_1}$, which is equivalent to $sel\ ?F\ s_{f_0} = sel\ ?F\ s_{f_1}$. We'll present this predicate in detail in a later paragraph.

Bind The most interesting rule is BIND, with several subtle elements. First, in order to sequentially compose e_1 and e_2, in the first two premises we require $\vdash_F$ judgments, to ensure that those computations have already been framed. The third premise encodes an application of consequence, to relate the `slprop`-postcondition Q_1 of e_1 to the slprop-precondition P_2 of e_2. Strictly speaking, we do not need a double implication here, but we generate equivalence constraints to ensure that our constraint solving heuristics do not implicitly drop resources. The premise $\Gamma, x{:}t_1, ?a \vDash_{tac} Q_1[x/y] \mathrel{-\!\!*} P_2 : \chi_1$ is a VC that is discharged by a tactic and, importantly, $?a$ is a propositional metavariable that the tactic must solve. For example, in order to prove $\Gamma, x{:}t_1, ?a \vDash_{tac} (r{-}{>}\,u) \mathrel{-\!\!*} (r{-}{>}\,v)$, where the interpretation of $r{-}{>}\,u$ is that the reference r points to u, a tactic could instantiate $?a := (u = v)$, i.e., the tactic is free to pick a hypothesis $?a$ under which the entailment is true. The fourth premise is similar, representing a use of consequence relating the postcondition of e_2 to a freshly picked metavariable $?Q$ for the entire postcondition, again not dropping resources implicitly. A technicality is the use of the selector

coercions χ_1, χ_2 witnessing the equivalences, which are needed to ensure that the generated pre- and postconditions are well-typed. Notice that this rule does not have an SMT proof obligation. Instead, we gather in the precondition the initial precondition R_1 and the relation between the intermediate post- and preconditions, S_1 and R_2. Importantly, we also include the tactic-computed hypotheses $?a$ and $?b$, enabling facts to be proved by the SMT solver to be used in the separation logic tactic. Finally, in the postcondition, we gather the intermediate and final postconditions.

Val The last rule checks that the inferred computation type for a Steel program matches a user-provided annotation, and is similar to most elements of BIND. As shown by the use of the entailment $\vdash_F$, it requires its premise to be framed. The next two premises are tactic VCs for relating the `slprop`-pre- and postconditions, with the same flavor as before, allowing the tactic to abduct a hypothesis under which the goal is validated. Finally, the last premise is an SMT goal, which includes the freshly abducted hypotheses, and a rule of consequence relating the annotated pre- and postconditon to what was computed. Annotated computation types are considered to not have any implicit frames, hence the use of $\vdash$ in the conclusion.

As an example, typechecking the `swap` program presented in Figure 4.6 proceeds as follows: The APP rule is applied to each of the `read` and `write` function applications. Each application of APP is followed by an application of FRAME; this enables the composition of the function applications using the BIND rule, whose premises require $\vdash_F$ judgments. Finally, an application of the OCAMLVal rule ensures that the annotated **`Steel`** computation type is admissible for this `swap` program.

We prove in the supplement that the addition of the $\vdash_F$ modalities and the removal of a nondeterministic frame and consequence rule do not compromise completeness – we can still build the same derivations, but with the additional structure, we have set the stage for tactics to focus on efficiently solving `slprop` goals, while building carefully crafted SMT-friendly VCs that can be fed as is to F*'s existing, heavily used SMT backend.

Why it Works: Proof-Oriented Programming In the introduction of this section, we claimed that prior attempts at using a WP-based VC generator for separation logic in F* did not scale. Here, we discuss some reasons why, and why the design we present here fares better. As a general remark, recall that we want the trusted computed base (TCB) of Steel to be the same as SteelCore, i.e., we

trust F* and its TCB, but nothing more. As such, our considerations for the scalability of one design over another will be based, in part, on the difficulty of writing efficient, untrusted tactics to solve various kinds of goals. Further, we aim for a Steel verifier to process an entire procedure in a single go and respond in no more than a few seconds or risk losing the user's attention. In contrast, in fully interactive verifiers, users analyze just a few commands at a time and requirements on interactive performance may be somewhat less demanding.

Separation logic provides a modular way to reason about memory, but properties about memory are only one of several concerns when proving a program. VCs for programs in F* contain many other elements: exhaustiveness checks for case analysis, refinement subtyping checks, termination checks, hypotheses encoding the definitions of let-bound names, and several other facts. In many existing F* developments a VC for a single procedure can contain several thousand logical connectives and the VC itself includes arbitrary pure F* terms. Martínez *et al.* [8]'s tactics for separation logic process this large term, applying verifiable but slow proof steps just to traverse the formula – think repeated application of inspecting the head symbol of the goal, introducing a binder, splitting a conjunction, introducing an existential variable – even these simple steps are not cheap, since they incur a call to the unifier on very large terms – until, finally an `slprop`-specific part of a VC is found, split from the rest and solved, while the rest of the VC is rewritten into propositional form and fed to the SMT solver. Although F*'s relatively fresh and unoptimized tactic system bears some of the blame, tactics like this are inherently inefficient. Anecdotally, in conversations with some Iris users, we are told that running its WP-computations on large terms would follow a similar strategy to Martínez *et al.*'s tactics and can also be quite slow. Instead, high-performance tactics usually make use of techniques like proof-by-reflection [66], but a reflective tactic for processing WP-based VCs is hard, since one would need to reflect the entire abstract syntax of pure F* terms and write certified transformations over it – effectively building a certified solver for separation logic.

A proof-oriented programming mindset suggests that producing a large unstructured VC and trying to write tactics to recover structure from it is the wrong way to go about things. Instead, our proof rules are designed to produce VCs that have the right structure from the start, separating `slprop` reasoning and other VCs by construction. The expensive unification-based tactics to process large VCs are no longer needed. We only need to run tactics

on very specific, well-identified sub-goals and the large SMT goals can be fed as is by F* to the SMT solver, once the tactics have completed.

Our tactics that focus on `slprop` implications are efficient because we use proof-by-reflection. Rather than reflect the entire syntax of F*, we only reflect the `slprop` skeleton of a term, and then can use certified, natively compiled decision procedures for rewriting in commutative monoids and AC-matching to (partially) decide `slprop` equivalence and solve for frames. What calls we make to the unifier are only on relatively small terms.

Correspondence to Our Implementation The two judgments $\vdash$ and $\vdash_F$ correspond to two new user-defined computation types in F*, namely **`Steel`** and **`SteelF`**. F*'s effect system provides hooks to allow us to elaborate terms written in direct style, `let x = e in e'` to `bind_M_M' [[e]] (fun x -> [[e']])` when `e` elaborates to `[[e]]` with a computation type whose head constructor is **`M`**, and when the elaboration `[[e']]` has a type with a head constructor **`M'`**. This allows us to compose, say, un-framed **`Steel`** computations with framed **`SteelF`** computations by first applying the frame around the first computation and then calling the Bind rule. As such, we provide 4 binds for each of the combinations, rather than a single bind and a factored frame. The precise details of how this works in F* are beyond the scope of this paper – Rastogi *et al.* [67] describe the facilities offered by F*'s effect system which we use in this work.

Steel has two other kinds of computation types, for atomic computations and for ghost (proof-only) computation steps. We apply the same recipe to generate VCs for them, inserting frames at the leaves, and including consequence and framing in copies of BIND and VAL used for these kinds of computations. Ultimately, we have six computation types, three of which are user-facing: **`Steel`**, **`SteelAtomic`** and **`SteelGhost`**. Behind the scenes, each of these has a framed counterpart introduced by the elaborator and eliminated whenever the user adds an annotation. These computation types are ordered in an effect hierarchy, allowing smooth interoperation between different kinds of computations; **`SteelAtomic`** computations are implicitly lifted to **`Steel`** computations when needed, while **`SteelGhost`** can be used transparently as either **`SteelAtomic`** or **`Steel`**.

An SMT-Friendly Encoding of Selectors We prove the soundness of our quintuples with selectors by reducing them to raw quintuples in SteelCore. In Steel-

Core quintuples $\{\ P \mid R\ \}\ x{:}t\ \{\ Q \mid S\ \}_{raw}$, we have `R:mempred P` and `S:mempost P Q`, capturing that `R` and `S` depend only on the `P` and `Q` fragments of the initial and final memories, respectively.

```
mempred (p:slprop) =
  f:(fpmem p -> prop){
    forall (m:fpmem p) (m':mem{disjoint m m'}).
      f m <==> f (join m m')}
mempost (p:slprop) (q:slprop) =
  f:(fpmem p -> mempred q){
    forall (m0:fpmem p) (m0':mem{disjoint m0 m0'})
      (m1:fpmem q). f m0 m1 <==> f (join m0 m0') m1}
```

Thus every user annotation in SteelCore's raw quintuples comes with an obligation to show that the `R` and `S` terms depend only on their specified footprint—these relational proofs on specifications can be overwhelming, and require reasoning about disjoint and joined memories, breaking the abstractions that separation logic offers. In comparison, selector predicates are self-framing by construction: the predicates R and S can only access the selectors of P and Q instead of the underlying memory, which are themselves self-framing. By defining selector predicates as an abstraction on top of the SteelCore program logic, we thus hide the complexity of the self-framing property from both the user and the SMT solver.

To preserve the modularity inherent to separation logic reasoning when using selector predicates, the postcondition of the FRAME rule previously presented contains the proposition $seleq\ ?F\ s_{f_0}\ s_{f_1}$, capturing that $sel\ ?F\ s_{f_0} = sel\ ?F\ s_{f_1}$.

Using this predicate, the SMT solver can derive that the selector of any `slprop` contained in the frame is the same in the initial and final memories, leveraging the fact that, for any `m:fpmem (p * q)`, `sel (p * q) m = (sel p m, sel q m)`. But as the size of the frame grows, this becomes expensive; the SMT solver needs to deconstruct and reconstruct increasingly large tuples.

Instead we encode `seleq` as the conjunction of equalities of the *atomic* `slprop`s selectors contained in the frame, where an *atomic* `slprop` does not contain a `*`. For instance, `p` and `q` are the atomic `slprop`s contained in `p * q` Our observation is that most specifications pertain to atomic `slprop`s; the swap function presented in the introduction for instance is specified using the selectors of `ptr r1` and `ptr r2`, instead of `(ptr r1 * ptr r2)`.

Once the frame has been resolved using the approach presented in Section 4 of the related publication [54], generating these equalities is straightforward using metaprogramming; we can flatten the frame according to the star operator, and generate the conjunction of equalities to pass to the SMT solver.

Selectors can alleviate the need for existentially quantified ghost variables; the value stored in a reference for instance can be expressed as a selector, decluttering specifications. But, not all `slprop`s have meaningful selectors, nor do we expect that they should. For example, when using constructions like PCMs to encode sharing disciplines, it is not always possible to define a selector that returns the partial knowledge of a resource. However, when applicable, selectors can significantly simplify specifications and proofs.

Evaluation This simplification can be illustrated by the two equivalent implementations of Figure 4.6. Moreover, we evaluated the Steel framework on several small-sized examples such as self-balancing trees and concurrent data structures. The details of these examples can be found in Section 5 of the related publication [54].

Summary of personal contributions I proposed an intermediate design and implementation for selectors and framing tactic in an early version of the Steel framework, on which I worked from August to October 2019. In particular, I implemented the support for selectors in an intermediate version of the RST Steel effect, as well as first version of the framing tactic that sorted the goals waiting to be solved through a syntactic check, and fed them to the unifier in the correct order. Lastly, I had minor contributions on the implementation of section 5.2 of the paper for the self-balancing trees, which was refined and rewritten by other Steel authors in the published version of the paper.

Conclusion

Starting with the analysis of the weaknesses of the Low* domain-specific languages, this chapter reflects a personal journey into the depths of stateful program verification. When designing domain-specific languages like Low* or Steel, the proof obligations that the programs will generate are more important that the expressiveness of the language itself. The quest of stateful program verification to model programs like Figure 4.1 and offer a reasonable level of proof automation for complex properties is not over.

However, we believe that progress in this area will come with the diversification of the proof backends used. The examples of Low⋆ and Iris (with its applications) have shown that tactics and SMT automation each solve one aspect of the program verification problem: tactics are adapted to the algebraic structure of proof obligation coming from separation logic, while SMT scales for proving functional equivalence properties for large programs like HACL⋆ and LibSignal⋆ (see Chapter 2).

If the idea of splitting verification conditions and dispatching them to several backends is not new in program verification (see Viper [68]), Steel chooses a principled way both in its foundations (inspired by Iris) and its implementation, sound by construction because completely mechanized within the F⋆ proof assistant.

Of course, the new Steel framework needs to be evaluated more thoroughly on real-world software to prove its effectiveness. However, once can note that the development of this new framework directly came from the observation of the shortcoming of the previous framework, Low⋆. These shortcomings (see Section 4.1.1) can only be noticed when attempting large-scale applications of the framework, largely beyond the traditional program verification evaluations on hundred-lines-of-code-long functions and classical data structure implementations.

Hence, this chapter illustrates the claims of Section 1.1.1 in the introduction of this dissertation. We believe that effective progress in program verification should embed the framework developers in a loop that goes from real-world applications to theoretical foundations. That loop has already been experimented successfully in high-profile projects like seL4 [69], and has somewhat driven the design and implementation of the F⋆ proof assistant.

Nevertheless, opportunities for applying the current state of the art of stateful program verification to real-world applications are becoming scarce, as a lot of verification-friendly domains like kernel programming or cryptography are reaching maturity. What is the next frontier for program verification? If more general systems programming naturally comes up as an attractive candidate, the techniques need to scale up to million lines of code, like the Coccinelle [70]–[72] tool for semantic grep on device driver code.

Climbing this giant scaling step will take years of work and is completely outside of the scope of an individual effort. Steel itself is a collaborative effort involving a little less than a dozen people. To continue exploring the idea of proof-oriented domain-specific language as part of our individual contribution, we decided to leave Steel and F⋆, and focus on expanding program verification

to a somewhat new area of application: legal expert systems. This will be the subject of Chapter 5 and Chapter 6.

References

[1] J. C. Reynolds, "Separation logic: a logic for shared mutable data structures", in *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, ser. LICS '02, USA: IEEE Computer Society, 2002, pp. 55–74, ISBN: 0769514839.

[2] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer, "Iris from the ground up: a modular foundation for higher-order concurrent separation logic", *Journal of Functional Programming*, vol. 28, 2018.

[3] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hritcu, K. Bhargavan, C. Fournet, and N. Swamy, "Verified low-level programming embedded in F*", *PACMPL*, vol. 1, no. ICFP, 17:1–17:29, Sep. 2017. DOI: 10.1145/3110261. [Online]. Available: http://arxiv.org/abs/1703.00053.

[4] J. Smans, B. Jacobs, and F. Piessens, "Implicit dynamic frames", *ACM Trans. Program. Lang. Syst.*, vol. 34, no. 1, May 2012, ISSN: 0164-0925. DOI: 10.1145/2160910.2160911. [Online]. Available: https://doi.org/10.1145/2160910.2160911.

[5] S. Blazy and X. Leroy, "Mechanized semantics for the Clight subset of the C language", *Journal of Automated Reasoning*, vol. 43, no. 3, pp. 263–288, 2009.

[6] E. Palmgren, "On universes in type theory", *Twenty five years of constructive type theory*, pp. 191–204, 1998.

[7] P. Martin-Löf, "An intuitionistic theory of types: predicative part", in *Studies in Logic and the Foundations of Mathematics*, vol. 80, Elsevier, 1975, pp. 73–118.

[8] G. Martinez, D. Ahman, V. Dumitrescu, N. Giannarakis, C. Hawblitzel, C. Hriţcu, M. Narasimhamurthy, Z. Paraskevopoulou, C. Pit-Claudel, J. Protzenko, *et al.*, "Meta-F*: proof automation with SMT, tactics, and metaprograms", in *European Symposium on Programming*, Springer, Cham, 2019, pp. 30–59.

[9] L. De Moura and N. Bjørner, "Efficient E-matching for SMT solvers", in *International Conference on Automated Deduction*, Springer, 2007, pp. 183–198.

[10] D. Jovanović and L. De Moura, "Solving non-linear arithmetic", in *International Joint Conference on Automated Reasoning*, Springer, 2012, pp. 339–354.

[11] 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 *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1465–1482.

[12] P. O'Hearn, J. Reynolds, and H. Yang, "Local reasoning about programs that alter data structures", in *International Workshop on Computer Science Logic*, Springer, 2001, pp. 1–19.

[13] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: a verification infrastructure for permission-based reasoning", in *International Conference on Verification, Model Checking, and Abstract Interpretation*, Springer, 2016, pp. 41–62.

[14] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: a modular reusable verifier for object-oriented programs", in *International Symposium on Formal Methods for Components and Objects*, Springer, 2005, pp. 364–387.

[15] M. Schwerhoff and A. J. Summers, "Lightweight support for magic wands in an automatic verifier", in *European Conference on Object-Oriented Programming (ECOOP)*, J. T. Boyland, Ed., ser. LIPIcs, vol. 37, Schloss Dagstuhl, 2015, pp. 614–638.

[16] B. Jacobs and F. Piessens, "The VeriFast program verifier", Technical Report CW-520, Department of Computer Science, Katholieke . . ., Tech. Rep., 2008.

[17] J.-C. Filliâtre and A. Paskevich, "Why3—where programs meet provers", in *European symposium on programming*, Springer, 2013, pp. 125–128.

[18] J.-C. Filliâtre, L. Gondelman, and A. Paskevich, "A pragmatic type system for deductive verification", Tech. Rep., 2016.

[19] A. Tafat and C. Marché, "Binary heaps formally verified in Why3", INRIA, Research Report 7780, Oct. 2011, http://hal.inria.fr/inria-00636083/en/.

[20] M. Pereira and A. Ravara, *Cameleer: a deductive verification tool for ocaml*, 2021. arXiv: 2104.11050 [cs.LO].

[21] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer, "Iris: monoids and invariants as an orthogonal basis for concurrent reasoning", *ACM SIGPLAN Notices*, vol. 50, no. 1, pp. 637–650, 2015.

[22] R. Krebbers, R. Jung, A. Bizjak, J.-H. Jourdan, D. Dreyer, and L. Birkedal, "The essence of higher-order concurrent separation logic", in *European Symposium on Programming*, Springer, 2017, pp. 696–723.

[23] R. Krebbers, A. Timany, and L. Birkedal, "Interactive proofs in higher-order concurrent separation logic", in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017, pp. 205–217.

[24] 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*, vol. 2, no. POPL, pp. 1–34, 2017.

[25] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich, "Verifying concurrent, crash-safe systems with perennial", in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 243–258.

[26] T. Chajed, J. Tassarotti, M. Theng, R. Jung, M. F. Kaashoek, and N. Zeldovich, "Gojournal: a verified, concurrent, crash-safe journaling system", in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021, pp. 423–439.

[27] P. E. de Vilhena, F. Pottier, and J.-H. Jourdan, "Spy game: verifying a local generic solver in iris", *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–28, 2019.

[28] G. Mével, J.-H. Jourdan, and F. Pottier, "Cosmo: a concurrent separation logic for multicore ocaml", *Proceedings of the ACM on Programming Languages*, vol. 4, no. ICFP, pp. 1–29, 2020.

[29] G. Mével and J.-H. Jourdan, "Formal verification of a concurrent bounded queue in a weak memory model", in *Proceedings of the 26th ACM SIGPLAN international conference on Functional programming*, 2021.

[30] J. Protzenko, *The KreMLin user manual*, 2019. [Online]. Available: https://fstarlang.github.io/lowstar/html/LinkedList4.html.

[31] M. J. Parkinson and A. J. Summers, "The relationship between separation logic and implicit dynamic frames", in *European Symposium on Programming*, Springer, 2011, pp. 439–458.

[32] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang, "Views: compositional reasoning for concurrent programs", in *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*, 2013, pp. 287–300.

[33] K. Bhargavan, C. Fournet, and M. Kohlweiss, "Mitls: verifying protocol implementations against real-world attacks", *IEEE Security & Privacy*, vol. 14, no. 6, pp. 18–25, 2016.

[34] N. Swamy, A. Rastogi, A. Fromherz, D. Merigoux, D. Ahman, and G. Martinez, "Steelcore: an extensible concurrent separation logic for effectful dependently typed programs", *Proc. ACM Program. Lang.*, vol. 4, no. ICFP, Aug. 2020. DOI: 10.1145/3409003. [Online]. Available: https://doi.org/10.1145/3409003.

[35] The Coq Development Team, *The Coq Proof Assistant Reference Manual, version 8.7*, Oct. 2017. [Online]. Available: http://coq.inria.fr.

[36] P. W. O'Hearn, "Resources, concurrency and local reasoning", in *CONCUR 2004 - Concurrency Theory*, P. Gardner and N. Yoshida, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 49–67, ISBN: 978-3-540-28644-8.

[37] M. Krogh-Jespersen, A. Timany, M. E. Ohlenbusch, S. O. Gregersen, and L. Birkedal, "Aneris: a mechanised logic for modular reasoning about distributed systems", *Submitted for publication*, 2019.

[38] J. K. Hinrichsen, J. Bengtson, and R. Krebbers, "Actris: session-type based reasoning in separation logic", *Proc. ACM Program. Lang.*, vol. 4, no. POPL, Dec. 2019. DOI: 10.1145/3371074. [Online]. Available: https://doi.org/10.1145/3371074.

[39] A. Nanevski, J. G. Morrisett, and L. Birkedal, "Hoare type theory, polymorphism and separation", *J. Funct. Program.*, vol. 18, no. 5-6, pp. 865–911, 2008. [Online]. Available: http://ynot.cs.harvard.edu/papers/jfpsep07.pdf.

[40] A. Nanevski, R. Ley-Wild, I. Sergey, and G. A. Delbianco, "Communicating state transition systems for fine-grained concurrent resources", in *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, 2014, pp. 290–310. DOI: 10.1007/978-3-642-54833-8_16. [Online]. Available: https://doi.org/10.1007/978-3-642-54833-8%5C_16.

[41] A. Nanevski, A. Banerjee, G. A. Delbianco, and I. Fábregas, "Specifying concurrent programs in separation logic: morphisms and simulations", *PACMPL*, vol. 3, no. OOPSLA, 161:1–161:30, 2019. DOI: 10.1145/3360587. [Online]. Available: https://doi.org/10.1145/3360587.

[42] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, *et al.*, "Dependent types and multi-monadic effects in F*", in *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 256–270.

[43] D. Ahman, C. Fournet, C. Hritcu, K. Maillard, A. Rastogi, and N. Swamy, "Recalling a witness: foundations and applications of monotonic state", *PACMPL*, vol. 2, no. POPL, 65:1–65:30, Jan. 2018. [Online]. Available: https://arxiv.org/abs/1707.02466.

[44] P. Hancock and A. Setzer, "Interactive programs in dependent type theory", in *Computer Science Logic*, P. G. Clote and H. Schwichtenberg, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 317–331, ISBN: 978-3-540-44622-4.

[45] W. Swierstra, "Data types à la carte", *Journal of Functional Programming*, vol. 18, no. 4, pp. 423–436, 2008. DOI: 10.1017/S0956796808006758.

[46] O. Kiselyov and H. Ishii, "Freer monads, more extensible effects", in *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, ser. Haskell '15, Vancouver, BC, Canada: Association for Computing Machinery, 2015, pp. 94–105, ISBN: 9781450338080. DOI: 10.1145/2804302.

2804319. [Online]. Available: https://doi.org/10.1145/2804302.2804319.

[47] L.-y. Xia, Y. Zakowski, P. He, C.-K. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic, "Interaction trees: representing recursive and impure programs in coq", *Proc. ACM Program. Lang.*, vol. 4, no. POPL, Dec. 2019. DOI: 10.1145/3371119. [Online]. Available: https://doi.org/10.1145/3371119.

[48] C. Bach Poulsen, A. Rouvoet, A. Tolmach, R. Krebbers, and E. Visser, "Intrinsically-typed definitional interpreters for imperative languages", *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2018. DOI: 10.1145/3158104. [Online]. Available: https://doi.org/10.1145/3158104.

[49] S. Brookes, "A semantics for concurrent separation logic", in *CONCUR 2004 - Concurrency Theory*, P. Gardner and N. Yoshida, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 16–34, ISBN: 978-3-540-28644-8.

[50] A. Buisse, L. Birkedal, and K. Støvring, "Step-indexed kripke model of separation logic for storable locks", *Electronic Notes in Theoretical Computer Science*, vol. 276, pp. 121–143, 2011, Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII), ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2011.09.018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066111001095.

[51] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv, "Local reasoning for storable locks and threads", in *Proceedings of the 5th Asian Conference on Programming Languages and Systems*, ser. APLAS'07, Singapore: Springer-Verlag, 2007, pp. 19–37, ISBN: 3540766367.

[52] A. Hobor, A. W. Appel, and F. Z. Nardelli, "Oracle semantics for concurrent separation logic", in *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, S. Drossopoulou, Ed., ser. Lecture Notes in Computer Science, vol. 4960, Springer, 2008, pp. 353–367. DOI: 10.1007/978-3-540-78739-6_27. [Online]. Available: https://doi.org/10.1007/978-3-540-78739-6%5C_27.

[53] M. Dodds, S. Jagannathan, M. J. Parkinson, K. Svendsen, and L. Birkedal, "Verifying custom synchronization constructs using higher-order separation logic", *ACM Trans. Program. Lang. Syst.*, vol. 38, no. 2, Jan. 2016, ISSN: 0164-0925. DOI: 10.1145/2818638. [Online]. Available: https://doi.org/10.1145/2818638.

[54] A. Fromherz, A. Rastogi, N. Swamy, S. Gibson, G. Martinez, D. Merigoux, and T. Ramananandro, "Steel: proof-oriented programming in a dependently typed concurrent separation logic", *Proc. ACM Program. Lang.*, vol. 5, no. ICFP, Aug. 2021. DOI: 10.1145/3473590. [Online]. Available: https://doi.org/10.1145/3473590.

[55] E. Brady, *Type-driven Development With Idris*. Manning, 2016, ISBN: 9781617293023. [Online]. Available: http://www.worldcat.org/isbn/9781617293023.

[56] K. R. M. Leino, "Dafny: an automatic program verifier for functional correctness", in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, Springer, 2010, pp. 348–370.

[57] K. R. M. Leino, P. Müller, and J. Smans, "Verification of concurrent programs with Chalice", in *Foundations of Security Analysis and Design V*, Springer, 2009, pp. 195–222.

[58] D. Kapur and P. Narendran, "Matching, unification and complexity", *SIGSAM Bull.*, vol. 21, no. 4, pp. 6–9, Nov. 1987, ISSN: 0163-5824. DOI: 10.1145/36330.36332. [Online]. Available: https://doi.org/10.1145/36330.36332.

[59] K. R. M. Leino and M. Moskal, "Usable auto-active verification", in *Usable Verification Workshop. http://fm. csl. sri. com/UV10*, Citeseer, 2010.

[60] S. S. Ishtiaq and P. W. O'Hearn, "BI as an assertion language for mutable data structures", in *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, 2001, pp. 14–26.

[61] A. Hobor and J. Villard, "The ramifications of sharing in data structures", in *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '13, Rome, Italy: Association for Computing Machinery, 2013, pp. 523–536, ISBN:

9781450318327. DOI: 10.1145/2429069.2429131. [Online]. Available: https://doi.org/10.1145/2429069.2429131.

[62] N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits, "Verifying higher-order programs with the dijkstra monad", in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13, Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 387–398, ISBN: 9781450320146. DOI: 10.1145/2491956.2491978. [Online]. Available: https://doi.org/10.1145/2491956.2491978.

[63] L. De Moura and N. Bjørner, "Z3: an efficient SMT solver", in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337–340.

[64] J. Brotherston, N. Gorogiannis, and R. L. Petersen, "A generic cyclic theorem prover", in *Programming Languages and Systems*, R. Jhala and A. Igarashi, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 350–367.

[65] R. Iosif, A. Rogalewicz, and T. Vojnar, "Deciding entailments in inductive separation logic with tree automata", in *Automated Technology for Verification and Analysis*, F. Cassez and J.-F. Raskin, Eds., Cham: Springer International Publishing, 2014, pp. 201–218.

[66] G. Gonthier, A. Mahboubi, and E. Tassi, "A Small Scale Reflection Extension for the Coq system", Inria Saclay Ile de France, Research Report RR-6455, 2016. [Online]. Available: https://hal.inria.fr/inria-00258384.

[67] A. Rastogi, G. Martínez, A. Fromherz, T. Ramananandro, and N. Swamy, *Programming and proving with indexed effects*, In submission, Jul. 2021. [Online]. Available: fstar-lang.org/papers/indexedeffects.

[68] E. Mullen, D. Zuniga, Z. Tatlock, and D. Grossman, "Verified peephole optimizations for CompCert", in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 448–461.

[69] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, *et al.*, "Sel4: formal verification of an OS kernel", in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 207–220.

[70] Y. Padioleau, J. L. Lawall, and G. Muller, “Smpl: a domain-specific language for specifying collateral evolutions in linux device drivers”, *Electronic Notes in Theoretical Computer Science*, vol. 166, pp. 47–62, 2007.

[71] J. Lawall and G. Muller, “Coccinelle: 10 years of automated evolution in the linux kernel”, in *2018 USENIX Annual Technical Conference (USENIXATC 18)*, 2018, pp. 601–614.

[72] D. Merigoux, “Semantic Patch Inference”, M.S. thesis, École polytechnique, Jul. 2016. [Online]. Available: `https://hal.inria.fr/hal-02936287`.

Part II.

High-Assurance Legal Expert Systems

5. MLANG: A Modern Compiler for the French Tax Code

Contents

Abstract

In the previous chapters, we have explored how domain-specific language design can help bring formal methods to the table in the event of a rewrite of critical applications in a new language. However, not all software gets re-written, and some legacy implementation seem to never want to die. A classic example of such legacy applications is banking software, which notoriously still runs on COBOL [1]–[3].

In this chapter, we make the case for domain-specific languages as gateways to efficient modernization and formalization of legacy system. Luckily, the French tax administration chose in 1988 to use a domain-specific language called M for its income tax computation algorithm. Thanks to this fortuitous decision, we were able to bring the aging system to the state of the art.

Il faut reconnaître à la pratique une logique qui n'est pas celle de la logique pour éviter de lui demander plus de logique qu'elle n'en peut donner et de se condamner ainsi soit à lui extorquer des incohérences, soit à lui imposer une cohérence forcée.

(Pierre Bourdieu,
Le sens pratique*,1980)*

Our new Constitution is now established, and has an appearance that promises permanency; but in this world nothing can be said to be certain, except death and taxes.

(Benjamin Franklin, in a letter to Jean-Baptiste Le Roy, 1789)

This chapter is based upon the following publication:

D. Merigoux, R. Monat, and J. Protzenko, "A modern compiler for the french tax code", in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2021, Virtual, Republic of Korea: Association for Computing Machinery, 2021, pp. 71–82, ISBN: 9781450383257. DOI: 10.1145/3446804.3446850. [Online]. Available: https://doi.org/10.1145/3446804.3446850

My personal contribution to this publication has been the retro-engineering of the M language, and half of the implementation of MLANG*. I also led the project and the discussions with the DGFiP.*

5.1. The Challenge of Everlasting Maintenance

In this section, we discuss the general challenges associated with long-term software maintenance, and how domain-specific languages can help. Then, we introduce the case study for the rest of the chapter: the French income tax computation algorithm.

5.1.1. The Language Obsolescence Trap

No[1] software, unlike diamonds, lasts forever. But some, because of the crucial objectives they accomplish, are intended to be used for an indefinite period of time. In the 1960s and 1970s, banking institutions around the world computerized their transaction management systems using the COBOL programming language. Sixty years later, COBOL is no longer taught at universities and is widely considered obsolete, but banks still use it to run the global economy [5]. The choice of a programming language is necessary to start a project. But even when choosing a language that is popular today, one is not immune to the computer *zeitgeist* that has made many languages, frameworks and technologies obsolete.

Considered on time scales of the duration of several careers, it is unlikely that a technology of today will still be taught at university in 70 or 80 years.

[1] Section 5.1.1 is largely inspired by a blog post that I originally wrote in French. The English translation credits should go to Dhruv Sharma.

And as the original programmers retire, it becomes impossible to rely on the labour market to provide a skilled, job-ready workforce.

What strategies should be adopted to guard against obsolescence in such long-term projects? A first solution is available to very large organizations: the creation of a generalist in-house language. This is the case of the Ada [6] language, created in the 1980s by the US Department of Defense (DoD). By imposing the use of this language in all its projects, the DoD was able to create, from scratch, a demand for Ada programmers sufficient for the Ada community to reach critical mass allowing it to survive in the long term. With enough users and long term funding, Ada has been able to receive many updates and its tooling continues to adapt to modern trends in programming languages.

However, not all organizations have the size, the level of planning and funding that the U.S. Department of Defense commands. Therefore, a second strategy might be relevant for smaller projects that also have constraint of long-term code maintenance. This chapter will make the claim that, thanks to the use of a dedicated language, only two part-time PhD students are needed to retrofit a formal semantics and modernize a compiler. Compare this with the work required to port a code base. Of course, it is always easier for an IT project manager to recruit a mass of low-skilled developers than to find and retain profiles with a Masters in Computer Science, major in Compilation and/or Theory.

With an established general-purpose language such as C or Java, one only has the power to change the code to adapt and modify their software. With a dedicated language and a well-tested compiler, the language itself becomes a tool at the service of the software's evolution, and no longer a necessary evil to choose at the beginning which would inevitably turn into insurmountable levels of technical debt anyway. A dedicated language with a compiler well mastered by specialists trained in formal methods is a formidable assurance of the project's future and interoperability. The reason for this is the agility of a dedicated language that can adapt to the evolution of functional requirements as time passes. By adding language features to cover new functional requirements, it is possible to keep the code concise and to the point. By maintaining and evolving both the code and language together, we are reminded that code and language form a coherent whole.

Beware, a dedicated language can become a double-edged sword without the right skillset. The semantics of a language shouldn't be changed on the spur of the moment, because a tiny change can break the entire code base.

For this reason, compiler engineers and formal methods specialists, far from being rendered obsolete by statistical learning methods, remain and should remain for a very long time absolutely crucial. The study of programming languages, compiler theory and formal methods provide a real specific skill that is extremely useful for many critical computing projects.

In a more figurative sense, the choice of the dedicated language and its associated competence in formal methods can be compared to the choice of craftsmanship in relation to an industrial process of porting code from one language to another. Craftsmanship has a long tradition of transmitting knowledge over the long term, based on a small number of independent practitioners trained in high-level skills with relative versatility. In contrast, the industry aims to transform the production process into a fairly rigid chain of standardized tasks performed by low-skilled workers. Like other production sectors, information technology tends to follow this process of industrialization over the course of the time.

But industrialization requires critical mass and often the search for infinite growth. We believe this analogy also applies to the programming languages. In this chapter, we put this principle to application in a context particularly favorable to its success : a forever-lasting critical codebase maintained in a mid-sized public organization, that already made the choice of a domain-specific language at the inception of the project. Our contributions should demonstrate the effectiveness of our signature methodology (Section 1.3.1) for the modernization of such a legacy critical infrastructure.

5.1.2. Case study: the French Income Tax Computation

The French Tax Code is a body of legislation amounting to roughly 3,500 pages of text, defining the modalities of tax collection by the state. In particular, each new fiscal year, a new edition of the Tax Code describes in natural language how to compute the final amount of income tax (IR, for *impôt sur le revenu*) owed by each household.

As in many other tax systems around the world, this computation is quite complex. France uses a bracket system (as in, say, the US federal income tax), along with a myriad of tax credits, deductions, optional rules, state-sponsored direct aid, all of which are parameterized over the composition of the household, that is, the number of children, their respective ages, potential disabilities, and so on.

Unlike, say, the United States, the French system relies heavily on automa-

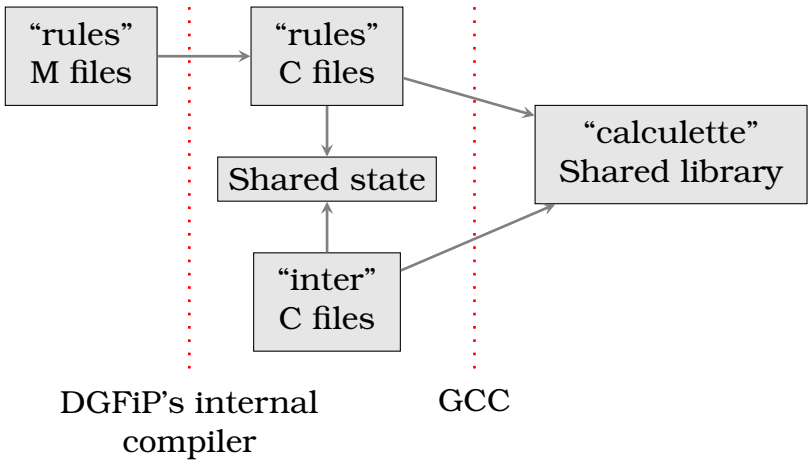

Figure 5.1: Legacy architecture

tion. During tax season, French taxpayers log in to the online tax portal, which is managed by the state. There, taxpayers are presented with online forms, generally pre-filled. If applicable, taxpayers can adjust the forms, e.g. by entering extra deductions or credits. Once the taxpayer is satisfied with the contents of the online form, they send in their return. Behind the scenes, the IR algorithm is run, and taking as input the contents of the forms, returns the final amount of tax owed. The taxpayer is then presented with the result at tax-collection time.

Naturally, the ability to independently reproduce and thus trust the IR computation performed by the DGFiP is crucial. First, taxpayers need to *understand* the result, as their own estimate may differ (explainability). Second, citizens may want to *audit* the algorithm, to ensure it faithfully implements the law (correctness). Third, a standalone, reusable implementation allows for a complete and precise *simulation* of the impacts of a tax reform, greatly improving existing efforts [7], [8] (forecasting).

Unfortunately, we are currently far from a transparent, open-source, reproducible computation. Following numerous requests (using a disposition similar to the United States' Freedom of Information Act), parts of the existing source code were published. In doing so, the public learned that *i)* the existing infrastructure is made up of various parts pieced together and that *ii)* key data required to accurately reproduce IR computations was not shared with the public.

The current, legacy architecture of the IR tax system is presented in Fig-

ure 5.1. The bulk of the tax code is described as a set of "rules" authored in M, a custom, non Turing-complete language. A total of 90,000 lines of M rules compile to 535,000 lines of C (including whitespace and comments) via a custom compiler. Rules are now mostly public [9]. Over time, the expressive power of rules turned out to be too limited to express a particular feature, known as *liquidations multiples*, which involves tax returns across different years. Lacking the expertise to extend the M language, the DGFiP added in 1995 some high-level glue code in C, known as "inter". The glue code is closer to a full-fledged language, and has a non-recursive call-graph which may call the "rules" computation multiple times with various parameters. The "inter" driver amounts to 35,000 lines of C code and has not been released.

Both "inter" and "rules" are updated every year to follow updates in the law, and as such, have been extensively modified over their 30-year operational lifespan.

Our goal is to address these shortcomings by bringing the French tax code infrastructure into the 21st century. Specifically, we wish to: *i)* reverse-engineer the unpublished parts of the DGFiP computation, so as to *ii)* provide an explainable, open-source, *correct* implementation that can be independently audited; furthermore, we wish to *iii)* modernize the compiler infrastructure, eliminating in the process any hand-written C that could not be released because of security concerns, thus enabling a host of modern applications, simulations and use-cases.

- We start with a reverse-engineered formal semantics for the M domain-specific language, along with a proof of type safety performed using the Coq [10] proof assistant (Section 5.2.1).
- To eliminate C code from the ecosystem, we extend the M language with enough capabilities to encode the logic of the high-level "inter" driver (Figure 5.1) – we dub the new design M++ (Section 5.2.2).
- To execute M/M++ programs, we introduce MLANG, a complete re-implementation which combines a reference interpreter along with an optimizing compiler that generates C and Python code (Section 5.2.2).
- We evaluate our implementation: we show how we attained 100% conformance on the legacy system's testsuite, then proceed to enable a variety of analyses and instrumentations to fuzz, measure and stress-test our new system (Section 5.3.1).

- We conclude with a plan for further modernization of the system, foreshadowing a big rewrite of the whole codebase (Section 5.3.2).

Our code is open-source and available on GitHub [11] and as an archived artifact on Zenodo [12]. We have engaged with the DGFiP, and following numerous discussions, iterations, and visits to their offices, we have been formally approved to replace the legacy compiler infrastructure with our new implementation, meaning that within a few years' time, all French tax returns should be processed using the compiler described in the present chapter.

5.2. Retrofitting a Domain-Specific Language

In this section, we apply our feature methodology of carving out a formal, executable subset of a real-world critical codebase to the DGFiP's income tax computation. Starting from the legacy domain-specific language M, we extend our efforts to a part of the code that ought to be programmed in a specific language, weren't it for historical mistakes.

5.2.1. The Formal Semantics of M

The 2018 version of the income tax computation [9] is split across 48 files, for a total of 92,000 lines of code. The code is written in M, the input language originally designed by the DGFiP. In order to understand this body of tax code, we set out to give a semantics to M.

Overview of M M programs are made up of two parts: declarations and rules. Declarations introduce input variables, intermediary variables, output variables and exceptions. Variables are either scalars or fixed-length arrays. Both variables and exceptions are annotated with a human-readable description. Variables that belong to the same section of the tax form are annotated with the same kind. Examples of kinds include "triggers tax credit", or "is advance payment". We will make use of those kinds later (Section 5.2.2) for partitioning variables, and quickly checking whether any variable of a given kind has a non-`undef` value.

Rules, on the other hand, capture the computational part of an M program; they are either variable assignments or raise-if statements.

As a first simplified example, the French tax code declares an input variable `V_0AC` for whether an individual is single (value `1`) or not (value `0`). Lacking any notion of data type or enumeration, there is no way to enforce statically that an individual cannot be married (`V_0AM`) and single (`V_0AC`) at the same time. Instead, an exception `A031` is declared, along with a human-readable description. Then, a rule raises an exception if the sum of the two variables is greater than 1. (The seemingly superfluous `+ 0` is explained later in this section.) For the sake of example, we drop irrelevant extra syntactic features, and for the sake of readability, we translate keywords and descriptions into English.

```
V_0AC : input family ... : "Checkbox : Single" type BOOLEAN ;
V_0AM : input family ... : "Checkbox : Married" type BOOLEAN ;
A031:exception :"A":"031":"00":"both married and single":"N";

if V_0AC + V_0AM + 0 > 1 then error A031 ;
```

As a second simplified example, the following M rule computes the value of a hypothetical variable `TAXBREAK`. Its value is computed from variables `CHILDRENCOUNT` (for the number of children in the household) and `TAXOWED` (for the tax owed before the break) – the assigned expression relies on a conditional and the built-in `max` function. This expression gives a better tax break to households having three or more children.

```
TAXBREAK = if (CHILDRENCOUNT+0 > 2)
  then max(MINTAXBREAK,TAXOWED * 20 / 100)
  else MINTAXBREAK endif;
```

For the rest of this paper, we abandon concrete syntax and all-caps variable names, in favor of a core calculus that faithfully models M: μM.

μM: A Core Model of M The μM core language omits variable declarations, whose main purpose is to provide a human-readable description string that relates them to the original tax form. The μM core language also eliminates syntactic sugar, such as statically bounded loops, or type aliases (e.g. `BOOLEAN`).

Finally, a particular feature of M is that rules may be provided in any order: the M language has a built-in dependency resolution feature that automatically re-orders computations (rules) and asserts that there are no loops in

variable assignments. In our own implementation (that we will describe in Section 5.2.2), we perform a topological sort; in our μM formalization, we assume that computations are already in a suitable order.

Syntax of μM We describe the syntax of μM in Figure 5.2. A program is a series of statements ("rules"). Statements are either raise-error-if, or assignments. We define two forms of assignment: one for scalars and the other for fixed-size arrays. The latter is of the form `a[X, n] := e`, where `X` is bound in `e` (the index is *always* named `X`). Using Haskell's list comprehension syntax, this is to be understood as $a := [e|X \leftarrow [0..n-1]]$.

Expressions are a combination of variables (including the special index expression `X`), values, comparisons, logic and arithmetic expressions, conditionals, calls to builtin functions, or index accesses. Most functions exhibit standard behavior on floating-point values, but M assumes the default IEEE-754 rounding mode, that is, rounding to nearest and ties to even. The detailed behavior of each function is described in Figure 5.3.

Values can be `undef`, which arises in two situations: references to variables that have not been defined (i.e. for which the entry in the tax form has been left blank) and out of bounds array accesses. All other values are IEEE-754 double-precision numbers, i.e. 64-bit floats. The earlier `BOOLEAN` type shown in the introductory example is simply an alias for a float whose value is implicitly `0` or `1`. There is no other kind of value, as a reference to an array variable is invalid. Function `present` discriminates the `undef` value from floats.

Typing μM Types in μM are either scalar or array types. M does not offer nested arrays. Therefore, typing is mostly about making sure scalars and arrays are not mixed up.

In Figure 5.4, a first judgment $\boxed{\Gamma \vdash e}$ defines expression well-formedness. It rules out references to arrays, hence enforcing that expressions have type scalar and that no values of type array can be produced. Furthermore, variables may have no assignment at all (if the underlying entry in the tax form has been left blank) but may still be referred in other rules. Rather than introduce spurious variable assignments with `undef`, we remain faithful to the very loose nature of the M language and account for references to undefined variables.

Then, $\boxed{\Gamma \vdash \langle \textit{program} \rangle \Rrightarrow \Gamma'}$ enforces well-formedness for a whole program

⟨*program*⟩ ::= ⟨*command*⟩ | ⟨*command*⟩ `;` ⟨*program*⟩

⟨*command*⟩ ::= `if` ⟨*expr*⟩ `then` ⟨*error*⟩
| ⟨*var*⟩ `:=` ⟨*expr*⟩ | ⟨*var*⟩ `[ X ;` ⟨*float*⟩ `] :=` ⟨*expr*⟩

⟨*expr*⟩ ::= ⟨*var*⟩ | `X` | ⟨*value*⟩ | ⟨*expr*⟩ ⟨*binop*⟩ ⟨*expr*⟩
| ⟨*unop*⟩ ⟨*expr*⟩ | `if` ⟨*expr*⟩ `then` ⟨*expr*⟩ `else` ⟨*expr*⟩
| ⟨*func*⟩ `(` ⟨*expr*⟩, . . . , ⟨*expr*⟩ `)` | ⟨*var*⟩ `[` ⟨*expr*⟩ `]`

⟨*value*⟩ ::= `undef` | ⟨*float*⟩

⟨*binop*⟩ ::= ⟨*arithop*⟩ | ⟨*boolop*⟩

⟨*arithop*⟩ ::= `+` | `-` | `*` | `/`

⟨*boolop*⟩ ::= `<=` | `<` | `>` | `>=` | `==` | `!=` | `&&` | `||`

⟨*unop*⟩ ::= `-` | `~`

⟨*func*⟩ ::= `round` | `truncate` | `max` | `min` | `abs`
| `pos` | `pos_or_null` | `null` | `present`

Figure 5.2: Syntax of the μM language

$e_1 \odot e_2, \odot \in \{+,-\}$	`undef`	$f_2 \in \mathbb{F}$
`undef`	`undef`	$0 \odot f_2$
$f_1 \in \mathbb{F}$	$f_1 \odot 0$	$f_1 \odot_{\mathbb{F}} f_2$

$e_1 \odot e_2, \odot \in \{\times,\div\}$	`undef`	$f_2 \in \mathbb{F}, f_2 \neq 0$	0
`undef`	`undef`	`undef`	`undef`
f_1	`undef`	$f_1 \odot_{\mathbb{F}} f_2$	0

$b_1 \langle \textit{boolop} \rangle b_2$	`undef`	$f_2 \in \mathbb{F}$
`undef`	`undef`	`undef`
$f_1 \in \mathbb{F}$	`undef`	$f_1 \langle \textit{boolop} \rangle_{\mathbb{F}} f_2$

$m(e_1, e_2), m \in \{\texttt{min}, \texttt{max}\}$	`undef`	$f_2 \in \mathbb{F}$
`undef`	0	$m_{\mathbb{F}}(0, f_2)$
$f_1 \in \mathbb{F}$	$m_{\mathbb{F}}(f_1, 0)$	$m_{\mathbb{F}}(f_1, f_2)$

`round`(`undef`) = `undef`
`round`($f \in \mathbb{F}$) = $\textbf{floor}_{\mathbb{F}}(f + \texttt{sign}(f) * 0.50005)$
`truncate`(`undef`) = `undef`
`truncate`($f \in \mathbb{F}$) = $\textbf{floor}_{\mathbb{F}}(f + 10^{-6})$
`abs`(`x`) ≡ `if x >= 0 then x else -x`
`pos_or_null` (`x`) ≡ `x >= 0`
`pos`(`x`) ≡ `x > 0`
`null`(`x`) ≡ `x = 0`
`present`(`undef`) = 0
`present`($f \in \mathbb{F}$) = 1

Figure 5.3: Function semantics. For context on `round` and `truncate` definitions, see Section 5.2.2

while returning an extended environment Γ'. We take advantage of the fact that scalar and array assignments have different syntactic forms. M disallows assigning different types to the same variable; we rule this out in T-ASSIGN-*. A complete μM program is well-formed if $\varnothing \vdash P \Rrightarrow \_$.

Operational Semantics of μM At this stage, seeing that there are neither unbounded loops nor user-defined (recursive) functions in the language, M is obviously *not* Turing-complete. The language semantics are nonetheless quite devious, owing to the `undef` value, which can be explicitly converted to a float via a `+ 0`, as seen in earlier examples. We proceed to formalize them in Coq [10], using the Flocq library [13]. This ensures we correctly account for all cases related to the `undef` value, and guides the implementation of MLANG (Section 5.2.2).

The semantics of expressions is defined in Figure 5.5. The memory environment, written Ω is a function from variables to either scalar values (usually denoted v), or arrays (written $(v_0, \ldots, v_{n-1})$). A value absent from the environment evaluates to `undef`.

The special array index variable `X` is evaluated as a normal variable. Conditionals reduce normally, except when the guard is `undef`: in that case, the whole conditional evaluates into `undef`. If an index evaluates to `undef`, the whole array access is `undef`. In the case of a negative out-of-bounds index access the result is 0; in the case of a positive out-of-bounds index access the result is `undef`. Otherwise, the index is truncated into an integer, used to access Ω. The behavior of functions, unary and binary operators is described in Figure 5.3.

Figuring out these (unusual) semantics took over a year. We initially worked in a black-box setting, using as an oracle for our semantics the simplified online tax simulator offered by the DGFiP. After the initial set of M rules was open-sourced, we simply manually crafted test cases and fed those by hand to the online simulator to adjust our semantics. This allowed us to gain credibility and to have the DGFiP take us seriously. After that, we were allowed to enter the DGFiP offices and browse the source of their M compiler, as long as we did not exfiltrate any information. This final "code browsing" allowed us to understand the "inter" part of their compiler, a well as nail down the custom operators from Figure 5.11.

For statements, the memory environment Ω is extended into Ω_c, to propagate the error case that may be raised by exceptions. An assignment updates a

Global function environment Δ:

$\Delta(\texttt{round}) = \Delta(\texttt{truncate}) = \Delta(\texttt{abs}) = \Delta(\texttt{pos})$
$= \Delta(\texttt{pos\_or\_null}) = \Delta(\texttt{null}) = \Delta(\texttt{present}) = 1$
$\Delta(\texttt{min}) = \Delta(\texttt{max}) = \Delta(\langle \textit{arithop} \rangle) = \Delta(\langle \textit{boolop} \rangle) = 2$

Judgment : $\boxed{\Gamma \vdash e}$ ("Under Γ, e is well-formed")

T-FLOAT
$$\frac{}{\Gamma \vdash \langle \textit{float} \rangle}$$

T-UNDEF
$$\frac{}{\Gamma \vdash \texttt{undef}}$$

T-VAR-UNDEF
$$\frac{x \notin \text{dom } \Gamma}{\Gamma \vdash x}$$

T-VAR
$$\frac{\Gamma(x) = \texttt{scalar}}{\Gamma \vdash x}$$

T-INDEX-UNDEF
$$\frac{x \notin \text{dom } \Gamma \qquad \Gamma \vdash e}{\Gamma \vdash x[e]}$$

T-CONDITIONAL
$$\frac{\Gamma \vdash e_1 \qquad \Gamma \vdash e_2 \qquad \Gamma \vdash e_3}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3}$$

T-INDEX
$$\frac{\Gamma(x) = \texttt{array} \qquad \Gamma \vdash e}{\Gamma \vdash x[e]}$$

T-FUNC
$$\frac{\Delta(f) = n \qquad \Gamma \vdash e_1 \qquad \cdots \qquad \Gamma \vdash e_n}{\Gamma \vdash f(e_1, \ldots, e_n)}$$

Judgment : $\boxed{\Gamma \vdash \langle \textit{command} \rangle \Rrightarrow \Gamma'}$ and

$\boxed{\Gamma \vdash \langle \textit{program} \rangle \Rrightarrow \Gamma'}$ ("P transforms Γ to Γ'")

T-COND
$$\frac{\Gamma \vdash e}{\Gamma \vdash \texttt{if } e \texttt{ then } \langle \textit{error} \rangle \Rrightarrow \Gamma}$$

T-SEQ
$$\frac{\Gamma_0 \vdash c \Rrightarrow \Gamma_1 \qquad \Gamma_1 \vdash P \Rrightarrow \Gamma_2}{\Gamma_0 \vdash c \texttt{ ; } P \Rrightarrow \Gamma_2}$$

T-ASSIGN-SCALAR
$$\frac{x \in \Gamma \Rightarrow \Gamma(x) = \texttt{scalar} \qquad \Gamma \vdash e}{\Gamma \vdash x \texttt{ := } e \Rrightarrow \Gamma[x \mapsto \texttt{scalar}]}$$

T-ASSIGN-ARRAY
$$\frac{x \in \Gamma \Rightarrow \Gamma(x) = \texttt{array} \qquad \Gamma[\texttt{X} \mapsto \texttt{scalar}] \vdash e}{\Gamma \vdash x[\texttt{X}, n] \texttt{ := } e \Rrightarrow \Gamma[x \mapsto \texttt{array}]}$$

Figure 5.4: Typing of expressions and programs

Judgment : $\boxed{\Omega \vdash e \Downarrow v}$ ("Under Ω, e evaluates to v")

D-VALUE
$$\frac{v \in \langle \textit{value} \rangle}{\Omega \vdash v \Downarrow v}$$

D-VAR-UNDEF
$$\frac{x \notin \mathrm{dom}\ \Omega}{\Omega \vdash x \Downarrow \texttt{undef}}$$

D-VAR
$$\frac{\Omega(x) = v}{\Omega \vdash x \Downarrow v}$$

D-COND-TRUE
$$\frac{\Omega \vdash e_1 \Downarrow f \qquad f \notin \{0, \texttt{undef}\} \qquad \Omega \vdash e_2 \Downarrow v_2}{\Omega \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v_2}$$

D-X
$$\frac{\Omega(\mathtt{X}) = v}{\Omega \vdash \mathtt{X} \Downarrow v}$$

D-COND-FALSE
$$\frac{\Omega \vdash e_1 \Downarrow 0 \qquad \Omega \vdash e_3 \Downarrow v_3}{\Omega \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v_3}$$

D-INDEX-NEG
$$\frac{\Omega \vdash e \Downarrow r \qquad r < 0}{\Omega \vdash x[e] \Downarrow 0}$$

D-COND-UNDEF
$$\frac{\Omega \vdash e_1 \Downarrow \texttt{undef}}{\Omega \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow \texttt{undef}}$$

D-INDEX-UNDEF
$$\frac{\Omega \vdash e \Downarrow \texttt{undef}}{\Omega \vdash x[e] \Downarrow \texttt{undef}}$$

D-INDEX-OUTSIDE
$$\frac{\Omega \vdash e \Downarrow r \qquad r \geqslant n \qquad |\Omega(x)| = n}{\Omega \vdash x[e] \Downarrow \texttt{undef}}$$

D-TAB-UNDEF
$$\frac{x \notin \mathrm{dom}\ \Omega}{\Omega \vdash x[e] \Downarrow \texttt{undef}}$$

D-INDEX
$$\frac{\Omega(x) = (v_0, \ldots, v_{n-1}) \qquad \Omega \vdash e \Downarrow r \qquad r \in [0, n) \qquad r' = \mathrm{truncate}_{\mathbb{F}}(r)}{\Omega \vdash x[e] \Downarrow v_{r'}}$$

D-FUNC
$$\frac{\Omega \vdash e_1 \Downarrow v_1 \qquad \cdots \qquad \Omega \vdash e_n \Downarrow v_n}{\Omega \vdash f(e_1, \ldots, e_n) \Downarrow f(v_1, \ldots, v_n)}$$

Figure 5.5: Operational semantics: expressions

Judgment : $\boxed{\Omega_c \vdash c \Rrightarrow \Omega'_c}$ and

$\boxed{\Omega_c \vdash P \Rrightarrow \Omega'_c}$ ("Under Ω_c, P produces Ω'_c")

D-ASSIGN

$$\frac{\Omega_c \neq \texttt{error} \qquad \Omega_c \vdash e \Downarrow v}{\Omega_c \vdash x \texttt{ := } e \Rrightarrow \Omega_c[x \mapsto v]}$$

D-ASSERT-OTHER

$$\frac{\Omega_c \neq \texttt{error} \qquad \Omega_c \vdash e \Downarrow v \qquad v \in \{0, \texttt{undef}\}}{\Omega_c \vdash \texttt{if } e \texttt{ then } \langle \textit{error} \rangle \Rrightarrow \Omega_c}$$

D-ASSERT-TRUE

$$\frac{\Omega_c \neq \texttt{error} \qquad \Omega_c \vdash e \Downarrow f \qquad f \notin \{0, \texttt{undef}\}}{\Omega_c \vdash \texttt{if } e \texttt{ then } \langle \textit{error} \rangle \Rrightarrow \texttt{error}}$$

D-ERROR

$$\frac{}{\texttt{error} \vdash c \Rrightarrow \texttt{error}}$$

D-SEQ

$$\frac{\Omega_{c,0} \vdash c \Rrightarrow \Omega_{c,1} \qquad \Omega_{c,1} \vdash P \Rrightarrow \Omega_{c,2}}{\Omega_{c,0} \vdash c \texttt{ ; } P \Rrightarrow \Omega_{c,2}}$$

D-ASSIGN-TABLE

$$\frac{\Omega_c \neq \texttt{error} \qquad \Omega_c[\texttt{X} \mapsto 0] \vdash e \Downarrow v_0 \qquad \cdots \qquad \Omega_c[\texttt{X} \mapsto n-1] \vdash e \Downarrow v_{n-1}}{\Omega_c \vdash x[\texttt{X}, n] \texttt{ := } e \Rrightarrow \Omega_c[x \mapsto (v_0, \ldots, v_{n-1})]}$$

Figure 5.6: Operational semantics: statements

valid memory environment with the computed value. If an assertion's guard evaluates to a non-zero float, an error is raised; otherwise, program execution continues. Rule D-ERROR propagates a raised error across a program. The whole-array assignment works by evaluating the expression in different memory environments, one for each index.

Type Safety We now prove type safety in Coq. Owing to the unusual semantics of the `undef` value, and to the lax treatment of undefined variables, this provides an additional level of guarantee, by ensuring that reduction always produces a value or an error (i.e. we haven't forgotten any corner cases in our semantics). Furthermore, we show in the process that the store is consistent with the typing environment, written $\Gamma \triangleright \Omega$. This entails store typing (i.e. values of the right type are to be found in the store) and proper handling of undefined variables (i.e. dom $\Omega \subseteq$ dom Γ).

Theorem (Expressions)**.** If $\Gamma \triangleright \Omega$ and $\Gamma \vdash e$, then there exists v such that $\Gamma \vdash e \Downarrow v$.

We extend $\triangleright$ to statements, so as to account for exceptions:

$$\Gamma \triangleright_c \Omega_c \iff \Omega_c = \texttt{error} \vee \Gamma \triangleright \Omega_c$$

Theorem (Statements)**.** If $\Gamma \vdash c \Rrightarrow \Gamma'$ et $\Gamma \triangleright_c \Omega_c$, then there exists Ω'_c such that $\Omega_c \vdash c \Rrightarrow \Omega'_c$ and $\Gamma' \triangleright_c \Omega'_c$.

We provide full proofs and definitions in Coq, along with a guided tour of our development, in the supplement [12].

5.2.2. Overcoming Historical Mistakes with Language Design

As described in Figure 5.1, the internal compiler of the DGFiP compiles M files (Section 5.2.1) to C code. Insofar as we understand, the M codebase originally expressed the whole income tax computation. However, in the 1990s (Section 5.1.2), the DGFiP started executing the M code twice, with slightly different parameters, in order for the taxpayer to witness the impact of a tax reform. Rather than extending M with support for user-defined functions, the DGFiP wrote the new logic in C, in a folder called "inter", for multi-year computations. This piece of code can read and write variables used in the M codebase using shared global state. To assemble the final executable,

⟨*program*⟩ ::= ⟨*fundecl*⟩*

⟨*fundecl*⟩ ::= ⟨*funname*⟩ `(` ⟨*var*⟩* `)` : ⟨*command*⟩*

⟨*command*⟩ ::= `if` ⟨*expr*⟩ `then` ⟨*command*⟩* else ⟨*command*⟩*
| `partition with` ⟨*var_kind*⟩ : ⟨*command*⟩*
| ⟨*var*⟩ = ⟨*expr*⟩ | ⟨*var*⟩* `<-` ⟨*fun*⟩`()` | `del` ⟨*var*⟩

⟨*expr*⟩ ::= ⟨*var*⟩ | ⟨*float*⟩ | `undef` | ⟨*expr*⟩ ⟨*binop*⟩ ⟨*expr*⟩ | ⟨*unop*⟩ ⟨*expr*⟩
| ⟨*builtin*⟩ `(` ⟨*expr*⟩, . . . , ⟨*expr*⟩ `)` | `exists(` ⟨*var_kind*⟩ `)`

⟨*binop*⟩ ::= ⟨*arithop*⟩ | ⟨*boolop*⟩

⟨*arithop*⟩ ::= `+` | `-` | `*` | `/`

⟨*boolop*⟩ ::= `<=` | `<` | `>` | `>=` | `==` | `!=` | `&&` | `||`

⟨*unop*⟩ ::= `-` | `~`

⟨*var_kind*⟩ ::= `taxbenefit` | `deposit` | ...

⟨*fun*⟩ ::= ⟨*funname*⟩ | `call_m`

⟨*builtin*⟩ ::= `present` | `cast`

Figure 5.7: Syntax of the M++ language

M-produced C files and hand-written "inter" C files are compiled by GCC and distributed as a shared library. Over time, the "inter" folder grew to handle a variety of special cases, multiplying calls into the M codebase. At the time of writing, the "inter" folder amounts to 35,000 lines of C code.

This poses numerous problems. First, the mere fact that "inter" is written in C prevents it from being released to the public, the DGFiP fearing security issues that might somehow be triggered by malicious inputs provided by the taxpayer. Therefore, the taxpayer cannot reproduce the tax computation since key parts of the logic are missing. Second, by virtue of being written in C, "inter" does not compose with M, hindering maintainability, readability and auditability. Third, C limits the ability to modernize the codebase; right now, the online tax simulator is entirely written in C using Apache's CGI feature (including HTML code generation), a very legacy infrastructure for Web-based development. Fourth, C is notoriously hard to analyze, preventing both the DGFiP and the taxpayer from doing fine-grained analyses.

To address all of these limitations, we design M++, a companion domain-specific language that is powerful enough to completely eliminate the hand-written C code, and overcome the historical mistake of the "inter" files with modern language design.

Concrete Syntax and New Constructions The chief purpose of the M++ domain-specific language is to repeatedly call the M rules, with different M variable assignments for each call. To assist with this task, M++ provides basic computational facilities, such as functions and local variables. In essence, M++ allows implementing a "driver" for the M code.

Figure 5.8 shows concrete syntax for M++. We chose syntax resembling Python, where block scope is defined by indentation. As the French administration moves towards a modern digital infrastructure, Python seems to be reasonably understood across various administrative services.

Figure 5.7 formally lists all of the language constructs that M++ provides. A program is a sequence of function declarations. M++ features two flavors of variables. Local variables follow scoping rules similar to Python: there is one local variable scope per function body; however, unlike Python, we disallow shadowing and have no block scope or `nonlocal` keyword. Local variables exist only in M++. Variables in all-caps live in the M variable scope, which is shared between M and M++, and obey particular semantics.

Semantics of M++ Two constructs support the interaction between M and M++: the `<-` and `partition` operators. They have slightly unusual semantics, in the way that they deal with the M variable scope. These semantics are heavily influenced by the needs of the DGFiP, as we strived to provide something that would feel intuitive to technicians in the French administration.

To precisely define the expected behavior, Figure 5.9 presents reduction semantics of the form $\Delta, \Omega_1 \vdash c \rightsquigarrow \Omega_2$, meaning command c updates the store from Ω_1 to Ω_2, given the functions declared in Δ.

We distinguish built-ins, which may only appear in expressions and do not modify the global store, from functions, which are declared at the top-level and may modify the store. The `call_m` operation is a special function. The `<-` operator takes a *function* call, and executes it in a copy of the memory. Then, only those variables that appear on the left-hand side see their value propagated to the parent execution environment. Thus, `call_m` only affects variables $\vec{X}$.

To execute the function call, the `<-` operator either looks up definitions in Δ, the environment of user-defined functions, or executes the M rules in the `call_m` case, relying on the earlier definition of $\Rrightarrow$ (Figure 5.6).

Worded differently, our semantics introduce a notion of call stack and treat the M computation as a function call returning multiple values. It is to be noted that the original C code had no such notion, and that the $\vec{X}$ were nothing more than mere comments. As such, there was no way to statically rule out potential hidden state persisting from one `call_m` to another since the global scope was modified in place. With this formalization and its companion implementation, we were able to confirm that there is currently no reliance on hidden state (something which we suspect took considerable effort to enforce in the hand-written C code), and were able to design a much more principled semantics that we believe will lower the risk of future errors.

The `partition` operation operates over a variable kind k (Section 5.1.2). The sub-block c of `partition` executes in a restricted scope, where variables having kind k are temporarily set to `undef`. Upon completion of c, the variables at kind k are restored to their original value, while other variables are propagated from the sub-computation into the parent scope. This allows running computations while "disabling" groups of variables, e.g. ignoring an entire category of tax credits.

```
1  compute_benefits():
2   if exists(taxbenefit) or exists(deposit):
3      V_INDTEO = 1
4      V_CALCUL_NAPS = 1
5      partition with taxbenefit:
6        NAPSANSPENA, IAD11, INE, IRE, PREM8_11 <- call_m()
7      iad11 = cast(IAD11)
8      ire = cast(IRE)
9      ine = cast(INE)
10     prem = cast(PREM8_11)
11     V_CALCUL_NAPS = 0
12     V_IAD11TEO = iad11
13     V_IRETEO = ire
14     V_INETEO = ine
15     PREM8_11 = prem
```

Figure 5.8: Example function defined in M++

Example Figure 5.8 provides a complete M++ example, namely the function `compute_benefits`. The conditional at line 2 uses a variable kind-check (Section 5.1.2) to see if any variables of kind "tax benefit" have a non-`undef` value. Then, lines 3-4 set some flags before calling M. Line 5 tells us that the call to M at line 6 is to be executed in a restricted context where variables of kind "tax benefit" are set to `undef`. Line 6 runs the M computation, over the current state of the M variables; five M output variables are retained from this M execution, while the rest are discarded. Lines 7-11 represent local variable assignment, where `cast` has the same effect as `+ 0` in M, namely, forcing the conversion of `undef` to 0. Then, lines 11-15 set M some variables as input for later function calls.

After clarifying the semantics of M (Section 5.2.1), and designing a new domain-specific language to address its shortcomings (M++), we now present MLANG, a modern compiler for both M and M++.

Architecture of MLANG MLANG takes as input an M codebase, an M++ file, and a file specifying assumptions (described in the next paragraph). MLANG currently generates Python or C; it also offers a built-in interpreter for computations. MLANG is implemented in OCaml, with around 9,000 lines of code. The general architecture is shown in Figure 5.10. The M files and the M++

Judgments:

$\boxed{\Delta, \Omega \vdash e \wr v}$ ("Under Δ, Ω, e evaluates into v")

$\boxed{\Delta, \Omega_1 \vdash c \rightsquigarrow \Omega_2}$ ("Under Δ, c transforms Ω_1 into Ω_2")

CAST-FLOAT
$$\frac{\Delta, \Omega \vdash e \wr f \qquad f \neq \texttt{undef}}{\Delta, \Omega \vdash \mathsf{cast}(e) \wr f}$$

CAST-UNDEF
$$\frac{\Delta, \Omega \vdash e \wr \texttt{undef}}{\Delta, \Omega \vdash \mathsf{cast}(e) \wr 0}$$

EXISTS-TRUE
$$\frac{\exists X \in \Omega, \mathsf{kind}(X) = k \wedge \Omega(X) \neq \texttt{undef}}{\Delta, \Omega \vdash \mathsf{exists}(k) \wr 1}$$

EXISTS-FALSE
$$\frac{\forall X \in \Omega, \mathsf{kind}(X) \neq k \vee \Omega(X) = \texttt{undef}}{\Delta, \Omega \vdash \mathsf{exists}(k) \wr 0}$$

CALL
$$\frac{\begin{array}{ll} \Omega_1 \vdash \mathsf{M\ rules} \Rrightarrow \Omega_2 & \text{if } \mathsf{f} = \mathsf{call\_m} \\ \Delta, \Omega_1 \vdash \Delta(f) \rightsquigarrow \Omega_2 & \text{otherwise} \end{array} \qquad \begin{array}{l} \Omega_3(Y) = \Omega_1(Y) \text{ if } Y \notin \vec{X} \\ \Omega_3(Y) = \Omega_2(Y) \text{ if } Y \in \vec{X} \end{array}}{\Delta, \Omega_1 \vdash \vec{X} \leftarrow \mathsf{f}() \rightsquigarrow \Omega_3}$$

PARTITION
$$\frac{\begin{array}{l} \Omega_2(Y) = \texttt{undef} \text{ if } \mathsf{kind}(Y) = k \\ \Omega_2(Y) = \Omega_1(Y) \text{ otherwise} \end{array} \qquad \Delta, \Omega_2 \vdash c \rightsquigarrow \Omega_3 \qquad \begin{array}{l} \Omega_4(Y) = \Omega_1(Y) \text{ if } \mathsf{kind}(Y) = k \\ \Omega_4(Y) = \Omega_3(Y) \text{ otherwise} \end{array}}{\Delta, \Omega_1 \vdash \texttt{partition with } k : c \rightsquigarrow \Omega_4}$$

DELETE
$$\frac{\Delta, \Omega_1 \vdash v = \texttt{undef} \rightsquigarrow \Omega_2}{\Delta, \Omega_1 \vdash \mathsf{del}\ v \rightsquigarrow \Omega_2}$$

Figure 5.9: Reduction rules of M++

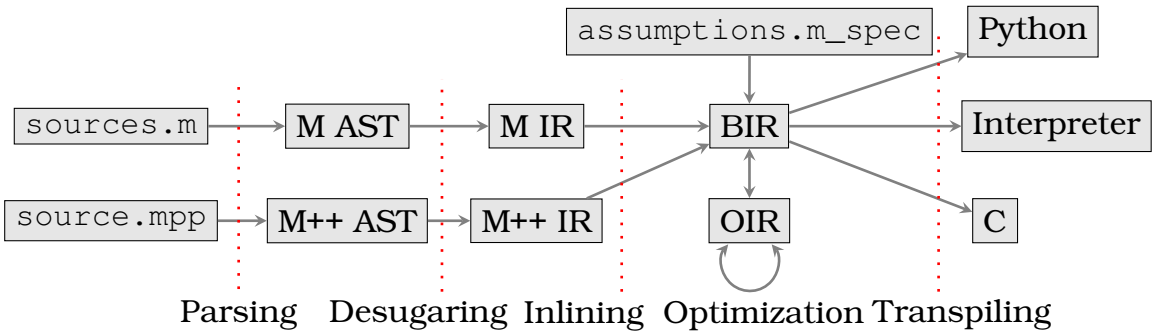

Figure 5.10: MLANG compilation passes

program are first parsed and transformed into intermediate representations. These intermediate representations are inlined into a single backend intermediate representation (BIR), consisting of assignments and conditionals. Inlining is aware of the semantic subtleties described in Figure 5.9 and uses temporary variable assignments to save/restore the shared M/M++ scope. BIR code is then translated to the optimization intermediate representation (OIR) in order to perform optimizations. OIR is the control-flow-graph (CFG) equivalent of BIR.

OIR is the representation on which we perform our optimizations (Section 5.3.1). For instance, in order to perform constant propagation, we must check that a given assignment to a variable dominates all its subsequent uses. A CFG is the best data structure for this kind of analysis. We later on switch back to the AST-based BIR in order to generate textual C output.

In M, a variable not defined in the current memory environment evaluates to `undef` (rule D-VAR-UNDEF, Figure 5.5). This permissive behavior is fine for an interpreter which has a dynamic execution environment; however, our goal is to generate efficient C and Python code that can be integrated into existing software. As such, declaring every single one of the 27,113 possible variables (as found in the original M rules) in C would be quite unsavory.

We therefore devise a mechanism that allows stating ahead of time which variables can be truly treated as inputs, and which are the outputs that we are interested in. Since these vary depending on the use-case, we choose to list these assumptions in a separate file that can be provided alongside with the M/M++ source code, rather than making this an intrinsic, immutable property set at variable-declaration time. Doing so increases the quality of the generated C or Python.

```
// my_var1 is a local variable always defined
#define my_truncate(a) ( my_var1=(a)+0.000001,floor(my_var1) )
#define my_round(a) (floor(
  (a<0) ? (double)(long long)(a-.50005)
        : (double)(long long)(a+.50005)))
```

Figure 5.11: Custom rounding and truncation rules

We call these *assumption files*; we have hand-written 5 of those. **All** is the empty file, i.e. no additional assumptions. This leaves 2459 input variables, and 10,411 output variables for the 2018 codebase. **Selected outs** enables all input variables, but retains only 11 output variables. **Tests** corresponds to the inputs and outputs used in the test files used by the DGFiP. **Simplified** corresponds to the simplified simulator released each year by the DGFiP a few months before the full income tax computation is released. There are 214 inputs, and we chose 11 output variables. **Basic** accepts as inputs only the marital status and the salaries of each individual of the couple. The output is the income tax.

The DGFiP's legacy system has a single backend that produces pre-ANSI (K&R) C. For each M rule, two C computations are emitted. The first one aims to determine whether the resulting value is defined. It operates on C's `char` type, where `0` is undefined or `1` is defined. The second computation is syntactically identical, except it operates on `double` and thus computes the actual arithmetic expression. This two-step process explains some of the operational semantics: with `0` being undefined, the special value `undef` is absorbing for e.g. the multiplication. Careful study of the generated code also allowed us to nail down some non-standard rounding and truncation rules which had until then eluded us. We list them in Figure 5.11; these are used to implement the built-in operators from Figure 5.2 in both our interpreter and backends.

Our backend generates C and Python from BIR. Since BIR only features assignments, arithmetic and conditionals, we plan to extend it with backends for JavaScript, R/MatLab and even SQL for in-database native tax computation. Depending on the DGFiP's appetite for formal verification, we may verify the whole compiler since the semantics are relatively small.

Implementing a new backend is not very onerous: it took us 500 lines for the C backend and 375 lines for the Python backend. Both backends are validated by running them over the entire test suite and comparing the result

with our reference interpreter.

Our generated code only relies on a small library of helpers which implement operations over M values. These helpers are aware of all the semantic subtleties of M and are manually audited against the paper semantics.

5.3. Modernizing Programs and Languages Together

Building on this custom-domain specific patchwork of languages handled through a unified compiler, we reach for a few low-hanging fruits that bring novel insights about the codebase and how it could be improve. Finally, we discuss future work for a complete and principled incremental rewrite of the income tax algorithm that would be compatible with the constrains of continuous service.

5.3.1. The Compiler as a Program Analysis Platform

A semantics-aware compiler for a domain-specific language can take advantage of all the domain-specific assumptions one can make on the programs. Hence, we implemented in MLANG several optimization and analysis passes over M and M++ programs. With a minimal compiler implementation effort, we are able to access critical global information about the precision of the income tax computation, or generate efficient code for batch processing.

Optimizations In the 2018 tax code, the initial number of BIR instructions after inlining M and M++ files together is 656,020. This essentially corresponds to what the legacy compiler normally generates, since it performs no optimizations.

Thanks to its modern compiler architecture, MLANG can easily perform numerous textbook optimizations, namely dead code elimination, inlining and partial evaluation. This allows greatly improving the quality of the generated code.

We now present a series of optimizations, performed on the OIR intermediate representation. The number of instructions after these optimizations is shown in Figure 5.12. Without any assumption (**All**), the optimizations shrink the generated C code to 15% of the unoptimized size (a factor of 6.5). With the most restrictive assumption file (**Simplified**), only 0.47% of the original instructions remain after optimization.

Spec. name	# inputs	# outputs	# instructions
All	2,459	10,411	129,683
Selected outs	2,459	11	99,922
Tests	1,635	646	111,839
Simplified	228	11	4,172
Basic	3	1	553

Figure 5.12: Number of instructions generated after optimization. Instructions with optimizations disabled: 656,020.

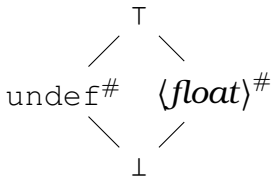

Figure 5.13: Definedness lattice

Due to the presence of `undef`, some usual optimizations are not available. For example, optimizing `e * 0` into `0` is incorrect when `e` is `undef`, as `undef * 0` = `undef`. Similarly, `e + 0` cannot be rewritten as `e`. Our partial evaluation is thus combined with a simple definedness analysis. The lattice of the analysis is shown in Figure 5.13; we use the standard sharp symbol of abstract interpretation [14] to denote abstract elements. The transfer function $\texttt{absorb}^\#$ defined in Figure 5.14 is used to compute the definedness in the case of the multiplication, the division and all operators in $\langle \mathit{boolop} \rangle$. The $\texttt{cast}^\#$ transfer function is used for the addition and the subtraction.

This definedness analysis enables finer-grained partial evaluation rules, such as those presented in Figure 5.15.

d_1	d_2	$\texttt{absorb}^\#(d_1, d_2)$	$\texttt{cast}^\#(d_1, d_2)$
$\texttt{undef}^\#$	$\texttt{undef}^\#$	$\texttt{undef}^\#$	$\texttt{undef}^\#$
$\texttt{undef}^\#$	$\langle \mathit{float} \rangle^\#$	$\texttt{undef}^\#$	$\langle \mathit{float} \rangle^\#$
$\langle \mathit{float} \rangle^\#$	$\texttt{undef}^\#$	$\texttt{undef}^\#$	$\langle \mathit{float} \rangle^\#$
$\langle \mathit{float} \rangle^\#$	$\langle \mathit{float} \rangle^\#$	$\langle \mathit{float} \rangle^\#$	$\langle \mathit{float} \rangle^\#$

Figure 5.14: Transfer functions over the definedness lattice, implicitly lifted to the full lattice.

$e + \texttt{undef} \rightsquigarrow e$	$e : \langle \mathit{float} \rangle^{\#} + 0 \rightsquigarrow e$
$e * 1 \rightsquigarrow e$	$e : \langle \mathit{float} \rangle^{\#} * 0 \rightsquigarrow 0$
$\texttt{max}(0, \texttt{min}(0, x)) \rightsquigarrow 0$	$\texttt{present}(\texttt{undef}) \rightsquigarrow 0$
$\texttt{max}(0, -\texttt{max}(0, x)) \rightsquigarrow 0$	$\texttt{present}(e : \langle \mathit{float} \rangle^{\#}) \rightsquigarrow 1$

Figure 5.15: Examples of optimizations

The optimizations for `+ 0` and `* 0` are invalid in the presence of IEEE-754 special values (NaN, minus zero, infinities) [15], [16]. We have instrumented the M code to confirm that these are valid on the values used. But for safety, these unsafe optimizations are only enabled if the `--fast_math` flag is set.

Performance of the Generated Code Due to the sheer size of the code and number of variables, generating efficient code is somewhat delicate – we had the pleasure of breaking both the Clang and Python parsers because of an exceedingly naïve translation. Thankfully, owing to our flexible architecture for MLANG, we were able to quickly iterate and evaluate several design choices.

We now show the benefits of a modern compiler infrastructure, and proceed to describe a variety of instrumentations, techniques and tweaking knobs that allowed us to gain insights on the the tax computation. By bringing the M language into the 21st century, we not only greatly enhance the quality of the generated code, but also unlock a host of techniques that significantly increase our confidence in the French tax computation.

We initially generated C code that would emit one local variable per M variable. But with tens of thousands of local variables, running the code required `ulimit -s`. We analyzed the legacy code and found out that the DGFiP stored all of the M variables in a global array. We implemented the same technique and found out that with `-O1`, we were almost as fast as the legacy code. We attribute this improvement to the fact that the array, which is a few dozen kB, which fits in the L2 cache of most modern processors. This is a surprisingly fortuitous choice by the DGFiP.

See Figure 5.16 for full results. In the grand scheme of things, the cost of computing the final tax is dwarfed by the time spent generating a PDF summary for the taxpayer (~200ms). The 500µs difference between the DGFiP's system and ours is thus insignificant.

Scheme	M compiler	C compiler	Bin. size	Time
Original	DGFiP	GCC `-O0`	7 Mo	~ 1.5 ms
Original	DGFiP	GCC `-O1`	7 Mo	~ 1.5 ms
Array	MLANG	Clang `-O0`	19 Mo	~ 4 ms
Array	MLANG	Clang `-O1`	10 Mo	~ 2 ms

Figure 5.16: Performance of the C code generated by various compilation schemes for the M code. The time measured is the time spent inside the main tax computation function for one fiscal household picked in the set of test cases. Size of the compiled binary is indicated. "Original" corresponds to the DGFiP's legacy system. "Local vars" corresponds to MLANG's C backend mapping each M variable to a C local variable.

The Cost of IEEE-754 Relying on IEEE-754 and its limited precision for something as crucial as the income tax of an entire nation naturally raises questions. Thanks to our new infrastructure, we were able to instrument the generated code and gain numerous insights.

We tweaked our backend to use the MPFR multiprecision library [17]. With 1024-bit floats, all tests still pass, meaning that there is no loss of precision with the double-precision 64-bit format.

We then instrumented the code to measure the effect of the IEEE-754 rounding mode on the final result. Anything other than the default (rounding to nearest, ties to even) generates incorrect results. The control-flow remains roughly the same, but some comparisons against 0 do give out different results as the computation skews negatively or positively. We plan in the future to devise a static analysis that could formally detect errors, such as comparisons that are always false, or numbers that may be suspiciously close to zero (denormals).

Nevertheless, floating-point computations are notoriously hard to analyze and reason about, so we set out to investigate replacing floats with integer values. In our first experiment, we adopted big decimals, i.e. a bignum for the integer part and a fixed amount of digits for the fractional part. Our test suite indicates that the integer part never exceeds 9999999999 (encodable in 37 bits); it also indicates that with 40 bits of precision for the fractional part, we get correct results. This means that a 128-bit integer would be a viable alternative to a `double`, with the added advantage that formal analysis tools would be able to deal with it much better.

Finally, we wondered if it was possible to completely work without floating-point and eliminate imprecision altogether, taking low-level details such as rounding mode and signed zeroes completely out of the picture.

To that end, we encoded values as fractions where both numerator and denominator are big integers. We observed that both never exceed 2^{128}, meaning we could conceivably implement values as a struct with two 128-bit integers and a sign bit. We have yet to investigate the performance impact of this change.

Test-case Generation The DGFiP test suite is painstakingly constructed by hand by lawyers year after year. From this test suite, we extracted 476 usable test cases that don't raise any exceptions (see Section 5.1.2). The DGFiP has no infrastructure to automatically generate cases that would exercise new situations. As such, the test suite remains relatively limited in the variety of households it covers. Furthermore, many of the hand-written tests are for previous editions of the tax code, and describe situations that would be rejected by the current tax code.

Generating test cases is actually non-trivial: the search space is incredibly large, owing to the amount of variables, but also deeply constrained, owing to the fact that most variables only admit a few possible values (Section 5.1.2), and are further constrained in relationship to other variables.

We now set out to automatically generate fresh (valid) test cases for the tax computation, with two objectives: assert on a very large number of test cases that our code and the legacy implementation compute the same result; and exhibit corner cases that were previously not exercised, so as to generate fresh novel tax situations for lawmakers to consider.

We start by randomly mutating the legacy test suite, in order to generate new distinct, valid test cases. If a test case raises an exception, we discard it. We obtain 1267 tests, but these are, unsurprisingly, very close to the legacy test suite and do not exercise very many new situations. They did, however, help us when reverse-engineering the semantics of M. We now have 100% conformance on those tests.

In order to better explore the search space, we turn to AFL [18]. The tool admits several usage modes – finding genuine crashes (e.g. segfaults), or generating test cases for further seeding into the rest of the testing pipeline. We focus on the latter mode, meaning that we generate an artificial "crash" when a synthesized testcase raises no M errors, that is, when we have found

a valid testcase. We first devise an injection from opaque binary inputs, which AFL controls, to the DGFiP input variables. Once "crashes" have been collected, we simply emit a set of test inputs that has the same format as the DGFiP.

Thanks to this very flexible architecture, we were able to perform fully general fuzzing exercising all input variables, as well as targeted fuzzing that focuses on a subset of the variables. The former takes a few hours on a high-end machine; the latter mere minutes. We synthesized around 30,000 tests cases, which we reduced down to 275 using afl-cmin.

So far, the fuzzer-generated test case have pointed out of a few bugs in MLANG's optimizations and backends. We plan to further use AFL to find find test cases that satisfy extra properties not originally present in the tax code, e.g. an excessively high marginal tax rate that might raise some legality questions.

We attempted to use dynamic symbolic execution tool KLEE [19], but found out that it only had extremely limited support for floating-point computations. As detailed earlier, we have found that integer based computations are a valid replacement for floats, and plan to use this alternate compilation scheme to investigate whether KLEE would provide interesting test cases.

Coverage Measurements Finally, we wish to evaluate how "good" our new test cases are. Code coverage seems like a natural notion, especially seeing that there is currently none in the DGFiP infrastructure. However, traditional code coverage makes little sense: conditionals are very rare in the generated code.

Rather, we focus on value coverage: for each assignment in the code, we count the number of distinct values assigned during the execution of an entire test case. This is a good proxy for test quality: the more different values flow through an assignment, the more interesting the tax situation is.

Figure 5.17 shows our measurements. The first take-away is that our randomized tests did not result in meaningful tests: the number of assignments that are uncovered actually increased. The tests we obtained with AFL, however, significantly increase the quality of test coverage. We managed to synthesize many tests that exercise statements previously unvisited by the DGFiP's test suite, and exhibit much more complex assignments (2 or more different values assigned).

Our knowledge of the existing DGFiP test suite is incomplete, as we only

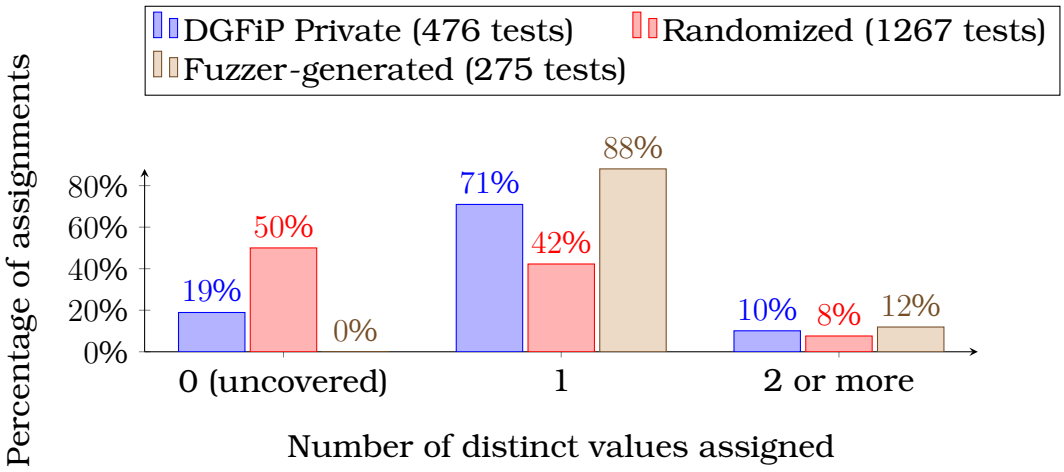

Figure 5.17: Value coverage of assignments for each test suite

have access to a partial set of tests. In particular, a special set of rules apply when the tax needs to be adjusted later on following an audit, and the tests for these have not been communicated to us. We hope to obtain visibility onto those in the future.

5.3.2. Thinking Ahead: Planning for a Smooth Transition

Although the work presented in this chapter could give the income tax legacy infrastructure a sunny future, a social problem sentences the M codebase to a short lifespan. Indeed, the M codebase is the brainchild of two DGFiP civil servants that have been charged of its development and maintenance since 1988. After more than 30 years of service, these civil servants are now nearing retirement age, leaving the DGFiP with a serious medium-term conundrum. Who will continue to update the tax computation algorithm each year with the recurring changes in tax law?

One option could be to train successors, a new generation of “M-masters”. This approach has been tried in the recent years, with little success. The reason of the failure is that given the complexity and intrication of the M codebase, it has empirically taken between three and five years to train a new “M-master”. This very long training period entails a lot of attrition for the trainees confronted to the general obsolescence of the infrastructure, and who may not picture themselves working on it for the next 30 years. Moreover, the DGFiP has no incentives to retain civil servants at the same position for

extended periods of time, due to its internal human ressources policy.

Hence, there is no other option than considering a complete rewrite of the M codebase. A civil servant, former project lead on the income tax computation, Christophe Gaie, outlines his thoughts on such a rewrite in a research article [20]. According to him, the critical availability constraints of the system mandates an incremental rewrite associated with an extended transition period where both systems run in parallel. Concretely, this would mean that the new codebase would slowly implement more and more features over time, defaulting on the old codebase for missing parts of the computation. Gaie proposes an API-based interoperability scheme for this transition period.

Improving on this proposal, we would keep the idea of a transition period where both systems run in parallel, but consider a different interoperability scheme. The shortcoming of Gaie's proposal is that it entails a massive refactoring of the entire DGFiP IT system. Currently, the income tax algorithm is distributed to the various DGFiP applications that require it either as a binary shared library (`.so`), either as C source files. Switching to an API-based interoperability scheme would require refactoring a myriad of DGFiP applications, from the GUI calculator used by tax collectors in local tax offices, to the mainframe that performs the batch processing of tax returns.

Instead, my proposal is to keep the current interoperability scheme by using compilation techniques to generate C source code from the high-level language in which the rewriting of the M codebase will be done. Anticipating on the next chapter, we could imagine a system where the compiler of the M rewriting language directly connects (at an intermediate representation level) with MLANG. Such a connection could enjoy a high level of assurance with respect to semantics preservation (for instance via partial or total certification), while maximizing toolchain reuse.

Of course, this proposal would only work if we choose for the language of the M rewriting a language that has both a formal semantics and a flexible compiler that can be easily repurposed. Interestingly, the kind of domain-specific languages we advocate for in this dissertation enjoys those two qualities.

Conclusion

In this chapter, we demonstrate how our methodology can be leveraged for a particular class of high-assurance software deemed to serve for an indefinite period of time. On top of bringing existing code to the formal world, we can use

the compiler infrastructure to help evolve the codebase with smart program analysis or certified interoperability.

If the DGFiP had chosen to write its income tax computation in a language that was mainstream in mainframe computing in 1988, we would have ended up with a COBOL or pure C implementation. Because C and COBOL are general-purpose languages with very tricky semantics, the only choice would have been to treat this code as a black box, with perhaps the faint hope of proving some form of memory safety.

One could argue that the simplicity of the M domain-specific language makes it closer to a toy language than a real object of study for formal methods research. Indeed, the formalization and analysis presented in the chapter, though absent from the original language design and thus novel, do not involve any complex or unusual semantic feature. To this valid criticism, we would answer that programming language research should be at least partially guided by the real-world artifacts that need our expertise. Far from the spotlights of the current trends, there are still domains where basic programming language techniques can create a lot of value for the users.

Beyond building modern, much-needed tools to applications on their way to obsolescence, the use of domain-specific languages can create novel formal artifacts that test the limits of current state-of-the-art tools. For instance, the M codebase, despite being completely loop-free, contains hundred of thousands of variables and disjunctions, as well as non-linear arithmetic (on floating-point values). Analyzing this kind of code is a challenge in itself, and may lead to future interesting developments for program analysis and prover tooling.

One of the intents of this work is to get formal methods out of its decades-long application trinity: compilers/cryptography/low-level systems. Business-related software has historically been neglected by the programming languages theory community. Very early on, this neglect led to the creation of COBOL [21] in 1959, whose standardization was done through a committee composed entirely of representatives from the US Army and big tech companies of the time. Later, in the Java golden era, the Unified Modeling Language (UML) [22] quickly become an international norm among software architects, but its variants based on a rigorous formalization [23]–[25] failed to have widespread industry impact. More recently, in the Business Rules Management Systems (BRMS) community, the leading open-source solution, Drools [26] is still based on an informal base language.

Formalizing an existing system is hard, especially after decades of feature

additions and evolving; the recent attempt of Benzaken and Contejean to find a formal semantics for SQL [27] is a stark reminder. If the methodology of this dissertation allows for an intermediate path by carving out a workable subset, being able to deal with the entire codebase is even better. To achieve this goal, and in the absence of a lucky historical legacy such as the M domain-specific language, it is paramount for programming language experts to influence the design process from its inception.

The motivations of the DGFiP for choosing to create their own domain-specific language were of course completely alien to this line of thinking in 1988. After some detective work, we were able to locate and contact the person directly responsible for the invention and development of the M language at the time: Dominique Fulcrand, principal tax inspector at the DGFiP. According to him, the project originated from the need to create a version of the income tax calculator accessible to the general public through Minitel [28], the French proto-Internet. A complete rewrite of the codebase was necessary because the historical COBOL implementation dating from the 1970s could not be run elsewhere than the GCOS Bull mainframe that it was operating on. Other DGFiP applications were also in need of a tax calculator at the time. Hence, the main design requirement was to provide a portable implementation of a tax calculator: C was thus chosen as the goto language, as it could be compiled with GCC to all the needed platforms.

However, writing tax computation rules directly in C was seen as a no-go. The M language then was created as a way to directly transpose how domain experts (tax inspectors) were thinking about tax computations: as a series of equations tying variables together. As such, the M language design did not incorporate any software engineering considerations. Retrospectively, this was a partial failure since it promoted an organizational shift where two non-programmers tax inspectors would write the M code completely and be in charge of the codebase maintenance and evolution. Because of their lack of computer science education, the code they produced turns out to be very difficult to maintain, and relies on a lot of implicit invariants.

Today, a number of scholars continue to recommend the use of low-code/no-code tools [29], [30] with the promise that non-programmers domain experts could write the code of this class of law-related applications called legal expert systems. Instead, we believe that the expertise of the software engineer is paramount to create and organize systems that can be efficiently run and maintained. Hence, we claim that the goal for such domain-specific languages is not for non-programmers domain experts to be able to write the code, but

rather to be able to review it as a way to tackle the specification correctness problem. This argument will be put to practice in the next chapter.

References

[1] A. Van Deursen and T. Kuipers, "Rapid system understanding: two COBOL case studies", in *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No. 98TB100242)*, IEEE, 1998, pp. 90–97.

[2] P. A. Hausler, M. G. Pleszkoch, R. C. Linger, and A. R. Hevner, "Using function abstraction to understand program behavior", *IEEE Software*, vol. 7, no. 1, pp. 55–63, 1990.

[3] C. M. Matei, "Modernization solution for legacy banking system using an open architecture", *Informatica Economica*, vol. 16, no. 2, p. 92, 2012.

[4] D. Merigoux, R. Monat, and J. Protzenko, "A modern compiler for the french tax code", in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2021, Virtual, Republic of Korea: Association for Computing Machinery, 2021, pp. 71–82, ISBN: 9781450383257. DOI: 10.1145/3446804.3446850. [Online]. Available: https://doi.org/10.1145/3446804.3446850.

[5] C. Thompson, *The code that controls your money*, 2020. [Online]. Available: https://www.wealthsimple.com/en-ca/magazine/cobol-controls-your-money.

[6] J. G. Barnes, *Programming in ADA*. Addison-Wesley Longman Publishing Co., Inc., 1984.

[7] C. Landais, T. Piketty, and E. Saez, *Pour une révolution fiscale: un impôt sur le revenu pour le 21 ème siècle*, 2011. [Online]. Available: https://www.revolution-fiscale.fr/simuler/irpp/.

[8] Équipe Leximpact de L'Assemblée nationale, *Leximpact*, 2019. [Online]. Available: https://leximpact.an.fr/.

[9] Direction Générale des Finances Publiques (DGFiP), *Les règles du moteur de calcul de l'impôt sur le revenu et de l'impôt sur la fortune immobilière*, version 2018.6.7, 2019. [Online]. Available: https://gitlab.adullact.net/dgfip/ir-calcul.

[10] The Coq Development Team, *The Coq Proof Assistant Reference Manual, version 8.7*, Oct. 2017. [Online]. Available: http://coq.inria.fr.

[11] D. Merigoux, R. Monat, and J. Protzenko, *Mlang source code*, 2020. [Online]. Available: https://github.com/MLanguage/mlang.

[12] ——, *A modern compiler for the french tax code - artifact*, version CC21-AEC-v1.1, Jan. 2021. DOI: 10.5281/zenodo.4456774. [Online]. Available: https://doi.org/10.5281/zenodo.4456774.

[13] S. Boldo and G. Melquiond, "Flocq: A unified library for proving floating-point algorithms in Coq", in *20th IEEE Symposium on Computer Arithmetic, ARITH 2011, Tübingen, Germany, 25-27 July 2011*, E. Antelo, D. Hough, and P. Ienne, Eds., IEEE Computer Society, 2011, pp. 243–252. DOI: 10.1109/ARITH.2011.40. [Online]. Available: https://doi.org/10.1109/ARITH.2011.40.

[14] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints", in *POPL*, ACM, 1977, pp. 238–252.

[15] S. Boldo, J. Jourdan, X. Leroy, and G. Melquiond, "Verified compilation of floating-point computations", *J. Autom. Reason.*, vol. 54, no. 2, pp. 135–163, 2015. DOI: 10.1007/s10817-014-9317-x. [Online]. Available: https://doi.org/10.1007/s10817-014-9317-x.

[16] J. Muller, N. Brunie, F. de Dinechin, C. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic (2nd Ed.)* Springer, 2018, ISBN: 978-3-319-76525-9. DOI: 10.1007/978-3-319-76526-6. [Online]. Available: https://doi.org/10.1007/978-3-319-76526-6.

[17] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: a multiple-precision binary floating-point library with correct rounding", *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, no. 2, 13–es, 2007.

[18] M. Zalewski, *American fuzzy lop*, 2014.

[19] C. Cadar, D. Dunbar, D. R. Engler, *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.", in *OSDI*, vol. 8, 2008, pp. 209–224.

[20] C. Gaie, "From secured legacy systems to interoperable services (the careful evolution of the french tax administration to provide new possibilities while ensuring the primary tax recovering objective)", *International Journal of Computational Systems Engineering*, vol. 6, no. 2, pp. 76–83, 2020.

[21] J. E. Sammet, "The early history of COBOL", in *History of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 1978, pp. 199–243, ISBN: 0127450408. [Online]. Available: https://doi.org/10.1145/800025.1198367.

[22] J. Rumbaugh, I. Jacobson, and G. Booch, "The unified modeling language", *Reference manual*, 1999.

[23] W. E. McUmber and B. H. Cheng, "A general framework for formalizing UML with formal languages", in *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, IEEE, 2001, pp. 433–442.

[24] M. von der Beeck, "Formalization of UML-statecharts", in *UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, M. Gogolla and C. Kobryn, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 406–421, ISBN: 978-3-540-45441-0.

[25] J. E. Smith, M. M. Kokar, and K. Baclawski, "Formal verification of UML diagrams: a first step towards code generation.", in *pUML*, Citeseer, 2001, pp. 224–240.

[26] M. Proctor, "Drools: a rule engine for complex event processing", in *Proceedings of the 4th International Conference on Applications of Graph Transformations with Industrial Relevance*, ser. AGTIVE'11, Budapest, Hungary: Springer-Verlag, 2012, pp. 2–2. DOI: 10.1007/978-3-642-34176-2_2. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-34176-2_2.

[27] V. Benzaken and E. Contejean, "A Coq mechanised formal semantics for realistic sql queries: formally reconciling sql and bag relational algebra", in *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2019, pp. 249–261.

[28] J.-Y. Rincé, *Le minitel*. FeniXX, 1994.

[29] J. Morris, “Spreadsheets for legal reasoning: the continued promise of declarative logic programming in law”, *Available at SSRN 3577239*, 2020.

[30] M. Waddington, “Research note. Rules as Code”, *Law in Context. A Socio-legal Journal*, vol. 37, no. 1, pp. 1–8, 2020.

6. Catala: A Specification Language for the Law

Contents

Abstract

The real-world legal expert system presented in the last chapter prompted this legitimate question: can we do better? The use of a domain-specific language by the French tax administration since the 1990's hints at a deeper problem when turning law into code. Indeed, the logical structure of legal texts breaks the design patterns of most programming languages, turning legal expert systems into spaghetti code.

After a careful analysis of this logical structure, we propose **Catala**, a novel proof-oriented domain-specific language for writing executable specifications of legal expert systems. The design process of this domain-specific language heavily involves lawyers, which are the domain experts in this case. We believe that **Catala** can be the entry point of future formal methods developments in the world of computational law.

On a vu, cependant, que la méthode de l'explicitation a pour particularité dominante de rationaliser l'écriture des textes. Cela conduit à se demander si on ne pourrait pas *ab initio* conformer leur rédaction à cette rationalité formelle.

(Pierre Catala, Le Droit à l'épreuve du numérique*, 1998)*

(...) the temptation is to surrender the vital logic which has actually yielded the conclusion and to substitute for it forms of speech which are rigorous in appearance and which give an illusion of certitude.

(Jonh Dewey, Logical Method and Law*, 1924)*

This chapter is based upon the following publication:

D. Merigoux, N. Chataing, and J. Protzenko, "Catala: a programming language for the law", *Proc. ACM Program. Lang.*, vol. 5, no. ICFP, Aug. 2021. DOI: 10.1145/3473582. [Online]. Available: https://doi.org/10.1145/3473582

My personal contribution to this publication has been the design of the ***Catala*** *language, its surface syntax, the implementation of the compiler and code examples. I co-authored the semantics of the language.*

6.1. Formal Methods for the Law

We now know that since at least 2000 B.C.E. and the Code of Ur-Nammu [2], various societies have attempted to edict, codify and record their governing principles, customs and rules in a set of legal texts – the law. Nowadays, most aspects of one's daily life are regulated by a set of laws or another, ranging from family law, tax law, criminal law, to maritime laws, business laws or civil rights law. No law is set in stone; laws are, over time, amended, revoked and modified by legislative bodies. The resulting legal texts eventually reflect the complexity of the process and embody the centuries of practice, debates, power struggles and political compromises between various parties.

The practice of law thus oftentimes requires substantial human input. First, to navigate the patchwork of exceptions, amendments, statutes and jurisprudence relevant to a given case. Second, to fully appreciate and identify the situation at play; understand whether one party falls in a given category or another; and generally classify and categorize, in order to interpret a real-world situation into something the law can talk about.

This latter aspect is perhaps the greatest challenge for a computer scientist: a general classification system remains an elusive prospect when so much human judgement and appreciation is involved. Fortunately, a subset of the law, called *computational law* or sometimes *rules as code*, concerns itself with situations where entities are well-defined, and where human appreciation, judgement or interpretation are not generally expected. Examples of computational law include, but are not limited to: tax law, family benefits, pension computations, monetary penalties and private billing contracts. All of these are algorithms in disguise: the law (roughly) defines a function that produces

outcomes based on a set of inputs.

As such, one might think computational law would be easily translatable into a computer program. Unfortunately, formidable challenges remain. First, as mentioned above, the law is the result of a centuries-long process: its convoluted structure demands tremendous expertise and training to successfully navigate and understand, something that a computer programmer may not have. Second, the language in which legal statutes are drafted is so different from existing programming languages that a tremendous gap remains between the legal text and its implementation, leaving the door open for discrepancies, divergence and eventual bugs, all with dramatic societal consequences.

Examples abound. In France, the military's payroll computation involves 174 different bonuses and supplemental compensations. Three successive attempts were made to rewrite and modernize the military paycheck infrastructure; but with a complete disconnect between the military statutes and the implementation teams that were contracted, the system had to be scrapped [3]. Software engineering failures are perhaps a fact of life in the IT world; but in this particular case, actual humans bore the consequences of the failure, with military spouses receiving a 3-cent paycheck, or learning years later that they owe astronomical amounts to the French state. Perhaps more relevant to the current news, the US government issued a decree intended to provide financial relief to US taxpayers whose personal economic situation had been affected by the Covid-19 pandemic. Owing to an incorrect implementation by the Internal Revenue Service (IRS), nearly one million Americans received an incorrect Economic Impact Payment (EIP), or none at all [4].

Both examples are similar, in that a seemingly pure engineering failure turns out to have dramatic consequences in terms of human livelihoods. When a family is at the mercy of the next paycheck or EIP, a bug in those systems could mean bankruptcy. In our view, these is no doubt that these systems are yet another flavor of critical software.

A natural thought is perhaps to try to simplify the law itself. Unfortunately, this recurrent theme of the political discourse often conflicts with the political reality that requires compromise and fined-grained distinctions. Hence, the authors do not anticipate a drastic improvement around the world concerning legal complexity. Therefore, our only hope for improvement lies on the side of programming languages and tools.

Tax law provides a quintessential example. While many of the implementations around the world are shrouded in secrecy, the public occasionally gets a glimpse of the underlying infrastructure. Recently, Merigoux *et al.* [5]

reverse-engineered the computation of the French income tax, only to discover that the tax returns of an entire nation were processed using an antiquated system designed in 1990, relying on 80,000 lines of code written in a custom, in-house language, along with 6,000 lines of hand-written C directly manipulating tens of thousands of global variables. This particular situation highlights the perils of using the wrong tool for the job: inability to evolve, resulting in hand-written C patch-ups; exotic semantics which make reproducing the computation extremely challenging; and a lack of accountability, as the cryptic in-house language cannot be audited by anyone, except by the handful of experts who maintain it. This is by no means a "French exception": owing to an infrastructure designed while Kennedy was still president, the IRS recently handed over $300,000,000's worth of fraudulent refunds to taxpayers [6]. The rewrite, decades in planning, keeps being pushed further away in the future [7].

In this work, we propose a new language, **Catala**, tailored specifically for the purpose of faithfully, crisply translating computational law into executable specifications. **Catala** is designed to follow the existing structure of legal statutes, enabling a one-to-one correspondence between a legal paragraph and its corresponding translation in **Catala**. Under the hood, **Catala** uses prioritized default logic [8]; to the best of our knowledge, **Catala** is the first instance of a programming language designed with this logic as its core system. **Catala** has clear semantics, and compiles to a generic lambda-calculus that can then be translated to any existing language. We formalize the compilation scheme of **Catala** with F* and show that it is correct. In doing so, we bridge the gap between legal statutes and their implementation; we avoid the in-house language trap; and we provide a solid theoretical foundation to audit, reproduce, evaluate and reuse computational parts of the law. Our evaluation, which includes user studies, shows that **Catala** can successfully express complex sections of the US Internal Revenue Code and the French family benefits computation.

The benefits of using **Catala** are many: lawmakers and lawyers are given a formal language that accurately captures their intent and faithfully mirrors the prose; programmers can derive and distribute a canonical implementation, compiled to the programming language of their choice; citizens can audit, understand and evaluate computational parts of the law; and advocacy groups can shed more light on oftentimes obscure, yet essential, parts of civil society.

6.1.1. The Language of Legislative Drafting

Legal statutes are written in a style that can be confounding for a computer scientist. While a program's control-flow (as a first approximation) makes forward progress, statutes frequently back-patch previous definitions and re-interpret earlier paragraphs within different contexts. The result more closely resembles assembly with arbitrary jumps and code rewriting, rather than a structured language.

To illustrate how the law works, we focus on Section 121 of the US Internal Revenue Code [9], our running example throughout this paper. Section 121 is written in English, making it accessible to an international audience without awkward translations; it features every single difficulty we wish to tackle with **Catala**; and it is a well-studied and well-understood part of the tax law. We go through the first few paragraphs of the section; for each of them, we informally describe the intended meaning, then highlight the surprising semantics of the excerpt. These paragraphs are contiguous in the law; we intersperse our own commentary in-between the quoted blocks.

Section 121 is concerned with the "Exclusion of gain from sale of principal residence". In common parlance, if the taxpayer sells their residence, they are not required to report the profits as income, hence making such profits non-taxable. Paragraph (a) defines the exclusion itself.

> **(a) Exclusion**
> Gross income shall not include gain from the sale or exchange of property if, during the 5-year period ending on the date of the sale or exchange, such property has been owned and used by the taxpayer as the taxpayer's principal residence for periods aggregating 2 years or more.

The part of the sentence that follows the "if" enumerates conditions under which this tax exclusion can be applied. This whole paragraph is valid *unless specified otherwise*, as we shall see shortly.

Out-of-Order Definitions Paragraph (b) immediately proceeds to list *limitations*, that is, situations in which (a) does not apply, or needs to be refined. Section 121 thus consists of a general case, (a), followed by a long enumeration of limitations ranging from (b) to (g). We focus only on (b). The first limitation (b)(1) sets a maximum for the exclusion, "generally" $250,000. Left implicit is the fact that any proceeds of the sale beyond that are taxed as regular income.

> **(b) Limitations**
> **(1) In general**
> The amount of gain excluded from gross income under subsection (a) with respect to any sale or exchange shall not exceed $250,000.

We remark that even though (b)(1) is a key piece of information for the application of Section 121, the reader will find it only if they keep going after (a). This is a general feature of legal texts: relevant information is scattered throughout, and (a) alone is nowhere near enough information to make a determination of whether the exclusion applies to a taxpayer.

Backpatching; Exceptions Entering (b)(2), paragraph (A) *modifies* (b)(1) *in place*, stating for "joint returns" (i.e. married couples), the maximum exclusion can be $500,000.

> **(A) $500,000 Limitation for certain joint returns**
> Paragraph (1) shall be applied by substituting "$500,000" for "$250,000" if—
> (i) either spouse meets the ownership requirements of subsection (a) with respect to such property;
> (ii) both spouses meet the use requirements of subsection (a) with respect to such property; and
> (iii) neither spouse is ineligible for the benefits of subsection (a) with respect to such property by reason of paragraph (3).

Several key aspects of paragraph (A) are worth mentioning. First, (A) *backpatches* paragraph (b)(1); the law essentially encodes a search-and-replace in its semantics.

Second, (A) *overrides* a previous general case under specific conditions. In a functional programming language, a pattern-match first lists the most specific matching cases, and catches all remaining cases with a final wildcard. A text of law proceeds in the exact opposite way: the general case in (a) above is listed first, then followed by limitations that modify the general case under certain conditions. This is by design: legal statutes routinely follow a "general case first, special cases later" approach which mirrors the legislator's intentions.

Third, conditions (i) through (iii) are a conjunction, as indicated by the "and" at the end of (ii). We note that (iii) contains a forward-reference to (3) which we have not seen yet. (Through our work, we fortunately have never encountered a circular reference.)

Re-interpreting If limitation (A) doesn't apply, we move on to (B), which essentially stipulates that the exclusion in (b)(1) should be re-interpreted for each spouse separately *as if they were not married*; the final exclusion is then the sum of the two sub-computations.

> **(B) Other joint returns**
> If such spouses do not meet the requirements of subparagraph (A), the limitation under paragraph (1) shall be the sum of the limitations under paragraph (1) to which each spouse would be entitled if such spouses had not been married. For purposes of the preceding sentence, each spouse shall be treated as owning the property during the period that either spouse owned the property.

We thus observe that the law is re-entrant and can call itself recursively under different conditions. This is indicated here by the use of the conditional tense, i.e. "would".

Out-of-Order Backpatching In another striking example, (3) cancels the whole exclusion (a) under certain conditions.

> **(3) Application to only 1 sale or exchange every 2 years**
> Subsection (a) shall not apply to any sale or exchange by the taxpayer if, during the 2-year period ending on the date of such sale or exchange, there was any other sale or exchange by the taxpayer to which subsection (a) applied.

Paragraph (3) comes a little further down; a key takeaway is that, for a piece of law, one must process the *entire* document; barring that, the reader might be missing a crucial limitation that only surfaces much later in the text.

A Final Example Paragraph (4) concerns the specific case of a surviving spouse that sells the residence within two years of the death of their spouse, knowing that the conditions from (A) applied (i.e. "returned true") *right before the date of the death*.

> **(4) Special rule for certain sales by surviving spouses**
> In the case of a sale or exchange of property by an unmarried individual whose spouse is deceased on the date of such sale, paragraph (1) shall be applied by substituting "$500,000" for

> "$250,000" if such sale occurs not later than 2 years after the date of death of such spouse and the requirements of paragraph (2)(A) were met immediately before such date of death.

Paragraph (4) combines several of the complexities we saw above. It not only back-patches (1), but also recursively calls (2)(A) under a different context, namely, executing (2)(A) at a *previous date* in which the situation was different. In functional programming lingo, one might say that there is a hidden lambda in (2)(A), that binds the date of the sale.

Formal Insights on Legal Logic We have now seen how legal statutes are written, the thought process they exhibit, and how one is generally supposed to interpret them. We wish to emphasize that the concepts described are by no means specific to tax law or the US legal system: we found the exact same patterns in other parts of US law and non-US legal systems. Section 121 contains many more paragraphs; however, the first few we saw above are sufficient to illustrate the challenges in formally describing the law.

The main issue in modeling legal texts therefore lies in their underlying logic, which relies heavily on the pattern of having a default case followed by exceptions. This nonmonotonic logic is known as *default logic* [10]. Several refinements of default logic have been proposed over time; the one closest to the purposes of the law is known as prioritized default logic [8], wherein default values are guarded by justifications, and defaults can be ordered according to their relative precedence. Lawsky [11] argues that this flavor of default logic is the best suited to expressing the law. We concur, and adopt prioritized default logic as a foundation for **Catala**.

In default logic, formulas include defaults, of the form $a : \vec{b_i}/c$, wherein: if formula a holds; if the $\vec{b_i}$ are consistent with the set of known facts; then c holds. One can think of a as the precondition for c, and the $\vec{b_i}$ as a set of exceptions that will prevent the default fact c from being applicable. Prioritized logic adds a strict partial order over defaults, to resolve conflicts when multiple defaults may be admissible at the same time.

The main design goal of **Catala** is to provide the design and implementation of a language tailored for the law, using default logic as its core building block, both in its syntax and semantics. **Catala** thus allows lawyers to express the general case / exceptions pattern naturally. We now informally present **Catala**.

Our introduction to legal texts above mixes informal, high-level overviews

of what each paragraph intends to express, along with excerpts from the law itself. Our English prose is too informal to express anything precisely; but "legalese" requires a high degree of familiarity with the law to successfully grasp all of the limitations and compute what may or may not be applicable to a given taxpayer's situation.

We now introduce **Catala** by example, and show how the subtleties of each paragraph can be handled unambiguously and clearly by **Catala**. Our guiding principle is twofold: we want to formally express the intended meaning without being obscured by the verbosity of legal prose; yet, we wish to remain close to the legal text, so that the formal specification remains in close correspondence with the reference legal text, and can be understood by lawyers. **Catala** achieves this with a dedicated surface language that allows legal experts to follow their usual thinking.

Metadata: Turning Implicits into Explicits Legal prose is very dense, and uses a number of concepts without explicitly defining them in the text. For instance, in Section 121, the notion of time period is implicit, and so are the various types of tax returns one might file (individual or joint). Furthermore, entities such as the taxpayers (whom we will call "Person 1" and "Person 2") need to be materialized. Finally, for each one of those entities, there are a number of inputs that are implicitly referred to throughout the legal statute, such as: time periods in which each Person was occupying the residence as their primary home; whether there was already a sale in the past two years; and many more, as evidenced by the myriad of variables involved in (i)-(iii).

Our first task when transcribing legal prose into a formal **Catala** description is thus to enumerate all structures, entities and variables relevant to the problem at stake. We provide the definitions and relationships between variables later on. This is a conscious design choice of **Catala**: before even talking about *how* things are computed, we state *what* we are talking about. In doing so, we mimic the behavior of lawyers, who are able to infer what information is relevant based on the legal text. We call this description of entities the *metadata*.

```
1 declaration structure Period:
2   data start content date
3   data end content date
4
5 declaration structure PersonalData:
```

```
6     data property_ownership content collection Period
7     data property_usage_as_principal_residence content
8       collection Period
9
10  declaration scope Section121SinglePerson:
11    context gain_from_sale_or_exchange_of_property
12      content money
13    context personal content PersonalData
14    context requirements_ownership_met condition
15    context requirements_usage_met condition
16    context requirements_met condition
17    context amount_excluded_from_gross_income_uncapped
18      content money
19    context amount_excluded_from_gross_income content money
20
21    context aggregate_periods_from_last_five_years content
22      duration depends on collection Period
```

Catala features a number of built-in types. **date**s are triples of a year, month and a day. **Catala** provide syntax for US-centric and non-US-centric input formats. Distinct from **date** is **duration**, the type of a time interval, always expressed as a number of days. If the law introduces durations such as "two years", it is up to the user to specify how "two years" should be interpreted. Booleans have type **condition**. Monetary amounts have type **money**. The higher-kinded type **collection** is also built-in.

The snippet above shows an excerpt from Section 121's metadata. The first two **declaration**s declare product types via the **structure** keyword. The type **Period** contains two fields, **start** and **end**.

A word about aesthetics: while programmers generally prize compactness of notation, advocating e.g. point-free-styles or custom operators, non-experts are for the most part puzzled by compact notations. Our surface syntax was designed in collaboration with lawyers, who confirmed that the generous keywords improve readability, thus making **Catala** easier to understand by legal minds.

Line 10 declares **Section121SinglePerson**, a **scope**. A key technical device and contribution of **Catala**, scopes allow the programmer to follow the law's structure, revealing the implicit modularity in legal texts. Scopes are declared in the metadata section: the **context** keyword indicates that

the value of the field might be determined later, *depending on the context*. Anticipating on Section 6.2.2, the intuition is that `scope`s are functions and `context`s are their parameters and local variables.

Context variables are declarative; once a **Catala** program is compiled to a suitable target language, it is up to the programmer to invoke a given scope with suitable initial values for those context variables that are known to be inputs of the program; after the **Catala** program has executed, the programmer can read from the context variables that they know are outputs of the program. From the point of view of **Catala**, there is no difference between input and output variables; but we are planning a minor syntactic improvement to allow programmers to annotate these variables for readability and simpler interoperation with hand-written code. If at run-time the program reads the value of a context variable that was left unspecified by the programmer, execution aborts.

The main purpose of Section 121 is to talk about the gain that a person derived from the sale of their residence (line 11), of type `money`. Paragraph (a) implicitly assumes the existence of time periods of ownership and usage of the residence; we materialize these via the `personal` field which holds two `collection Period`s. These in turn allow us to define whether the ownership and usage requirements are met (of type `condition`, lines 14-15). A further condition captures whether *all* requirements are met (line 16). The outcome of the law is the amount that can be excluded from the gross income, of type `money` (line 19). (The need for an intermediary variable at line 18 becomes apparent when talking about split scopes.) A local helper computes the aggregate number of days in a set of time periods; the helper takes a single argument of type `collection Period` (line 21) and, being a local closure, can capture other `context` variables.

Scopes and Contexts: Declarative Rules and Definitions We now continue with our formalization of (a) and define the context-dependent variables, as well as the relationships between them. **Catala** is declarative: the user relates `context` variables together, via the `definition` keyword, or the `rule` keyword for `condition`s. We offer separate syntax for two reasons. First, for legal minds, conditions and data are different objects and reflecting that in the surface syntax helps with readability. Second, there is a core semantic difference: booleans (conditions) are false by default in the law; however, other types of data have no default value. Internally, **Catala** desugars `rule`s to

definitions equipped with a default value of **false** (see Section 6.2.1).

```
1  scope Section121SinglePerson:
2    rule requirements_ownership_met under condition
3      aggregate_periods_from_last_five_years of
4         personal.property_ownership
5         >=^ 730 day
6    consequence fulfilled
7    rule requirements_usage_met under condition
8      aggregate_periods_from_last_five_years of
9        personal.property_usage_as_principal_residence
10       >=^ 730 day
11   consequence fulfilled
12   rule requirements_met under condition
13     requirements_ownership_met and requirements_usage_met
14   consequence fulfilled
15   definition amount_excluded_from_gross_income_uncapped
16     equals
17     if requirements_met then
18       gain_from_sale_or_exchange_of_property
19     else $0
```

Lines 2-4 capture the ownership requirement, by calling the helper function **aggregate_periods_...** with argument **property_ownership**, a previously-defined context variable. (The full definition of the helper, which involves another context variable for the date of sale, is available in the artifact [12].) Paragraph (a) states "for periods aggregating 2 years or more": for the purposes of Section 121, and as defined in Regulation 1.121-1(c)(1), a year is always 365 days. **Catala** resolves the ambiguity by simply not offering any built-in notion of yearly duration, and thus makes the law clearer. The **^** suffix of the comparison operator **>=^** means that we are comparing durations.

The ownership requirement is "fulfilled" (i.e. defined to **true**) under a certain condition. This is our first taste of prioritized default logic expressed through the syntax of **Catala**: the built-in default, set to **false**, is overridden with a rule that has higher priority. This is a simple case and more complex priorities appear in later sections. However, this example points to a key insight of **Catala**: rather than having an arbitrary priority order resolved at run-time between various **rules**, we encode priorities statically in the surface syntax

of the language, and the pre-order is derived directly from the syntax tree of rules and definitions. We explain this in depth later on Section 6.2.1.

Similarly, lines 5-7 define the usage requirement using the **`rule`** keyword to trigger a **`condition`**: the value of **`requirements_usage_met`** is **`false`** unless the boolean expression at lines 6-7 is **`true`**. One legal subtlety, implicit in (a), is that ownership and usage periods do not have to overlap. The **Catala** program makes this explicit by having two collections of time periods.

The requirements are met if both ownership and usage requirements are met (lines 9-11). In that case, the income gain can be excluded from the income tax (lines 12-13). The latter is defined via the **`definition`** keyword, as **`rule`** is reserved for booleans.

We have now formalized Paragraph (a) in **Catala**. At this stage, if the user fills out the remaining inputs, such as the gain obtained from the sale of the property, and the various time periods, the interpreter automatically computes the resulting value for the amount to be excluded from the gross income. The interpreter does so by performing a control-flow analysis and computing a topological sort of assignments. Cycles are rejected, since the law is not supposed to have dependency cycles. (Section 6.2.2 describes the full semantics of the language.)

We note that a single sentence required us to explicitly declare otherwise implicit concepts, such as the definition of a year; and to clarify ambiguities, such as whether time periods may overlap. With this concise example, we observe that the benefits of formalizing a piece of law are the same as formalizing any piece of critical software: numerous subtleties are resolved, and non-experts are provided with an explicit, transparent executable specification that obviates the need for an expert legal interpretation of implicit semantics.

Split Scopes: Embracing the Structure of the Law We now move on to limitations (paragraph (b) of Section 121). A key feature of **Catala** is that it enables a literate programming style [13], where each paragraph of law is immediately followed by its **Catala** transcription. Now that we're done with (a), we insert a textual copy of the legal prose for (b), then proceed to transcribe it in **Catala**.

```
1 scope Section121SinglePerson:
2   definition gain_cap equals $250,000
3   definition amount_excluded_from_gross_income equals
4     if amount_excluded_from_gross_income_uncapped >=$
5       gain_cap
```

```
6       then gain_cap
7       else amount_excluded_from_gross_income_uncapped
```

In Paragraph (b)(1), the law overwrites the earlier definition from (a) and redefines it to be capped by $250,000. In line with our earlier design choices, we rule out confusion and rely on the auxiliary variable (the "uncapped" variant), to then compute the final amount excluded from the gross income (lines 3-7). Out-of-order definitions that are provided at a later phase in the source are an idiomatic pattern in **Catala**.

Complex Usage of the Default Calculus; Exceptions Before making any further progress, we need to introduce new entities to take into account the fact that we may now possibly be dealing with a joint return. We introduce a new abstraction or, in **Catala** lingo, scope: **`Section121Return`**.

```
1  declaration structure CoupleData:
2    data personal1 content PersonalData
3    data personal2 content PersonalData
4
5  declaration enumeration ReturnType:
6    -- SingleReturn content PersonalData
7    -- JointReturn content CoupleData
8
9  declaration scope Section121Return:
10   context return_data content ReturnType
11   context person1 scope Section121SinglePerson
12   context person2 scope Section121SinglePerson
13   context gain_cap content money
```

We follow the out-of-order structure of the law; only from here on do we consider the possibility of a joint return. Having introduced a new level of abstraction, we need to relate the **`ReturnType`** to the persons involved. We do so by introducing new equalities, of which we show the first one.

```
1  scope Section121Return:
2    definition person1.personal equals match return_data with
3    -- SingleReturn of personal1 : personal1
4    -- JointReturn of couple : couple.personal1
```

Having set up a proper notion of joint return, we now turn our attention to (b)(2)(A).

```
1  scope Section121Return:
2    definition gain_cap equals person1.gain_cap
3    rule paragraph_A_applies under condition
4      (return_data is JointReturn) and
5      (person1.requirements_ownership_met or
6        person2.requirements_ownership_met) and
7      (person1.requirements_usage_met and
8        person2.requirements_usage_met) and
9      (not (paragraph_3_applies of
10       person1.other_section_121a_sale)) and
11     (not (paragraph_3_applies of
12       person2.other_section_121a_sale))
13   consequence fulfilled
14   exception definition gain_cap under condition
15     paragraph_A_applies consequence equals $500,000
```

Until now, the gain cap was defined to be that of the taxpayer, that is, Person 1 (line 2). We now need to determine whether the conditions from Paragraph (A) apply (line 3). To that end, we introduce an intermediary variable, **`paragraph_A_applies`**. This variable will be used later on for (B), whose opening sentence is "if such spouses do not meet the requirements of (A)".

We now introduce the notion of **`exception`** (line 14). In **Catala**, if, at run-time, more than a single applicable definition for any **`context`** variable applies, program execution aborts with a fatal error. In the absence of the **`exception`** keyword, and in the presence of a joint return that satisfies paragraph (A), the program would be statically accepted by **Catala**, but would be rejected at run-time: there are two definitions for **`gain_cap`**, both their conditions hold (**`true`** and **`paragraph_A_applies`**), and there is no priority ordering indicating how to resolve the conflict. The **`exception`** keyword allows solving this very issue. The keyword indicates that, in the pre-order of definitions, the definition at line 14 has a higher priority than the one at 2.

Generally, **Catala** allows an arbitrary tree of definitions each refined by exceptions, including exceptions to exceptions (which we have encountered in the law); the rule of thumb remains: only one single definition should be

applicable at a time, and any violation of that rule indicates either programmer error, or a fatal flaw in the law.

Recapping Modeling the law is labor-intensive, owing to all of the implicit assumptions present in what is seemingly "just legalese". In our experience, this process is best achieved through pair-programming, in which a **Catala** expert transcribes a statute with the help of a legal expert. We thus stop here our **Catala** tutorial and defer the full modelization of Section 121 to the artifact [12]. Briefly, modeling (B) requires introducing a new scope for a two-pass processing that models the re-entrancy ("if such spouses had not been married"). Modeling the forward-reference to (3) requires introducing a helper `paragraph_3_applies` whose `definition` is provided later on, after Paragraph (3) has been suitably declared (line 10, above).

As this tutorial wraps up, we look back on all of the language features we presented. While **Catala** at first glance resembles a functional language with heavy syntactic sugar, diving into the subtleties of the law exhibits the need for two features that are not generally found in lambda-calculi. First, we allow the user to define context variables through a combination of an (optional) default case, along with an arbitrary number of special cases, either prioritized or non-overlapping. The theoretical underpinning of this feature is the *prioritized default calculus*. Second, the out-of-order nature of definitions means that **Catala** is entirely declarative, and it is up to the **Catala** compiler to compute a suitable dependency order for all the definitions in a given program. Fortunately, the law does not have general recursion, meaning that we do not need to compute fixed points, and do not risk running into circular definitions. Hence, our language is not Turing-complete, purposefully.

We mentioned earlier that we have found both US and French legal systems to exhibit the same patterns in the way their statutes are drafted. Namely, the general case / exceptions control-flow; out-of-order declarations; and overrides of one scope into another seem to be universal features found in all statutes, regardless of the country or even the language they are drafted in. Based on conversations with S. Lawsky, and a broader assessment of the legal landscape, we thus posit that **Catala** captures the fundamentals of legal reasoning.

6.1.2. A Century of Language Design

Catala follows a long tradition of scholarly works that aim to extract the logical essence of legal statutes, starting as early as 1924 [14]. To provide some context, we compare our work with two seminal articles in the field.

In his visionary 1956 article, Allen [15] notes that symbolic logic can be used to remove ambiguity in the law, and proposes its use for a wide range of applications: legal drafting, interpretation, simplification and comparison. Using custom notations that map transparently to first-order logic, Allen does not provide an operational tool to translate law into formalism but rather points out the challenges such as law ambiguity and rightfully sets the limits of his approach, stating for instance that in generality, "filling of gaps in legislation by courts cannot and should not be entirely eliminated". Interestingly, he also manually computes a truth table to prove that two sections of the US Internal Revenue Code are equivalent.

The vision laid out by Allen is refined in 1986 by Sergot *et al.* [16]. This article narrows the range of its formalism to statutory law (as opposed to case law), and focuses on the British Nationality Act, a statute used to determine whether a person can qualify for the British nationality based on various criteria. Co-authored by Robert Kowalski, this works features the use of Prolog [17] as the target programming language, showing the usefulness of declarative logic programming for the formalization task. However, the work acknowledges a major limitation concerning the expression of negation in the legal text, and points out that "the type of default reasoning that the act prescribes for dealing with abandoned infants is non-monotonic", confirming the later insights of Lawsky [11]. A major difference with **Catala** is the absence of literate programming; instead, Sergot *et al.* derived a synthetic and/or diagram as the specification for their Prolog program.

The logic programming community later built on non-monotonicity by proposing increasingly refined languages for legal discourse [18], [19]. These languages all choose some form of deontic logic [20] as formal base, as its categorization between obligation and permission fits legal reasoning. By extending deontic logic with non-monotonic defeasability [21], it is possible to model very accurately the reasoning subtleties of real-world legal cases [22]. The community did not however reach an consensus over the correct way to formally model legal argumentation in theory, as argued by McCarty [23].

Thus, the line of work around logic programming never took hold in the industry and the large organizations managing legal expert systems. The

reasons, various and diagnosed by Leith [24], mix the inherent difficulty of translating law to code, with the social gap between the legal and computer world. As a reaction, several and so far unsuccessful attempts were made to automate the translation using natural language processing techniques [25], [26]. Others propose to lower the barriers to the programming world using low-code/no-code tools, so that lawyers can code their own legal expert systems [27].

The main recent research direction around the formalization of law is spearheaded by optimistic proponents of computational law [28], promising a future based on Web-based, automated legal reasoning by autonomous agents negotiating smart contracts on a blockchain-powered network [29]–[32].

By contrast, we focus on the challenges related to maintaining existing legal expert systems in large public or private organizations, providing essential services to millions of citizens and customers. **Catala** aims to provide an industrial-grade tool that enables close collaboration of legal and IT professionals towards the construction of correct, comprehensive and performant implementations of algorithmic statutory law. The formal foundation of **Catala** is inspired by the prior logic programming research. More precisely, it exactly implements McCarty suggestion of 1997 [23]: "If we used only stratified negation-as-failure with metalevel references in our representation language, we would have a powerful normal form for statutes and regulations."

6.2. Catala as a Formalization Platform

We now formally introduce the semantics and compilation of **Catala**. Notably, we focus on what makes **Catala** special: its default calculus. To that end, we describe a series of compilation steps: we desugar the concrete syntax to a *scope language*; we define the semantics of scopes via a translation to a *default calculus*; we then finally compile the *default calculus* to a language equipped with exceptions, such as OCaml. This last part is where the most crucial compilation steps occur: we prove its soundness via a mechanization in the F* proof assistant.

6.2.1. Default Calculus, a Simple Desugared Core

The scope language resembles **Catala**'s user-facing language: the notion of scope is still present; `rule`s and `definition`s remain, via a unified `def` dec-

Scope name	S			
Sub-scope instance	S_n			
Location	ℓ	::=	x	scope variable
		|	$S_n\,[x]$	sub-scope variable
Type	τ	::=	`bool` | `unit`	base types
		|	$\tau \to \tau$	function type
Expression	e	::=	x | `true` | `false` | `()`	variable, literals
		|	$\lambda\ (x : \tau)\ .\ e$ | $e\ e$	λ-calculus
		|	ℓ	location
		|	d	default term
Default	d	::=	$\langle \vec{e} \mid e :\!- e \rangle$	default term
		|	$\circledast$	conflict error term
		|	$\varnothing$	empty error term
Atom	a	::=	`def` $\ell : \tau = \langle \vec{e} \mid e :\!- e \rangle$	location definition
		|	`call` S_n	sub-scope call
Scope declaration	σ	::=	`scope` $S : \vec{a}$	
Program	P	::=	$\vec{\sigma}$	

Figure 6.1: The scope language, our first intermediate representation

laration devoid of any syntactic sugar. Perhaps more importantly, definitions are provided in-order and our usage of default calculus becomes clear.

Figure 6.1 presents the syntax of the scope language. We focus on the essence of **Catala**, i.e. how to formalize a language with default calculus at its core; to that end, and from this section onwards, we omit auxiliary features, such as data types, in order to focus on a core calculus.

To avoid carrying an environment, a reference to a sub-scope variable, such as **`person1.personal`** earlier, is modeled as a reference to a sub-scope annotated with a unique identifier, such as **`Section121SinglePerson`**$_1$.personal. Therefore, locations are either a local variable x, or a sub-scope variable, of the form $S_n\,[x]$. Note that sub-scoping enables scope calls nesting in all generality. However, we do not allow in our syntax references to sub-scopes' sub-scopes like $S_n\,[S'_{n'}\,[x]\,]$, as this would unnecessarily complicate our semantics model.

Types and expressions are standard, save for default terms d of the form $\langle \vec{e_i} \mid e' :\!- e'' \rangle$. This form resembles default logic terms $a : \vec{b_i}/c$ introduced earlier (Section 6.1.1); the *exceptions* $\vec{b_i}$ become $\vec{e_i}$; the *precondition* a becomes e'; and the *consequence* c becomes e''. We adopt the following reduction semantics for d. Each of the exceptions e_i is evaluated; if two or more are valid

(i.e. not of the form $\varnothing$), a conflict error $\circledast$ is raised. If exactly one exception e_i is valid, the final result is e_i. If no exception is valid, and e' evaluates to `true` the final result is e''. If no exception is valid, and e' evaluates to `false`, the final result is $\varnothing$. We provide a full formal semantics of default terms later.

The syntactic form $\langle \vec{e_i} \mid e' \text{ :- } e'' \rangle$ encodes a *static* tree of priorities, baking the pre-order directly in the syntax tree of each definition. We thus offer a restricted form of prioritized default logic, in which each definition is its own world, equipped with a static pre-order.

Atoms a either define a new location, gathering all default cases and exceptions in a single place; or, rules indicate that a sub-scope needs to be called to compute further definitions.

We now explain how to desugar the surface syntax, presented in Section 6.1.1, to this scope language.

Syntactic sugar Table 6.1 presents rewriting rules, whose transitive closure forms our desugaring. These rules operate within the surface language; Table 6.1 abbreviates surface-level keywords for easier typesetting.

In its full generality, **Catala** allows exceptions to definitions, followed by an arbitrary nesting of exceptions to exceptions. This is achieved by a label mechanism: all exceptions and definitions are *labeled*, and each exception refers to the definition or exception it overrides. Exceptions to exceptions are actually found in the law, and while we spared the reader in our earlier tutorial, we have found actual use-cases where this complex scenario was needed. Exceptions to exceptions remain rare; the goal of our syntactic sugar is to allow for a more compact notation in common cases, which later gets translated to a series of fully labeled definitions and exceptions.

After desugaring, definitions and exceptions form a forest, with exactly one root **`definition`**s node for each variable X, holding an n-ary tree of **`exception`** nodes.

We start with the desugaring of **`rule`** which, as mentioned earlier, is a boolean definition with a base case of **`false`** (i). Definitions without conditions desugar to the trivial `true` condition (ii).

The formulation of (iiia) allows the user to provide multiple definitions for the same variable X without labeling any of them; thanks to (iiia), these are implicitly understood to be a series of exceptions without a default case. The surface syntax always requires a default to be provided; internally, the **`nodefault`** simply becomes **`condition true consequence equals`** $\varnothing$.

Table 6.1.: Desugaring the surface language into an explicit surface syntax

	Syntactic sugar	**...rewrites to**
(i)	`rule X under cond. Y cons. fulfilled`	`label` $\mathtt{L}_X$ `def. X equals false` (inserted once) `exception` $\mathtt{L}_X$ `def. X under cond. Y cons. equals true`
(ii)	`def. X equals Y`	`def. X under cond. true cons. equals Y`
(iiia)	`def. X ...` (multiple definitions of `X`, no exceptions)	`label` $\mathtt{L}_X$ `def. X nodefault` (inserted once) `exception` $\mathtt{L}_X$ `def. X ...`
(iiib)	`def. X ...` (single definition of `X`)	`label` $\mathtt{L}_X$ `def. X ...`
(iv)	`exception def. X`	`exception` $\mathtt{L}_X$ `def. X`

We provide another alternative to the fully labeled form via (iiib); the rule allows the user to provide a single base definition, which may then be overridden via a series of exceptions. To that end, we introduce a unique label L_X which un-annotated exceptions are understood to refer to (iv).

Materializing the default tree Equipped with our default expressions d, we show how to translate a scattered series of **Catala** definitions into a single **`def`** rule from the scope language. We write $X, L \rightsquigarrow d$, meaning that the definition of X labeled L, along with all the (transitive) exceptions to L, collectively translate to d. We use an auxiliary helper $\mathsf{lookup}(X, L) = C, D, \vec{L}_i$, meaning that at label L, under condition C, X is defined to be D, subject to a series of exceptions labeled L_i.

Rule D-LABEL performs the bulk of the work, and gathers the exception labels L_i; each of them translates to a default expression d_i, all of which appear on the left-hand side of the resulting translation; if all of the d_i are empty, the expression evaluates to D guarded under condition C. As an illustration, if no exceptions are to be found, the translation is simply $\langle \ | \ C :\!- D \rangle$. Finally, rule D-ENTRYPOINT states that the translation starts at the root **`definition`** nodes.

$$\textsc{D-Label}\quad \frac{\mathsf{lookup}(X, L) = C, D, \vec{L}_i \qquad X, L_i \rightsquigarrow d_i}{X, L \rightsquigarrow \langle \vec{d}_i \quad | \quad C \ :- \ D \rangle}$$

$$\textsc{D-EntryPoint}\quad \frac{X, L \rightsquigarrow \langle \vec{d}_i \quad | \quad C \ :- \ D \rangle}{\mathsf{label}\ L\ \mathsf{definition}\ X \ \ldots \rightsquigarrow \texttt{def}\ X = \langle \vec{d}_i \quad | \quad C \ :- \ D \rangle}$$

Figure 6.2: Building the default tree and translating surface definitions

Reordering definitions Our final steps consists in dealing with the fact that `def`s remain unordered. To that end, we perform two topological sorts. First, for each scope S, we collect all definitions and re-order them according to a local dependency relation $\rightarrow$:

$$\begin{cases} y \rightarrow x & \text{if def } x = \ldots y \ldots \\ S_n \rightarrow x & \text{if def } x = \ldots S_n[y] \ldots \\ y \rightarrow S_n & \text{if def } S_n[x] = \ldots y \ldots \end{cases}$$

After re-ordering, we obtain a scope S where definitions can be processed linearly. Sub-scope nodes of the form S_n become `call`s, to indicate the position at which the sub-scope computation can be performed, i.e. once its parameters have been filled and before its outputs are needed.

We then topologically sort the scopes themselves to obtain a linearized order. We thus move from a declarative language to a functional language where programs can be processed in evaluation order. In both cases, we detect the presence of cycles, and error out. General recursion is not found in the law, and is likely to indicate an error in modeling. Bounded recursion, which we saw in Section 6.1.1, can be manually unrolled to make it apparent.

From the Scope Language to a Default Calculus For the next step of our translation, we remove the scope mechanism, replacing `def`s and `call`s with regular λ-abstractions and applications. The resulting language, a core lambda calculus equipped only with default terms, is the *default calculus* (Figure 6.3). The typing rules of the default calculus are standard (Figure 6.4); we note that the error terms from the default calculus are polymorphic.

Type	τ	::=	`bool` | `unit`	boolean and unit types
		|	$\tau \to \tau$	function type
Expression	e	::=	x | s | `true` | `false` | `()`	variable, top-level name, literals
		|	$\lambda\ (x : \tau)\ .\ e$ | $e\ e$	λ-calculus
		|	d	default term
Default	d	::=	$\langle \vec{e} \mid e :\!- e \rangle$	default term
		|	$\circledast$	conflict error term
		|	$\varnothing$	empty error term
Top-level declaration	σ	::=	`let` s = e	
Program	P	::=	$\vec{\sigma}$	

Figure 6.3: The default calculus, our second intermediate representation

$$\textsc{ConflictError}\quad \Gamma \vdash \circledast : \tau \qquad \textsc{EmptyError}\quad \Gamma \vdash \varnothing : \tau$$

$$\textsc{T-Default}\quad \frac{\Gamma \vdash e_i : \tau \qquad \Gamma \vdash e_{\text{just}} : \texttt{bool} \qquad \Gamma \vdash e_{\text{cons}} : \tau}{\Gamma \vdash \langle e_1, \ldots, e_n \mid e_{\text{just}} :\!- e_{\text{cons}} \rangle : \tau}$$

Figure 6.4: Typing rules for the default calculus

Reduction rules We present small-step operational semantics, of the form $\boxed{e \longrightarrow e'}$. For efficiency, we describe reduction under a context, using a standard notion of value (Figure 6.5), which includes our two types of errors, $\circledast$ and $\varnothing$. We intentionally distinguish regular contexts C_λ from general contexts C.

Figure 6.6 presents the reduction rules for the default calculus. Rule D-ContextConflictError follows standard call-by-value reduction rules for non-error terms; D-Beta needs no further comment. $\circledast$ is made fatal by D-ContextConflictError: the reduction aborts, under *any context* C. The behavior of $\varnothing$ is different: such an error propagates only up to its enclosing "regular" context C_λ; this means

Values	v	::=	$\lambda\ (x : \tau)\ .\ e$	functions
		|	`true` | `false`	booleans
		|	$\circledast$ | $\varnothing$	errors
Evaluation contexts	C_λ	::=	$\cdot\ e$ | $v\ \cdot$	function application evaluation
		|	$\langle \vec{v} \mid \cdot :\!- e \rangle$	default justification evaluation
		|	$\langle \vec{v} \mid \texttt{true} :\!- \cdot \rangle$	default consequence evaluation
	C	::=	C_λ	regular contexts
		|	$\langle \vec{v}, \cdot, \vec{e} \mid e :\!- e \rangle$	default exceptions evaluation

Figure 6.5: Evaluation contexts for the default calculus

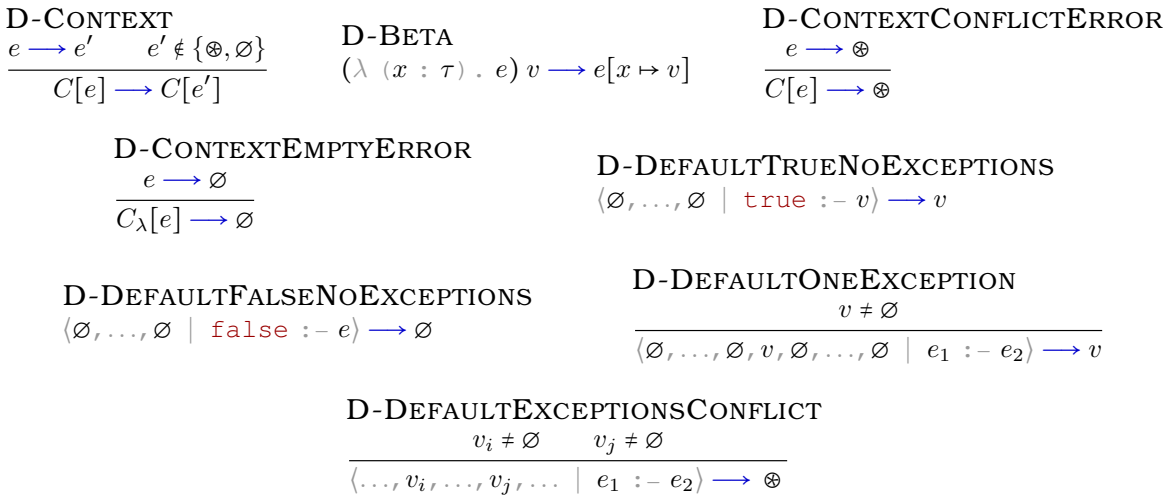

Figure 6.6: Reduction rules for the default calculus

that such an $\varnothing$-error can be caught, as long as it appears in the exception list of an enclosing default expression. Therefore, we now turn our attention to the rules that govern the evaluation of default expressions.

If no exception is valid, i.e. if the left-hand side contains only $\varnothing$s; and if after further evaluation, the justification is `true` for the consequence v, then the whole default reduces to v (D-DefaultTrueNoExceptions). If no exception is valid, and if the justification is `false`, then we do not need to evaluate the consequence, and the default is empty, i.e. the expression reduces to $\varnothing$. If *exactly* one exception is a non-empty value v, then the default reduces to v. In that case, we evaluate neither the justification or the consequence (D-DefaultOneException). Finally, if two or more exceptions are non-empty, we cannot determine the priority order between them, and abort program execution (D-DefaultExceptionsConflict).

Compiling the scope language We succinctly describe the compilation of the scope language to the default calculus in Figure 6.7. Our goal is to get rid of scopes in favor of regular lambda-abstractions, all the while preserving the evaluation semantics; incorrect order of evaluation might lead to propagating premature errors (i.e. throwing exceptions too early).

We assume for simplicity of presentation that we are equipped with tuples, where $(x_1, \ldots, x_n)$ is concisely written $(\vec{x})$. We also assume that we are equipped with let-bindings, of the form `let` $(x_1, \ldots, x_n) = e$, for which we adopt the same

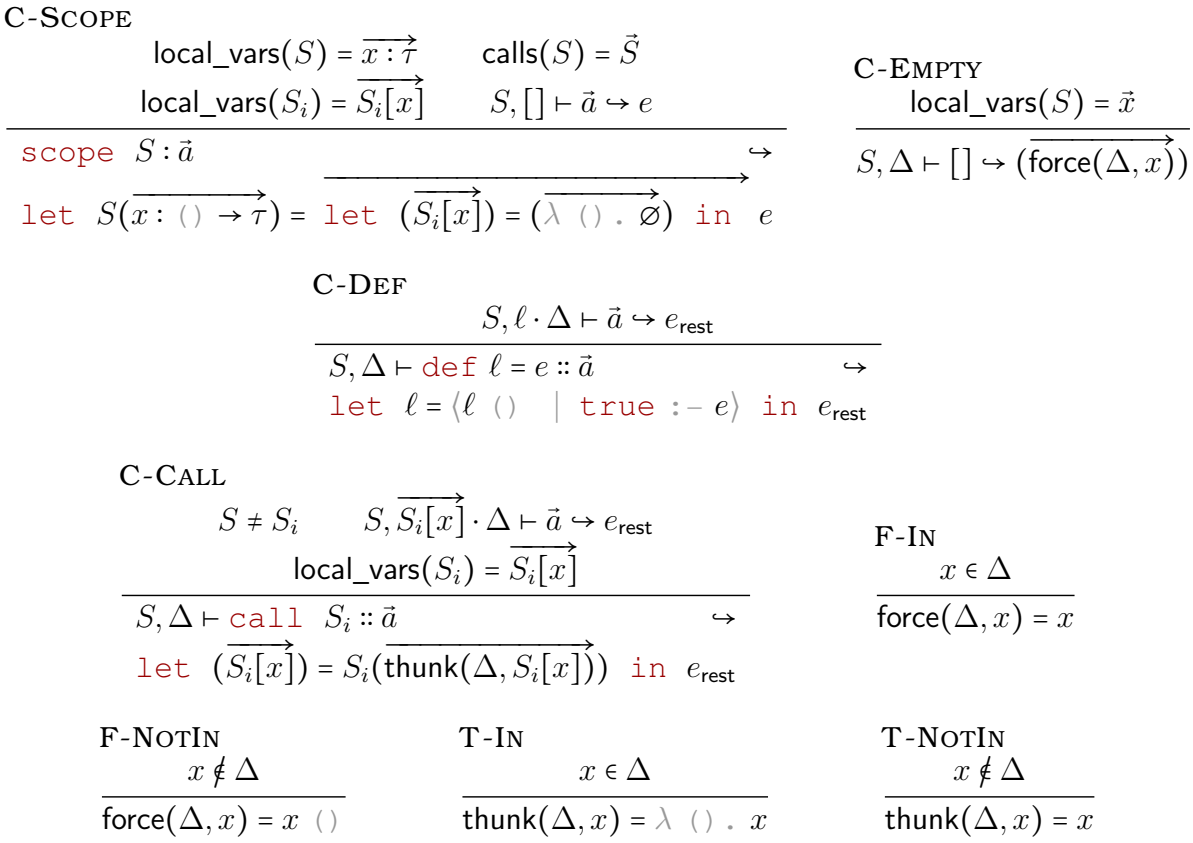

Figure 6.7: Compiling the scope language to a default calculus

concise notation. Given a scope S made up of atoms $\vec{a}$, $\mathsf{local\_vars}(S)$ returns all variables x for which $\texttt{def}\ x \in \vec{a}$. Note that this does not include rules that override sub-scope variables, which are of the form $\texttt{def}\ S_n[x]$. We now turn to our judgement, of the form $\boxed{S, \Delta \vdash a \hookrightarrow e}$, to be read as "in the translation of scope S, knowing that variables in Δ have been forced, scope rule r translates to default calculus expression e".

A scope S with local variables $\vec{x}$ compiles to a function that takes an n-tuple $(\vec{x})$ containing *potential overrides* for all of its context variables (C-Scope). In the translation, each x therefore becomes a thunk, so as to preserve reduction semantics: should the caller decide to leave a local variable x_i to be $\varnothing$, having a thunk prevents D-ContextEmptyError from triggering and returning prematurely. Rule C-Scope performs additional duties. For each one of the sub-scopes S_i used by S, we set all of the arguments to S_i, denoted $\overrightarrow{S_i[x]}$, to be a thunked $\varnothing$ to start with.

Advancing through the scope S, we may encounter definitions or calls. For definitions (C-Def), we simply insert a shadowing let-binding, and record that ℓ has been forced by extending Δ. Whether ℓ is of the form x or $S_n[x]$, we know that the previous binding was thunked, since our previous desugaring guarantees that any variable ℓ now has a single definition. The rewritten default expression gives the caller-provided argument higher precedence; barring any useful information provided by the caller, we fall back on the existing definition e. This key step explains how our law-centric syntax, which allows caller scopes to override variables local to a callee scope, translates to the default calculus.

For calls (C-Call), we ensure all of the arguments are thunked before calling the sub-scope; the return tuple contains *forced* values, which we record by extending Δ with all $\overrightarrow{S_i[x]}$. The premise $S \neq S_i$ captures the fact that recursion is not allowed.

Finally, after all rules have been translated and we are left with nothing but the empty list [] (C-Empty), we simply force all scope-local variables $\vec{x}$ and return them as a tuple.

Discussion of Design Choices The current shape of **Catala** represents the culmination of a long design process for the language. We now discuss a few of the insights we gained as we iterated through previous versions of **Catala**.

For the surface language, a guiding design principle was to always guarantee that the **Catala** source code matches the structure of the law. We have

managed to establish this property not only for Section 121 (Section 6.1.1), but also for all of the other examples we currently have in the **Catala** repository. To achieve this, a key insight was the realization that every piece of statutory law we looked at (in the US and French legal systems) follows a general case / exceptions drafting style. This style, which we saw earlier, means that the law encodes a *static* priority structure. Intuitively, computing a given value boils down to evaluating an n-ary *tree* of definitions, where nodes and edges are statically known. The children of a given node are mutually-exclusive *exceptions* to their parent definition; either one exception applies, and the computation stops. Or if no exception applies, the parent definition is evaluated instead. This recursive evaluation proceeds all the way up to the root of the tree, which represents the initial default definition.

The surface language was crafted to support encoding that tree of exceptions within the syntax of the language via the label mechanism. This informed our various choices for the syntactic sugar; notably, we make it easy to define n mutually-exclusive definitions in one go thanks to syntactic sugars (i) and (iiia) in Table 6.1.

A consequence of designing **Catala** around a *static* tree of exceptions for each defined value is that the internal default calculus representation was *drastically* simplified. In the original presentation of prioritized default logic, values have the form $\langle e_1, \ldots, e_n | e_c : e_d | \leqslant \rangle$ where $\leqslant$ is a pre-order that compares the e_i *at run-time* to determine the order of priorities. We initially adopted this very general presentation for **Catala**, but found out that this made the semantics nearly impossible to explain; made the rules overly complicated and the proof of soundness very challenging; and more importantly, was not necessary to capture the essence of Western statutory law. Dropping a run-time pre-order $\leqslant$ was the key insight that made the internal default calculus representation tractable, both in the paper formalization and in the formal proof.

The scope language was introduced to eliminate the curious scoping rules and parent-overrides of definitions, which are unusual in a lambda-calculus. We initially envisioned a general override mechanism that would allow a parent scope to override a sub-scope definition at any depth; that is, not just re-define sub-scope variables, but also sub-sub-scope variables and so on. We were initially tempted to go for this most general solution; however, we have yet to find a single example of statutory law that needs this feature; and allowing sub-sub-scope override would have greatly complicated the translation step. We eventually decided to not implement this feature, to keep the compiler, the

Type	τ	::=	`bool` | `unit`	boolean and unit types
		|	$\tau \to \tau$	function type
		|	`list` τ	list type
		|	`option` τ	option type
Expression	e	::=	x | s | `true` | `false` | `()`	variable, scopes, literals
		|	λ `(`x `:` τ`)` `.` e | $e\ e$	λ-calculus
		|	`None` | `Some` e	option constructors
		|	`match` e `with`	option destructuring
			| `None` $\to e$ | `Some` $x \to e$	
		|	`[`$\vec{e}$`]` | `fold_left` $e\ e\ e$	list introduction and fold
		|	`raise` ε | `try` e `with` $\varepsilon \to e$	exceptions
Exception	ε	::=	$\varnothing$	empty exception
		|	$\circledast$	conflict exception
Declaration	σ	::=	`let` s = e	
Program	P	::=	$\vec{\sigma}$	

Figure 6.8: The enriched lambda calculus, our final translation target

paper rules and the formalization simple enough.

All of those insights stemmed from a close collaboration with lawyers, and the authors of this paper are very much indebted to Sarah Lawsky and Liane Huttner for their invaluable legal insights. Barring any legal expertise, we would have lacked the experimental evaluation that was able to justify our simplification choices and revisions of the language.

6.2.2. Bringing Non-monotonicity to Functional Programming

While sufficient to power the **Catala** surface language, the relatively simple semantics of our default calculus are non-standard. We now wish to compile to more standard constructs found in existing programming languages. We remark that the reduction semantics for default terms resembles that of exceptions: empty-default errors propagate ("are thrown") only up to the enclosing default term ("the try-catch"). Confirming this intuition and providing a safe path from **Catala** to existing programming languages, we now present a compilation scheme from the default calculus to a lambda calculus enriched with a few standard additions: lists, options and exceptions.

Figure 6.8 shows the syntax of the target lambda calculus. In order to focus on the gist of the translation, we introduce `list` and `option` as special, built-in datatypes, rather than a full facility for user-defined inductive types. For

C-DEFAULT

$$\frac{e_1 \Rightarrow e_1' \quad \cdots \quad e_n \Rightarrow e_n' \quad e_{\text{just}} \Rightarrow e_{\text{just}}' \quad e_{\text{cons}} \Rightarrow e_{\text{cons}}'}{\langle e_1, \ldots, e_n \mid e_{\text{just}} :- e_{\text{cons}} \rangle \Rightarrow \texttt{let } r_{\text{exceptions}} = \texttt{process\_exceptions } [\lambda \_ \to e_1'; \ldots; \lambda \_ \to e_n'] \texttt{ in } \texttt{match } r_{\text{exceptions}} \texttt{ with Some } e' \to e' \mid \texttt{None} \to \texttt{if } e_{\text{just}}' \texttt{ then } e_{\text{cons}}' \texttt{ else raise } \varnothing}$$

C-EMPTYERROR

$$\varnothing \Rightarrow \texttt{raise } \varnothing$$

C-CONFLICTERROR

$$\circledast \Rightarrow \texttt{raise } \circledast$$

C-VAR

$$x \Rightarrow x$$

C-LITERAL

$$\frac{e \in \{(), \texttt{true}, \texttt{false}\}}{e \Rightarrow e}$$

C-ABS

$$\frac{e \Rightarrow e'}{\lambda\ (x : \tau)\ .\ e \Rightarrow \lambda\ (x : \tau)\ .\ e'}$$

C-APP

$$\frac{e_1 \Rightarrow e_1' \qquad e_2 \Rightarrow e_2'}{e_1\ e_2 \Rightarrow e_1'\ e_2'}$$

Figure 6.9: Translation rules from default calculus to lambda calculus

those reasons, we offer the minimum set of operations we need: constructors and destructors for `option`, and a left fold for `list`s. We omit typing and reduction rules, which are standard. The only source term that does not belong to the target lambda calculus is the default term $\langle \vec{e} \mid e_{\text{just}} :- e_{\text{cons}} \rangle$. Hence, translating this term is the crux of our translation.

Our translation is of the form $\boxed{e \Rightarrow e'}$, where e is a term of the default calculus and e' is a term of the target lambda calculus. Figure 6.9 presents our translation scheme. The semantics of default terms are intertwined with those of $\varnothing$ and $\circledast$. The translation of $\varnothing$ and $\circledast$ is simple: both compile to exceptions in the target language. We now focus on C-DEFAULT, which deals with default terms. As a warm-up, we start with a special case: $\langle \mid e_{\text{just}} :- e_{\text{cons}} \rangle$. We translate this term to $\texttt{if } e_{\text{just}} \texttt{ then } e_{\text{cons}} \texttt{ else raise } \varnothing$, which obeys the evaluation semantics of both D-DEFAULTTRUENOEXCEPTIONS and D-DEFAULTFALSENOEXCEPTIONS. This simple example serves as a blueprint for the more general case, which has to take into account the list of exceptions $\vec{e}$, and specifically count how many of them are $\varnothing$.

In the general case, C-DEFAULT relies on a helper, `process_exceptions`; each exception is translated, thunked, then passed to the helper; if the helper returns `Some` , exactly one exception did not evaluate to $\varnothing$; we return it. If the helper returns `None`, no exception applied, and we fall back to the simple case we previously described.

```
process_exceptions : list (unit → τ) → option τ
process_exceptions ≜ fold_left (λ (a : option τ) (e′ : unit → τ).
                        let e′ : τ = try Some (e′()) with ∅ → None in
                        match (a, e′) with
                            | (None, e′) → e′
                            | (Some a, None) → Some a
                            | (Some a, Some e′) → raise ⊛ ) None
```

Figure 6.10: `process_exceptions` translation helper

We now review **`process_exceptions`** defined in Figure 6.10. It folds over the list of exceptions, with the accumulator initially set to `None`, meaning no applicable exception was found. Each exception is forced in order, thus implementing the reduction semantics of the default calculus. The accumulator transitions from `None` to `Some` if a non-empty exception is found, thus implementing a simple automaton that counts the number of non-empty exceptions. If two non-$\varnothing$ exceptions are found, the automaton detects an invalid transition and aborts with a non-catchable $\circledast$.

Certifying the Translation The translation from scope language to default calculus focuses on turning scopes into the lambda-abstractions that they truly are underneath the concrete syntax. This is a mundane transformation, concerned mostly with syntax. The final step from default calculus to lambda calculus with exceptions is much more delicate, as it involves compiling custom evaluation semantics. To rule out any errors in the most sensitive compilation step of **Catala**, we formally prove our translation correct, using F* [33]–[35], a proof assistant based on dependent types, featuring support for semi-automated reasoning via the SMT-solver Z3 [36].

Correctness statement We wish to state two theorems about our translation scheme. First, typing is preserved: if `de` $\rightrightarrows$ `le` and $\varnothing \vdash$ `de` : `dtau`, then $\varnothing \vdash$ `le` : `ltau` in the target lambda calculus where `ltau` is the (identity) translation of `dtau`. Second, we want to establish a simulation result, i.e. the compiled program `le` faithfully simulates a reduction step from the source language, using n

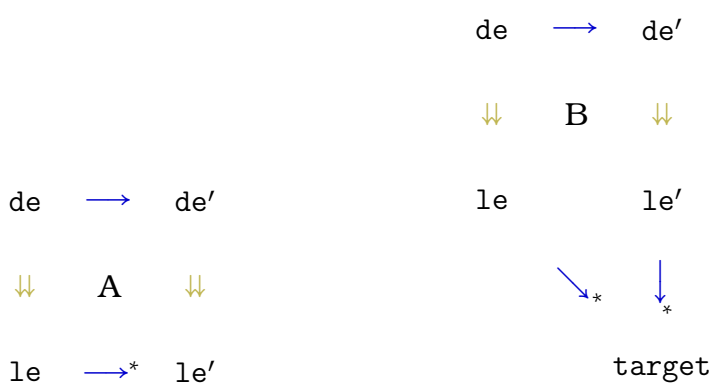

Figure 6.11: Translation correctness theorems. A shows a regular simulation; B shows our variant of the theorem.

steps in the target language.

The usual simulation result is shown in Figure 6.11, A. If `de` is a term of the default calculus and if `de` $\longrightarrow$ `de′`, and `de` $\rightrightarrows$ `le`, then there exists a term `le′` of the lambda calculus such that `le` $\longrightarrow$ `*le′` and `de′` $\rightrightarrows$ `le′`. This specific theorem does not apply in our case, because of the thunking we introduce in our translation. As a counter-example, consider the reduction of e_1 within default term $\langle v_0 , e_1 \mid e_{\text{just}} :- e_{\text{cons}} \rangle$. If e_1 steps to e'_1 in the default calculus, then the whole term steps to $\langle v_0 , e'_1 \mid e_{\text{just}} :- e_{\text{cons}} \rangle$. However, we translate exceptions to thunks; and our target lambda calculus does not support strong reduction, meaning $\lambda \_ \to e_{\lambda,1}$ does not step into $\lambda \_ \to e'_{\lambda,1}$. Diagram A is therefore not applicable in our case.

The theorem that actually holds in our case is shown as diagram B (Figure 6.11). The two translated terms `le` and `le′` eventually reduce to a common form `target`. Taking the transitive closure of form B, we obtain that if `de` reduces to a value `dv`, then its translation `le` reduces to a value `lv` that is the translation of `dv`, a familiar result.

Overview of the proof We have mechanically formalized the semantics of both the default calculus and target lambda calculus, as well as the translation scheme itself, inside the F* proof assistant. Figure 6.12 shows the exact theorem we prove, using concrete F* syntax; the theorem as stated establishes both type preservation and variant B, via the **`take_l_steps`** predicate and the existentially quantified **`n1`** and **`n2`**. We remark that if the starting term `de` is a value to start with, we have `le = translate_exp de`. Inspecting `translate_exp` (elided), we establish that source values translate to identical target values.

```
module D = DefaultCalculus; module L = LambdaCalculus
val translation_correctness (de: D.exp) (dtau: D.ty) : Lemma
  (requires (D.typing D.empty de dtau)) (ensures (
    let le = translate_exp de in
    let ltau = translate_ty dtau in
    L.typing L.empty le ltau ∧ begin
      if D.is_value de then L.is_value le else begin
        D.progress de dtau; D.preservation de dtau;
        let de' = Some?.v (D.step de) in
        translation_preserves_empty_typ de dtau;
        translation_preserves_empty_typ de' dtau;
        let le' : typed_l_exp ltau = translate_exp de' in
        exists (n1 n2:ℕ) (target: typed_l_exp ltau).
          (take_l_steps ltau le n1 == Some target ∧
           take_l_steps ltau le' n2 == Some target) end end))
```

Figure 6.12: Translation certification theorem, in F⋆

Proof effort and engineering Including the proof of type safety for the source and target language, our F⋆ mechanization amounts to approximately 3,500 lines of code and required 1 person-month. We rely on partial automation via Z3, and the total verification time for the entire development is of the order of a few minutes. The choice of F⋆ was not motivated by any of its advanced features, such as its effect system: the mechanization fits inside the pure fragment of F⋆. Our main motivation was the usage of the SMT solver which can typically perform a fair amount of symbolic reasoning and definition unrolling, thus decreasing the amount of manual effort involved.

To focus the proof effort on the constructs that truly matter (i.e. default expressions), the semantics of lists, folds and options are baked into the target calculus. That is, our target calculus does not support user-defined algebraic data types. We believe this is not a limitation, and instead allows the proof to focus on the key parts of the translation. We use De Bruijn indices for our binder representation, since the unrolling of `process_exceptions` results in variable shadowing. Given those simplifications, we were surprised to find that our proof still required 3,500 lines of F⋆. A lot of the complexity budget was spent on the deep embedding of the `process_exceptions` helper. It is during the mechanization effort that we found out that theorem A does not hold, and that we need to establish B instead. Our mechanized proof thus

significantly increases our confidence in the **Catala** compilation toolchain; the proof is evidence that even for a small calculus and a simple translation, a lot of subtleties still remain.

While F* extracts to OCaml, we chose *not* to use the extracted F* code within the **Catala** compiler. First, the proof does not take into account all language features. Second, the actual translation occupies about 100 lines of code in both the production **Catala** compiler and the proof; we are content with comparing both side-by-side. Third, the **Catala** compiler features advanced engineering for locations, error messages, and propagating those to the proof would be difficult.

6.3. Catala As an Interdisciplinary Vehicle

We claim that the design process of **Catala** as well as our comprehensive code co-production process proposal maximizes the potential for adoption by professionals. To support this claim, we report early user study results and demonstrate an end-to-end use case with the computation of an important French social benefit.

6.3.1. Bringing Code Review to Lawyers

Catala's design has been supervized and influenced by lawyers since its inception. Indeed, the project started out of Sarah Lawsky's insight on the logical structure of legal statutes [11], [37]–[39]. As well as providing the formal base building block of **Catala**, lawyers also reviewed the syntax of **Catala**, choosing the keywords and providing insights counter-intuitive to programmers, such as the `rule/definition` distinction of Section 6.1.1.

We also conducted a careful analysis of the production process of legal expert systems. We found that in France, administrative agencies always use a V-shaped development cycle for their legal expert systems. In practice, lawyers of the legal department take the input set of legal statutes and write a detailed natural language specification, that is supposed to make explicit the different legal interpretations required to turn the legal statute into an algorithm. Then, legal expert systems programmers from the IT department take the natural specification and turn it into code, often never referring back to the original statute text.

Exclusive interviews conducted by the authors with legal expert systems programmers and lawyers inside a high-profile French administration reveal that this theoretical division of labor is artificial. Indeed, the natural language specification often proves insufficient or ambiguous to programmers, which leads to programmers having to spend hours on the phone with the lawyers to clarify and get the algorithm right. Furthermore, the validation of the implementation depends on lawyer-written test cases, whose number and quality suffer from budget restrictions.

This insight suggests that a more agile development process associating lawyers and programmers from the beginning would be more efficient. We claim the **Catala** is the right tool for the job, since it allows lawyers and programmers to perform pair programming on a shared medium that locally combines the legal text as well as the executable code.

We do not expect lawyers to write **Catala** code by themselves. A number of frameworks such as Drools [40] are built on this promise. For our part, we believe that software engineering expertise is needed to produce maintainable, performant, high-quality code. Hence, we envision for lawyers to act as observers and reviewers of the code production process, safeguarding the correctness with respect to the legal specification.

We don't expect adoption difficulties from the programmers' side, since **Catala** is morally a pure functional language with a few oddities that makes it well-suited to legal specifications. To assess our claim of readability by lawyers, we conducted a small user study with $N = 18$ law graduate students enrolled in the Masters program "Droit de la création et numérique" (Intellectual Property and Digital Law) at Université Panthéon-Sorbonne. The students are anonymous recruits, enrolled in a course taught by a lawyer friend of the project. The study was conducted during a 2-hour-long video conference, during which the students were able to submit feedback in real time thanks to an online form. None of the students had any prior programming experience; this question was asked orally at the beginning of the session.

The methodology of the study is the following: the participants were given a 30 min. presentation of **Catala**'s context and goals, but were not exposed to any program or syntax. Then, participants were briefed during 15 min. about Section 121 and its first paragraph (Section 6.1.1) and received a copy of the corresponding **Catala** code. Finally, the participants were told the protocol of the study: 10 min. for reading the **Catala** code of Section 121, then fill the questionnaire listed in Table 6.2; 15 min. of collective Q&A with the **Catala** authors (over group video conference) about the **Catala** code of Section 121,

Table 6.2.: Questions of the user study

#	Exact text of the question
(1)	Do you understand the project? Is there anything that is unclear about it?
(2)	Can you read the code without getting a headache?
(3)	Can you understand the code?
(4)	Can you link the code to the meaning of the law it codifies?
(5)	Can you certify that the code does exactly what the law says and nothing more? If not, are there any mistakes in the code?

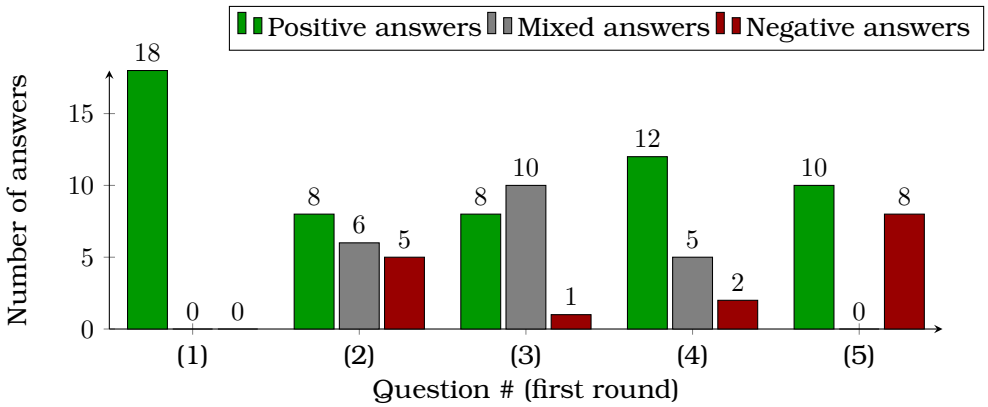

Figure 6.13: Results of the first round of questions in the user study

then fill a second time the same questionnaire listed in Table 6.2.

The participants and experimenters carried out the study according to the aforementioned protocol. After all students had filled the questionnaire for the second time, a short debriefing was conducted. The full answers of the participants to all the questions of both rounds are available in the artifact corresponding to this paper [12]. The answers given by the participants in free text were interpreted as positive, negative or mixed by the authors. Figure 6.13 and Figure 6.14 show the results for the first and second fillings of the questionnaire by the participants.

These early results, while lacking the rigor of a scientific user study, indicate a relatively good reception of the literate programming paradigm by lawyers. The significant increase in positive answers between the first and second round of questions indicates that while puzzled at first, lawyers can quickly grasp the intent and gist of **Catala** once given a minimal amount of Q&A time

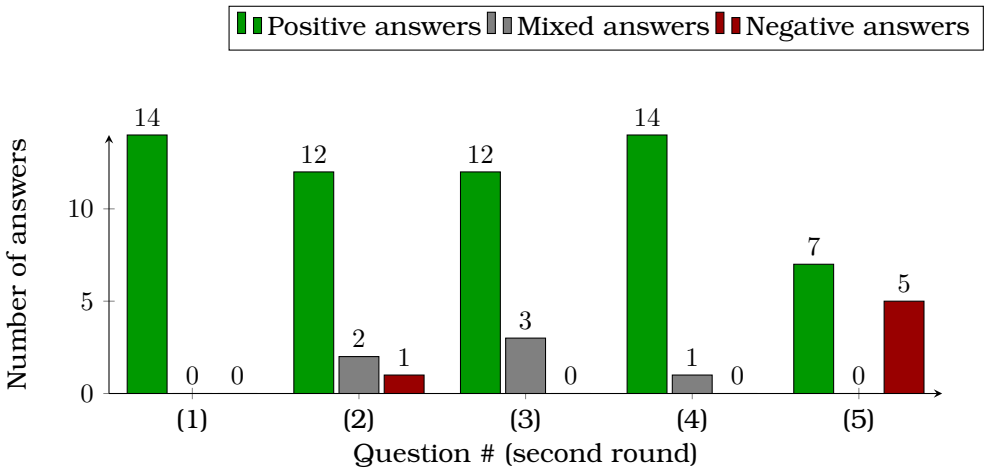

Figure 6.14: Results of the second round of questions in the user study

(15 minutes in our study). One participant to the study was enthusiastic about the project, and contacted the authors later to join the project and tackle the modeling of French inheritance law in **Catala**.

After investigation, we believe that the large number of negative answers for question (5) in the second round could be explained by a lack of familiarity with the US Internal Revenue Code from the French lawyers. Indeed, the wording of the question ("certify") implies that the lawyer would be confident enough to defend their opinion in court. We believe, from deeper interactions with lawyers closer to the project, that familiarity with the formalized law combined with basic **Catala** training could bring the lawyers' confidence to this level.

We deliberately introduced a bug in the code shown to the lawyers in this user study. The bug involved a $\leqslant$ operator replaced by $\geqslant$. Of the 7 who answered "Yes" to (5) in the second round, 2 were able to spot it, which we interpret to be a very encouraging indication that lawyers can make sense of the **Catala** code with just a two-hour crash course.

6.3.2. Improving Compliance in Legal Expert Systems

Based on the formalization of Section 6.2, we implement **Catala** in a standalone compiler and interpreter. The architecture of the compiler is based on a series of intermediate representations, in the tradition of CompCert [41] or Nanopass [42]. Figure 6.15 provides a high-level overview of the architecture,

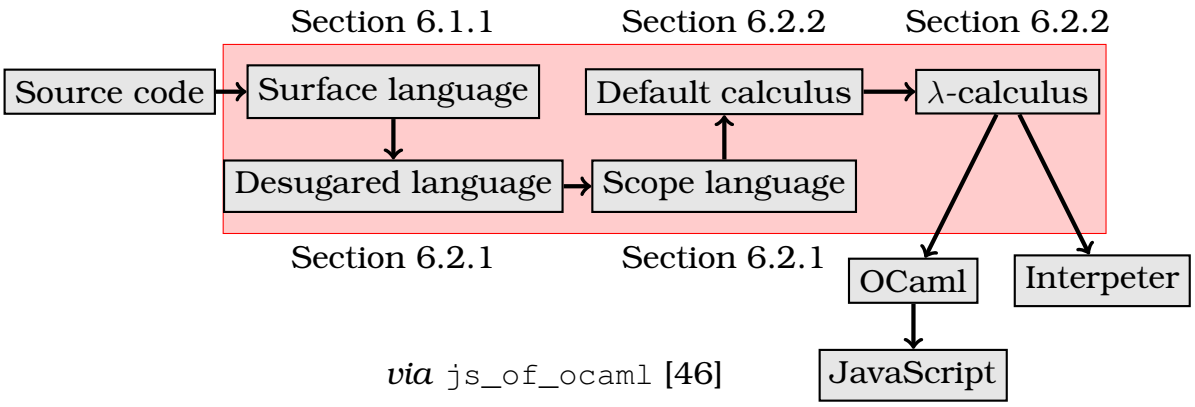

Figure 6.15: High-level architecture of the **Catala** compiler (red box)

```
[ERROR] Syntax error at token "years"
[ERROR] Message: expected a unit for this literal, or a valid operator
[ERROR]          to complete the expression
[ERROR] Autosuggestion: did you mean "year", or maybe "or", or maybe "and",
[ERROR]                 or maybe "day", or maybe ".", or maybe ">", [...]
[ERROR] Error token:
[ERROR]    --> section_121.catala_en
[ERROR]     |
[ERROR] 180 |       if date_of_sale_or_exchange <=@ period.begin +@ 5 years then
[ERROR]     |                                                       ^^^^^
[ERROR] Last good token:
[ERROR]    --> section_121.catala_en
[ERROR]     |
[ERROR] 180 |       if date_of_sale_or_exchange <=@ period.begin +@ 5 years then
[ERROR]     |                                                     ^
```

Figure 6.16: Example of **Catala** syntax error message

with links to relevant sections alongside each intermediate representation.

The compiler is written in OCaml and features approximately 13,000 lines of code. This implementation, available as open-source software on GitHub and in the artifact accompanying this paper [12], makes good use of the rich and state-of-the art library ecosystem for compiler writing, including **`ocamlgraph`** [43] for the e.g. the two topological sorts we saw (Section 6.2.1), **`bindlib`** [44] for efficient and safe manipulation of variables and terms, and the **`menhir`** parser generator [45]. Thanks to these libraries, we estimate that the development effort was 5 person-months.

Usability We devoted a great of attention towards the usability of the compiler. Indeed, while we don't expect lawyers to use **Catala** unaccompanied, we

would not want to restrict its usage to λ-savvy functional programmers. To improve the programmer experience, we use the special parser error reporting scheme of **menhir** [47], to provide customized and context-aware syntax error messages that depend on the set of tokens acceptable by the grammar at the site of the erroneous token (see Figure 6.16). The shape of the error messages is heavily inspired by the Rust compiler design [48]. Error messages flow through the compiler code via a unique exception containing the structured data of the error message:

```
exception StructuredError of
  (string * (string option * Pos.t) list)
```

This structure enables on-the-fly rewriting of error messages as they propagate up the call stack, which is useful for e.g. adding a new piece of context linking to a code position of a surrounding AST node. In this spirit, error messages for scope variable cycle detection display the precise location for where the variables in the cycle are used; error messages for default logic conflict errors $\circledast$ show the location of the multiple definitions that apply at the same time for a unique variable definition.

Finally, we have instrumented the **Catala** interpreter with helpful debugging features. Indeed, when pair programming with a lawyer and a programmer over the formalization of a piece of law, it is helpful to see what the execution of the program would look like on carefully crafted test cases. While test cases can be directly written in **Catala** using a top-level scope that simply defines the arguments of the sub-scope to test, the compilation chain inserts special log hooks at critical code points. When executing a test case, the interpreter then displays a meaningful log featuring code positions coupled with the position inside the legal statute for each default logic definition taken.

We believe that this latter feature can easily be extended to provide a comprehensive and legal-oriented explanation of the result of a **Catala** program over a particular input. Such an explanation would help increase trust of the system by its users, e.g. citizens subject to a complex taxation regime; thereby constituting a concrete instance of a much sought-after "explainable AI" [49], [50].

Performance We ran the French family benefits **Catala** program described in a later paragraph; it is as complex as Section 121 of the US Internal Revenue Code but featuring approximately 1500 lines of **Catala** code (literate

programming included). Given the description of a French household, we benchmark the time required to compute the amount of monthly family benefits owed.

The **Catala** interpreter for this program runs in approximately 150ms. We conclude that the performance of the interpreter remains acceptable even for a production environment. When the **Catala** code is compiled to OCaml, execution time drops to 0.5ms. Therefore, we conclude that performance problems are, at this stage of the project, nonexistent.

Extensible Compiler Backend A crucial consideration in designing a DSL is the interoperability story within existing environments. While some DSLs operate in isolation, we envision **Catala** programs being exposed as reusable libraries that can be called from any development platform, following the needs of our adopters. In the context of legal systems, this is a very strong requirement: such environments oftentimes include legacy, mainframe-based systems operating in large private organizations or government agencies [51]. Furthermore, since the algorithms that **Catala** is designed to model are at the very heart of e.g. tax collection systems, proposing a classic interoperability scheme based on APIs or inter-language FFIs might create an undesirable barrier to adoption; a system designed in the 1960s probably has no notion of API or FFI whatsoever!

Instead, we choose for **Catala** an unusually simple and straightforward interoperability scheme: direct source code generation to virtually any programming language. This solution is generally impractical, requiring considerable workarounds to reconcile semantic mismatches between target and source, along with a significant run-time support library. In the case of **Catala**, however, our intentional simplicity makes this "transpiling" scheme possible.

Indeed, the final intermediate representation of the **Catala** compiler is a pure and generic lambda calculus operating over simply-typed values. By re-using standard functional compilation techniques such as closure conversion [52], we claim that it is possible to compile **Catala** to any programming language that has functions, arrays, structures, unions, and support for exceptions. We also believe that a more complex version of the compilation scheme presented in Section 6.2.2 would remove the need for exceptions (in favor of option types), but leave this as future work.

The run-time required for generated programs only has to include an infinite precision arithmetic library (or can default to fixed-sized arithmetic and floats)

and a calendar library to compute the days difference between two dates, taking leap years into account. We demonstrate this with the OCaml backend of the **Catala** compiler, which amounts to 350 lines of compiler code and 150 lines of run-time code (excluding the `zarith` [53] and `calendar` [54] libraries). Merely compiling to OCaml already unlocks multiple target environments, such as the Web, via the `js_of_ocaml` compiler [46]. We thus effortlessly bring **Catala** to the Web.

A Look Back to Section 121 We have used Section 121 of the US Internal Revenue Code as a support for introducing **Catala** in Section 6.1.1. But more interestingly, this piece of law is also our complexity benchmark for legal statutes, as it was deemed (by a lawyer collaborator) to be one of the most difficult sections of the tax code. This reputation comes from its dense wording featuring various layers of exceptions to every parameter of the gross income deduction.

We have so far formalized it up to paragraph (b)(4), which is approximately 15% of the whole section and around 350 lines of code (including the text of the law), but contain its core and most used exceptions. We include the result of this effort in the artifact [12]. The current formalization was done in four 2-hour sessions of pair programming between the authors and lawyers specialized in the US Internal Revenue Code. Progress is relatively slow because we consider in the process every possible situation or input that could happen, as in a real formalization process. However, this early estimate indicates that formalizing the whole US Internal Revenue Code is a completely reachable target for a small interdisciplinary team given a few years' time.

While finishing the formalization of Section 121 is left as future work, we are confident that the rest of the section can be successfully expressed in **Catala**: the maze of exceptions is localized to (a) and (b), and the rest of the limitations are just a long tail of special cases; with our general design that supports arbitrary trees of exceptions in default logic, this should pose no problem.

Case Study: French Family Benefits Section 6.3 argues that **Catala** is received positively by lawyers. This is only half of the journey: we need to make sure **Catala** is also successfully adopted by the large private or public organization where legacy systems are ripe for a principled rewrite. To support our claims concerning the toolchain and interoperability scheme in a real-world setting,

we formalized the entire French family benefits computation in **Catala** and exposed the compiled program as an OCaml library and JavaScript Web simulator. The full code of this example can be found in the supplementary material of this article, although it is written in French.

A crucial part of the French welfare state, family benefits are distributed to households on the basis of the number of their dependent children. Created in the early 1930's, this benefit was designed to boost French natality by offsetting the additional costs incurred by child custody to families. Family benefits are a good example of a turbulent historical construction, as the conditions to access the benefits have been regularly amended over the quasi-century of existence of the program. For instance, while family benefits were distributed to any family without an income cap, a 2015 reform lowered the amount for wealthy households [55], [56].

The computation can be summarized with the following steps. First, determine how many dependent children are relevant for the family benefits (depending on their age and personal income). Second, compute the base amount, which depends on the household income, the location (there are special rules for overseas territories) and a coefficient updated each year by the government to track inflation. Third, modulate this amount in the case of alternating custody or social services custody. Fourth, apply special rules for when a child is exactly at the age limit for eligibility, or when the household income is right above a threshold. All of these rules are specified by 27 articles of the French Social Security Code, as well as various executive orders.

The **Catala** formalization of this computation amounts to approximately 1,500 lines of code, including the text of the law. The code is split between 6 different **`scope`**s featuring 63 **`context`** variables and 83 **`definition`**s and **`rule`**s. We believe these numbers to fairly representative of the authoring effort required for a medium-size legal document. Distributed as an OCaml library, our code for the French family benefits is also used to power an online simulator (see Figure 6.17).

After writing the code as well as some test cases, we compared the results of our program with the official state-sponsored simulator `mes-droits-sociaux.gouv.fr`, and found no issue. However, the case where a child is in the custody of social services was absent from the official simulator, meaning we could not compare results for this case. Fortunately, the source code of the simulator is available as part of the OpenFisca software project [57]. The OpenFisca source file corresponding to the family benefits, amounts to 365 lines of Python. After close inspection of the OpenFisca code, a discrepancy

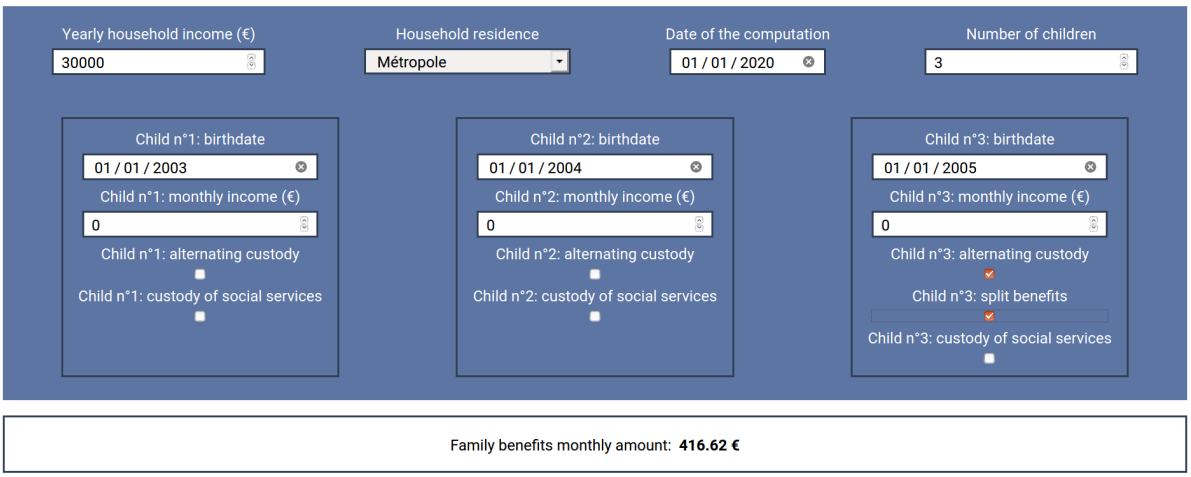

Figure 6.17: Screenshot of the Web family benefits simulator powered by **Catala**

was located with the **Catala** implementation. Indeed, according to article L755-12 of the Social Security Code, the income cap for the family benefits does not apply in overseas territories with single-child families. This subtlety was not taken into account by OpenFisca, and was fixed after its disclosure by the authors.

Formalizing a piece of law is thus no different from formalizing a piece of software with a proof assistant. In both cases, bugs are found in existing software, and with the aid of a formal mechanization, can be reported and fixed.

Conclusion

The last chapter of this dissertation is also the most forward-thinking and holistic illustration of the proof-oriented domain-specific language design methodology that we advocate for. The collaboration with domain experts on the topic was paramount; it should be noted that we were extremely lucky to collaborate directly with Sarah Lawsky, who is probably the only tax law professor who also holds a PhD degree in formal logic. In fact, Sarah Lawsky and the prior logic programming researchers working on formal modeling of law already had done the hard work of uncovering the right paradigms for an efficient formalization. However, this community of logicians was miss-

ing a proper programming languages culture, as the logical languages they produced never incorporated elements that could have made them actually usable or deployable.

The added value of this work and **Catala** is thus to adapt the existing logical research about legal formalization into an artifact that is both usable and deployable in the critical software context of the organizations that run legal expert systems. Hence, to get the end result, three layers of intermediation and distinct expertise were needed. First, the legal layer with its knowledge of statutes and their interpretation. Second, the logic layer with its formalism and fine-grained modeling of legal reasoning into mathematical theories. Then, the programming language layer with its constraints about usability, software implementation and maintainability. All three layers are equally important in this chain that goes from *in vivo* domain practices to digital tools that faithfully automate some of them without perverting their intent.

The risk of perverting the intent of legal practices through automation is real, and stems from the simplifications that must occur at each level of the chain. Indeed, the goal of law is to regulate the reality of the world, where everything can happen and whose complexity will forever escape the formalist's high-modernist rule. It is then important to always check that the end product, as simple as an extended lambda calculus can be, allows for programs written in it the flexibility of the original legal text. This problem is thoroughly explored in a recent article of the MIT Computational Law Report [58], and we believe that **Catala**'s design partially incorporate this contraint; for instance, two different interpretations of a same article can be both encoded, and discriminated against at run-time with a user input or condition. However, we believe that the source **Catala** code should enjoy a high-level of consensus before execution, to avoid *ex post* court cases where an affected party would contest the translation of law into **Catala** code. That is why we strongly advocate for the code of legal expert systems to be open source and accept external contributions.

References

[1] D. Merigoux, N. Chataing, and J. Protzenko, "Catala: a programming language for the law", *Proc. ACM Program. Lang.*, vol. 5, no. ICFP, Aug. 2021. DOI: 10.1145/3473582. [Online]. Available: https://doi.org/10.1145/3473582.

[2] Wikipedia contributors, *Code of ur-nammu — Wikipedia, the free encyclopedia*, [Online; accessed 22-Feb-2021], 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Code_of_Ur-Nammu&oldid=998720276.

[3] J. Monin, *Louvois, le logiciel qui a mis l'armée à terre*, 2018. [Online]. Available: https://www.franceinter.fr/emissions/secrets-d-info/secrets-d-info-27-janvier-2018.

[4] Government Accountability Office (GAO), *COVID-19: urgent actions needed to better ensure an effective federal response – report to congressional committees*, Appendix 24, first table, 2021. [Online]. Available: https://www.gao.gov/reports/GAO-21-191/#appendix24.

[5] D. Merigoux, R. Monat, and J. Protzenko, "A modern compiler for the french tax code", in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2021, Virtual, Republic of Korea: Association for Computing Machinery, 2021, pp. 71–82, ISBN: 9781450383257. DOI: 10.1145/3446804.3446850. [Online]. Available: https://doi.org/10.1145/3446804.3446850.

[6] Anne Broache, *IRS trudges on with aging computers*, 2008. [Online]. Available: https://www.cnet.com/news/irs-trudges-on-with-aging-computers/.

[7] Frank R. Konkel, *The IRS system processing your taxes is almost 60 years old*, 2018. [Online]. Available: https://www.nextgov.com/it-modernization/2018/03/irs-system-processing-your-taxes-almost-60-years-old/146770/.

[8] G. Brewka and T. Eiter, "Prioritizing default logic", in *Intellectics and computational logic*, Springer, 2000, pp. 27–45.

[9] Internal Revenue Service, *Exclusion of gain from sale of principal residence*. [Online]. Available: https://www.law.cornell.edu/uscode/text/26/121.

[10] R. Reiter, "A logic for default reasoning", *Artificial Intelligence*, vol. 13, no. 1, pp. 81–132, 1980, Special Issue on Non-Monotonic Logic.

[11] S. B. Lawsky, "A Logic for Statutes", *Florida Tax Review*, 2018.

[12] M. Denis, C. Nicolas, and P. Jonathan, *Catala: a programming language for the law*, version 1.0.0, May 2021. DOI: 10.5281/zenodo.4775161. [Online]. Available: https://doi.org/10.5281/zenodo.4775161.

[13] D. E. Knuth, “Literate Programming”, *The Computer Journal*, vol. 27, no. 2, pp. 97–111, Jan. 1984.

[14] J. Dewey, “Logical method and law”, *Cornell LQ*, vol. 10, p. 17, 1924.

[15] L. E. Allen, “Symbolic logic: a razor-edged tool for drafting and interpreting legal documents”, *Yale LJ*, vol. 66, p. 833, 1956.

[16] M. J. Sergot, F. Sadri, R. A. Kowalski, F. Kriwaczek, P. Hammond, and H. T. Cory, “The british nationality act as a logic program”, *Commun. ACM*, vol. 29, no. 5, pp. 370–386, May 1986.

[17] A. Colmerauer and P. Roussel, “The birth of prolog”, in *History of Programming Languages—II*. New York, NY, USA: Association for Computing Machinery, 1996, pp. 331–367.

[18] L. T. McCarty, “A language for legal discourse i. basic features”, in *Proceedings of the 2nd International Conference on Artificial Intelligence and Law*, ser. ICAIL ’89, Vancouver, British Columbia, Canada: Association for Computing Machinery, 1989, pp. 180–189, ISBN: 0897913221. DOI: 10.1145/74014.74037. [Online]. Available: https://doi.org/10.1145/74014.74037.

[19] G. Sartor, “A formal model of legal argumentation”, *Ratio Juris*, vol. 7, no. 2, pp. 177–211, 1994.

[20] G. H. Von Wright, “Deontic logic”, *Mind*, vol. 60, no. 237, pp. 1–15, 1951.

[21] L. T. McCarty, “Defeasible deontic reasoning”, *Fundamenta Informaticae*, vol. 21, no. 1, 2, pp. 125–148, 1994.

[22] ——, “An implementation of eisner v. macomber”, in *Proceedings of the 5th International Conference on Artificial Intelligence and Law*, ser. ICAIL ’95, College Park, Maryland, USA: Association for Computing Machinery, 1995, pp. 276–286, ISBN: 0897917588. DOI: 10.1145/222092.222258. [Online]. Available: https://doi.org/10.1145/222092.222258.

[23] ——, “Some arguments about legal arguments”, in *Proceedings of the 6th international conference on artificial intelligence and law*, 1997, pp. 215–224.

[24] P. Leith, “The rise and fall of the legal expert system”, *International Review of Law, Computers & Technology*, vol. 30, no. 3, pp. 94–106, 2016.

[25] M. A. Pertierra, S. Lawsky, E. Hemberg, and U.-M. O'Reilly, "Towards formalizing statute law as default logic through automatic semantic parsing.", in *ASAIL@ ICAIL*, 2017.

[26] N. Holzenberger, A. Blair-Stanek, and B. Van Durme, "A dataset for statutory reasoning in tax law entailment and question answering", *arXiv preprint arXiv:2005.05257*, 2020.

[27] J. Morris, "Spreadsheets for legal reasoning: the continued promise of declarative logic programming in law", *Available at SSRN 3577239*, 2020.

[28] M. Genesereth, "Computational law", *The Cop in the Backseat*, 2015.

[29] T. Hvitved, "Contract formalisation and modular implementation of domain-specific languages", Ph.D. dissertation, Citeseer, 2011.

[30] V. Scoca, R. B. Uriarte, and R. De Nicola, "Smart contract negotiation in cloud computing", in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, IEEE, 2017, pp. 592–599.

[31] X. He, B. Qin, Y. Zhu, X. Chen, and Y. Liu, "Spesc: a specification language for smart contracts", in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, IEEE, vol. 1, 2018, pp. 132–137.

[32] J. Zakrzewski, "Towards verification of ethereum smart contracts: a formalization of core of solidity", in *Working Conference on Verified Software: Theories, Tools, and Experiments*, Springer, 2018, pp. 229–247.

[33] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, *et al.*, "Dependent types and multi-monadic effects in F*", in *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 256–270.

[34] G. Martinez, D. Ahman, V. Dumitrescu, N. Giannarakis, C. Hawblitzel, C. Hriţcu, M. Narasimhamurthy, Z. Paraskevopoulou, C. Pit-Claudel, J. Protzenko, *et al.*, "Meta-F*: proof automation with SMT, tactics, and metaprograms", in *European Symposium on Programming*, Springer, Cham, 2019, pp. 30–59.

[35] D. Ahman, C. Hritcu, K. Maillard, G. Martínez, G. Plotkin, J. Protzenko, A. Rastogi, and N. Swamy, "Dijkstra monads for free", in *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, ACM, Jan. 2017, pp. 515–529. DOI: 10.1145/3009837.3009878. [Online]. Available: https://www.fstar-lang.org/papers/dm4free/.

[36] L. De Moura and N. Bjørner, "Z3: an efficient SMT solver", in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337–340.

[37] S. Lawsky, *Formal Methods and the Law*, 2018. [Online]. Available: https://popl18.sigplan.org/details/POPL-2018-papers/3/Formal-Methods-and-the-Law.

[38] S. B. Lawsky, "Formalizing the Code", *Tax Law Review*, vol. 70, no. 377, 2017.

[39] ——, "Form as formalization", *Ohio State Technology Law Journal*, 2020.

[40] M. Proctor, "Drools: a rule engine for complex event processing", in *Proceedings of the 4th International Conference on Applications of Graph Transformations with Industrial Relevance*, ser. AGTIVE'11, Budapest, Hungary: Springer-Verlag, 2012, pp. 2–2. DOI: 10.1007/978-3-642-34176-2_2. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-34176-2_2.

[41] X. Leroy, "Formal certification of a compiler back-end or: programming a compiler with a proof assistant", *SIGPLAN Not.*, vol. 41, no. 1, pp. 42–54, Jan. 2006.

[42] A. W. Keep and R. K. Dybvig, "A nanopass framework for commercial compiler development", in *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, 2013, pp. 343–350.

[43] S. Conchon, J.-C. Filliâtre, and J. Signoles, "Designing a generic graph library using ml functors.", *Trends in functional programming*, vol. 8, pp. 124–140, 2007.

[44] R. Lepigre and C. Raffalli, "Abstract representation of binders in ocaml using the bindlib library", *arXiv preprint arXiv:1807.01872*, 2018.

[45] F. Pottier and Y. Régis-Gianat, *The menhir parser generator*, 2014. [Online]. Available: http://cambium.inria.fr/~fpottier/menhir/.

[46] J. Vouillon and V. Balat, "From bytecode to javascript: the js_of_ocaml compiler", *Software: Practice and Experience*, vol. 44, no. 8, pp. 951–972, 2014.

[47] F. Pottier, "Reachability and error diagnosis in LR (1) parsers", in *Proceedings of the 25th International Conference on Compiler Construction*, 2016, pp. 88–98.

[48] J. Turner, *Shape of errors to come*, https://blog.rust-lang.org/2016/08/10/Shape-of-errors-to-come.html, 2016.

[49] R. Goebel, A. Chander, K. Holzinger, F. Lecue, Z. Akata, S. Stumpf, P. Kieseberg, and A. Holzinger, "Explainable ai: the new 42?", in *International cross-domain conference for machine learning and knowledge extraction*, Springer, 2018, pp. 295–303.

[50] D. Doran, S. Schulz, and T. R. Besold, "What does explainable ai really mean? a new conceptualization of perspectives", *arXiv preprint arXiv:1710.00794*, 2017.

[51] K. Leswing, *New Jersey needs volunteers who know COBOL, a 60-year-old programming language*, 2020. [Online]. Available: https://www.cnbc.com/2020/04/06/new-jersey-seeks-cobol-programmers-to-fix-unemployment-system.html.

[52] Y. Minamide, G. Morrisett, and R. Harper, "Typed closure conversion", in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on principles of programming languages*, 1996, pp. 271–283.

[53] A. Miné, X. Leroy, P. Cuoq, and C. Troestler, *The zarith ocaml library*, 2011. [Online]. Available: https://github.com/ocaml/Zarith.

[54] J. Signoles, *The calendar ocaml library*, 2011. [Online]. Available: https://github.com/ocaml-community/calendar.

[55] M.-F. Clergeau, "Plaidoyer pour la modulation", FR, *Travail, genre et sociétés*, vol. 35, no. 1, pp. 173–177, 2016. DOI: 10.3917/tgs.035.0173. [Online]. Available: https://doi.org/10.3917/tgs.035.0173.

[56] M.-A. Blanc, "La modulation des allocations familiales : une erreur historique", FR, *Travail, genre et sociétés*, vol. 35, no. 1, pp. 157–161, 2016. DOI: 10.3917/tgs.035.0157. [Online]. Available: https://doi.org/10.3917/tgs.035.0157.

[57] S. Shulz, “Un logiciel libre pour lutter contre l’opacité du système sociofiscal”, *Revue francaise de science politique*, vol. 69, no. 5, pp. 845–868, 2019.

[58] L. Diver, “Interpreting the rule(s) of code: performance, performativity, and production”, *MIT Computational Law Report*, Jul. 15, 2021, https://law.mit.edu/pub/interpretingtherulesofcode. [Online]. Available: `https://law.mit.edu/pub/interpretingtherulesofcode`.

Epilogue

At the end of this journey into proof-oriented domain-specific language design, it is useful to come back to the methodology of Section 1.3.1 and Figure 1.9 of page 36. We can summarize what steps of this methodology are illustrated by the chapters of this dissertation.

Chapter 2: LibSignal⋆ is a comprehensive case study for the methodology. By re-purposing an existing program verification framework, we could bring to the Web ecosystem the high assurance cryptographic standards existing for native implementations with WHACL⋆. Moreover, the LibSignal⋆ artefact has been crafted by carving out a critical part from a real-world implementation, lowering the barrier to a transfer to industrial users.

Chapter 3: hacspec focuses on the second and third steps of the methodology, by filling the gap between cryptographers and proof engineers around the specifications of cryptographic primitives and protocols. It also demonstrates the power of a custom compilation platform to target multiple proof frameworks and take advantage of their diversity and distinct strengths.

Chapter 4: Steel is a exercise in proof-oriented language design inside a general-purpose proof assistant, corresponding to step four of the methodology. This experiment shows that to tackle the difficult problem of stateful program verification, a divide-and-conquer strategy ought to be applied to proof obligations. Only then can adapted proof automation backends be efficiently used to push the program verification frontier forward.

Chapter 5: MLANG applies step two of the methodology to an existing, real-world domain-specific language for critical tax software. Although the work presented does not feature fully-fledged functional correctness verification, the first static analysis performed on the codebase show promising future work opportunities.

Chapter 6: Catala also illustrates step two of the methodology, in an attempt to re-design tax domain-specific languages with solid logic foundations adapted to legal reasoning. By enjoying a partially certified compilation from a fully formalized core language, this contribution is also related to steps four and five of the methodology. Future work should connect **Catala** to deep or shallow embedding inside proof assistant for more complex verification.

In the three years and a half that spanned the completion of these contributions, our convergence to language design as opposed to other specific skills in formal methods is visible. Language design as an activity is hard to evaluate quantitatively, as its success inherently tied to the user experience of all the beneficiaries of the project: source code writer, receptors of generated code, etc. Recently, Coblenz *et al.*[1] lay out a comprehensive framework for this evaluation with user-centered methods. We believe this kind of evaluation process is the best tool currently available to guide programming language design research, but it comes with limitations. Correctly interpreting user feedback and avoiding all the bias that comes with subjective perception is hard, and the right solutions required a big investment in time and social science research skills. Coblenz *et al.* note:

> "Late in the project, we found that designing and running user studies of low-level features typically required much more time than implementing the features".

This observation might come as a paradox to some members of the programming languages community, that highly value technical difficulty and mathematical elegance when reviewing the works of their peers, and tend to disregard this sort of "soft science" contributions. To interpret this paradox, we claim that the technical aspect of language design has enjoyed an attention hypertrophy compared to the social aspect of language design. The verification frameworks presented in Section 1.1.1 and Section 1.1.2 have made it easy to produce new languages that can be connected to a rich technical infrastructure ecosystem. Meanwhile, too few new languages are designed with user-centric methods in mind; we believe that this limitation actively

[1] M. Coblenz, G. Kambhatla, P. Koronkevich, J. L. Wise, C. Barnaby, J. Sunshine, J. Aldrich, and B. A. Myers, "Pliers: a process that integrates user-centered methods into programming language design", *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 28, no. 4, pp. 1–53, 2021

hurts the advancement of the formal methods field, by restricting it to a set of well-known technical niches in which progress now comes at higher cost and with diminishing returns.

Unfortunately, mastering the technical state of the art is still a hard requirement before innovating from it, and that is why three chapters of this dissertation are dedicated to more traditional areas of program verification. The last two chapters present practical examples of how to leverage the current state of the art technical tooling and frameworks to a new domain, thus opening many new research leads and opportunities. We acknowledge that replicating this kind of individual journey is unlikely, the one presented here being the specific product of a particular personal trajectory. Hence, we rather advocate for increased interdisciplinary collaboration, moving from individuals to teams with a shared objective driven by real-world applications. Building the right environment for this sort of work to happen in good conditions is an organizational challenge by itself; but our future work in this area is bound to include a holistic approach to innovation that includes and goes beyond the technical challenges presented in this dissertation.

Table of contents

RÉSUMÉ

La vérification de programme consiste en l'analyse d'un programme informatique vu comme un artefact formel, afin de prouver l'absence de certaines catégories de bogues avant l'éxécution. Mais pour utiliser un cadriciel de vérification de programme, il faut auparavant traduire le code source originel du programme dans le langage formel du cadriciel. De plus, il est possible d'utiliser plusieurs cadriciels de vérification pour prouver des propriétés de plus en plus spécialisées à propos du programme.
Pour répondre au besoin de traductions multiples du programme source vers différents cadriciels de vérification de programme ayant chacun leur paradigme de preuve, nous défendons l'utilisation de langages dédiés orientés vers la preuve. Ces langages dédiés devraient être pensés comme une sur-couche au dessus des cadriciels de preuves, avec un design qui incorpore et distribue les obligations de preuves entre les prouveurs.
De plus, le programme originel est souvent déjà traduit depuis des spécifications d'exigences informelles liées au domaine d'activité afférent. Afin de raffermir le maillon le plus haut de la chaîne de confiance, nous soutenons que les langages dédiés orientés vers la preuve peuvent aider les experts du domaine à relire la spécification du programme, spécification à la base d'ultérieurs développements d'implantations vérifiées.
Cette dissertation traite du design et de l'utilité des langages dédiés orientés vers la preuve au travers de cinq études de cas. Ces études de cas portent sur des domaines allant des implantations cryptographiques aux systèmes experts légaux, et sur des logiciels à haut niveau d'assurance actuellement utilisés en production.
Chaque étude de cas donne son nom à l'un des chapitres de cette dissertation. LibSignal* est une implémentation vérifiée du protocole cryptographique Signal à destination du Web. hacspec est un langage dédié pour les spécifications cryptographiques en Rust. Steel est un cadriciel de vérification de programme utilisant la logique de séparation à l'intérieur de l'assistant de preuve F*. MLANG est un compilateur pour un langage dédié aux calculs fiscaux utilisé par la DGFiP. Enfin, **Catala** est un nouveau langage qui permet l'encodage de spécifications législatives dans un code source exécutable et analysable.

MOTS CLÉS

vérification de programme, langage dédié, méthodes formelles, cryptographie, droit

ABSTRACT

Program verification consists in analyzing a computer program as a formal artifact in order to prove the absence of certain categories of bugs before execution. But to use a program verification framework, one has to first translate the original source code of the program to verify in the formal language of the framework. Moreover, one might use different verification frameworks to prove increasingly specialized properties about the program.
To answer the need for multiple translations of the source program to various program verification frameworks with different proof paradigms, we advocate for the use of proof-oriented domain-specific languages. These domain-specific language should act as a frontend to proof backends, with a language design that incorporates and distributes the proof obligations between provers.
Moreover, the original program has often already been translated from informal domain-specific requirements that act as its specification. To close the top layer of the chain of trust, we claim that proof-oriented domain-specific language can help domain experts review the program specification at the base of formally verified implementation developments.
This dissertation discusses the design and usefulness of proof-oriented domain-specific languages in five case studies. These case studies range from the domain of cryptographic implementations to legal expert systems, and often target real-world high-assurance software.
Each of the case study gives its name to a chapter of this dissertation. LibSignal* is a verified implementation of the Signal cryptographic protocol for the Web. hacspec is a domain-specific language for cryptographic specifications in Rust. Steel is a separation-logic-powered program verification framework for the F* proof assistant. MLANG is a compiler for a tax computation domain-specific language used by the French tax authority. Finally, **Catala** is a novel language for encoding legislative specifications into executable and analyzable artifacts.

KEYWORDS

programme verification, domain-specific language, formal methods, cryptography, law