Formally Verified Native Code Generation in an Effectful JIT

Turning the CompCert Backend into a Formally Verified JIT Compiler

AURÈLE BARRIÈRE, Univ Rennes, Inria, CNRS, IRISA, France
SANDRINE BLAZY, Univ Rennes, Inria, CNRS, IRISA, France
DAVID PICHARDIE, Meta, France

Modern Just-in-Time compilers (or JITs) typically interleave several mechanisms to execute a program. For faster startup times and to observe the initial behavior of an execution, interpretation can be initially used. But after a while, JITs dynamically produce native code for parts of the program they execute often. Although some time is spent compiling dynamically, this mechanism makes for much faster times for the remaining of the program execution. Such compilers are complex pieces of software with various components, and greatly rely on a precise interplay between the different languages being executed, including on-stack-replacement. Traditional static compilers like CompCert have been mechanized in proof assistants, but JITs have been scarcely formalized so far, partly due to their impure nature and their numerous components. This work presents a model JIT with dynamic generation of native code, implemented and formally verified in Coq. Although some parts of a JIT cannot be written in Coq, we propose a proof methodology to delimit, specify and reason on the impure effects of a JIT. We argue that the daunting task of formally verifying a complete JIT should draw on existing proofs of native code generation. To this end, our work successfully reuses CompCert and its correctness proofs during dynamic compilation. Finally, our prototype can be extracted and executed.

CCS Concepts: • **Software and its engineering** → **Just-in-time compilers**; • **Theory of computation** → **Program verification**.

Additional Key Words and Phrases: verified compilation, just-in-time compilation, CompCert compiler

ACM Reference Format:
Aurèle Barrière, Sandrine Blazy, and David Pichardie. 2023. Formally Verified Native Code Generation in an Effectful JIT: Turning the CompCert Backend into a Formally Verified JIT Compiler. *Proc. ACM Program. Lang.* 7, POPL, Article 9 (January 2023), 29 pages. https://doi.org/10.1145/3571202

1 INTRODUCTION

Formally verified compilers are compilers that come with a machine-checked proof of correctness; it establishes that the compiler does not introduce bugs during compilation. Such compilers are programmed using a proof assistant, but can also be run independently. For instance, a compiler written in the purely functional Gallina programming language of Coq can be automatically extracted to an equivalent OCaml program. If one trusts the extraction mechanism, the properties about the Coq code transfer to the extracted OCaml code. While standard (namely ahead-of-time) compilation has been the target of many verification works, such as CompCert [Kästner et al. 2018; Leroy 2006, 2009a; Leroy et al. 2016], Vellvm [Zhao et al. 2012, 2013], Crellvm [Kang et al. 2018], and CakeML [Kumar et al. 2014; Lööw et al. 2019; Owens et al. 2017; Tan et al. 2016] just-in-time compilation poses new challenges that have yet to be verified in a proof assistant.

Authors' addresses: Aurèle Barrière, Univ Rennes, Inria, CNRS, IRISA, Rennes, France, aurele.barriere@irisa.fr; Sandrine Blazy, Univ Rennes, Inria, CNRS, IRISA, Rennes, France, sandrine.blazy@irisa.fr; David Pichardie, Meta, Paris, France.

2475-1421/2023/1-ART9
https://doi.org/10.1145/3571202

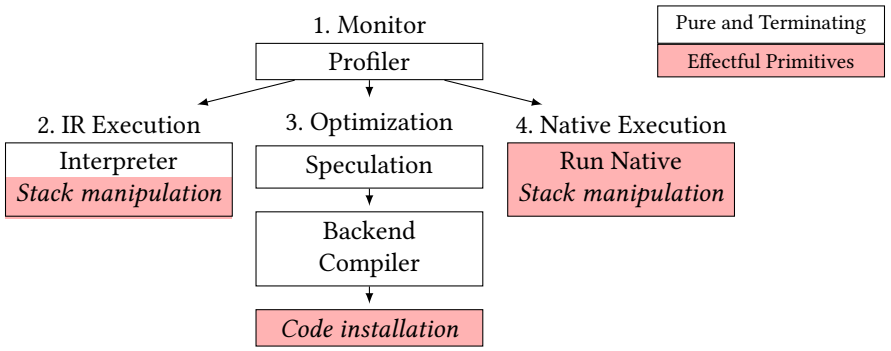

Fig. 1. Key components of a JIT with native code generation.

Instead of translating a whole program once for all, then executing its output, just-in-time compilers (later referred as JIT) present an alternative approach: they start with interpretation, but parts of the program are translated at the last possible moment during execution. This is often useful to execute dynamic languages such as Python and JavaScript [PyPy 2022; V8 2022], but also other languages such as Java [HotSpot 2022], using information known only at run-time.

In essence, a JIT interleaves the execution of a program with its optimization. Optimizations are either standard (static) optimizations or JIT-specific dynamic optimizations. In practice, the dynamic optimization part of modern JITs often consists in dynamically generating native code for efficiency, for some parts of the program (typically functions) likely to be executed a lot. When executing a program with a JIT, the execution of high-level non-optimized code is then interleaved with the execution of native code that has been dynamically generated. With JIT compilation, the program evolves during its execution. The program being executed at some point in time depends on what dynamic optimizations have been made before and cannot be known before execution.

A JIT is much more complex than a standard compiler; it needs to orchestrate the interplay of several key components illustrated in Figure 1, including a standard compiler (the box called backend compiler). To simplify the figure, we consider that programs are represented at an intermediate level (called IR) where optimizations and interpretation are performed, even if some JITs rather interpret bytecode. First, a monitor (component 1.) is in charge of gathering information and compilation hints about the program execution using a profiler, and regularly suggests functions to be optimized. This monitoring step is a simple computation that calls the relevant JIT component among three possible ones: either interpretation of the program at the IR level if there is nothing to optimize (2.) or at the native level if some function has been optimized (4.) or program optimization (3.).

A typical optimization of modern JITs is dynamic speculation. Given some likely assumption suggested by the monitor, it allows the JIT to specialize some functions of the program at the only cost of dynamically checking that the assumption holds. For instance, some JITs can speculate on values or types of variables in a function, and create a version of that function specialized to the assumed behavior. This is especially useful for dynamic languages. Other optimizations such as those found in a standard backend-compiler may be triggered as well. They are followed by a code installation pass that inserts the machine bytes produced by the compiler in an executable code section of memory. When calling such a compiled function, the JIT can then jump to where it was installed, which can lead to significant speedups compared to IR interpretation. Moreover, JITs with speculative optimizations must include a deoptimization mechanism to switch from native code execution to the interpretation of the original function if some speculative assumption does

```
Function F (): (IR)
  x ← 1
  y ← Call G(x)
  Return y
```

```
Function G (a): (IR)
  b ← a+1
  Return b
```

```
$Function G: (x86, simplified)
  call  _Pop
  leal  1(%eax), %edi
  call  _Push
  ret
```

Fig. 2. Two functions of a program, where G has been compiled to x86 using our custom calling conventions.

not hold. This may happen in the middle of the execution of a function and requires performing *on-stack replacement*: replacing the current native stackframe with a stackframe of the interpreter.

Our goal is to formally verify such a JIT in Coq, but we face several JIT-specific challenges. Such challenges do not occur in existing verified compilers such as CompCert, that can be written as a pure and terminating Gallina function. First, some JIT components simply cannot be written directly in Coq, for instance the installation of bytes in memory or calling native functions. Second, a JIT interleaves the execution of two languages, the IR and the native code. Both share some data structures (*i.e.*, a stack and a heap). Stack manipulation is made particularly difficult with on-stack replacement as one needs to be able to synthesize an interpreter stackframe. There is little hope of defining mutable, global Coq data-structures that can also be accessed and modified by native code.

Our solution is to define a small set of primitives for everything that cannot be written in Coq in a JIT (*i.e.*, the pink boxes of Figure 1). This includes the shared data-structures and their operations, but also installing and running machine code. These simple primitives are not implemented in Coq, but can be specified using Coq functions on an abstract model of the JIT memory. We use a variation of free monads, a pure functional encoding of effectful programs [Swierstra 2008], to represent in Coq a program such as the JIT with unimplemented primitives. Primitives can be called from both native code and components defined in Coq, allowing the interoperability that JITs intrinsically need. We define custom calling conventions using these primitives.

For instance, consider a program being executed by our JIT containing a function F calling a function G, as seen on Figure 2. At some point during the execution of the program, the JIT may have already compiled G to x86, as seen on the right of this Figure. When interpreting F, our interpreter sees the call to another function and uses a primitive to save its current environment to the JIT execution stack. The intepreter then returns to the monitor, asking to call function G. The monitor then uses primitives to push the call argument to the stack, then to load and execute the x86 code corresponding to G. This x86 code has been generated such that it starts by using a primitive (_Pop) to get the call argument. After doing the computation, it then uses another primitive (_Push) to push the return value of function G to the JIT execution stack. Finally, it returns to the monitor. To execute the remaining of function F, the monitor uses primitives to get the result and the top interpreter stackframe, then calls the interpreter again.

Primitive specifications allow us to reason on the behavior of an effectful JIT with multiple languages and shared data-structures, as if the JIT was entirely programmed in Coq. Specifically we prove that, using an abstract model of the JIT memory and an abstract specification of the primitives, executing a program with the JIT outputs the same behavior than specified by the program semantics. The result, which is the subject of this paper, is a Coq JIT called FM-JIT (Free-Monadic JIT) that comes with a correctness proof, but can also be extracted and completed with primitive implementations to be executable. If one trusts this implementation to match its specification, the Coq correctness property extends to the executable JIT. To the best of our knowledge, FM-JIT is the first formally verified and mechanized JIT including 1) a standard optimizing compiler backend to dynamically produce native code, and 2) the interoperability of native code and interpretation, with support for on-stack-replacement. This represents a substantial step toward the verification of realistic modern

JIT compilers, some of the most complex execution engines. FM-JIT uses the CompCert backend to dynamically generate native code corresponding to a function.

Unless noted otherwise, all results presented in this paper have been mechanically verified using the Coq proof assistant [Inria 2022]. The complete development, including mechanized proofs, is available as an artifact [Barrière et al. 2022]. Specifically, we claim the following novel contributions:

- **JIT design:** We introduce FM-JIT, a Coq JIT design with dynamic native code generation and execution. Its dynamic compilation provides support for speculative instructions, an advanced feature of modern JITs. This design clearly specifies each JIT component and their interplays, which are seldom described in the literature.
- **New proof techniques:** Using a variation of free monads, we develop a proof methodology for formally verifiable and executable programs with impure and non-terminating components, with a minimal trusted code base. It relies on refinement to switch between different primitive specifications for more modular proofs. We apply this methodology to FM-JIT.
- **Proof reuse:** As JITs reuse static compilation techniques to generate native code, we argue that formally verified JITs should reuse formally verified static compilers proofs to alleviate the proof burden of such complex pieces of software. We demonstrate that it is possible to reuse both CompCert proof techniques (namely backward simulations) and CompCert correctness proofs in FM-JIT, without any modification of existing proofs.
- **JIT correctness:** We prove correct that any behavior of FM-JIT according to the specifications of all its impure components, is a behavior of its input program. The proof composes the complex correctness proofs of all its components, including its backend compiler.
- **Runnable JIT:** Combining Coq extraction to OCaml and a C implementation of the primitives, we obtain a runnable JIT that dynamically generates and executes actual native code. As expected, we observe speedups compared to interpretation alone.

Our work has limitations that we state here. JITs with speculation typically insert their assumptions and specialize their functions during execution. The insertion of dynamic speculation is out of scope of this work, and FM-JIT does not include the full speculation component of Figure 1. This orthogonal problem has been investigated in [Barrière et al. 2021]. However, we show that we can provably compile speculative instructions by allowing our input programs to be already specialized. The simple primitive implementations used for the runnable JIT are also not proved correct, but can be audited manually and compared to their Coq specification (see Section 6.4). Just like CompCert, the FM-JIT correctness proof stops at the assembly level when compiling functions. We then trust an assembler to produce equivalent machine code.

This paper is organized as follows. First, Section 2 recalls the required background on the CompCert compiler. Then, Section 3 gives an overview of FM-JIT. Section 4 introduces our Free Monad methodology to represent and reason in Coq about an effectful program such as FM-JIT. Section 5 details our proof techniques and their application to the correctness of FM-JIT. Section 6 presents some perspectives about FM-JIT as an executable JIT. Related work is discussed in Section 7, followed by conclusions.

2 BACKGROUND ON THE COMPCERT VERIFIED C COMPILER

CompCert [Leroy 2009a] is the first commercially available optimizing compiler that is formally verified. It consists of a frontend from C to RTL (3-address code), and a backend from RTL to native code. CompCert targets several architectures but we focus on x86-64 assembly. This section introduces the ingredients of the correctness proof of its passes, further detailed in [Leroy 2009b].

In CompCert, a small-step semantics defines an execution relation between semantic states and associates to each program the set of its possible behaviors (termination, divergence and

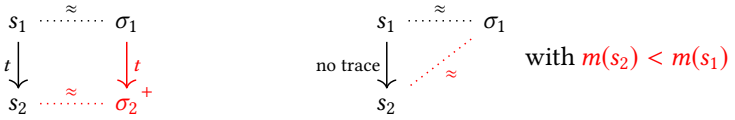

Fig. 3. Forward-simulation diagram with measure. Black lines are hypotheses, red lines are conclusions. On the left of each diagram are the source program and its current state s_1; the target program and its current state σ_1 are on the right. Vertical lines represent steps and horizontal lines the matching relation $\approx$.

going-wrong), including its trace of observable I/O events (*e.g.*, calls to external library functions). Diverging executions observe finite or infinite traces. A generic notion of transition semantics is defined; it consists of a type of program states (where some are initial and others final for a terminating execution) and a step relation $\longrightarrow$ over these states. CompCert defines several transition relations (*e.g.*, star and plus transitive closures) from the generic $\longrightarrow$. Given a program `P` written in a language defined by its semantics `sem`, we write `program_behaves (sem p) beh` to mean that the execution of `P` according to `sem` observes a behavior `beh`.

The correctness theorem is stated as follows: if CompCert produces code `C` from source `S`, then every observable behavior `bc` of `C` (`program_behaves (sem C) bc`) is a possible behavior `bs` of `S` (`program_behaves (sem S) bs`): either `bc` is `bs`, or `bc` improves `bs` (`behavior_improves bs bc`), meaning that the finite trace of events observed before `C` went wrong is a prefix of the trace observed during the execution of `S`. As a compiler, CompCert is decomposed into several passes, and the correctness theorem results from the correctness of each compiler pass, and so does the correctness of its backend. The standard technique to prove the correctness of a pass is to prove a backward simulation (*i.e.*, every behavior of a transformed program `C` is a behavior of the source `S`). It is often hard to prove a backward simulation; for passes that preserve nondeterminism, it is easier to reason on forward simulations (*i.e.*, every behavior of `S` is also a behavior of `C`). A backward simulation from `C` to `S` can be constructed from a forward simulation from `S` to `C` when `C` is deterministic.

Last, the most general forward-simulation diagram is defined as follows. The correctness proof of a compiler pass from language L_1 to language L_2 relies on a forward simulation diagram shown in Figure 3 and expressed in the following theorem. Given a program P_1 and its transformed program P_2, each transition step in P_1 with trace t must correspond to transitions in P_2 with the same trace t and preserve as an invariant a relation $\approx$ between states of P_1 and P_2. In order to handle diverging execution steps and rule out the infinite stuttering problem (that may happen when infinitely many consecutive steps in P_1 are simulated by no step at all in P_2), the theorem uses a measure over the states of language L_1 that strictly decreases in cases where stuttering could occur. It is generically noted $m(\cdot)$ and is specific to each compiler pass. In CompCert's parlance, this diagram is denoted by `forward_simulation (sem1 P1) (sem2 P2)`, where `semi` defines the semantics of L_i.

3 OVERVIEW OF FM-JIT AND ITS CORRECTNESS THEOREM

This section presents the salient features of FM-JIT: its architecture, source language, impure primitives, and its main correctness theorem. While minimal, it precisely captures the essence of the dynamic generation of native code and its execution in a JIT.

3.1 The Architecture of FM-JIT

A JIT resembles an interaction loop in that it executes or optimizes the code until the program finishes. Figure 4 shows the state machine that describes how the JIT monitor alternates between its components, from IR execution, to optimization and native execution. FM-JIT is defined with a function representing the transitions of this state machine and calling the relevant JIT component.

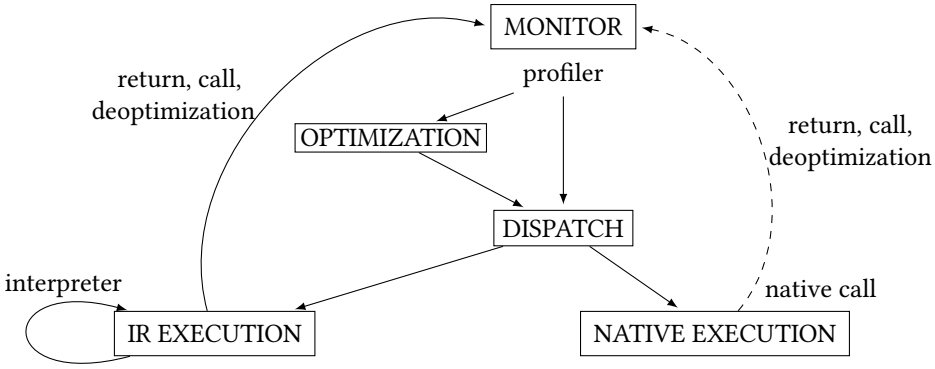

Fig. 4. A JIT architecture as a state machine. Dashed line represents multiple steps (see Section 4.6).

Each transition can be represented as a terminating Gallina function (possibly using primitives). However, the execution of native code may be non-terminating if the JIT compiled a diverging function. This is why one transition of the JIT is represented in a dashed line in Figure 4, meaning that it corresponds to a sequence of elementary transitions. We first ignore that non-atomic transition, then explain how it is defined and specified in Section 4.6.

The profiler corresponds to a collection of heuristics that inspect the current execution and suggest optimizations. The profiling heuristics themselves are out of scope of our verification work, as they are often highly empirical. Our formalization takes as input any of these heuristics. The benefits of clearly separating these from the rest of FM-JIT is that our correctness results do not depend on the implementation of such heuristics, which should only have an impact on performance, but not on correctness.

An optimization is made of three steps. First, the IR function to compile is translated into RTL (see Section 5.1). Second, the CompCert backend is called on the translated RTL to produce some x86 code. Third, this x86 code is installed in an executable portion of the memory. This last part cannot be written directly in Gallina, and is part of the primitives we specify (as listed on Section 3.3).

Whenever the profiler did not suggest any optimization, the JIT resumes the program execution. Depending on whether or not the current function has been compiled yet, execution is dispatched to either the interpreter or some dynamically generated native code. This execution component stays active until the execution reaches a *synchronization point* (either a function call, or a function return, or a deoptimization trigger), where it returns to the monitor. The monitor is then tasked with managing the stack. It may push arguments to the stack if the synchronization point is a function call, or even synthesize a new interpreter stackframe if the synchronization point is a deoptimization trigger. After a deoptimization, execution will always be dispatched to IR execution, since the new current stackframe is an interpreter one.

3.2 The Low-Level Intermediate Representation CoreIR

The source language that FM-JIT executes is CoreIR. Its syntax is depicted Figure 5. There is an infinite number of registers r, and each can contain a value v, namely a 64-bit integer. A function has an identifier f, an entry point l and a list of arguments r^*. Its code is represented by its control-flow graph, namely a mapping from labels l representing program points to instructions i. Functions can have up to two versions, an optional one where speculation has been inserted, and an original one to deoptimize to if needed.

Expressions:

$$e ::= r + r \mid r - r \mid r * r \mid r = r \mid r < r \mid r \,\%\, r \mid - r \mid v \mid r = 0 \mid r + v \mid r * v$$

Programs:

$$V ::= l \mapsto i \quad \text{Code}$$
$$F ::= \{r^*, l, V, \mathit{option}\ V\} \quad \text{Function}$$
$$P ::= f \mapsto F \quad \text{Program}$$

Instructions:

$$i ::= \mathbf{Nop}\ l \mid r \leftarrow e\ l \quad \text{no-operation and operations}$$
$$\mid \mathbf{Cond}\ r\ l_t\ l_f \mid \mathbf{Print}\ r\ l \quad \text{branching, output value}$$
$$\mid r \leftarrow \mathbf{Call}\ f\ r^*\ l \mid \mathbf{Return}\ r \quad \text{call and return}$$
$$\mid r \leftarrow \mathbf{MemGet}\ r_{ad}\ l \mid r_{ad} \leftarrow \mathbf{MemSet}\ r\ l \quad \text{memory load and store}$$
$$\mid \mathbf{Assume}\ r\ f.l\ [(r \leftarrow e)^*]\ l \quad \text{speculation}$$

Fig. 5. Syntax of CoreIR.

CoreIR was introduced in [Barrière et al. 2021] as an extension of RTL, with some instructions for speculation, inspired by [Flückiger et al. 2018]. The **Assume** instruction represents a speculation. To execute it, its condition (what is being speculated on) is dynamically checked. If it holds, execution continues to the next label. Otherwise, deoptimization happens, and the JIT monitor must build an interpreter stackframe following the deoptimization metadata contained in that instruction. For instance, `Assume x F3.12 [a←5, b←y] 14` will check if `x` is true (any value other than 0). If it is, execution jumps to `14`. Else, execution deoptimizes into the original version of function `F3`, at label `12`. To reconstruct the environment of that function, we put 5 in register `a` and evaluate `y` in the current environment for the value of `b`. This work shows that such inserted speculations can be correctly compiled down to native code by the backend of FM-JIT. A CoreIR program also gets access to a part of the memory, the heap (a fixed-size array of 64-bit integer values), using **MemGet** and **MemSet** instructions.

CoreIR is simplified compared to [Barrière et al. 2021]. First, the **Anchor** instruction, useful to insert speculations, has been removed since speculation insertion is out-of-scope of FM-JIT. Second, the **Assume** instruction is simplified to only allow the synthesis of a single stackframe at once when deoptimizing. Synthesizing multiple stackframes has to be done when inlining functions, but FM-JIT does not perform inlining and instead compiles functions one at a time. Finally, the operations have been modified to better resemble RTL. The semantics of CoreIR is close to the RTL semantics and the semantics of the **Assume** instruction is defined in [Barrière et al. 2021].

3.3 The Minimal Set of Impure Primitives of FM-JIT

This section describes the parts of FM-JIT that cannot be written in Coq. Since their implementation has to be trusted, we keep the smallest possible interface of impure primitives that a JIT with native code generation requires. FM-JIT uses the following impure primitives:

- `HeapGet x` to get the heap value at address `x`.
- `HeapSet x y` to set `x` in the heap at address `y`.
- `Push x`, to push a value `x` on the stack.
- `Pop`, to pop a value from the top of the stack.
- `Push_IRSF sf` to add an interpreter stackframe `sf` to the stack.
- `OpenSF` to get the top stackframe of the stack.
- `Install_Code asm` to install some native code `asm` generated by the optimizer.
- `Load_Code fun_id` to load some installed native code for function `fun_id`.

- `Check_installed fun_id` to check if some function `fun_id` has been compiled.
- `Print x` to print some value `x`.

In static compilation, one simply generates code that modifies the memory, while in a JIT the memory is a data-structure directly modified during execution. `HeapSet` and `HeapGet` are used to modify and access the heap. These primitives are called by the IR interpreter when interpreting **MemSet** and **MemGet** instructions, but also by the native code of a function that originally contained these instructions. Section 4.2 details how primitives are called from Coq components and an example of our generated native code is shown later on Figure 17c.

The execution stack contains both interpreter stackframes and stackframes for the native code. Stackframes for the native code consist in 64-bit integers pushed on top of the stack. The native code we generate uses custom calling convention, involving `Pop` and `Push` to add or get integers from the stack, for instance to get function arguments or push return values. Examples of these custom conventions are shown in Section 5.1. When deoptimizing, the native code also pushes its deoptimization metadata to the stack using `Push`. The monitor uses both primitives when calling a native function (pushing arguments), or when returning from a native call (popping the return value), or after native-code deoptimization (popping the deoptimization metadata before creating the corresponding stackframe). But the stack also contains interpreter stackframes, that can be pushed by the interpreter with `Push_IRSF` when an interpreted function calls another. Technically, we could design our interpreter such that it uses stackframes made of 64-bit integers as well, and then use the `Push` primitive for both native and interpreter stackframes. Such an interpreter would be more difficult to write and prove correct than our current version using these Coq records, which closely follows the CoreIR semantics. Having a dedicated primitive also shows that our design is not tied to a particular interpreter implementation. After a function return, the monitor uses `OpenSF` to get the top stackframe, and dispatches execution accordingly.

Finally, FM-JIT must also install the native code it generated. The optimizer uses `Install_Code` to allocate some space in the memory, write machine bytes and make that memory executable. This also includes calling an assembler to produce machine code from the assembly code generated by the JIT. The addresses of these allocations are stored, and `Load_Code` can then be used when running native code to return the address of the machine code corresponding to a function identifier. `Check_Installed` is used by the monitor to decide if it should call the interpreter or the native code for function calls. Note that tracking which function has been installed could be done purely in Coq, removing this last primitive from the list. We include this primitive anyway to define a comprehensive interface of everything a JIT needs to do with its executable memory. Note that the three parts of the JIT memory (execution stack, heap and executable codes) are disjoint.

As JITs differ from static compilers in their need to do effectful impure computations, this list already sheds some light on the way a formally verified JIT should be designed. The impure effects of FM-JIT are restrained to that small list of impure primitives. Executing native code is yet another impure computation done by the JIT that is handled differently and explained in Section 4.6. All other things done by FM-JIT can be written directly in the pure programming language of Coq.

3.4 The Correctness Theorem of FM-JIT

CompCert simulations state that any behavior of a compiled program matches a behavior of its source program. While the program of a JIT dynamically evolves during execution, one can still prove a similar correctness theorem if we compare the semantics of the input program to some small-step semantics (called `jit_sem`) describing the behavior of FM-JIT executing a program. In short, every step consists in a transition of the state machine of Figure 4. Many of these transitions are pure Coq functions that can be used directly to define small steps. Other use some of the impure

primitives listed in Section 3.3. These cannot be written as Coq functions, but can be specified with Coq monadic functions. `jit_sem` is defined in Section 4.3, where the specification of the primitives is used to define the remaining transitions. Last, we reuse the CompCert simulation framework (see Section 2) to prove FM-JIT correct by proving the following theorems (see details in Section 5.5), where `prim_spec` is a specification of the impure primitives (see Section 4.3), and `CoreIR_sem` is the small-step semantics of the CoreIR language.

```
Theorem jit_correctness_simulation:
  ∀ p, backward_simulation (CoreIR_sem p) (jit_sem p prim_spec).
Theorem jit_correctness:
  ∀ p beh, program_behaves (jit_sem p prim_spec) beh →
    ∃ beh', program_behaves (CoreIR_sem p) beh' ∧ behavior_improves beh' beh.
```

The above theorem states that for any source program `p`, if the JIT behaves in some behavior `beh`, then `beh` improves a behavior of the original program semantics. Behavior improvement means that if `p` goes wrong, FM-JIT is allowed to avoid going wrong and can produce any behavior `beh` after that point. A corollary states that if `p` does not go wrong, then any JIT behavior is exactly a behavior of `p`. This theorem strongly resembles the one from CompCert, only replacing the semantics of the compiled program with the semantics `jit_sem` of FM-JIT.

Note that FM-JIT is not allowed to go wrong if `p` does not. In particular, when the compilation of a function fails (because some analysis did not converge), FM-JIT simply cancels the optimization and keeps interpreting safely. Similarly, some primitives may fail during execution of FM-JIT, but we prove that they go wrong only when the source program goes wrong. For instance, heap accesses can fail if the index is out of bounds. We thus know that FM-JIT will only execute a failing heap access if the source program did so.

4 SPECIFYING FM-JIT WITH A MONADIC ENCODING

A static compiler like CompCert is written as a pure and terminating program. Hence, it is a prime candidate for a traditional extraction workflow: one can write a compiler as a Coq function, then one can extract that function to an equivalent executable OCaml function. However, JITs are impure and effectful programs. Yet, these impure parts (*e.g.*, calling native code or interacting with global data-structures) are not the only things a JIT does. A JIT must also translate code (with its backend compiler), interpret code, and orchestrate its components (with its monitor). All these remaining parts, making up for most of a JIT's work, can be written in a pure functional language and can be formally verified. In this section, we are tasked with the challenge of verifying in Coq a program that can only be partially written in Coq.

Our solution is to design a formalism inspired by free monads to write incomplete Coq programs, that may contain pure computations, but may also contain calls to some unimplemented impure primitives. In short, these incomplete programs contain all of the pure parts of the program, and can be written in Coq. In fact, free monads have been used for years in pure functional languages to represent incomplete programs [Swierstra 2008]. Incomplete programs are not executable yet, as they lack some implementations. FM-JIT can be written as an incomplete program, using the list of primitives of Section 3.3, and also writing every pure computation of a JIT: most of the monitor, dynamic compiler, and interpreter.

Obviously, this is not enough on its own, we want a JIT that is both verified and executable. For the verification of such an incomplete program, we can fill its holes with specifications for the primitives. In Section 4.1, we present state and errors monads, which provide a good formalism to encode in a pure functional language all of our primitive specifications. For the JIT, we can write

```
Inductive sres (state A:Type) : Type :=
| SError : errmsg → sres state A
| SOK : A → state → sres state A.

Definition smon (state A:Type) : Type :=
  state → sres state A.
```

```
Definition sret state A (x:A) : smon state A :=
  fun (s: state) ⇒ SOK x s.
Definition sbind state A B (f: smon state A)
                (g: A → smon state B) : smon state B :=
  fun (s: state) ⇒ match (f s) with
    | SError msg ⇒ SError msg
    | SOK a s' ⇒ g a s'
    end.
```

Fig. 6. Defining the Coq state and error monad.

pure Coq monadic functions working on an abstract model of the JIT memory to specify each primitive. The free monad used to write FM-JIT is defined in Section 4.2. Next, Section 4.3 shows that an incomplete program can be completed with such primitive specifications. The result is a Coq function that specifies the behavior of the complete program. We use that Coq function to define the small-step semantics of FM-JIT that we can then prove correct.

To get an impure executable JIT from such an incomplete program, we must move away from Coq. We extract it to an incomplete OCaml program. In OCaml, where effects are possible and C functions can even be called, we define impure implementations of each primitive, working on actual shared data-structures and really calling native code. In Section 4.4, we show that both the incomplete program and the primitive implementations can be composed together to get an executable impure program.

Using free monads not only has the advantage of clearly identifying and specifying the various effectful components of a JIT, it also allows us to switch between different specifications of our data-structures. This is the *refinement* methodology of Section 4.5, facilitating the proofs of a multi-language JIT. Moreover, the meta-theory needed to reason about our free monad implementation is very lightweight (a few hundreds of Coq lines). In particular, we only ever use free monads to write terminating computations, which eliminates the need for coinductive reasoning. We show in Section 4.6 that our monads can be used in conjunction with the small-step semantics framework of CompCert to reason about possibly infinite behaviors.

There are two main drives behind our chosen formalism. First, the meta-theory used to reason about effectful programs should be as simple and lightweight as possible, to not hinder already difficult proofs. Second, our formalism should be entirely compatible with CompCert so that we can reuse both its correctness proof and its simulation framework, which already handles proof composition and diverging executions. In the end, this free monad formalism could be used for any verification work trying to extend the CompCert simulation framework to effectful programs.

4.1 An Existing Solution for Specifying Effects: State and Error Monads

A standard way to encode global effects in pure functional languages is to use *state monads*, an abstraction of computations modifying a global state. In FM-JIT, we use a variation of the state monad that also includes errors: our primitives may modify a global state, but also fail (for instance, when trying to pop an empty stack). That state and error monad definition can also be found in CompCert. The CompCert definition of state monads that we reuse is on the left of Figure 6. Intuitively, a state monad of type `smon A` represents computations that either return a value of type `A` and possibly change the global state, or fail. The `smon` type is parameterized by a type `state` representing global states, which contain a model of the stack, heap, and executable codes. The type `sres` represents the possible return values of the monads. Such state monads are executable

```
Inductive primitive: Type → Type :=
| Prim_Push: int → primitive unit
| Prim_Pop: primitive int
| Prim_HeapSet: int → int → primitive unit
| Prim_HeapGet: int → primitive int
...
```

```
Inductive free (T : Type) : Type :=
| pure (x : T) : free T
| impure R (prim : primitive R)
    (cont : R → free T) : free T
| ferror (e : errmsg) : free T.
```

Fig. 7. Definition in Coq of free monads.

Coq functions, taking as argument an initial global state, and returning its return value as well as the new global state (or an error).

Like all monads, state monads come with helpful constructors `sret` and `sbind` to build complex monadic computations, as defined on the right of Figure 6. The constructor `sbind` sequentially chains together monadic computations, constructing entire programs that may have global effects. For instance, executing `sbind f g` in some global state `s` first executes `f` on `s`. If that computation succeeded and returned a value `a` and modified the global state to `s'`, then we execute `g` on `a` and `s'`.

We could write FM-JIT as a state monadic computation whose state contains a model of the stack, the heap and the generated native codes. However, extracting such a JIT to OCaml would only produce a pure OCaml program, because the Coq extraction only targets a pure subset of OCaml. With such an approach, there is no hope to get an executable JIT that installs and calls actual native code. In the end, state monads are a great formalism to specify primitives that modify a global state, but this is not enough to implement an effectful executable JIT.

4.2 Our Solution to Write FM-JIT in Coq: Free Monads

Since the primitives cannot be written in Coq, FM-JIT is an incomplete Coq program. Free monads are a convenient formalism to write incomplete programs, namely programs with some holes left to represent the primitives that have not yet been implemented. Free monads provide a DSL to write such programs given a list of primitives that they may use. Such incomplete programs will either be completed with specifications of the primitives (Section 4.3), or with impure implementations of the primitives (Section 4.4). First, we inductively define the primitives our free monad definition uses as on the left of Figure 7. We see that one possible primitive is `Prim_Push`, taking an `int` as argument and not returning any value.

Effectful computations are defined on the right of Figure 7. A term of type `free T` is called a free computation; it encodes a function that computes some value of type `T`, possibly using some primitives. The computation is either a pure computation, not using any primitive, an error, or an impure computation. Such an impure computation calls an unimplemented primitive `prim`, and then a continuation `cont` encodes the rest of that computation (possibly using other primitives), given the return value of the primitive. This simple type is quite easy to manipulate. For instance, one can write LTac tactics that automatically check that some free computation only uses some primitives and this helps us prove that the interpreter does not install any new native code.

Free monads are monads, and as such come with the monadic constructors on the left of Figure 8. Binding impure free computations entails adding primitives to the continuation. These constructors allow us to define complex free computations, such as all the components of FM-JIT. For instance, the optimizer step is detailed on the right of Figure 8. This code compiles and installs some function. First we call `IRtoRTL` which generates RTL code. This is a pure computation, using no primitive (hence the `fret`). We then call the CompCert backend that generates some equivalent x86 code. Finally, we call a primitive, `Prim_Install_Code`, that installs the generated code in an executable portion of the

```
Fixpoint fbind X Y
   (f: free X) (g: X → free Y) : free Y :=
  match f with
  | pure x ⇒ g x
  | impure R prim cont ⇒
     impure prim (fun x ⇒ fbind (cont x) g)
  | ferror e ⇒ ferror e
  end.
Definition fret X (x:X) : free X := pure x.
Definition fprim R (p:primitive R) : free R :=
  impure p fret.
```

```
Notation "'do' X ← A ; B" :=
  (fbind A (fun X ⇒ B)).

Definition optimizer (f:function) : free unit :=
  do f_rtl ← fret (IRtoRTL f);
  (* using CompCert backend *)
  do f_x86 ← fret (backend f_rtl);
  (* impure computation *)
  fprim (Prim_Install_Code f_x86).
```

Fig. 8. Free monadic constructors and using them to write FM-JIT.

```
Record monad_spec (mstate:Type): Type :=
  mk_mon_spec {
    init_state : mstate;
    prim_push: int → smon mstate unit;
    prim_pop: smon mstate int;
    prim_heapset: int → int → smon state unit;
    prim_heapget: int → smon state int
    ... }.
```

```
Definition get_prim R S (p:primitive R)
              (i:monad_spec S) : smon S R :=
  match p with
  | Prim_Push x ⇒ (prim_push i) x
  | Prim_Pop ⇒ (prim_pop i)
  | Prim_HeapSet x y ⇒ (prim_heapset i) x y
  | Prim_HeapGet x ⇒ (prim_heapget i) x
  ...
```

Fig. 9. Coq monadic specifications of representative primitives.

memory. While this looks like a simple program, `backend` is a pure function containing all of the CompCert backend, from RTL to x86. Here, the free monad encoding is used to orchestrate complex pure transformations with calls to the impure interface. To be executable, such a computation requires an implementation for `Prim_Install_Code`.

The entire JIT can be represented this way. As explained in Section 3, defining FM-JIT consists in defining the transition of a state machine. We can then describe the JIT as a function `jit_step` with the type `jit_state → free (jit_state * trace)`, where `jit_state` is the type of states of the state machine, as on Figure 4, but also contains the CoreIR functions and the data updated by the profiler.

4.3 Monadic Specifications and Semantics of Free Monads

FM-JIT is an incomplete Coq program, written using free monads and lacking some implementation for the primitives it uses. In order to derive its semantics `jit_sem`, we specify each primitive using a state and error monads (with the definitions of Section 4.1). Since our primitives work on global data-structures and may fail, the state and error monad is adequate for specifying them. We define in Figure 9 a *monadic specification* as a record containing an initial global state and a state-monad computation for each primitive. It is parameterized by the type of global states, containing a model of the stack, the heap, and the executable installed code of FM-JIT. Moreover, we define `get_prim` to access the corresponding specification of a primitive with its arguments. An example of specification is given in Figure 12 for the `heap_get` primitive.

Furthermore, to fill the holes of incomplete programs, a function called `free_to_state` in Figure 10 transforms any free monad computation `f` into a state monad computation. It simply replaces

```
Fixpoint free_to_state (A S:Type) (f: free A) (i:monad_spec S): smon S A := match f with
  | pure a ⇒ sret a
  | ferror e ⇒ fun (s ⇒ SError e)
  | impure R prim cont ⇒ sbind (get_prim prim i) (fun r:R ⇒ free_to_state (cont r) i)
end.
```

Fig. 10. Turning free computations into state and error computations.

$$\texttt{jit\_sem}\ \frac{\texttt{free\_to\_state}\ (\texttt{jit\_step}\ js_1)\ i\ ms_1\ =\ \texttt{SOK}\ (js_2, t)\ ms_2}{(js_1, ms_1) \xrightarrow{t} (js_2, ms_2)}$$

Fig. 11. The JIT small-step semantic rule.

```
int64_t heap_get (int64_t x){
  assert (x < HEAP_SIZE);
  int64_t val = jit_heap[x];
  return val; }
```

```
Definition heap_get (x:int): smon int :=
  fun s ⇒ if (Int.lt x heap_size) then
      SOK (PMap.get (pos_of_int x) (heap s)) s
      else SError ("MemGet out of memory range").
```

Fig. 12. A C primitive implementation and its Coq monadic specification, that reuses CompCert libraries `Int` (to compare 64-bit integers) and `PMap`.

recursively any call to a primitive `prim` by its specification. It uses `get_prim` to get the primitive specification, and the continuation of an impure computation is bound to the result using the state-monad bind.

Now that free computations can be completed with primitive specifications, we can define the small-step semantics of the entire JIT. State and error monadic computations are executable Coq functions, so one can execute the one corresponding to the JIT transitions. The semantics states of the JIT execution contain both a `jit_state` (js_1), the data that can be written and manipulated in Coq (a state of Figure 4 also including the CoreIR functions), and a state of the state-monadic specification `mstate` ($ms1$), a model of the data structures that we cannot write in Coq. One simply goes from one of such state to another according to the execution of the state-monad computation given by completing the JIT free transitions `jit_step`. The single small-step semantic rule is defined in Figure 11, where i is a monadic specification, and t is the observed trace. In the figure, we omit the types `A` and `S`, implemented respectively with `jit_state * trace` and the `mstate` type of i.

4.4 An Impure Implementation for FM-JIT

Once completed it with monadic specifications, we extract FM-JIT to OCaml and complete it with impure and effectful implementations of the primitives. The result is an effectful OCaml JIT that can be executed. We write C functions for interacting with our global data-structures. For instance, Figure 12 shows the C implementation for the heap access primitive. On the right is its monadic specification used in the JIT semantics. While the C function accesses some global array `jit_heap`, the specification accesses a map contained in its monadic state `s`. Both the global array and the monadic state are unchanged and the primitive fails for out-of-bound accesses.

Next, Figure 13 shows an interpreter of free monad computations in OCaml, where `exec_prim` executes the C function corresponding to the primitive. It directly interprets the incomplete program

```
let rec free_interpreter (f: A free) : A = match f with
  | Coq_pure (a) → a
  | Coq_ferror (e) → print_error e; failwith "Free monad error"
  | Coq_impure (prim, cont) → let x = exec_prim prim in free_interpreter (cont x)  end.
```

Fig. 13. Executing free computations in OCaml.

itself, without resorting to state monads and `free_to_state`. Running our free computation `jit_step` through this `free_interpreter` until the program execution finishes, we get an executable JIT in OCaml that calls effectful primitives.

4.5 Facilitating the Correctness Proofs with Refinement

In our approach there is a small list of JIT-specific functions to manually audit: our primitives implementations must match their specifications. One could then try to write specifications that closely match what the impure implementation is doing, but in practice that monadic specification can be hard to reason with in the JIT correctness proof. In particular, for a fast access to the stack using only `Pop` and `Push` in the generated native code, one would like the execution stack to be a simple array of integers. However, when writing our proof invariants, it would be simpler if these integers were structured in several lists, each list corresponding to a specific stackframe. Moreover, in the final executable implementation, the execution stack is split in two parts: one that holds the interpreter stackframes, and one that holds the integer stackframes. Some simulation invariants (*e.g.*, in Section 5.1) are however easier to write if there is a single stack containing both interpreter and native stackframes. Then, proving the compilation of a function correct simply requires substituting its future stackframes instead of moving them from one structure to the other.

We argue that an advantage of using free monads is the ability to switch between different monadic specifications. One can then define a monadic specification that is close to the actual primitive implementations (called `prim_spec`), and another *reference* monadic specification (or `ref_spec`), with which proofs are easier to conduct. In this section, we show that once we prove that `prim_spec` *refines* `ref_spec`, then the correctness theorems about the JIT semantics using the reference implementation can be propagated to the JIT semantics using `prim_spec`. We use this technique to switch between two specifications of the execution stack: an unstructured stack of integers, close to the C implementation, and a reference one where the stack is structured for easier stack invariants. This refinement methodology takes advantage of free monads to modularly separate the correctness arguments for stack manipulation and stack representation.

The benefit is a more modular reasoning: first we prove correct the JIT using `ref_spec`, then we prove the refinement. So, as composing CompCert simulations facilitates modular proofs, instead of re-developing new proof techniques for modularity, we define our refinement relation so that we can reuse the simulation composition technique of CompCert. More precisely, a monadic specification i *refines* another j if there exists a relation $\simeq$ between monad states of i and monad states of j (written $i \simeq j$) such that for each primitive p,

$$\begin{aligned} \forall s_i\ s_j\ s_i'\ args\ r, \quad s_i \simeq s_j \quad \wedge \quad p_i\ args\ s_i \ = \ \mathtt{SOK}\ r\ s_i' \rightarrow \\ \exists s_j', \quad s_i' \simeq s_j' \quad \wedge \quad p_j\ args\ s_j \ = \ \mathtt{SOK}\ r\ s_j' \end{aligned}$$

where p_k is the state monad for p in the monadic specification k. This definition purposely resembles the forward-simulation definition of CompCert, but relating primitive executions instead of small-step semantics. We then prove the following theorem, using the refinement relation to build a simulation invariant.

```
Theorem refinement:
  ∀ (prog_state istate jstate:Type) (prog:prog_state → free (prog_state * trace))
    (i:monad_spec istate) (j:monad_spec jstate),
    refines i j → forward_simulation (jit_sem prog i) (jit_sem prog j).
```

As the JIT behavior is deterministic, that forward simulation is used to construct a backward simulation. Using that theorem on FM-JIT and the two monadic specifications discussed above, we prove the main correctness theorem of Section 3.4 with the reference implementation, and then propagate that correctness theorem to the semantics using `prim_spec`, without additional proof effort. We see the refinement theorem being used in Section 5.5. Proving the refinement (`prim_spec` ≃ `ref_spec`) is straightforward. In practice, while proving that an unstructured stack corresponds to a structured stack (a Coq list of stackframes containing themselves lists of integers) is not a difficult thing, its justification should not hinder the proofs of every program transformation.

4.6 Non-Atomicity of Transitions: Small-Step Semantics of x86 to the Rescue

As in the `optimizer` example of Figure 8, almost every transition of the JIT state-machine is written as a terminating function of type `free T` for some return type `T`. However, more than effectful transitions, a JIT compiler may also include non-terminating transitions, as the compiled program may not terminate. The dashed transition of Figure 4 represents a possibly diverging computation: the JIT may have compiled and executed a non-terminating function. This is not an issue with the IR interpreter which can be defined with some fuel (*i.e.*, an integer limiting its maximum number of steps) and return regularly to the monitor only to be called again. But the dynamically generated native code may be stuck in a loop without any return, call or deoptimization, and the fuel technique cannot be used to represent such an infinite execution. This possibly diverging transition cannot be specified like other primitives, with a terminating free computation.

One solution could be to extend our free monad to a coinductive structure to represent and reason about possibly non-terminating effectful programs, like in [Xia et al. 2020]. However, this would require coinductive reasoning on such transitions. On the other hand, the CompCert simulations are already equipped to reason about non-terminating executions, as long as such executions are defined by small-step semantics. Moreover, x86 semantics in CompCert are already specified with executable small-step semantics, looping a function that computes the next semantic state.[1] As a result, the non-atomic transition of the JIT can be decomposed into several atomic small steps of CompCert x86 semantics. This has two advantages: not only can we avoid writing coinductive proofs in Coq and instead reuse those of CompCert, but this also facilitates the reuse of the CompCert correctness theorem, expressed in terms of these x86 semantics, without modifying it.

To specify such native calls, we first define FM-JIT as a *Non-Atomic State Machine* (or NASM), whose transitions, defined on Figure 14, are either atomic steps or a possibly infinite sequence of native code steps. In that definition, `Trace` is the type of observable events, and `State` is the type of state-machine states. Next, we extend the OCaml `free_interpreter` of Section 4.4 so that it loads and calls native code when seeing a `LoadAndRun` transition. Such a transition is only available from the `NATIVE EXECUTION` state of Figure 4. Finally, to give small-step semantics to NASM, we specify the call to native code with three free computations `step_`, `start_` and `end_` as pictured on Figure 15. The `step_` function is the small-step transition of x86, as defined in CompCert. We simply extend it so that every call to one of our primitives is specified with its monadic effect. Moreover, we extend the small-step semantics definition of Figure 11 with rules that execute `start_` when seeing

[1] More precisely, most of x86 semantics are defined with a function in CompCert. External calls are specified with an inductive non-executable parameter. However, in the subset of x86 that our JIT generates, the only such calls are calls to our JIT primitives that we can specify as usual with Coq state-monad functions.

```
Inductive nasm_transition (State Trace:Type): Type :=
| Atomic: free (Trace * State) → nasm_transition
| LoadAndRun: nasm_transition.
```

Fig. 14. Non-Atomic State Machine definition.

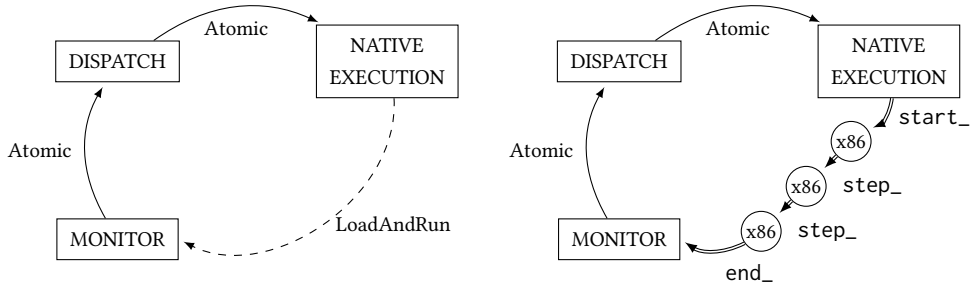

Fig. 15. Giving small-step semantics to non-atomic transitions in a NASM.

a `LoadAndRun` transition (building an initial x86 semantic state), then loop `step_` until a final x86 semantic state is reached, and execute `end_` to get back the next JIT state. In the case of a diverging native execution, these new JIT semantics will stay in x86 semantic states as intended, and the entire JIT behavior will be considered diverging.

With these simple definitions unfolding native calls according to their CompCert semantics, one can define small-step semantics for the entire JIT even in the presence of non-atomic transitions. And this allows us reusing the CompCert proof without modification (see Section 5.4).

5 PROVING THE CORRECTNESS OF FM-JIT

As JITs reuse code transformations of static compilers to generate native code, we argue that formally verifying the backend compiler of a JIT should reuse the formal verification of static compilers. In this section, we show that by carefully transforming our IR, we can directly reuse the proof of the CompCert backend, in order to prove the correctness of the native-code generation in FM-JIT. Intuitively, one could think that we can directly use this backend dynamically to transform some parts of the JIT program, and then because it generated "equivalent" code, the JIT execution semantics should not change. However, this is not as straightforward.

First, the CompCert backend correctness theorem is established by relating the semantics of an x86 program to the semantics of a RTL one. This means that our dynamic compilation step should be split in two passes: first FM-JIT generates a piece of RTL for a given function, then it uses the CompCert backend to generate some x86 from that RTL function. To prove these passes modularly, we need to reason about the intermediate program, where some RTL has been generated but not yet compiled. The JIT semantics of Figure 11 contains semantic states for the execution of x86 and CoreIR, but nothing for RTL since the RTL is never executed by the JIT. To reason about our compilation step modularly, we define mixed semantics in Section 5.2 that include semantic states for CoreIR, x86 but also RTL. These semantics are used to specify the backend of FM-JIT.

Finally, the correctness theorem of CompCert states that the observable behavior of a program is preserved. Nothing in this theorem is related to the effects on the memory, that are indeed not preserved by the CompCert backend. In fact, the backend goes from an abstract stack in RTL (a

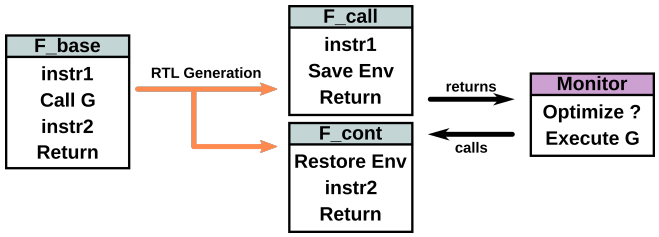

Fig. 16. Transforming CoreIR function `F_base` into two RTL functions `F_call` and `F_cont` using custom calling conventions to return to the monitor between calls.

list of RTL stackframes), to an actual execution stack. This is an issue for FM-JIT. We want to be sure that every modification to the heap done by the RTL function will be compiled to some x86 code that also modifies the heap similarly, otherwise executing the rest of the program after that function may differ. To avoid that issue while not modifying the code of the CompCert backend, we make the generated code interact with the stack and the heap only through external calls to 5 of the FM-JIT primitives (Section 5.1). This means not relying on CompCert to compile function calls, but instead generating RTL code that uses our primitives. We then define custom calling conventions relying on our primitives that the generated code uses. For instance, even though CompCert only compiles programs with no arguments, our solution consists in generating RTL programs that start by popping their arguments off the stack. In the end, the native programs we generate may use the CompCert memory to spill registers, however for FM-JIT, we do not use the stack and heap handled by CompCert, but rather the shared data-structures manipulated by free monads.

5.1 Splitting RTL Programs to Directly Reuse CompCert Proofs

CompCert only allows the compilation of complete programs and uses its own calling conventions. By generating several pieces of code that interact with the stack only through our JIT primitives, we demonstrate that both CompCert and its theorem can be used for formally verified native code generation in FM-JIT. Our backend compilation process has two steps. For a given function to compile, we first generate several RTL programs. Then we compile each of them using the CompCert backend. The verification of these two passes is discussed in Sections 5.3 and 5.4.

To reuse the CompCert backend with custom calling conventions, we split our CoreIR functions at function calls when generating RTL. The only JIT primitives that can be called from that RTL code are `HeapGet`, `HeapSet`, `Push`, `Pop` and `Print`. The first two are used to interact with the heap. Stack primitives `Pop` and `Push` are used to save the live environment, but also store return values or function arguments. The last one is used for any compiled function with **Print** instructions.

Figure 16 shows how to split each function `F` before compiling it. When FM-JIT decides to optimize `F`, it first splits its original version `F_base` in two functions: `F_call` and `F_cont`, its continuation after the call. We add primitives to these functions that save and restore the environment (the live registers). Finally, `F_call` and `F_cont` can be compiled with a backend that preserves the primitive calls. Each one is defined as a whole RTL program. After optimization, FM-JIT will start by calling the compiled function `F_call`. When it encounters the call to function `G`, the generated native code returns to the JIT monitor, which may now optimize or simply execute `G`. When this call returns, the monitor calls the continuation function `F_cont`.

Figure 17 shows an example of a CoreIR function being compiled. The `Fun1` function does a computation, then calls another function `Fun7`, then does another computation and returns. When generating RTL, because there is only one call in `Fun1`, we split the function into two RTL programs.

```
Function Fun1 (reg1):
  reg2 ← reg1 + 4
  reg3 ← Call Fun7(reg2)
  reg3 ← reg1 + reg3
  Return reg3
```

(a) A CoreIR function.

```
$1() {
  x8 = "Pop"()
  x9 = x8 + 4 (int)
  x1 = "Push" (x8)
  x1 = "Close"(1, 2)
  x1 = "Push"(x9)
  x1 = "Push"(1)
  x1 = "Push"(7)
  x7 = RETCALL
  return x7 }

$2() {
  x10 = "Pop"()
  x8 = "Pop"()
  x10 = x8 + x10
  x1 = "Push"(x10)
  x7 = RETRET
  return x7 }
```

(b) Two Generated RTL programs.

```
# Generated by CompCert
$2:
leaq   32(%rsp), %rax
movq   %rax, 0(%rsp)
movq   %rbx, 8(%rsp)
call   _Pop
movq   %rax, %rbx
call   _Pop
leal
0(%eax,%ebx,1),%edi
call   _Push
movl   $RETRET, %eax
movq   8(%rsp), %rbx
addq   $24, %rsp
ret
```

(c) The x86 program for the continuation.

Fig. 17. Compilation of a function Fun1 by FM-JIT.

The first one ($1) starts by getting the function argument (reg1/x8) off the stack. After an instruction for the computation, it performs a call by first saving the live register x8 on the stack. It then closes the current stackframe by pushing the identifier of the current function and the label of the call (to identify the corresponding continuation function). Close is simply implemented by several calls to Push. After that, we push the call arguments, the number of arguments, and the identifier of the function we want to call (Fun7). Finally, we return with the constant RETCALL, returning to the monitor but indicating that the function wants to call another one. The monitor may now pop the function identifier and decide to optimize or execute Fun7. After the call to Fun7, its return value has been pushed to the stack. The execution then follows with the second program $2; it starts by getting the return value of Fun7, and restores the live register x8 that was pushed earlier. Finally, it ends by returning another constant, RETRET to indicate to the monitor that the execution has finished. The CompCert backend then provably preserves the calls to JIT primitives, as seen on the x86 code produced for the second RTL program of Figure 17.

During the RTL generation pass, **Assume** instructions are compiled as branches. If the speculation holds, we proceed in the rest of the program. Else, we push the deoptimization metadata on the stack and return with another constant RETDEOPT. If deoptimization occurs, we return to the monitor which will read that data from the stack and reconstruct the corresponding interpreter state.

5.2 Mixed Semantics: Interleaving Pieces of Executions Related to Three Languages

The compilation done by FM-JIT is done in two passes, and its correctness proof can also be decomposed into two correctness proofs using a simulation for each pass. This allows us to prove the splitting of functions and the use of our custom calling conventions (generating RTL) independently from the correctness of native code generation. This modular approach to compilation correctness using simulations that are then composed together is the same strategy used by CompCert to prove each of its passes independently. To express these two simulations, we need to define formal semantics for the intermediate JIT programs that are obtained after running our first RTL generation pass, jut like CompCert defines formal semantics for its intermediate languages. In the case of FM-JIT, such intermediate programs contain CoreIR, RTL and x86 code, and in this section we present

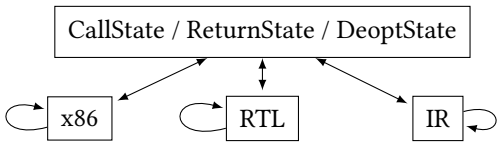

Fig. 18. Semantic states of the mixed semantics.

their formal semantics, called *mixed semantics*. Note that programs with pieces of RTL are never executed by the JIT, which waits until the backend compilation has entirely finished, and every piece of RTL is removed before resuming execution. The semantic states of these mixed semantics include the semantic states of each of these three languages semantics. To interface them, we also define three states forming a synchronization interface: CallState, ReturnState and DeoptState. Succintly, each function call, return or deoptimization goes through shared synchronization states; this is shown on Figure 18.

Figure 19 shows representative rules of the mixed semantics. The semantic states are pairs, of one mixed state of Figure 18 and one monadic state of the reference monadic specification (containing the stack and the heap). First, one stays in a language according to its semantics until reaching a function call, a return or a deoptimization. For instance, in rule (step x86), we follow the execution of the x86 semantics. On any instruction that is not an external call, the monadic state is unchanged. There is a similar rule reusing RTL semantics, (step RTL). However, we extend both these semantics with monadic rules when calling JIT primitives. For instance, in rule (push x86), if the current x86 instruction calls the primitive `Push`, we update the monadic state *ms* with the execution of the primitive specification. We move to the next x86 state `s'` with function `next_state`, moving to the return address. Along with (step x86), these rules form the modified x86 small-step semantics that we also use to specify the `step_` function of Section 4.6.

When the x86 or RTL semantics reach a final state, we move to the corresponding synchronization state. For instance in (x86 return), upon seeing the constant `RETRET`, we move to a ReturnState. In our x86 and RTL programs, the return value is pushed on the stack, so we move to a state that does not contain the return value itself, but rather some indication (`OnStack`) that it has to be popped. In rule (call x86), we see one way to go from a synchronization state to a x86 state. If we were about to call function `f` with some arguments `args`, and see that `f` had been compiled to some native program `p`, then we would push arguments to the stack and move to the initial semantic states of `p`, mimicking the behavior of the monitor and extending it to RTL executions.

Mixed semantics include other similar rules for CoreIR and RTL. A final rule also steps from a ReturnState to a final semantic state if the stack is empty. While CallStates and ReturnStates step to any of the three languages, DeoptStates always reconstruct an interpreter state for CoreIR. Finally, all rules depicted on Figure 19 are silent and produce an empty trace. The only time an observable behavior is produced is when executing the primitive `Prim_Print`, either when calling it from x86 or RTL, or when interpreting a CoreIR **Print** instruction. These are the observable events preserved by the JIT execution.

5.3 Correctness of RTL Generation

The first compilation pass generates several RTL programs for a given CoreIR function `F`: one program for the `F` entry, and one continuation program for each function call in `F`. The generated code must contain the calls to JIT primitives that are going to be preserved by the CompCert backend and used by the native code. This means that this pass stops producing interpreter stackframes but instead uses native stackframes. Proving this pass correct then means proving correct the change of

$$\text{step x86}\ \frac{\texttt{s} \rightarrow^{x86} \texttt{s'} \quad \text{not external call}}{\texttt{(s, ms)} \rightarrow \texttt{(s', ms)}} \qquad \text{step RTL}\ \frac{\texttt{s} \rightarrow^{\text{RTL}} \texttt{s'} \quad \text{not external call}}{\texttt{(s, ms)} \rightarrow \texttt{(s', ms)}}$$

$$\text{push x86}\ \frac{\begin{array}{c}\texttt{find\_instr(s) = Call Prim\_Push [v]} \quad \texttt{next\_state (s, retval) = s'} \\ \texttt{free\_to\_state (Prim\_Push v) ms = SOK retval ms'}\end{array}}{\texttt{(s, ms)} \rightarrow \texttt{(s', ms')}}$$

$$\text{x86 return}\ \frac{\texttt{s} \rightarrow^{x86} \texttt{Final RETRET}}{\texttt{(s, ms)} \rightarrow \texttt{(ReturnState OnStack, ms)}}$$

$$\text{call x86}\ \frac{\begin{array}{c}\texttt{free\_to\_state (Prim\_Load\_Code f) ms = SOK p ms} \\ \texttt{free\_to\_state (push\_args args) ms = SOK tt ms'}\end{array}}{\texttt{(CallState f args, ms)} \rightarrow \texttt{(initial\_state p, ms')}}$$

Fig. 19. Some representative rules of the mixed semantics.

calling conventions in a function. Our proof is a forward simulation relating the mixed semantics of the current JIT program before and after transforming to RTL a given CoreIR function.

Building our invariant is crucial to proving the simulation. There are three main cases in the invariant for the transformation of a function F to RTL. First, because we are only compiling a single function, we need to relate identical semantic states when outside of that function (*refl*). However, even in that case, the execution stack can differ: some interpreter stackframes for F may have been replaced with equivalent native stackframes containing the live registers at the time of the call. Another possible case of the invariant (*rtl*) happens when executing the new RTL function (or one of the continuation). RTL semantic states are related to CoreIR semantic states. The two states must agree on live registers (not all registers, as only the live ones are restored after a call). Finally, the synchronization states (Callstate, Returnstate or Deoptstate) differ when reached from RTL or CoreIR. In RTL, the arguments of such states (like call arguments, the return value or the deoptimization metadata) have been pushed to the stack instead of directly given by the interpreter. A last invariant case *synchro* expresses that.

Figure 20 showcases an example of the invariant preservation in our simulation proof. The execution on the left corresponds to executing the program before F is transformed to RTL. Before calling the transformed function F, semantic states are related with the *refl* invariant. Then, we prove that the beginning of the execution of F in CoreIR matches its execution in RTL using the *rtl* invariant, even though in RTL the arguments have to be popped first. As we see a function call to G (at 12), the CoreIR interpreter simply builds a Callstate containing the arguments. In RTL however, we need to push that to the stack and end on a different Callstate. Both are related with the *synchro* invariant. Execution of G then proceeds; when it returns after several steps, the source execution simply goes back in the middle of the CoreIR F. On the RTL side, we show that by going into the corresponding continuation program F.2 and popping the return value and environment off the stack, we get to semantic states matched with the *rtl* invariant.

In practice, this proof is conducted in two steps: we first generate RTLblock, a language we wrote where labels can be associated to basic blocks of instructions (instead of single instructions like in RTL). We then transform that RTLblock code into RTL code, unfolding the basic blocks. To that end, we extended the mixed semantics of Section 5.2 to also include the semantics of RTLblock. Using RTLblock as an intermediate language between CoreIR and RTL allows us to have simpler invariants, where every CoreIR instruction is matched with a single basic block. Both steps are proved correct with forward simulations that we compose.

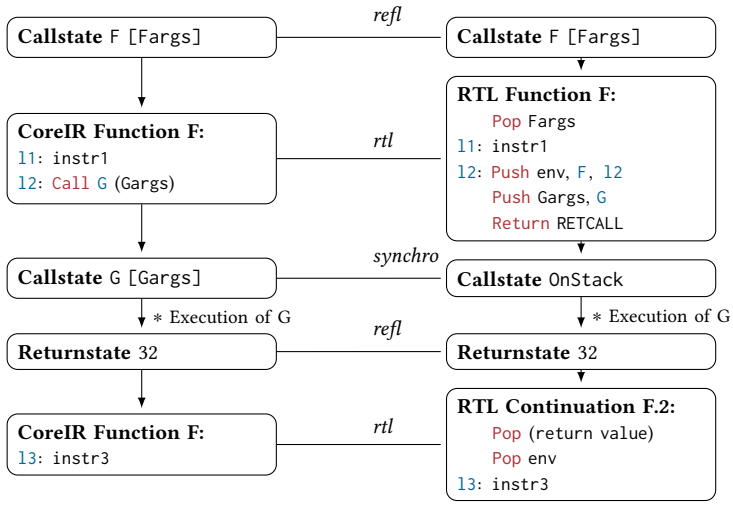

Fig. 20. Preservation of the invariant while transforming Function F to RTL.

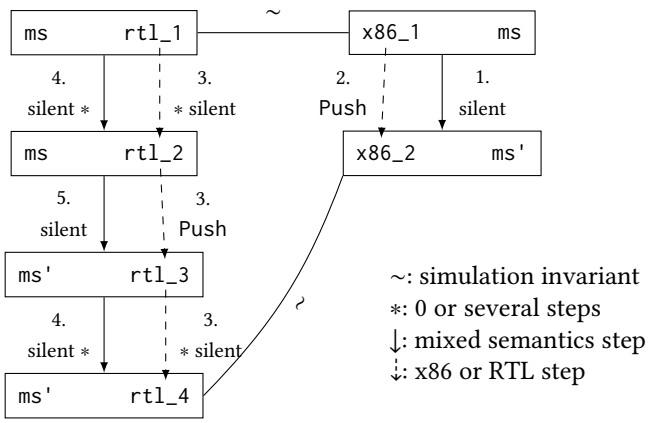

Fig. 21. Preserving a primitive call in the mixed semantics.

5.4 Correctness of Native Code Generation

To prove correct the pass that uses the CompCert backend to transform RTL into x86, most of the effort lies in reusing CompCert simulations, relating x86 and RTL semantics, to construct a backward simulation on mixed semantics. We first define a simulation invariant that relates two states of the mixed semantics $(rtls, ms)$ and $(x86s, ms)$ where $rtls$ and $x86s$ are semantic states of respectively RTL and x86, related by a CompCert simulation. Since the programs have the same effects, the monadic state ms must be identical. Next, we prove a backward simulation: every semantic step of the mixed semantics after calling the CompCert backend is related to some steps of the mixed semantics before calling the backend. Because the mixed semantics combine x86 and RTL semantics, we can reuse CompCert simulations again.

For instance, Figure 21 illustrates one of the interesting cases of the proof: when the mixed step (1.) of the compiled program is calling the Push primitive (rule (push x86) of Figure 19). We

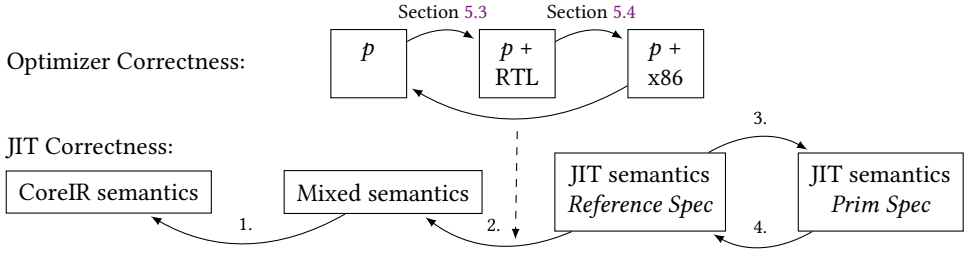

Fig. 22. Composing Simulations for the JIT Correctness Theorem.

prove that this corresponds to a step (2.) with an observable behavior in the x86 semantics. Then, using the CompCert backward simulation, we know that this step is matched by some steps in RTL semantics. These RTL steps emit the same behavior and can be split into three parts (3.) around the non-silent step. We prove that silent RTL steps correspond to silent mixed steps (4.), and that the RTL call at state `rtl_2` corresponds to a silent mixed step (5.) where the monadic effect of `Push` has been applied, turning `ms` into `ms'` just like it did on the x86 side. Finally, we have proved that the single silent step on the right can be matched with steps on the left that have the same monadic effect and thus preserve the invariant. We then prove the entire backward simulation on mixed semantics. Outside of the compiled code, the invariant is simply reflexive.

5.5 Putting Pieces Together: Main Correctness Theorem of FM-JIT

Finally, we prove the entire JIT correct by relating its semantics `jit_sem` to those of its original CoreIR program. To do so, we write three more backward simulations. Figure 22 illustrates the different composed simulations in our proof, where right-facing arrow represent forward simulations, and left-facing arrow represent backward simulations. First (1.), we prove that for any CoreIR program, its CoreIR semantics and its mixed semantics are simulated. CoreIR semantics use neither free monads nor the global data-structures of FM-JIT. On such a program with no native code yet, it suffices to prove that using the JIT stack and heap on a CoreIR-only program is equivalent.

Then (2.), we prove that for any program, its mixed semantics and its JIT semantics using the reference monadic specification are backward simulated. There are two main challenges in this proof. First, one needs to prove that inserting dynamic optimizations does not change the behavior. This is done by reusing the nested simulation technique of [Barrière et al. 2021]. In a few words, inserting optimizations dynamically in the execution of a program can be proved with a backward simulation as long as this optimization step is proved correct with a backward simulation. Our optimizer proof purposely fits this criteria, as we compose the two simulations of Sections 5.3 and 5.4. Second, we prove that the JIT semantics correspond to the mixed semantics interleaved with optimizations. This entails relating states of the mixed semantics (Figure 18) to states of the JIT state-machine (Figure 4). This also requires proving that the extended semantics of Section 4.6 for non-atomicity match the behavior of the mixed semantics when calling x86 code. To assemble these two arguments in practice, we define other intermediate semantics and compose two backward simulations.

All of these simulations are written for the reference specification, where the JIT stack is structured. We then use the refinement theorem of Section 4.5 to prove a forward simulation (3.) between the JIT semantics using the reference specification and the JIT semantics using `prim_spec`, a monadic specification much closer to the actual impure C primitive implementations. This is a

forward simulation that can be transformed into a backward simulation (4.) since the JIT behavior is deterministic.

Composing all these backward simulations, we get the `jit_correctness` theorem of Section 3.4. Every JIT behavior using `prim_spec` of a program `p` refines a behavior of the CoreIR semantics of `p`.

6 ASSESSMENT OF FM-JIT AND PERSPECTIVES

This section first explains how to run FM-JIT. Then, it discusses some of our design choices and some optimizations left as future work. Last, the section quantifies our formal development and our proof effort, before detailing our trusted code base.

6.1 Running the Executable FM-JIT

We wrote a C implementation for all the JIT primitives, that the OCaml free interpreter of Section 4.4 can call. Each of the stack and heap primitives uses a global array of 64-bit integers, and resembles its monadic specification (see Figure 12). The primitives to install code are more involved and use system calls. The implementation of `Install_Code` allocates memory with `mmap`, writes into it after assembling the output of CompCert, then makes it executable with `mprotect`. We use the `elf` library[2] to get the binary code of the assembled file.

After writing a parser for CoreIR and simple profiler heuristics (optimizing a function after a given number of calls), we can run FM-JIT on actual programs. This helps us validate experimentally that FM-JIT is executable and that primitive implementations behave as expected. Modern JIT compilers generate native code dynamically to execute faster than simply using an interpreter and this experiment shows that this is true of FM-JIT as well. On some programs, we observe up to 40 times speedups when dynamically compiling hot functions, compared to running our extracted interpreter alone. This is the case for instance for a naive program searching for prime numbers, when the function that uses a loop to check if a number is prime gets compiled. Of course, these speedups are only as good as the profiler and real programs may require better heuristics. When compiling a function that is barely called after, we can observe an execution time overhead due to calling the optimizer. We have tested successfully FM-JIT on all the features of CoreIR, including programs with speculation and deoptimization. These examples programs are included in the artifact [Barrière et al. 2022].

6.2 Design Choices and Possible FM-JIT Optimizations

The design space of modern JIT compilers is particularly large, and ungoverned by any standard. For instance, some JITs compile entire functions like we do (HotSpot), and others compile execution traces (TraceMonkey). Some JITs use other IRs, like SSA or Sea-of-Nodes, that extend RTL-like languages with richer properties. This is not a JIT-specific issue, and there exist previous works on using other IRs in a verified static compiler [Barthe et al. 2014]. The main drive behind the design choices of FM-JIT is to define a simple yet representative architecture that captures JIT-specific behaviors (like interleaving native code and interpretation) that have yet to be formalized.

One may wonder if our synchronization interface, going back and from the monitor at each function call, can be a bottleneck for execution. In [Barrière et al. 2021], CoreJIT used an unverified optimization when compiling a call to another already compiled function: the LLVM backend generated a direct call to that function. This means that going back to the monitor is only needed when going to the interpreter is needed, and execution can stay at the native level as long as possible. We believe that, as future work, a similar optimization asking CompCert to compile function calls for compiled functions, could be done and verified using FM-JIT. This is a benefit of

[2] https://man7.org/linux/man-pages/man5/elf.5.html

having formalized and mechanized native code generation in a JIT: optimizations whose correctness depends on several components (the backend and the monitor) are now possible to prove.

Similarly, using external calls in the native code for each stack and heap interaction could be detrimental to execution times. These external calls define a clear interface of the impure effects of the JIT. One could now imagine possible optimizations, either inlining x86 implementations of the primitives as an additional step of the backend, or defining custom builtin functions at the RTL level that CompCert can also inline and compile. Having defined semantics for native code execution in a JIT compiler, such optimizations could be proved correct in FM-JIT.

6.3 Proof Reuse

Our correctness proof successfully reuses a lot from CompCert. The simulation framework is used for the final JIT theorem. Liveness analysis used when generating RTL is done using the Kildall library of CompCert. The behavior theorem that goes from a backward simulation to the final theorem of Section 3.4 also comes from CompCert and is the only proof using coinduction.

The entire Coq development represents around 15K lines we wrote, and includes more than 160K lines from CompCert. This proof reuse allowed our proofs to entirely focus on JIT-specific issues. We also wrote around 1K lines of OCaml for the free interpreter, profiling, parsing, pretty-printing and interfacing with C primitives. We include around 46K OCaml lines from CompCert. Our C library is smaller than 500 lines.

6.4 Trusted Code Base

While most of the JIT is proved correct in Coq, there is still some code that has to be trusted in order to fully trust the OCaml executable JIT:

- The Coq to OCaml extraction of the free JIT.
- The OCaml function that loops the JIT step should correspond to the JIT semantics. This is a just a few lines of code repeatedly calling the free interpreter until reaching a final state.
- The free interpreter of Section 4.4. This calls C functions from OCaml.[3]
- The primitive implementations should correspond to their monadic specifications (Figure 12).
- The call to native code should correspond to the three monadic specifications `start_`, `step_` and `end_` of Section 4.6.

The task of verifying that the primitive implementations comply with their monadic specifications is out of scope of this paper. In our work, we prove correct exactly the parts of the JIT that are extracted to OCaml, given a specification of the rest. This orthogonal verification work could be done with the help of Hoare logic as well, for instance using VST [Appel 2015].

First, note that when implementations deviate too much from their specification, one could use the refinement methodology to define another specification, closer to the C implementations. However, there are a still few examples where the monadic specifications may not exactly match up with their implementations. First, even in the `prim_spec` specification, the stack is assumed to be infinite. In practice, we have implemented it as a finite array of 64-bits integers. This is not a JIT-specific issue, and CompCert also assumes to be working with an infinite memory. There exists a CompCert variant [Wang et al. 2019] that allows reasoning about bounded stack usage that we could investigate in a JIT setting as future work.

Second, in the formal model, the native code is represented by its x86 AST, just like in CompCert. In practice however, we call an assembler to generate machine bytes for the native code. These machine bytes are installed in the memory, not the x86 AST. This is not a JIT-specific issue, and we decided to go as far as CompCert goes in our formal model, at the cost of trusting the assembler.

[3]Currently, this uses `Obj.magic`. We could avoid that if Coq extraction could generate a GADT for the `primitive` type.

Also, in its monadic specifications, the primitive that installs code never fails. In practice however, our calls to `mmap` could fail if we ran out of memory to install the dynamically generated code. We could solve this issue by allowing the primitive specification to non-deterministically fail and in such cases cancel the optimization step. In our experiments so far, we have never encountered this issue. Finally, we also need to trust that calling the native code is correctly specified with the three monads of Section 4.6, `start_`, `step_` and `end_`. The `step_` function simply reuses the CompCert x86 semantics, with an exception for primitives as seen on Figure 19. In our experiments, we did not find any bugs with the execution of native code linked with primitives.

The CompCert theorem only holds for complete programs, where every piece of code has been compiled as a whole. This is not the case of the functions compiled by FM-JIT. We add three simple axioms in our Coq development to still reuse the CompCert theorem. The first one strengthens an existing axiom of CompCert, specifying that calls to our JIT primitives have a precise annotated behavior in CompCert semantics, leaving unchanged the memory handled by CompCert. We also assume that for each primitive and compiled function, there exists a place in CompCert memory where they have been allocated. In practice, primitives have been compiled outside of the memory modeled by CompCert, as part of the JIT C library, but without this assumption the RTL programs we produce would have no semantics. However, we then edit the generated code by CompCert to jump to the actual addresses of the primitives. These two axioms are reminiscent of the way CompCert has dealt with some helper functions for integer arithmetics. These axioms are not incomplete proofs, but consequences of the need of CompCert to model a view of the complete memory. Extending CompCert to support separate compilation, where several programs have different views of the memory, requires new proof techniques [Song et al. 2020; Stewart et al. 2015].

Last, we include a simple axiom that could be proved by unfolding CompCert code transformations: the CompCert backend does not generate any new built-ins call that is not already in the RTL programs we create. This was validated in our experiments.

7 RELATED WORK

7.1 Formally Verified JITs

Other works have tackled the issue of formalizing some parts of modern JITs. Focusing on the range analysis Javascript JITs perform, a DSL that uses an SMT solver to write and prove the correctness of range analysis in JITs has been developed in [Brown et al. 2020]. Focusing on speculations, semantics preservation proofs for speculative optimizations were first studied in the Sourir intermediate representation [Flückiger et al. 2018]. Sourir introduced formal semantics for speculative instructions like our **Assume**, and arguments for the correctness of inserting and manipulating them. CoreJIT [Barrière et al. 2021] mechanized these formal semantics and verified in Coq dynamic code transformations inserting and manipulating speculative instructions. CoreJIT itself was a prototype of a JIT compiler using CoreIR, with only interpretation and a speculative dynamic optimizer, from CoreIR to CoreIR. This prototype could also be extracted to OCaml and was completed with an unverified and unspecified LLVM backend compiler. FM-JIT shares several similarities with CoreJIT. Both purposely share a similar architecture, and both require their dynamic optimizations to be proved correct with a backward simulation. As such simulations compose, one could easily imagine implementing the speculative optimizations of CoreJIT in FM-JIT before the backend, at the only cost of adapting the proofs to the monadic semantics. However, CoreJIT did not formalize the generation of native code, its LLVM backend was not part of the formal model, and there was no proof about the interaction between IR interpretation and native execution. Modern JITs rely greatly on that interplay which remained, until FM-JIT, an open verification problem. The CoreJIT model also restricted itself to a pure subset of JIT components,

with no shared data-structure, no code installation, no call to native code and with no insight for the verification of a realistic effectful JIT. We believe that FM-JIT has successfully tackled these remaining challenges. In the end, FM-JIT and CoreJIT target different and complementary essential verification issues of modern JIT compilers, using compatible designs.

[Myreen 2010] is another work of JIT formal verification for a stack-based bytecode, which in particular targets the challenge of dynamically generating x86 code. Proofs are mechanized with HOL4, and the result is an executable JIT compiler which dynamically generates native code for each called function. To that end, this work defines semantics for self-modifying x86 code. However, this JIT workflow is not characteristic of modern JIT compilers that interleave multiple tiers of execution (interpretation and native code execution). There are no speculations and without an interpreter, no on-stack-replacement. We believe that FM-JIT proposes a design more typical of modern JIT compilers, enabling reasoning about their precise interoperability.

7.2 Verifying Effectful Programs in Coq

Our work is not the first Coq mechanization about effectful programs. There are various ways to go around the limitations of Gallina as a programming language. For instance, a monadic approach is followed with Isabelle/HOL for the verified seL4 microkernel [Cock et al. 2008], in order to refine monadic functional specifications into a C implementation. Moreover, [Pit-Claudel et al. 2020] avoid the extraction from Coq to OCaml to produce effectful verified programs. They directly transform a functional specification into a fully linked assembly program represented as a Coq term. Modifying Coq extraction so that it can produce effectful OCaml programs has also been investigated for programs using mutable arrays [Sakaguchi 2018]. Programs are written using a state monad encoding, and the improved extraction produces efficient OCaml code using mutable data-structures. In contrast, our formalism allows us to use extraction to OCaml without modifications, but only translating the pure parts of the program.

Other contributions have explored using variations of free monads before us, but our version was designed to be lightweight, specialized to our JIT and compatible with CompCert. FreeSpec [Letan and Régis-Gianas 2020] uses a free monadic definition similar to ours to encode programs with effects. Their definition is more general in the sense that one can compose several interfaces of primitives, while our single interface of primitives is specialized to those used by the JIT. Our monadic definitions mainly differ in the way primitives are specified. We use state monads allows to simply define small-step operational semantics *à la* CompCert.

Interaction Trees [Xia et al. 2020] are a coinductive variant of free monads; they are used in [Zakowski et al. 2021] to define the semantics of a subset of LLVM. Like in our approach, computations using interaction trees can be extracted to OCaml and executed using effectful event handlers, much like our `free_interpreter`, where events include the calls to effectful primitives. Their use of a coinductive structure allows interaction trees to represent diverging computations with some added monadic constructors, at the only cost of developing a library for coinductive reasoning. Coinductive reasoning can be difficult to work with in Coq. We entirely avoid this issue by breaking down what a JIT does in small, atomic computations (the transitions of Figure 4). Even possibly diverging transitions can be broken down themselves in small steps (Figure 15). Finally, our lightweight monadic library does not have the slightest coinductive proof but we rather reuse one of CompCert, going from a simulation to the `jit_correctness` theorem of Section 3.4. Interaction Trees are an impressive framework for the verification of effectful diverging programs, but in the specific case of a formally verified JIT, there is much to gain by staying close to CompCert and its proof techniques to reuse its correctness proof.

Our refinement methodology is also reminiscent of other works using refinement to prove the correctness of a concrete implementation using a more convenient abstraction. For instance, Isabelle

includes a refinement framework which has been used to verify network flow algorithms [Lammich and Sefidgar 2019]. In contrast, our refinement methodology is designed specifically for the novel interaction between the free monad formalism and the CompCert simulation framework. Our refinement definition purposely resembles a CompCert forward simulation.

8 CONCLUSION

JIT compilers are complex pieces of software relying on cutting edge techniques. Not only do they generate native code, like static compilers, but they also include an entire execution environment with various components. Modern JITs have been scarcely formalized, and are in dire need of demystification if one ever wants strong guarantees on the execution of a JIT. The need for such guarantees is made more crucial by the adoption of JITs in many critical environments, including to run possibly adversarial programs as in most of modern Web browsers.

While there is no general agreement on the components of a JIT and their interplay, we believe that FM-JIT is a reasonable proposal that works well in the context of compiler verification. This mechanized JIT in Coq captures the essence of a JIT with native code generation, where each component is specified and proved correct. We demonstrate that, just like JIT compilers reuse static compilation techniques, formally verified JIT compilers can reuse formally verified static compilers like CompCert. Writing a JIT with native code generation entirely in Coq is impossible due to its intrinsically effectful nature. However, we were able to design a JIT where effects are delimited to a very restricted set of primitives that are specified in Coq. This methodology also allows us to extract FM-JIT to OCaml and complete it with effectful implementations of the primitives. In the end, we have a JIT that is both executable and formally verified in Coq.

With these essential JIT-specific issues solved, one can now imagine many ways to go forward in verifying more realistic JIT compilers. An interesting and short-term one would be to extend FM-JIT with the speculation insertion of CoreJIT [Barrière et al. 2021], now that speculative instruction compilation is provably feasible. Another direction for future work is to extend our JIT to a more realistic input language, such as WebAssembly [Watt et al. 2021] which already has a semantics mechanized in Coq.

ARTIFACT AVAILABILITY

A virtual machine image containing FM-JIT and the tests discussed in Section 6.1 is available as an artifact [Barrière et al. 2022]. The FM-JIT development, including its proofs, can also be found at https://github.com/Aurele-Barriere/FM-JIT.

REFERENCES

Andrew W. Appel. 2015. Verification of a Cryptographic Primitive: SHA-256. *ACM Trans. Program. Lang. Syst.* 37, 2 (2015), 7:1–7:31. https://doi.org/10.1145/2701415

Aurèle Barrière, Sandrine Blazy, Olivier Flückiger, David Pichardie, and Jan Vitek. 2021. Formally verified speculation and deoptimization in a JIT compiler. *Proc. ACM Program. Lang.* POPL (2021). https://doi.org/10.1145/3434327

Aurèle Barrière, Sandrine Blazy, and David Pichardie. 2022. Artifact for Formally Verified Native Code Generation in an Effectful JIT. https://doi.org/10.5281/zenodo.7149192

Gilles Barthe, Delphine Demange, and David Pichardie. 2014. Formal Verification of an SSA-Based Middle-End for CompCert. *ACM Trans. Program. Lang. Syst.* 36, 1 (2014), 4:1–4:35. https://doi.org/10.1145/2579080

Fraser Brown, John Renner, Andres Nötzli, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Towards a verified range analysis for JavaScript JITs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020*. ACM, 135–150. https://doi.org/10.1145/3385412.3385968

David Cock, Gerwin Klein, and Thomas Sewell. 2008. Secure Microkernels, State Monads and Scalable Refinement. In *Proc. of TPHOLs 2008*, Vol. 5170. Springer, 167–182. https://doi.org/10.1007/978-3-540-71067-7_16

Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek. 2018. Correctness of speculative optimizations with dynamic deoptimization. POPL (2018). https://doi.org/10.1145/3158137

HotSpot 2022. *Java HotSpot Performance Engine*. HotSpot. https://openjdk.org/groups/hotspot/

Inria 2022. *The Coq proof assistant reference manual*. Inria. http://coq.inria.fr Version 8.12.1.

Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. 2018. Crellvm: verified credible compilation for LLVM. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018*. ACM, 631–645. https://doi.org/10.1145/3192366.3192377

Daniel Kästner, Jörg Barrho, Ulrich Wünsche, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. 2018. CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler. In *ERTS2 2018 - 9th European Congress Embedded Real-Time Software and Systems*. 3AF, SEE, SIE, 1–9. https://hal.inria.fr/hal-01643290

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *Proceedings of POPL*. https://doi.org/10.1145/2535838.2535841

Peter Lammich and S. Reza Sefidgar. 2019. Formalizing Network Flow Algorithms: A Refinement Approach in Isabelle/HOL. *J. Autom. Reason.* 62, 2 (2019), 261–280. https://doi.org/10.1007/s10817-017-9442-4

Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of POPL*. https://doi.org/10.1145/1111037.1111042

Xavier Leroy. 2009a. Formal verification of a realistic compiler. *Commun. ACM* (2009). https://doi.org/10.1145/1538788.1538814

Xavier Leroy. 2009b. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert - A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. SEE. https://hal.inria.fr/hal-01238879

Thomas Letan and Yann Régis-Gianas. 2020. FreeSpec: specifying, verifying, and executing impure computations in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP*. https://doi.org/10.1145/3372885.3373812

Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony C. J. Fox. 2019. Verified compilation on a verified processor. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019*. ACM, 1041–1053. https://doi.org/10.1145/3314221.3314622

Magnus O. Myreen. 2010. Verified just-in-time compiler on x86. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*. ACM, 107–118. https://doi.org/10.1145/1706299.1706313

Scott Owens, Michael Norrish, Ramana Kumar, Magnus O. Myreen, and Yong Kiam Tan. 2017. Verifying efficient function calls in CakeML. *PACMPL* 1, ICFP (2017), 18:1–18:27. https://doi.org/10.1145/3110262

Clément Pit-Claudel, Peng Wang, Benjamin Delaware, Jason Gross, and Adam Chlipala. 2020. Extensible Extraction of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and Proofs. In *Proc. of IJCAR 2020*, Vol. 12167. Springer, 119–137. https://doi.org/10.1007/978-3-030-51054-1_7

PyPy 2022. *PyPy Python Implementation*. PyPy. https://www.pypy.org/

Kazuhiko Sakaguchi. 2018. Program Extraction for Mutable Arrays. In *Functional and Logic Programming - 14th International Symposium, FLOPS 2018*, Vol. 10818. Springer, 51–67. https://doi.org/10.1007/978-3-319-90686-7_4

Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2020. CompCertM: CompCert with C-assembly linking and lightweight modular verification. *Proc. ACM Program. Lang.* 4, POPL (2020), 23:1–23:31. https://doi.org/10.1145/3371091

Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Proc. ACM Program. Lang. POPL 2015*. ACM, 275–287. https://doi.org/10.1145/2676726.2676985

Wouter Swierstra. 2008. Data types à la carte. *J. Funct. Program.* (2008). https://doi.org/10.1017/S0956796808006758

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. 2016. A new verified compiler backend for CakeML. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016*. ACM, 60–73. https://doi.org/10.1145/2951913.2951924

V8 2022. *V8 Javascript Engine*. V8. https://v8.dev/

Yuting Wang, Pierre Wilke, and Zhong Shao. 2019. An abstract stack based approach to verified compositional compilation to machine code. *Proc. ACM Program. Lang.* 3, POPL (2019), 62:1–62:30. https://doi.org/10.1145/3290375

Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. 2021. Two Mechanisations of WebAssembly 1.0. In *Formal Methods - 24th International Symposium, FM 2021*, Vol. 13047. Springer, 61–79. https://doi.org/10.1007/978-3-030-90870-6_4

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* POPL (2020). https://doi.org/10.1145/3371119

Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. 2021. Modular, compositional, and executable formal semantics for LLVM IR. ICFP (2021). https://doi.org/10.1145/3473572

Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the Symposium on Principles of Programming Languages, POPL*. https://doi.org/10.1145/2103656.2103709

Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2013. Formal verification of SSA-based optimizations for LLVM. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013*. ACM. https://doi.org/10.1145/2491956.2462164