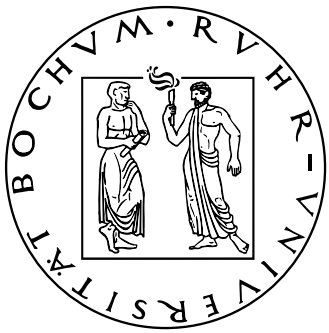
Securing Verified Monadic F* Programs against Linked Unverified Code

Cezar-Constantin Andrici

Dissertation

December 2026

Securing Verified Monadic F* Programs against Linked Unverified Code



Dissertation

---

zur Erlangung des Grades eines Doktors der Naturwissenschaften
der Fakultät für Informatik
an der Ruhr-Universität Bochum

---

von Cezar-Constantin Andrici

Bochum, im Dezember 2026

Author information
Name: Cezar-Constantin Andrici
Place of birth: Iași, Romania
cezar.andrici@gmail.com
https://cezarandrici.com

Reviewers:
Cătălin Hriţcu
Tenured Faculty, Max Planck Institute for Security and Privacy, Germany
Clara Schneidewind
Research Group Leader, Max Planck Institute for Security and Privacy, Germany
Yannick Forster
Tenured Researcher, Inria Paris, France

Thesis submitted: August 17, 2026
Thesis defense: December 2026
Last revision: tba

Abstract

Shallow embeddings that use monads to represent effects are popular in proof-oriented languages because they are convenient for formal verification. In F*, for instance, embedded programs with effects such as input-output (IO) or mutable state can be annotated with refinement types and pre- and postconditions, which allows verifying their correctness and security. Verified programs are then extracted to mainstream languages like OCaml or C and linked into larger codebases. This last step, however, voids the formal guarantees established through verification: bugs and security vulnerabilities can slip back in. Extraction is unverified, so it can itself introduce bugs, and it erases the specifications, turning the statically enforced interface of the verified program into mere assumptions, which the surrounding unverified code—whether buggy, compromised or malicious—can easily invalidate.

This dissertation introduces novel techniques to protect verified monadic F* programs against linked unverified code in three settings. First, we secure IO programs, verified to satisfy trace properties, by enforcing their specification on the interface with the unverified code using a verified combination of higher-order contracts and reference monitoring. Second, we secure stateful programs that dynamically share ML-style mutable references with unverified code by tracking during verification which references are shared and guaranteeing that non-shared ones remain unchanged across calls into unverified code. And third, we securely extract IO programs using translation validation in a novel way, in which only a first step has to be validated, while the second step is verified once and for all. For each of the three settings, the dissertation introduces a secure compilation framework targeting an ML-like language. The three frameworks are themselves written in F*, and verified in F* to provide *soundness* and *security* guarantees. *Soundness* guarantees that if one verifies a global specification of a source program, then the specification is satisfied when the compiled program is linked with arbitrary *unverified* code. *Security* guarantees that no additional security violation is introduced by compilation, established by proving Robust Relational Hyperproperty Preservation (RrHP), a very strong secure compilation criterion that implies full abstraction as well as preservation of all trace properties and hyperproperties against arbitrary linked adversarial code. The frameworks are illustrated on case studies, including a simple web server that securely interacts with unverified request handlers and a verified cooperative multi-threading scheduler that securely interacts with unverified threads. Building the frameworks required putting into practice recent work on how to verify monadic programs in F* using Dijkstra monads, leading to contributions of independent interest: partial Dijkstra monads, which make monadic embeddings usable in F*; support for verifying IO programs with SMT automation; and the first monadic representation of F*'s Monotonic State effect, with a machine-checked soundness proof.

All formal results in this thesis have been formalized in the proof-oriented language F*.

Kurzfassung

This is a machine translation of the abstract. I do not speak German, so I cannot guarantee that it's accurate. Please refer to the English version of the abstract.

Flache Einbettungen (*shallow embeddings*), die Monaden zur Darstellung von Effekten verwenden, sind in beweisorientierten Sprachen beliebt, da sie sich gut für die formale Verifikation eignen. In F* etwa können eingebettete Programme mit Effekten wie Ein-/Ausgabe (IO) oder veränderlichem Zustand mit Verfeinerungstypen (*refinement types*) sowie Vor- und Nachbedingungen annotiert werden, wodurch sich ihre Korrektheit und Sicherheit verifizieren lässt. Verifizierte Programme werden anschließend in verbreitete Sprachen wie OCaml oder C extrahiert und in größere Codebasen eingebunden. Dieser letzte Schritt hebt jedoch die durch die Verifikation erzielten formalen Garantien wieder auf: Fehler und Sicherheitslücken können sich erneut einschleichen. Die Extraktion ist unverifiziert und kann daher selbst Fehler einführen; zudem löscht sie die Spezifikationen und verwandelt so die statisch durchgesetzte Schnittstelle des verifizierten Programms in bloße Annahmen, die der umgebende unverifizierte Code—sei er fehlerhaft, kompromittiert oder bösartig—leicht verletzen kann.

Diese Dissertation stellt neuartige Techniken vor, um verifizierte monadische F*-Programme in drei Szenarien gegen eingebundenen unverifizierten Code zu schützen. Erstens sichern wir IO-Programme ab, für die verifiziert wurde, dass sie Trace-Eigenschaften erfüllen, indem wir ihre Spezifikation an der Schnittstelle zum unverifizierten Code mittels einer verifizierten Kombination aus Verträgen höherer Ordnung (*higher-order contracts*) und Laufzeitüberwachung (*reference monitoring*) durchsetzen. Zweitens sichern wir zustandsbehaftete Programme ab, die veränderliche Referenzen im ML-Stil dynamisch mit unverifiziertem Code teilen, indem wir während der Verifikation nachverfolgen, welche Referenzen geteilt werden, und garantieren, dass die nicht geteilten Referenzen über Aufrufe in den unverifizierten Code hinweg unverändert bleiben. Und drittens extrahieren wir IO-Programme auf sichere Weise, indem wir Übersetzungsvalidierung (*translation validation*) auf neuartige Weise einsetzen: Nur ein erster Schritt muss validiert werden, während der zweite Schritt ein für alle Mal verifiziert ist. Für jedes der drei Szenarien stellt die Dissertation ein Framework für sichere Kompilierung vor, dessen Zielsprache ML-artig ist. Die drei Frameworks sind selbst in F* geschrieben und in F* verifiziert, sodass sie *Korrektheit* (*soundness*) und *Sicherheit* (*security*) garantieren. *Korrektheit* garantiert, dass eine verifizierte globale Spezifikation eines Quellprogramms auch dann erfüllt ist, wenn das kompilierte Programm mit beliebigem *unverifiziertem* Code gebunden wird. *Sicherheit* garantiert, dass durch die Kompilierung keine zusätzlichen Sicherheitsverletzungen entstehen; nachgewiesen wird dies durch einen Beweis von Robust Relational Hyperproperty Preservation (RrHP), einem sehr starken Kriterium für sichere Kompilierung, das sowohl volle Abstraktion (*full abstraction*) als auch die Erhaltung aller Trace-Eigenschaften und Hypereigenschaften gegenüber beliebigem eingebundenem gegnerischem Code impliziert. Die Frameworks werden anhand von Fallstudien veranschaulicht, darunter ein einfacher Webserver, der sicher mit unverifizierten Request-Handlern interagiert, und ein verifizierter kooperativer Multithreading-Scheduler, der sicher mit unverifizierten Threads interagiert. Der Aufbau der Frameworks erforderte die praktische Umsetzung neuerer Arbeiten zur Verifikation monadischer Programme in F* mittels Dijkstra-Monaden und führte zu Beiträgen von eigenständigem Interesse: partielle Dijkstra-Monaden, die monadische Einbettungen in F* praktisch nutzbar machen; Unterstützung für die Verifikation von IO-Programmen mit SMT-Automatisierung; und die erste monadische Repräsentation des Monotonic-State-Effekts von F*, samt maschinengeprüftem Korrektheitsbeweis.

Alle formalen Ergebnisse dieser Dissertation wurden in der beweisorientierten Sprache F* formalisiert.

Acknowledgements

My doctoral studies took almost five years, and they were a rewarding journey. I was very fortunate to spend them abroad, at the Max Planck Institute for Security and Privacy in Bochum: an international and diverse place, demanding in what it expects of research and generous in what it offers in return. It sent me to conferences and summer schools around the world, and I met remarkable people along the way. Thank you to everyone who makes such opportunities possible.

I consider myself extraordinarily lucky to have ended up working with my advisor, Cătălin Hriţcu, who first took me on as a summer intern at Inria Paris in 2019. I am grateful for that chance: I knew almost nothing about any of the things I ended up doing with him, and I loved all of it. The following year, just after the first COVID lockdown ended, he called to ask whether I would like to come to Bochum for a second internship in a few weeks. I said yes, and the internship thankfully ended before the second lockdown began. It was that second internship that convinced me to do research, and to do a PhD. Since his funding was drying up, I applied to a few doctoral schools, and he guided me through the entire process, advising me on where to go and reviewing my applications. That left me with many opportunities to pick from, but in a turn of events—I liked Cătălin, I liked the project, and my wife and I both liked Bochum—I chose to continue working with him. What also attracted me to the MPI was its philosophy on doctoral studies: a PhD is not about the research, but about the individual, about becoming someone who can do research independently. This also means that the journey can be shorter or longer depending on the person, and that mattered to me. I think I have become that someone. I am deeply grateful to Cătălin for this, for the freedom he gave me, for being *always* supportive, for letting me do the hard work from the beginning (especially writing intros), for his patience, and for all his advice along the way. Thank you, Cătălin!

I am grateful to Clara Schneidewind and Yannick Forster for agreeing to serve on my committee, for the time and care they put into reading this thesis, and for the feedback they provided.

Thank you to the programming languages and security communities: to the many anonymous reviewers who read this work as it took shape, and to everyone who discussed this work with me and gave feedback. All of it made the work better.

I also want to thank my previous advisor, Ștefan Ciobâcă, who hooked me on software verification from his very first presentation, and later introduced me to Cătălin. I wrote my first paper with him, and we collaborated again on the first paper of my PhD. From him I learned to treat workshops as a chance to settle a project's direction and have it reviewed by others, which always turned out to be a great step towards a paper. He has continued to advise me over the years, and has kept me connected to academic life in Romania.

I want to thank Exequiel Rivas for advising me during my first internship and teaching me about Dijkstra monads, Guido Martínez for showing me all the ins and outs of F*, Kenji Maillard for then making sure I also learned many of the limitations of Dijkstra monads and F*, Éric Tanter for helping me navigate the literature on gradual verification and higher-order contracts, Théo Winterhalter for elucidating my questions about type theory, and Danel Ahman for helping me understand algebraic effects. Thank you all for the long discussions, for everything you taught me, and for these years of working together. I am grateful to Éric and Danel for writing reference letters on my behalf. I am also grateful to everyone who develops and maintains F*: this thesis rests on their work, and they were always friendly and responsive to my queries and bug reports.

It was great to be part of the Formally Verified Security group. Thank you to Carmine, Jérémy, Rob, Théo, Lef, Dongjae, Aïna, Maxi, Sebastian, Léon, Joseph, Raluca, Basile, Julay, Yonghyun, Jonnathan, and Abigail: the colleagues I had over the years are what made working from the office a pleasure. One rewarding part of the PhD was being a teaching assistant for two courses, and for that I have to thank Federico, Sebastian, and Sven: coordinating with them was easy, the material we put together turned out great, and the grading went smoothly. Another was supervising two interns, Ruxandra Icleanu and Abigail Pribisova, and seeing the work we did together turn into published results.

Beyond my own group, I want to thank all my colleagues at the MPI for the great atmosphere at the institute, which comes as much from the diverse and very cool research going on all around as from the many social events they put together over the years. I only ever organized a handful of walks from the institute to the Kemnader See, and I hope someone keeps them going. Thank you to Abraham, Selin, Smirity, and Jing for the book club, and my apologies for not managing to finish the books these last few months. And thank you to the administrative and IT staff at the institute, who kept everything running smoothly.

I am also grateful to Danel Ahman for having me visit the University of Tartu in Estonia (March 2025), where we tried to construct a Dijkstra monad for concurrency; to Son Ho for having me over at Azure Research in Cambridge, UK (December 2025), to work on verifying Rust code that uses raw pointers; and to Amin Timany for having me visit the Aarhus University in Denmark (May 2026), where I learned about Guarded Interaction Trees. It was great to visit these places, to discuss the projects going on there, and to meet the people behind them.

In the summer of 2026, I had the privilege of interning at Azure Research in Cambridge, UK, in the Security and Privacy group. Thank you to my collaborators Son, Antoine, Cédric, Haoyi, and Mantas for the interesting work we did together. It was fascinating to work on scaling the verification of Rust code with separation logic in Lean, with the help of AI agents.

I want to thank my parents for their unconditional support, love, and guidance, both long before any of this started and every day since. Thank you to them, and Ema's parents, and Diana, for showing us what a family is—something we now hope to pass on to our own child. And thank you to the rest of our family and to our friends, for being so happy to see us whenever we came back to Romania, for hosting us in other countries, and for visiting us in Bochum.

Finally, I want to thank my wife, Ema, for being an amazing person and a constant source of support during my PhD. We moved together from Romania to Germany, and we made these into great years. I have to admit that she was the one who gave me the courage to have a child in the middle of it all, and what a great decision that turned out to be. Seriously, kids are underrated. Thank you, Ema, for bringing so much fulfillment into my life.

And to our daughter, Otilia: you joined this journey halfway through, and you seem to have enjoyed it at least as much as I did.

Cambridge, August 2026

Contents

List of Figures

1 Introduction

Software bugs can cause severe malfunctions and security vulnerabilities. For example, the Heartbleed vulnerability was caused by a single missing bounds check, which risked exposing the confidential traffic of a substantial fraction of the internet [1].

[1]: Durumeric et al. (2014), *The Matter of Heartbleed*

Using formal verification, one can guarantee that a program satisfies correctness and security properties, ruling out whole classes of bugs and vulnerabilities. It requires defining a specification that characterizes how the program is supposed to behave, and then verifying that all possible executions of the program satisfy the specification. For example, one can verify that a web server responds to every request, guaranteeing availability; that a monitor enforces an access-control policy, guaranteeing confidentiality; or that a program does not corrupt a private key stored in its memory, guaranteeing integrity. Having a formal guarantee moves the trust from the program to the specification and to the tools that check it. A specification can be easier to trust than the program itself because it describes the program's behavior at a higher level of abstraction, omitting implementation details. For example, to verify that a web server responds to every request, one can view its behavior as the traces of system calls it can perform, requiring each read from a socket to eventually be followed by a write; and to verify the integrity of a private key, one can view the program's behavior as a relation between the initial and the final state of its memory, requiring the value of the key to stay unchanged.

The ease of formally verifying a program depends in part on the programming language in which it is written. Most programs do more than compute values: they also perform *effects*, such as interacting with the operating system (Input-Output) or mutating memory. Supporting many effects can be desirable, but it makes verification hard: if the language does not have a mechanism to track which effects are used, then reasoning about just a part of a program has to account for all the effects—even those the program part does not use. Therefore, it is easier to verify a program written in a language that contains only the effects necessary to implement it.

In this thesis we work with F* [2], a state-of-the-art proof-oriented language for writing and verifying programs. F*'s type system tracks the effects each program part uses. At its core is a pure total subset, in which programs have no effects and always terminate. Starting from this subset, F* is extensible with user-defined effects [3]. Therefore, within F*, one can *shallowly embed* a language that has only the necessary effects: one embeds the effects using monads, while everything else—variables, functions, control flow, data types—is inherited from F*. A program written in the embedded language is thus itself an F* program, which one can annotate with specifications using refinement types and pre- and postconditions, and then prove that the program satisfies the specification (with the help of SMT automation or tactics) obtaining a formal guarantee of correctness. After verification, to execute a shallowly embedded program, it is *extracted* to a target language like OCaml [4] or C [5], where each operation of the embedded effects (e.g., opening a file) is mapped to corresponding code in the target language. One success story of this approach is EverCrypt [6]: a cryptographic provider written and verified in F* (specifically its Low* [5] embedded language) and then extracted to C code that was integrated into Mozilla Firefox, the Linux kernel, and WireGuard [7].

[2]: Swamy et al. (2016), *Dependent Types and Multi-Monadic Effects in F**

[3]: Rastogi et al. (2021), *Programming and Proving with Indexed Effects*

[5]: Protzenko et al. (2017), *Verified Low-Level Programming Embedded in F**

[6]: Protzenko et al. (2020), *EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider*

[7]: Ahman et al. (2025), *Project Everest: Perspectives from Developing Industrial-grade High-Assurance Software*

Linking extracted F* programs into a larger unverified codebase, however, leads to *losing the formal guarantees* established through verification: bugs and security vulnerabilities can slip back in. There are two reasons for this. First, extraction

is unverified: the translation itself can introduce bugs, and each operation of the embedded effects is only assumed to behave like the target code it is mapped to. Second, the surrounding unverified code with which the verified code is linked (e.g., Firefox or Linux kernel) has to be trusted: it could inadvertently or maliciously (if compromised by an attacker) break the internal invariants of the verified code, leading to unsoundness and insecurity. Trusting the unverified code is especially dangerous because extraction erases the annotated specifications (refinement types and pre- and postconditions), since they have no correspondence in the target language. Erasure is problematic for the specifications on the interface of the verified program—i.e., preconditions that have to be satisfied before calling into the verified program (e.g., that an index is within bounds), and postconditions that have to be satisfied when returning control back to the verified program. In F*, such specifications are enforced statically by typing, but after extraction they become *assumptions* that unverified code can easily invalidate. While details can vary, these problems are not specific to F*: they are general to other proof-oriented languages such as Rocq, Lean, Dafny, Isabelle or HOL4.

1.1 Contributions at a High Level

This thesis presents three formally secure compilation frameworks that take verified effectful F* programs (i.e., source programs that use monadic effects such as IO or mutable state) to ML-like target languages. The three frameworks are themselves written in F* and are verified to provide two formal guarantees about the compiled program:

- **Soundness** guarantees that if one verifies a global specification of a source program, then the specification is satisfied when the compiled program is linked with arbitrary *unverified* code.
- **Security** guarantees that no additional security violation is possible after compilation. We establish it by proving Robust Relational Hyperproperty Preservation (RrHP), the strongest secure compilation criterion of Abate et al. [8], which implies full abstraction as well as preservation of all trace properties and hyperproperties against arbitrary linked adversarial code.[1]

1: Soundness cannot be obtained for free by instantiating RrHP: the premise of RrHP requires showing that the source program *robustly* satisfies the specification, i.e., against every source context it may be linked with, which is a sophisticated proof that has to be redone for each program. Our soundness theorems have no such premise.

[9]: Maillard et al. (2019), *Dijkstra Monads for All*

The first two compilation frameworks focus on enforcing the existing specifications on the interface between the verified and the unverified code—by converting them into dynamic checks—so that soundness and security are achieved. Both the source and target languages of the frameworks are shallowly embedded in F*, the target language modeling an ML-like language. The third framework is about how to define inside F* extraction to a language different than F* and obtain the formal guarantees: it translates shallowly embedded programs to a deep embedding—i.e., a syntactic representation—in a lambda calculus extended with the necessary effects.

As part of these frameworks, we had to put into practice recent work [9] on how to define Dijkstra monads to verify monadic programs in F*, which led to interesting contributions that we present in Chapter 2: introducing *partial* Dijkstra monads, necessary for usable shallow embeddings in F*; adding support for verifying IO programs with SMT automation; and proposing the first monadic representation of Monotonic State [10] together with a machine-checked soundness proof.

We motivate and summarize each framework's contributions at a high-level next, and we discuss their underlying technical contributions in more detail at the beginning of the corresponding chapters of the thesis (§2.1, §3.1, §4.1, and §5.1).

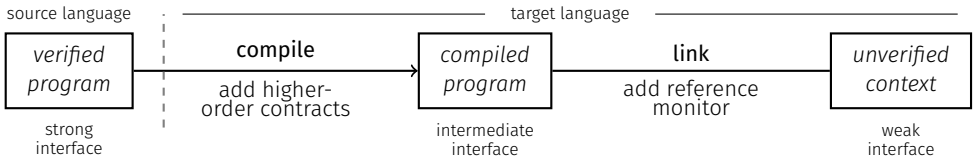


Figure 1.1: An overview of SCIO★. A strong interface contains refinements and pre- and postconditions, while a weak interface contains no specifications. An intermediate interface contains only the specification enforced by the reference monitor through the access-control policy.

1.1.1 Securing Verified IO Programs against Unverified Code

We start by introducing SCIO★, which provides a *formally verified* way to systematically convert into dynamic checks all the assumptions made by verified F★ programs with IO about linked unverified F★ code with IO—e.g., reading and writing files and network sockets. We look at this problem through the lens of secure compilation [8, 11]: **SCIO★ is a formally secure compilation framework for IO programs** consisting of a compiler and a linker between two languages shallowly embedded in F★.

[8]: Abate et al. (2019), *Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation*
[11]: Patrignani et al. (2019), *Formal Approaches to Secure Compilation: A survey of fully abstract compilation and related work*

The *source language* is an F★ subset in which a verified *partial source program* interacts with a *source context* via a *strong higher-order interface*, consisting of specifications expressed as refinement types and pre- and postconditions. Refinement types are used to constrain the values of a base type with a logical formula (e.g., the value of an integer is larger than zero). The pre- and postconditions of a function can depend on its arguments, can specify its result, and can also specify its IO behavior by considering the trace of past IO events—each time an IO operation is performed, an event containing the arguments and the result is appended to the trace. A precondition can constrain the trace of past IO events at the time the function is called (e.g., it could require that a file is currently open by looking back in the trace for an open event for the corresponding file descriptor and making sure that no close event happened afterwards for this descriptor). A postcondition can additionally take into account the IO events produced by the function itself when it returns (e.g., the function closed all file descriptors that it opened) and can also specify the relation between the result value and the return-time trace (e.g., the returned value was read from a file).

The *target language* is a smaller F★ subset in which the compiled program is linked with an adversarial *target context* that has a *weak interface* without refinement types, preconditions, or concrete postconditions (see Figure 1.1). The partial source program and the target context can only interoperate securely when their interfaces match, which is not the case in our setting because the strong interface of the verified program contains concrete specifications expressed as refinement types and pre- and postconditions, while the weak interface of the context does not.

SCIO★ enables a verified partial source program to be securely compiled and linked against an arbitrary target context. To achieve this secure interoperability SCIO★ uses a combination of (1) *reference monitoring* [12–14], introduced by the linker, and (2) *higher-order contracts* [15–19], introduced by the compiler. These techniques have, as far as we know, not been applied so far to our setting, where we are protecting statically verified code from unverified code, and where we aim for strong formal security guarantees. We use reference monitoring and higher-order contracts to bridge the gap between the strong interface of the source program and the weak interface of the untrusted target context by giving the compiled program an *intermediate interface* in between (see Figure 1.1).

(1) The reference monitor [12–14] records in its state information about the trace of IO events that happened so far during the execution. The monitor uses this information to enforce an *access control policy* by performing a dynamic check before each IO operation of the context (e.g., preventing access to a file descriptor associated with a password file or network socket). SCIO⋆ implements this by linking the target context with a secure IO library that dynamically enforces the policy. This policy enforcement on each IO operation of the context is necessary, because checking some postconditions when the context returns would be too late to prevent bad IO events from happening. For example, a postcondition of the context could state that it has neither accessed the passwords file, nor the network. If we only detect a violation of this policy after the context returns, the damage has already been done, with passwords leaked over the network. Instead, we use the monitor to prevent illegal IO events from actually happening, and the trace is then unaffected as we do not record such prevented events. However, reference monitoring is not enough on its own, since the strong interface contains refinement types and some pre- and postconditions that cannot be enforced at the level of IO operations, but have to be enforced on the higher-order boundary between the program and the context (e.g., the precondition of a callback sent to the context can require that its argument is an open file descriptor).

[12]: Anderson (1973), *Computer Security Technology Planning Study*
[13]: Ritchie et al. (1978), *The UNIX time-sharing system*
[14]: Ames Jr. (1981), *Security Kernels: A Solution or a Problem?*

(2) Higher-order contracts [15–19] are used by SCIO⋆ during compilation to convert the assumptions on the boundary between the partial program and the context into dynamic checks. The SCIO⋆ compiler wraps the program in new functions with weaker types and adds dynamic checks before each function call and after each function return crossing the boundary between program and context. This dynamically enforces refinement types on arguments and results, as well as pre- and some postconditions of functions. To enforce pre- and postconditions related to IO behavior, the higher-order contracts access the state of the monitor to get information about the IO events that happened before the function was called and during the execution of the function.

[15]: Findler et al. (2002), *Contracts for higher-order functions*
[16]: Disney et al. (2011), *Temporal higher-order contracts*
[17]: Scholliers et al. (2015), *Computational contracts*
[18]: Moore et al. (2016), *Extensible access control with authorization contracts*
[19]: Tov et al. (2010), *Stateful Contracts for Affine Types*

As mentioned above, SCIO⋆ is written in F⋆ and verified in F⋆ to be sound and secure.[2] SCIO⋆ supports both the case in which the untrusted context is a library used by the program and the case in which the program is a library used by the untrusted context (§3.6.5). In §3.2 we illustrate SCIO⋆ at work on a simple web server example.

2: Soundness is Theorem 3.6.1 and security is Theorem 3.6.2.

1.1.2 Secure Sharing of Mutable References between Verified and Unverified Code

We look next at how to enforce the specifications on the interface between the verified and the unverified code for stateful programs represented using a monadic effect called Monotonic State [10], instantiated to obtain first-order ML-style mutable references, which are dynamically allocated and unforgeable, which cannot be deallocated, and whose type cannot be changed. Monotonic State has facilitated the verification of large projects [20, 21] (e.g., EverCrypt [6]) that were later integrated into Firefox, Linux, and Windows, yet it offers no guarantees that linking a verified program with unverified code is sound or secure.

[10]: Ahman et al. (2018), *Recalling a Witness: Foundations and Applications of Monotonic State*
[20]: Zinzindohoué et al. (2017), *HACL*: A Verified Modern Cryptographic Library*
[21]: Swamy et al. (2022), *Hardening attack surfaces with formally proven binary format parsers*
[6]: Protzenko et al. (2020), *EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider*

The most challenging scenario for ensuring soundness involves dynamic sharing of dynamically allocated mutable references with unverified code. This is so because sharing a reference is a transitive property: the contents of the reference also gets shared, including any references reachable from it. Sharing is also permanent: once shared, a reference is shared forever as it can be stashed away by unverified code, and writes to a shared reference make the newly written values shared as well. It is

thus challenging to track which references are shared and which ones are not, so that one could still apply static verification. To more concretely explain our contributions, let's look at a **motivating example** that illustrates these points:

```
1 let prog (lib : (ref (ref int) → cb:(unit → unit))) =
2   let secret : ref int = alloc 42 in
3   let r : ref (ref int) = alloc (alloc 0) in
4   let cb = lib r in
5   r := alloc 1;
6   cb ();
7   assert (!secret == 42)
```

It consists of a program that takes as argument an unverified library. The library gets as input a reference to a reference to an integer, and returns a callback `cb` to the program. The program allocates a reference `r` (line 3) and passes it to the library (line 4). The library can store `r` and the reference `r` points to, and read and modify them in later callbacks. Thus, after the first call into the library (line 4), the program and the library share both `r` and the reference `r` points to. Between the initial call into the library and the call into the callback `cb` (line 6), the program modifies `r` pointing it to a newly allocated reference (line 5), and thus this new reference also becomes shared, even if not passed to `cb` directly—the sharing happens by updating an already shared reference. So in this example, a total of three references allocated by the program are shared with the library, even if only one reference was explicitly passed. Here is an adversarial library that writes to all shared references every time the callback is invoked:

```
let lib r =
  let g : ref (list (ref int)) = alloc [] in
  (λ () →
    g := !r :: !g;              (** saves where r points to each time the callback is called **)
    iter (λ r' → r' := 0) !g; (** writes to all these shared integer references **)
    r := alloc 0)               (** points r to a new location **)
```

To be able to verify a stateful program that calls into unverified code, for each such call one has to specify which of the program's references get modified, also called the footprint of a call. If one cannot determine such a footprint, then in order to ensure soundness, one is forced to conservatively assume that the call can modify *all* references, which makes verification very challenging, because one loses almost all properties of references. For instance, in `prog`, we would lose that `secret` equals `42`, and we would not be able to prove the assertion on line 7. Yet, as explained above, because references are dynamically allocated, and because of the transitive and persistent nature of their dynamic sharing, it is hard to determine which references are shared and which ones not.

To our knowledge, no existing technique allows for the sound verification of stateful programs that can be linked with arbitrary unverified code while sharing dynamically allocated mutable references with it. A simpler case considered by Dreyer, Neis, and Birkedal [22] (also imaginable for other separation logic work [23]) is to have no passing of references to unverified code (i.e., no dynamic sharing), but this cannot always be ensured, because existing code cannot always be clearly separated to avoid passing references. While one could consider rewriting the code to avoid reference passing by wrapping them in closures [24, 25], or in some cases to only pass references at abstract types, this would involve refactoring the unverified code to use a different interface. Other techniques allow passing references by checking and restoring the properties of the not passed references after a call into unverified code [26, 27], but this is done dynamically and adds an extra cost—for `prog` above, the contents of `secret`

[23]: Reynolds (2002), *Separation Logic: A Logic for Shared Mutable Data Structures*

[24]: Swasey et al. (2017), *Robust and Compositional Verification of Object Capability Patterns*

[25]: Sammler et al. (2020), *The high-level benefits of low-level sandboxing*

would be checked/restored twice, once for each call into the unverified code.

The solution we propose allows the sound verification of stateful F* programs even if the verified program is linked with unverified code and references are dynamically shared. Our solution builds on the idea that being shareable is a monotonic property of references, meaning that it can be treated as a logical invariant once it is established. Instead of tracking precisely which references cross the boundary between the verified and the unverified code, we overapproximate by adding a computationally irrelevant labeling mechanism in which fresh references are labeled as private, and that allows the user to dynamically label a reference as shareable, which is permanent. Our overapproximation is sound since we statically verify that only references that are labeled as shareable can cross into unverified code. In our example, this requires the user to add the following calls to label_shareable:

```
1 let safe_prog lib =
2   let secret = alloc 42 in
3   let r = alloc (alloc 0) in label_shareable (!r); label_shareable r;
4   let cb = lib r in
5   let v = alloc 1 in label_shareable v; r := v;
6   cb ();
7   assert (!secret == 42)
```

To correctly overapproximate, we also maintain a global invariant on the heap stating that shareable references always point only to other shareable references—preventing one from accidentally pointing a shareable reference to a not shareable reference (e.g., the original r:=alloc 1 would not be allowed on line 5 because the fresh reference returned by alloc 1 is labeled private). The labels simplify specifying what the unverified code can do—it can modify only shareable references—which allows preserving the properties of private references (e.g., one is able to prove the assertion on line 7 since secret is labeled private). Note that we have only one notion of references for which we keep track of how the labels change dynamically.

We realize this solution as SecRef*, a secure compilation framework inspired from SCIO*, which is also written and verified in F* to be sound and secure, and whose source and target languages are likewise shallowly embedded in F*, in this case on top of Monotonic State.[3] The guarantee that the contents of private references are unchanged by calls into unverified code is exposed in the interface on which the verified program can rely, and does not require introducing a reference monitor to enforce it. The remaining refinement types and pre- and postconditions that the verified code expects from the unverified code are converted into dynamic checks about the shared references by using higher-order contracts.

3: Soundness is Theorem 4.5.3 and security is Theorem 4.5.4.

SecRef* relies on the first monadic representation for the Monotonic State effect (§2.5), which is a contribution of our work that can be of independent interest. Finally, we use SecRef* to build a simple cooperative multi-threading scheduler that is verified and that securely interacts with unverified threads (§4.6).

1.1.3 Formally Secure Extraction of IO Programs

Until now, we focused on how to enforce the specifications that appear on the interface between the verified and the unverified code. Extraction of F* programs is, however, still unverified and part of the trusted computing base.

To address this, we aim to provide *formal security guarantees* for extracted F* code even when linked with arbitrary unverified code. We are not aware of any proof-oriented language whose extraction provides such strong secure compilation guaran-

tees [8, 11]. Most existing work focuses on verifying *correctness* of extraction [28–33], not security, and, moreover, extraction is not fully verified. To fully verify extraction of shallowly embedded programs, one would have to define the programs, define extraction, and connect the two, *all within the same language*. This represents a significant challenge: the programs are *shallowly embedded*, whereas the extraction function operates on a *deep embedding* [28]. Connecting the two requires a metaprogram, and the most direct way of obtaining formal guarantees would seem to be verifying this metaprogram. This is challenging, for a start because verifying a metaprogram requires a mechanized meta-theory for the proof-oriented language, but there has been steady progress towards this goal for many such languages [32–37]. Another challenge though is that the standard primitive connecting a shallow and a deep embedding is *quotation*—a metaprogram that translates a shallowly embedded program into its deep embedding. As far as we know, quotation has not been verified for any proof-oriented language, and even for Rocq, which has a mature mechanized meta-theory [37], verifying quotation remains a big unsolved challenge.*

[28]: Forster et al. (2024), *Verified Extraction from Coq to OCaml*

[37]: Sozeau et al. (2025), *Correct and Complete Type Checking and Certified Erasure for Coq, in Coq*

Instead of verifying extraction, other works *certify* each extraction result using translation validation to obtain *formal correctness guarantees* [38–42]. A metaprogram is used on a shallowly embedded program to generate both the extracted code and a proof that the two are related. This is an instance of translation validation because to ensure trust each such proof has to be checked by the proof-oriented language (e.g., type-checked), which can also fail because the metaprogram is not verified and the proof can be wrong too or fail to verify (e.g., fail during unification or SMT solving), thus, ideally, one would limit the use of translation validation.

[38]: Pit-Claudel et al. (2022), *Relational compilation for performance-critical applications: extensible proof-producing translation of functional models into low-level code*

[39]: Myreen et al. (2014), *Proof-producing Translation of Higher-order logic into Pure and Stateful ML*

[40]: Abrahamsson et al. (2020), *Proof-Producing Synthesis of CakeML from Monadic HOL Functions*

[41]: Mullen et al. (2018), *Œuf: minimizing the Coq extraction TCB*

[42]: Forster et al. (2019), *A Certifying Extraction with Time Bounds from Coq to Call-By-Value Lambda Calculus*

We build on this idea, but structure certifying extraction in a new way that minimizes the reliance on translation validation, confining it to the first of two extraction steps and verifying the second once and for all. The first step, which we call *relational quotation*, defines a typing relation that precisely characterizes the shallowly embedded source language [43] and uses a metaprogram to construct a typing derivation for the given program. This metaprogram is simple, since the typing derivation follows the structure of the original program. Once we validate the typing derivation against the original program—which happens simply by type checking the derivation—we pass it to the second step: a verified syntax-generation function that generates the corresponding target syntax, which we can semantically relate back to the original program because the typing relation uses dependent types to tie a derivation to the specific program it was constructed for.

[43]: Prinz et al. (2022), *Deeper Shallow Embeddings*

We apply this general idea to build SEIO*, a framework for extracting shallowly embedded F* programs with IO and refinement types to a deeply embedded simply typed λ-calculus while providing formal secure compilation guarantees. The formal guarantees of SEIO*—i.e., soundness and security[4]—go beyond the state of the art in verified and certifying extraction, which so far has focused on correctness rather than security.

4: Soundness is Theorem 5.5.2 and security is Theorem 5.5.3.

* Verification of quotation was started in MetaRocq [37], but (as of 12 August 2026) it has not progressed much in the last 3 years: https://github.com/MetaRocq/metarocq/tree/9.1/quotation/theories

1.2 List of Publications and Outline

The results presented in this thesis were published in the following peer-reviewed conference papers. I was the main author of these publications:

[44] Cezar-Constantin Andrici, Ştefan Ciobâcă, Catalin Hritcu, Guido Martínez, Exequiel Rivas, Éric Tanter, and Théo Winterhalter. "Securing Verified IO Programs Against Unverified Code in F*." In: *Proc. ACM Program. Lang.* 8.POPL (2024)

[45] Cezar-Constantin Andrici, Danel Ahman, Cătălin Hriţcu, Ruxandra Icleanu, Guido Martínez, Exequiel Rivas, and Théo Winterhalter. "SecRef*: Securely Sharing Mutable References between Verified and Unverified Code in F*." In: *Proc. ACM Program. Lang.* 9.ICFP (2025)

[46] Cezar-Constantin Andrici, Abigail Pribisova, Danel Ahman, Cătălin Hriţcu, Exequiel Rivas, and Théo Winterhalter. "Misquoted No More: Securely Extracting F* Programs with IO." in: *Proc. ACM Program. Lang.* 10.ICFP (2026)

It also draws on the following additional (i.e., not included in the publications above) peer-reviewed extended abstracts presented at a conference (TYPES) and a workshop (HOPE), both without pre-proceedings:

[47] Théo Winterhalter, Cezar-Constantin Andrici, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, and Exequiel Rivas. *Partial Dijkstra Monads for All.* TYPES. 2022

[48] Cezar-Constantin Andrici, Théo Winterhalter, and Cătălin Hriţcu. *Verifying non-terminating programs with IO in F*.* HOPE. 2022

Origin of Text. The following list details the origin of the text of the individual chapters and sections:

Chapter 1 is new text mixed with contributions taken from the papers' abstracts and introductions;
§2.2 is the F* primer we had in papers greatly expanded with new text covering the language features used throughout the thesis, e.g., dependent and refinement types, universes, and the effect system;
§2.3 is based on the extended abstract on partial Dijkstra monads [47] extended with new text introducing what Dijkstra monads are and more details about Maillard et al. [9]'s recipe to build one;
§2.4 is based on the SCIO* paper [44] and the extended abstract on verifying non-terminating programs with IO [48], reworked with a Freer monad and with the explanations expanded;
§2.5 is based on the SecRef* paper [45], with §2.5.2 reworked with a Freer monad and the explanations are expanded;
§2.6 is based on the SCIO* paper [44], the SecRef* paper [45], and the extended abstract on verifying non-terminating programs with IO [48];
Chapter 3, Chapter 4, and Chapter 5 are text taken from the SCIO* [44], SecRef* [45], and SEIO* [46] papers, respectively. In each of them, the contributions (§3.1, §4.1, and §5.1), the related work (§3.9, §4.7, and §5.8), and the summary are reworked, while the rest is updated to correctly reference the other chapters and revised here and there. Additionally, §3.3 now only explains how the MIO effect differs from the IO effect of §2.4;
Chapter 7 is new text mixed with contributions taken from the three papers.

I wrote the new text of this thesis—which serves to contextualize previously mentioned papers and extended abstracts into a cohesive whole. I used generative AI (Claude Code) to assist with writing this new text, directing the AI interactively at the sentence level to expand and refine my own draft sentences.

Outline. Chapter 2 gives an introduction to F* (§2.2) and how to verify effectful programs using Dijkstra monads, and how to define one (§2.3). It also introduces partial Dijkstra monads necessary in F* (§2.3.3). Then, it goes forwards showing how to define Dijkstra monads to verify IO programs (§2.4), stateful programs (§2.5) and non-terminating programs (§2.4.2). Chapter 3 presents SCIO*, our secure compilation framework for verified programs with IO. Chapter 4 presents SecRef*, our secure compilation framework for verified stateful programs that dynamically share mutable references with the unverified code. Chapter 5 presents SEIO*, our formally secure extraction framework for F* programs with IO. We conclude in Chapter 7 by discussing how the three frameworks could be unified, and the remaining challenges towards a longer-term goal, which we explain next.

Longer-Term Goal. Achieving strong secure compilation criteria such as RrHP is extremely difficult. We see this thesis as several important steps towards building a formally secure compilation chain from F* to a safe subset of OCaml. This is a formidable research challenge though, so we carefully limited the scope of this work by focusing on IO and mutable state as the only side effects (studied separately), and by assuming that the unverified code is written either in a shallowly embedded subset of F* that respects a weak interface (for SCIO* and SecRef*) or in a deeply embedded simply typed λ-calculus (for SEIO*). Importantly, RrHP is vertically composable—chaining two compilation steps that each satisfy RrHP yields a compilation chain that satisfies RrHP as well—which gives us hope that our frameworks can be composed in the future. We discuss the remaining challenges in composing the three frameworks, and reaching the goal in Chapter 7.

1.3 Artifacts

All the papers and extended abstracts this thesis is based on are accompanied by publicly available artifacts. The artifacts contain the full implementations of SCIO*, SecRef*, and SEIO*, as well as a full mechanization in F* of their soundness and security guarantees and of the results presented in this thesis. The artifacts of the papers were peer-reviewed by the Artifact Evaluation Committees of the corresponding conferences and were awarded the reusable badge. A bundle of all these artifacts can be found in the following repository:

`https://github.com/andricicezar/fstar-io`

Generative AI was used for developing part of the SEIO* artifact as disclosed at the end of §5.1. The author takes responsibility for the artifact development. The artifacts depend on version v2026.03.24 of F*: newer versions have since changed F*'s theory and may further rethink the effect mechanism to focus on Separation Logic and improve its SMT automation. We discuss in Chapter 6 how the results of this thesis could apply to other proof-oriented languages.

2 F* and Dijkstra Monads

2.1 Contributions and Outline

This chapter gives a primer of F* and introduces Dijkstra monads, the tools we use throughout the thesis to specify and verify effectful programs.

§2.2 recalls the F* syntax and notation used throughout the thesis. §2.3 then presents Dijkstra monads, monads indexed by specifications: it first introduces how Dijkstra monads are used to specify and verify effectful programs (§2.3.1), and then shows how to define one from a computational monad, a specification monad, and a monad morphism relating them (§2.3.2). Readers already familiar with F* and Dijkstra monads can safely skip this material and jump to the contributions described next.

§2.3.3 presents a contribution of this thesis: *partial Dijkstra monads* [47]. The general recipe of Maillard et al. [9] for constructing Dijkstra monads often produces Dijkstra monads that do not support partiality coming from preconditions, which makes them practically unusable in F*. We characterize the partiality required by F* in terms of a **require** operation, and we show how the recipe can be extended so that it produces partial Dijkstra monads, covering many usual effects such as state, IO, non-determinism, and unrecoverable exceptions. The effects defined in this thesis, including the representation of Monotonic State below, rely on this extension.

§2.4 presents another contribution: the IO effect, which supports verification of trace properties of IO programs. The effect is defined on top of a freer monad [49], and its specifications are pre- and postconditions over traces of IO events, engineered to take advantage of F*'s SMT automation. The IO effect is at the base of SCIO* (Chapter 3). We also show how the effect can be extended to non-terminating programs, which is needed to give meaningful specifications to infinite loops, such as the main loop of a web server.

[49]: Kiselyov et al. (2015), *Freer monads, more extensible effects*

§2.5 also presents a contribution that later chapters build on: the first monadic representation of Monotonic State [10], using a freer monad, which overcomes a limitation in the original work of Ahman et al. [10]. We provide this representation together with a machine-checked soundness proof in F*, done differently than the original paper proof. While we expect this monadic representation of Monotonic State to be generally useful, it is also crucial for the mechanized proofs of SecRef* (Chapter 4).

Artifact. The contributions of this chapter are mechanized in F* and Rocq. The formalization of partial Dijkstra monads accompanies the corresponding extended abstract [47]. The IO effect is part of the artifact of SCIO*, which is available on Zenodo [50] and GitHub [51]. The extension to non-termination is a standalone effect that accompanies the extended abstract on verifying non-terminating IO programs [48]. The representation of Monotonic State is part of the artifact of SecRef*, available on Zenodo [52] and GitHub [53]. All of these artifacts can also be found in the bundle of this thesis (§1.3). The formalization of partial Dijkstra monads is 4209 lines[1] of Rocq and F*. The IO effect is 624 lines, of which the Dijkstra monad—the computational monad, the specification monad, the monad morphism, and the proof that this morphism is lax—takes 273 lines. The extension to non-termination is 2474 lines, of which the Dijkstra monad takes 1006 lines (two thirds of that is the specification monad alone,

1: Comment and blank lines were not counted towards these numbers.

which is where accounting for divergence is paid for). The representation of Monotonic State is 524 lines, of which the Dijkstra monad takes 217 lines.

2.2 F* Primer

F* [2] is a dependently-typed language (like Rocq or Lean) part of the ML family of languages (its syntax is similar to OCaml: e.g., val, let, match, etc.). In here we explain some of the concepts of F* necessary to understand the rest of the thesis. For a complete reference check the online book on F* [54].

[2]: Swamy et al. (2016), *Dependent Types and Multi-Monadic Effects in F**

[54]: Swamy et al. (2026), *Proof-Oriented Programming in F* for v2026.03.24*

Here is a quick rundown of its syntax: lambda abstractions are written λ b_1 ... b_n→ t (where t ranges over both types and terms), whereas contiguous binders of the same type may be written (v_1 ... v_n: t). We omit the type in a binding when it can be inferred. More generally, one can write an underscore (_) in place of a type or term and let the F* type checker infer it. Pair types are written as t_1 & t_2, and are introduced using the notation (_, _). Dependent pair types, in which the type of the second component may depend on the value of the first, are written as x:t_1 & t_2, and are introduced using the notation (| _, _ |) (see Listing 2.1). Binding occurrences b take the form x:t or #x:t for an implicit argument. Type variables such as α and β stand for implicit type arguments whose binding occurrences are omitted. A special kind of implicit argument is a type-class constraint {| d : c t1 .. tn |}, which gets solved by a tactic during elaboration. Binary functions can be made infix by using backticks: x `op` y stands for op x y, e.g., l_1 `append` l_2 concatenates two lists. Finally, when writing explicitly monadic code one writes let! x = e_1 in e_2 for bind.

```
let pack (#a:Type) (x:a)
  : (t:Type & t)
  = (| _, x |)
```

Listing 2.1: pack packages a value together with its type. The underscore leaves the first component of the pair to be inferred, and in a call such as pack 42 the implicit type argument is inferred to be int.

New inductive types are defined using the type keyword, by listing the constructors of the type together with their signatures. For example, the type either a b in Listing 2.2 has two constructors, Inl and Inr, each taking a value of the corresponding type. Values of an inductive type are inspected with match, which branches on the constructor used to build the value. For each constructor, F* also automatically generates a boolean discriminator and a projector for each of its arguments: e.g., Inr? x tests whether x has the shape Inr y, and Inr?.v (Inr y) projects out y.

```
type either (a b:Type) =
| Inl : v:a → either a b
| Inr : v:b → either a b
```

Listing 2.2: either a b is the disjoint sum of a and b: a value is either an a tagged with Inl, or a b tagged with Inr.

Types are organized in a predicative, countable hierarchy of universes, with Type_0 being the lowest universe. We generally omit universe annotations. We use Type_0 also to write propositions, including ⊤ (True) and ⊥ (False).

Refinement types are written x:t{p}, where p is a proposition that may mention x, and represents a type inhabited by the values of t that satisfy p. For example, this is how natural numbers are defined in F*: nat = x:int{x≥0}. Since arrows are dependent, a refinement on the result of a function may mention its arguments: e.g., x:int → r:int{r == x+1} is the type of the successor function on integers. A refinement type x:t{p} is a subtype of t, so a value of the refined type can be used directly at type t. Conversely, using a value of type t at type x:t{p} makes F* generate a proof obligation. Such obligations are collected in a verification condition (one per top-level definition) that is discharged by an SMT solver or by tactics. Thus, unlike with a dependent pair, the proof of the refinement is not carried by the value: no proof term is ever constructed. As a synonym, the squash p type is defined as the refinement _:unit{p}, and can be seen as the type of computationally-irrelevant proofs of p.

Another distinctive feature of F* is its **effect system** [3], which tracks in the type of each computation the side effects it may perform (e.g., state, exceptions, IO). F* distinguishes value types from computation types: the function type b_1 → ... → b_n→ C is a value type, but its codomain C is a computation type, which bundles the effect

[3]: Rastogi et al. (2021), *Programming and Proving with Indexed Effects*

of the function together with its result type and, depending on the effect, further indices—often used to express a specification.

The default effect is Tot: e.g., int → int is equivalent to int → Tot int. Tot classifies total functions, which have no side effects and terminate on all inputs. Another primitive effect is Pure, which also classifies total computations, but has two further indices: a precondition on the inputs (introduced by the keyword requires) and a postcondition relating inputs and result (introduced by ensures). In fact, Tot t is a synonym for Pure t (requires T) (ensures λ _ → T). To call a Pure function, one first has to prove its precondition, and in exchange, one gets to assume the postcondition about the result. Conversely, when defining a Pure function, its body may assume the precondition, which makes it possible to define partial functions (e.g., Listing 2.3).

```
let get_inl (x:either α β)
  : Pure α
    (requires (Inl? x))
    (ensures λ _ → T)
  = match x with
    | Inl v → v
```

Listing 2.3: The match is partial—it handles only the Inl case—yet F⋆ accepts it, since the precondition rules out Inr-shaped arguments.

Finally, the effect system is user-extensible: one can define new effects by giving a representation for the effectful computations together with the combinators to compose them—this is the topic of the rest of this chapter. Different effects are related by *lifts*: a lift from effect E_1 into effect E_2 describes how to view a computation in E_1, together with its specification, as a computation in E_2, and is inserted automatically by F⋆ when the two effects are mixed. For example, later chapters define effects for IO and for mutable state, each equipped with a lift from Pure, so Pure code can be called directly in both these effects.

2.3 Dijkstra Monads as Effects in F⋆

In F⋆, an effect is based on an *indexed monad* [3]: effectful computations are represented by a monad whose type carries additional indices, and programs written in direct style are automatically elaborated by F⋆ into this monadic representation. We focus only on a specific kind of effects, at the core of how F⋆ verifies effectful programs [55]: effects whose monad is indexed by a *specification*, a structure called a Dijkstra monad [2].

[3]: Rastogi et al. (2021), *Programming and Proving with Indexed Effects*

[2]: Swamy et al. (2016), *Dependent Types and Multi-Monadic Effects in F**

2.3.1 Introduction to Dijkstra Monads

A Dijkstra monad D a w classifies effectful computations returning values of type a and obeying specification $w : W\ a$, where W is a specification monad. For instance, the state Dijkstra monad ST is specified by the monad W^{ST}, where state is the type of the state:

```
type W^ST (a:Type) = (a & state → Type₀) → (state → Type₀)
```

W^{ST} is the type of a predicate transformer taking a postcondition on the final state and a result value and returning a precondition on the initial state. We have the following Dijkstra monad interface, with the operations of ST on the left, their specifications in W^{ST} on the right, and the bind below[2]:

```
val return^ST : x:α → ST α (return^W x)
val get^ST : unit → ST state get^W
val put^ST : s:state → ST unit (put^W s)
```

```
let return^W x = λ p s₀ → p (x, s₀)
let get^W = λ p s₀ → p (s₀, s₀)
let put^W s = λ p _ → p ((), s)
```

2: We omit the binders of the specification variables $w_m : W^{ST}\ \alpha$ and $w_f : W^{ST}\ \beta$

```
val bind^ST : m:ST α w_m → f:(x:α → ST β (w_f x)) → ST β (bind^W w_m w_f)
let bind^W w_m w_f = λ p s_0 → w_m (λ (x, s_1) → w_f x p s_1) s_0
```

We explain the specifications on the right as follows. If we take a postcondition $p : \alpha * \text{state} \to \text{Type}_0$, we say it holds on program $\text{return}^{ST}\ x$ if we can prove $\text{return}^{W}\ x\ p$ on the initial state s_0, or in other words if $p\ (x, s_0)$ holds. For p to hold on $\text{get}^{ST}\ ()$ then it must hold on return value and final state both equal to the initial state: $p\ (s_0, s_0)$. For p to hold on $\text{put}^{ST}\ s$ it must hold on final state s and trivial unit value $()$: $p\ ((), s)$; the initial state is ignored. Such typed Dijkstra monad interfaces allow F* to compute verification conditions simply by dependent type inference.

To understand how the verification condition is computed, let us look at the following example of a stateful program where the state is just an integer, i.e., we instantiate state with int. The program incr below increments the state, and the user annotates it with a specification—call it w—stating that the final state is greater than or equal to the initial one:

```
let incr () : ST unit (λ p s_0 → ∀ s_1. s_1 ≥ s_0 ⟹ p ((), s_1)) =
  let x = get () in
  put (x + 1)
```

F* elaborates this program into the operations and binds of the Dijkstra monad: $\text{bind}^{ST}\ (\text{get}^{ST}\ ())\ (\lambda\ x \to \text{put}^{ST}\ (x + 1))$, and then infers its type using the interface above. The types of the operations $\text{get}^{ST}\ ()$ and $\text{put}^{ST}\ (x + 1)$ carry their specifications, get^{W} and $\text{put}^{W}\ (x + 1)$, as indices, and the type of bind^{ST} binds the two together with bind^{W}. The type inferred for incr is therefore $\text{ST unit } w_{\text{incr}}$, where the specification computed by F* is $w_{\text{incr}} = \text{bind}^{W}\ \text{get}^{W}\ (\lambda\ x \to \text{put}^{W}\ (x + 1))$, which unfolds to $\lambda\ p\ s_0 \to p\ ((), s_0 + 1)$. Note how the inferred specification is computed only from the specifications of the operations, not from their implementations.

The inferred specification does not have to match the annotated w: here w is weaker, as it only states that the state does not decrease. To connect the two, one needs an order[3] on W and a subsumption combinator for the Dijkstra monad:

```
val subcomp^ST : w:W^ST α → w':(W^ST α){w ⊑ w'} → ST α w → ST α w'
```

where $w \sqsubseteq w'$ and holds iff $\forall\ p\ s_0.\ w'\ p\ s_0 \implies w\ p\ s_0$—i.e., it is sound to replace the specification of a computation with one that requires a stronger precondition. It is the refinement $w \sqsubseteq w'$ that produces the verification condition: F* applies subcomp^{ST} to go from the inferred w_{incr} to the annotated w, producing the verification condition $w_{\text{incr}} \sqsubseteq w$, which unfolds to the following verification condition $\forall\ p\ s_0.\ (\forall\ s_1.\ s_1 \geq s_0 \implies p\ ((), s_1)) \implies p\ ((), s_0 + 1)$ that is discharged by the SMT solver (e.g., by instantiating s_1 with $s_0 + 1$).

Specifications written as predicate transformers, like the annotation of incr, are awkward to read and write. It is more natural to specify a program by a precondition on the initial state and a postcondition relating the initial state, the result, and the final state—the requires/ensures style we have seen on Pure in §2.2. One can offer this style on top of ST by a synonym that embeds a precondition-postcondition pair into a weakest precondition:

```
effect ST' (a:Type) (pre:state → Type_0) (post:state → a & state → Type_0) =
  ST a (λ p s_0 → pre s_0 ∧ (∀ r. post s_0 r ⟹ p r))
```

3: This order is a form of refinement [56] that compares specifications by precision. The intent is to overapproximate, *e.g.* the $\top$ precondition should $\sqsubseteq$ the $\bot$ precondition.

With the synonym, the annotation of incr can be written as:

val incr () : ST' unit (requires (λ _ → T)) (ensures (λ s_0 (_, s_1) → $s_1 \geq s_0$))

The user thus works with pre- and postconditions, but underneath F* still works with weakest preconditions: type checking simply unfolds the synonym, and then works as explained before.

One can use Dijkstra monads in any dependently-typed language [9], however, ending up with a verification condition may not be desired if one cannot use an SMT to discharge it like in F*.

2.3.2 How to Define a Dijkstra Monad

Dijkstra Monads for All (DM4All) [9] introduces a generic way to construct Dijkstra monads. For any computational monad M, and for any ordered specification monad W with order $\sqsubseteq$, if there is a monad morphism $\theta : M\ \alpha \to W\ \alpha$ then one can define the following Dijkstra monad:

[9]: Maillard et al. (2019), *Dijkstra Monads for All*

type D (a:Type) (w:W a) = m:(M a){θ m $\sqsubseteq$ w}

The idea is that a computation is indexed by w if its image by θ is smaller than w, which means that the specification computed by θ is refined by the one given in the index.

For instance, from the usual state computational monad St a = state → a & state to the specification monad W^{ST} of the previous section one can define the monad morphism θ m = λ p s_0 → p (m s_0). This construction of Dijkstra monads neatly separates the syntax (M) from the specification (W) and semantics (θ) as one can *forget* the refinement and extract the value in M a from D a w.

Having these ingredients, DM4All tells us how to define the return and bind of the Dijkstra monad D. The return of D is the return of M: given x : α, return^M x has to inhabit D α (return^W x), i.e., satisfy the refinement θ (return^M x) $\sqsubseteq$ return^W x, which follows from the morphism law for return, θ (return^M x) ≡ return^W x, where ≡ is the equivalence induced by the order: $w_1 \equiv w_2$ stands for $w_1 \sqsubseteq w_2 \wedge w_2 \sqsubseteq w_1$. The bind of D is the bind of M: given m : D α w_m and f : (x:α → D β (w_f x)), bind^M m f has to inhabit D β (bind^W w_m w_f), i.e., satisfy the refinement θ (bind^M m f) $\sqsubseteq$ bind^W w_m w_f, where θ (bind^M m f) ≡ bind^W (θ m) (λ x → θ (f x)) from the morphism law for bind. To prove bind^W (θ m) (λ x → θ (f x)) $\sqsubseteq$ bind^W w_m w_f, we need bind^W to be monotonic (i.e., to preserve $\sqsubseteq$) in both of its arguments, and then the goal follows from the refinements carried by m and f: θ m $\sqsubseteq$ w_m and ∀ x. θ (f x) $\sqsubseteq$ w_f x, which relate the arguments of the two bind^W applications pointwise. Thus, the requirements that θ is a monad morphism and that W is an order monad are what make return and bind of D definable. The subsumption combinator of D also relies on transitivity of $\sqsubseteq$.

While very general, the DM4All construction often produces Dijkstra monads that do not support partiality on standard computational monads, which makes them practically unusable in F*, as explained next.

2.3.3 Partial Dijkstra Monads for All

For a Dijkstra monad to be practically usable in F*, it must come with a lift from Pure. Without one, computations in the new effect cannot call existing Pure code, nor use

the machinery built on top of it (e.g., Lemma, assume, admit, Ghost). Even refinement types stop working: a program like

```
let id (x:int) : ST' (r:int{r ≥ 0}) (requires λ _ → x ≥ 0) = x
```

is rejected, because the proof obligation generated by the refinement $r \geq 0$ is a Pure verification condition, and without a lift from Pure F* has no way to discharge it from the effect's precondition.

The Pure effect (also a Dijkstra monad) of F* represents in fact *partial* computations, as for instance one can use the precondition to discard provably unreachable branches of a pattern-matching Listing 2.3, and recursive functions can loop on arguments not satisfying the precondition.

To understand the problem, compare the following two types, both of which allow the function to be called only on non-negative integers: $x{:}int\{x \geq 0\} \to int$ and $x{:}int \to Pure\ int\ (requires\ x \geq 0)$. The first is a total function that takes an explicit proof of $x \geq 0$ as an argument, which every call site has to supply. The second moves the proposition into the precondition of the effect, which moves the responsibility to enforce it to the Dijkstra monad—the problem with the DM4All construction from §2.3.2 is that computations m in $D\ a\ w$ can be coerced to type $M\ a$, by just forgetting the $\theta\ m \sqsubseteq w$ refinement, thus losing the precondition. This means that in order for D to support partiality, the underlying computational monad M should already support partiality.

We define a *partial Dijkstra monad* as a Dijkstra monad D whose computational monad M and specification monad W support partiality, i.e., have **require** operations, and which also has such an operation itself:

$$\begin{aligned}
&val\ require^M : pre{:}Type_0 \to M\ pre\\
&val\ require^W : pre{:}Type_0 \to W\ pre\\
&val\ require^D : pre{:}Type_0 \to D\ pre\ (require^W\ pre)
\end{aligned}$$

A **require** operation returns a proof of the proposition **pre** that can then be used by the continuation, and to be able to call it, one first has to prove the precondition **pre**. The **require** operations of M and W have to be related via the monad morphism, i.e., $\theta\ (require^M\ pre) \sqsubseteq require^W\ pre$. We argue that the existence of such an operation is tantamount to supporting partiality. We show that the DM4All construction can be made to produce partial Dijkstra monads—thus usable in F*—and that such monads have a lift from Pure.

Most computational monads do not accept such a **require** though. For instance, for the state monad, **require pre** would need to have type $state \to pre\ \&\ state$, which one cannot inhabit for an arbitrary $pre{:}Type_0$. We show that one can apply the state monad transformer to the partiality monad $G\ a = pre{:}Type_0\ \&\ (pre \to a)$, and keep W^{ST} since it supports $require^W\ pre = \lambda\ p\ s_0.\ \exists\ (h{:}pre).\ p\ h\ s_0$, to define a partial Dijkstra monad for state that has a lift from Pure.

We provide several ways to build computation (and specification) monads that support a **require** construct. First, we provide an account of *Dijkstra Monads for Free* [57] that fits in this setting. Basically, Dijkstra Monads for Free produces a partial Dijkstra monad from a computational monad obtained by applying a monad transformer T to the partiality monad $G\ a$ and the specification monad obtained by applying T to the continuation monad $W^{cont}\ a = (a \to Type_0) \to Type_0$. This confirms the empirical observation that Dijkstra Monads for Free yields Dijkstra monads that are usable in F*. Second, we provide a construction for adding an extra Require constructor to the signature of a freer monad, which we also use later (in §2.4 and §2.5), allowing for deep occurrences of **require** within computations. Together these cases cover

many usual effects such as I/O, non-determinism, state, unrecoverable exceptions, etc. Since the rest of the details are technical and specific to F*, we invite readers who want to learn more to look at the artifact (§1.3).

2.4 Verifying IO Programs in F*

Next, we look at how to verify IO programs by defining a new effect in F* based on a Dijkstra monad. The effect defined in §2.4.1 is used in Chapter 3 and it was used to verify a terminating simple web server in that chapter. Statically verifying functional correctness of programs with input-output (IO) is difficult and typically involves talking about *traces* of IO events (i.e., system calls) that occur during a specific run of the program. Ideally, one would hope to be able to leverage a certain level of automation to alleviate the proof burden. In this spectrum, F* is a good candidate in that it makes verification of programs practical thanks to SMT-based automation. We take advantage of this to define a new effect IO in F* that supports verification of trace properties of IO programs. Beyond the definition of such an effect, we strive to make it practical by tuning it to benefit as much as possible from F*'s automation capabilities.

2.4.1 The Dijkstra Monad for Verifying Terminating IO Programs

We seek to write IO computations as terms of a type IO α pre post. The arguments to IO can be summarized as follows: α is the return type of the computation, pre is a precondition over the trace of past IO events (which we sometimes refer to as *history*) that must be satisfied to be able to call the computation, and post is a postcondition over the result and the trace produced by the current computation.

For instance, a program reading a string from a file descriptor fd and returning its length:

```
let file_length (fd : file_descr) : IO nat (requires is_open fd) (ensures p) =
  length (read fd)
```

In this example, we specify that the program returns a natural number while using the IO effect, and we also state that it should only be called on an *open* file descriptor. The precondition is a predicate that must be verified by the history of the program, *i.e.* the list of events that happened before it was run. is_open fd will for instance check that an Open event occurred for fd and that it was not followed by a Close event. The postcondition is a predicate on the same history, but also on the list of newly emitted events and on the final value. For instance, we could take p to state that there exists a string s whose length is the result r and such that the only emitted states that a read was performed on file fd resulting in s:

```
let p = λ h r lt → ∃ s. lt == [ ERead fd s ] ∧ r == length s
```

To define the IO effect, we use the DM4All construction presented in §2.3.2: we first define a computational monad and a specification monad, and then relate them by a monad morphism, which gives us the Dijkstra monad necessary to instantiate F*'s effect mechanism.

[49]: Kiselyov et al. (2015), *Freer monads, more extensible effects*
[58]: Apfelmus (2010), *The Operational Monad Tutorial*
[59]: Kiselyov et al. (2013), *Extensible effects: an alternative to monad transformers*

The Computational Monad. For the computational monad, we use a freer monad [49, 58, 59] that is parametric in the underlying primitive operations. Freer monads are particularly advantageous when it comes to representing IO [9, 60, 61], as its computations have an inherent tree structure. In these trees, each node corresponds to a system call, encapsulating both the operation itself and its arguments, which act as *output* of the program. Furthermore, each operation node has a corresponding child node for each result, which act as *input* for the program. For simplicity, we define operations for file management and for socket communication, but these account only for one possible instantiation of the monad. Our approach can be extended to other IO operations as needed. Moreover, the freer monad could be used to implement other effects such as exceptions, state, and non-determinism by extending the signature of effects [62] (e.g., we use the freer monad in §2.5 to define an effect for state.) The available operations are described by a signature `fsig : Type → Type`, an indexed type where a value of type `fsig b` represents an operation together with its arguments, and where `b` is the type of the operation's result. The freer monad over such a signature is then:[45]

```
type ffree (fsig:Type → Type) (a:Type) =
| Return : a → ffree fsig a
| Call : #b:Type → op:fsig b → k:(b → ffree fsig a) → ffree fsig a
```

As is standard with freer monads, elements of `ffree fsig a` denote computations modeled as trees whose nodes (`Call`) are operation calls and whose leaves (`Return`) contain return values. The type constructor `ffree fsig` forms a monad independently of the signature: its return is `Return`, and its bind is defined by recursion on the tree, grafting the second computation onto the leaves of the first [62].

For `IO`, we give the following signature to the operations:

```
type io_fsig : Type → Type =
| Openfile : string → io_fsig (either file_descr err)
| Read : file_descr → io_fsig (either buffer err)
| Write : file_descr * buffer → io_fsig (either unit err)
| Close : file_descr → io_fsig (either unit err)
| ...
```

The signature we give enables all IO operations to return errors, as their result type is defined to be either a proper result or an error. For example, the `Read` operation has as argument type `file_descr`, and as return type `either buffer err`. The left case of `either` describes a successful read from the file descriptor, while the right case uses the type `err` to signal an error. The computational monad is obtained by instantiating `ffree` with the signature of the operations (`io_fsig`):

```
type io a = ffree io_fsig a
```

The Specification Monad. The `IO` monadic effect uses a specification monad that captures the behavior of a computation as a predicate transformer, i.e., a function that given a postcondition, returns a precondition that is strong enough to guarantee the postcondition after execution of the computation. Our predicates are properties written over a trace of past IO events. These events are defined following the signature of the operations: for each IO operation, we have an event constructor capturing the operation name, argument and result of the operation call:

[62]: Bauer et al. (2015), *Programming with algebraic effects and handlers*

4: For readers familiar with interaction trees [63], a freer monad is essentially the same structure, except that it is defined inductively instead of coinductively and that it lacks the `Tau` constructor for silent steps.

5: F* has prenex universe polymorphism, but no cumulativity, which makes a definition this general awkward to use. In the artifact we adapt the definition depending on the setting.

```
type event = | EOpenfile : a:string → r:(either file_descr err) → event | ...
```

For better SMT automation, we have opted to define events as a variant type, with each operation having its own constructor. This approach proved to be more effective in F* compared to using dependent pairs based on io_fsig. Traces are then represented simply as lists of events. This allows writing a precondition over the entire history of events, while the postcondition is written over the result of the computation and the events produced by it—we refer to it as *local trace*. That the postcondition is only over the local trace of the computation and not over the history plus the local trace makes our specification monad akin to update monads [64], and promotes more local and modular reasoning[6].

[64]: Ahman et al. (2013), *Update Monads: Cointerpreting Directed Containers*

6: An analogous, but slightly simplified predicate transformer monad for IO was also considered by Maillard et al. [9].

We define the specification monad hist that captures transformers from postconditions to preconditions:

```
type trace = list event
type hist a = hist_post:(lt:trace → r:a → Type0) → hist_pre:(h:trace → Type0)
let hist_return (x:α) : hist α = λ post h → post [] x
let hist_bind (w:hist α) (k:α → hist β) : hist β =
  λ post h →
    w (λ lt r → k r (λ lt' r' → post (lt @ lt') r') (rev lt @ h)) h
```

In this representation of specification, we interpret the history of events in preconditions as given in reverse chronological order. This is another factor contributing to successful SMT automation: Consider for instance the is_open predicate that checks whether some file descriptor fd is open according to the history. is_open proceeds by looking at events in the trace to see whether there is some EOpenfile _ (Inl fd) event that was not followed by an EClose fd _ event. When such a predicate is checked, most often the full history is not known, rather it is some abstract value h. Therefore, when one adds more events, the abstract history chunk is found at the end, last_event :: h, which is good because the SMT can reduce is_open fd (last_event :: h) to true if last_event is EOpenfile _ (Inl fd).

Predicate transformers can be naturally organized as continuation-like monads [65]. Indeed, as in the case of ffree, the type hist comes equipped with a monadic structure given by combinators hist_return and hist_bind. In addition, a specification monad also needs to come equipped with an order between computations of the same type. For hist, we define this order as follows:

```
let (⊑) wp1 wp2 = ∀ h post. wp2 post h ⟹ wp1 post h
```

Combinators hist_return and hist_bind are monotone with respect to this order. To form an ordered monad we restrict to only those predicate transformers that are monotonic—i.e., which map stronger postconditions to stronger preconditions—which we enforce by refining type hist with:

```
let hist_wp_monotonic (wp:hist α) =
  ∀ p1 p2. (∀ lt r. p1 lt r ⟹ p2 lt r) ⟹ (∀ h. wp p1 h ⟹ wp p2 h)
```

The Monad Morphism and the Dijkstra Monad. As the third ingredient for constructing a Dijkstra monad, we need a monad morphism θ from the computational monad io to the specification monad hist. For a freer monad, θ can be defined generically for any signature of operations and any specification monad W: it is parametric in an assignment op^W of specifications to the operations, and it goes by recursion on the tree structure of the computation [9].

[9]: Maillard et al. (2019), *Dijkstra Monads for All*

```
let rec θ (op^W:(#b:Type → fsig b → W b)) (m:ffree fsig a) : W a =
  match m with
  | Return v → return^W v
  | Call op k → bind^W (op^W op) (λ x → θ op^W (k x))
```

The return of ffree fsig is mapped to the return of W, and for an operation call the bind of W is used to compose the specification of the operation with the recursive application of θ to the continuation.

For IO, we give semantics to the operations (i.e., op^W) by mapping each operation to its specification. For illustration, Openfile fnm is mapped to openfile^W fnm, defined as follows, and similarly for the other operations:

```
let openfile^W (fnm:string) : hist (either file_descr err) =
  λ post h → ∀ (r:either file_descr err). post [EOpenfile fnm r] r
```

The precondition is trivial, and the postcondition has to hold for any result r the operation may return, with the local trace consisting of the single event corresponding to the operation call.

Partiality. As discussed in §2.3.3, to practically verify programs one has to be able to write computations that are defined only under a pure precondition. We support such partial computations by extending the signature io_fsig with an extra operation [47]:

```
| Require : pre:Type_0 → io_fsig (squash pre)
```

The operation carries a proposition pre, and its result type squash pre is a proof-irrelevant witness of pre, which thus becomes available to the continuation. The assignment op^W maps Require pre to the specification require^W pre, defined as follows:[7]

```
let require^W (pre:Type_0) : hist (squash pre) = λ post h → pre ∧ post [] ()
```

This specification requires pre to hold as a precondition, and passes the empty local trace to the postcondition: Require is a purely logical operation that performs no IO and has no associated event.

7: The () in post [] () is the result the operation passes to the postcondition: since squash pre is proof-irrelevant, its only inhabitant is ().

The IO Effect. Given all the ingredients above, we define the Dijkstra monad using the DM4All construction (§2.3.2):

```
type io_dm (a:Type) (wp:hist a) = c:(io a){θ op^W c ⊑ wp}
```

So far io_dm is just an ordinary type in F*. In order to turn it into the IO effect discussed earlier to support direct-style programming (where monadic returns and binds are implicit), and, more importantly, to benefit from SMT-automation, we use a mechanism for extending F* with new effects [3]. The mechanism expects a monad with a fixed number of indices—for io_dm a single index, the specification—together with its operations: a return, a bind that is polymorphic in the indices, a subsumption combinator, and a branching combinator if_then_else.

[3]: Rastogi et al. (2021), *Programming and Proving with Indexed Effects*

```
effect IOwp (a:Type) (w:hist a) with { repr = io_dm; return = io_dm_return; ... }

effect IO (a:Type) (pre:trace → Type0) (post:trace → a → trace → Type0) =
  IOwp a (λ p h → pre h ∧ (∀ lt r. post h r lt ⟹ p lt r))
```

The last effect, IO, is derived as a pre- and postcondition based effect synonym and is justified by the existence of a embedding from pre- and postcondition pairs to predicate transformers [9]. We mark the effect IOwp (and thus also the derived effects) total, reifiable, and reflectable. Since our monadic representation is based on an inductive type, we indicate to F* that our effect is total. Totality means that any program that type-checks against these effects is guaranteed to terminate—which also means that any program we write in these effects has to be proven to terminate. Reification allows us to reveal the underlying freer-monad-based representation of these effects in proofs, and reflection allows us to reflect the freer monad operation calls to operations in these effects, with the specification of the reflected code determined by θ.

The IO Effect Operations We have introduced the IO effect on top of a freer monad whose operations include Openfile, Read, Write, and Close. However, IO computations do not need to know about these underlying operations. Instead, we provide functions that wrap the IO operations and provide a meaningful specification derived from op^{W}, e.g.,:

```
let read (fd:file_descr) :
  IO (either buffer err)
    (requires (λ h → is_open fd h))
    (ensures  (λ h r lt → lt == [ERead fd r])) =
  IO?.reflect (Call (Read fd) Return)
```

This specification asks the file descriptor to be open in the history as a precondition, and asserts that the computation has as a local trace a single event which corresponds to the performed operation.

2.4.2 Non-termination

Programs such as web servers are expected to run indefinitely, so when verifying them one has to give a meaningful specification to an infinite loop. In this setting, partial correctness specifications, which only conclude about terminating runs, are not enough. Extending the IO effect (§2.4.1) to take into account infinite runs requires us to considerably complicate the specifications so that they can conclude in both terminating and non-terminating cases. Thankfully, in practice, such as in the case of a web server, most parts of the program are in fact terminating and typically only the main loop is non-terminating. We take advantage of this and of F*'s sub-effecting mechanism and work with two effects: the terminating IO effect, and a second effect that also allows non-termination. Terminating programs can then be verified in the simpler terminating setting where automation is more efficient, and they can later be lifted to the more general effect to be used in the main loop.

To support non-termination, we extend the freer monad from §2.4 with an extra constructor corresponding to unbounded iteration[8]:

```
| Iter : #b:Type → #c:Type →
      f:(b → ffree fsig (either b c)) → i:b → k:(c → ffree fsig a) → ffree fsig a
```

As for Call, this constructor carries a continuation k, so that bind can push its own continuation past an Iter node. Unbounded iteration itself is then iter f i = Iter f i Return. The idea is that iter f i first applies the loop body f to i and when this returns Inl j, the loop continues with iter f j. When it returns Inr x the whole loop returns x : a.

To specify iter, we take inspiration from *Dijkstra Monads Forever* [68], which uses interaction trees [63], a coinductive data type in Rocq, to represent potentially infinite programs and their runs. There, iter is defined using the following co-fixed-point:

```
iter f i = match f i with Inl j → tau ; iter f j | Inr x → Return x
```

A silent step—called tau—is needed in order to ensure the co-fixed-point is productive. A program which loops silently is thus an infinite sequence of taus.

We reproduce essentially the same specifications, with two important differences: (i) instead of just proving various program logic rules such as loop invariants, we provide a specification combinator iter^W to compute the specification iter^W w i of iter f i from a specification w of its body f; (ii) we do not attempt to have an unfolding equation for iter and instead only have it for iter^W.

F* does not support coinduction natively, but we can nevertheless easily represent infinite traces of events as sequences indexed by natural numbers: stream a = nat → a. More interestingly, for iter^W we want the equation above to hold at the level of specifications:

```
iterW w i = match w i with Inl j → tauW ; iterW w j | Inr x → returnW x
```

We cannot take this as a definition though, since the recursion is not well-founded, so we instead define iter^W by an impredicative encoding of the corresponding co-fixed-point, from which we then prove the equation above for ≡ (equivalence between two Ws, i.e., ⊑ in both directions). Concretely, we first define the one-step expansion of this equation, which computes the specification of one iteration of the loop, using k for the recursive occurrence:

```
let iter_expand w i (k:β → W α) : W α =
  match w i with | Inl j → tauW ; k j | Inr x → returnW x)
```

We then take iter^W w to be the greatest fixed point of iter_expand w, which we encode impredicatively by quantifying over all specifications that survive one expansion:

```
let iterW w i = λ post h →
  ∃ (it:β → W α). (∀ j. iter_expand w j it ⊑ it j) ∧ it i post h
```

The existential in the definition of iter^W makes verification using this effect impractical because of terrible SMT automation.[9] This remains true even though F* has support for tactics and sub-effecting keeps the non-terminating effect confined to the main loop, so that most of a program is still verified in the terminating IO effect. Because of this, we do not look at non-termination in the rest of this thesis, and it is not part of any of the frameworks we present (Chapter 3, Chapter 4 and Chapter 5). Adding non-termination to the frameworks is left as future work (Chapter 7).

8: This works as a purely syntactic counterpart for a dagger operator ($-^\dagger$) in the spirit of monads with iterative structure [66, 67].

9: SMT solvers are very good at existentials, which they can simply Skolemize. In F*, however, a verification condition is discharged by asking the solver to refute its negation, so existentials end up as ∀s, which are only exploited through incomplete, trigger-based heuristics.

```
val heap : Type0
val mref: a:Type0 → rel:preorder a → Type0
type ref (a:Type0) = mref a (trivial_preorder a)
val contains: #rel:preorder α → heap → mref α rel → Type0
val sel: #rel:preorder α → h:heap → r:mref α rel{h `contains` r} → α
val heap_write: #rel:_ → h:heap → r:mref α rel{h `contains` r} → x:α{sel h r `rel` x} → heap
val heap_alloc: rel:preorder α → heap → α → mref α rel & heap
val addr_of: #rel:preorder α → mref α rel → erased pos
val next_addr: heap → erased pos
let (⪯) h0 h1 =
 ∀ a rel (r:mref a rel). h0 `contains` r ⟹ (h1 `contains` r ∧ (sel h0 r) `rel` (sel h1 r))

val alloc: #rel:preorder α → init:α →
  MST (mref α rel) (requires (λ h0 → ⊤))
                   (ensures  (λ h0 r h1 → h0 ⪯ h1 ∧ fresh r h0 h1 ∧ modifies !{} h0 h1 ∧
                                          addr_of r == next_addr h0))
val read: #rel:_ → r:mref α rel → MST α (requires (λ h0 → h0 `contains` r))
                                        (ensures  (λ h0 v h1 → h0 == h1 ∧ v == sel h1 r))
val write: #rel:_ → r:mref α rel → v:α →
  MST unit (requires (λ h0 → h0 `contains` r ∧ (sel h0 r) `rel` v))
           (ensures  (λ h0 () h1 → h0 ⪯ h1 ∧ modifies !{r} h0 h1 ∧ equal_dom h0 h1))

val witnessed: p:(heap → Type0) → Type0
let stable_heap_predicate = p:(heap → Type0){∀ h0 h1. p h0 ∧ h0 ⪯ h1 ⟹ p h1}

val witness: p:stable_heap_predicate → MST unit (requires (λ h0 → p h0))
                                                (ensures  (λ h0 () h1 → h0 == h1 ∧ witnessed p))
val recall: p:stable_heap_predicate → MST unit (requires (λ _ → witnessed p))
                                               (ensures  (λ h0 () h1 → h0 == h1 ∧ p h1))
```

Figure 2.1: Interface of Monotonic State. The first block gives the model of heaps: the abstract types **heap** and **mref a rel** of references monotonic with respect to the preorder **rel** (with **ref a** the references with a trivial preorder), the operations used to specify the effectful ones—**contains**, **sel**, **heap_write**, **heap_alloc**, **addr_of**, and **next_addr**—and the preorder ⪯ on heaps. The second block gives the operations of the MST effect, **alloc**, **read**, and **write**. The last block gives **stable_heap_predicate**, the type of the heap predicates that are stable with respect to ⪯, together with the state-independent token **witnessed p** and the operations **witness** and **recall** producing and using it.

2.5 Monotonic State Represented Using a Dijkstra Monad

In §2.3, we used state as the running example of how to define a Dijkstra monad, obtaining the ST effect. ST is, however, too simple for practical verification: stable facts about the state—such as a reference remaining allocated—must be threaded manually through every specification. For this reason, Ahman et al. [10] introduced Monotonic State in F*, which underlies large verified developments such as EverCrypt [6]. In this section, we first present the interface of Monotonic State, an F* effect called MST (§2.5.1). We then give the first monadic representation of MST, using a freer monad: in the original work of Ahman et al. [10], the effect was simply assumed, without any definition within F* (§2.5.2). This representation enables us to prove soundness of MST within F* itself (§2.5.3), whereas Ahman et al. [10] gave their proof only on paper. Chapter 4 is built on Monotonic State, and its monadic representation is crucial for the mechanized proofs done there.

2.5.1 The Interface of Monotonic State

Monotonic State is an F⋆ effect characterizing stateful computations whose state evolves according to a given preorder, with additional features to make specifying and verifying such computations more convenient. In particular, given some predicate on states that is stable for any state changes respecting the preorder, one can witness that the predicate $\mathbf{p}$ holds in one part of a program, resulting in a state-independent token $\mathbf{witnessed\ p}$ witnessing this action, and then use this token to recall that the predicate $\mathbf{p}$ still holds at arbitrary later program points and in later states.

Ahman et al. [10] considered a general notion of Monotonic State, in which the effect is parameterized by an arbitrary type of states and a preorder on them. In this thesis, we pick as states monotonic heaps (type **heap**, which comes from F⋆'s standard library), with the preorder $\preceq$ on heaps enforcing that allocated references stay allocated forever, and monotonic references preserve their preorders. With this, we obtain our MST effect, which models first-order[10] ML-style mutable references.

10: Providing a monadic representation that allows storing effectful functions given F⋆'s type theory with predicative, countable hierarchy of universes is an open research problem, further detailed in §2.5.2 and in (§2.6).

The interface of MST, presented in Figure 2.1, hides the concrete representation of the heap and references, and the implementation of the operations that manipulate them. The interface provides proof that the operations preserve the preorder through a series of lemmas (which we elide).

The basic reference type is that of monotonic references, $\mathbf{mref\ a\ rel}$, which have individual preorders (**rel**) enforced by the preorder on the heaps $\preceq$. ML-style references $\mathbf{ref\ a}$ are a special case for the total preorder that relates all the $\mathbf{a}$-values. Behind the interface, the references have concrete addresses which are hidden from the user to ensure their unforgeability. For reasoning, however, the interface exposes a function **addr_of** that returns an **erased**, computationally irrelevant value that can only be used in specifications, but not branched on in programs, ensuring the abstraction boundary is respected.

The MST effect comes with operations for reading from, writing to, and allocating fresh references. The precondition of **write** requires the new value to be related to the value of the reference in the current heap. This ensures that the reference evolves according to the preorder of the reference. The **write** and **alloc** operations both contain a modifies clause in their postconditions, which is the usual way of specifying that only specific references are allowed to be modified (the footprint of the call). Specifically, for a set $\mathbf{s}$ of addresses, modifies $\mathbf{s\ h_0\ h_1}$ holds if any references contained in $\mathbf{h_0}$ and not contained in $\mathbf{s}$ remain unmodified. Above, $\mathbf{!\{r\}}$ is F⋆'s syntax for the singleton set containing the address of $\mathbf{r}$. Notice that the preconditions of **read** and **write** require the reference to be contained, yet for simplicity, we often elide this detail from the other sections.

For users of MST, it is important to note that both the original work of Ahman et al. [10] and our work on MST rely for their soundness on the logical **witnessed** tokens being abstract, unforgeable, and only constructible using the **witness** operation of the MST effect. Therefore, one must avoid using strong abstraction and parametricity breaking classical axioms in code written using the MST effect, such as the axiom of strong excluded middle that equates **prop** and **bool** and allows one to test the validity of arbitrary propositions, because that leads to an inconsistency in F⋆.[11] In the past, the designers of F⋆-based projects were very careful with such axioms, because of their abstraction- and parametricity-breaking nature: they were used only by library designers, and avoided in user code. This is no longer the case in F⋆ versions newer than v2026.03.24, the one the artifacts of this thesis are based on (§1.3): some of these axioms became built in—strong excluded middle, for instance, is now registered as

11: Monotonic State is Incompatible with Strong Excluded Middle: https://github.com/FStarLang/FStar/issues/2814

an implicit coercion from prop to bool (in the Ghost effect), and is thus available by default.[12]

12: Refactor prop: https://github.com/FStarLang/FStar/pull/4212

2.5.2 MST as a Dijkstra Monad with a Free-Monad-Based Representation

In the work of Ahman et al. [10], the MST effect and the operations discussed above were simply assumed in F* (using the assume keyword) and not given any representation or definitions within F*. This was the case because the standard state monad does not support the witness and recall operations: these operations are supposed to produce and use tokens of type witnessed p, but the witnessed type is based on a hybrid modal logic [10], which has no counterpart in F*, so the type can only be axiomatized.

In this thesis, we improve on this situation by giving a freer-monad-based *representation* for MST in F*, from which it can be defined as a Dijkstra monad. The freer monad allows us to axiomatize the operations witness and recall, and separately define a computational state-passing semantics for MST computations, and use that semantics to prove, for the first time, the soundness of the MST effect interface within F* itself (§2.5.3), whereas Ahman et al. [10] gave their soundness proof only on paper because there was no representation of MST computations to work with inside F*.

The axiomatic nature of the MST effect definition is due to the use of a freer monad as its representation, whose elements are inert trees and where no actual computation takes place. As such, the specifications assigned by the monad morphism play the same role as the assumed types of operations in the existing work, but now we also have a representation to work with and prove soundness separately.

To define the MST effect, we use the DM4All construction presented in §2.3.2: we first define a computational monad and a specification monad, and then relate them by a monad morphism, which gives us the Dijkstra monad.

The Computational Monad. We define the computational monad using the freer monad ffree introduced in §2.4.1. We obtain the computational representation of MST by instantiating ffree with the signature of the Monotonic State operations:[13]

13: We omit the require operation needed to implement partiality, as presented in §2.3.3.

```
type mst_ops : Type0 → Type u#1 =
| Read : #b:Type0 → #rel:preorder b → r:mref b rel → mst_ops b
| Write : #b:Type0 → #rel:preorder b → r:mref b rel → v:b → mst_ops unit
| Alloc : #b:Type0 → #rel:preorder b → init:b → mst_ops (mref b rel)
| Witness : p:stable_heap_predicate → mst_ops unit
| Recall : p:stable_heap_predicate → mst_ops unit

type mst_repr (a:Type u#a) : Type u#(max 1 a) = ffree mst_ops a
```

Observe that Witness and Recall have the result type unit. The two operations are more like logical operations than computational ones: they leave the heap unchanged, and at the level of the computational representation no witnessed p tokens are passed to the continuation—the tokens only appear and are passed around in the logical specifications. The Require operation we omitted above is logical in the same way: it leaves the heap unchanged and only passes to the continuation a proof-irrelevant witness of the precondition it carries (§2.3.3).

Let us also discuss why we cannot store higher-order effectful functions. Notice that mst_repr transforms a Type u#a into a Type u#(max 1 a), this is because the operations

Read, Write, and Alloc quantify over Type_0, which places mst_ops in Type u#1. Thus, it increases the universe level, which makes functions represented using this monad unstorable in our references. There is no known monadic representation that allows storing effectful functions given F*'s type theory with predicative, countable hierarchy of universes. We support references that point to other references but cannot point to effectful functions.

The Specification Monad. As the specification monad, we use the monad of weakest preconditions mst_w:

```
let mst_w (a:Type) =
  wp:(post:(a → heap → Type0) → pre:(heap → Type0))
    {∀ post post'.
      (∀ v h1. post v h1 ⟹ post' v h1) ⟹ (∀ h0. wp post h0 ⟹ wp post' h0)}
```

mst_w is almost the specification monad W^{ST} from §2.3.1, with the type of states specialized to heap (and the postcondition curried): a specification transforms any postcondition over return values and final heaps into a precondition over initial heaps. In addition, the refinement type restricts us to *monotonic* predicate transformers, which map stronger postconditions to stronger preconditions. The return of mst_w requires the postcondition to hold of the returned value and the unchanged initial heap, as mst_w_return v post h_0 = post v h_0, while the bind composes two specifications by using the precondition of the second as the postcondition of the first, as mst_w_bind wp wp' post h_0 = wp (λ v h_1 → wp' v post h_1) h_0. Finally, mst_w is an ordered specification monad, with the order $\sqsubseteq$ defined so that wp $\sqsubseteq$ wp' holds when $\forall$ post h_0 . wp' post h_0 $\Longrightarrow$ wp post h_0. Thanks to the monotonicity refinement, mst_w_bind is monotonic (i.e., preserves $\sqsubseteq$) in both of its arguments, as needed by the DM4All construction to define the bind of the Dijkstra monad (§2.3.2).

The Monad Morphism and the Dijkstra Monad. As the final ingredient for defining a Dijkstra monad, we define the monad morphism θ, which assigns weakest-precondition-based specifications to MST operations and computations. We reuse the generic definition of θ from §2.4.1, which is parametric in an assignment op^{W} of specifications to the operations. For illustration, it maps Witness p to $\mathrm{witness}^{\mathrm{W}}$ p and Recall p to $\mathrm{recall}^{\mathrm{W}}$ p, defined as follows, and similarly for the other operations.

```
let witness^W (p:stable_heap_predicate) : mst_w unit =
  λ post h0 → p h0 ∧ (witnessed p ⟹ post () h0)

let recall^W (p:stable_heap_predicate) : mst_w unit =
  λ post h0 → witnessed p ∧ (p h0 ⟹ post () h0)
```

The specification $\mathrm{witness}^{\mathrm{W}}$ matches exactly the specification of witness from §2.5.1 using the correspondence between pre- and postconditions and predicate transformers [9], and the same holds for all the other operations.

The MST Effect. The MST effect is defined similarly to the IO effect (§2.4.1), using the same mechanism for extending F* with new effects [3].

[3]: Rastogi et al. (2021), *Programming and Proving with Indexed Effects*

2.5.3 Soundness of the Free-Monad-Based Dijkstra Monad for Monotonic State

As the original work of Ahman et al. [10] preceded the defining of Dijkstra monads from monad morphisms [9] and the extension of F* with corresponding effects [3], the MST effect interface was simply assumed in F*, with its computational interpretation defined and the soundness of its axiomatization proved only on paper. Also, the proof of Ahman et al. [10] only applied to a λ-calculus much less powerful than full F*. To improve on this, we now use the freer-monad-based representation of MST to define a computational state-passing semantics for MST and use it to give a soundness proof in F* that the specifications computed with θ from the free monad representation are computationally satisfied. A soundness proof is desirable because choosing a specification for the freer monad operations in θ is not necessarily the same as being sound with respect to the intended computational behavior.

At the core of our soundness proof is the idea that the **witnessed** tokens are merely a convenience for communicating logical information from one part of a program to another one, and any program that is provable using **witnessed** tokens ought to also be provable without them. To be able to vary the relevance of **witnessed** tokens, we consider a variant of θ, written θ_w, which is parametric in the tokens $w : (heap \to Type_0) \to Type_0$ used in the specifications of Witness and Recall, but is otherwise defined like θ. We also define a predicate **witnessed_before** that captures which stable heap predicates are recalled in a computation, but were not witnessed in the same computation. In particular, when we can prove **empty** `witnessed_before` **m**, then any **witnessed p** token recalled in **m**:mst_repr α had to originate from an earlier call to **witness p** in **m**, which is a property of closed programs.

```
let rec witnessed_before (preds:set stable_heap_predicate) (m:mst_repr α) : Type0 =
  match m with
  | Return _ → ⊤
  | Call (Read r) k → ...
  | Call (Write r v) k → ...
  | Call (Alloc init) k → ∀ r. preds `witnessed_before` (k r)
  | Call (Witness pred) k → (union preds (singleton pred)) `witnessed_before` (k ())
  | Call (Recall pred) k → pred `mem` preds ∧ preds `witnessed_before` (k ())
```

We define the state-passing semantics **run_mst** for a closed program **m**. We refine **m** as being verified to satisfy a specification **wp**, but we instantiate the **w** parameter of θ_w with the trivial tokens **witnessed_trivial p** = ⊤, modeling that witnessing and recalling are computationally irrelevant. Using refinement types, we require that the weakest precondition **wp post** holds of the initial heap, and we ensure that the postcondition **post** holds of the return value and the final heap. During the recursive traversal of **m**, similarly to the work of [10],the heap is augmented with a set of predicates witnessed during execution and with a refinement type ensuring that these predicates hold for the given heap. The refinement holds as the heap evolves, since the operations that manipulate the heap preserve the preorder which the witnessed predicates are stable for. During the traversal, the refinement with **witnessed_before** ensures that any predicates witnessed are contained in the set of predicates of the augmented heap.

```
type heap_w_preds =
  h:heap & preds:(set stable_heap_predicate){∀ pred. pred `mem` preds ⟹ pred h}
let rec run_mst_with_preds
  (wp:state_wp α)
  (m:mst_repr α{θ_witnessed_trivial m ⊑ wp})
  (post:(α → heap → Type_0))
  (hp_0:heap_w_preds{wp post (fst hp_0) ∧ (snd hp_0) `witnessed_before` m})
: (r:α & hp_1:heap_w_preds{post r (fst hp_1)})
= match m with
  | Return v → (v, hp_0)
  | Call (Alloc #b #rel init) k →
      let (r,h_1) = heap_alloc rel (fst hp_0) init in
      run_mst_with_preds (θ_witnessed_trivial (k r)) (k r) post (h_1,snd hp_0)
  | Call (Read r) k → ...
  | Call (Write r v) k → ...
  | Call (Witness p) k →
      let lp = (snd hp_0) `union` (singleton p) in
      run_mst_with_preds (θ_witnessed_trivial (k ())) (k ()) post (fst hp_0, lp)
  | Call (Recall p) k → run_mst_with_preds (θ_witnessed_trivial (k ())) (k ()) post hp_0

let run_mst
  (wp:state_wp α)
  (m:mst_repr α{θ_witnessed_trivial m ⊑ wp ∧ empty `witnessed_before` m})
  (post:(α → heap → Type_0))
: h_0:heap{wp post h_0} → r:α & h_1:heap{post r h_1}
= λ h_0 → let (r, hp_1) = run_mst_with_preds wp m post (h_0, empty) in (r, fst hp_1)
```

Using run_mst, we define the soundness theorem which states that if a closed program m is well-typed against a specification that uses abstract witnessed tokens (modeled using universal quantification), then for any postcondition post and initial heap h_0, if the weakest precondition wp post holds in h_0, the postcondition holds for the return value v in the final heap h_1.

```
let soundness (m:mst_repr α) (wp:state_wp α)
: Lemma
    (requires (∀ w. θ_w m ⊑ wp) ∧ empty `witnessed_before` m)
    (ensures  ∀ post h_0. wp post h_0 ⟹ let (v,h_1) = run_mst wp m post h_0 in post v h_1)
= ()
```

The soundness proof is not completely axiom-free because it relies for the witnessed tokens on an axiomatic presentation of hybrid modal logic [69] in F*.

2.6 Related Work

The contributions of this chapter—partial Dijkstra monads (§2.3.3), the verification of IO programs (§2.4), and the monadic representation of Monotonic State (§2.5)—are developed in the context of F*. They were necessary for the work presented in the SCIO* (Chapter 3) and SecRef* (Chapter 4) chapters. In the rest of this section, we discuss how they relate to other work.

[9]: Maillard et al. (2019), *Dijkstra Monads for All*
[68]: Silver et al. (2021), *Dijkstra monads forever: termination-sensitive specifications for interaction trees*
[70]: Gladshtein et al. (2026), *Foundational Multi-Modal Program Verifiers*

Dijkstra Monads in Other Proof-Oriented Languages. Dijkstra monads are not specific to F*: they have also been implemented in Rocq [9, 68] and Lean [70]. In Rocq, however, without support for discharging the gathered verification conditions with an

SMT solver, which is a big advantage we get from using F*. In Lean, more recently, Loom [70] was introduced, a verification framework for effectful Lean programs, which like F* is based on Dijkstra monads [9] and has automation via Lean's powerful tactics and via SMT solvers.

[70]: Gladshtein et al. (2026), *Foundational Multi-Modal Program Verifiers*

Verification of IO Programs. We verify computations represented by a freer monad parameterized by the signature of the operations with respect to Hoare-style specifications. In the same style, by using Hoare-style specifications, the FreeSpec framework [60] for Rocq, the Ynot library [71] for Rocq, and Penninckx, Jacobs, and Piessens [72] and Penninckx, Timany, and Jacobs [73] present different ways to verify partial correctness specifications for IO computations—while we verify full correctness specifications. The FreeSpec framework uses a very similar computational monad as ours, but allows choosing the internal state for verification, e.g., the trace in our case. The Ynot library uses, similarly to us, properties that are defined over traces of events. Penninckx et al. present an approach to verifying IO programs using Petri nets and separation logic, implemented in VeriFast [74].

[60]: Letan et al. (2020), *FreeSpec: specifying, verifying, and executing impure computations in Coq*

[71]: Malecha et al. (2011), *Trace-based verification of imperative programs with I/O*

Guéneau et al. [75] and Férée et al. [76] develop a program logic based on characteristic formulae for CakeML source programs with IO and use this framework to verify Unix-style command-line utilities such as cat, sort, grep, diff, and patch, when called on finite input files, which guarantees termination. Åman Pohjola, Rostedt, and Myreen [77] later build on top of this a new program logic for verifying non-terminating CakeML computations with IO by using clocked semantics: evaluation of a program is indexed by a natural number bounding the number of allowed recursive calls. In more recent work, Myreen [78] distils the main ideas into a Hoare logic for diverging IO programs, which he proves sound and complete. The goals of these works seem similar to what we achieve in F*, and Åman Pohjola et al.'s use of a clock seems related to our use of silent steps on the trace. A more precise comparison to these techniques would be interesting.

Interaction trees [79] were used to define a program logic using a monad morphism in the style of Dijkstra monads [68] to verify non-terminating impure computations in Rocq. A case study verifies an HTTP Key-Value Server [80] that is part of the verified operating system CertiKOS [81]. The web server is written in C and the trace properties are verified in Rocq, requiring the manual application of tactics to prove verification goals. More recently, Ioannidis et al. [82] proposed TICL, a temporal logic for verifying effectful, nondeterministic, and possibly non-terminating programs in Rocq, whose structural proof rules follow the syntax of the program instead of the underlying transition system, and which can express liveness properties. In comparison to the application of tactics necessary in Rocq, the IO effect takes advantage of SMT automation in F*.

[79]: Xia et al. (2020), *Interaction trees: representing recursive and impure programs in Coq*

[82]: Ioannidis et al. (2025), *Structural Temporal Logic for Mechanized Program Verification*

Verification of Stateful Monadic Programs. The MST effect is based on Monotonic State [10], which was used extensively in F*, e.g., for the verification of EverCrypt [6]. More recent developments added support for Separation Logic in F*: initial support required extensive usage of tactics [83, 84], and later projects improved on the automation, but departed from the effect mechanism of F* [85].

[83]: Swamy et al. (2020), *SteelCore: An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs*

[84]: Fromherz et al. (2021), *Steel: proof-oriented programming in a dependently typed concurrent separation logic*

[85]: Ebner et al. (2025), *PulseCore: An Impredicative Concurrent Separation Logic for Dependently Typed Programs*

As discussed in §2.5.2, there is no known monadic representation of a *higher-order store* for effectful functions around F*'s type theory with predicative, countable hierarchy of universes. Koronkevich and Bowman [86] propose an acyclic higher-order store, but this would not be sufficient to model OCaml (e.g., no support for Landin's Knot [87]). In a different type theory than that of F*, Frumin, Timany, and

Birkedal [88] adapt Interaction Trees to Guarded Type Theory to support cyclic higher-order stores, and also call/cc, shift and reset [89].

2.7 Summary

This chapter gave a primer of the proof-oriented programming language F*, and presented Dijkstra monads, monads indexed by specifications, which we use to define F* effects: one defines a computational monad and a specification monad, and then relates the two by a monad morphism. Following this recipe, this chapter made three contributions. We extended the Dijkstra monad construction to partial computations, i.e., computations with preconditions (§2.3.3). We defined the IO effect, which supports verification of trace properties of IO programs and is engineered to take advantage of F*'s SMT automation, and we showed how to extend it to non-terminating programs (§2.4). Finally, we gave the first monadic representation of Monotonic State, which enabled us to prove the soundness of the MST effect within F* itself (§2.5). The next two chapters of this thesis build on these effects: SCIO* (Chapter 3) is based on the IO effect, while SecRef* (Chapter 4) is based on the MST effect.

Securing Verified IO Programs against Unverified Code

3

3.1 Contributions

In this chapter, we present how to secure verified IO programs against unverified code in F⋆ with strong machine-checked security guarantees. The contributions of this chapter are:

We introduce SCIO⋆, an F⋆ framework for securely compiling a verified partial IO program and linking it against an IO context with a weak higher-order interface. We bridge the interface gap between the verified program and the untrusted context by using a combination of *higher-order contracts* and a *reference monitor*, which share state that records information about prior IO events. Moreover, SCIO⋆ takes advantage of the program being statically verified and performs no dynamic checks when the program performs IO or when it passes control to the context.

SCIO⋆ is statically verified to add enough dynamic checks to guarantee that the program and the context can interoperate securely. This verification includes the implementation of higher-order contracts, the wrapping of the context's IO library with the monitor's access control checks, as well as their combination. Moreover, we prove in F⋆ the soundness of the entire SCIO⋆ framework, showing that a global trace property holds for the compiled program linked with the untrusted context. This proof is done modularly by typing and also heavily benefits from F⋆'s SMT automation.

Computations in our shallowly embedded source and target languages are represented using a new monadic effect MIO—i.e., Monitored IO—that is based on the IO monad presented in §2.4.1. In addition to usual IO operations (e.g., reading and writing files and network sockets), MIO contains a `get_mstate` operation, which models in a simple abstract way access to the reference monitor state, and which allows us to implement the dynamic checks done by the monitor and higher-order contracts. In addition, at the specification level we distinguish between events produced by the program and those produced by the context, which enables enforcing a stronger specification on the untrusted context.

To model in a simple way that our shallowly embedded contexts cannot directly access the IO operations and the monitor's internal state we propose a novel use of flag-based effect polymorphism. The flag is an index of the MIO monad that controls which operations a computation can access. Flag-based effect polymorphism is necessary since F⋆ gives up on more general kinds of effect polymorphism in order to gain from SMT automation. In addition to this shallow embedding of contexts, we introduce a syntactic representation of target contexts in a small deeply embedded language that can be translated into the shallow embedding.

The security of SCIO⋆ is established by a machine-checked proof in F⋆ that it satisfies Robust Relational Hyperproperty Preservation (RrHP). Intuitively this ensures that SCIO⋆ provides enough protection to the compiled program so that linked target contexts do not enjoy more attack power than a source context would have against the original source program. While proofs of such criteria are generally challenging [8, 11, 90–92], we have carefully set things up in SCIO⋆ so that our proof is simple by construction, in particular because: (1) our languages are shallowly embedded in F⋆; (2) we used flag-based effect polymorphism to model the context; (3) we designed our higher-order contracts mechanism so that we can define both compilation and back-translation in a way that they satisfy a syntactic inversion law that immediately

[8]: Abate et al. (2019), *Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation*
[11]: Patrignani et al. (2019), *Formal Approaches to Secure Compilation: A survey of fully abstract compilation and related work*
[90]: Devriese et al. (2017), *Modular, Fully-abstract Compilation by Approximate Back-translation*
[91]: New et al. (2016), *Fully abstract compilation via universal embedding*
[92]: Jacobs et al. (2022), *Purity of an ST monad: full abstraction by semantically typed back-translation*

implies RrHP (i.e., compiling the program and linking it with the context is syntactically equal to back-translating the context and linking it with the program). We prove RrHP for both the case in which the context is a library used by the program and the case in which the program is a library used by the context.

As a main case study, we statically verify a simple web server in F*, compile it to our target language, and link it against some adversarial request handlers and a non-adversarial request handler serving files. This illustrates that our languages are expressive enough to write interesting code and that the MIO monadic effect enables effective verification.

Outline. We start by illustrating the key ideas of SCIO* on the verified web server case study (§3.2). We then present the MIO effect (§3.3), the implementation of higher-order contracts (§3.4), and the dynamic enforcement of specifications over a monitor state (§3.5). We put these pieces together to define SCIO* and prove it soundly enforces a global safety property and it satisfies RrHP (§3.6). Next, we explain how we execute the web server in OCaml (§3.7) and illustrate a few more examples (§3.8). Finally, we discuss related work (§3.9) and give a summary of the chapter (§3.10).

Artifact. This chapter comes with an artifact in F* that contains a formalization of the contributions above. The artifact contains the SCIO* framework, the mechanized proofs of sound enforcement of a global trace property and of RrHP, the web server, as well as a few other examples. To find the artifact check §1.3, or the original artifact on Zenodo [50] and GitHub [51]. The artifact is 4496 lines of F* code[1] containing 856 top-level definitions, 115 of which are lemmas.

1: Comment and blank lines were not counted towards these numbers.

3.2 SCIO* in Action

```
type req_handler =
  (client:file_descr) →
  (req:buffer{valid_http_request req}) →
  (send:(res:buffer{valid_http_response res} → MIO (either unit err)
                                                 (requires (λ h → did_not_respond h))
                                                 (ensures  (λ _ _ lt → ∃ r. lt = [EWrite _ client r]))) →
  MIO (either unit err) (requires (λ h → did_not_respond h))
                        (ensures  (λ h r lt → (wrote_to client lt ∨ Inr? r) ∧
                                              handler_only_opens_and_reads_files_from_folder lt ∧
                                              web_server_only_writes lt)

let web_server (handler:req_handler) :
    MIO unit (requires (λ h → ⊤))
             (ensures  (λ _ _ lt → every_request_gets_a_response lt)) =
  let s = socket () in setsockopt s SO_REUSEADDR true; bind s "0.0.0.0" 3000; listen s 5; ...
  let client = select_client s in
  let req = get_req client in
  if Inr? (handler client req (write client)) then sendError 400 client;
  close client; ...
```

Figure 3.1: SCIO* case study: web server that takes a request handler as argument. The types are simplified.

We illustrate the key ideas of SCIO* using its main case study as a running example:[2] the partial program is a simple web server that is verified in F* to respond to every request, while the unverified context is a request handler represented as a higher-order function. The web server has the initial control and it gets the request handler as an argument. We carefully crafted the case study to be higher-order and to contain the interesting types of specifications SCIO* supports. The case study is still moderately realistic, since we can run our web server and it can handle HTTP requests from a real browser (§3.7). We also applied SCIO* to a few other examples we discuss in §3.8.

2: The F* syntax used here is recalled in §2.2.

We start with the strong interface of the web server, how it is verified (§3.2.1), and the assumptions that it makes about the handler, which is our adversarial target context (§3.2.2). We then explain how the intermediate interface bridges the gap between the web server and the handler (§3.2.3), before presenting the weak interface of the handler (§3.2.4). We then introduce the executable checks and access control policy the SCIO* requires to enforce the specification (§3.2.5). We use the checks to weaken the assumptions the web server makes about the untrusted handler by using higher-order contracts (§3.2.6). Finally, we strengthen the type of a target handler using reference monitoring to enforce the policy (§3.2.7).

3.2.1 The Verified Partial Program (Web Server)

Figure 3.1 illustrates the most important part of the web server's implementation. It starts by opening a TCP socket (line 15) and then, inside an elided terminating loop,[3] it accepts clients and waits for incoming data in a non-blocking way. The web server waits for requests from multiple clients at the same time. For each client that sends data (line 16), the web server reads it and validates the HTTP request (line 17), and then it passes the request to the handler (line 18). If the handler failed to respond to the request—i.e., it returned an error value (tagged with Inr)—then the web server responds with an error (400) to the client.

3: SCIO* supports recursion, as long as F* can prove termination.

The web server takes as argument a handler of type **req_handler** and its result type (line 14) is that of an MIO computation that returns a unit and has a trivial precondition (indicated by requires) and a postcondition (indicated by ensures). The pre- and the postcondition are part of the type, as indices of the MIO monadic effect. Similarly, the type **req_handler** also contains many interesting refinement types and pre- and postconditions. We highlight the following specifications that appear on the types of the web server and handler:

1. The postcondition of the **web_server** (line 14) ensures that it responds to all accepted clients.
2. The first two arguments of the handler are a file descriptor **client** and a buffer **req** that is guaranteed to contain a valid HTTP request thanks to the refinement type (line 3). The **handler** also has as precondition that no response has been sent yet to the latest request (line 7).
3. The third argument of the **handler** is a callback **send** (lines 4-6) that expects as argument a buffer that contains a valid HTTP response and requires that no response has been sent yet to the latest request—i.e., the same precondition as the **handler**. The concrete callback for **send** is passed by the web server and it simply writes the response to the client.
4. The type of the **handler** ensures that either it wrote to the file descriptor it got as argument or it otherwise returns an error value (tagged with Inr, line 8).
5. The type of the **handler** also ensures that it only opens files from a specific folder, reads only from its own opened files, and closes only them (line 9). It also ensures

that during its execution only the trusted **web_server** can write (so only when the **handler** calls the **send** callback, line 10). Any other IO operation of the **handler** or the **web_server** is not permitted by the postcondition.

These specifications describe how the web server behaves and also what assumptions the web server makes about the request handler. The assumptions about the request handler are necessary to verify that the web server satisfies its postcondition. In particular, without the postcondition of the handler ensuring that it writes to the client when it returns a success value (tagged with Inl), one would not be able to verify that the web server responds to every request. The postcondition of the web server is defined using the following predicate that checks whether every request (read from a file descriptor) is followed at some point by a response (write to the same file descriptor).

```
let every_request_gets_a_response (lt:trace) : Type0 =
  let rec aux = (λ lt read_fds →
    match lt with
    | [] → read_fds == []
    | ERead (fd, _) (Inl _) :: tl → aux tl (fd :: read_fds)
    | EWrite (fd, _) _ :: tl → aux tl (filter (λ fd' → fd ≠ fd') read_fds)
    | _ :: tl → aux tl read_fds) in
  aux lt []
```

The proof that the web server satisfies its postcondition is done mostly automatically with the help of the SMT solver. The implementation is split into several functions, which makes verification more modular by feeding smaller verification conditions to the SMT solver. We needed to state and prove a few lemmas (some using interactive proofs [93]), about the **every_request_gets_a_response** predicate, which had to be proven by induction on the trace, something that the SMT solver cannot do on its own. However, once these lemmas were proven, the SMT solver was able to exploit them to prove the specifications of the various parts of our web server without further manual intervention. This is thanks to the SMT automation F⋆ provides for monadic effects and to our careful design of the MIO monadic effect (§3.3).

[93]: Martínez et al. (2019), *Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms*

3.2.2 The Assumptions about the Context (Handler)

The strong interface of the web server includes the type **req_handler**, which defines the expected specification of the handler (the other components of a strong interface are given in §3.6.1 but they are not yet relevant here). A traditional compiler that just erases specifications[4] would for instance convert **req_handler** into a type without refinement types and without pre- and postconditions:

```
type too_weak_handler_type =
  file_descr → buffer → (buffer → MIO (either unit err)) → MIO (either unit err)
```

However, it would be unsound and insecure to directly link an arbitrary inhabitant of this type to the partial program. The simplest example of an adversarial handler that breaks soundness is the one that immediately returns a success value. This breaks specification 4 of the web server that expects the handler to write to the client at least once when it returns a success value.

```
let adversarial_handler1 client req send = Inl ()
```

To securely compile the web server, it is required to add dynamic checks to enforce the assumptions about the context—i.e., specifications 3, 4 and 5. SCIO⋆ enforces specification 3 and 4 using higher-order contracts and enforces specification 5 using

4: As mentioned in §1.1.1, current extraction mechanisms from proof-oriented languages [5, 94–97] just erase specifications, even the extraction mechanisms that are formally verified to be correct [30, 98, 99].

an access control policy. The postcondition of the handler (4-5) is the most interesting because it is enforced in part by higher-order contracts and in part by reference monitoring. The first part of the postcondition—i.e., checking whether the handler wrote to the client (4)—can be soundly enforced by a contract when the handler returns, but not using reference monitoring at the level of the IO operations. The second part of the postcondition 5 specifies the IO behavior of the handler, which cannot be securely checked by a contract when the handler returns. The postcondition 5 for instance specifies that the handler does not open files outside of a specific folder: if the untrusted handler opens a password file from a different folder we could detect that when it gives back control to the web server; however, this still breaks our postcondition since the bad event has already happened and it is too late to do anything about it. Therefore, we prevent the violation from happening by using reference monitoring—i.e., using an access control policy that enforces the two predicates of specification 5 (more in §3.2.7).

Because the web server is statically verified, we do not have to dynamically enforce anything when the server passes control to the handler—i.e., the precondition of the handler, or that the web server passes a valid HTTP request to the handler (2). This also includes when control *returns* from the web server into the handler, such as when the **send** callback returns after being invoked by the handler—no dynamic enforcement is needed because **send** is verified to satisfy its postcondition.

3.2.3 Bridging the Gap with the Intermediate Interface

SCIO* combines reference monitoring for recording and controlling IO events together with higher-order contracts that have access to the monitor state. Together they bridge the gap between the strong interface of the verified program and the weak interface of the untrusted context—e.g., between the **req_handler** type and something like the **too_weak_handler_type** with erased specifications above. The bridging produces a middle point we call the *intermediate interface* (Figure 1.1), which both the compiled partial program and the monitored target context share.

The meeting point between what our higher-order contracts do not enforce and what the reference monitor enforces is the access control policy. Therefore, an intermediate interface contains types annotated with no specifications except that each function has a postcondition stating that it satisfies the access control policy. We denote this by using the short notation MIO a ⊤ Σ, where ⊤ says that there is no precondition and Σ is the specification of the access control policy encoded as a postcondition. For instance, the intermediate interface version of **req_handler** looks as follows:

```
type intermediate_handler_type =
  file_descr→buffer→(buffer→MIO (either unit err) ⊤ Σ)→MIO (either unit err) ⊤ Σ
```

We define compilation as converting a program with a strong interface into a program with this kind of intermediate interface. Therefore, the compilation of the web server produces a function that expects a request handler of **intermediate_handler_type**. The Σ in this type is presented in §3.2.7—e.g., it prevents the handler from opening files outside a specific folder.

The target context gets an intermediate interface during target linking, when the reference monitor is added, because the monitor enforces that the context satisfies the access control policy. Since monitoring is done at the level of IO operations, to monitor the target context it is enough to link it with a secure IO library that dynamically enforces the access control policy. On the other hand, the compiled

program can call directly the default IO operations that only record the IO events in the monitor state, but performs no dynamic checks.

3.2.4 The Weak Interface of the Unverified Context (Handler)

The context gets its specification from the reference monitor, which enforces the access control policy for the IO operations. Therefore, we make explicit the dependency of the unverified context on the IO library by modeling the unverified context as being *parametric* over the library and also over the abstract access control policy enforced by the library. This definition captures the intuition that "the IO library gives specification to the context" because the abstract specification of the context is given by what IO operations the context uses, and if it uses only operations that enforce an abstract access control policy, then we can show that the context satisfies the abstract policy. So intuitively, the type of the unverified request handler looks more like this in SCIO*:

```
type still_too_weak_handler_type = Σ:erased _ → io_lib Σ → file_descr → buffer →
  (buffer → MIO (either unit err) ⊤ Σ)) → MIO (either unit err) ⊤ Σ
```

where io_lib Σ is the type of the IO libraries that enforce a policy that has abstract specification Σ.

The weak interface clarifies that the abstract specification of the context depends on the specification enforced by the secure IO library. The fact that an abstract specification is still part of the weak interface does not invalidate the intuition that the context is unverified because the weak interface is parametric in this specification. One cannot verify any concrete specification when type-checking the context's code, but it is trivial to show that it inherits the abstract specification of the secure IO library. We also have further evidence that our weak interfaces are expressive enough to represent unverified contexts: (1) the implementation of our handlers, including the larger non-adversarial one that serves files; (2) a syntactic representation of simply-typed target contexts and a translation from any syntactic expression to our shallow embedding (see §3.6.6).

The benefit of defining weak interfaces like this in SCIO* is that after instantiation the monitored context has a specification in its type, which greatly simplifies the soundness proof of our secure compilation chain (§3.6.3).

This type would, however, not prevent the handler from directly calling the IO operations provided by the MIO monadic effect. In proof-oriented languages like Coq one could simply quantify over the monadic effect [9]—i.e., a computation monad indexed by a specification monad (see §3.3 for details)—and use such effect polymorphism to prevent access to IO operations. However, F* doesn't allow quantifying over effects because it needs the concrete implementation of the specification monad to generate verification conditions for the SMT solver. Therefore, we were inspired by the work of Brachthäuser, Schuster, and Ostermann [100] on *parametric effect polymorphism* and combined their idea with the idea of indexing a monadic effect with a flag [3].

[9]: Maillard et al. (2019), *Dijkstra Monads for All*

[100]: Brachthäuser et al. (2020), *Effects as capabilities: effect handlers and lightweight effect polymorphism*

[3]: Rastogi et al. (2021), *Programming and Proving with Indexed Effects*

So, for a start, we added a flag index to the MIO effect. A flag value is simply an inhabitant of a variant type tflag with four constructors: NoOps, GetMStateOps, IOOps and AllOps. By indexing a computation with an element of this type we can restrict the operations that the computation can perform: (1) NoOps means that no operation can be used in the computation, (2) GetMStateOps means that only get_mstate operation can be used, (3) IOOps means that only IO operations (open, read, etc.) can be used, but not get_mstate (4) AllOps means that the computation can access any of the operations. Because the target context is parametric in the flag, it prevents calls to

any of the operations as the flag could take the value NoOps. We refer to this form of effect polymorphism as *flag-based effect polymorphism*. So the actual weak type of an unverified handler looks like this:

```
type weak_handler_type = fl:erased tflag → Σ:erased _ → io_lib Σ fl → file_descr →
  buffer → (buffer → MIO (either unit err) fl ⊤ Σ)) → MIO (either unit err) fl ⊤ Σ
```

where fl is the flag (marked as erased so only usable at the specification level) which is also an index of io_lib. The instantiation of the target context happens during target linking, where the flag- and specification-polymorphic context is passed the AllOps flag, the concrete specification of an access control policy, and the secure IO operations that enforce the concrete access control policy.

The use of flag-based effect polymorphism is key to correctly model the attack capabilities of the source and target contexts, enabling us to prove RrHP (§3.6.4).

3.2.5 Providing the Dynamic Checks and the Access Control Policy

Before discussing how the dynamic checks are enforced, we first note that since some F* specifications are not directly executable, SCIO* requires the dynamic checks to be used inside the higher-order contracts and the access control policy to be enforced by the reference monitor. The F* type checker asks for the necessary dynamic checks, and it also verifies that they indeed imply the specification. Since this verification is done in F* it takes advantage of its support for SMT automation and interactive proofs [93]. SCIO* also leverages type classes in our implementation of higher-order contracts which gives more automation in figuring out the dynamic checks, which works great on simpler cases. While one can make arbitrary assumptions about the context, not all of them have sound dynamic checks that are both *precise enough* and *efficiently enforceable*, thus it becomes a design decision on what assumptions are made and what dynamic checks are picked. For our example, SCIO* requires dynamic checks that imply the precondition of send (3) and the first part of the postcondition of the handler (4), as well as an access control policy that implies the other part (5).

While our specifications use traces, these can be impractical for dynamic enforcement because they grow indefinitely. Instead, SCIO* gives the ability to pick a different state for the monitor as long as it abstracts the trace, and efficient custom checks over this abstract state that don't have to traverse a linear trace. We discuss more about how we do this in §3.5, but until then, in what follows we assume that the monitor stores the trace of events to simplify the technical details.

3.2.6 Compilation Using Higher-Order Contracts

The SCIO* compiler converts a program with a strong interface into a program with an intermediate interface. For example, compilation of the web server wraps it into a new function that takes as argument an intermediately-typed handler. The intermediately-typed handler then is made strongly-typed (of type req_handler) by enforcing higher-order contracts on it. SCIO* implements higher-order contracts by using two dual functions: export converts strongly-typed values into intermediately-typed values, and import converts intermediately-typed values into strongly-typed values. The import and export functions are based on the types of the values and defined in terms of each other, which is needed for supporting higher-order functions [15].

[15]: Findler et al. (2002), *Contracts for higher-order functions*

Import of the intermediately-typed handler wraps it inside of a new function of type req_handler, in which first the strongly-typed arguments are exported. The most

interesting is the export of send: export wraps it into a new intermediately-typed function that enforces the refinement type and send's precondition (3). Not enforcing the refinement or the precondition of send would be unsound, a situation in which F* would not even let us call send, since it would enable the handler to respond to the same client more than once or to call it with an invalid HTTP response:

```
let adversarial_handler2 client req send =
  send (Bytes.of_string "hello") // invalid HTTP response
```

In case the dynamic checks of the refinement and of the precondition succeed, the send function is called and its result is exported and returned, otherwise if one of the checks fails then Inr Contract_failure is returned. This is why the return type of send is either unit err, meaning that it can fail. Thus, when a contract fails (or when the monitor prevents a bad IO operation), an error is returned, which allows the program and the context to handle the errors.[5]

5: We allow for error recovery, since it would be unacceptable for our web server to crash whenever a dynamic check fails. Our secure compilation theorem allows the information flow entailed by error recovery, as discussed in §3.6.4.

After exporting the strongly-typed arguments, the intermediately-typed handler is called with them and an intermediately-typed result is obtained. After importing the result, but before returning it, we have to dynamically enforce the postcondition—i.e., that it wrote to the client. This dynamic check prevents the previously presented adversarial_handler1 attack from working because after being imported, instead of returning Inl (), it returns the error Contract_failure, thus falling into the branch of the web server that checks whether the handler ran into an error and responds with HTTP error 400, ensuring that every request gets a response (1).

Since SCIO* gets a proof that enforcing this dynamic check implies the postcondition 4 (§3.2.5) and since the intermediately-typed handler already satisfies postcondition 5, it is easy for the SMT solver to finish the proof needed to type the handler with the strong type req_handler.

SCIO* uses higher-order contracts not only during compilation but also during back-translation, the dual of compilation. For the secure compilation proofs in §3.6.4, we have to define back-translation from a target context with a weak interface to a source context that is still flag polymorphic, which involved designing our higher-order contracts mechanism so that it can handle flag-based effect polymorphism. For that we had to abstract away the enforcement of the checks from the higher-order contracts and introduced a new notion of *effectful checks* (see §3.4.2).

3.2.7 Enforcing the Access Control Policy by Reference Monitoring

In our example, the assumption that we have to enforce using reference monitoring is postcondition 5 of the handler, stating that it can only open, read, and close its own files, and that only the web server is allowed to write during the execution (when the handler calls send). Our traces are informative so that we can distinguish between the events produced by the partial program (e.g., web server) and those produced by the context (e.g., handler). This is because every event of the trace contains a bit of information of type caller that can have the value Prog or Ctx. This bit allows us to enforce a stronger specification on the untrusted context than on the partial program—e.g., the handler *cannot* directly write to any file descriptor, but the web server can write to the clients.

We use the fact that the monitor enforces an access control policy on the context to give it a specification. However, using the policy directly to give specification would say only half of the story because the policy characterizes only what the context can do—i.e., it would not specify what the partial program does in case the context calls

(back) the partial program. Because of this, the SCIO* framework requires both an access control policy, denoted by Π of type policy, and its specification, denoted by Σ of type policy_spec.

```
type policy_spec = h:trace → caller → op:io_ops → arg:mio_sig.args op → Type0
type policy (Σ:policy_spec) =
  h:trace → op:io_ops → arg:mio_sig.args op → r:bool{r ⟹ Σ h Ctx op arg}
```

The access control policy is enforced *only* on the IO calls of the context, while its specification characterizes the IO calls of the context *and* the IO calls of the partial program during the execution of the context, which is reflected by the extra caller argument above. The refinement on the returned value of the policy makes sure that the policy implies the specification for the events of the context.

Here we give an example of an access control policy Π and its specification Σ that enforces the postcondition of the handler (5). The intuition is that the policy Π allows the handler only to open, read, and close its own files, while the specification Σ says that the trace produced by the handler contains the events allowed by the policy plus the writes done by the send callback.

```
val Σ : policy_spec
let Σ h caller op arg : Type0 =
  match caller, op, arg with
  | Ctx, Openfile, fnm → fnm `in_folder` "/temp"
  | Ctx, Read, fd → is_opened_by_Ctx fd h
  | Ctx, Close, fd → is_opened_by_Ctx fd h
  | Prog, Write, fd → ⊤
  | _, _, _ → ⊥
```

```
val Π : policy Σ
let Π h op arg : bool =
  match op, arg with
  | Openfile, fnm → fnm `in_folder` "/temp"
  | Read, fd → is_opened_by_Ctx fd h
  | Close, fd → is_opened_by_Ctx fd h
  | _, _ → false
```

We can encode such a policy as a postcondition by stating that each event that happened during the execution satisfies Σ. For that we use the following function enforced_locally to encode it as MIO a fl (λ _ → ⊤) (λ h lt → enforced_locally Σ h lt), which we shorten to MIO a fl ⊤ Σ (from §3.2.3).

```
let rec enforced_locally (Σ : policy_spec) (h lt: trace) : Type0 =
  match lt with
  | [] → ⊤
  | e :: t → let (| caller, op, arg, _ |) = destruct_event e in
              Σ h caller op arg ∧ enforced_locally Σ (e::h) t
```

The following three examples of adversarial request handlers all try to violate the access control policy: handler3 tries to open a file outside of the /temp folder, which is the only folder authorized by the contract; handler4 tries to write directly to the client, bypassing the send; handler5 tries to use IO operations outside of those authorized by opening a socket. All of them are prevented from executing by the reference monitor.

```
let adversarial_handler3 sec_io client req send =
  sec_io Openfile ("/etc/passwd",[O_RDWR],0x650)
let adversarial_handler4 sec_io client req send =
  sec_io Write (client,(Bytes.of_string "hello"))
let adversarial_handler5 sec_io client req send = sec_io Socket ()
```

These examples are parametric in the IO library (sec_io) as described in §3.2.4, which was hidden for simplicity in the previous examples. The type of the IO library (io_lib) is defined using a single dependent function that takes as arguments the operation (e.g., Read) and the arguments (e.g., file descriptor) and returns the result of the IO operation (e.g., buffer). The postcondition of the IO library guarantees what we

mentioned earlier, that the reference monitor enforces the access control policy and if an IO operation is prevented from executing the trace does not change.

```
type io_lib (fl:erased tflag) (Σ:policy_spec) = (op : io_ops) → (arg : mio_sig.args op) →
  MIO (mio_sig.res op arg) fl ⊤
    (ensures (λ h r lt →
      enforced_locally Σ h lt ∧
      (match r with
       | Inr Contract_failure → lt == []
       | r' → lt == [convert_call_to_event Ctx op arg r'])))
```

To generate the secure IO library, we wrap the default IO library (denoted by call_io) into new functions that when invoked first enforce the access control policy Π, by using the enforce_policy function below. We enforce the access control policy on each operation by retrieving the monitor state using the get_mstate operation, and checking if the operation is allowed or not. For the secure IO library, the caller is always set to be the context (denoted by Ctx).

```
val enforce_policy : #Σ:policy_spec → Π:policy Σ → io_lib Σ AllOps
let enforce_policy Π op arg =
  if Π (get_mstate ()) op arg then call_io Ctx op arg else (Inr Contract_failure)
```

3.3 The MIO Monadic Effect

In this section we explain how we represent IO programs and their specifications using a new monadic effect that we call MIO (for *Monitored IO*). MIO is a variant of the IO effect presented in §2.4.1: we write computations as terms of type MIO α fl pre post, where, as for IO, α is the return type of the computation, pre is a precondition over the trace of past IO events (the history) that must be satisfied to be able to call the computation, and post is a postcondition over the result and the trace produced by the current computation. MIO differs from IO in three ways: (1) it has an extra operation, get_mstate, that returns the state of the reference monitor, and with which we implement the dynamic checks done by the monitor and by the higher-order contracts; (2) its operations also have a *caller*, marking whether a call is made by the partial program or by the context, which allows us to enforce a stronger specification on the untrusted context (§3.2.7); (3) it has the extra flag index fl, which constrains the operations the computation can use and enables the flag-based effect polymorphism we use to model the context (§3.2.4). We now revisit the ingredients of the DM4All construction (§2.3.2) and explain how each of them changes with respect to IO.

3.3.1 The Dijkstra Monad behind MIO

Recall that the computational monad behind IO §2.4.1 is a freer monad, parametric in a signature of operations (§2.4.1). The first two differences appear at this level: the operations of MIO include an extra operation, GetMState, and every operation call carries a caller. While for IO we described the operations by an indexed type, for MIO we describe them by a type of operation names together with a signature record assigning to each operation the type of its argument and the type of its result. Keeping the operation, its argument, and the caller separate is convenient for this chapter, which often quantifies over them independently.

```
type ops_sig (ops:Type) = { args : ops → Type ; res : (op:ops) → args op → Type }

type mio_ops =
  | Openfile | Read | Write | Close
  | Socket | Setsockopt | Bind | SetNonblock | ...
  | GetMState

val mio_sig : ops_sig mio_ops
```

The first constructors are the IO operations, with the three dots standing for further IO operations needed by the web server of §3.2. Their signatures in `mio_sig` stay as for IO, so all IO operations can return errors: e.g., Read has argument type `file_descr` and result type `either buffer err`. We write `io_ops` for the subtype of `mio_ops` that excludes GetMState (i.e., `type io_ops = x:mio_ops{x =!= GetMState}`), which we use when quantifying over only the IO operations. The new operation, GetMState, has argument type `unit`, and its result type is the type of the monitor state. For simplicity, we assume until §3.5 that GetMState returns the current trace, and there we update it to return an abstract state.

To obtain the computational monad of MIO we turn this into a signature in the style of the freer monad (§2.4). An operation of MIO packs the operation name, its argument and the caller—an inhabitant of the variant type `caller` recording whether the computation making the call is the partial program (Prog) or the context (Ctx):

```
type caller = | Prog | Ctx
type mio_fsig : Type → Type =
  | Op : caller → op:mio_ops → arg:mio_sig.args op → mio_fsig (mio_sig.res op arg)
type m (a:Type) = ffree mio_fsig a
```

The specification monad is `hist`, unchanged from IO, except the events over which traces are defined: since operation calls now carry a caller, the events recording them capture it as well. As before, there is an event constructor for each IO operation, but none for GetMState, since reading the monitor state is not observable IO behavior.

To relate `m` and `hist`, the monad morphism θ is defined by recursion on the computation tree, as for IO, using an assignment `mio_wps` of specifications to operations, which takes the caller, the operation, and its argument. Each IO operation keeps its specification from IO, except that the single event passed to the postcondition now also records the caller. The specification of the new operation is $\text{get\_mstate}^{\text{W}}$:

```
let get_mstate^W : hist trace = λ post h → post [] h
```

It captures the history `h` and passes it to the postcondition as the result of the operation, together with an empty local trace, since GetMState performs no IO. This specification is what later allows us to statically verify the dynamic checks that read the monitor state. Having the three ingredients, we define `mio_dm α wp` using the DM4All construction (§2.3.2).

3.3.2 The Flag Index and the `MIO` Effect

The Dijkstra monad `mio_dm` accounts for GetMState and the caller, but not yet for the flag. Recall from §3.2.4 that a flag is an inhabitant of the variant type `tflag` (NoOps, GetMStateOps, IOOps, or AllOps) that describes which operations a computation is allowed to perform. We add the flag as an extra index by further refining `mio_dm` with the predicate `satisfies : m α → tflag → bool`, which decides whether the underlying

computation uses only the operations allowed by the flag: e.g., a computation satisfies IOOps if it contains no call to GetMState, it satisfies NoOps if it contains no call at all, and any computation satisfies AllOps. The representation of the MIO monadic effect is then:

```
type mio (a:Type) (fl:erased tflag) (wp:hist a) = c:(mio_dm a wp){c `satisfies` fl}
```

The flag is **erased** since it plays a role only during verification and has no computational relevance. Given all the ingredients above, we define the MIO effect from **mio** the same way we defined IO, using F*'s effect mechanism [3], which gives us direct-style programming and SMT-backed verification.

erased marks a value that exists only during verification.

[3]: Rastogi et al. (2021), *Programming and Proving with Indexed Effects*

3.3.3 Wrappers for the **MIO** Operations

Computations in MIO do not use the underlying constructors directly. Instead, we provide a wrapper for the IO operations, **call_io**, with a meaningful specification derived from **mio_wps**.

```
let call_io (c:caller) (op:io_ops) (arg:mio_sig.args op) :
  MIO (mio_sig.res op arg) IOOps
    (requires (λ h → T))
    (ensures  (λ h r lt → lt == [convert_call_to_event c op arg r])) =
  MIO?.reflect (Call (Op c op arg) Return)
```

The specification asserts that the computation has as local trace the single event corresponding to the performed operation, and the flag IOOps indicates that the computation performs only IO operations. For writing code, we define synonyms of **call_io** for each IO operation, such as **openfile**, **read**, and **close**, which take the caller as their first argument. Similarly, we define **get_mstate**, a wrapper for GetMState that takes no arguments and returns a **trace** (for now, we change this in §3.5). Its flag is GetMStateOps, and its specification ensures that the result is the history (**r == h**) and that the local trace is empty (**lt == []**). Since GetMState produces no event, the caller passed to Call is irrelevant, and we simply set it to Prog:

```
let get_mstate () :
  MIO trace GetMStateOps
    (requires (λ h → T))
    (ensures  (λ h r lt → r == h ∧ lt == [])) =
  MIO?.reflect (Call (Op Prog GetMState ()) Return)
```

3.4 Higher-Order Contracts

We gave a brief presentation of how higher-order contracts work in §3.2.6, and now we give a more detailed picture of how the **import** and **export** functions are implemented. The two functions are designed to be used during the compilation of the partial program and during the back-translation of the context, which are explained later in §3.6, so the design was influenced by the need to work with flag-based effect polymorphism and by the need to define back-translation. As we said, **import** and **export** are two dual functions that convert between strongly-typed values and intermediately-typed values. Because the **import** and **export** functions are implemented in F* and the conversion happens between F* types directly, the correctness of the type conversion is verified by F* typing.

```
class interm (ityp:Type) (fl:tflag) (Σ:policy_spec) = {}
instance interm_unit fl Σ : interm unit fl Σ = {}
instance interm_file_descr fl Σ : interm file_descr fl Σ = {}
...
instance interm_err fl Σ : interm err fl Σ = {}
instance interm_pair fl Σ t1 {| interm t1 fl Σ |} t2 {| interm t2 fl Σ |} : interm (t1 * t2) fl Σ = {}
instance interm_either fl Σ t1 {| interm t1 fl Σ |} t2 {| interm t2 fl Σ |} : interm (either t1 t2) fl Σ = {}
instance interm_arrow fl Σ t1 {| interm t1 fl Σ |} t2 {| interm t2 fl Σ |} : interm (t1 → MIO t2 fl ⊤ Σ) fl Σ = {}
```

Figure 3.2: The type class representing intermediate types.

3.4.1 Intermediate Types

To begin, we define what we mean by strong types and intermediate types. We define *intermediate types* as a subset of F* types, by using the trivial interm type class from Figure 3.2. The interm instances represent the types of the target language after the reference monitor was added to give them a specification by enforcing the access control policy (§3.2.7). We provide instances for every base (unrefined) type, as well as pairs, sums, options, and non-dependent functions. The interm type class is indexed by an operations flag (fl, introduced in §3.2.4) and the specification of an access control policy (Σ). These indices ensure that higher-order functions are invariant over the flag fl and the policy Σ. This is realized by only defining instances that respect this property. We need intermediately-typed higher-order functions to be invariant over the flag and the policy so that they are aligned with the weakly-typed higher-order functions which are also invariant over those.

We do not give a definition for *strong* types. We consider strong types the types from which we can export from and we can import into. This definition is open because we implement import and export using type classes. We define instances for intermediate types, base refined types, pairs, sums, options, and functions with pre- and postconditions.

3.4.2 Exportable and Importable Type Classes

The import and export functions are fields of two type classes named importable_to and exportable_from. The classes are indexed by the *strong* type styp and they have as a field the *intermediate* type ityp. Function export is defined as taking a value of the strong type styp and returning a value of the intermediate type ityp, and function import is defined as taking a value of the intermediate type ityp and returning either a value of the strong type styp or an error in case that the dynamic check failed. The two type classes are additionally indexed by the operations flag fl and the specification Σ; these extra indices are required by the interm_ityp constraint on ityp to be an intermediate type.

```
class exportable_from (styp : Type) (fl : erased tflag) (Σ : policy_spec) (cks : checks) = {
  ityp : Type;
  interm_ityp : interm ityp fl Σ;
  export : eff_checks fl cks → styp → ityp; }

class importable_to (styp : Type) (fl : erased tflag) (Σ : policy_spec) (cks : checks) = {
  ityp : Type;
  interm_ityp : interm ityp fl Σ;
  import : ityp → eff_checks fl cks → either styp err; }
```

One special case is functions that are flag-based effect polymorphic. We want the wrapping of an effect polymorphic function to result also in an effect polymorphic function. This is necessary later in §3.6.4 when we define back-translation from a target context to source context that are both flag-based effect polymorphic. Our contracts dynamically enforce pre- and postconditions about IO behavior, for which we have to use the effectful **get_mstate** operation—however, we *cannot* use it directly without losing the effect polymorphism. Our solution is to abstract away how the enforcement of the checks is done by parameterizing the **import** and **export** functions over, what we call, *effectful checks*. The effectful checks are indexed by the same operations flag **fl** as for the wrapped function, so they can be used inside the contract without affecting the effect polymorphism. Effectful checks are obtained by merging the **get_mstate** operation with the checks. The compilation framework automatically converts the checks into effectful checks and passes them when calling the **import** or the **export** functions. This approach forces us to index the type classes with the collection of checks that have to be enforced.

The **cks** collection contains dynamic checks to enforce the pre- and postconditions of the strong type **styp**. However, it does not include dynamic checks to enforce refinement types because we do not need to use the **get_mstate** operation to enforce them. To learn about how we convert the refinement types into dynamic checks, we refer the reader to the work of Tanter and Tabareau [101] because we use the mechanism they proposed, and most refinement types could anyway also be easily converted to pre- and the postconditions.

[101]: Tanter et al. (2015), *Gradual certified programming in Coq*

3.4.3 Exporting and Importing Functions

The situation is most interesting for functions, where exporting and importing make use of each other. The basic idea is that in order to export a function **f**, we create a intermediately-typed function that imports its argument, calls **f**, and exports the result back to an intermediate type [15]. Essentially, the composition **export** · **f** · **import**. Dually, an intermediately-typed function **g** is imported roughly as **import** · **g** · **export**. However, as **import** can fail (it returns a sum type), we also require the codomain of each imported and exported function to "include" errors. Below is the instance for exporting simple functions, i.e., of type $\mathbf{t}_1 \to$ MIO **fl** (**either** $\mathbf{t}_2$ **err**) T Σ without other pre- and postconditions:

[15]: Findler et al. (2002), *Contracts for higher-order functions*

```
instance exportable_smpl_arr (t1 t2:Type) (fl:erased tflag) (Σ:policy_spec)
  (cks:checks{EmptyNode? cks})
  {| d1 : importable_to t1 fl Σ (left cks) |}
  {| d2 : exportable_from t2 fl Σ (right cks) |}
: exportable_from (t1 → MIO (either t2 err) fl ⊤ Σ) Σ fl cks
= { ityp = d1.ityp → MIO (either d2.ityp err) fl ⊤ Σ;
    interm_ityp = solve;
    export = (λ cks (f:(t1 → MIO (either t2 err) fl ⊤ Σ)) (x:d1.ityp) →
      match x' <- d1.import x (left cks); f x' with      (** do-notation **)
      | Inr err → Inr err
      | Inl y' → Inl (d2.export (right cks) y')) }
```

The function is exported to the type $i_1 \rightarrow$ MIO fl (either i_2 err) ⊤ Σ, where i_1,i_2 are the intermediate types corresponding to the strong types t_1,t_2 according to their instances. F* automatically checks that the type of the function is intermediate too (using the interm_arrow instance from Figure 3.2). Exporting produces a function that imports its argument, applies f, and exports the result with Inl. If either importing or the function itself fail (returning Inr), the error is returned instead.

We see here that the cks collection has the structure of a binary tree, defined using the inductive type checks. Since this function does not have a pre-/postcondition to enforce, the root node must be an EmptyNode, and its left and right children are used to import t_1 and export t_2. For a function with pre- and postconditions, of type a → MIO b pre post, the tree takes the shape Node ck left right, where ck is a dynamic check testing whether the postcondition post indeed holds, and left and right are sub-trees for a and b respectively. For structured, non-arrow types such as tuples or sums, we also use an EmptyNode left right node, as there is no immediate pre- and postcondition to enforce, but there may be some deeper within the type. The Leaf node is used for base types when there are no checks to perform.

When importing or exporting a function with pre- and postconditions, the cks collection becomes important. The cks collection must contain a boolean predicate for each pre- and postcondition (that needs enforcement). Hence, the instances of importable_to and exportable_from require a cks collection and alongside a proof that each check implies the actual (propositional) pre- or postcondition. Crucially, the dynamic checks have the same structure as the postcondition so that they can also distinguish between the history and the local trace of a computation, and they can also distinguish the events of the partial program from the events of the context. Since the types are higher-order, the trace of a computation can contain events generated by both the partial program and the context. Therefore, we give the following type to dynamic checks:

```
type ck_typ (t1 t2 : Type) = t1 → h:trace → t2 → lt:trace → bool
```

Given the type of the argument and return value of a function, a dynamic check is a boolean predicate over the value of the argument, the history before the call was made, the return value, and the local trace (i.e., the events generated during the activation of the function).

To enforce the dynamic checks, the compiler automatically generates the corresponding effectful checks with the same tree structure. The effectful checks are passed as an argument to the import and export functions. An effectful check guarantees that it enforces the corresponding dynamic check. An effectful check is done in two steps: a "setup" and an actual check. First, for the setup, we must mark the point in history where the function was initially called, in order to able to determine its local trace when it returns. This is accomplished by the effectful check because after the first

call when it captures the initial trace, it returns a second function that enforces the dynamic check.

Enforcing the Postcondition When Importing. We are now ready to import a function with an intermediate type into one with a strong type that has a pre- and a postcondition. The precondition does not need to be dynamically checked, because one can always strengthen the precondition of a function using subtyping, thus we only have to enforce the postcondition.

Here we show just a combinator that is used in an importable_to instance that focuses only on enforcing the postcondition. The combinator takes an intermediately-typed function f, a dynamic check ck, and its effectful counterpart eff_ck, and returns a strongly-typed variant of f. The combinator first "sets up" the effectful check, obtaining the do_ck, then it calls f, and then it enforces the postcondition by running do_ck, returning an error if the check fails.

```
let enforce_post (t1 t2 : Type) (fl:erased tflag) (Σ:policy_spec)
  (ck : ck_typ t1 (either t2 err))
  (eff_ck : eff_ck_typ fl ck)
  (pre : t1 → trace → Type0)
  (post : t1 → trace → either t2 err → trace → Type0)
  (c1_post : squash (∀ x h r lt. pre x h ∧ enforced_locally Σ h lt ∧ ck x h r lt ⟹ post x h r lt))
  (c2_post : squash (∀ x h lt. pre x h ∧ enforced_locally Σ h lt ⟹ post x h (Inr Contract_failure) lt))
  (f : t1 → MIO (either t2 err) fl ⊤ Σ)
: (x:t1) → MIO (either t2 err) fl (pre x) (post x)
= λ x →
    let do_ck = eff_ck x in
    let r : either t2 err = f x in
    if do_ck r then r else Inr Contract_failure
```

To import a function, two constraints have to hold between the pre- and postcondition, the dynamic check, and the access control policy, constraints that ensure that the postcondition can be soundly enforced. The first constraint, c1_post, ensures that the policy together with the dynamic check enforce the postcondition. This constraint allows us to return the result when the check succeeds by giving us that the postcondition holds. The second constraint, c2_post, ensures that the postcondition can be enforced even if the dynamic check fails as long as the policy was enforced. This constraint allows us to return Inr Contract_failure when the check fails after we called the function.

Enforcing the Precondition When Exporting. Exporting a strongly-typed function implies converting its precondition into a dynamic check and involves a mechanism similar to the one shown above for postconditions. The main difference is that checking preconditions does not require a setup-check split, and can be done at once. To denote a dynamic check for a precondition, we reuse the ck_typ type above, using unit for the codomain and providing an empty local trace to the checks. We show below the enforce_pre combinator that is used in instances of exportable_from:

```
let enforce_pre t1 t2 fl Σ
  (ck : ck_typ t1 unit) (eff_ck : eff_ck_typ fl ck)
  (pre : t1 → trace → Type0)
  (post : t1 → trace → either t2 err → trace → Type0)
  (c_pre : squash (∀ x h. ck x h () [] ⟹ pre h))
  (c_post : squash (∀ h lt r. pre h ∧ post h r lt ⟹ enforced_locally Σ h lt))
  (f : (x:t1 → MIO (either t2 err) fl (pre x) (post x)))
: (x:t1 → MIO (either t2 err) fl ⊤ Σ)
= λ (x:t1) →  if eff_ck x () then f x else Inr Contract_failure
```

To export a function, also two constraints have to hold. The first constraint, **c_pre**, ensures that the dynamic check enforces the precondition. This constraint gives us that the precondition holds after running the effectful check, allowing us to call the function. The second constraint, **c_post**, ensures that we can weaken the postcondition to the access control policy by subtyping.

Effectful checks are the key technical novelty to define back-translation between a target and a source context that are both flag-based effect polymorphic (§3.6.4).

3.5 Dynamic Enforcement over a Monitor State

So far, we have described our specifications and dynamic checks over IO traces. While traces are intuitive and very expressive, making them a fine choice for specifications, they are inadequate for an implementation as their size may grow without bound. We would rather store only the information that is *needed* to implement the relevant dynamic checks. For example, if all dynamic checks are about whether file descriptors are open or not, the monitor could keep a set of currently open file descriptors and efficiently probe this set instead of inspecting a trace. Importantly, independent of this runtime aspect we wish to still write specifications over traces, to retain full generality and a single *lingua franca*. SCIO* allows for such setups with full flexibility, allowing the programmer to choose exactly what the runtime monitor state should be and how the checks are performed, while still guaranteeing correctness over traces. We explain the mechanism in the rest of this section.

3.5.1 The Monitor State

The entire SCIO* development is in fact parameterized over a type of *monitor state*, which represents the runtime information needed to implement the dynamic checks. Most of the definitions shown previously actually have an extra argument of type **mstate**. We elided them for simplicity of the previous sections, but will make them explicit here.

```
type mstate = { typ : Type; abstracts : typ → trace → Type0 }
```

The **mstate.typ** type must be instantiated by the programmer, who chooses it according to the dynamic checks that the whole program must perform. For instance, the type **list file_descr** would be a sensible choice if all one needs to check is whether file descriptors are open. The monitor state is exactly **mstate.typ**. Program traces are never materialized: they only appear in specifications.

The programmer must also relate the monitoring state to the trace of the program by defining a predicate **abstracts**. This relation specifies how the monitor state abstractly models the trace, and must be kept invariably true when applied to the current

monitoring state and the current (ghost, immaterial) trace. For open file descriptors, one could use (λ fds h → ∀ fd. fd `mem` fds ⟺ is_open fd h), stating that a descriptor fd is in the list if and only if it is open according to the current trace h.

So in fact, the get_mstate operation does not return a trace, but a monitor state. Its postcondition also guarantees that the returned state abstracts the current trace (which is still accessible in specifications). The implementation of mio_wps is adjusted similarly.

```
let get_mstate (#mst:mstate) () :
  MIO mst.typ GetMStateOps
    (requires (λ h → ⊤))
    (ensures  (λ h s lt → s `mst.abstracts` h ∧ lt == [])) =
  MIO?.reflect (Call Prog GetMState () Return)
```

The recording of IO operations in the monitor state is abstracted away in SCIO⋆. The lower-level implementation of the state updates is explained in §3.7, where we discuss execution in OCaml.

3.5.2 Enforcing the Access Control Policy and the Checks over the Monitor State

The access control policy and higher-order contracts must also work over monitor states. A policy, which must soundly enforce a higher-level policy_spec over traces, now takes a monitor state as argument, instead of a trace. The policy must ensure that, when it returns true, the policy specification is satisfied for *any* trace that is abstracted by the current monitor state.

```
type policy (mst:mstate) (Σ:policy_spec) =
  s0:mst.typ → op:io_ops → arg:io_sig.args op →
  r:bool{r ⟹ (∀ h. s0 `mst.abstracts` h ⟹ Σ h Ctx op arg)}
```

Coming back to the open files example, this means that a policy can only inspect the set of files to decide whether the operation should be allowed, and the answer must be right for any trace that would lead to the current open file set.

As for the dynamic checks (ck_typ), they are defined over an initial and final state of the function:

```
type ck_typ (mst:mstate) (t1 t2 : Type) = t1 → s0:mst.typ → t2 → s1:mst.typ → bool
```

Relating these states to the trace is done in the preconditions of the import and export functions from §3.4.3. One is the first constraint required when exporting a function (c_pre from paragraph 3.4.3) and the second one is the first constraint required when importing a function (c1_post from paragraph 3.4.3):

```
(c_pre : squash (∀ x s0 s1 h. s0 `mst.abstracts` h ∧ s1 `mst.abstracts` h ∧
     ck x s0 () s1 ⟹ pre x h))
(c1_post : squash (∀ s0 s1 x h r lt . pre x h ∧ enforced_locally Σ h lt ∧
     s0 `mst.abstracts` h ∧ s1 `mst.abstracts` (rev lt @ h) ∧ ck x s0 r s1 ⟹ post x h r lt))
```

As explained in §3.4.2, we have to abstract away the enforcement of the checks by using effectful checks. They are represented by the type eff_ck_typ below. Enforcing a dynamic check ck is done in two steps: a "setup" and an actual check. The setup is done by calling the effectful check, which captures the current state s_0 and returns it together with the second part of the effectful check. The returned state is labeled with erased, thus it cannot be used in the computation: it is only there in order to specify

the effectful check. The second part, of type **eff_ck_typ_cont**, is called to enforce the dynamic check **ck** and guarantees that the property holds: (b ⟺ ck x s_0 y s_1). This function also returns an erased state, again only for specification purposes. This gives us the ability to enforce the dynamic check on two different states—e.g., capture s_0 before calling a function and s_1 after the function returns—and this enables us to enforce postconditions. Inside the **enforce_pre** and **enforce_post** combinators, the specifications of the effectful check that s_0 and s_1 abstract the histories, plus the constraints, make it very easy to prove that the enforcement is done correctly.

```
type eff_ck_typ_cont (mst:mstate) (fl:erased tflag) (t1 t2:Type)
  (ck:ck_typ mst t1 t2) (x:t1) (s0:mst.typ) =
  y:t2 → MIO (erased mst.typ * bool) mst fl ⊤
    (ensures (λ h1 (s1, b) lt → lt == [] ∧ s1 `mst.abstracts` h1 ∧ (b ⟺ ck x s0 y s1)))

type eff_ck_typ (mst:mstate) (fl:erased tflag) (t1 t2:Type)
  (ck:ck_typ mst t1 t2) =
  x:t1 →
  MIO (s0:erased mst.typ & eff_ck_typ_cont mst fl t1 t2 ck x s0) mst fl ⊤
    (ensures (λ h0 (| s0, _ |) lt → s0 `mst.abstracts` h0 ∧ lt == []))
```

Converting a pure dynamic check into an effectful check is straightforward:

```
let make_check_eff mst (t1 t2:Type) (ck:ck_typ mst t1 t2)
: eff_ck_typ mst GetMStateOps ck
= λ (x:t1) →
    let s0 = get_mstate () in
    let cont = (λ (y:t2) →
      let s1 = get_mstate () in
      (hide s1, ck x s0 y s1)) in
    (| hide s0, cont |)
```

3.5.3 The Web Server's Monitor State

For our case study, we choose the following implementation of the monitor state: (a) a list of file descriptors that were opened by the context, (b) a single boolean flag that determines whether the current request was answered, and (c) a list of the file descriptors that were written to. We do not need to track *all* of the open files, but only those opened by the context. This state contains enough information to enforce the access control policy (5), the precondition of **send** (3), and the first part of the postcondition of the handler (4), and in **abstracts** we can see a conjunction of all of these.

```
let myst : mstate = {
  typ = { ctx_opened : list file_descr; responded : bool; written : list file_descr };
  abstracts = (λ s h →
    (∀ fd. fd `mem` s.ctx_opened ⟺ is_opened_by_Ctx fd h) ∧
    (responded ⟺ ¬(did_not_respond h)) ∧
    (∀ fd. fd `mem` s.written ⟺ wrote_to fd h)); }
```

The access control policy is already defined in §3.2.7 and to use it with our chosen state we have to replace the **is_opened_by_Ctx fd h** with **fd `mem` s.ctx_opened**. Below are the dynamic checks required to import the handler, defined as a **cks** collection of type **checks** (§3.4.3). The first node is for the postcondition of the handler (4) and the next one is for the precondition of **send** (3).

```
let handler_cks : checks myst =
  Node (| file_descr, unit,
      (λ client s0 _ s1 → not (client `mem` s0.written) && client `mem` s1.written) |)
    (Node (| Bytes.bytes, unit,
        (λ res s0 _ _ → not (s0.responded) && valid_http_response res) |)
      Leaf Leaf)
    Leaf
```

When instantiating the importable_to type class using these checks, F⋆ automatically proves that three of the four constraints hold, asking for help only to prove that the first dynamic check implies the postcondition of the handler. Moreover, SCIO⋆ guarantees that no other check is necessary.

3.6 SCIO⋆: Formally Secure Compilation Framework

We now put all the pieces together and define the SCIO⋆ secure compilation framework (i.e., languages, compiler, and linker in §3.6.1), give semantics to the source and target languages (§3.6.2), and describe our machine-checked proofs that SCIO⋆ soundly enforces a global trace property (§3.6.3) and satisfies by construction a strong secure compilation criterion (§3.6.4). We present this in the setting where the partial program has initial control and the context is the library, then also describe the dual setting when the context has initial control and the program is the library (§3.6.5).

3.6.1 Secure Compilation Framework

We first use the definitions from the previous sections to formally define our source and target languages, compiler, and linker—to which we jointly refer as the SCIO⋆ secure compilation framework. We define our languages using shallow embeddings, which is a common way to express DSLs in F⋆ [5, 84, 102], and which also simplifies our proofs. We represent the partial program as a function (since in F⋆ modules are not first-class objects and one cannot reason about them). We now describe the setting where the partial program has the initial control, thus we make the partial program get the context as argument (after the context is instantiated with the flag, etc)—the setting in which the context starts is outlined in §3.6.5.

[5]: Protzenko et al. (2017), *Verified Low-Level Programming Embedded in F**
[84]: Fromherz et al. (2021), *Steel: proof-oriented programming in a dependently typed concurrent separation logic*
[102]: Bhargavan et al. (2021), *DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code*

The SCIO⋆ framework is listed in Figure 3.3 and explained step by step below. As explained before, the partial program and the context share a higher-order interface. The difference between the source and the target language is that in the source language, the partial program and the context share a *source* interface, while in the target language they share a *target* interface. The source interface (type $\mathbf{interface}^S$) is a dependent record type that contains the type of the context (denoted by ctype), the access control policy Π and its specification Σ (both explained in §3.2.7), and the dynamic checks, cks, that are enforced by the higher-order contracts (explained in §3.4). The interface also contains a type class constraint importable_ctype ensuring that the guarantees the type of the context (ctype) provides to the program can be enforced using the policy Π and the dynamic checks cks. The importable_ctype constraint also ensures that ctype is flag-polymorphic.

The target interface (type $\mathbf{interface}^T$) is another record type that contains the type of the context (ctype), but this time, the field weak_ctype requires ctype to be a weak type (explained in §3.2.4). The interface also includes the policy Π and its specification Σ, which are used during target linking.

```
type interface^S = {
  ctype : erased tflag → Type;
  Σ : policy_spec; Π : policy Σ;
  cks : checks;
  importable_ctype : fl:erased tflag → importable_to (ctype fl) fl Σ cks;
  ψ : post_cond; }
type interface^T  =  {
  ctype : erased tflag → policy_spec → Type;
  Σ : policy_spec; Π : policy Σ;
  weak_ctype : fl:erased tflag → interm (ctype fl Σ) fl Σ; }
let compile_interface (I^S:interface^S) : interface^T = { (** denoted by I^S↓ **)
  Σ = I^S.Σ; Π = I^S.Π;
  ctype = (λ fl _ → (I^S.importable_ctype fl).ityp);
  weak_ctype = (λ fl → (I^S.importable_ctype fl).interm_ityp); }

type prog^S I^S = fl:erased tflag → I^S.ctype (fl+IOOps) →
  unit → MIO int (fl+IOOps) ⊤ I^S.ψ
type ctx^S I^S = fl:erased tflag → io_lib fl I^S.Σ → eff_checks I^S.cks →
  I^S.ctype fl
type whole^S = ψ : post_cond & (unit → MIO int AllOps ⊤ ψ)
let link^S #I^S (P:prog^S I^S) (C:ctx^S I^S) = (** denoted by C[P] **)
  (| I^S.ψ, P AllOps (C AllOps (enforce_policy call_io I^S.Π) (make_checks_eff I^S.cks)) |)

type prog^T I^T = I^T.ctype AllOps I^T.Σ → unit → MIO int AllOps ⊤ ⊤
type ctx^T I^T = fl:erased tflag → #Σ':erased policy_spec → io_lib fl Σ' → I^T.ctype fl Σ'
type whole^T = unit → MIO int AllOps ⊤ ⊤
let link^T #I^T (P:prog^T I^T) (C:ctx^T I^T) = (** denoted by C[P] **)
  P (C AllOps (enforce_policy call_io I^T.Π))

let compile_prog #I^S (P:prog^S I^S) : prog^T (I^S↓ ) = (** denoted by P↓ **)
  λ (C : (I^S↓ ).ctype AllOps I^S.Σ) → P AllOps (import C (make_checks_eff I^S.cks))
let back_translate_ctx #I^S (C:ctx^T I^S↓ ) : ctx^S I^S = (** denoted by C^T↑  **)
  λ fl sec_io → import (C fl sec_io)
```

Figure 3.3: The SCIO★ secure compilation framework. Idealized mathematical notation.

The type of the partial source program (**prog**^{S}) and of the target context (**ctx**^{T}) can also be found in Figure 3.3. The partial source program is a computation in the MIO monadic effect that can call IO operations (**fl**+IOOps), but that, since it is statically verified, has no need to call the **get_mstate** operation, so it is otherwise parametric in the flag. The type of the target context uses **ctype** from the target interface $\mathbf{I}^T$ and the constraint **weak_ctype** from $\mathbf{I}^T$ makes sure that **ctype** is a weak type parametric in the flag. Thus the target context *cannot* directly call the IO operations or the operation **get_mstate** because of flag-based effect polymorphism (as explained in §3.2.4), and has to instead use the secure IO operations (of type **io_lib fl** Σ', type explained in §3.2.7) to do any IO.

The compilation of a partial source program (**compile_prog**) is defined as explained in §3.2.6. The result of the compilation is a partial target program (**prog**^{T}) that expects a target context that was instantiated so that it satisfies specification $\mathbf{I}^T.\Sigma$. The MIO computation of the partial target program (type **prog**^{T}) has no pre- and postcondition and is indexed by the flag AllOps to be able to run the dynamic checks that call the **get_mstate** operation.

The target linking (**link**^{T}) first instantiates the target context and then it applies the partial program to the instantiated context. The context is instantiated with the AllOps

flag (since the context calls into the secure IO operations and wrapped callbacks that use get_mstate) and with the secure IO operations, thus obtaining an instantiated context that satisfies specification $\mathrm{I}^T\Sigma$ (as also explained in §3.2.7). The result of the target linking is a whole program (whole^T), which is a MIO computation with no pre- and postcondition, that takes a unit and returns an int.

3.6.2 Trace-Producing Semantics

We define a trace-producing semantics to reason about whole programs. For that, we use trace properties as predicates over complete IO traces, which are traces of terminated executions together with the final result of the whole program (which is an int, like in UNIX).

```
type trace_property = trace * int → Type0
```

Whole programs are computations in the monadic effect MIO, so to reason about them we have to reveal their monadic representation using the reify construct of F* [103]. Thus, we define the following trace-producing semantics function beh, used in security theorems (§3.6.3):

[103]: Grimm et al. (2018), *A Monadic Framework for Relational Verification: Applied to Information Security, Program Equivalence, and Optimizations*

```
let beh (wt:whole^T) : trace_property = beh_m (reify wt)
```

beh is defined for whole target programs, but it can also be used to define trace-producing semantics for whole source programs. For concision, we write beh for both target and source programs.

Having reify reveal the underlying representation of the computation, we define the semantics function beh_m over the computational monad m. In its definition, we use the already presented monad morphism θ (3.3.1), which gives a weakest-precondition semantics to m, and which we adapt into a trace-producing semantics by embedding weakest-preconditions to predicates [9]:

[9]: Maillard et al. (2019), *Dijkstra Monads for All*

```
let beh_m (wt : m int) : trace_property = λ (lt,res) → ∀ p. θ wt p [] ⟹ p lt res
```

Because we use the monad morphism when defining beh_m, we can lift the post-condition of a computation to a trace property—for example, for a computation w of type mio int AllOps ⊤ ψ (3.3.1), where ψ is a postcondition, we can state that $\mathrm{beh\_m}(w) \subseteq \psi$, where tp ⊆ post is defined as ∀ lt r. tp (lt, r) ⟹ post [] r lt. This follows naturally from the representation of the Dijkstra monad (explained in 3.3.1) because the refinement in the type of the monad becomes an extrinsic property.

3.6.3 Soundness of Hybrid Enforcement with respect to Global Trace Property

In our source language, one can verify that the behavior of the partial program together with the context satisfy a global trace property. One can do this by encoding the global trace property into the postcondition of who has the initial control, so for now the partial program (the dual setting is explained in §3.6.5). For example, we verified that the web server "responds to every request" by encoding this property in the postcondition of the web server, and this property characterizes both the behavior of the web server and of the request handler. Since the web server has the initial control, its postcondition naturally becomes the postcondition of the whole program.

We prove in F⋆ that the global trace property verified in the source also holds in the target because of the dynamic checks added by SCIO⋆. Thus, we show that the added higher-order contracts and the enforced access control policy soundly enforce the global trace property.

When the partial program has the initial control, its postcondition (denoted by $\mathsf{I}^S.\psi$ in Figure 3.3) is the global trace property and we show that the property holds after compilation and linking the compiled partial program against any target context.

Theorem 3.6.1 (Soundness)

$$\forall I^S.\, \forall P : \mathrm{prog}^S\ I^S.\, \forall C^T : \mathrm{ctx}^T\ I^S{\downarrow}\ .\ \mathrm{beh}(C^T[P{\downarrow}]) \subseteq I^S.\psi$$

Proof sketch. We unfold the compilation and the target linking and obtain that

$$C^T[P{\downarrow}] == P\ \mathrm{AllOps}\ (\mathrm{import}\ (C^T\ \mathrm{AllOps}\ (\mathrm{enforce\_policy\ call\_io}\ I^T.\Pi))\ (\mathrm{make\_checks\_eff}\ I^S.\mathrm{cks})).$$

Before unfolding the term $C^T[P\ {\downarrow}]$ has the weak type $\mathrm{whole}^T\ =\ \mathsf{unit} \to$ MIO int AllOps ⊤ ⊤, but the term after unfolding is an instantiation of the source program P, which has the stronger type unit → MIO int AllOps ⊤ $I^S.\psi$. From the term after unfolding, we can lift the postcondition to a trace property (as explained in §3.6.2) and have that $\mathrm{beh}(P\ \mathrm{AllOps}\ (\mathrm{import}\ (C^T\ ...)\ ...)) \subseteq I^S.\psi$, which implies that also $\mathrm{beh}(C^T[P{\downarrow}]) \subseteq I^S.\psi$. □

The proof follows so quickly because of the intrinsic specifications that are part of the types of the source partial program and of the imported context. Crucially, we lifted the postcondition of the partial program to an extrinsic property about the whole program. Less obvious is the role of the specification of the imported context in the proof: the behavior of the partial program is verified, including on how and if it calls the context, based on the specification of the context. Thus, it is enough to pass to the partial program a context with the correct specifications—i.e., with the correct type. The specifications are given to the target context by instantiating the context with the secure IO operations which enforce the access control policy, and by the dynamic checks enforced by the import function which is verified to be correct. We know that the access control policy and the checks give the correct specification to the context because of the type constraint ($I^S.\mathrm{importable\_ctype}$) between them. So the proof of soundness is done modularly by typing and also heavily benefits from SMT automation in F⋆ [2, 9].

[2]: Swamy et al. (2016), *Dependent Types and Multi-Monadic Effects in F**
[9]: Maillard et al. (2019), *Dijkstra Monads for All*

3.6.4 Robust Relational Hyperproperty Preservation (RrHP)

We prove that SCIO⋆ robustly preserves relational hyperproperties, which is the strongest secure compilation criterion of Abate et al. [8], and in particular stronger than full abstraction, as proved by Abate et al. [8] for a determinate setting with IO that closely matches ours. Relational hyperproperties [8] are a very broad class of security properties that includes trace properties (such as safety and liveness), hyperproperties [104] (such as noninterference), and also properties that relate the behaviors of multiple programs (such as trace equivalence). Robust Relational Hyperproperty Preservation (RrHP) states that for any relational hyperproperty that a collection of source programs robustly satisfies—i.e., the programs satisfy this relational hyperproperty when each of them is linked with the same arbitrary source context—then the compilation of these programs will also robustly satisfy the same

[8]: Abate et al. (2019), *Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation*
[104]: Clarkson et al. (2010), *Hyperproperties*

relational hyperproperty with respect to an arbitrary target context. Intuitively, in order to achieve such a strong criterion, the various parts of the secure compilation framework have to work together to provide enough protection to the compiled programs so that a linked target context doesn't have more attack power than a source context would have against the original source programs.

One challenge in stating RrHP in our setting is that, as explained in §3.2, SCIO* allows the program and the context to recover from contract and monitoring errors. This is necessary for functionality, as it would be unacceptable for our web server to crash whenever a dynamic check fails. This does trade off some security though, because the results of these dynamic checks can potentially be used by the target context to mount some attacks that rely on the information they indirectly reveal about the program and its secrets. Our RrHP theorem allows this information flow that improves functionality because we align the attack capabilities of the source and the target contexts,[6] so that the recoverable errors caused by enforcement are possible in both contexts. We achieve this by also explicitly passing the effectful checks to the source context, which enables the source context to mount the same kind of attacks as the target context.

6: Aligning the attack capabilities of the source and target contexts is a common necessity in secure compilation [8, 11].

To state RrHP, we present the definitions of source contexts and source linking from Figure 3.3. The definition of source contexts ($\mathbf{ctx}^S$) differs from the definition of target contexts ($\mathbf{ctx}^T$) in two ways. First, the type of source contexts is indexed by a source interface (I^S) instead of a target interface (I^T). Second, a source context has an extra argument for the effectful checks that we pass to it in order to align the attack capabilities of the source and target contexts. We pass the effectful checks to the source context during source linking (see $\mathbf{link}^S$ in Figure 3.3), in contrast to what happens at the target level where the compiled partial program is the one that passes the effectful checks to the imported target context. Finally, the source context also has to be parametric in the flag because it should not have access to the **get_mstate** operation, since otherwise it would be able to distinguish between different source programs and their secrets just by looking at the trace, which would mean that source programs could robustly satisfy only trace properties, but not hyperproperties, making the RrHP theorem much weaker.

We are now ready to state RrHP and we use a property-free characterization that was proposed by Abate et al. [8] and that is generally better tailored for proofs:

Theorem 3.6.2 (Robust Relational Hyperproperty Preservation (RrHP))

$$\forall I^S.\, \forall C^T : \mathrm{ctx}^T\ I^S{\downarrow}\ .\ \exists C^S : \mathrm{ctx}^S\ I^S.\, \forall P : \mathrm{prog}^S\ I^S.\, \mathrm{beh}(C^T[P{\downarrow}]) = \mathrm{beh}(C^S[P])$$

Proof sketch. Looking at the quantifier structure, to prove this theorem one has to create a source context by defining a back-translation that only takes the target context as argument. In our case, we can define back-translation by partially applying **import** (see **back_translate_ctx** in Figure 3.3), making it very similar to what compilation does to the target context. These definitions of compilation, back-translation, and source linking allow us to prove that compiling the program and linking it with the context is syntactically equal to back-translating the context and linking it with the program: $\forall I^S.\, \forall C^T : \mathrm{ctx}^T\ I^S{\downarrow}\ .\ \forall P : \mathrm{prog}^S\ I^S.\ \ C^T[P{\downarrow}] = C^T{\uparrow}\ [P]$. This syntactic inversion law is proved by just unfolding the definitions and makes the proof of the RrHP criterion immediate. □

While RrHP is the strongest secure compilation criterion of Abate et al. [8], and such criteria are generally challenging to prove [8, 11, 90–92], we have set things up so that it holds by construction, so our proof is easy. This simplicity is possible because (1) our

[8]: Abate et al. (2019), *Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation*
[11]: Patrignani et al. (2019), *Formal Approaches to Secure Compilation: A survey of fully abstract compilation and related work*
[90]: Devriese et al. (2017), *Modular, Fully-abstract Compilation by Approximate Back-translation*
[91]: New et al. (2016), *Fully abstract compilation via universal embedding*
[92]: Jacobs et al. (2022), *Purity of an ST monad: full abstraction by semantically typed back-translation*

languages are shallowly embedded in F*; (2) we use flag-based effect polymorphism to model the context; and (3) we design our higher-order contracts mechanism so that we can define both compilation and back-translation. These elements allow us to avoid a sophisticated logical relations argument [8, 90, 91] by instead proving a syntactic inversion law relating not only compilation and back-translation, but also source and target linking. Target linking is important here, because it adds the checks that enforce the access control policy on the IO operations of the context.

Defining the back-translation function is often the most interesting part of the proof of such secure compilation criteria. Also in our setting it is non-trivial to define back-translation: we had to redesign the higher-order contracts mechanism (as explained in §3.4.2) so that back-translation results in source contexts that are flag-based effect polymorphic.

3.6.5 Dual Setting: The Context Has Initial Control

In previous subsections we formally defined the SCIO* framework in the setting when the partial program has the initial control and uses the context as a library. In this subsection, we show that SCIO* and its security theorems also work in the dual setting—i.e., when the context has initial control and uses the partial program as a library. In this presentation we focus on the main differences compared to the previous setting (Figure 3.3). At the level of the interfaces, we replace the type of the context (ctype) with a type for the partial program (ptype) and redefine the notions of partial program and context so that the context takes the partial program as argument:

```
type prog^S I^S = fl:erased tflag → I^S.ptype (fl+IOOps)
type ctx^T I^T = fl:erased tflag → Σ':erased _ → io_lib fl Σ' →
  I^T.ptype fl → unit → MIO int fl ⊤ Σ'
```

The type of the partial program has to be exportable, since we have to prepare the partial program by exporting it before passing it to the context. The definition of compilation and back-translation change so that instead of importing the context, the partial program is exported.

For this dual setting, we had to redefine the soundness theorem. When the partial program has the initial control, it is statically verified to satisfy its postcondition, thus the soundness theorem guarantees that the resulting target whole program also satisfies this postcondition. When the unverified context has the initial control, however, we only know that it is monitored, thus, the soundness theorem for this setting guarantees instead that the resulting target whole program satisfies the specification of the access control policy Σ.

Theorem 3.6.3 (Soundness-Dual)

$$\forall I^S.\ \forall P : \mathrm{prog}^S\ I^S.\ \forall C^T : \mathrm{ctx}^T\ I^S{\downarrow}\ .\ \mathrm{beh}(C^T[P{\downarrow}]) \subseteq I^S.\Sigma$$

We proved this soundness theorem using the same strategy as for Theorem 3.6.1. We also proved the RrHP theorem in this dual setting using the same strategy as for Theorem 3.6.2, since the same syntactic inversion law also holds in this setting. All proofs have been machine-checked in F*. The artifact also contains instantiations of this dual setting with some examples (one presented in §3.8).

3.6.6 Syntactic Representation of Target Contexts

In §3.6.1 we defined target contexts as flag-polymorphic functions taking Σ' and io_lib as arguments:

type $\mathrm{ctx}^T\ \mathrm{I}^T$ = fl:erased tflag → #Σ':erased policy_spec → io_lib fl Σ' → I^T.ctype fl Σ'

These functions are good enough to represent general contexts that make calls to the IO operations provided by the io_lib argument. However, this definition of contexts might not be entirely transparent about the expressive capabilities of the target contexts and the fact that they are unverified. To make these more apparent, we also introduce a syntactic representation of target contexts. For this, we define a small deeply embedded language and a total translation function taking an arbitrary syntactic expression in this language and translating it to a function of type $\mathrm{ctx}^T\mathrm{I}^T$ representing a target context. Our deeply embedded language is a simply-typed lambda calculus with primitive types (such as bytes, int, etc.), and we wrote the adversarial handlers of §3.2 also as syntactic expressions in this language, and then translated those expressions to obtain handlers that behave, when executed, as the original ones do (§3.7). We have reproved Theorems 3.6.1 and 3.6.2 also when the target context is a syntactic expression, in which case target linking and back-translation do the extra step of first translating this expression into a context with a weak interface. This chapter involves no verified extraction: source and target are both shallowly embedded in F⋆, so this deeply embedded language only makes the capabilities of target contexts more apparent by defining a simple back-translation. We look at how to obtain soundness and RrHP for extraction in Chapter 5.

3.7 Running the Case Study in OCaml

We run our main case study by extracting both the compiled web server and the handlers from our target language (a subset of F⋆) to OCaml using the standard extraction mechanism of F⋆ and test that they execute as intended when linked with wrapped up versions of the realistic IO library of OCaml, which reads and writes to files and network sockets. While securing and formally verifying this extra step going from our target language to OCaml is out of scope for this work, this experiment still offers empirical evidence that several attacks attempted by the adversarial request handlers written in our target language are blocked by the dynamic checks of either the higher-order contracts or the reference monitor, even at the OCaml level. Therefore, the interesting compilation step we secure and verify here works as expected also after further extraction to OCaml, and this should provide a good base for the more challenging task of building a larger secure compilation framework that protects verified programs against arbitrary contexts written in a safe subset of OCaml (Chapter 6). The web server was linked against the adversarial handlers from §3.2 and a non-adversarial handler that responds to HTTP requests from a real browser.

To run the web server, we implemented the recording of IO operations in the monitor state in F⋆ using a variant of the state effect that extracts to native OCaml references. This allowed us to verify that the individual updates of the monitor state are done correctly. Our implementation is parametric in an initial state init_st that abstracts the empty trace, and a verified upd_st function that, given a monitor state and an event, produces a new state that abstracts the new trace.

```
type init_st (mst:mstate) = s:mst.typ{s `mst.abstracts` []}
type upd_st mst =
  s0:mst.typ → e:event →
  s1:mst.typ{∀ h. s0 `mst.abstracts` h ⟹ s1 `mst.abstracts` e::h}
```

3.8 Other Classes of Examples

In this section, we illustrate the applicability of SCIO⋆ by presenting three more examples: (1) the dual setting of §3.6.5 where the context now has the initial control; (2) a higher-order context that returns a callback; (3) different instantiations of the monitor state, including a stateless monitor.

1) Dual Setting. Our logging IO library prints to the console all the IO requests and guarantees that no IO requests from the context can happen without being logged. In this example, the context has the initial control and it gets the partial program as argument, where the partial program is a logging function that writes to the console its arguments. To obtain our desired guarantees, we pick a specification that states that each IO operation done by the context is preceded by a logging operation done by the program, and each logging operation must be followed by an IO operation done by the context. We encode this as the specification of the access control policy:

```
let Σ h caller op arg : policy_spec =
  match caller, op with
  | Ctx, _ → h ≠ [] ∧ hd h == EWrite Prog (stdout, to_string op) (Inl ())
  | Prog, Write → h == [] ∨ get_caller (hd h) == Ctx
  | _ → ⊥
```

Therefore, the context is forced to call the program before being able to do any IO operations. We instantiated the SCIO⋆ framework with this example, thus, thanks to Theorem 3.6.3, the resulting target whole program satisfies the following specification: no IO request from the context can happen without being logged.

2) Archiving Library. In this example, the partial program has the initial control and uses an untrusted higher-order function (of type zip below) to archive files. The untrusted function takes as first argument the file descriptor of the archive, writes the main header of the archive to it, and then returns another function (of type zip_file below) that the program has to call to add files into the archive. The specification of the archiving function guarantees that it only reads and writes to the files opened by the partial program.

```
type zip_file =
  fd:file_descr → MIO (either unit err) mymst
    (requires (λ h → is_open fd h))
    (ensures  (λ _ _ lt → only_reads_and_writes_to_fds_opened_by_prog lt))

type zip =
  afd:file_descr → MIO (either zip_file err) mymst
    (requires (λ h → is_open afd h))
    (ensures  (λ _ _ lt → only_reads_and_writes_to_fds_opened_by_prog lt))
```

This highlights that SCIO⋆ fully supports higher-order contexts, including ones returning functions.

3) The Monitor State. Our artifact contains multiple examples that show different instantiations of the monitor state, in addition to the one of the web server from §3.5.3. The archiving library from 2 has as state the entire trace of past IO events. The IO logging library example has as monitor state only the last event that happened. We also have an example where the monitor is stateless. These different examples showcase that the monitor state can be chosen flexibly.

3.9 Related Work

Secure Compilation. There are many approaches to secure compilation [8, 11], but to our knowledge, the only ones to support secure compilation of *formally verified programs against unverified contexts* are those of Agten, Jacobs, and Piessens [27] and Strydonck, Piessens, and Devriese [105]. They protect programs verified with separation logic against adversarial contexts using protected module architectures [27, 106, 107] or linear capabilities [108]. While they focus on stateful code and prove full abstraction, SCIO* focuses on code that can perform IO, and we establish RrHP with machine-checked proofs in F*.

[27]: Agten et al. (2015), *Sound Modular Verification of C Code Executing in an Unverified Context*

Dependent Interoperability. Strong interfaces in SCIO* contain refinement types and pre- and postconditions that can depend on function arguments and results. Converting refinement types into dynamic checks is inspired by Tanter and Tabareau [101], who introduce a mechanism based on type classes. We extend this idea to convert pre- and postconditions, and to go beyond pure functions, addressing new challenges. We see SCIO* as a first step towards achieving secure compilation from F* to OCaml (Chapter 6). The next steps could build upon work on *dependent interoperability* [109, 110].

[101]: Tanter et al. (2015), *Gradual certified programming in Coq*

Higher-Order Contracts. Findler and Felleisen [15] pioneered higher-order contracts, now a standard feature of Racket [111, Chapter 8]. Several works have explored extensions to stateful contracts, e.g., Disney, Flanagan, and McCarthy [16] propose temporal higher-order contracts, Scholliers, Tanter, and Meuter [17] propose computational contracts, and Tov and Pucella [19] study stateful contracts for affine types. Stateful contracts can be added at the boundary between the partial program and the context, and they can be used to implement an IO reference monitor. In particular, Moore et al. [18] propose authorization contracts for implementing access control monitors for software components. The key novelty of SCIO* is to integrate statically-verified code with untrusted code in a tool that is itself verified. It would be interesting to explore whether soft contract verification [112] could be used in SCIO* to eliminate dynamic checks that can be verified statically.

[15]: Findler et al. (2002), *Contracts for higher-order functions*
[16]: Disney et al. (2011), *Temporal higher-order contracts*
[17]: Scholliers et al. (2015), *Computational contracts*
[19]: Tov et al. (2010), *Stateful Contracts for Affine Types*

Runtime Verification. There is extensive prior work on how to monitor program executions either by using runtime verification or instrumentation. Java-MOP is a framework for Monitoring-Oriented Programming (MOP) that builds on Aspect-Oriented Programming (AOP) in Java [113–115]. They focus on synthesizing the monitor from formal specifications (e.g., in LTL) to instrument the whole program. Automatically synthesizing the dynamic checks is an open challenge in our setting. In this chapter, we focus instead on highly expressive specifications, on the programmability and efficiency of the checks, and on obtaining formal security guarantees, by automatically verifying that the checks are always enough to guarantee the specifications in the strong interface.

Our work currently only deals with a single policy. Prior work on flow-based monitors—intensively studied in the AOP literature from both the points of view of expressiveness [116, 117] and efficiency [118–120]—could be helpful to extend our work to support multiple, localized policies for specific contexts and linking points.

Gradual Verification. Bader, Aldrich, and Tanter [121] and Wise et al. [122] propose gradual program verification to easily combine dynamic and static verification in the same language at a very fine granularity, using variants of Hoare logic or separation logic with *imprecise* logical formulas. SCIO* is a hybrid verification and compilation framework with a coarser interoperability granularity (program vs. context), and targets the IO effect.

[121]: Bader et al. (2018), *Gradual Program Verification*
[122]: Wise et al. (2020), *Gradual verification of recursive heap data structures*

Reasoning about Robust Safety. Interoperability between trusted and untrusted code has also been studied when reasoning about robust safety [24, 123, 124]. Sammler et al. [25] show that sandboxing enables reasoning about robust safety when having a rich type system that contains the Any type, a type inhabited by all values, and a higher-order contract mechanism going between types and Any. A main difference with SCIO* is that in their work all interactions between the trusted and the untrusted code happen through the Any type. Also, they discuss only robust safety related to the memory model, not trace properties.

[25]: Sammler et al. (2020), *The high-level benefits of low-level sandboxing*

3.10 Summary

In this chapter, we have shown that it is possible to secure verified IO programs against unverified code in F* with strong formal guarantees (RrHP). The result is SCIO*, a secure compilation framework that allows for the sound verification of IO programs that can be compiled and then linked with arbitrary unverified code, by enforcing the specifications that are on the interface using a reference monitor and higher-order contracts. In the next chapter, we apply the ideas presented here to secure *stateful* programs against unverified code in F*.

We discuss future work more in Chapter 6. Here we mention that an interesting line for future work would be to use parametricity to prove a noninterference theorem formalizing that our flag-based effect polymorphic context cannot directly call GetMState or the IO operations. There are, however, at least two challenges to overcome for achieving this: (1) our noninterference statement is significantly more complex than prior work in this space [125, 126]; and (2) the erased type, its interaction with the primitive Ghost effect of F*, and their parametricity properties would first need to be better formally understood.

[125]: Algehed et al. (2019), *Simple noninterference from parametricity*
[126]: Algehed et al. (2021), *Dynamic IFC Theorems for Free!*

Securely Sharing Mutable References between Verified and Unverified Code

4

4.1 Contributions

In this chapter, we present how to securely share ML-style mutable references between verified F⋆ programs and unverified code, with strong machine-checked security guarantees. The contributions of this chapter are:

We introduce Labeled References, a computationally irrelevant labeling mechanism built on top of Monotonic State [10] (§2.5) that allows one to verify stateful F⋆ programs that share ML-style mutable references with unverified code. The labeling mechanism tracks which references are shareable with the unverified code and which ones are not. The labels simplify specifying which references can be modified by a call into unverified code—i.e., only shareable references can be modified. Our solution allows one to pass between the verified and the unverified code mutable data structures such as linked lists and callbacks that encapsulate and modify references.

[10]: Ahman et al. (2018), *Recalling a Witness: Foundations and Applications of Monotonic State*

We introduce *SecRef⋆*, a secure compilation framework built around Labeled References and inspired by SCIO⋆, our previous work for the IO effect (Chapter 3). Like SCIO⋆, we shallowly embed the source and target language in F⋆. The only effect our languages have is state, with first-order ML-style mutable references. The SecRef⋆ compiler takes a program verified using Labeled References and protects it so that it can be securely linked with unverified code, which can only modify shareable references. The compiler converts some of the specifications at the boundary between verified and unverified code into dynamic checks by adding higher-order contracts, and we verified in F⋆ the soundness of this enforcement with respect to the specification of the program.

Moreover, a machine-checked proof in F⋆ shows that a stateful program verified using Labeled References, then compiled with SecRef⋆, can be securely linked with arbitrary unverified code. For this we prove Robust Relational Hyperproperty Preservation (RrHP), the proof following the proof done for SCIO⋆.

Underlying these results is a monadic representation of Monotonic State that we contribute in Chapter 2 (§2.5), which is crucial for our mechanized proofs above.

Finally, we put our framework into practice by developing a case study inspired by a simple cooperative multi-threading model. In this case study, we showcase that we can statically verify a simple scheduler even when it interacts with and manages unverified threads.

Outline. §4.2 starts with the key ideas of SecRef⋆. §4.3 builds Labeled References on top of Monotonic State. §4.4 then presents our higher-order contracts. §4.5 shows how all pieces of SecRef⋆ come together and how we prove that SecRef⋆ satisfies soundness and RrHP. §4.6 presents the cooperative multi-threading case study. Finally, §4.7 discusses related work, followed by an additional example (§4.8), and §4.9 gives a summary of the chapter.

Artifact. This chapter comes with an F* artifact that includes the full implementation of SecRef*, the machine-checked proofs, and examples. The examples can be extracted to native OCaml code (without any monadic representation). To find the artifact check §1.3, or check the original artifact on Zenodo [52] or Github [53]. The artifact is 5480 lines of F* code[1] containing 807 top-level definitions, 90 of which are lemmas. Of these, the implementation of Labeled References takes 984 lines.

1: Comment and blank lines were not counted towards these numbers.

4.2 Key Ideas of SecRef*

In this section, §4.2.1 introduces the running example of an autograder sharing references with unverified code, and showing how to verify it. §4.2.2 then explains how we represent unverified code and §4.2.3 how we add higher-order contracts. Lastly, beyond private and shareable, §4.2.4 introduces a third label, encapsulated, for references that are not shareable with the unverified code, but that can be indirectly updated by verified callbacks that are passed to the unverified code.

```
1  type linked_list (a:Type) =
2  | LLNil → linked_list a
3  | LLCons : a → ref (linked_list a) → linked_list a

5  type student_hw = ll:ref (linked_list int) →
6    LR (either unit err)
7      (requires (λ h0 → witnessed (is_shareable ll)))
8      (ensures  (λ h0 r h1 → modif_only_shareable h0 h1 ∧ same_labels h0 h1 ∧
9        (Inr? r ∨ (sorted ll h1 ∧ no_cycles ll h1 ∧ same_values ll h0 h1))))

11 let autograder (test:list int) (hw:student_hw) (grade:mref (option int) grade_preorder) :
12   LR unit
13     (requires (λ h0 → is_private h0 grade ∧ None? (sel h0 grade))
14     (ensures  (λ h0 () h1 → Some? (sel h1 grade) ∧
15       same_labels h0 h1 ∧ modif_shareable_and !{grade} h0 h1)) =
16     let ll = create_llist test in
17     label_llist_as_shareable ll;
18     witness (is_shareable ll);
19     let res = hw ll in
20     gr := Some (determine_grade res)
```

Figure 4.1: The illustrative example of a verified autograder. Types are simplified.

4.2.1 Static Verification Using Labeled References

In Figure 4.1 we show how to use Labeled References to verify an autograder. The autograder takes as argument a function submitted by a student to solve the homework (hw of type student_hw) and an initially empty (None) reference where the grade will be stored (line 11). The function provided by the student is supposed to sort a linked list in place. The autograder creates a shareable linked list, executes the homework on it, checks the result, and sets the grade (lines 16-20). The main property verified for the autograder is that the grade stays private and that it gets set to some value (lines 14-15).

Naturally, the homework is unverified since it is written by a student. Thus, since the autograder calls the unverified homework, to be able to verify it, one has to assume

a specification for the behavior of the homework. SecRef* then ensures that the homework satisfies the specification. Soundly specifying the behavior of unverified code is one of the main contributions of this chapter, and we explain this below for **student_hw** (lines 6-9 in Figure 4.1), illustrating the assumed specification:

1. The precondition requires that the linked list is labeled as shareable, statically guaranteeing that the verified autograder passes only shareable references to the homework (line 7).
2. The postcondition ensures that only shareable references have been modified, and existing references have the same label (line 8). This is a universal property of all unverified computations, which work only with shareable references (§4.2.2).
3. The postcondition also ensures that if the computation succeeds (by returning Inl), the linked list is sorted, without cycles, and contains the same values (line 9). This is the part of the specification that gets enforced dynamically using higher-order contracts (§4.2.3).

The way we specify the footprint in Figure 4.1 in the postcondition of the **student_hw** (item (2) above, line 8) is novel. We specify it as the set of references that are labeled as shareable (using the predicate **modif_only_shareable**). The usual way of specifying it as the set of references that are in the linked list (i.e., the root reference, the one that follows, and so on to the last reference in the linked list) [10, 23, 127] does not scale in our setting, where we also need to specify higher-order unverified code. Imagine that the unverified function is in fact a result of a bigger piece of unverified code and that another linked list was already shared with it (as in the example from the introduction). This means that the unverified function can also modify that linked list, thus, to soundly specify its behavior one has to add to the footprint the references in that linked list too. Moreover, references that were shared through subsequent updates to the first linked list also have to be added. Using our labeling mechanism tames the complexity of globally tracking the shared references and allows stating the footprint in a simple way.

[10]: Ahman et al. (2018), *Recalling a Witness: Foundations and Applications of Monotonic State*
[23]: Reynolds (2002), *Separation Logic: A Logic for Shared Mutable Data Structures*
[127]: Leino (2010), *Dafny: An Automatic Program Verifier for Functional Correctness*

The assumption that unverified code modifies only shareable references is true if (1) the unverified code cannot forge references (which we assume), (2) we treat the references allocated by the unverified code as shareable references (which we do and explain in §4.2.2), and (3) the verified code shares only shareable references with the unverified code. To ensure this last point we added the precondition for calling the unverified code that the passed reference has to be shareable (item (1), line 7). This precondition in combination with a global invariant on the heap enforces that the explicitly passed reference is shareable, and that all references that are reachable from the reference are shareable too. The global invariant on the heap ensures that a shareable reference can point only to other shareable references. This means that the SecRef* user has to manually track only the label of the root reference of a complex data structure (e.g., the head of a linked list). The global invariant is necessary to prevent accidental sharing by subsequent writes to shareable references (e.g., the example from the introduction). The user never has to prove directly the preservation of the global invariant though, because the operations to manipulate references provided by Labeled References take care of that. The only extra proof burden appears when labeling a reference as shareable or writing to a shareable reference, one has to prove some conditions ensuring that the affected reference will point only to shareable references after the operation.

The **witnessed** token used in the precondition (line 7) is produced by the **witness** operation of Monotonic State, which provides two such operations: (1) **witness**, allowing one to convert stable heap predicates into heap-independent ones, and (2) **recall**, allowing one to recover an already witnessed predicate for any future heap. For example,

being shareable is a stable property with respect to our preorder that shareable references stay shareable. Thus if one knows that a reference r is shareable in a heap h (i.e., is_shareable r h), one can witness this and get a heap-independent token witnessed (is_shareable r), which can be recalled later, say when the heap is h', to prove that is_shareable r h'.[2] For intuition, making is_shareable r into a heap-independent token witnessed (is_shareable r) can be thought of as defining a proposition, which states that there exists some past heap h in which is_shareable r h was true, and as is_shareable r is stable with respect to any allowed heap changes, is_shareable r h' is also true in any future heap h' from h.

2: For readers familiar with Separation Logic, this is similar to invariants and their introduction and elimination rules.

To be able to statically verify the specifications of the autograder (lines 13-15) we reiterate that giving a specification to the unverified function is necessary. Since the autograder calls into unverified code, it inherits the property that it can modify shareable references, thus, its footprint is defined as the union of all shareable references and the singleton set of references containing only the grade (line 15). The autograder's code stores the grade inside a monotonic reference (denoted by mref on line 11), a feature of Monotonic State [10] that allows putting a statically enforced preorder on the value stored in a reference. For the grade, the preorder ensures that the grade cannot be modified once it is set. To be able to set the grade (on line 20), however, one has to know that the grade was not set by calling into the homework, which we know because the footprint of the call states that it modifies only shareable references, and we know that the grade was private before the call. The preorder on the grade is defined as follows:

```
let grade_preorder (g0 g1:option int) : Type0 =
  match g0 with
  | None → T
  | Some v0 → Some? g1 ∧ v0 = Some?.v g1
```

In SecRef*, one cannot share a monotonic reference with the unverified code because there would be no way to guarantee the preorder of the reference, but the verified code can use monotonic references. One can share only normal references, denoted by ref, as long as they are labeled as shareable. Normal references are just monotonic references with the trivial preorder.

4.2.2 Representation for Unverified Code

We give a representation for unverified code as a shallow embedding in F* satisfying two constraints: first, obviously, it has to be a representation for *unverified* code (i.e., it requires only trivial logical reasoning), and second, we need to be able to show that any piece of unverified code using this representation satisfies the universal property that it modifies only shareable references.

We found a representation that satisfies both these constraints and that is also unaware of the labeling mechanism, by using the effect MST. The representation uses abstract predicates as well as a version of the reference-manipulating operations that mention only these abstract predicates in their specifications. The intuition why this is an appropriate representation for *unverified* code in F* is that one can sequence the new operations freely, because the operations require and ensure only the abstract predicates, so one only has to carry the abstract predicates around from postconditions to preconditions, but one never has to do nontrivial logical reasoning. To make it even more apparent that it is an appropriate representation, in the artifact we also have a syntactic representation, a small deeply embedded λ-calculus with first-order ML-style references, and we show a total (back)translation function in

```
type poly_alloc_t (𝓘:heap → Type₀) (φ:ref α → Type₀) (≼:preorder heap) =
  init:α → MST (ref α) (requires (λ h₀ → 𝓘 h₀ ∧ every_mref φ init))
                       (ensures  (λ h₀ r h₁ → h₀ ≼ h₁ ∧ 𝓘 h₁ ∧ φ r))

type poly_read_t (𝓘:heap → Type₀) (φ:ref α → Type₀) (≼:preorder heap) =
  r:ref α → MST α (requires (λ h₀ → 𝓘 h₀ ∧ φ r))
                  (ensures  (λ h₀ v h₁ → h₀ ≼ h₁ ∧ 𝓘 h₁ ∧ every_mref φ v))

type poly_write_t (𝓘:heap → Type₀) (φ:ref α → Type₀) (≼:preorder heap) =
  r:ref α → v:α → MST unit (requires (λ h₀ → 𝓘 h₀ ∧ φ r ∧ every_mref φ v))
                           (ensures  (λ h₀ _ h₁ → h₀ ≼ h₁ ∧ 𝓘 h₁))
```

Figure 4.2: Types of the reference-manipulating operations polymorphic in $\mathcal{I}$, φ and $\preccurlyeq$. The $\textbf{every}_{\text{mref}}$ combinator is defined using a type class and here it is elided that type α is constrained by this type class.

F* from any well-typed syntactic λ-calculus expression to the shallowly embedded representation with abstract predicates.

The type of the new reference-manipulating operations is presented in Figure 4.2. They use three abstract predicates in a way that allows the operations to still be freely sequenced: (1) a global invariant on the heap ($\mathcal{I}$), required and guaranteed by all operations; (2) a predicate on references (φ) constraining references stored in the heap by **write** and **alloc**, ensured of freshly allocated references using **alloc**, and ensured of references returned from the heap by **read**; (3) a preorder on heaps ($\preccurlyeq$) and a guarantee that for all our operations the heap evolves according to this preorder. In Figure 4.2 we also use the $\textbf{every}_{\text{mref}}$ combinator that takes a predicate on references (in our case φ) and a value of an inductive type, then recursively decomposes the value by pattern matching. For each reference, it applies the predicate, ensuring that all reachable references satisfy it.

We call this representation, a *polymorphic interface*, and we used it for instance to write the unverified homework of 4.3, which sorts a linked list and which takes as argument the three operations and combines them freely.

The abstract predicates are also used for specifying the homework in the target language, and we explain how F* shows that the homework satisfies its postcondition, even if it is abstract. The first conjunct of the postcondition is about the preorder on heaps $\preccurlyeq$, which is satisfied because all our 3 operations ensure that the heap evolves according to $\preccurlyeq$, which is a reflexive and transitive relation. The second conjunct about global invariant $\mathcal{I}$ is satisfied because it is required in the precondition of **poly_student_hw**, and then every operation requires and ensures this invariant. The third and last conjunct states that any reference that is returned has to satisfy predicate φ. This conjunct is satisfied because a reference can be allocated (which ensures φ), read from the heap (which ensures φ), or is received from the outside, which is also guaranteed because of the precondition requiring that the input linked list satisfies φ. The precondition and the postcondition of **poly_student_hw** ensure that every reference that crosses the verified-unverified boundary satisfies the predicate φ.

Now time for some magic: we show that we can instantiate the polymorphic interface we give to unverified code, so that this code gets closer to something that can be safely linked with verified code. We illustrate this instantiation step by step for **poly_student_hw** above. Instantiating the abstract invariant, $\mathcal{I}$, lifts all the functions to the effect LR, which is just a synonym for the effect MST with an extra global invariant, ensuring that shareable points to shareable, so after instantiation we have the same effect as in the type **student_hw** from Figure 4.1. We instantiate φ with

```
type poly_student_hw = ℐ:(heap → Type0) → φ:(ref α → Type0) → ≼:preorder heap →
 (* set of operations that use the abstract predicates *)
 alloc:poly_alloc_t ℐ φ (≼) → read:poly_read_t ℐ φ (≼) → write:poly_write_t ℐ φ (≼) →
 (* the actual type of the homework *)
 ll:ref (linked_list int) →
 MST (either unit err)
  (requires (λ h0 → ℐ h0 ∧ φ ll))
  (ensures  (λ h0 r h1 → h0 ≼ h1 ∧ ℐ h1 ∧ every_mref φ r))

let unverified_student_hw : poly_student_hw = λ my_alloc my_read my_write →
 let rec sort (ll:ref (linked_list int)) =
  match my_read ll with
  | LLNil → ()
  | LLCons x tl → begin
    sort tl;
    match my_read tl with
    | LLNil → ()
    | LLCons x' tl' →
      if x ≤ x' then () else (
        let new_tl = my_alloc (LLCons x tl') in
        sort new_tl;
        my_write ll (LLCons x' new_tl))
  end
 in sort
```

Figure 4.3: An unverified homework sorting a linked list[3], written against the polymorphic interface of Figure 4.2.

3: We elided the use of fuel to have a proof of termination, which is needed since for simplicity our work assumes that state is the only side effect. The full implementation is in the artifact.

λ r → witnessed (is_shareable r), which becomes the same precondition as in type student_hw. Then, we instantiate the preorder ≼, with
λ h_0 h_1 → modif_only_shareable h_0 h_1 ∧ same_labels h_0 h_1, which means we now have as postcondition the universal property that unverified code modifies only shareable references, which is also the first part of the postcondition of student_hw. So unverified code is instantiated during linking to obtain a type with concrete pre- and postconditions, now having what we call an *intermediate interface*. This is a first step towards bridging the gap between verified and unverified code (see Figure 4.4), and in §4.2.3 we show how we additionally use higher-order contracts during the compilation of the verified program to fully bridge this gap.

But first, let us still mention that instantiating the three operations from Figure 4.2 is straightforward. We instantiate allocation by allocating a reference and then labeling it as shareable. And we instantiate **read** and **write** using the default operations of Labeled References, which because of the specs given by the three concrete predicates, they are now specialized for shareable references. For **read** an interesting interaction happens between $\textbf{every}_{\text{mref}}$ φ r and the global invariant to show that whatever was read from the heap is also shareable. The $\textbf{every}_{\text{mref}}$ combinator stops when reaching any reference because once it was established that a reference is shareable, the global invariant offers the guarantee that all references reachable from this reference are also shareable.

4.2.3 Higher-Order Contracts in SecRef*

After the instantiation, an unverified computation of type unverified_student_hw almost has the type, which we call a *strong interface*, student_hw. The only missing

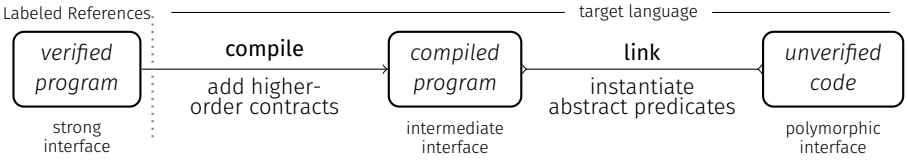


Figure 4.4: An overview of SecRef⋆.

piece is a part of the postcondition (item 3 in §4.2.1), which states that after the homework returns, if the run was successful, then the linked list is sorted, has no cycles and contains the same values as before. This part is enforced by SecRef⋆ using higher-order contracts [15], which are used to enforce only properties of shareable references or values passed when unverified code gives control to the verified code. Since the verified code has no insights into what the unverified code does with the shareable references and with which values it gives them, no logical predicates can be established about them. Therefore, it is the choice of the user if they want to strengthen the types and use higher-order contracts, or if they want to manually establish the logical predicates by adding dynamic checks. The advantage of strengthening the specification is that if one would instead link with some verified program, then there would be no need for the dynamic checks, and for unverified code, one would use SecRef⋆.

[15]: Findler et al. (2002), *Contracts for higher-order functions*

We implement the idea of Stateful Contracts [15] using wrapping and by implementing two dual functions **export** and **import**: **export** takes existing specs and converts them into dynamic checks (e.g., converts preconditions of arrows and refinements on the arguments to dynamic checks), while **import** adds dynamic checks to have more specs (e.g., adds dynamic checks to strengthen the postconditions of arrows and to add refinements on the result). The functions **export** and **import** are defined in terms of each other to support higher-order functions. In SecRef⋆, when the verified program has initial control, the **import** function is used during both compilation and back-translation (the dual of compilation, used for the secure compilation proof in §4.5). During compilation **import** is used to add dynamic checks to the verified-unverified boundary of the compiled program to protect it. During back-translation **import** is used to add dynamic checks to the unverified code so that it can be typed as having the necessary specification. In the dual setting, when the unverified code has the initial control, the **export** function is used.

Our implementation of higher-order contracts is verified not to modify the heap in any way (it only reads from the heap). SecRef⋆ also verifies if the dynamic checks imply the specification that has to be enforced. Since F⋆ specifications are not directly executable, SecRef⋆ leverages type classes to automatically infer the check. If this fails though, the user is asked to manually provide the check and prove that the check implies the specification, which benefits from SMT automation.

4.2.4 Encapsulated References

We want to enable the verified program to mutate private references inside the callbacks it passes to the unverified code. Such references are not shared with the unverified code, but are modified indirectly by the callback calls the unverified code makes. To enable this use case, we introduce a third label, **encapsulated**. Only references labeled as private can become encapsulated, and they remain encapsulated forever. Since encapsulated references are used only in verified code, there is no need to add a constraint that limits to which kind of references they point.

Encapsulated references are only directly modified by the verified program, but an unknown number of times, since the unverified code can call back as many times as it likes. The callback can also be called at later times, since it can be stashed away. The advantage of encapsulated references is that one knows *how* they get modified, thus, one can encapsulate *monotonic* references to take advantage of a preorder, and/or use refinement types and/or dependent types for the value stored.

Encapsulated references are especially useful when the unverified code has initial control. The unverified code can call multiple times the verified program, but in a first-order setting the calls are independent, and any reference allocated by the program gets lost when it gives control back to the unverified code. If the program is higher-order though, it can have callbacks with persistent state by using encapsulated references. We illustrate this case in Figure 4.5 using the example of a pseudo-number generator that has an initialization phase where it gets a seed as an argument and returns a callback that generates a number on each call. To generate a new number for every call to the callback, the program allocates an encapsulated reference, **counter**, with which it keeps track of how many times it has been called. This is a monotonic reference with the preorder that it can only increase. Since the generator does not make any assumption about the unverified code (the preconditions are trivial), SecRef* does not add any dynamic checks during compilation.

```
1 val generate_nr : int → int → int (* a pure computation *)
2 let post = λ h0 _ h1 → modif_shareable_and_encaps h0 h1 ∧ same_labels h0 h1
3 let prng (seed:int) :
4   LR (unit → LR int (requires (λ _ → ⊤)) (ensures post))
5     (requires (λ _ → ⊤))
6     (ensures  post) =
7     let counter : mref int (≤) = alloc 0 in
8     label_encapsulated counter;
9     (λ () → counter := !counter + 1; generate_nr seed (!counter))
```

Figure 4.5: The illustrative example of a pseudo-number generator

This last example also reveals that the presentation in Figure 4.1 and §4.2.2 is simplified to improve readability, and in fact, SecRef* uses the more general predicate **modif_shareable_and_encaps** (instead of the more specific **modif_only_shareable**) as a postcondition for unverified code.

4.3 Labeled References

We now explain how we build our reference labeling mechanism on top of Monotonic State (§2.5).

4.3.1 The Labels

As illustrated in §4.2, in SecRef* we work with three labels for references:

```
type label = | Private | Shareable | Encapsulated
```

When a reference is allocated, it starts labeled with **private**. The user can then relabel the reference either with **shareable** or **encapsulated**. Labeling a reference with **shareable** or **encapsulated** is permanent. We ensure this using the following preorder on reference labels:

```
let ≼lab : preorder label =
 λ l l' →
   match l, l' with
   | Private, _ | Shareable, Shareable | Encapsulated, Encapsulated → ⊤
   | _, _ → ⊥
```

To track references' labels and how they evolve in programs, we assume a ghost top-level monotonic reference[4] that stores a map from the addresses of references to their labels

4: We prefer to assume the top-level reference to avoid precisely modeling ghost state [85, 128].

```
type label_map_t =
  erased (mref (pos → erased label) (λ m0 m1 → ∀ r. (m0 r) ≼lab (m1 r)))

assume val label_map : label_map_t
```

The heap relation $\preceq$ defined in Figure 2.1 from §2.5 ensures that label_map can only evolve according to $\preccurlyeq_{\text{lab}}$, which in turn is what makes being shareable or encapsulated a stable property of references:

```
let is_shareable : mref_stable_heap_predicate =
 λ #a #rel (r:mref a rel) h →
   h `contains` label_map ∧ (* necessary to show stability *)
   Shareable? ((sel h label_map) (addr_of r))

let is_encaps : mref_stable_heap_predicate =
 λ #a #rel (r:mref a rel) h →
   h `contains` label_map ∧ (* necessary to show stability *)
   Encapsulated? ((sel h label_map) (addr_of r))
```

where mref_stable_heap_predicate is an mref-parameterized stable heap predicate. In contrast, as private references can get relabeled, being private is *not* a stable property of the heap:

```
let is_private_addr : pos → heap → Type0 = λ r h → Private? ((sel h label_map) r)

let is_private : mref_heap_predicate =
 λ #a #rel (r:mref a rel) h → is_private_addr (addr_of r) h
```

We mark the contents of the map and the map itself as erased, to on the one hand ensure that labels can only be used in specifications and programs cannot branch on them, and on the other hand, to avoid passing label_map to computationally relevant operations that expect non-erased arguments. The latter allows label_map to be erased in extracted code (as erased is extracted to unit).

Finally, we find it useful to define variants of the modifies clause discussed in §2.5 to specify that the footprint of a computation modifies only shareable or encapsulated, or both kinds of references:

```
let modif_only_shareable_and_encaps (h0:heap) (h1:heap) : Type0 =
 ∀ a rel (r:mref a rel).
   (¬(eq_addrs r label_map) ∧ ¬(is_shareable r h0 ∨ is_encaps r h0)) ⟹
     sel h0 r == sel h1 r

let modif_shareable_and (h0:heap) (h1:heap) (s:set nat) : Type0 =
 ∀ a rel (r:mref a rel).
   (¬(eq_addrs r label_map) ∧ ¬(is_shareable r h0 ∨ (addr_of r) `mem` s)) ⟹
     sel h0 r == sel h1 r
```

4.3.2 The Global Invariant for Shareable References

We introduced our global invariant on the heap, that shareable references can only point to shareable references, in §4.2.1. To formally state it in §4.3.3, we want to quantify over all references of all types that are labeled as shareable and check that the stored value contains only shareable references. A convenient way to state the invariant is using a deep embedding of the types a shareable reference can store values of. These are built from base types, sums, products and references, collectively known in the literature as full ground types [129–131]. Arrows are not part of the embedding since our heap is only first-order (see §2.5.2). The embedding is given by:

[129]: Murawski et al. (2012), *Algorithmic Games for Full Ground References*
[130]: Murawski et al. (2018), *Algorithmic games for full ground references*
[131]: Kammar et al. (2017), *A monad for full ground reference cells*

```
type full_ground_typ =
| SUnit | SInt | SBool
| SSum : full_ground_typ → full_ground_typ → full_ground_typ
| SPair : full_ground_typ → full_ground_typ → full_ground_typ
| SRef : full_ground_typ → full_ground_typ
| SLList : full_ground_typ → full_ground_typ
```

and comes with the evident recursive inclusion to_Type : full_ground_typ → Type_0 into Type_0 types.

To check that a stored value contains only shareable references, we use the deep embedding to define a predicate forall_refs which checks that every reference in a given value of full_ground_typ type satisfies a predicate. We then lift this predicate to properties of references with respect to heaps. As these predicates are parametric in pred, we can instantiate them with different predicates.

```
let rec forall_refs (pred:mref_predicate) (#t:full_ground_typ) (x:to_Type t) : Type₀ =
  let rcall #t x = forall_refs pred #t x in
  match t with
  | SUnit | SInt | SBool → ⊤
  | SSum t₁ t₂ → match x with | Inl x' → rcall x' | Inr x' → rcall x'
  | SPair t₁ t₂ → rcall x.₁ ∧ rcall x.₂
  | SRef t' → pred x
  | SLList t' → match x with | LLNil → ⊤ | LLCons v xsref → rcall v ∧ pred xsref

let forall_refs_heap (pred:mref_heap_predicate) h x = forall_refs (λ r → pred r h) x
```

Finally, to be able to define the invariant, we transitively close these predicates on a heap:

```
let trans_forall_refs_heap (h:heap) (pred:mref_stable_heap_predicate) =
  ∀ (t:full_ground_typ) (r:ref (to_Type t)). h `contains` r ∧ pred r h ⟹
    forall_refs_heap pred h (sel h r)
```

This general version we apply to is_shareable and contains predicates, e.g., so as to specify that every shareable reference r points to a value (sel h r) that contains only shareable references.

4.3.3 The LR Effect

We define the Labeled References effect LR on top of the Monotonic State effect MST (§2.5) by restricting MST computations with a global invariant that enforces the correct usage of shareable references and the ghost state storing reference labels. The invariant is defined as a conjunction of conditions:

```
let contains_pred #a #rel (r:mref a rel) h = h `contains` r

let lr_inv (h:heap) : Type0 =
  (* contained references point to contained references: *)
  trans_forall_refs_heap h contains_pred ∧
  (* shareable references point to shareable references: *)
  trans_forall_refs_heap h is_shareable ∧
  (* the labeling map is contained in the heap: *)
  h `contains` label_map ∧
  (* the labeling map remains private: *)
  is_private (label_map) h ∧
  (* all yet to be allocated references are labeled as private: *)
  (∀ r. r ≥ next_addr h ⟹ is_private_addr r h)
```

The last condition ensures that any freshly allocated reference is private by default and will only become shareable by explicitly labeling it so, allowing for fine-grained control over which references can cross the verified-unverified code boundary. This also applies to encapsulated references.

The LR effect is then defined on top of MST by packaging the invariant both into pre- and postconditions. By being an effect synonym as opposed to a new effect, we can seamlessly coerce any MST computation into LR, as long as it satisfies the invariant, something that is very helpful for linking verified code with unverified code (see §4.2.2). In detail, LR is defined in F* as follows:

```
effect LR (a:Type) pre post =
  MST a
    (requires (λ h0 → lr_inv h0 ∧ pre h0))
    (ensures  (λ h0 r h1 → lr_inv h1 ∧ post h0 r h1))
```

By providing the MST operations we discussed in §2.5 with suitable stronger pre- and postconditions, we can lift them to the LR effect. For allocation, the LR operation lr_alloc #a #rel init additionally requires that if the type a happens to be a full_ground_typ, then every reference contained in the initial value init has to be contained in the heap. In its postcondition, lr_alloc additionally guarantees that the freshly allocated reference is labeled private and no other references become shareable during its allocation. Modulo the use of lemmas proving that allocation of references preserves lr_inv, lr_alloc is defined simply as alloc, with the type

```
val lr_alloc #a (#rel:preorder a) (init:a) :
  LR (mref a rel)
    (requires (λ h0 →
      ∀ t. to_Type t == a ⟹ forall_refs_heap contains_pred h0 #t init))
    (ensures  (λ h0 r h1 →
      alloc_post #a #rel init h0 r h1 ∧ is_private r h1 ∧ becomes_shareable !{} h0 h1))
```

The specification of LR's lr_read operation remains the same as for MST's read (see Figure 2.1). For lr_write #a #rel r v, we strengthen the precondition by additionally requiring that the reference is different from the ghost state holding the labeling map, and that if a happens to be a full_ground_typ, then every reference contained in v has to be contained in the heap, and that if r happens to be a shareable reference, then every reference contained in v has to be shareable as well. In return, the postcondition additionally guarantees that no new references become shareable while writing into r. As with lr_alloc, modulo the use of some lemmas, lr_write is defined as write.

```
val lr_write #a (#rel:preorder a) (r:mref a rel) (v:a) :
  LR unit
    (requires (λ h0 → write_pre #a #rel r v ∧ ¬(eq_addrs r label_map) ∧
      (∀ t. to_Type t == a ⟹
        forall_refs_heap contains_pred h0 #t v ∧
        (is_shareable r h0 ⟹ forall_refs_heap is_shareable h0 #t v))))
    (ensures  (λ h0 () h1 →
      write_post r v h0 () h1 ∧ becomes_shareable !{} h0 h1))
```

Similarly to reading from a reference, witnessing and recalling stable predicates also retain the exact pre- and postconditions as listed in Figure 2.1, as they also keep the whole heap unchanged.

In programs and in particular in the examples in the rest of the chapter, programmers can use lr_alloc, lr_read, and lr_write via the usual ML-style syntax, as alloc, !, and := respectively.

In addition to the operations lifted from the MST effect, the LR effect comes with two important additional operations, which allow private references to be relabeled as shareable or encapsulated.

```
val label_shareable (#t:full_ground_typ) (r:ref (to_Type t)) :
  LR unit
    (requires (λ h0 → h0 `contains` r ∧ is_private r h0 ∧
      forall_refs_heap is_shareable h0 (sel h0 r)))
    (ensures  (λ h0 () h1 → equal_dom h0 h1 ∧ modifies !{label_map} h0 h1 ∧
      is_private (label_map) h1 ∧ becomes_shareable !{sr} h0 h1))

val label_encapsulated (#a:Type) (#rel:preorder a) (r:mref a rel) :
  LR unit
    (requires (λ h0 → h0 `contains` r ∧ is_private r h0 ∧ ¬(eq_addrs r label_map)))
    (ensures  (λ h0 () h1 → equal_dom h0 h1 ∧ modifies !{label_map} h0 h1 ∧
      is_private (label_map) h1 ∧ becomes_encapsulated !{r} h0 h1))
```

Both operations can be called with private references contained in the heap, with label_shareable requiring that the reference r has to store full_ground_typ-values (this helps to prove that label_shareable preserves the global invariant, as noted in §4.3.2), that r has a trivial preorder (via the type ref of non-monotonic references, see Figure 2.1), and that any references r points to have already been labeled as shareable. This last point means that when labeling complex data structures as shareable (e.g., linked lists), we have to first apply label_shareable at the end of chains of references. The label_encapsulated operation on the other hand applies to private monotonic references with arbitrary preorders, which in turn necessitates the explicit comparison with label_map—something that follows implicitly for label_shareable due to the preorders of r and label_map being different. These operations are defined by appropriately modifying the ghost state storing the labeling.

Usability. From our experience working on the examples in the chapter, we report that the library interface is usable and, compared to the Monotonic State interface, the extra burden to the user is reasonable for the extra functionality. The main two differences are that the user has to specify both the footprint and what gets shared for each call, and to keep track of the labels of the references. Some of the details of the library are explicit in the interface, in particular predicates about the label_map reference, which are not relevant to the user, but the user does not have to deal with them because of F*'s SMT automation. We did not find a way to hide these details

without losing expressiveness. On the other hand, the use of a deep embedding to implement the full ground types—which was necessary to prove preservation of the invariant inside the stateful operations—is not great for SMT automation, as sometimes the user has to help F* figure out what the type is.

4.4 Higher-Order Contracts

As explained in §4.2.3, SecRef* uses higher-order contracts at the boundary between verified and unverified code to ensure properties about returned or passed shareable references that cannot be statically determined. The higher-order contracts are built around two functions, **import** and **export**, that map from *intermediate* types to *source* types, and vice versa (Figure 4.4). They are similar to the higher-order contracts of the SCIO* framework (§3.4), but this time specific to state: the contracts inspect the heap and they are polymorphic in the three abstract predicates. We first introduce intermediate types in §4.4.1, and then explain in §4.4.2 how the **export** and **import** functions are defined.

4.4.1 The Polymorphic and Intermediate Interfaces

The verified program and the unverified code communicate through an interface, which is different in the source language compared to the target language. To represent the interface of unverified code as presented in §4.2.2, we define in Figure 4.6 the type class **poly_iface**, parameterized by three abstract predicates (**threep**). The types that can appear on the interface of unverified code are thus determined by the instances we provide for the **poly_iface** class. We provide instances for base types, sums, products, references, and *arrow types*. The instance for arrows is defined so that **poly_iface** can be used to describe higher-order interfaces (e.g., **poly_student_hw** from §4.2.2). Notice that the instances for references and linked lists take a full ground type (§4.3.2). All types that have a representation as a full ground type can be characterized by the **poly_iface** class.

Having defined polymorphic interfaces necessary to represent unverified code, we define intermediate interfaces as polymorphic interfaces instantiated with some concrete predicates, and intermediate types as instances of such interfaces. Because we represent unverified code using polymorphism (§4.2.2), our higher-order contracts

```
type threep = 𝓘:(heap → Type₀) & φ:(#a:_ → #rel:_ → mref a rel) & ≼:(preorder heap)

class poly_iface (preds:threep) (t:Type) = { wt : every_mref_tc t }
instance poly_iface_int preds : poly_iface preds int
instance poly_iface_sum preds a b {|poly_iface preds a|} {|poly_iface preds b|}:poly_iface preds (either a b)
instance poly_iface_ref preds (t_emb:full_ground_typ) : poly_iface preds (ref (to_Type t_emb))
instance poly_iface_arrow 𝓘 φ (≼) a b
 {| c1:poly_iface (𝓘, φ, ≼) a |}
 {| c2:poly_iface (𝓘, φ, ≼) b |}
 : poly_iface preds
    (x:a → MST b
     (requires (λ h₀ → 𝓘 h0 ∧ c1.wt.every_mref x φ))
     (ensures  (λ h₀ r h₁ → 𝓘 h1 ∧ h0 ≼ h1 ∧ c2.wt.every_mref φ r)))
```

Figure 4.6: Showing the polymorphic interface implemented with type class **poly_iface**. Selected instances.

have to be enforced on functions under polymorphism as well, just as in our setting for IO the contracts are enforced under flag-based effect polymorphism (§3.2.4), which means that our contracts do not know what they enforce until the very end, when the type is instantiated and the actual checks are provided.

4.4.2 Exporting and Importing

The functions **export** and **import** are implemented using two type classes, named **exportable_from** and **importable_to**. These are the stateful counterparts of the classes carrying the same names in the SCIO* framework (§3.4.2). To build an instance for **exportable_from**, one has to provide a function from a strong type (**styp**) to an intermediate type (**ityp**), and a proof that it preserves the $\textbf{every}_{\text{mref}}$ combinator. Such a proof is easy to provide because a reference is not affected by the **export** function since its type is restricted to be a full ground type—i.e., exporting a reference is identity. Its dual, **importable_to** is similar, but it has to account for potential breaches of contract by allowing **import** to fail with an error. We also provide a **safe_importable_to** class for types for which importing always succeeds.

To be able to export/import functions that are polymorphic in the three predicates (**threep**), the type classes are parameterized with some implementation of the three predicates. The **specs_of_styp** is just an inductive type that semi-syntactically represents the pre- and postcondition that appear on type **styp**, structured as a tree. It allows us to be able to provide the contracts at importing/exporting time, when the three predicates become concrete. The functions **export** and **import** take the contracts as a tree of type **hoc_tree specs_of_styp**, which has the same structure as **specs_of_typ**, and its type guarantees that each pre- and postcondition that has to be enforced has a contract.

```
class exportable_from (preds:threep) (styp:Type) (specs_of_styp:spec_tree preds) = {
  c_styp : every_mref_tc styp;
  ityp : Type;
  c_ityp : poly_iface preds ityp;
  export : styp → hoc_tree specs_of_styp → ityp;
  export_preserves_φ : x:styp → φ:_ → hocs:_ →
    Lemma (every_mref φ x) (ensures (every_mref φ (export x hocs))); }

class importable_to (preds:threep) (styp:Type) (specs_of_styp:spec_tree preds) = {
  c_styp : every_mref_tc styp;
  ityp : Type;
  c_ityp : poly_iface preds ityp;
  import : ityp → hoc_tree specs_of_styp → either styp err;
  import_preserves_φ : x:ityp → φ:_ → hocs:_ →
    Lemma (every_mref φ x) (ensures (every_mref φ (import x hocs))); }
```

For intermediate types (and thus already in the source language), these type classes are trivially instantiated with identities, and for operators such as **option**, one proceeds recursively. For instance, the **export** function for **option a** when a is itself exportable will *map* the **export** function for **a**. What is more interesting is importing and exporting functions that may have pre- and postconditions.

Exporting Functions. We provide a general instance to export a function **f** that has source type **a → MST (either b err) pre post** to a function of the intermediate type **a' → MST (either b' err) pre' post'**, where **a'** and **b'** are the intermediate counterparts of **a** and **b**, and **pre'** and **post'** are the pre- and postcondition that appear in

poly_iface_arrow in Figure 4.6. For this, we need an import function (a' → a) for type a and an export function (b → b') for type b, which we then compose with f. To be able to call f, we have to ensure that pre holds, which we cannot do in general without a dynamic check, thus, the instance requires the existence of a check function which approximates the precondition:

```
check : x:a → MST (either unit err)
  (requires (pre' x))
  (ensures  (λ h0 r h1 → h0 == h1 ∧ (Inl? r ⟹ pre x h0)))
```

which either returns an error or succeeds by enforcing pre. Notice that check may safely assume pre' as it holds when it is called. Finally, one has to ensure that post' holds, for which we require that the original postcondition of f satisfies the following property: $\forall$ x h_0 r h_1. post x h_0 r h_1 ⟹ post' h_0 r h_1. One detail that we omitted is that pre' and post' are polymorphic in the type of x and r, compared to pre and post that use the types a and b. Therefore, because we call import and export, we also have to prove that pre' x ⟹ pre' (import x) and post' h_0 r h_1 ⟹ post' h_0 (export r) h_1, which we do by using the two lemmas included in the type classes, export_preserves_φ and import_preserves_φ.

Importing Functions. Similarly, we can import a general function f of the following intermediate type x:a' → MST (either b' err) pre' post' to the following source type x:a → MST (either b err) pre post. Because the type already accounts for errors, we use the safe_importable_from class to avoid duplicating errors. This time we export the input from a to a', call the function, and import the output from b' to b. To be able to call the function f, we need to enforce pre' using pre, hence we only provide an instance under the assumption $\forall$ x h_0. pre x h_0 ⟹ pre' x h_0. Similarly, once the function is run, we only know that post' is verified, so we also require $\forall$ x h_0 e h_1. pre x h_0 ∧ post' h_0 (Inr e) h_1 ⟹ post x h_0 (Inr e) h_1, which ensures we do preserve the postcondition of the error case. Often this will be discharged automatically by having the postcondition trivially true for errors. The more interesting part is when the function succeeds, and we need to verify the postcondition. For this, the instance also requires a stateful contract [15], see select_check below:

[15]: Findler et al. (2002), *Contracts for higher-order functions*

```
type cb_check a b {| every_mref_tc b |} pre post x eh =
  r:b → MST (either unit err)
    (requires (λ h1 → post' eh r h1))
    (ensures  (λ h1 rck h1' → h1 == h1' ∧ (Inl? rck ⟹ post x eh r h1)))

select_check : x:a →
  MST (eh:erased heap {pre x eh} & cb_check a (either b err) pre post x eh)
    (requires (pre x))
    (ensures  (λ h0 r h1 → reveal r._1 == h0 ∧ h0 == h1))
```

A stateful contract is first run before calling f to read from the heap information necessary to enforce the postcondition, which it can store in the closure, and then the callback of the contract is run after f returns to verify the postcondition post, or return an error otherwise. The same_values predicate from the autograder example (Figure 4.1) is enforced by SecRef* using a stateful contract.

```
let J_c = Ir_inv (* from §4.3.3. The three concrete predicates, defined as in §4.2.2.*)
let φ_c = λ r → witnessed (is_contained r) ∧ witnessed (is_shareable r)
let ≼_c = λ h_0 h_1 → modif_shareable_and_encaps h_0 h_1 ∧ same_labels h_0 h_1

type interface^S = {
  ctype : threep → Type;
  specs_of_ctype : threep → spec_tree threep;
  hocs : hoc_tree (specs_of_ctype (J_c, φ_c, ≼_c));
  imp_ctype : preds:threep → safe_importable_to preds (ctype preds) (specs_of_ctype preds);
  ψ : heap → int → heap → Type_0 }
type interface^T = {
  ctype : threep → Type;
  interm_ctype : preds:threep → poly_iface preds (ctype preds); }
let compile_interface (I^S:interface^S) : interface^T = { (** denoted by I^S↓ **)
  ctype = (λ preds → (I^S.imp_ctype preds).ityp);
  interm_ctype = (λ preds → (I^S.imp_ctype preds).c_ityp); }

type prog^S I^S = I^S.ctype (J_c, φ_c, ≼_c) → LR int ⊤ I^S.ψ
type ctx^S I^S = I^S.ctype (J_c, φ_c, ≼_c)
type whole^S = ψ : post_cond & (unit → LR int ⊤ ψ)
let link^S #I^S (P:prog^S I^S) (C:ctx^S I^S) = (| I^S.ψ, λ () → P C |) (** denoted by C[P] **)

type prog^T I^T = I^T.ctype (J_c, φ_c, ≼_c) → LR int ⊤ ⊤
type ctx^T I^T = J:_ → φ:_ → ≼:_ →
  poly_alloc_t J φ (≼) → poly_read_t J φ (≼) → poly_write_t J φ (≼) → I^T.ctype (J, φ, ≼)
type whole^T = unit → LR int ⊤ ⊤
let link^T #I^T (P:prog^T I^T) (C:ctx^T I^T) =
  λ () → P (C J_c φ_c (≼_c) ctx_alloc ctx_read ctx_write) (** denoted by C[P] **)

let compile_prog #I^S (P:prog^S I^S) : prog^T (I^S↓ ) = (** denoted by P↓ **)
  λ C → P ((I^S.imp_ctype (J_c, φ_c, ≼_c)).safe_import C I^S.hocs)
let back_translate_ctx #I^S (C:ctx^T I^S↓ ) : ctx^S I^S = (** denoted by C^T↑ **)
  (I^S.imp_ctype (J_c, φ_c, ≼_c)).safe_import (C (J_c, φ_c, ≼_c)) I^S.hocs
```

Figure 4.7: The SecRef* secure compilation framework. Idealized mathematical notation. Type **threep** is presented in Figure 4.6. Types poly_alloc_t, poly_read_t, and poly_write_t are presented in Figure 4.2.

4.5 SecRef*: Formally Secure Compilation Framework

In this section we present the formalization of SecRef* (Figure 4.7). The formalization is mechanized in F*, in the style of the SCIO* framework (§3.6.1), and instantiates the compiler model presented by Abate et al. [8] with source and target languages that are shallowly embedded. We start by presenting the source (§4.5.1) and target language (§4.5.2). We continue by giving a semantics to them (§4.5.3). Then, we prove soundness and RrHP by carefully defining compilation and back-translation (§4.5.4). For all of these, we first present the setting when the verified program has initial control, however, SecRef* is verified to be secure also when the unverified code has initial control (also §4.5.4).

[8]: Abate et al. (2019), *Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation*

4.5.1 Source Language

As we did for SCIO* (§3.6.1), we make a clear split between the verified program and the code it is linked with. In the source language, we refer to the former as a partial source program, and the latter as a source context. The latter is also a verified piece of code, but for secure compilation we think of it as a target context (aka unverified) that was back-translated (the dual of compilation) to a source context (aka verified).

The partial source program and context communicate through a higher-order interface of type $\mathbf{interface}^S$, which contains a type ($\mathbf{ctype}$), a type class constraint on this type ($\mathbf{imp\_ctype}$), and a postcondition ψ. As we want the type $\mathbf{ctype}$ to be polymorphic in the three predicates in our source language, so as to obtain a polymorphic type in the target language, we include the fields $\mathbf{specs\_of\_ctype}$ and $\mathbf{hocs}$ to be able to apply higher-order contracts to a polymorphic type as explained in §4.4.2. The representation of the partial source programs ($\mathbf{prog}^S$) is a computation in the monadic effect LR that takes as argument a value of type $\mathbf{ctype}$ instantiated with the concrete predicates, returns an integer, and ensures the postcondition ψ. The representation of source contexts ($\mathbf{ctx}^S$) is just a value of type $\mathbf{ctype}$ instantiated with the concrete predicates. Source linking ($\mathbf{link}^S$) is then defined just as function application. A whole source program is a dependent pair between a postcondition and a computation in effect LR from unit to integer that guarantees the postcondition.

The type class constraint $\mathbf{imp\_ctype}$ guarantees that the specification on the interface can be soundly enforced by SecRef*. The assumptions made about the unverified code appear on this source interface (e.g., the assumptions the autograder makes about the homework in Figure 4.1), and $\mathbf{imp\_ctype}$ ensures that they are one of the two types of assumptions that SecRef* supports (§4.2.1).

4.5.2 Target Language

The formalization of the target language is similar to the source language, with two notable differences: the representation of partial target programs ($\mathbf{prog}^T$) does not require the computation to be verified with respect to a postcondition, since it has the trivial postcondition; and target contexts ($\mathbf{ctx}^T$) use the representation from §4.2.2. The type $\mathbf{ctype}$ in the target interface (of type $\mathbf{interface}^T$), now is constrained with the $\mathbf{poly\_iface}$ type class from §4.4.1.

A partial target program (see $\mathbf{prog}^T$) expects the context to satisfy the concrete specification it assumes about the context: to only modify shareable and encapsulated references. Therefore, linking is done at the types of the target language, but after linking, the communication between the program and the context happens at an

intermediate type that contains specifications. This can be seen in the definition of target linking ($\mathbf{link}^T$). It instantiates the target context with the concrete predicates $\mathcal{I}_c$, φ_c, and $\preccurlyeq_c$, thus strengthening the type of the context.

During linking, the concrete implementations of the operations **alloc**, **read**, and **write** are provided. As explained in §4.2.2, their definitions are straightforward (specifications and proofs elided):

```
let ctx_alloc init = let r = lr_alloc init in label_shareable r; r
let ctx_read = lr_read
let ctx_write = lr_write
```

The representation of target contexts is a reasonable representation for unverified code—logical reasoning is not necessary to prove the pre- and postconditions that use the abstract predicates. As noted in §4.2.2, we deeply embed a total simply typed language with first-order ML-style references, and provide a translation from any typed expression in this language to a context of type **ctype** to the representation of target contexts $\mathbf{ctx}^T$. The translation is defined structurally recursively on the typing derivations. Formalizing this translation revealed multiple edge cases in higher-order settings, and also the fact that monotonic references cannot be shared with unverified code. This gives us sufficient confidence that our representation is reasonable, in addition to multiple examples we implemented using this representation in the artifact (e.g., the homework in §4.2.2).

4.5.3 Semantics

Our source and target languages are shallowly embedded. Since whole source and target programs are computations in the LR effect, we can define the same semantics for both. We define this semantics as predicates over the initial heap, the final result (an integer), and the final heap.

```
type state_sem = heap → state → heap → Type0
let (⊆) (s1 s2:state_sem) = ∀ h0 r h1. s1 h0 r h1 ⟹ s2 h0 r h1
```

To define the semantics of whole programs, we must first reveal the monadic representation of LR using the **reify** construct provided by F⋆ [3]. We define the semantics function **beh** by adapting the monad morphism θ from §2.5.2 [9], as we also did for IO in §3.6.2.

[3]: Rastogi et al. (2021), *Programming and Proving with Indexed Effects*

```
let beh_mst (m:mst_repr int) : state_sem = λ h0 r h1 → ∀ p. θ m p h0 ⟹ p r h1
let beh (W:unit → LR int ⊤ ⊤) = beh_mst (reify (W ()))
```

Since we use θ for both the Dijkstra monad and our semantics function, we can easily turn the intrinsic property that a whole source program satisfies a postcondition ψ into an extrinsic property:

Theorem 4.5.1 (Soundness of whole source programs)

$$\forall(\psi, W) : \mathrm{whole}^S.\ \mathrm{beh}(W) \subseteq \psi$$

Proof sketch. For a whole source program, we know its underneath monadic representation is m:(mst_repr int){θ m ⊑ (λ p h_0 → ∀ r h_1.ψ h_0 r h_1 ⟹ p r h_1)} (§2.5.2), which implies that beh m ⊆ ψ. □

The proof is so straightforward because the type of a whole source program contains an intrinsic specification. We just lift the specification to an extrinsic property of the whole program. Intuitively, the proof that the program satisfies the specification was already done modularly by F* typing.

The semantics function is written in the same language as the code it reasons about, meaning that the semantics a program is given depends on F*'s consistency. Therefore, as part of our TCB we include the following F* features that SecRef* relies on: F*'s type system, the effect mechanism (which has a paper proof), the axiomatized Ghost effect, and the type abstractions provided by the module system (e.g., the implementation of references is hidden behind a module interface).

4.5.4 Soundness and Robust Relational Hyperproperty Preservation (RrHP)

In our setting for IO we showed (§3.6.3 and §3.6.4) that soundness and RrHP can be proved using a Syntactic Inversion Law, informally, *Compilation+Target Linking=Back-translation+Source Linking*. Such a strong result is achievable because we are working with shallow embeddings, we take advantage of polymorphism to model the context, and we implement higher-order contracts using two dual functions. We managed to adapt our earlier compilation model in our setting for state, and to prove the law and the two criteria by following the same proofs.

We define the compilation of partial programs (**compile_prog** in Figure 4.7, denoted as $P\downarrow$) as wrapping the partial source program in a new function of type $\mathbf{prog}^T$. The partial target program takes as argument an instantiated target context whose specification states that it modifies only shareable references. Therefore, to enforce the other assumptions made by the partial source program, SecRef* adds to the compiled program the higher-order contracts using **safe_import**.

We define the back-translation of target contexts (denoted by $C\uparrow$) as instantiating the context and then adding the higher-order contracts to produce a source context. One immediately observes that this is exactly what happens during compilation plus target linking, and since source linking is just function application, it means syntactic inversion law can be proved just by unfolding definitions.

Theorem 4.5.2 (Syntactic inversion law)

$$\forall I^S.\, \forall C^T : \mathrm{ctx}^T\ I^S\downarrow .\, \forall P : \mathrm{prog}^S\ I^S.\, C^T[P\downarrow] = C^T\uparrow [P]$$

SecRef* allows one to statically verify that a partial program satisfies a postcondition ψ. We prove a soundness theorem that guarantees that compiling the partial program and then linking it with arbitrary unverified code produces a whole target program that also satisfies the postcondition ψ (when the partial program has initial control).

Theorem 4.5.3 (Soundness)

$$\forall I^S.\, \forall P : \mathrm{prog}^S\ I^S.\, \forall C^T : \mathrm{ctx}^T\ I^S\downarrow .\, \mathrm{beh}(C^T[P\downarrow]) \subseteq I^S.\psi$$

Proof sketch. We rewrite the goal with Theorem 4.5.2 to $C^T[P\downarrow] = C^T\uparrow[P]$. Now, $C^T\uparrow[P]$ builds a whole source program and we can apply Theorem 4.5.1. □

We prove that SecRef* robustly preserves relational hyperproperties. Abate et al. [8] have this as their strongest secure compilation criterion, and they show it is stronger than full abstraction.

Theorem 4.5.4 (Robust Relational Hyperproperty Preservation (RrHP))

$$\forall I^S.\, \forall C^T : \mathrm{ctx}^T\ I^S\downarrow .\ \exists C^S : \mathrm{ctx}^S\ I^S.\, \forall P : \mathrm{prog}^S\ I^S.\, \mathrm{beh}(C^T[P\downarrow]) = \mathrm{beh}(C^S[P])$$

Proof sketch. The proof follows by back-translating the target context ($C^T\uparrow$) and using it to instantiate the existential of a source context, after which we conclude with Theorem 4.5.2. □

Finally, we have proved that SecRef* also satisfies RrHP (Theorem 4.5.4) in two other settings, the same two as in our setting for IO (§3.6.5 and §3.6.6). **The dual setting: when the context has initial control.** In this setting, the proofs are similar, since Theorem 4.5.2 again holds. Instead of soundness, we proved the dual of soundness, a less interesting criterion stating that the resulting whole target program modifies only shareable references. As a sanity check that our shallow embedding of unverified code is correct, we also proved RrHP in **the setting where target contexts are represented with typed syntactic expressions** of the deeply-embedded language mentioned in §4.2.2, a case in which target linking and back-translation first do the extra step of translating the expression into the representation for target contexts (ctx^T). The proofs can be found in the artifact.

4.6 Case Study: Simple Stateful Cooperative Multi-threading

To test our framework further, we verify a simple stateful cooperative multi-threading case study. In this case study, we are particularly interested in showcasing how scheduling properties can be verified, even when the verified scheduler is orchestrating tasks that are themselves unverified.

Our model of cooperative multi-threading is inspired by the work of Dolan et al. [132] in the context of OCaml, where effect handlers are used to program concurrent systems. In this model, running threads are represented as computation trees. A computation tree is either a final value or a computation yielding control to the next task. When yielding control, the continuation is a stateful computation that returns a new computation tree. We can model these trees as follows:

[132]: Dolan et al. (2017), *Concurrent System Programming with Effect Handlers*

```
type atree (a:Type0) =
  | Return : a → atree a
  | Yield : continuation a → atree a

and continuation a =
  unit → LR (atree a)
    (requires (λ h0 → ⊤))
    (ensures  (λ h0 _ h1 → h0 ≼c h1)
```

Continuations represent unverified code, and their specification consists of a trivial precondition and a postcondition stating the unverified code only modifies shareable (or encapsulated) references using the relation $\preccurlyeq_c$ from Figure 4.7. As before, we omit reference containment requirements from pre- and postconditions, along with fuel parameters used for proving termination.

We model the scheduler state using a private reference, which stores (i) an execution history (a list of thread IDs), (ii) the ID of the next thread to be executed, and (iii) a count of the number of finished threads (to which we refer as *inactive*). We refine the scheduler state by adding a simple fairness property: each thread must have taken either ticks or ticks + 1 steps, ensuring fair progress among all threads (with ticks logically incremented on each round of the round-robin progress):

```
let fairness_ticks (limit : int) (ticks : nat) (hist : list int) (next : int) = ...

let fairness (limit : int) (hist : list int) (next : int) =
  ∃ (ticks:nat). fairness_ticks limit ticks hist next
```

In the refined scheduler state, we also ensure that both the next thread and inactive counts are within valid ranges. The argument k is used to keep track of the number of threads passed to the scheduler on initialization. In detail, we define it as follows:

```
type count_st k =
  hist:(list int) & next:int & inact:int{fairness k hist next ∧ 0 ≤ next < k ∧ 0 ≤ inact ≤ k}

let counter_preorder k : preorder (count_st k) = λ v v' → v.1 `prefix_of` v'.1

let counter_t k = mref (count_st k) (counter_preorder k)
```

The scheduler is implemented as a structurally recursive function with the following type:

```
val scheduler (r:ref int) (tasks:list (continuation unit)) (counter:counter_t (length tasks))
: LR unit
    (requires (λ h0 → is_private counter h0 ∧ is_shareable r h0))
    (ensures  (λ h0 _ h1 →
      modif_shareable_encaps_and h0 h1 !{counter} ∧ gets_shared empty h0 h1))
```

It takes a global shared reference r to an integer (which can be accessed by any of the involved stateful computations), a list of tasks to be executed (represented as unit-typed continuations), and the reference to the current state of the scheduler (which is required to be private in the precondition). On each iteration, the scheduler picks the next task to be run, runs it, and acts depending on whether the outcome was a final value or a computation yielding control. We are guaranteed that at each step we run a next task from the given task list and only shared references are modified. This invariant allows us to verify the fairness property of the scheduler's state, as we know that the counter reference was not modified by the tasks. In comparison to the postcondition of unverified code, notice that the postcondition for the scheduler

ensures that no labels are modified, but we do not claim to have only modified shared and encapsulated variables: the scheduler state gets updated as the scheduler runs the tasks.

Building on the scheduler implementation, we can write a function to run a list of computations.

```
let run (v:int) (tasks:list (ref int → continuation unit){length tasks > 0}) :
  LR int
    (requires (λ h0 →
      forall_refs_heap contains_pred h0 v ∧ forall_refs_heap is_shareable h0 v))
    (ensures  (λ h0 _ h1 → h0 ≼c h1)) =

    let counter : counter_t (length tasks) = alloc (| [], 0, 0 |) in
    let r = alloc v in
    label_shareable r;
    let tasks = map (λ (f : ref int → continuation unit) → f r) tasks in
    scheduler r tasks counter;
    !r
```

Here we allocate two references: one that is shared by the different tasks (and thus it is labeled shareable) and one that is used to store the scheduler's state (and which is kept private). We then run the computations using the scheduler and in the end return the final value of the shared reference. The scheduler starts with an empty history, the first task selected, and zero inactive tasks. After its execution, the final history satisfies the fairness predicate, indicating that all tasks took the same number of steps. However, this fact is not made explicit in the postcondition of run, which only focuses on the references modified during the execution. As a final remark, we note that the function run can also be called by unverified code to cooperatively execute other unverified code. Importantly, run has the exact same postcondition as unverified code (as used in the continuations).

4.7 Related Work

Verification of Stateful Code. Labeled References are encoded in F⋆ on top of Monotonic State, and this enables specifying the footprint of a call using labels, which allows us to tame the complexity of globally tracking the dynamically shared references. As explained in §4.2.1, the usual way of specifying the footprint of a call, by providing the precise set of references that gets modified [10, 23, 127], does not scale to higher-order unverified code when dynamic sharing of references is allowed. This approach enabled verification of stateful F⋆ programs linked with unverified code. Similar approaches [24, 25] were used to build logics on Iris [133] for verifying such programs, with which we compare next.

Capabilities and Robust Safety. Swasey, Garg, and Dreyer [24] introduce OCPL, a logic for verifying object capability patterns. It can be used to verify concurrent, non-terminating, stateful programs that are fully higher-order, including support for higher-order references. OCPL has a notion of high and low references, where high references can be wrapped in closures that are passed to the untrusted code, while low references are considered shared between the trusted and untrusted code.

In OCPL, low references can only point to low references, and callbacks that are crossing the verified-unverified boundary can take as argument and return only low references. This notion of high and low references seems to correspond to the private/encapsulated and shareable labels in Labeled References. Swasey, Garg, and Dreyer [24] prove a robust safety theorem stating that code verified with OCPL can be safely linked with arbitrary unverified code. They then use OCPL to verify the robust safety of using wrappers like dynamic sealing, the caretaker pattern and the membrane pattern. By comparison, SecRef* is a secure compilation framework using higher-order contracts that dynamically enforce specifications about shared references. Soundness of SecRef* (Theorem 4.5.3) intuitively provides similar guarantees to OCPL's robust safety theorem. SecRef* additionally satisfies RrHP (Theorem 4.5.4), the preservation of a large class of properties including robust safety.

Georges et al. [26] propose Cerise: a program logic for reasoning about a low-level capability machine. It guarantees robust safety of verified code that interacts with unverified code in multiple scenarios: when the verified-unverified code share statically allocated memory, when they use wrappers in the style of Swasey, Garg, and Dreyer [24], and when the unverified code can allocate dynamically its own memory. To obtain safety in the last scenario, they use activation records, a technique involving saving the content of the references that are not in the call's footprint, and restoring them after the call, but it is done dynamically and adds an extra cost. In our work, unverified code can also allocate its own memory dynamically, and we show how to specify the footprint using labels so that there is no need to perform dynamic restoring of references when calling into unverified code.

[26]: Georges et al. (2024), *Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code*

In a setting where, instead of ML-style references, one uses integer pointers that can be dereferenced, Sammler et al. [25] showed that one can enforce robust safety by using sandboxing. They allow sharing dynamically allocated pointers between verified and unverified code under some conditions: they have to clearly distinguish between the *high heap* of the trusted code and the *low heap* of the untrusted code, something we do not have to do because our references are unforgeable.

Secure Compilation. The approaches of Agten, Jacobs, and Piessens [27] and Strydonck, Piessens, and Devriese [105] support secure compilation of formally verified stateful programs against unverified contexts. Agten, Jacobs, and Piessens [27] securely compile modules verified with separation logic [23], but written in the C language, which normally lacks memory safety. To overcome that, they add runtime checks at the boundary between verified and unverified code, and also use two static notions of memory: local, which is never shared with the unverified code, and heap memory, which the unverified code can read and write arbitrarily because of the lack of memory safety. The local memory is protected using hardware enclaves, and if one wants to share data from local memory, one has to copy it to the heap memory. To protect the heap memory when calling into the unverified code, they first hash the complement of the assumed footprint of the call, and when control is returned, they recompute the hash and compare it, to make sure that the unverified code did not change heap locations outside the footprint. What we do is different because we work with memory safe languages with ML-style references, which act like capabilities. We show that if the source and target language have capabilities, one can avoid these expensive hash computations by getting the footprint right. Moreover, we have only one notion of references, which can be kept private or be dynamically shared with the unverified code by using the labeling mechanism.

[27]: Agten et al. (2015), *Sound Modular Verification of C Code Executing in an Unverified Context*

[105]: Strydonck et al. (2021), *Linear capabilities for fully abstract compilation of separation-logic-verified code*

Strydonck, Piessens, and Devriese [105] compile stateful code verified with separation logic to a target language with hardware-enforced *linear* capabilities, which cannot

be duplicated at runtime. The main idea is to require the unverified context to return the linear capabilities back so that the verified code can infer that the context cannot have stored them. Linear capabilities are, however, an unconventional hardware feature that is challenging to efficiently implement in practice [134]. In contrast to their solution, SecRef* provides soundness and security in a target language with ML-style mutable references, which can be implemented using standard capabilities [135], but this required us to overcome the challenge of unverified contexts that stash references.

New, Bowman, and Ahmed [136] propose a fully abstract compiler from a simply typed λ-calculus with unit, sums, pairs, and recursive types to a target language that has additional support for exceptions. Their compiler is not, however, addressing ML-references or verified code as ours does. The full abstraction property is a security criterion that was shown to be implied by the criterion we prove about our compiler, RrHP (Theorem 4.5.4), by Abate et al. [8]. New, Bowman, and Ahmed [136] employ a comparable approach to ours to prove full abstraction, involving a translation between two shallow embeddings.

[136]: New et al. (2016), *Fully abstract compilation via universal embedding*

Gradual Verification is a combination of static and dynamic verification proposed by Bader, Aldrich, and Tanter [121]. For assertions that cannot be proven statically using the existing code annotations, one can opt to let the tool insert dynamic checks that dynamically enforce the assertion. DiVincenzo et al. [137] introduced Gradual C0 to do gradual verification for stateful programs. Gradual C0 requires analyzing the unverified code, while our DSL does not. Gradual C0 requires the unverified code to determine what checks to add, and it can be many, while SecRef* avoids adding checks to preserve logical invariants about private references. SecRef* adds higher-order contracts to enforce pre- and postconditions at the boundary of verified-unverified code. In the absence of unverified code, the dynamic checks are necessary to be able to establish logical predicates about shareable references.

[121]: Bader et al. (2018), *Gradual Program Verification*

[137]: DiVincenzo et al. (2025), *Gradual C0: Symbolic Execution for Gradual Verification*

Secure Interoperability. There is also related work based on typed approaches to safe interoperability between high and low-level code that share references. The main difference with those works is that they look at safe interoperability between languages with different semantics, while we look at code *verified* in the traditional Hoare/separation-logic style and unverified code. Patterson et al. [138] present a multi-language that supports interoperability between a simply typed functional language and a typed assembly language. Their multi-language allows integrating assembly code into the functional language, and reason about their mix. Patterson and Ahmed [139] present the notion of linking types, a more expressive way to annotate the interface, that enables one to reason about behavioral equality using only one language, even though the code may be linked with code written in a different language with more expressive power. Patterson, Wagner, and Ahmed [140] show how to soundly encapsulate foreign code using linked types, so that the foreign code is unobservable and the invariants of the language are undisturbed.

[138]: Patterson et al. (2017), *FunTAL: reasonably mixing a functional language with assembly*

[139]: Patterson et al. (2017), *Linking Types for Multi-Language Software: Have Your Cake and Eat It Too*

[140]: Patterson et al. (2023), *Semantic Encapsulation using Linking Types*

4.8 Additional Example: Guessing Game

To showcase the use of encapsulated references, in Figure 4.8 we present another example of a verified program that plays the “guess the number I’m thinking of” with an unverified player. The verified program gives to the player a range and a callback that the player can use to guess the number. The verified program saves the guesses

of the player using an encapsulated monotonic reference that has a preorder that ensures that no guess is forgotten (line 13). SecRef* does not add any dynamic checks using higher-order contracts during compilation because the postcondition of the player follows from the universal property of unverified code (§4.2.2).

```
1  type cmp = | LT | GT | EQ

3  let post = λ h0 _ h1 → modif_shareable_and_encaps h0 h1 ∧ same_labels h0 h1

5  type player_t =
6    (l:int ∗ r:int ∗ (guess:int → LR cmp (λ _ → ⊤) post)) → LR int (λ _ → l < r) post

8  let play_guess (l pick r:int) (player:player_t) :
9    LR (bool ∗ list int)
10     (requires (λ _ → l < pick ∧ pick < r))
11     (ensures  post) =

13   let guesses : mref (list int) (λ l l' → l `prefix_of` l') = alloc [] in
14   label_encapsulate guesses;
15   let final_guess =
16     player (λ x →
17       guesses := guesses @ [x];
18       if pick = x then EQ else if pick < x then LT else GT) in
19   guesses := guesses @ [final_guess];
20   if pick = final_guess then (true, !guesses)
21   else (false, !guesses)
22
```

Figure 4.8: Example of a function that plays "Guess the number I'm thinking of"

4.9 Summary

In this chapter, we took the ideas from Chapter 3 and adapted them to securing stateful programs against unverified code in F* with strong formal secure compilation guarantees (RrHP). The IO and stateful effects differ because references are unforgeable (and file descriptors are not), references satisfy some laws—e.g., a read always reflects the most recent update, but file descriptors offer no such guarantee—, and state is verified using a relation between initial and final state (instead of traces of events). The result is SecRef*, a secure compilation framework that allows for the sound verification of stateful programs that can be compiled and then linked with arbitrary unverified code, importantly allowing dynamic sharing of mutable references. What we showed is that, compared to SCIO*, in SecRef* if one puts the extra effort into labeling references during verification to know which ones are shared with the unverified code, then a reference monitor is not needed. In the next chapter, we switch back to IO programs, but this time to securely extract them.

5 Secure Extraction of IO Programs

5.1 Contributions

In this chapter, we show how to define extraction in F* while providing formal guarantees. The contributions of this chapter are:

We introduce *relational quotation* (§5.3), a translation validation technique for quoting shallowly embedded programs in a dependently typed language. The idea is to define a typing relation that precisely characterizes the shallowly embedded source language [43], and to use a metaprogram to construct a typing derivation for a given program by following its structure. Checking that a derivation is indeed a valid typing derivation for the original program actually happens by type checking the derivation. This works because the typing relation uses dependent types to guarantee that a derivation corresponds to a specific shallowly embedded program.

[43]: Prinz et al. (2022), *Deeper Shallow Embeddings*

Relational quotation lets us structure certifying extraction in a new way that enables *minimizing* the reliance on translation validation, confining it to the first of two steps, and *verifying* the second once and for all (Figure 5.1). From a valid typing derivation, the second step generates the corresponding target syntax, which we can semantically relate back to the original program because the dependent type of the derivation ties it to that program.

We apply this general idea to build SEIO*, a framework for extracting shallowly embedded F* programs with IO—a language we call IO*—to λ_{io}, a deeply embedded, terminating, simply typed, call-by-value λ-calculus with IO primitives, while providing formal secure compilation guarantees. To relate a simply typed, shallowly embedded IO* program with a deeply embedded λ_{io} one, we use two cross-language logical relations [141, 142]: one that semantically relates the extracted λ_{io} program to the original IO* source program, and one that does the opposite (§5.4).

[141]: Neis et al. (2015), *Pilsner: a compositionally verified compiler for a higher-order imperative language*
[142]: Ahmed (2015), *Verified Compilers for a Multi-Language World*

The logical relations are then used to provide machine-checked proofs about SEIO* for simply typed source programs (§5.5). First, we prove the soundness of the extraction, in the form of compiler correctness: the extracted program has the same behavior as the original verified program. We then prove that SEIO* satisfies Robust Relational Hyperproperty Preservation (RrHP) [8].

[8]: Abate et al. (2019), *Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation*

Finally, we extend SEIO* to support *refinement types* (§5.6), which attach logical constraints to types. This extension involves minimal changes to the typing relation, the verified syntax-generation step, and the secure compilation proof. Extending the relational quotation metaprogram is, however, challenging because of the way F* type-checks refinement types: their logical constraints are transparently collected in a single weakest precondition, which F* discharges using SMT, without producing a

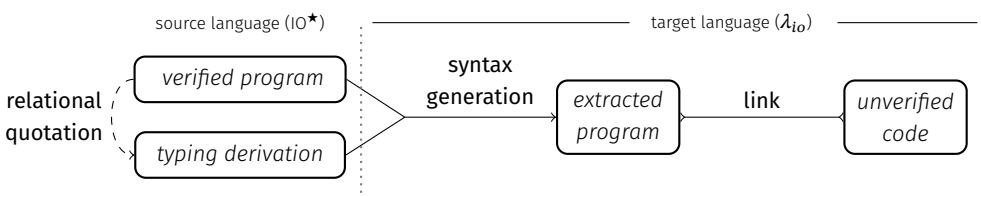


Figure 5.1: An overview of extraction using SEIO*. Solid lines are verified pure F* functions. The dashed arrow represents the relational quotation metaprogram, the result of which is checked by F* to be a typing derivation for the original verified program.

proof term. To work around this, our quotation has to recompute the precondition from scratch during quotation and, to get the formal secure compilation guarantees, reprove it.

Outline. §5.2 starts by illustrating the key ideas of SEIO⋆ on a simple running example. §5.3 details how relational quotation works for IO⋆. §5.4 formalizes the IO⋆ and λ_{io} languages, and the two logical relations between them. §5.5 shows how all the pieces of SEIO⋆ come together and how we use the logical relations to prove that extraction is sound, in the form of compiler correctness, and that it satisfies RrHP. §5.6 shows how to add support for refinements. §5.7 provides more details on our running example, and discusses how we further compile λ_{io} in practice using the Peregrine framework [143]. Finally, we compare to related work in §5.8, and do a summary and discuss future work in §5.9.

[143]: Chapman et al. (n.d.), *Peregrine Project: a unified middle-end for code generation from proof assistants*

Artifact. This chapter comes with an artifact in F⋆ that contains SEIO⋆ and a complete machine-checked proof that it satisfies RrHP. The artifact was developed in two phases. In the first phase, covering the core formalization of SEIO⋆ (§5.2–§5.5), the proofs of some of the low-level lemmas were completed by Claude Opus 4.6+, which is extremely good at doing F⋆ proofs with minimal human guidance [144]. At that point the artifact included over 900 top-level definitions (including type and function definitions, lemmas, and test programs) and for around 80 of them the actual definition was produced by Claude, usually lemmas for which we had written the statement and Claude produced the complete proof including by adding additional lemmas—e.g., proving the running example correct (§5.2.3). The largest such proofs are over 200 lines long, look natural to us, and would have taken us much longer to write. In the second phase, the artifact was extended to support refinement types (§5.6) and Claude was used to update existing lemmas, mainly the compatibility lemmas. Claude also helped on writing the relational quotation metaprogram (§5.3.2). To find the artifact check §1.3, or check the original artifact on Zenodo [145]. The artifact is 15112 lines of F⋆ code[1] containing 1366 top-level definitions, 440 of which are lemmas. The two logical relations and their compatibility lemmas take 6107 lines, 40% of the artifact, and relational quotation a further 2510 lines.

[144]: Swamy (2026), *Agentic Proof-Oriented Programming*

1: Comment and blank lines were not counted towards these numbers.

5.2 Key Ideas

We present the main ideas of SEIO⋆ using the simple running example of a verified wrapper for unverified AI agents. We first give an overview of SEIO⋆ (§5.2.1), then we present relational quotation (§5.2.2), and finally we present what guarantees SEIO⋆ provides when extracting the wrapper (§5.2.3).

```
1 let validate : string → task_t → string → bool = ...

3 let wrapper (f: string) (task: task_t) (agent: string → task_t → io unit) : io (either unit err) =
4   let! contents = read_file f in
5   agent f task;!
6   let! new_contents = read_file f in
7   if validate contents task new_contents
8   then io_return (Inl ())
9   else io_return (Inr Failed_validation)
```

Figure 5.2: Running example: a validation wrapper for unverified agents.

5.2.1 Overview of SEIO*

In Figure 5.2 (lines 3-9) we implement a validation **wrapper** for unverified AI agents. Such a wrapper is useful for tasks where it is easier to check the validity of the result than doing the task itself. An agent is supposed to perform such a task on a file **f**, and the wrapper validates that the agent performed the task correctly. The wrapper saves the initial contents of the file (line 4), then calls the agent (line 5), and finally checks that the task was performed correctly (lines 6-7) using a function **validate**. If calling the wrapper is successful (denoted by returning Inl ()), then the new contents of the file correspond to correctly performing the task on the original file.

The wrapper is written in the **io** monad, which includes operations such as **openfile**, **read**, **write**, **close**, etc. Each of these operations can fail. The function **read_file** used in the Figure 5.2 just opens a file, reads its contents, and closes it back. At this level the agent is modeled as a computation in the **io** monad, but it will actually be implemented in λ_{io}.

The wrapper is shallowly embedded in F* and we want to define extraction such that it produces a syntactic expression in λ_{io}, which is a deep embedding of a call-by-value (CBV) λ-calculus with IO primitives.

```
let extracted_wrapper =
  ELambda (ELambda (ELambda
    eLet (eReadFile (EVar 2)) (
    eLet (eApp 2 (EVar 1) (EVar 3) (EVar 2)) (
    eLet (eReadFile (EVar 4)) (
    EIf (eApp 3 eValidate (EVar 2) (EVar 5) (EVar 0))
      (EInl Ett) (EInr EFailedValidation))))))
```

The **eLet**, **eReadFile**, **eApp x** and **eValidate** are combinators that help with readability.

One can generate **extracted_wrapper** starting from **wrapper** only by using a metaprogram. Existing work on certifying extraction [38–42] would use a metaprogram to generate both **extracted_wrapper** and a proof that the **wrapper** and the **extracted_wrapper** are semantically related, proof that has to be type-checked.

The main insight that distinguishes SEIO* (Figure 5.1) from previous work is that the metaprogram does not have to directly produce a deeply embedded program. Instead, SEIO* uses a metaprogram to produce a typing derivation for the original shallowly embedded **IO*** program, which we call *relational quotation*. Once we validate that the typing derivation is valid for the original program extraction continues with a pure F* function that generates λ_{io} syntax, which we verify once and for all that it produces code guaranteed to be semantically related to the original program. This makes SEIO* rely less on translation validation and more on verification compared with existing work [38–42], and is further detailed in the next subsection.

[38]: Pit-Claudel et al. (2022), *Relational compilation for performance-critical applications: extensible proof-producing translation of functional models into low-level code*

[39]: Myreen et al. (2014), *Proof-producing Translation of Higher-order logic into Pure and Stateful ML*

[40]: Abrahamsson et al. (2020), *Proof-Producing Synthesis of CakeML from Monadic HOL Functions*

[41]: Mullen et al. (2018), *Œuf: minimizing the Coq extraction TCB*

[42]: Forster et al. (2019), *A Certifying Extraction with Time Bounds from Coq to Call-By-Value Lambda Calculus*

5.2.2 Relational Quotation

Relational quotation is a technique to implement quotation for a subset of a dependently typed language that uses translation validation to ensure correctness. Relational quotation involves using dependent types to define a *typing relation* that carves out the subset of the proof-oriented language that represents the shallowly embedded source language. Arbitrary recursive F* functions are outside this subset, but recursive functions over specific inductive types can still be handled by adding explicit iterators. For example, for a simply typed pure language shallowly embedded in F* the typing relation would have the following signature:

```
type typing : Γ:typ_env → a:Type → (eval_env Γ → a) → Type
```

As we are dealing with a shallow embedding, this typing relation is *not* defined over syntactic types and expressions. Instead, the typing relation follows how Prinz, Kavvos, and Lampropoulos [43] do "deeper shallow embeddings" and it relates a typing environment (a partial map from variables to F⋆ types, denoted by Γ), an *F⋆ type* (a), and an *open F⋆ value*, which is a function from an evaluation environment (a map from variables to F⋆ values) to a value of type a.

[43]: Prinz et al. (2022), *Deeper Shallow Embeddings*

To understand why we need to work with open F⋆ values, consider we want to build a typing derivation for the identity function on ints λ (x:int) → x. We need a way to refer to the body of the function, and usually one would just have the variable x as the body, but this is not possible when working with a shallow embedding because we cannot have a free variable x as an F⋆ value. This is why we "open" F⋆ values by assuming that we have an evaluation environment γ that contains the free variables. We do this by representing open values as functions from evaluation environments to values (eval_env Γ → a), where variables are de Bruijn indices, the evaluation environment is guaranteed to contain a value for each entry in the typing environment (because of the type eval_env Γ), and the evaluation environment behaves like a stack, with the usual push, hd and tail operations. For example, for a typing environment Γ = extend int empty, we can represent the body of the identity function as body = λ γ → hd γ. Further, we represent the identity function as λ γ x → body (push γ x), which, by unfolding the body, simplifies to λ γ x → hd (push γ x), which reduces to λ _ x → x, which is syntactically equal to the (thunked) identity function. Open F⋆ values enable us to define typing rules for free variables, which is key to define a typing relation that supports lambda abstractions and characterizes a shallowly embedded programming language.

Our typing rules resemble standard typing rules, but instead of syntax expressions, they operate on F⋆ types and open F⋆ values. For example, the rule for introducing the literal false (Qfalse), types F⋆'s false as a bool, where bool is an F⋆ type.

```
type typing : Γ:typ_env → a:Type → (eval_env Γ → a) → Type =
| Qfalse : #Γ:typ_env →
  typing Γ bool (λ _ → false) (* this is F*'s bool and false *)
| QAxiom : #Γ:typ_env → #a:Type →
  typing (extend a Γ) a (λ γ → hd γ)
| QWeaken : #Γ:typ_env → #a:Type → #b:Type → #x:(eval_env Γ → a) →
  typing Γ a x →
  typing (extend b Γ) a (λ γ → x (tail γ))
| QApp : #Γ:typ_env → #a:Type → #b:Type →
  #f:(eval_env Γ → a → b) → #x:(eval_env Γ → a) →
  typing Γ (a → b) f →
  typing Γ a x →
  typing Γ b (λ γ → (f γ) (x γ))
| QLambda : #Γ:typ_env → #a:Type → #b:Type → #body:(eval_env (extend a Γ) → b) →
  typing (extend a Γ) b body →
  typing Γ (a → b) (λ γ x → body (push γ x))
| QInl : #Γ:typ_env → #a:Type → #b:Type → #v:(eval_env Γ → a) →
  typing Γ v →
  typing Γ (either a b) (λ γ → Inl (v γ))
... (* Qtrue, QIf, Qtt, QInr, QCase, etc., are similarly defined *)
```

The rule QAxiom introduces the most recently bound variable (index 0) by projecting the head of the evaluation environment. QWeaken weakens the environment by discarding the most recent binding via tail and recursing on the shorter environment.

The QApp rule types F⋆'s function application by requiring that f and x are well-typed in the same environment and producing the open value $\lambda\ \gamma \rightarrow (f\ \gamma)\ (x\ \gamma)$, which applies the function to its argument pointwise over the environment. The QLambda rule requires typing the body in an environment extended with a fresh binding of type a. The resulting open value $\lambda\ \gamma\ x \rightarrow \text{body}\ (\text{push}\ \gamma\ x)$ pushes the argument onto the evaluation environment before evaluating the body.

For the simplicity of exposition, the typing rules above were given for simply typed pure F⋆ code. To also support typing of code that uses the io monad, we actually follow the typing rules of the fine-grained call-by-value λ-calculus (FGCBV) [146], which makes a clear distinction between values and potentially effectful computations. Intuitively, values are terms in introduction forms (which do not evaluate further by themselves, such as λ–abstractions, Boolean values, and pairs of values), and computations contain different kinds of elimination forms (such as function application, conditional statements, and pattern matching), the monadic operations of returning values and sequentially composing multiple computations, and file operations. The typing relations for values and computations are then defined by mutual induction, and have the following signatures:

[146]: Levy et al. (2003), *Modelling environments in call-by-value programming languages*

```
type typing : Γ:typ_env → a:Type → (eval_env Γ → a) → Type
type typing_io : Γ:typ_env → a:Type → (eval_env Γ → io a) → Type
```

We present the full definition of these typing relations in Figure 5.3 in §5.3.1.

We call the language carved out by these typing relations IO⋆. Next, we present a typing derivation for the wrapper that proves that it is written in IO⋆.[2] As wrapper is a closed program, the typing judgement is applied to the empty typing environment.

2: qLet, qReadFile, qValidate, qApp x and qVar x are helper functions to simplify readability.

```
let wrapper_typing : typing empty _ (λ _ → wrapper) =
 QLambda (QLambda (QLambdaIO
   qLetIO (qReadFile (qVar 2)) (
   qLetIO (QAppIO (QApp (qVar 1) (qVar 3)) (qVar 2)) (
   qLetIO (qReadFile (qVar 4))
   QIfIO (qApp 3 qValidate (qVar 2) (qVar 5) QAxiom)
     (QReturn (QInl Qtt))
     (QReturn (QInr QFailedValidation))))))
```

There are three important things to notice about this derivation. First, the type of the derivation is typing empty _ (λ _ → wrapper), which directly relates the derivation with the original wrapper. Second, we do not have to spell out the implicit arguments for typing environments, types and open F⋆ values/computations (remember that in F⋆ definitions such as typing such implicit arguments are preceded by #). Third, the structure of the derivation precisely follows the structure of wrapper.

To type check such a derivation, we rely on F⋆ inferring all the implicit arguments, which given the structure of the typing relation F⋆ should always be able to do. The intermediate typing environment and F⋆ values in the derivation are inferred with help from the top-level type we specify for the whole derivation (e.g., if QInl Qfalse is typed as typing empty (either bool unit) (λ _ → Inl false), then it gets elaborated to QInl #empty #bool #unit (Qfalse #empty)). Since we work with simple types, it should always be possible to infer these implicits. The more complex implicits for open F⋆ values can be inferred because of the structure of the typing rules that propagates them bottom-up. The bottom-up propagation of open values *constructs* a program that at the end has to be *equal* to the wrapper, and only if that is the case does the derivation type-check. F⋆ can *unify* the two programs because the derivation follows the syntactic structure of the program, and because the passing of the evaluation environment around gets simplified away. We already saw how the simplification

happens when we looked at the identity example: during unification, expression such as tail (push x γ) and hd (push x γ) are simplified to γ and x. The evaluation environment can always be simplified away when one tries to type a (closed) F* value because the empty environment is used.

Relational quotation is a form of translation validation because the derivation has to be type-checked and we cannot formally guarantee that the derivations generated by the metaprogram will type-check (without verifying both the metaprogram, which includes the standard F* quotation, and the F* type-checker itself). This doesn't increase the TCB, since the F* type-checker is already trusted for verifying the original shallowly embedded programs.

The metaprogram necessary to produce a typing derivation is rather simple (§5.3.2). It takes an F* program (e.g., wrapper), quotes it using standard F* quotation, and produces a typing derivation (e.g., wrapper_typing) by following the structure of the program and applying the corresponding typing rules. The fact that we did not have to spell out the implicits for the manual derivation of the wrapper above is also true for the metaprogram, since the implicits get instantiated automatically when type-checking the derivation. Just by type-checking, the derivation is checked to be a valid typing derivation for the original program, which is a *syntactic* match.

It is great that relational quotation needs a simple metaprogram because this metaprogram is not verified. Previous work [38–42] on certifying extraction produces both a deep embedding and a proof. Moreover, at first sight, wrapper_typing looks similar to extracted_wrapper, but we deal with two different languages: IO* is a FGCBV language, while λ_{io} is a CBV language. Previous work deals with the complexity of *semantically* relating two different language semantics during translation validation, while in SEIO* this only happens in the verified syntax generation step (§5.5).

5.2.3 Formally Secure Extraction from IO* to λ_{io}

From the typing derivation, it is very easy to produce extracted_wrapper. SEIO* has a pure F* function that goes recursively over the derivation and produces the expected λ_{io} program. Each constructor of the typing relation corresponds directly to a syntactic form in λ_{io}: e.g., QBind maps to a let-binding, QOpenfile to a call to the openfile primitive, and QLambda / QLambdaIO to lambda abstractions.

To relate IO* programs with λ_{io} programs, we give trace-producing semantics to IO* and λ_{io}, meaning that for each IO operation, we associate an event. The events include both the argument and the result of an operation. The events get appended in order to build a local trace. A possible successful local trace for the wrapper is the following, where fd and fd' are universally quantified, but guaranteed to be fresh.

```
[EvOpenfile f (Inl fd); EvRead fd (Inl contents); EvClose fd;  (** first read_file **)
 ... agent's trace ...;
EvOpenfile f (Inl fd'); EvRead fd' (Inl new_contents);  (** second read_file **)
EvClose fd']
```

We use the trace-producing semantics of IO* to verify the wrapper, since it allows us to write postconditions over a final result and a local trace of events. We verify the wrapper to satisfy a safety property in arbitrary contexts: which states that for any file, task and agent, if the wrapper runs successfully (the Inl case), looking at the trace of events, the contents last read from the file corresponds (according to the validate function) to performing the task on the contents first read from the file.

```
let wrapper_correct f task agent =
 {|T|} wrapper f task agent {| λ r lt →
   Inl? r ⟹ validate (fst_read_from f lt) task (last_read_from f lt) |}
```

Since SEIO⋆ guarantees that whenever extraction succeeds it preserves safety against adversarial contexts we can get that the **extracted_wrapper** also satisfies this safety property when it is linked with arbitrary unverified agents.

5.3 Relational Quotation of IO⋆

This section details how relational quotation works for IO⋆. We first present the typing relation that carves out the IO⋆ subset of F⋆ (§5.3.1), and then describe the metaprogram that generates typing derivations (§5.3.2).

5.3.1 Typing Relation

The typing relation operates over a fixed set of simple F⋆ types. These include base types (**unit**, **bool**, **string**, **file_descr**, and **nat**), pure function types (a → b), effectful function types (a → io b), product types (a & b), and sum types (**either** a b). The full typing relation is given in Figure 5.3. Following FGCBV [146], the relation is split into two mutually defined judgements: **typing** for values and $\mathbf{typing}_{\mathbf{io}}$ for **io** computations.

[146]: Levy et al. (2003), *Modelling environments in call-by-value programming languages*

The typing relation is defined on the interface of the **io** monad, which provides **io_return** and **io_bind** with their standard types, and a few primitives for file operations: **io_openfile** (open a file by name), **io_read** / **io_write** (read from / write to a file descriptor), and **io_close** (close a file descriptor). Each operation can fail, so e.g., **io_openfile** returns **io (either file_descr err)**.

Values (typing). The value typing rules from §5.2.2 (Qfalse, QAxiom, QWeaken, QApp, QLambda) are complemented with analogous rules for other introduction forms: constants (Qtrue, Qtt, QStringLit), constructors (QInl, QInr, QMkpair), and projections (QFst, QSnd). Two rules deserve special attention. First, QCase allows pattern matching on **either** a b when both branches produce a *value*: the scrutinee is typed as a value, and each branch is typed in the environment extended with the matched component. QIf is similar for Booleans, and QNRec is the usual iterator for natural numbers. In the FGCBV terminology of Levy, Power, and Thielecke [146], these are *complex values*: elimination forms whose result is a value rather than a computation. Second, QLambdaIO is the rule for effectful λ-abstractions: it types a value of type a → io b by requiring the body to be a well-typed **io** computation.

Computations ($\mathbf{typing}_{\mathbf{io}}$). The rules for typing **io** computations include: QReturn for monadic return, QBind for sequencing (where the continuation is typed in the environment extended with the intermediate result), and IO operations (QOpenfile, QRead, QWrite, QClose), each of which requires its argument to be a well-typed value. The remaining rules handle elimination forms whose result is a computation: QAppIO for applying an effectful function (a → io b), and QCaseIO (resp. QIfIO) for pattern matching (resp. conditionals) where the branches are computations. Note that case analysis and conditionals appear in both relations—QCase/QIf for values and QCaseIO/QIfIO for computations—because the same elimination can yield either kind of term depending on the context.

```
type typing : Γ:typ_env → a:Type → (eval_env Γ → a) → Type =
| QCase : #Γ:typ_env → #a:Type → #b:Type → #c:Type →
  #sc:(eval_env Γ→either a b) → #il:(eval_env (extend a Γ)→c) → #ir:(eval_env (extend b Γ)→c) →
  typing Γ (either a b) sc → typing (extend a Γ) c il → typing (extend b Γ) c ir →
  typing Γ c (λ γ → match sc γ with | Inl x → il (push γ x) | Inr y → ir (push γ y))
| QNRec : #Γ:typ_env → #a:Type →
  #n:(eval_env Γ→nat) → #z:(eval_env Γ→a) → #s:(eval_env Γ→a→a) →
  typing Γ nat n → typing Γ a z → typing Γ (a→a) s →
  typing Γ a (λ γ → nrec #a (n γ) (z γ) (s γ))
| QLambdaIO : #Γ:typ_env → #a:Type → #b:Type → #body:(eval_env (extend a Γ) → io b) →
  typing_io (extend a Γ) b body →
  typing Γ (a → io b) (λ γ x → body (push γ x))
(* Qfalse, QAxiom, QWeaken, QApp, QLambda were presented in 2.2 *)
(* Qtt, Qtrue, QStringLit, QIf, QMkpair, QFst, QInl, QZero and QSucc are similarly defined *)

and typing_io : Γ:typ_env → a:Type → (eval_env Γ → io a) → Type =
| QOpenfile : #Γ:typ_env → #fnm:(eval_env Γ → bool) →
  typing Γ string fnm →
  typing_io Γ (either file_descr err) (λ _ → io_openfile fnm)
| QReturn : #Γ:typ_env → a:Type → #x:(eval_env Γ → a) →
  typing Γ a x →
  typing_io Γ a (λ _ → io_return x)
| QBind : #Γ:typ_env → #a:Type → #b:Type → #m:(eval_env Γ→io a) → #k:(eval_env (extend a Γ)→io b) →
  typing_io Γ a m → typing_io (extend a Γ) b k →
  typing_io Γ b (λ γ → io_bind (m γ) (λ x → k (push γ x)))
| QAppIO : #Γ:typ_env → #a:Type → #b:Type → #f:(eval_env Γ → a → io b) → #x:(eval_env Γ → a) →
  typing Γ (a → io b) f → typing Γ a x →
  typing_io Γ b (λ γ → (f γ) (x γ))
| QCaseIO : #Γ:typ_env → #a:Type → #b:Type → #c:Type →
  #sc:(eval_env Γ→either a b) → #il:(eval_env (extend a Γ)→io c) → #ic:(eval_env (extend b Γ)→io c) →
  typing Γ (either a b) sc → typing_io (extend a Γ) c il → typing_io (extend b Γ) c ic →
  typing_io Γ c (λ γ → match sc γ with | Inl x → il (push γ x) | Inr y → ic (push γ y))
```

Figure 5.3: The typing relation for IO⋆.

Expressivity of IO⋆. The language carved out by these typing relations is a simply typed FGCBV language with IO primitives. It supports higher-order functions (both pure and effectful), pairs, sums, Booleans, strings, file descriptors, and natural numbers. All IO⋆ programs are terminating. Recursive F⋆ functions are not currently quotable. However, useful recursion patterns can still be recovered by extending the quoted-type universe and the typing relation with specific inductive types and their iterators; we include natural numbers with constructors zero, successor, and the usual iterator, which suffices to express functions such as factorial that are typically written in a recursive form. Supporting refinement types, dependent types, recursive types, and general recursion is otherwise left as future work (Chapter 6).

5.3.2 Generating Derivations Using a Metaprogram

We write our relational quotation metaprogram to produce derivations directly in F⋆, which supports metaprogramming by providing a primitive effect and a set of primitives that allow metaprograms to interact with F⋆'s type-checker via an API called Meta-F⋆ [93]. In this section, we present how the key ideas about the metaprogram (§5.2.2) are realized.

[93]: Martínez et al. (2019), *Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms*

We define a metaprogram generate_derivation that takes the quotation of an F* program, and tries to create a derivation for it. If the program is not in the IO* language, then the metaprogram will fail. To produce the quotation of the wrapper (Figure 5.2), we use the backtick operator, which is a special F* quotation primitive for top-level definitions (`wrapper). To call the instantiated metaprogram inside F*, we use the special splice_t primitive, which expects the metaprogram to return a list of F* terms representing new top-level let bindings. It requires that the new bindings are well typed; and in this case there is just one with new binding named wrapper_typing.

```
%splice_t[wrapper_typing] (generate_derivation (`wrapper))
```

The generate_derivation metaprogram does the following steps:

1. **Goal Formation:** It constructs the expected type for the derivation by using the empty typing environment and thunking the program: typing empty _ (λ _ → wrapper).
2. **Term Generation:** It generates a term for the derivation with uninstantiated implicit arguments. This is a syntactic mapping from F* syntax to the typing relation constructors. The mapping distinguishes between pure terms and computations in the io monad, selecting appropriate typing rules. We have to keep track of the types of the binders so that we know when to use QApp or QAppIO. We map F*'s locally nameless representation to our representation, and we inline top-level definitions.
3. **Type-checking:** Finally, we have to prove that the derivation term has the expected type (from step 1). Using Meta-F*, we split it into three steps to have more control over proving the type equality:
    a) **Implicit Argument Instantiation:** It calls Meta-F* to infer the implicit arguments of the generated term (from step 2) using the expected top-level type (from step 1). This is one call and no special guidance is necessary for inference to be complete. The unification of the typing environment and of the types should be decidable, since the typing rules correspond to a simply typed FGCBV language. The unification of open values and computations should always succeed because of the structure of the typing rules, which propagate them bottom-up.
    b) **Typing inference:** At this moment, Meta-F* does not know yet the type of the derivation term. It got instruction just to instantiate the implicit arguments, which could be incomplete. Since we know the instantiation should be complete, the metaprogram calls Meta-F* to infer the type of the derivation term.
    c) **Type Equality Check:** It checks that the inferred type of the derivation matches the expected type (from step 1). Crucially, this step checks that the program constructed by the derivation is equal to the original program (wrapper). To solve the equality, it manually unfolds some specific definitions, and then it uses F*'s reflexivity tactic (trefl), which first applies F*'s normalization to unfold definitions and to reduce primitive operations in both programs, and then performs unification. The equality check should always succeed because the derivation follows the structure of the program.

5.4 Relating the Trace-Producing Semantics of IO* and λ_{io}

In this section we formally relate the shallowly embedded IO* language with the deeply embedded λ_{io} language that IO* programs are extracted to. λ_{io} is an ML-like

language based on the simply typed λ-calculus. We relate it to IO* by equipping both languages with trace-producing semantics.

The formalization in this section and the proofs in the next are what let us verify the second extraction step—syntax generation—once and for all, instead of relying on translation validation for it, as all prior work on certifying extraction had to. This step is also not a standard verified-compilation step: it relates the original *shallowly embedded* IO* program, which has no operational semantics of its own, to a *deeply embedded* λ_{io} program, which does. The two cross-language logical relations we define in this section are precisely what bridges this gap, and they are the foundation on which the correctness and security proofs of §5.5 rest.

5.4.1 Syntax of λ_{io}

We begin by defining the syntax of λ_{io} expressions as the inductive F* type exp.

```
type exp =
| EVar : v:var → exp (* variables represented as de Bruijn indices *)
| ELam : exp → exp (* λ–abstraction *)
| EFileDescr : file_descr → exp (* file descriptor *)
| EString : string → exp (* string constant *)
| EZero : exp
| ESucc : exp → exp
| ENRec : exp → exp → exp → exp (* iterator *)
| ERead : exp → exp (* reading from a file *)
| EWrite : exp → exp → exp (* writing to a file *)
| EOpen : exp → exp (* opening a file *)
| EClose : exp → exp (* closing a file *)
(* there are also constructors for EUnit, Etrue, Efalse, EApp, EIf,
   EPair, EFst, ESnd, ECase, EInl, EInr *)
```

We also recursively define a predicate is_value e that identifies expressions e which are values, i.e., that are closed expressions in introduction form, such as λ-abstractions and pairs of values.

The ERead, EWrite, EOpen, and EClose expressions represent the extension of the simply typed λ-calculus to include effectful IO operations. These operations can fail. Their results are of type either a err, where the left injection represents successful execution and the right injection of error represents a failed execution. We use strings to represent file names and the data that can be read from or written to a file. We use file descriptors to represent the unique descriptor generated by the operating system with each open file call. For simplicity, these file descriptors are modeled as natural numbers. ERead takes a file descriptor and returns an optional string, EWrite takes a file descriptor and a string and returns an optional unit, EOpen takes a string and returns an optional file descriptor, and EClose takes a file descriptor and returns an optional unit.

5.4.2 Small-Step Trace-Producing Operational Semantics of λ_{io}

Next, we equip λ_{io} with an operational semantics. Our semantics is based on sequences of events, called *traces*, which arise from the execution of IO operations (ERead, EWrite, EOpen, and EClose). Events are indexed by the arguments and result of the operation, e.g., executing the read operation ERead (EFileDescr fd) produces an event EvRead fd r, where r is the optional string that was read.

To link the computations being executed to the traces they produce, we represent the small-step operational semantics of λ_{io} as a reduction relation defined as an inductive type indexed by two λ_{io} expressions (representing one step of execution), a list of past events (history), and an optional event indexed by the history. The optional event allows us to explicitly differentiate between effectful and pure reduction steps. Pure reduction steps are indexed by None, and reduction steps involving the execution of IO operations are indexed by the corresponding events. As is standard practice, we define the computation rules of the operational semantics on closed expressions [147]. The reduction rules include congruence rules to transform subexpressions into value forms, and computation rules to execute expressions when they have been reduced to redex forms. To illustrate the step relation, we present one pure reduction rule (β-reduction), and the congruence rule and two reduction rules for the (un)successful execution of file opening. Reduction rules for other expressions (in particular, other IO operations) are defined analogously, and can be found in the artifact.

[147]: Ahmed (2004), *Semantics of Types for Mutable State*

```
type step : closed_exp → closed_exp → h:history → option (event_h h) → Type =
...
| SBeta : e11:exp{is_closed (ELam e11)} → e2:value → h:history →
  step (EApp (ELam e11) e2) (subst_beta e2 e11) h None
...
| SOpen : #str #str':closed_exp → #h:history → #oev:option (event_h h) →
  step str str' h oev →
  step (EOpen str) (EOpen str') h oev
| SOpenReturnSuccess : str:string → h:history →
  step (EOpen (EString str)) (EInl (EFileDescr (fresh_fd h))) h
    (Some (EvOpen str (Inl (fresh_fd h))))
| SOpenReturnFail : str:string → h:history →
  step (EOpen (EString str)) (EInr (EString "err")) h (Some (EvOpen str (Inr "err")))
```

We note that the optional event argument of step is indexed by the history because certain effectful steps use the history to generate the λ_{io} expression to which they step. For instance, in SOpenReturnSuccess, the function fresh_fd: history → file_descr is used to generate a fresh unique file descriptor based on the previously generated file descriptors listed in the history h. Meanwhile, in SOpenReturnFail, the event EvOpen records with the error "err" that opening the file was unsuccessful.

The reader should observe that this operational semantics does not actually model any of the files being accessed as some form of state. Instead, the semantics models all the possible IO events that might be produced if actual files were to be accessed by programs written in λ_{io}.

To compute the reflexive-transitive closure of the step relation, we want to consider traces of events corresponding to the events produced by the sequence of steps. To maintain the well-formedness of events with respect to a particular history for each of the individual reduction steps, we define a notion of local traces, which are traces that are well formed with respect to a particular history from which they start. The well-formedness of a trace is calculated recursively.

```
let rec well_formed_local_trace (h:history) (lt:trace) =
  match lt with
  | [] → T
  | EvOpen _ (Inl fd) :: tl → fd == (fresh_fd h) ∧ well_formed_local_trace (ev::h) tl
  | ev :: tl → well_formed_local_trace (ev::h) tl
```

This definition of well-formedness ensures that all the events in the local trace are well formed with respect to the given history *and* the events preceding them in the local trace. This refinement on traces greatly simplifies formalization and reasoning

about the validity of a local trace. Monotonically increasing local traces are trivially well-formed, and the order of sequential traces is clear from histories by which the local traces are indexed. For example, a local trace lt2:local_trace (h++lt1) can only be appended to a local trace typed as lt1:local_trace h, giving lt1@lt2 : local_trace h. Here the append function (++) : (h:history) → local_trace h → history reverses the given local trace and prepends it to the history. Therefore, the most recent events appear at the beginning of the history and are in reverse order. For local traces, we use above the built-in append function (@) : #h:history → lt:local_trace h → lt':local_trace (h++lt) → local_trace h with the additional refinement that the resulting local trace is well formed with respect to the history h.

We then represent the reflexive-transitive closure of step as an inductive type steps indexed by two closed λ_{io} expressions, a history, and a local trace modeling the events produced by the sequence of steps and indexed by the history where these steps start. The definition is mostly standard, with local traces getting concatenated when the corresponding reduction steps are concatenated.

```
type steps : closed_exp → closed_exp → h:history → local_trace h → Type =
| SRefl : e:closed_exp → h:history → steps e e h []
| STrans : #e1:_ → #e2:_ → #e3:_ → #h:_ →
  #oev:option (event_h h) → #lt:local_trace (h++(as_lt oev)) →
  step e1 e2 h oev → steps e2 e3 (h++(as_lt oev)) lt → steps e1 e3 h ((as_lt oev) @ lt)
```

We define the type beh_λ e e' h lt to represent steps e e' h lt where e' is an irreducible expression.

5.4.3 Syntax and Semantics of IO*

We represent the IO computations of the shallowly embedded IO* language in using the freer monad io we defined in §2.4.1 (without partiality).

We give semantics to computations represented using io in the tradition of Dijkstra monads [9], but without constructing one. For the specification monad, we redefine hist on top of well-formed histories and local traces.

[9]: Maillard et al. (2019), *Dijkstra Monads for All*

```
type hist a = h:history → (lt:local_trace h → r:a → Type0) → Type0
```

In order to give a semantics to IO* computations in hist, we reuse the monad morphism θ from §2.4.1. We use θ to specify the semantics of IO* computations in terms of predicates on the local traces the computations produce and the values they return. Formally, the predicate $\text{beh}_\star$ fs_e h we define is the strongest postcondition corresponding to the predicate transformer θ fs_e. [9]

```
let beh⋆ #a (fs_e:io a) (h:history) =
  λ lt res → ∀ p. (θ fs_e) h p ⟹ p lt res
```

5.4.4 Quoted Types at Which Programs Are Related

The next predicate characterizes the types we want to be able to extract, and, thus, the types of λ_{io} and IO* expressions that we want to be able to relate with the logical relations. It includes base types (unit, bool, file_descr, and nat), function types (both pure and effectful), and product and sum types.

```
noeq type type_rep : Type → Type =
| QUnit : type_rep unit
| QBool : type_rep bool
| QFileDescriptor : type_rep file_descr
| QNat : type_rep nat
| QArr : #a:Type → #b:Type → type_rep a → type_rep b → type_rep (a → b)
| QArrIO : #a:Type → #b:Type → type_rep a → type_rep b → type_rep (a → io b)
| QPair : #a:Type → #b:Type → type_rep a → type_rep b → type_rep (a & b)
| QSum : #a:Type → #b:Type → type_rep a → type_rep b → type_rep (either a b)
```

We define the type qType as a dependent pair between an F* type and a proof that it satisfies the predicate, as t:Type_0 & type_rep t. Formally, the environments and the typing relation are defined in terms of qType, but we hide it in the chapter for readability.

5.4.5 Target-to-Source and Source-to-Target Logical Relations for Relating Programs

To verify SEIO*, we have to formally relate λ_{io} programs with IO* programs. At a high level, two programs are related if they have the same behavior—i.e., they evaluate to the same *results* and produce the same *local traces*. We relate a λ_{io} program to an IO* program using two logical relations, one for each direction. In the target-to-source relation, for all λ_{io} behavior, there must exist corresponding IO* behavior. In the source-to-target relation, we have the dual requirement.

As is standard [147], each logical relation is defined from a value relation and, because IO* is a FGCBV language, two expression relations: a pure expression relation and an IO expression relation. The way that we define IO* expressions, as either F* values or F* computations in the io monad, allows us to differentiate between pure and effectful expressions.

Logical Relations for *Closed* λ_{io} and IO* Expressions

For the *target-to-source* logical relation, we define the value relation qt ∋ (h, fs_v, v) between histories h, IO* values fs_v, and λ_{io} values v by induction on (quoted) types in the natural way, as shown below. The only subtlety in this definition is that in the case of the two kinds of function types, the definition is given in the style of Kripke's possible worlds semantics, where the function arguments are quantified and the applications considered in histories extended with arbitrary compatible local traces.

```
let (∋) (qt:qType) (h:history) (fs_v:qt.1) (v:value) =
 match qt.2 with
 | QUnit → fs_v == () ∧ v == EUnit
 | QFileDescriptor → v == EFileDescr fs_v
 | QString → v == EString fs_v
 | QBool → (fs_v == true ∧ v == Etrue) ∨ (fs_v == false ∧ v == Efalse)
 | QArr qt1 qt2 →
   let ELam body = v in
   ∀ (arg:value) (fs_arg:qt1.1) (lt:local_trace h).
     qt1 ∋ (h++lt, fs_arg, arg) ⟹ (qt2 ⊇ (h++lt, fs_v fs_arg, subst_beta arg body))
 | QArrIO qt1 qt2 →
   let ELam body = v in
   ∀ (arg:value) (fs_arg:qt1.1) (lt:local_trace h).
     qt1 ∋ (h++lt, fs_arg, arg) ⟹ (qt2 ⊇io (h++lt, fs_v fs_arg, subst_beta arg body))
 (* QPair, QSum and QNat are defined as expected *)
```

The two expression relations are defined below. They specify that the behavior of the λ_{io} expression must imply a corresponding behavior for the `IO*` expression. The difference is that the pure relation requires the local trace to be empty. These relations express that any behavior possible for a λ_{io} expression that an `IO*` expression is extracted to was already possible for the `IO*` expression.

```
and (⊇) (qt:qType) (h:history) (fs_e:qt.1) (e:closed_exp) =
 ∀ (e':closed_exp) (lt:local_trace h).
   beh_λ e e' h lt ⟹ (qt ∋ (h, fs_e, e') ∧ lt == [])

and (⊇io) (qt:qType) (h:history) (fs_e:io qt.1) (e:closed_exp) =
 ∀ (e':closed_exp) (lt:local_trace h).
   beh_λ e e' h lt ⟹
     (∃ (fs_e':qt.1). qt ∋ (h++lt, fs_e', e') ∧ beh_★ fs_e h lt fs_e')
```

For proving our security criterion in §5.5, it is useful to restrict these relations to hold for *all* histories. For example, we define $\texttt{fs\_e} \overset{\ni}{\approx} \texttt{e}$ to mean that ∀ (h:history). t ∋ (h, fs_e, e). Analogously, we define $\texttt{fs\_e} \overset{\supseteq}{\approx} \texttt{e}$ and $\texttt{fs\_e} \overset{\supseteq_{io}}{\approx} \texttt{e}$ to mean that ∀ (h:history). t ⊇ (h, fs_e, e) and ∀ (h:history). t $\supseteq_{io}$ (h, fs_e, e).

For the *source-to-target* logical relation, the value relation qt ∈ (h, fs_v, v) is defined analogously by induction on types, with the only difference between the two logical relations being how the pure and IO expression relations are defined that the cases for the two function types refer to.

```
let (∈) (qt:qType) (h:history) (fs_v:qt.1) (v:value) = ...
and (⊆) (qt:qType) (h:history) (fs_e:qt.1) (e:closed_exp) =
 ∃ (e':closed_exp). beh_λ e e' h [] ∧ qt ∈ (h, fs_e, e')

and (⊆io) (qt:qType) (h:history) (fs_e:io qt.1) (e:closed_exp) =
 ∀ (fs_e':qt.1) (lt:local_trace h).
   beh_★ fs_e h lt fs_e' ⟹
     (∃ (e':closed_exp). qt ∈ (h++lt, fs_e', e') ∧ beh_λ e e' h lt)
```

The definitions are dual to the target-to-source logical relation defined above. Here we require that the behavior of the `IO*` expression must imply a corresponding behavior for the λ_{io} expression.

Similarly to the target-to-source relation, we also find it useful to restrict these relations to hold for all histories. For instance, we define $\texttt{fs\_e} \overset{\in}{\approx} \texttt{e}$ to mean that ∀ (h:history). t ∈ (h, fs_e, e).

Logical Relations for *Open* λ_{io} and IO* Expressions

While closed expressions are evaluated at runtime, static type-checking rules operate on open expressions that might contain free variables. To use the logical relations we defined above on open expressions, we define type environments (mapping from variables to optional qTypes), IO* evaluation environments (mapping from variables to IO* values), and λ_{io} evaluation environments (mapping from variables to λ_{io} values—in other words, these are (parallel) substitutions for λ_{io}).

The type environment maps to optional qTypes because we use de Bruijn indices to represent variables. Therefore, every (natural number) index must have a corresponding entry, but only the free variables we seek to instantiate map to the appropriate qTypes. Both evaluation environments are indexed by the type environment. The λ_{io} evaluation environment is additionally indexed by a Boolean flag which specifies whether the substitution is in fact just a variable renaming.

We say that IO* and λ_{io} evaluation environments are logically related at a particular type environment Γ if the two evaluation environments map variables in Γ to related values.

```
let (∼) (#Γ:typ_env) (h:history) (γ⋆:eval_env Γ) #b (γλ:gsub Γ b) =
  ∀ (x:var). Some? (Γ x) ⟹ Some?.v (Γ x) ∋ (h, γ⋆ x, γλ x)

let (≍) (#Γ:typ_env) (h:history) (γ⋆:eval_env Γ) #b (γλ:gsub Γ b) =
  ∀ (x:var). Some? (Γ x) ⟹ Some?.v (Γ x) ∈ (h, γ⋆ x, γλ x)
```

Now, we can relate open λ_{io} and IO* expressions at a given type environment by specifying that for every related λ_{io} and IO* evaluation environment indexed by the type environment, the corresponding closed λ_{io} and IO* expressions must be related at the expected type. We present the definition for the target-to-source logical relation below. The source-to-target relation is analogous.

```
let (⊳) (#Γ:typ_env) (qt:qType) (fs_oe:eval_env Γ → qt.1) (oe:exp) =
  fv_in_env Γ oe ∧
  ∀ b (γλ:gsub Γ b) (γ⋆:eval_env Γ) (h:history).
    γ⋆ `(∼) h` γλ ⟹ qt ⊇ (h, fs_oe γ⋆, gsubst γλ oe)
```

The predicate fv_in_env specifies that all free variables in the λ_{io} expression are mapped to Some qt by the type environment Γ. Open IO* values are modeled as functions from an evaluation environment that closes them to the corresponding closed IO* value. In the definition, fs_oe $\gamma_\star$ is the result of replacing all the free variables in fs_oe with their values under the IO* evaluation environment $\gamma_\star$, with gsubst γ_λ oe doing the same but for oe using the λ_{io} evaluation environment γ_λ. For open λ_{io} expressions and IO* computations, their closed variants must be in the IO expression relation.

5.4.6 Compatibility of the Logical Relations with Typing Rules

As part of proving our security criterion in §5.5.2, we prove in §5.5.1 the fundamental property of logical relation for both of our logical relations, relating a source program to the corresponding compiled code. As is typical, our proofs of the fundamental property also proceed by induction on typing derivations, and to keep the proof manageable, one proves a number of compatibility lemmas showing that the logical relations satisfy rules analogous to the typing rules of the language.

```
let app_fn_arg #Γ (#a #b:qType) (fs_f:eval_env Γ → (a.₁ → b.₁)) (fs_x:eval_env Γ → a.₁) (f x:exp)
: Lemma
  (requires fs_f ⊳ f ∧ fs_x ⊳ x)
  (ensures  (λ γ → (fs_f γ) (fs_x γ)) ⊳ EApp f x)

let app_io_fn_arg #Γ (#a #b:qType) (fs_f:eval_env Γ → (a.₁ → io b.₁)) (fs_x:eval_env Γ → a.₁) (f x:exp)
: Lemma
  (requires fs_f ⊳ f ∧ fs_x ⊳ x)
  (ensures  (λ γ → (fs_f γ) (fs_x γ)) ⊳_io EApp f x)

let app_io_fn_io_arg #Γ (#a #b:qType) (fs_f:eval_env Γ → io (a.₁ → io b.₁)) (fs_x:eval_env Γ → io a.₁) (f x:exp)
: Lemma
  (requires fs_f ⊳_io f ∧ fs_x ⊳_io x)
  (ensures
   (λ γ → io_bind (fs_f γ) (λ f' → io_bind (fs_x γ) (λ x' → f' x'))) ⊳_io EApp f x)
```

Since our logical relations relate both pure and IO expressions, we need to prove three kinds of compatibility lemmas for each typing rule:

1. when the subexpressions are pure expressions and the composite expression is pure;
2. when some of the subexpressions are pure expressions and some are IO expressions and the composite expression is an IO expression;
3. when the subexpressions are IO expressions and the composite is also an IO expression.

We illustrate this with the three kinds of compatibility lemmas for the source-to-target logical relation corresponding to the typing rule for function application. The app_fn_arg lemma considers a pure arrow and a pure argument. The app_io_fn_arg lemma considers an arrow that takes a pure argument and returns an IO computation and a pure argument. The app_io_fn_io_arg lemma considers an IO arrow and an IO argument.

We sketch the proof of app_io_fn_io_arg. Full details of the proofs are available in our artifact.

1. For an arbitrary λ_{io} evaluation environment and related $\texttt{IO}^\star$ evaluation environment, we close the open $\texttt{IO}^\star$ and λ_{io} expression with these environments. For the rest of the proof sketch, we use e == EApp f x and fs_e == io_bind fs_f (io_bind fs_x) to refer to these closed expressions. We consider arbitrary irreducible closed λ_{io} expression e' and local trace lt:local_trace h such that beh_λ e e' h lt, and we need to prove ∃ (fs_e':b.₁). b ∋ (h++lt, fs_e', e') ∧ $\text{beh}_\star$ fs_e h lt fs_e'.
2. We then destruct the steps from e to e' into the intermediate steps of the subexpressions.
   a) For the lambda expression, we get beh_λ f (ELam f') h lt1 and beh_λ e (EApp (ELam f') x) h lt1.
   b) Since the reduction of the lambda expression corresponds to lt1:local_trace h, we reduce the argument expression starting from the history h++lt1. For the argument expression, we get beh_λ x x' (h++lt1) lt2 and beh_λ (EApp (ELam f') x) (subst_beta x' f') (h++lt1) lt2.
   c) Finally, we get that beh_λ (subst_beta x' f') e' ((h++lt1)++lt2) lt3 and lt == lt1@lt2@lt3.
3. Next, we use the fact that relatedness of the λ_{io} and $\texttt{IO}^\star$ evaluation environments is closed under history extension to apply the induction hypothesis

and get information about the IO⋆ behavior corresponding to these three sequences of steps, specifically that
∃ (fs_x':$a._1$). $beh_\star$ fs_x (h++lt1) lt2 fs_x' and
∃ (fs_f':$a._1 \to$ io $b._1$). $beh_\star$ fs_f h lt1 fs_f'.

4. By unfolding the IO arrow, since we know that
beh_λ (subst_beta x' f') e' ((h++lt1)++lt2) lt3 and lt == lt1@lt2@lt3, we get that
∃ (fs_e':$b._1$). b ∋ (h++lt, fs_e', e') ∧ $beh_\star$ (fs_f' fs_x') ((h++lt1)++lt2) lt3 fs_e'.
5. Since we know that $beh_\star$ fs_x (h++lt1) lt2 fs_x' and
$beh_\star$ (fs_f' fs_x') ((h++lt1)++lt2) lt3 fs_e', we use the fact that $beh_\star$ is a monad morphism to get that $beh_\star$ (io_bind fs_x fs_f') (h++lt1) (lt2@lt3) fs_e'.
6. Finally, we utilize this lemma once more with $beh_\star$ fs_f h lt1 fs_f' to get that
$beh_\star$ (io_bind fs_f (io_bind fs_x)) h (lt1@lt2@lt3) fs_e', which is exactly the desired $beh_\star$ behavior. □

Other compatibility lemmas for the target-to-source relation always follow this general proof pattern:

1. Destruct beh_λ e e' h lt into subexpressions' beh_λ behavior.
2. Apply the induction hypothesis to get subexpressions' $beh_\star$ behavior.
3. Combine $beh_\star$ behaviors to get the composite $beh_\star$ behavior corresponding to beh_λ e e' h lt.

Because of how we defined our source-to-target relation, we have two general patterns for proving compatibility lemmas depending on whether we are proving relatedness in the pure or IO expression relation. In the pure expression relation, we do not destruct $beh_\star$ fs_e h lt fs_e'. We can directly use the induction hypothesis to get the subexpressions' beh_λ behavior and combine these beh_λ behaviors to get the composite beh_λ behavior corresponding to $beh_\star$ fs_e h lt fs_e'.

In the IO expression relation, we follow a dual pattern to the target-to-source proof pattern above:

1. Destruct $beh_\star$ fs_e h lt fs_e' into subexpressions' $beh_\star$ behavior.
2. Apply the induction hypothesis to get subexpressions' beh_λ behavior.
3. Combine beh_λ behaviors to get the composite beh_λ behavior corresponding to $beh_\star$ fs_e h lt fs_e'.

It is worth noting that destructing $beh_\star$ behaviors is more complicated than destructing beh_λ behaviors. While beh_λ behaviors are defined inductively and can be effectively pattern-matched on, the $beh_\star$ behaviors are defined in terms of logical implications (§5.4.3), leading to indirect reasoning about destructing local traces by instantiating them with suitable postconditions.

5.5 SEIO⋆: Formally Secure Extraction

SEIO⋆ involves two steps: relational quotation and syntax generation. Relational quotation is a translation validation technique that involves a metaprogram, while syntax generation is a pure F⋆ function that we verify once and for all. In this section, we explain how we instantiate the compilation model of Abate et al. [8] with SEIO⋆ (§5.5.1) to prove that extraction is sound, in the form of compiler correctness, and that it satisfies RrHP (§5.5.2).

[8]: Abate et al. (2019), *Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation*

5.5.1 Overview of the Compilation Model

When we instantiate the compilation model of Abate et al. [8] with SEIO* we cannot talk about the relational quotation metaprogram, since in F* we cannot state any extrinsic property about a metaprogram. So we do this instantiation such that the final criterion we prove about the model directly relates the shallowly embedded program with its extraction produced by SEIO*.

The source language is IO*. A partial source program (denoted by P, of type progS) is a dependent pair containing (1) an F* function in the io monad taking a context as an argument and returning a Boolean, and (2) a typing derivation that proves the program is in the IO* language. The partial source program and the source context share an interface (of type interface) that contains only the type of the context ct. A source context can be any F* code with a type that can be quoted (§5.4.4). Source linking (linkS, denoted by $C^S[P]$) is function application.

```
noeq type interface = { ct : Type; qtyp: type_rep ct; }

(** Source **)
type progS (i:interface) =
  ps:(i.ct → io bool) & (typing empty (i.ct → io bool) (λ _ → ps))
type ctxS (i:interface) = i.ct
type wholeS = io bool
let linkS (#i:interface) (ps:progS i) (cs:ctxS i) : wholeS = (dfst ps) cs
```

While each partial source program includes a typing derivation, it is there only to be able to do compilation (one can think of it as a compilation hint) and it is not relevant for the semantics of the partial program. One can see above that during source linking, the typing derivation is discarded, and whole source programs are just io computations. This is key to stating a secure compilation criterion that directly relates the shallowly embedded source program with its extraction.

```
(** Target **)
type progT (i:interface) = value
type ctxT (i:interface) = ct:value & typing_λ empty ct i.ct
type wholeT = closed_exp
let linkT (#i:interface) (pt:progT i) (ct:ctxT i) : wholeT = EApp pt (dfst e)
```

Compilation is our syntax generation function, taking a source program P and producing a target program denoted by $P{\downarrow}$, which is a value in the target language λ_{io} (of type progT). A target program can be linked securely with any syntactically typed target context (denoted by C^T, of type ctxT). To type target contexts, we introduce a simple syntactic typing relation, typing_λ, defined as expected. For simplicity, when defining the typing relation for λ_{io}, we reuse the notion of types (§5.4.4) and typing environments from IO*. We also have the same source and target interfaces. Finally, it is worth noting that the extracted program $P{\downarrow}$ is not necessarily syntactically well typed, but we prove it is semantically well typed as part of the logical relation proofs below. The target of our extraction is basically untyped, similarly to Malfunction [28], as opposed to the default F* and Rocq extraction that has to work hard to work around the limitations of a syntactic type system [4].

Compilation is defined recursively on the typing relation of IO* and generates λ_{io} syntax. We prove in F* that the source and compiled programs are related by both logical relations. Since compilation returns a λ_{io} value, we show that the source and compiled program are related for all histories by the two value relations (by $\overset{\ni}{\approx}$ and

[28]: Forster et al. (2024), *Verified Extraction from Coq to OCaml*

[4]: Letouzey (2004), *Programmation fonctionnelle certifiée : L'extraction de programmes dans l'assistant Coq. (Certified functional programming: Program extraction within Coq proof assistant)*

$\overset{\in}{\approx}$). We state the appropriate notion of the fundamental property for $\overset{\ni}{\approx}$ direction as a theorem; the $\overset{\in}{\approx}$ direction is analogous.

Theorem 5.5.1 (Fundamental property)

$$\forall I^S.\ \forall P\ :\ \mathrm{prog}^S\ I^S.\ P \overset{\ni}{\approx} P{\downarrow}$$

Proof. Proof by applying the relevant compatibility lemmas discussed in §5.4.6. □

Finally, we give semantics to whole source and target programs using $\rightsquigarrow_S$ and $\rightsquigarrow_T$, which are defined in terms of $\mathrm{beh}_\star$ (§5.4.3) and beh_λ (§5.4.2) starting from the empty history ([]).

```
let (⇝S) (ws:wholeS) ((It, r):(local_trace [] * bool)) =
  behλ ws [] It r

let (⇝T) (wt:wholeT) ((It, r):(local_trace [] * bool)) =
  behλ wt (if r then Etrue else Efalse) [] It
```

Having a semantics, we can prove soundness of extraction, in the form of compiler correctness for programs of type unit → io bool:

Theorem 5.5.2 (Compiler Correctness)

$$\forall P.\ \forall t.\ P() \rightsquigarrow_S t \Leftrightarrow \ \mathrm{EApp}\ P{\downarrow}\ \ \mathrm{EUnit}\ \rightsquigarrow_T t$$

Proof sketch. The equivalence follows from Theorem 5.5.1 and the analogous theorem for $\overset{\in}{\approx}$. □

We cannot state soundness the way we did in Chapter 3 and Chapter 4 (Theorem 3.6.1 and Theorem 4.5.3), where the global specification of the verified program is enforced intrinsically, using Dijkstra monads. In SEIO⋆, the io monad carries no specification and programs are verified extrinsically, thus, we state soundness as compiler correctness instead. One could in principle get that the extracted program satisfies a trace property by instantiating RrHP, but RrHP's premise quantifies over arbitrary source contexts, which makes it impractical to get such a proof, reason for which a separate criterion is useful.

5.5.2 Robust Relational Hyperproperty Preservation (RrHP)

We show in F⋆ that SEIO⋆ satisfies Robust Relational Hyperproperty Preservation (RrHP), the strongest secure compilation criterion of Abate et al. [8]. RrHP not only implies full abstraction (i.e., preserving observational equivalence), but also ensures the preservation of all trace properties and hyperproperties (like noninterference) against adversarial contexts.

Theorem 5.5.3 (Robust Relational Hyperproperty Preservation (RrHP))

$$\forall I^S.\ \forall C^T.\ \exists C^S.\ \forall P\ :\ \mathrm{prog}^S\ I^S.\ \forall t.\ (C^T[P{\downarrow}] \rightsquigarrow_T t \Leftrightarrow C^S[P] \rightsquigarrow_S t)$$

The statement of RrHP includes an existential quantifier, $\exists C^S$, which we instantiate using a back-translation function (denoted by $C^T{\uparrow}$) from target contexts to source contexts. Since target contexts are syntactically well-typed, back-translation is defined

recursively on the typing derivations of λ_{io}. The body of RrHP is an equivalence, so we prove each direction separately by showing that the source context obtained by back-translation is related to the original target context with respect to one of our two logical relations. These results are another instance of the fundamental property of logical relations for our two relations, so the proofs again rely on compatibility lemmas from §5.4.6.

To prove the right-to-left direction of RrHP (⇒), we use the logical relation that relates each target behavior to a source behavior (§5.4.5).

Theorem 5.5.4 (RrHP ⇒)

$$\forall I^S.\ \forall C^T.\ \forall P.\ \forall t.\ C^T[P\downarrow] \rightsquigarrow_T t \Rightarrow C^T\uparrow [P] \rightsquigarrow_S t$$

Proof sketch. By Theorem 5.5.1, we know that $P \overset{\ni}{\approx} P\downarrow$. Since back-translation is verified and C^T is a value, we know that $C^T\uparrow \overset{\ni}{\approx} C^T$. Since our programs are functions, it follows that $(P\ C^T\uparrow) \overset{\ni_{io}}{\approx} (\text{EApp}\ P\downarrow\ C^T)$. Since linking is function application, this is all we need. □

This RrHP direction already implies Robust Relational Subset-Closed Hyperproperty Preservation (RrSCHP), which is a weaker criterion than RrHP, but still one of the strongest secure compilation criteria of Abate et al. [8]. They show that RrSCHP implies full abstraction, and also ensures the preservation of trace properties and subset-closed hyperproperties against adversarial contexts.

To prove the other direction of RrHP, we use the source-to-target logical relation (§5.4.5).

Theorem 5.5.5 (RrHP ⇐)

$$\forall I^S.\ \forall C^T.\ \forall P.\ \forall t.\ C^T\uparrow [P] \rightsquigarrow_S t \Rightarrow C^T[P\downarrow] \rightsquigarrow_T t$$

Proof sketch. This is dual to the proof of the Theorem 5.5.4, just this time using the logical relation that relates each source behavior to a target behavior. □

5.6 Adding Support for Refinement Types

Until now, we have focused on the extraction of simply typed IO programs. F⋆ also has refinement types, which allow one to add logical constraints to types—e.g., one can define the type of positive 8-bit integers by refining the type of natural numbers as $\mathbf{x{:}nat\{x \leq 255\}}$. In this section we show how to extend SEIO⋆ to support IO programs, whose types are simple types, possibly decorated with refinements.

5.6.1 Relational Quotation for Refinements

Using refinements we can define $\mathbf{nat8}$ and a safe increment function that requires a proof that the number will not overflow. This can be done by refining the argument of the function like this:

```
type nat8 = x:nat{x ≤ 255}
let incr_nat8 (x:nat8{x + 1 ≤ 255}) : nat8 = x + 1
```

Now to call function incr_nat8, one has to prove at the call site that x + 1 ≤ 255. The way F⋆ type-checks a top-level definition that uses refinements is by computing a weakest precondition, which usually is automatically proven by SMT. This weakest precondition collects all the refinements, thus, if one proves the weakest precondition, one knows that all the refinements are satisfied. For example, for function incr_nat8, the weakest precondition looks conceptually like ∀ (x:nat). x + 1 ≤ 255 ⟹ incr_nat8 x ≤ 255.

In order to handle refinements, we extend the typing relation to mimic what F⋆ does—i.e., the typing relation also computes a precondition. For that, we index the typing relation in a precondition φ that is a predicate on the evaluation environment. The precondition must act on the evaluation environment to be able to refer to variables in the environment, namely when dealing with λ-abstractions. Precondition φ is used to refine the type of open F⋆ values by specifying that an open F⋆ value accepts only evaluation environments that satisfy the precondition.

```
type typing : Γ:typ_env → φ:(eval_env Γ → Type0) → a:Type → (γ:(eval_env Γ){φ (γ)} → a) → Type
```

The prior typing rules stay unchanged except for the fact that now they have to compute the precondition, which they do in the expected manner, similar to what F⋆ does.

```
| Qfalse : #Γ:typ_env → typing Γ (λ _ → T) bool (λ _ → false)

| QApp : #Γ:typ_env → #a:Type → #b:Type →
  #φf:(eval_env Γ → Type0) → #f:(γ:(eval_env Γ){φf (γ)} → a → b) →
  #φx:(eval_env Γ → Type0) → #x:(γ:(eval_env Γ){φx (γ)} → a) →
  typing Γ φf (a → b) f →
  typing Γ φx a x →
  typing Γ (λ γ → φf (γ) ∧ φx (γ)) b  (λ γ → (f γ) (x γ))

| QLambda : #Γ:typ_env → #a:Type → #b:Type →
  #φb:(eval_env (extend a Γ) → Type0) →
  #body:(γ:(eval_env (extend a Γ)){φb (γ)} → b) →
  typing (extend a Γ) φb b body →
  typing Γ (λ γ → ∀ (x:a). φb (push γ x)) (a → b) (λ γ x → body (push γ x))
```

We add only one typing rule specific to refinements, called QRef, that allows us to subtype a value of type x:α{p x} to type x:α{q x} as long as one can prove q x. This rule is sufficient because F⋆ treats all types as containing the trivial refinement—i.e., type α and x:α{T} are equivalent.

```
| QRef : #Γ:typ_env → #a:Type →
  #φv:(eval_env Γ → Type0) → #p:(a→Type0) → #v:(γ:(eval_env Γ){φv (γ)}→x:a{p x}) →
  typing Γ φv (x:a{p x}) v →
  #q:(a → Type0)
  typing Γ (λ γ → φv (γ) ∧ q (v γ)) (x:a{q x}) (λ γ → v γ)
```

We can use the typing rule manually to change the refinement of a value when it is necessary. For example, for the incr_nat8 function, we can use it like this:

```
let incr_nat8_typing : typing empty _ (x:nat8{x+1≤ 255} → nat8) (λ _ → incr_nat8) =
  QLambda (QRef (QSucc (QRef QAxiom)))
```

The second QRef is used to erase the refinement on x because the successor function expects just a natural number. The first QRef is used to refine x+1 with the fact that it is smaller than 255. The precondition of this derivation is inferred automatically because it is computed bottom-up given the structure of the typing rules.

During quotation, the metaprogram is in a challenging situation because F* transparently subtypes values between refinements, which means there is no syntactic marker to know when to use the QRef rule. Therefore, the metaprogram inserts QRef in all locations where typing information comes from the environment. Compared to the other rules, the metaprogram sometimes spells out some of the implicits of the rule QRef using the information it gets from the environment, but otherwise, the metaprogram works the same as described in §5.3.2.

Refined Functions With the support for refinements we just added to SEIO*, we can also refine functions. For example, we can refine incr_nat8 like this:

```
let incr_nat8' : f:(x:nat8{x + 1 ≤ 255} → nat8){∀ x. f x == x + 1} = incr_nat8
```

The refinement states that for all inputs, incr_nat8 increments them. This refinement could be written more idiomatically in F* as x:nat8{x + 1 ≤ 255} → y:nat8{y == x+1}, but for that we would need support for dependent types, which is left as future work (§5.9).

5.6.2 Formal Secure Extraction of Refinements

To have the same formal guarantees when extracting IO code with refinements, we have to make minimal changes to the compilation model from §5.5 and the logical relations (§5.4.5).

First, we have to update our logical relations. The logical relations for closed expressions remain unchanged, since the IO* expression is already typed at a refinement type, which makes the relation imply that the λ_{io} expression also satisfies the refinement. The logical relations for open expression have to be updated to account for the fact that now open F* values need the precondition to be satisfied. Therefore, we changed it so that open IO* and λ_{io} expressions are related *only for evaluation environments that satisfy the precondition*. We present the modified target-to-source logical relation below. The same change is expressed in the source-to-target logical relation.

```
let (⊳) (#Γ:typ_env) (qt:qType) (#φ:eval_env Γ →  Type0)
  (fs_oe:(γ:(eval_env Γ){φ (γ)} → qt.1)) (oe:exp) =
  fv_in_env Γ oe ∧
  ∀ b (γλ:gsub Γ b) (γ★:eval_env Γ) (h:history).
    (γ★ `(∼) h` γλ ∧ φ (γ★)) ⟹ t ⊇ (h, fs_oe γ, gsubst γλ oe)
```

This modification does not affect the compatibility lemmas in a meaningful way because the precondition shows up on the left side of the implication. It affects, however, our proof of compilation. Before, the proof first established that the IO* program and the generated λ_{io} program are in the relation for open expressions, and then it concluded that they are related in the relation for closed expressions by using the empty evaluation environment. Now, one has to do the extra step to prove that the precondition is satisfied for the empty evaluation environment. This is

why we had to modify our model slightly by requiring that the source program has a satisfiable precondition. The modified model supports partial programs that possibly use refinement types and have a simply typed interface.

```
type progS (i:interface) =
  ps:(i.ct → io bool) &
  φ:(eval_env empty →  Type0){φ empty_eval} & (typing empty φ (i.ct → io bool) ps)
```

Unfortunately, we cannot build a proof that the precondition is satisfied using the metaprogram because F⋆ saves neither the precondition it builds nor a proof that the precondition is satisfiable. Since the main way to prove the preconditions in F⋆ is by using the SMT, we tried to compute the same preconditions as F⋆ so that if the SMT succeeded initially when the program was type-checked, it should also succeed when quoting, which seems to work well in practice. It is worth noting that related work on certifying extraction runs into a similar issue [39, 40].

[39]: Myreen et al. (2014), *Proof-producing Translation of Higher-order logic into Pure and Stateful ML*
[40]: Abrahamsson et al. (2020), *Proof-Producing Synthesis of CakeML from Monadic HOL Functions*

5.7 SEIO⋆ in Action

This section provides more details on how to run the running example. In §5.7.1, we first present a concrete **validate** function and agent definitions that make this **validate** hold or not. Later, in §5.7.2, we briefly explain how to connect our framework with other tools to obtain an executable file that allows us to test SEIO⋆ in action.

5.7.1 A Verified Wrapper for Unverified AI Agents

We tested our framework with the verified wrapper for agents we presented in §5.2. For the **validate** construction we used a very simple condition:

```
let validate olds task news = eq_string task news
```

that tests whether the task description coincides with the new content of the file. The function **eq_string** is assumed as a primitive constant of λ_{io}. Other string operations, such as concatenation, could be added and treated in a similar fashion.

The verified wrapper can be run together with agents written directly as λ_{io} expressions. Even when the primitives present currently in λ_{io} are limited, we still tested a number of things. These are some agents we have tried that work as intended:

```
(* bad: this agent does nothing *)
let lazy_agent : exp = ELam (ELam EUnit)

(* good: this agent writes the expected string on the file *)
let write_agent : exp =
  ELam (ELam (ECase (EOpen (EVar 1))
             (EApp (ELam (EClose (EVar 1))) (EWrite (EVar 0) (EVar 1)))
             EUnit))

(* bad: this agent writes the string to the file twice *)
let write_twice_agent : exp =
  ELam (ELam (ECase (EOpen (EVar 1))
             (EApp (ELam (EApp (ELam (EClose (EVar 2)))
                 (EWrite (EVar 1) (EVar 2))))
                 (EWrite (EVar 0) (EVar 1))) EUnit))
```

5.7.2 Unverified Compilation Step from λ_{io} to $\lambda_{\square}$

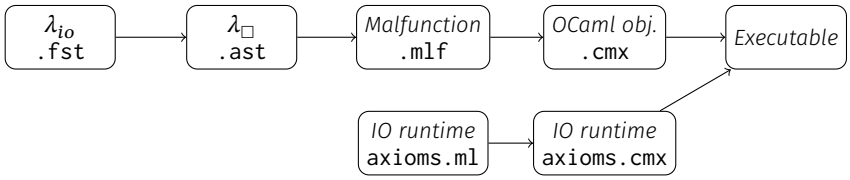


Figure 5.4: Compilation pipeline from λ_{io} to executable.

To build an actual executable from the programs we can write in SEIO*, we complement the proposed extraction mechanism with an unverified step compiling from λ_{io} to $\lambda_{\square}$ (*lambdabox*), and then using the tooling provided by the Peregrine project [143] to compile to OCaml (Malfunction [28, 148]) to obtain code that can be later linked with a runtime in OCaml.

The $\lambda_{\square}$ language was introduced in the context of program extraction for Rocq [4]. It is a variant of untyped λ-calculus with a construction $\square$ (box) that replaces proofs and types that are computationally irrelevant. In our use case, we do not erase proofs, as λ_{io} already does not contain any, and we use it just as an intermediate language that allows us to use the rest of the compilation pipeline provided by Peregrine.

Using Peregrine, the $\lambda_{\square}$ code is translated to Malfunction, which is a low-level untyped program representation that can be used to compile to OCaml objects. The Peregrine framework uses this program representation as its output for OCaml. This output can later be compiled using Malfunction tooling [148] to generate OCaml objects, which can be linked with other OCaml objects that implement IO primitives such as `openfile`, `readfile`, etc. as well as other primitives of $\lambda_{\square}$ such as string equality. This compilation and linking sequence is presented in Figure 5.4.

[143]: Chapman et al. (n.d.), *Peregrine Project: a unified middle-end for code generation from proof assistants*

[28]: Forster et al. (2024), *Verified Extraction from Coq to OCaml*

[148]: Dolan (2016), *Malfunctional Programming*

[4]: Letouzey (2004), *Programmation fonctionnelle certifiée : L'extraction de programmes dans l'assistant Coq. (Certified functional programming: Program extraction within Coq proof assistant)*

5.8 Related Work

Certifying Extraction Work on *CakeML synthesis* [39, 40] shows how starting from a shallowly embedded program in HOL one can synthesize an equivalent CakeML [149] program (a subset of Standard ML). Their synthesis happens outside of HOL, in Standard ML, and it produces a deeply embedded program in HOL and a correctness proof that has to be accepted by HOL, which is a form of translation validation. They show how to synthesize CakeML programs from HOL functions that feature polymorphism, first-order references, arrays, exceptions, and IO operations. We think we can extend SEIO* with these features by following their lead to obtain certifying extraction from F* with a smaller TCB and stronger security guarantees than in their work [39, 40]. Another difference comes from the languages we use, they work in HOL, a simply typed language, while we work in F*, a dependently typed language, which enabled us to factor out relational quotation as a separate step.

Œuf [41] is a compiler with end-to-end guarantees from a simply typed subset of Gallina (the language of Rocq) to Assembly code. They use translation validation for quotation, which produces a deeply embedded program. They validate this by showing in Rocq that the denotation of the deep embedding is equal to the Gallina program. They define denotation as a pure Rocq function that takes a deeply embedded program and produces a shallowly embedded one. This is similar to what happens in relational quotation, where the typing derivation also constructs a program that is checked to be equal with the program that is quoted. From our experience, using

a typing relation works better than separately defining a deep embedding and a denotation function. For example, it is more natural to keep in the typing derivation extra information necessary to reconstruct the shallow embedding, information which may have no place in a standard deep embedding. We used relational quotation for a language with the IO effect, while Œuf has no support for effects.

Forster and Kunze [42] present a certifying extraction from pure Rocq functions of simple polymorphic types to an untyped call-by-value λ-calculus. They present a plugin for Rocq (a metaprogram) that given a shallowly embedded program, produces a deeply embedded program. The plugin comes with a series of tactics that automate the process of validating that the two programs are related. They have no support for effects, while SEIO* supports IO.

[42]: Forster et al. (2019), *A Certifying Extraction with Time Bounds from Coq to Call-By-Value Lambda Calculus*

Pit-Claudel et al. [38] present a Rocq framework that treats compilation as a proof-search problem—i.e., finding a deeply embedded low-level program that is in a relation with the shallowly embedded program. This is also a form of translation validation, since the search is done using tactics (metaprograms) whose result has to be type-checked by Rocq. The framework may not know how to automatically compile a program, in which case it shows where it got stuck, and the proof engineer can take control. They illustrate the versatility of the framework by showing how to compile Gallina programs using terminating loops, various flat data structures such as mutable first-order cells and arrays, nondeterminism, and IO. Pit-Claudel et al. [38] present their framework as a translation validation tool for specific performance-critical programs, directly targeting a low-level program. We on the other hand try to minimize the reliance on translation validation.

[38]: Pit-Claudel et al. (2022), *Relational compilation for performance-critical applications: extensible proof-producing translation of functional models into low-level code*

The work above and our relational quotation *share a challenge* that comes from trying to relate a shallowly embedded program with a deeply embedded program: one is able to relate only what is expressible by a user defined relation (e.g., pattern matching is hard to do: when synthesizing CakeML [39, 40], they first automatically generate the relation based on the datatypes used).

Typing Relation for Shallow Embeddings *McBride [150]* presents the first deep embedding indexed by shallow types, and claims it is a "new reflective technology". McBride's idea was advanced by *Prinz, Kavvos, and Lampropoulos [43]*, who show how to do a "deeper shallow embedding" of a dependently-typed language by using an embedding very similar to our typing relation for shallow embeddings. They show an embedding that supports dependent types and/or affine types, something we did not try. We show, however, how to support monadic code and refinement types. To our knowledge, we are the first to use such a typing relation for doing quotation and certifying extraction.

[150]: McBride (2010), *Outrageous but meaningful coincidences: dependent type-safe syntax and evaluation*

[43]: Prinz et al. (2022), *Deeper Shallow Embeddings*

Matsuda et al. [151] propose a framework that leverages unembedding, a reification technique that converts finally-tagless (shallow) representations to de Bruijn-indexed (deep) terms, with formal correctness guarantees. Unembedding does not apply to our monadic shallowly embedded code because it is not in finally-tagless style.

[151]: Matsuda et al. (2023), *Embedding by Unembedding*

Some similarities exist also with work on canonical structures [152].

Logical Relations Our logical relations, and their related proofs, follow the logical relation for semantic typing presented by Ahmed [147] (§2.2) for a λ-calculus extended with dynamically allocated immutable cells. We differ from it in three ways: our logical relations are cross-language [141, 142] and asymmetric, connecting a shallow and a deep embedding. They relate a program of a FGCBV language to one of a CBV language, and relates two trace-producing semantics.

[147]: Ahmed (2004), *Semantics of Types for Mutable State*

Asymmetric relations are common in work on certified extraction, and Forster and Kunze [42] and work on CakeML synthesis [39, 40] also use logical relations. The most complex relation is used in CakeML synthesis: it supports multiple effects (first-order state, exceptions and IO), polymorphic functions and total recursive functions, and relates evaluation semantics. Our logical relations are different because they relate trace-producing semantics and they imply the strongest secure compilation criterion (RrHP) of Abate et al. [8].

[8]: Abate et al. (2019), *Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation*

Logical relations were used to prove fully abstract compilation [11, 91, 153] and Abate et al. [8] later show that they can also be used to prove their stronger RrHP secure compilation criterion for a simple translation from a statically typed language to a dynamically typed one.

Verified Extraction CakeML [149] is a subset of Standard ML that has a verified compiler that can target ARMv6, ARMv8, x86-x64, MIPS-64, RISC-V and Silver RSA architectures. Together with their synthesis from HOL to CakeML, they have end-to-end guarantees of correctness from the original program written in HOL to the low-level code that targets a specific architecture. It would be interesting to extend SEIO* to target CakeML with a verified secure compilation step.

Hupel and Nipkow [33] and Nezamabadi, Myreen, and Tan [32] present verified extraction from Isabelle/HOL and Dafny to CakeML, but there is no formal connection between the formalized meta-theory of Isabelle/HOL or Dafny and the languages themselves.

[33]: Hupel et al. (2018), *A Verified Compiler from Isabelle/HOL to CakeML*

[32]: Nezamabadi et al. (2026), *Verified VCG and Verified Compiler for Dafny*

Forster, Sozeau, and Tabareau [28] present a verified compiler from Gallina to Malfunction (an intermediate untyped language of OCaml). Their compiler is based on the existing MetaRocq [37] project that formalizes Rocq in itself. They prove a realizability relation that guarantees that the extracted untyped code operationally behaves as the initial Rocq program. They do not support effectful code, and quotation is neither verified nor certifying.

[28]: Forster et al. (2024), *Verified Extraction from Coq to OCaml*

CertiCoq [30] and ConCert [29, 31] are two partially verified compilers for Rocq that target C, Elm, Rust, WebAssembly and blockchain languages. They also rely on unverified quotation.

Secure Compilation of Verified Code There are two secure compilation frameworks for F*, SCIO* [44] for verified IO programs, and SecRef* [45] for verified stateful programs. SCIO* and SecRef* look at how to convert refinement types, pre- and postconditions into dynamic checks using higher-order contracts and a reference monitor. SEIO* is orthogonal to them, since it focuses on providing guarantees for the extraction step itself. They are both proved to satisfy RrHP, but they are not fully verified: they rely on the unverified extraction mechanism of F* to OCaml. Moreover, even if we prove the same criterion, the proofs are very different. Their proof [44, 45] of RrHP follows directly from a syntactic inversion law, while our SEIO* proof of RrHP is semantic and involves two logical relations. This difference is necessary, because while SCIO*/SecRef* only compiles between shallow embeddings, our SEIO* work extracts a shallow embedding to a deeply embedded language with a standard operational semantics. Beyond such secure extraction from proof assistants, there are many other applications of secure compilation [8, 11].

5.9 Summary and Future Work

We have shown that formally secure extraction of shallowly embedded effectful programs can be achieved by structuring certifying extraction in a new way that confines translation validation only to a first step—relational quotation—while using a once-and-for-all verified syntax generation function for the rest. We have illustrated this general idea for F* by developing SEIO*, a framework that extracts F* programs with IO while providing machine-checked secure compilation guarantees (RrHP).

SEIO* supports extraction of programs with file-based IO, whose types are simple types, possibly decorated with refinements. In the future, it will be interesting to address some of its limitations, and add support for recursive functions and dependent types. To quote recursive F* functions we would need F* to expose the combinators used to define recursive functions. For now, we defined an iterator for the specific inductive type of natural numbers and showed how to extract functions written in terms of that iterator [41]. We think this method could scale to a user defined fixpoint combinator [38–40], but the combinator would still not be idiomatic and it would be difficult to use compared to F*'s native support for recursion. To support dependent types, we would have to modify the typing relation to be closer to Prinz, Kavvos, and Lampropoulos [43] by having the F* types explicitly depend on the evaluation environment, just like we did for values. We expect that quoting dependently-typed programs would still follow the syntactic structure of the program and the type checking would be similar to how it works for simply typed programs. Redoing the proofs of security would require to also extend the logical relations. Logical relations for dependent types tend to be much more involved and rely on strong meta-theoretical assumptions such as induction-recursion, [154] on impredicativity [155–157], or alternatively by considering only a finite number of universes [158, 159]. While this seems mostly orthogonal from the focus of our work, the intricacies of logical relations prevent us from making a claim about the difficulty or feasibility of the extension.

Improving the performance of the relational quotation metaprogram for refinement types is necessary before doing other extensions. We show how to support refinement types by defining a typing relation that computes a precondition and has one typing rule specific to refinements. Having the precondition as an argument on each typing rule consumes a lot of memory because the precondition grows with the size of the derivation. The metaprogram also overuses the subtyping rule (QRef), by using it in all positions where typing information comes from the environment, but not all positions require subtyping.

A different direction, inspired by Prinz, Kavvos, and Lampropoulos [43], is to strengthen the typing relation used in relational quotation so that it tracks how often each variable is used, yielding an affine or linear discipline. This would turn relational quotation into a technique for *layering additional type checking on top of F**—for instance, enforcing that mutable references are used affinely even though F* has no native support for affine types.

[41]: Mullen et al. (2018), *Œuf: minimizing the Coq extraction TCB*

[38]: Pit-Claudel et al. (2022), *Relational compilation for performance-critical applications: extensible proof-producing translation of functional models into low-level code*

[39]: Myreen et al. (2014), *Proof-producing Translation of Higher-order logic into Pure and Stateful ML*

[40]: Abrahamsson et al. (2020), *Proof-Producing Synthesis of CakeML from Monadic HOL Functions*

[154]: Abel (2013), *Normalization by evaluation: Dependent types and impredicativity*

[155]: Adjedj et al. (2024), *Martin-löf à la coq*

[156]: Jang et al. (2025), *McTT: A Verified Kernel for a Proof Assistant*

[157]: Liu et al. (2024), *Functional Pearl: Short and Mechanized Logical Relation for Dependent Type Theories*

[158]: Abel et al. (2017), *Decidability of conversion for type theory in type theory*

[159]: Poiret et al. (2026), *Divide and Check: Logical Relations, No Algorithms Attached*

6 Conclusion

Securely compiling verified code and linking it with adversarial unverified code is a general open research problem for all proof-oriented programming languages (Rocq, Isabelle/HOL, Dafny, etc). In this thesis we make substantial progress towards solving it by proposing three formally secure compilation frameworks. In particular, we provide machine-checked proofs that our frameworks soundly preserve a global specification and, moreover, satisfy a secure compilation criterion called RrHP, which is stronger than full abstraction. Next, we discuss how the contributions of this thesis could be applied to other proof-oriented languages, and suggest how to evaluate the performance of the dynamic checks that our frameworks add.

Applying Our Results to Other Proof-Oriented Languages. While our contributions are developed in the context of F*, we believe that many ideas of this thesis could be applied to solving the same general problem also in other proof-oriented programming languages, especially those based on dependent type theory.

To recap, our frameworks are built around the following ingredients: Dijkstra monads (§2.3.1), higher-order contracts (§3.4), reference monitoring (§3.5), relational quotation (§5.3), and the soundness and RrHP proofs (§3.6, §4.5, and §5.5). What these require from the host language is dependent types. SMT automation is not needed for any of our results, but it is what makes verifying programs in our frameworks practical.

Dijkstra monads were recently implemented in Lean by Loom [70], a verification framework for effectful Lean programs that, like F*, combines them with tactics and SMT automation, which makes it a promising host for porting SCIO* (Chapter 3) and SecRef* (Chapter 4). Dijkstra monads have also been implemented in Rocq [9, 68], so in principle a direct port to Rocq of our whole SCIO* and SecRef* frameworks seems possible, just that the Dijkstra monads will not be as usable without support for discharging the gathered verification conditions with an SMT solver.

The ideas behind our frameworks are, however, not tied to Dijkstra monads. For example, in Rocq a more usable solution could probably be built more quickly by targeting an interactive verification framework for monadic programs, such as Ynot [71], FreeSpec [60], ITrees [79], or Iris [160]. While the precise details will vary based on the choice of verification framework, some general ideas that we expect to be portable to other frameworks are: (1) the internal representation of the Dijkstra monad as a freer monad is directly compatible with how FreeSpec and ITrees represent computations, and which could provide a model to the axioms used by Ynot; (2) the formally verified combination of higher-order contracts and reference monitoring; (3) the soundness proof and the secure compilation proof of RrHP, which should stay simple even in these frameworks, since all the main ingredients seem portable.

Relational quotation and the general idea to split extraction into two distinct phases also do not depend on F*—only on dependent types—so they could also be applied to more standard dependently typed proof assistants like Rocq and Lean.

Performance Evaluation. It would be interesting to do a performance evaluation of the added reference monitor and higher-order contracts. SCIO* and SecRef* already add the minimal necessary dynamic checks, and the user of the frameworks can choose how efficiently they are performed by providing their implementation. To improve performance further, one could try to extend our frameworks so that they statically analyze unverified code and discards the contracts using soft contract verification [112, 161].

[70]: Gladshtein et al. (2026), *Foundational Multi-Modal Program Verifiers*

[9]: Maillard et al. (2019), *Dijkstra Monads for All*

[68]: Silver et al. (2021), *Dijkstra monads forever: termination-sensitive specifications for interaction trees*

[71]: Malecha et al. (2011), *Trace-based verification of imperative programs with I/O*

[60]: Letan et al. (2020), *FreeSpec: specifying, verifying, and executing impure computations in Coq*

[160]: Vistrup et al. (2025), *Program Logics à la Carte*

7 Towards Formal Secure Extraction to OCaml

We see the frameworks presented in this thesis as important steps towards building a formally secure compilation chain from F⋆ to a safe subset of OCaml. Building such a compiler would face multiple challenges in modeling a bigger safe subset of OCaml in a way that verification is practical, and obtaining formal end-to-end secure compilation guarantees like RrHP.

Modeling More of the Effects of OCaml require a significant extension of the frameworks, yet monadic representations are expressive enough [162, 163]. A particular challenge is modeling *higher-order store* for effectful functions around F⋆'s predicative, countable hierarchy of universes, for which only partial solutions exist in related work (§2.6). Extending relational quotation and SEIO⋆ (Chapter 5) to more effects should be possible as indicated by prior work on CakeML synthesis [39, 40] and relational compilation [38].

[39]: Myreen et al. (2014), *Proof-producing Translation of Higher-order logic into Pure and Stateful ML*
[40]: Abrahamsson et al. (2020), *Proof-Producing Synthesis of CakeML from Monadic HOL Functions*
[38]: Pit-Claudel et al. (2022), *Relational compilation for performance-critical applications: extensible proof-producing translation of functional models into low-level code*

After extending the representation of the monadic effect, then one has to figure out how this impacts the soundness and security guaranteed by our frameworks. While we studied these separately for IO in SCIO⋆ (Chapter 3) and for mutable state in SecRef⋆ (Chapter 4), how other effects would impact soundness and security is an open question. For example, adding non-termination would involve assuming that untrusted code can always loop.

Finally, modeling more effects will impact how easy it is to verify programs. In this thesis, we showed that we can verify sizeable case studies when using the IO and mutable state effects separately. This was possible because they were designed to take advantage of SMT automation. Further, combining multiple effects, like IO and mutable state raises questions about how the specifications should look like (i.e., traces vs relation between heaps) and what kind of properties one wants to verify. Probably some form of doing verification under effect polymorphism could help reduce the effort [164]. One can also consider using Separation Logic to do verification: Vistrup, Sammler, and Jung [160] show how to use ITrees to embed and verify programs using Iris. Initial support for Separation Logic in F⋆ required extensive usage of tactics [83, 84], and more recent projects improved on the automation, but departed from the effect mechanism of F⋆ [85].

[164]: Vilhena et al. (2021), *A separation logic for effect handlers*

[85]: Ebner et al. (2025), *PulseCore: An Impredicative Concurrent Separation Logic for Dependently Typed Programs*

End-to-End Verified Extraction to OCaml. Connecting the SCIO⋆ (Chapter 3) and SEIO⋆ (Chapter 5) frameworks raises several challenges: first, SCIO⋆ works around F⋆'s lack of general effect polymorphism by making use of a restricted form called flag polymorphism to model that the context is prevented from directly calling the IO primitives to avoid the access control done by a reference monitor (§3.2.4). Second, SCIO⋆ makes use of F⋆'s effect mechanism (§2.3), and while effects can be reified to their underlying monadic representation [103], support for reasoning about the obtained monadic computations is at the moment still too limited in F⋆ to be able to do the extrinsic proofs that would be needed to connect the two frameworks. It is worth noting that the two challenges above are specific to F⋆.

[103]: Grimm et al. (2018), *A Monadic Framework for Relational Verification: Applied to Information Security, Program Equivalence, and Optimizations*

The other missing step is connecting SEIO⋆ to OCaml. To build an actual executable, SEIO⋆ currently complements the verified extraction to λ_{io} (§5.4) with an unverified step compiling from λ_{io} to $\lambda_{\square}$, followed by the compilation pipeline of the Peregrine project [143], which compiles $\lambda_{\square}$ to a low-level untyped intermediate language of

[143]: Chapman et al. (n.d.), *Peregrine Project: a unified middle-end for code generation from proof assistants*

OCaml (§5.7), called Malfunction [28, 148]. To obtain end-to-end formal security guarantees, these steps would have to be verified too: the step from $\lambda_{\square}$ to Malfunction was verified by Forster, Sozeau, and Tabareau [28] for pure programs, so their result has to be extended to support monadic programs and extended to verify security.

[28]: Forster et al. (2024), *Verified Extraction from Coq to OCaml*
[148]: Dolan (2016), *Malfunctional Programming*

Bibliography

References in citation order.

[1] Zakir Durumeric et al. “The Matter of Heartbleed.” In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. IMC ’14. Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 475–488. DOI: 10.1145/2663716.2663755 (cited on page 1).

[2] Nikhil Swamy et al. “Dependent Types and Multi-Monadic Effects in F*.” In: *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, Jan. 2016, pp. 256–270 (cited on pages 1, 12, 13, 55).

[3] Aseem Rastogi et al. *Programming and Proving with Indexed Effects*. July 2021. URL: https://www.fstar-lang.org/papers/indexedeffects/ (cited on pages 1, 12, 13, 20, 27, 28, 38, 44, 80).

[4] Pierre Letouzey. “Programmation fonctionnelle certifiée : L’extraction de programmes dans l’assistant Coq. (Certified functional programming: Program extraction within Coq proof assistant).” PhD thesis. University of Paris-Sud, Orsay, France, 2004 (cited on pages 1, 106, 112).

[5] Jonathan Protzenko et al. “Verified Low-Level Programming Embedded in F*.” In: *PACMPL* 1.ICFP (Sept. 2017), 17:1–17:29. DOI: 10.1145/3110261 (cited on pages 1, 36, 52).

[6] Jonathan Protzenko et al. “EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider.” In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 983–1002. DOI: 10.1109/SP40000.2020.00114 (cited on pages 1, 4, 24, 30).

[7] Danel Ahman et al. *Project Everest: Perspectives from Developing Industrial-grade High-Assurance Software*. Draft. Sept. 2025. URL: https://project-everest.github.io/assets/everest-perspectives-2025.pdf (cited on page 1).

[8] Carmine Abate et al. “Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation.” In: *32nd IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 2019, pp. 256–271. DOI: 10.1109/CSF.2019.00025 (cited on pages 2, 3, 7, 33, 55–57, 60, 79, 82, 86, 89, 105–108, 114).

[9] Kenji Maillard et al. “Dijkstra Monads for All.” In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019). DOI: 10.1145/3341708 (cited on pages 2, 8, 11, 15, 18, 19, 21, 27–30, 38, 54, 55, 80, 100, 117).

[10] Danel Ahman et al. “Recalling a Witness: Foundations and Applications of Monotonic State.” In: *PACMPL* 2.POPL (2018), 65:1–65:30 (cited on pages 2, 4, 11, 24–26, 28, 30, 63, 65, 66, 84).

[11] Marco Patrignani, Amal Ahmed, and Dave Clarke. “Formal Approaches to Secure Compilation: A survey of fully abstract compilation and related work.” In: *ACM Computing Surveys* (2019) (cited on pages 3, 7, 33, 56, 60, 114).

[12] James Anderson. *Computer Security Technology Planning Study*. ESD-TR-73-51, US Air Force Electronic Systems Division (1973). Section 4.1.1 http://csrc.nist.gov/publications/history/ande72.pdf. 1973. URL: http://csrc.nist.gov/publications/history/ande72.pdf (cited on pages 3, 4).

[13] D.M. Ritchie and K. Thompson. *The UNIX time-sharing system*. The Bell System Technical Journal, 57(6 (part 2)):1905+. 1978 (cited on pages 3, 4).

[14] Stanley R. Ames Jr. “Security Kernels: A Solution or a Problem?” In: *1981 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 27-29, 1981*. IEEE Computer Society, 1981, p. 141. DOI: 10.1109/SP.1981.10016 (cited on pages 3, 4).

[15] Robert Bruce Findler and Matthias Felleisen. “Contracts for higher-order functions.” In: *7th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Ed. by Mitchell Wand and Simon L. Peyton Jones. ACM, 2002, pp. 48–59. DOI: 10.1145/581478.581484 (cited on pages 3, 4, 39, 46, 60, 69, 77).

[16] Tim Disney, Cormac Flanagan, and Jay McCarthy. “Temporal higher-order contracts.” In: *16th ACM SIGPLAN international conference on Functional Programming (ICFP)*. Ed. by Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy. ACM, 2011, pp. 176–188. DOI: 10.1145/2034773.2034800 (cited on pages 3, 4, 60).

[17] Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. “Computational contracts.” In: *Sci. Comput. Program.* 98 (2015), pp. 360–375. DOI: 10.1016/j.scico.2013.09.005 (cited on pages 3, 4, 60).

[18] Scott Moore et al. “Extensible access control with authorization contracts.” In: *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Ed. by Eelco Visser and Yannis Smaragdakis. ACM, 2016, pp. 214–233. DOI: 10.1145/2983990.2984021 (cited on pages 3, 4, 60).

[19] Jesse A. Tov and Riccardo Pucella. “Stateful Contracts for Affine Types.” In: *19th European Symposium on Programming (ESOP)*. Ed. by Andrew D. Gordon. Vol. 6012. Lecture Notes in Computer Science. Springer, 2010, pp. 550–569. DOI: 10.1007/978-3-642-11957-6_29 (cited on pages 3, 4, 60).

[20] Jean-Karim Zinzindohoué et al. “HACL*: A Verified Modern Cryptographic Library.” In: *ACM Conference on Computer and Communications Security*. ACM, 2017, pp. 1789–1806 (cited on page 4).

[21] Nikhil Swamy et al. “Hardening attack surfaces with formally proven binary format parsers.” In: *PLDI ’22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. Ed. by Ranjit Jhala and Isil Dillig. ACM, 2022, pp. 31–45. DOI: 10.1145/3519939.3523708 (cited on page 4).

[22] Derek Dreyer, Georg Neis, and Lars Birkedal. “The impact of higher-order state and control effects on local relational reasoning.” In: *J. Funct. Program.* 22.4-5 (2012), pp. 477–528. DOI: 10.1017/S095679681200024X (cited on page 5).

[23] John C. Reynolds. “Separation Logic: A Logic for Shared Mutable Data Structures.” In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817 (cited on pages 5, 65, 84, 85).

[24] David Swasey, Deepak Garg, and Derek Dreyer. “Robust and Compositional Verification of Object Capability Patterns.” In: *PACMPL* 1.OOPSLA (2017), 89:1–89:26. DOI: 10.1145/3133913 (cited on pages 5, 61, 84, 85).

[25] Michael Sammler et al. “The high-level benefits of low-level sandboxing.” In: *Proc. ACM Program. Lang.* 4.POPL (2020), 32:1–32:32. DOI: 10.1145/3371100 (cited on pages 5, 61, 84, 85).

[26] Aïna Linn Georges et al. “Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code.” In: *J. ACM* 71.1 (2024), 3:1–3:59. DOI: 10.1145/3623510 (cited on pages 5, 85).

[27] Pieter Agten, Bart Jacobs, and Frank Piessens. “Sound Modular Verification of C Code Executing in an Unverified Context.” In: *42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 581–594. DOI: 10.1145/2676726.2676972 (cited on pages 5, 60, 85).

[28] Yannick Forster, Matthieu Sozeau, and Nicolas Tabareau. “Verified Extraction from Coq to OCaml.” In: *Proc. ACM Program. Lang.* 8.PLDI (2024), pp. 52–75. DOI: 10.1145/3656379 (cited on pages 7, 106, 112, 114, 120).

[29] Wolfgang Meier et al. “CertiCoq-Wasm: A Verified WebAssembly Backend for CertiCoq.” In: *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP ’25. Denver, CO, USA: Association for Computing Machinery, 2025, pp. 127–139. DOI: 10.1145/3703595.3705879 (cited on pages 7, 114).

[30] Abhishek Anand et al. “CertiCoq: A verified compiler for Coq.” In: *3rd Workshop on Coq for Programming Languages (CoqPL)*. 2017 (cited on pages 7, 36, 114).

[31] Danil Annenkov et al. “Extracting functional programs from Coq, in Coq.” In: *Journal of Functional Programming* 32 (2022), e11. DOI: 10.1017/S0956796822000077 (cited on pages 7, 114).

[32] Daniel Nezamabadi, Magnus O. Myreen, and Yong Kiam Tan. “Verified VCG and Verified Compiler for Dafny.” In: *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP ’26. Rennes, France: Association for Computing Machinery, 2026, pp. 171–186. DOI: 10.1145/3779031.3779092 (cited on pages 7, 114).

[33] Lars Hupel and Tobias Nipkow. “A Verified Compiler from Isabelle/HOL to CakeML.” In: *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Ed. by Amal Ahmed. Vol. 10801. Lecture Notes in Computer Science. Springer, 2018, pp. 999–1026. DOI: 10.1007/978-3-319-89884-1_35 (cited on pages 7, 114).

[34] Mario Carneiro. “Lean4Lean: Towards a formalized metatheory for the Lean theorem prover.” In: *CoRR* abs/2403.14064 (2024). DOI: 10.48550/ARXIV.2403.14064 (cited on page 7).

[35] Bohdan Liesnikov and Jesper Cockx. “Building a Correct-by-Construction Type Checker for a Dependently Typed Core Language.” In: *Programming Languages and Systems - 22nd Asian Symposium, APLAS 2024, Kyoto, Japan, October 22-24, 2024, Proceedings*. Ed. by Oleg Kiselyov. Vol. 15194. Lecture Notes in Computer Science. Springer, 2024, pp. 63–83. DOI: 10.1007/978-981-97-8943-6_4 (cited on page 7).

[36] Joshua M. Cohen and Philip Johnson-Freyd. “A Formalization of Core Why3 in Coq.” In: *Proc. ACM Program. Lang.* 8.POPL (Jan. 2024). DOI: 10.1145/3632902 (cited on page 7).

[37] Matthieu Sozeau et al. “Correct and Complete Type Checking and Certified Erasure for Coq, in Coq.” In: *J. ACM* 72.1 (Jan. 2025). DOI: 10.1145/3706056 (cited on pages 7, 114).

[38] Clément Pit-Claudel et al. “Relational compilation for performance-critical applications: extensible proof-producing translation of functional models into low-level code.” In: *PLDI ’22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. Ed. by Ranjit Jhala and Isil Dillig. ACM, 2022, pp. 918–933. DOI: 10.1145/3519939.3523706 (cited on pages 7, 91, 94, 113, 115, 119).

[39] Magnus O. Myreen and Scott Owens. “Proof-producing Translation of Higher-order logic into Pure and Stateful ML.” In: *Journal of Functional Programming (JFP)* 24.2-3 (May 2014), pp. 284–315. DOI: 10.1017/S0956796813000282 (cited on pages 7, 91, 94, 111–115, 119).

[40] Oskar Abrahamsson et al. “Proof-Producing Synthesis of CakeML from Monadic HOL Functions.” In: *Journal of Automated Reasoning (JAR)* (2020) (cited on pages 7, 91, 94, 111–115, 119).

[41] Eric Mullen et al. “Œuf: minimizing the Coq extraction TCB.” In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2018. Los Angeles, CA, USA: Association for Computing Machinery, 2018, pp. 172–185. DOI: 10.1145/3167089 (cited on pages 7, 91, 94, 112, 115).

[42] Yannick Forster and Fabian Kunze. “A Certifying Extraction with Time Bounds from Coq to Call-By-Value Lambda Calculus.” In: *10th International Conference on Interactive Theorem Proving, ITP 2019, Portland, OR, USA, September 9-12, 2019*. Ed. by John Harrison, John O’Leary, and Andrew Tolmach. Vol. 141. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 17:1–17:19. DOI: 10.4230/LIPICS.ITP.2019.17 (cited on pages 7, 91, 94, 113, 114).

[43] Jacob Prinz, G. A. Kavvos, and Leonidas Lampropoulos. “Deeper Shallow Embeddings.” In: *13th International Conference on Interactive Theorem Proving, ITP 2022, Haifa, Israel, August 7-10, 2022*. Ed. by June Andronick and Leonardo de Moura. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 28:1–28:18. DOI: 10.4230/LIPICS.ITP.2022.28 (cited on pages 7, 89, 92, 113, 115).

[44] Cezar-Constantin Andrici et al. “Securing Verified IO Programs Against Unverified Code in F*.” In: *Proc. ACM Program. Lang.* 8.POPL (2024), pp. 2226–2259. DOI: 10.1145/3632916 (cited on pages 8, 114).

[45] Cezar-Constantin Andrici et al. “SecRef*: Securely Sharing Mutable References between Verified and Unverified Code in F*.” In: *Proc. ACM Program. Lang.* 9.ICFP (2025). DOI: 10.1145/3747522 (cited on pages 8, 114).

[46] Cezar-Constantin Andrici et al. “Misquoted No More: Securely Extracting F* Programs with IO.” In: *Proc. ACM Program. Lang.* 10.ICFP (2026) (cited on page 8).

[47] Théo Winterhalter et al. *Partial Dijkstra Monads for All*. TYPES. 2022. URL: https://types22.inria.fr/files/2022/06/TYPES_2022_paper_18.pdf (cited on pages 8, 11, 20).

[48] Cezar-Constantin Andrici, Théo Winterhalter, and Cătălin Hriţcu. *Verifying non-terminating programs with IO in F**. HOPE. 2022. URL: https://github.com/andricicezar/fstar-io/tree/hope-submission (cited on pages 8, 11).

[49] Oleg Kiselyov and Hiromi Ishii. “Freer monads, more extensible effects.” In: *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*. ACM, 2015, pp. 94–105. DOI: 10.1145/2804302.2804319 (cited on pages 11, 17, 18).

[50] Cezar-Constantin Andrici et al. *Artifact for the POPL 2024 paper ‘Securing Verified IO Programs Against Unverified Code in F*’*. Zenodo. Version 1. Nov. 2023. DOI: 10.5281/zenodo.10125015. URL: https://doi.org/10.5281/zenodo.10125015 (cited on pages 11, 34).

[51] Cezar-Constantin Andrici et al. *Artifact for the POPL 2024 paper ‘Securing Verified IO Programs Against Unverified Code in F*’*. GitHub. Version 1. Nov. 2023. URL: https://github.com/andricicezar/fstar-io/tree/popl24/sciostar (cited on pages 11, 34).

[52] Cezar-Constantin Andrici et al. *SecRef*: Securely Sharing Mutable References Between Verified and Unverified Code in F* - ICFP 2025 Artifact*. June 2025. DOI: 10.5281/zenodo.15659350 (cited on pages 11, 64).

[53] Cezar-Constantin Andrici et al. *Artifact for the ICFP 2025 paper ‘SecRef*: Securely Sharing Mutable References Between Verified and Unverified Code in F*’*. GitHub. Version 1. 2025 (cited on pages 11, 64).

[54] Nikhil Swamy, Guido Martínez, and Aseem Rastogi. *Proof-Oriented Programming in F* for v2026.03.24*. 2026. URL: https://github.com/FStarLang/PoP-in-FStar/tree/cd2974840ca7edc4e9c5ce9dac9286628162aa14 (cited on page 12).

[55] Nikhil Swamy et al. “Verifying Higher-order Programs with the Dijkstra Monad.” In: *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation*. PLDI ’13. 2013, pp. 387–398 (cited on page 13).

[56] Wouter Swierstra and Tim Baanen. “A predicate transformer semantics for effects (functional pearl).” In: *Proc. ACM Program. Lang.* 3.ICFP (2019), 103:1–103:26. DOI: 10.1145/3341707 (cited on page 14).

[57] Danel Ahman et al. “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 (cited on page 16).

[58] Heinrich Apfelmus. “The Operational Monad Tutorial.” In: *The Monad.Reader* 15 (2010), pp. 37–55 (cited on pages 17, 18).

[59] Oleg Kiselyov, Amr Sabry, and Cameron Swords. “Extensible effects: an alternative to monad transformers.” In: *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*. ACM, 2013, pp. 59–70. DOI: 10.1145/2503778.2503791 (cited on pages 17, 18).

[60] Thomas Letan and Yann Régis-Gianas. “FreeSpec: specifying, verifying, and executing impure computations in Coq.” In: *9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*. Ed. by Jasmin Blanchette and Catalin Hritcu. ACM, 2020, pp. 32–46. DOI: 10.1145/3372885.3373812 (cited on pages 18, 30, 117).

[61] Wouter Swierstra and Thorsten Altenkirch. “Beauty in the beast.” In: *ACM SIGPLAN Workshop on Haskell*. Ed. by Gabriele Keller. ACM, 2007, pp. 25–36. DOI: 10.1145/1291201.1291206 (cited on page 18).

[62] Andrej Bauer and Matija Pretnar. “Programming with algebraic effects and handlers.” In: *J. Log. Algebraic Methods Program.* 84.1 (2015), pp. 108–123. DOI: 10.1016/j.jlamp.2014.02.001 (cited on page 18).

[63] Li-yao Xia et al. “Interaction trees: representing recursive and impure programs in Coq.” In: *Proc. ACM Program. Lang.* 4.POPL (2020), 51:1–51:32. DOI: 10.1145/3371119 (cited on pages 18, 22).

[64] Danel Ahman and Tarmo Uustalu. “Update Monads: Cointerpreting Directed Containers.” In: *19th International Conference on Types for Proofs and Programs, TYPES 2013, April 22-26, 2013, Toulouse, France*. 2013, pp. 1–23. DOI: 10.4230/LIPIcs.TYPES.2013.1 (cited on page 19).

[65] Kurt Jensen. *Connection between Dijkstra’s Predicate-Transformers and Denotational Continuation-Semantics*. DAIMI Report Series 7.86. 1978. URL: http://ojs.statsbiblioteket.dk/index.php/daimipb/article/download/6502/5621 (cited on page 19).

[66] Sergey Goncharov, Stefan Milius, and Christoph Rauch. “Complete Elgot Monads and Coalgebraic Resumptions.” In: *Electronic Notes in Theoretical Computer Science* 325 (2016). The Thirty-second Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXII), pp. 147–168. DOI: https://doi.org/10.1016/j.entcs.2016.09.036 (cited on page 22).

[67] Sergey Goncharov et al. “Unifying Guarded and Unguarded Iteration.” In: *Proceedings of the 20th International Conference on Foundations of Software Science and Computation Structures - Volume 10203*. Berlin, Heidelberg: Springer-Verlag, 2017, pp. 517–533. DOI: 10.1007/978-3-662-54458-7_30 (cited on page 22).

[68] Lucas Silver and Steve Zdancewic. “Dijkstra monads forever: termination-sensitive specifications for interaction trees.” In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–28. DOI: 10.1145/3434307 (cited on pages 22, 29, 30, 117).

[69] Jason Reed. “A Hybrid Logical Framework.” PhD thesis. Carnegie Mellon University, 2009 (cited on page 29).

[70] Vladimir Gladshtein et al. “Foundational Multi-Modal Program Verifiers.” In: *Proc. ACM Program. Lang.* 10.POPL (2026), pp. 2233–2264. DOI: 10.1145/3776719 (cited on pages 29, 30, 117).

[71] Gregory Malecha, Greg Morrisett, and Ryan Wisnesky. “Trace-based verification of imperative programs with I/O.” In: *J. Symb. Comput.* 46.2 (2011), pp. 95–118. DOI: 10.1016/j.jsc.2010.08.004 (cited on pages 30, 117).

[72] Willem Penninckx, Bart Jacobs, and Frank Piessens. “Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs.” In: *24th European Symposium on Programming (ESOP)*. Ed. by Jan Vitek. Vol. 9032. Lecture Notes in Computer Science. Springer, 2015, pp. 158–182. DOI: 10.1007/978-3-662-46669-8_7 (cited on page 30).

[73] Willem Penninckx, Amin Timany, and Bart Jacobs. “Abstract I/O Specification.” In: *CoRR* abs/1901.10541 (2019) (cited on page 30).

[74] Bart Jacobs et al. “VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java.” In: *3rd International Symposium on NASA Formal Methods (NFM)*. Ed. by Mihaela Gheorghiu Bobaru et al. Vol. 6617. Lecture Notes in Computer Science. Springer, 2011, pp. 41–55. DOI: 10.1007/978-3-642-20398-5_4 (cited on page 30).

[75] Armaël Guéneau et al. “Verified Characteristic Formulae for CakeML.” In: *26th European Symposium on Programming (ESOP)*. Ed. by Hongseok Yang. Vol. 10201. Lecture Notes in Computer Science. Springer, 2017, pp. 584–610. DOI: 10.1007/978-3-662-54434-1_22 (cited on page 30).

[76] Hugo Férée et al. “Program Verification in the Presence of I/O - Semantics, Verified Library Routines, and Verified Applications.” In: *10th International Conference on Verified Software. Theories, Tools, and Experiments (VSTTE)*. Ed. by Ruzica Piskac and Philipp Rümmer. Vol. 11294. Lecture Notes in Computer Science. Springer, 2018, pp. 88–111. DOI: 10.1007/978-3-030-03592-1_6 (cited on page 30).

[77] Johannes Åman Pohjola, Henrik Rostedt, and Magnus O. Myreen. “Characteristic Formulae for Liveness Properties of Non-Terminating CakeML Programs.” In: *10th International Conference on Interactive Theorem Proving (ITP)*. Ed. by John Harrison, John O’Leary, and Andrew Tolmach. Vol. 141. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 32:1–32:19. DOI: 10.4230/LIPIcs.ITP.2019.32 (cited on page 30).

[78] Magnus O. Myreen. *A Hoare Logic for Diverging Programs*. https://github.com/HOL-Theorem-Prover/HOL/tree/develop/examples/Hoare-for-divergence. 2020 (cited on page 30).

[79] Li-yao Xia et al. “Interaction trees: representing recursive and impure programs in Coq.” In: *Proc. ACM Program. Lang.* 4.POPL (2020), 51:1–51:32. DOI: 10.1145/3371119 (cited on pages 30, 117).

[80] Hengchu Zhang et al. “Verifying an HTTP Key-Value Server with Interaction Trees and VST.” In: *12th International Conference on Interactive Theorem Proving (ITP)*. Ed. by Liron Cohen and Cezary Kaliszyk. Vol. 193. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 32:1–32:19. DOI: 10.4230/LIPIcs.ITP.2021.32 (cited on page 30).

[81] Ronghui Gu et al. “CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels.” In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Ed. by Kimberly Keeton and Timothy Roscoe. USENIX Association, 2016, pp. 653–669 (cited on page 30).

[82] Eleftherios Ioannidis et al. “Structural Temporal Logic for Mechanized Program Verification.” In: *Proc. ACM Program. Lang.* 9.OOPSLA2 (2025), pp. 1148–1175. DOI: 10.1145/3763091 (cited on page 30).

[83] Nikhil Swamy et al. “SteelCore: An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs.” In: *Proc. ACM Program. Lang.* 4.ICFP (Aug. 2020). DOI: 10.1145/3409003 (cited on pages 30, 119).

[84] Aymeric Fromherz et al. “Steel: proof-oriented programming in a dependently typed concurrent separation logic.” In: *Proc. ACM Program. Lang.* 5.ICFP (2021), pp. 1–30. DOI: 10.1145/3473590 (cited on pages 30, 52, 119).

[85] Gabriel Ebner et al. “PulseCore: An Impredicative Concurrent Separation Logic for Dependently Typed Programs.” In: *Proc. ACM Program. Lang.* PLDI (2025). DOI: 10.1145/3729311 (cited on pages 30, 71, 119).

[86] Paulette Koronkevich and William J. Bowman. “Type Universes as Kripke Worlds.” In: *Proc. ACM Program. Lang.* 9.ICFP (2025) (cited on page 30).

[87] P. J. Landin. “The Mechanical Evaluation of Expressions.” In: *The Computer Journal* 6.4 (Jan. 1964), pp. 308–320. DOI: 10.1093/comjnl/6.4.308 (cited on page 30).

[88] Dan Frumin, Amin Timany, and Lars Birkedal. “Modular Denotational Semantics for Effects with Guarded Interaction Trees.” In: *Proc. ACM Program. Lang.* 8.POPL (2024), pp. 332–361. DOI: 10.1145/3632854 (cited on page 30).

[89] Sergei Stepanenko et al. “Context-Dependent Effects in Guarded Interaction Trees.” In: *Programming Languages and Systems*. Ed. by Viktor Vafeiadis. Cham: Springer Nature Switzerland, 2025, pp. 286–313 (cited on page 31).

[90] Dominique Devriese et al. “Modular, Fully-abstract Compilation by Approximate Back-translation.” In: *Log. Methods Comput. Sci.* 13.4 (2017). DOI: 10.23638/LMCS-13(4:2)2017 (cited on pages 33, 56, 57).

[91] Max S. New, William J. Bowman, and Amal Ahmed. “Fully abstract compilation via universal embedding.” In: *21st ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Ed. by Jacques Garrigue, Gabriele Keller, and Eijiro Sumii. ACM, 2016, pp. 103–116. DOI: 10.1145/2951913.2951941 (cited on pages 33, 56, 57, 114).

[92] Koen Jacobs, Dominique Devriese, and Amin Timany. “Purity of an ST monad: full abstraction by semantically typed back-translation.” In: *Proc. ACM Program. Lang.* 6.OOPSLA1 (2022), pp. 1–27. DOI: 10.1145/3527326 (cited on pages 33, 56).

[93] Guido Martínez et al. “Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms.” In: *28th European Symposium on Programming (ESOP)*. Springer, il 2019, pp. 30–59. DOI: 10.1007/978-3-030-17184-1_2 (cited on pages 36, 39, 96).

[94] P. Letouzey. “Coq Extraction, an Overview.” In: *LTA '08*. Vol. 5028. Lecture Notes in Computer Science. Springer-Verlag, 2008 (cited on page 36).

[95] *Dafny Reference Manual*. 2023. URL: https://dafny.org/latest/DafnyRef/DafnyRef (cited on page 36).

[96] Barry Bond et al. “Vale: Verifying High-Performance Cryptographic Assembly Code.” In: *26th USENIX Security Symposium*. Ed. by Engin Kirda and Thomas Ristenpart. USENIX Association, 2017, pp. 917–934 (cited on page 36).

[97] Aymeric Fromherz et al. “A Verified, Efficient Embedding of a Verifiable Assembly Language.” In: *PACMPL* 3.POPL (2019) (cited on page 36).

[98] Matthieu Sozeau et al. “Coq Coq correct! Verification of type checking and erasure for Coq, in Coq.” In: *Proc. ACM Program. Lang.* 4.POPL (2020), 8:1–8:28. DOI: 10.1145/3371076 (cited on page 36).

[99] Zoe Paraskevopoulou, John M. Li, and Andrew W. Appel. “Compositional optimizations for CertiCoq.” In: *Proc. ACM Program. Lang.* 5.ICFP (2021), pp. 1–30. DOI: 10.1145/3473591 (cited on page 36).

[100] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. “Effects as capabilities: effect handlers and lightweight effect polymorphism.” In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 126:1–126:30. DOI: 10.1145/3428194 (cited on page 38).

[101] Éric Tanter and Nicolas Tabareau. “Gradual certified programming in Coq.” In: *11th Symposium on Dynamic Languages (DLS)*. Ed. by Manuel Serrano. ACM, 2015, pp. 26–40. DOI: 10.1145/2816707.2816710 (cited on pages 46, 60).

[102] Karthikeyan Bhargavan et al. “DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code.” In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2021, pp. 523–542. DOI: 10.1109/EuroSP51992.2021.00042 (cited on page 52).

[103] Niklas Grimm et al. “A Monadic Framework for Relational Verification: Applied to Information Security, Program Equivalence, and Optimizations.” In: *The 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. Jan. 2018 (cited on pages 54, 119).

[104] Michael R. Clarkson and Fred B. Schneider. “Hyperproperties.” In: *J. Comput. Secur.* 18.6 (Sept. 2010), pp. 1157–1210 (cited on page 55).

[105] Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. “Linear capabilities for fully abstract compilation of separation-logic-verified code.” In: *J. Funct. Program.* 31 (2021), e6. DOI: 10.1017/S0956796821000022 (cited on pages 60, 85).

[106] Pieter Agten et al. “Secure Compilation to Modern Processors.” In: *25th IEEE Computer Security Foundations Symposium (CSF)*. Ed. by Stephen Chong. IEEE Computer Society, 2012, pp. 171–185. DOI: 10.1109/CSF.2012.12 (cited on page 60).

[107] Marco Patrignani et al. “Secure Compilation to Protected Module Architectures.” In: *ACM Trans. Program. Lang. Syst.* 37.2 (2015), 6:1–6:50. DOI: 10.1145/2699503 (cited on page 60).

[108] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. “StkTokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities.” In: *J. Funct. Program.* 31 (2021), e9. DOI: 10.1017/S095679682100006X (cited on page 60).

[109] Peter-Michael Osera, Vilhelm Sjöberg, and Steve Zdancewic. “Dependent interoperability.” In: *6th Workshop on Programming Languages Meets Program Verification (PLPV)*. Ed. by Koen Claessen and Nikhil Swamy. ACM, 2012, pp. 3–14. DOI: 10.1145/2103776.2103779 (cited on page 60).

[110] Pierre-Évariste Dagand, Nicolas Tabareau, and Éric Tanter. “Foundations of dependent interoperability.” In: *J. Funct. Program.* 28 (2018), e9. DOI: 10.1017/S0956796818000011 (cited on page 60).

[111] Matthew Flatt and PLT. *The Racket Reference*. https://docs.racket-lang.org/reference/. URL: https://docs.racket-lang.org/reference/ (cited on page 60).

[112] Phuc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. “Soft contract verification.” In: *19th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Ed. by Johan Jeuring and Manuel M. T. Chakravarty. ACM, 2014, pp. 139–152. DOI: 10.1145/2628136.2628156 (cited on pages 60, 117).

[113] Feng Chen, Marcelo D’Amorim, and Grigore Roşu. “A Formal Monitoring-Based Framework for Software Development and Analysis.” In: *Formal Methods and Software Engineering*. Springer Berlin Heidelberg, 2004, pp. 357–372. DOI: 10.1007/978-3-540-30482-1_31 (cited on page 60).

[114] Feng Chen and Grigore Roşu. “Java-MOP: A Monitoring Oriented Programming Environment for Java.” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2005, pp. 546–550. DOI: 10.1007/978-3-540-31980-1_36 (cited on page 60).

[115] Dongyun Jin et al. “JavaMOP: Efficient Parametric Runtime Monitoring Framework.” In: *Proceeding of the 34th International Conference on Software Engineering (ICSE’12)*. IEEE, June 2012, pp. 1427–1430. DOI: http://dx.doi.org/10.1109/ICSE.2012.6227231 (cited on page 60).

[116] Éric Tanter. “Expressive scoping of dynamically-deployed aspects.” In: *7th International Conference on Aspect-Oriented Software Development (AOSD)*. Ed. by Theo D’Hondt. ACM, 2008, pp. 168–179. DOI: 10.1145/1353482.1353503 (cited on page 61).

[117] Rémi Douence, Pascal Fradet, and Mario Südholt. “Trace-Based Aspects.” In: *Aspect-Oriented Software Development*. Ed. by Robert E. Filman et al. Boston: Addison-Wesley, 2005, pp. 201–217 (cited on page 61).

[118] Hidehiko Masuhara, Gregor Kiczales, and Christopher Dutchyn. “A Compilation and Optimization Model for Aspect-Oriented Programs.” In: *12th International Conference on Compiler Construction (CC)*. Ed. by Görel Hedin. Vol. 2622. Lecture Notes in Computer Science. Springer, 2003, pp. 46–60. DOI: 10.1007/3-540-36579-6_4 (cited on page 61).

[119] Pavel Avgustinov, Julian Tibble, and Oege de Moor. “Making trace monitors feasible.” In: *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Ed. by Richard P. Gabriel et al. ACM, 2007, pp. 589–608. DOI: 10.1145/1297027.1297070 (cited on page 61).

[120] Eric Bodden, Laurie J. Hendren, and Ondrej Lhoták. “A Staged Static Program Analysis to Improve the Performance of Runtime Monitoring.” In: *21st European Conference on Object-oriented Programming (ECOOP)*. Ed. by Erik Ernst. Vol. 4609. Lecture Notes in Computer Science. Springer, 2007, pp. 525–549. DOI: 10.1007/978-3-540-73589-2_25 (cited on page 61).

[121] Johannes Bader, Jonathan Aldrich, and Éric Tanter. “Gradual Program Verification.” In: *19th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Ed. by Isil Dillig and Jens Palsberg. Vol. 10747. Lecture Notes in Computer Science. Springer, 2018, pp. 25–46. DOI: 10.1007/978-3-319-73721-8_2 (cited on pages 61, 86).

[122] Jenna Wise et al. “Gradual verification of recursive heap data structures.” In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 228:1–228:28. DOI: 10.1145/3428296 (cited on page 61).

[123] Orna Kupferman and Moshe Y. Vardi. “Robust Satisfaction.” In: *CONCUR ’99: 10th International Conference on Concurrency Theory*. 1999, pp. 383–398. DOI: 10.1007/3-540-48320-9_27 (cited on page 61).

[124] Andrew D. Gordon and Alan Jeffrey. “Types and Effects for Asymmetric Cryptographic Protocols.” In: *Journal of Computer Security* 12.3-4 (2004), pp. 435–483 (cited on page 61).

[125] Maximilian Algehed and Jean-Philippe Bernardy. “Simple noninterference from parametricity.” In: *Proc. ACM Program. Lang.* 3.ICFP (2019), 89:1–89:22. DOI: 10.1145/3341693 (cited on page 61).

[126] Maximilian Algehed, Jean-Philippe Bernardy, and Catalin Hritcu. “Dynamic IFC Theorems for Free!” In: *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*. IEEE, 2021, pp. 1–14. DOI: 10.1109/CSF51468.2021.00005 (cited on page 61).

[127] K. Rustan M. Leino. “Dafny: An Automatic Program Verifier for Functional Correctness.” In: *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. LPAR’10. Dakar, Senegal: Springer-Verlag, 2010, pp. 348–370 (cited on pages 65, 84).

[128] Ralf Jung et al. “Higher-order ghost state.” In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. Ed. by Jacques Garrigue, Gabriele Keller, and Eijiro Sumii. ACM, 2016, pp. 256–269. DOI: 10.1145/2951913.2951943 (cited on page 71).

[129] Andrzej S. Murawski and Nikos Tzevelekos. “Algorithmic Games for Full Ground References.” In: *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part II*. Ed. by Artur Czumaj et al. Vol. 7392. Lecture Notes in Computer Science. Springer, 2012, pp. 312–324. DOI: 10.1007/978-3-642-31585-5_30 (cited on page 72).

[130] Andrzej S. Murawski and Nikos Tzevelekos. “Algorithmic games for full ground references.” In: *Formal Methods Syst. Des.* 52.3 (2018), pp. 277–314. DOI: 10.1007/S10703-017-0292-9 (cited on page 72).

[131] Ohad Kammar et al. “A monad for full ground reference cells.” In: *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, 2017, pp. 1–12. DOI: 10.1109/LICS.2017.8005109 (cited on page 72).

[132] Stephen Dolan et al. “Concurrent System Programming with Effect Handlers.” In: *Trends in Functional Programming - 18th International Symposium, TFP 2017, Canterbury, UK, June 19-21, 2017, Revised Selected Papers*. Ed. by Meng Wang and Scott Owens. Vol. 10788. Lecture Notes in Computer Science. Springer, 2017, pp. 98–117. DOI: 10.1007/978-3-319-89719-6_6 (cited on page 82).

[133] Ralf Jung et al. “Iris from the ground up: A modular foundation for higher-order concurrent separation logic.” In: *J. Funct. Program.* 28 (2018), e20. DOI: 10.1017/S0956796818000151 (cited on page 84).

[134] Aïna Linn Georges et al. “Efficient and provable local capability revocation using uninitialized capabilities.” In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–30. DOI: 10.1145/3434287 (cited on page 86).

[135] Robert N. M. Watson et al. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)*. Tech. rep. UCAM-CL-TR-951. University of Cambridge, Computer Laboratory, Oct. 2020. DOI: 10.48456/tr-951 (cited on page 86).

[136] Max S. New, William J. Bowman, and Amal Ahmed. “Fully abstract compilation via universal embedding.” In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. Ed. by Jacques Garrigue, Gabriele Keller, and Eijiro Sumii. ACM, 2016, pp. 103–116. DOI: 10.1145/2951913.2951941 (cited on page 86).

[137] Jenna DiVincenzo et al. “Gradual C0: Symbolic Execution for Gradual Verification.” In: *ACM Trans. Program. Lang. Syst.* 46.4 (Jan. 2025). DOI: 10.1145/3704808 (cited on page 86).

[138] Daniel Patterson et al. “FunTAL: reasonably mixing a functional language with assembly.” In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. Ed. by Albert Cohen and Martin T. Vechev. ACM, 2017, pp. 495–509. DOI: 10.1145/3062341.3062347 (cited on page 86).

[139] Daniel Patterson and Amal Ahmed. “Linking Types for Multi-Language Software: Have Your Cake and Eat It Too.” In: *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*. Ed. by Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi. Vol. 71. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 12:1–12:15. DOI: 10.4230/LIPICS.SNAPL.2017.12 (cited on page 86).

[140] Daniel Patterson, Andrew Wagner, and Amal Ahmed. “Semantic Encapsulation using Linking Types.” In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Type-Driven Development, TyDe 2023, Seattle, WA, USA, 4 September 2023*. Ed. by Youyou Cong and Pierre-Évariste Dagand. ACM, 2023, pp. 14–28. DOI: 10.1145/3609027.3609405 (cited on page 86).

[141] Georg Neis et al. “Pilsner: a compositionally verified compiler for a higher-order imperative language.” In: *20th ACM SIGPLAN International Conference on Functional Programming*. 2015, pp. 166–178. DOI: 10.1145/2784731.2784764 (cited on pages 89, 113).

[142] Amal Ahmed. “Verified Compilers for a Multi-Language World.” In: *1st Summit on Advances in Programming Languages*. Vol. 32. LIPIcs. Schloss Dagstuhl, 2015, pp. 15–31. DOI: 10.4230/LIPIcs.SNAPL.2015.15 (cited on pages 89, 113).

[143] James Chapman et al. *Peregrine Project: a unified middle-end for code generation from proof assistants*. https://peregrine-project.github.io. URL: https://peregrine-project.github.io (cited on pages 90, 112, 119).

[144] Nikhil Swamy. *Agentic Proof-Oriented Programming*. RiSE MSR Blog. Feb. 2026. URL: https://risemsr.github.io/blog/2026-02-04-nik-agentic-pop/ (cited on page 90).

[145] Cezar-Constantin Andrici et al. *Misquoted No More: Securely Extracting F* Programs with IO - Artifact*. June 2026. DOI: 10.5281/zenodo.20534723 (cited on page 90).

[146] Paul Blain Levy, John Power, and Hayo Thielecke. “Modelling environments in call-by-value programming languages.” In: *Inf. Comput.* 185.2 (2003), pp. 182–210. DOI: 10.1016/S0890-5401(03)00088-9 (cited on pages 93, 95).

[147] Amal Jamil Ahmed. “Semantics of Types for Mutable State.” PhD thesis. Princeton University, 2004 (cited on pages 99, 101, 113).

[148] Stephen Dolan. “Malfunctional Programming.” In: *ML Family Workshop 2016*. 2016 (cited on pages 112, 120).

[149] Ramana Kumar et al. “CakeML: a verified implementation of ML.” In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. ACM, 2014, pp. 179–192. DOI: 10.1145/2535838.2535841 (cited on pages 112, 114).

[150] Conor McBride. “Outrageous but meaningful coincidences: dependent type-safe syntax and evaluation.” In: *Proceedings of the ACM SIGPLAN Workshop on Generic Programming, WGP 2010, Baltimore, MD, USA, September 27-29, 2010*. Ed. by Bruno C. d. S. Oliveira and Marcin Zalewski. ACM, 2010, pp. 1–12. DOI: 10.1145/1863495.1863497 (cited on page 113).

[151] Kazutaka Matsuda et al. “Embedding by Unembedding.” In: *Proc. ACM Program. Lang.* 7.ICFP (2023), pp. 1–47. DOI: 10.1145/3607830 (cited on page 113).

[152] Georges Gonthier et al. “How to make ad hoc proof automation less ad hoc.” In: *Journal of Functional Programming* 23.4 (2013), pp. 357–401 (cited on page 113).

[153] Dominique Devriese, Marco Patrignani, and Frank Piessens. “Fully-Abstract Compilation by Approximate Back-Translation.” In: *43nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2016 (cited on page 114).

[154] Andreas Abel. “Normalization by evaluation: Dependent types and impredicativity.” In: *Habilitation. Ludwig-Maximilians-Universität München* (2013) (cited on page 115).

[155] Arthur Adjedj et al. “Martin-löf à la coq.” In: *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2024, pp. 230–245 (cited on page 115).

[156] Junyoung Jang et al. “McTT: A Verified Kernel for a Proof Assistant.” In: *Proceedings of the ACM on Programming Languages* 9.ICFP (2025), pp. 190–221 (cited on page 115).

[157] Yiyun Liu, Jonathan Chan, and Stephanie Weirich. “Functional Pearl: Short and Mechanized Logical Relation for Dependent Type Theories.” In: (2024) (cited on page 115).

[158] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. “Decidability of conversion for type theory in type theory.” In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), pp. 1–29 (cited on page 115).

[159] Josselin Poiret, Kenji Maillard, and Nicolas Tabareau. “Divide and Check: Logical Relations, No Algorithms Attached.” In: (2026) (cited on page 115).

[160] Max Vistrup, Michael Sammler, and Ralf Jung. “Program Logics à la Carte.” In: *Proc. ACM Program. Lang.* 9.POPL (Jan. 2025). DOI: 10.1145/3704847 (cited on pages 117, 119).

[161] Phuc C. Nguyen et al. “Soft contract verification for higher-order stateful programs.” In: *Proc. ACM Program. Lang.* 2.POPL (2018), 51:1–51:30. DOI: 10.1145/3158139 (cited on page 117).

[162] Remy Seassau et al. “Formal Semantics and Program Logics for a Fragment of OCaml.” In: *Proc. ACM Program. Lang.* 9.ICFP (Aug. 2025). DOI: 10.1145/3747509 (cited on page 119).

[163] NICOLAS CHAPPE et al. “Choice trees: Representing and reasoning about nondeterministic, recursive, and impure programs in Rocq.” In: *Journal of Functional Programming* 35 (2025), e21. DOI: 10.1017/S0956796825100105 (cited on page 119).

[164] Paulo Emílio de Vilhena and François Pottier. “A separation logic for effect handlers.” In: *Proc. ACM Program. Lang.* 5.POPL (2021), 33:1–33:28. DOI: 10.1145/3434314 (cited on page 119).