Master research Internship

Internship report

Formally Verified Just-In-Time Compilation

Domain: Programming Languages

Author:
Aurèle Barrière

Supervisors:
Sandrine Blazy

Celtique

Abstract: In recent years, just-in-time compilers have proved to be useful in many cases. By interleaving interpretation and compilation at run-time, just-in-time compilers achieve far better performance than standard interpretation. Compilation inside a just-in-time compiler is done differently from standard static compilation: being made on-the-fly, it has access to run-time information. Because compilation happens during the execution, performance depends on striking a good balance between the time spent compiling and the time saved by the optimizations. Compilers in general have been the subject of many works of formal verification, but just-in-time compilers include many new components and differ in many ways with more traditional compilers. Compiler correctness is crucial to the development of reliable software, and the popularity increase of just-in-time compilation leads us to believe that just-in-time verification should be studied. This internship focuses on formal verification techniques for just-in-time compilation, and presents a verified just-in-time compiler in Coq.

Contents

1 Introduction 1

2 Formal Verification 3
2.1 Proof Assistants . . . 4
2.2 Formal Verification of Compilers . . . 5

3 Just-in-Time Compilers 6
3.1 Presentation . . . 6
3.2 Trace-based Just-in-Time Compilers . . . 7
3.3 Partial Evaluation . . . 9
3.4 A Real Just-in-Time Compiler: Graal . . . 9
3.5 Formalizing Just-in-Time Compilers . . . 11

4 A JIT in Coq 13
4.1 The JIT IR language . . . 14
4.2 The Coq JIT compiler . . . 19
4.3 Implementation . . . 23

5 Proving the JIT compiler 24
5.1 Compiler Correctness . . . 24
5.2 Simulations for static compilers . . . 26
5.3 Simulations for a JIT compiler . . . 28
5.4 Proving optimizations correct . . . 34

6 Future Work: Transparency 37

7 Conclusion 38

1 Introduction

Programming language implementations have long been divided between interpretation and compilation. Traditionally, a compiler takes as input an entire program and translates it to some independent machine-readable code. This translation is done statically, meaning that it occurs once before the execution. After that, executing the program simply consists in executing the translated code, and the compiler is not needed anymore.

On the other hand, interpreters perform an evaluation of each instruction *on-the-fly*, meaning that it happens during execution. The interpreter is active during the entire execution. When executing a program, an interpreter considers an *interpretation window*, the next few instructions to perform. Semantically equivalent machine-readable instructions are then generated and immediately executed, and the whole process loops until termination of the program. In some cases, the interpretation window can be as small as a single instruction.

Both approaches have their own advantages. Compiled programs typically tend to perform faster. Compilers include many optimizations, and many of them rely on being able to consider the future instructions or the entire control flow. Interpreters lack this global vision, as they only translate a few instructions at a time. Furthermore, when a program loops or calls the same function, the translation to machine code will be performed once using a compiler. Indeed, the generated code can also include loops and function calls. An interpreter will most likely perform the same evaluation at each iteration. But interpreted languages, while being in general less performant, still have advantages. As there is no static compilation stage, the program can start being executed without waiting for the entire program to be translated. In some cases where some program needs to be executed as soon as the code is available, choosing an interpreted language makes sense. For instance, web development makes extensive use of interpreted languages such as JavaScript. Moreover, as an interpreter has access to dynamic information while still working on its source code (and not a compiled version), interpretation leads to better diagnoses.

In general, it makes sense to adapt your translation strategy to a language and its objectives. As time goes by, many *dynamically-typed* languages have been increasingly used, such as Python, PHP or JavaScript. In a dynamically-typed language, the type of each object is only known during the execution. The type of some object can change, and many functions or operators must first check the type of arguments before executing the right code. Dynamic typing is a design choice, leading to higher-level programs allowing rapid prototyping.

For instance, consider the program Figure 1. If the language is dynamically typed, the interpreted code corresponding to an iteration might need to check the type of `a` and `array`, then use the correct addition and product methods. Simple programs end up having complex control flows. This might lead to the instructions of Figure 2 for each iteration.

```
for (int i=0; i<length; i++) {
  sum[i]     = a + array[i];
  product[i] = a * array[i];
}
```

Figure 1: A simple loop

However, *just-in-time compilers* (or JIT compilers) have proven useful in many cases, especially for dynamically-typed languages. In a JIT compiler, programs start by being interpreted, but code that might be run frequently (loop iterations or some methods) may be compiled and optimized *dynamically*. When encountering such code, the JIT compiler can then simply execute the compiled version, then go back to the interpreter for the rest of the execution. As a result, a JIT compiler interleaves interpretation, compilation and execution of compiled code. The main advantage of using a JIT compiler is performance. Even if

```
if (a is int & array[i] is int) {
  sum[i] = int_add(a, array[i]); }
if (a is float & array[i] is float) {
  sum[i] = float_add(a, array[i]); }
if (a is string & array[i] is string) {
  sum[i] = string_add(a, array[i]); }
if (a is int & array[i] is int) {
  product[i] = int_mult(a, array[i]); }
if (a is float & array[i] is float) {
  product[i] = float_mult(a, array[i]); }
if (a is string & array[i] is string) {
  product[i] = string_mult(a, array[i]); }
i = i+1;
```

Figure 2: The interpreted instructions of Figure 1

```
assume (a is int);
assume (array[i] is int);

ai = array[i];
sum[i] = int_add(a, ai);
product[i] = int_mult(a, ai);
i = i+1;
```

Figure 3: The optimized iteration of the loop of Figure 1, with speculative optimization.

the compilation stage happens during the execution, in many cases this overhead is absorbed by the time saved by executing optimized versions of the frequently run (or *hot*) code. JIT compilation then gathers the advantages of both interpretation and compilation. The programs are fast, can use dynamic features, and the execution starts without a compilation stage. Note that the compiler inside a JIT compiler differs from standard static compilers. Indeed, as it is run on-the-fly, it leverages dynamic information about the program: the values or types of some variables for instance. This proves especially useful for dynamically-typed languages. When executing a loop, we can assume most-of-the time that the variables will keep the same types. The compiler can then compile the iteration with a type assumption. If the assumption fails, the execution is redirected to the interpreter. But in most cases, the optimized version can be used. This technique is called *speculative optimization*. In our previous example, starting from the second iteration, the compiled code could be the one of Figure 3, where the `assume` instructions tell the compiler to go back to the interpretation if the assumption fails. On top of speculative optimizations, one can also use standard optimization techniques. In this example, `array[i]` is fetched only once. This version should run faster than the interpreted instructions of Figure 2 for a big array.

Formal verification methods aim at defining formal ways to reason about a program execution and guarantee properties about it. Examples of such properties can include semantic equivalence or safety properties. To this end, verification of interpreters and compilers has been crucial. The correctness of a compiler can often be expressed as “the compiler does not introduce any bug”, a bug-free source code should lead to a bug-free compiled code. While compiler and interpreter verification are extensively studied topics, few works have tackled JIT compiler verification.

Section 2 and 3 are a presentation of state-of-the-art formal verification and JIT compilers. These sections are mostly taken from the bibliographic report, except for Section 3.5.2 and more superficial changes. Section 4 presents the implementation of our JIT compiler, written in Coq [2]. We defined its input and target language to allow speculative optimization. The JIT optimizer includes two optimizations passes so far. Our work also includes a proof of correctness. Informally, it means that the behavior of the JIT matches the behavior of the source program. We build on classical formal verification techniques (formal semantics and simulation relations) and our proofs have been written and proved in Coq. These proofs are presented in Section 5.

2 Formal Verification

As software is used in many critical situations such as healthcare, energy or transportation, designing reliable software is crucial. Formal verification is a domain that aims at formally proving the correctness of software. For instance, a formal verification proof might state that a program is correct with respect to its *specification*. Specifications describe formally the expected properties about a program behavior. In order to complete such proofs, one needs a formal way to express specifications, and a formal way to describe a program behavior.

Formalizing a program behavior is known as defining its *semantics* [26]. A way to prove properties about a program is to directly reason about its semantics. There are many ways to define it. One category of them is *operational semantics*. There are two kinds: *big-step* operational semantics, where the program is directly related to its end result (for instance a return value). And *small-step* semantics, where a program behavior is defined as a sequence of configurations in a transition system. Each transition is an individual step of the execution. A configuration can represent an intermediate state, containing for instance the values of each variable and the next instructions to execute. As an instruction is executed, the configuration changes. Defining an operational semantics implies defining a set of formal rules that dictate these changes. For instance, in a simple imperative language, one could define the configurations as pairs (c, σ) of c, the code left to execute and σ, a memory state. Then, one could have the following rule: if $(\texttt{c_1}, \sigma) \rightarrow (\emptyset, \sigma')$ then $(\texttt{c_1;c_2}, \sigma) \rightarrow (\texttt{c_2}, \sigma')$, where $\rightarrow$ indicates a transition between configurations, and $\emptyset$ means no more code to execute. This rule means that for each sequence of instructions `c_1;c_2`, we can execute every instruction of `c_1` and resume the execution of `c_2` with σ', the updated memory state.

Other ways to formally reason about programs include *program logics* such as Hoare logic. In Hoare logic, there is a specification language and a program language, that can be combined to write Hoare triples, of the form $\{P\}C\{Q\}$. P and Q are written in the specification language, and are respectively called precondition and postcondition of the program C. They can be written in predicate logic, using the program variables. Proving the Hoare triple $\{P\}C\{Q\}$ means that if P holds, then Q holds after executing C if this execution terminates. Finally, Hoare logic includes logic rules to prove a Hoare triple. For instance, the composition rule states that, if $\{P\}$ `c_1` $\{Q\}$ and $\{Q\}$ `c_2` $\{R\}$ hold, then $\{P\}$ `c_1;c_2` $\{R\}$ holds. Using a simple imperative language, one could prove for instance the Hoare triple $\{x = 1\}$ `x:=x+1; y:=x` $\{y = 2\}$, meaning that after executing the program with the initial value of x being 1, the value of y will be 2. Hoare logic has had many extensions to deal with various programming languages features and implementations.

A common extension to reason about pointers and memory is *separation logic*. Correctness is still modeled with a triple $\{P\}C\{Q\}$. However, the formulas for preconditions and postconditions are augmented with a new operator $*$. Informally, $P_1 \, * \, P_2$ means that the heap (or memory) can be split into two disjoint parts, one where P_1 holds, and another where P_2 holds. This operator is convenient when dealing with dynamically linked data structures and multiple objects in the memory. For instance, if $\mathit{ArrayMem}(a, p)$ means that the array a is located at address p in the memory, then we can represent two disjoint arrays as follows: $\mathit{ArrayMem}(a_1, p_1) * \mathit{ArrayMem}(a_2, p_2)$. Without this operator, one would need, in conjunction with the two *ArrayMem* predicates, to add the constraint $p_1 \neq p_2$. When dealing with many objects in the memory, such constraints would not be convenient. The separation operator allows local reasoning with the frame rule: if $\{P\}C\{Q\}$ holds, then $\{P * R\} \; C \; \{Q * R\}$ too. This means that one can simplify the goals by only considering the useful parts of the memory, then use the frame rule to integrate it in a more global proof.

```
Fixpoint length (l:list nat) :=
  match l with
  | nil ⇒ 0
  | a:: l' ⇒ S(length l')
  end.
```

```
Theorem length_app: forall l1 l2,
  length(l1 ++ l2) = length l1 + length l2.
Proof.
  intros l1 l2. induction l1.
  - (* empty list *) simpl. reflexivity.
  - (* inductive case *) simpl. rewrite IHl1. reflexivity.
Qed.
```

Figure 4: An example of proof in Coq of a program. We first define a program called `length` which computes the length of a list. We then prove a property of this program: the length of the concatenation of two lists is the sum of lengths of each list, by induction on the first list.

2.1 Proof Assistants

Once the semantics is formally defined, one could prove any program correctness on paper. But in most cases, this approach would be extremely tedious. Proof assistants are software designed to write and check proofs. Most are interactive, and provide tools to automate proofs as much as possible. Automatic provers also exist, but are still not adequate to conduct the kind of verification proof we address here. There are many different proof assistants. Three main ones are Coq [2], Isabelle [3] and HOL [1]. The use of proof assistants has not been confined to formal verification of programs. For instance, the four-color theorem, a graph coloring problem, has been proved using Coq [14].

Coq allows the user to write programs, in a functional language very similar to OCaml. It also allows the user to write theorems, and prove them using Ltac, the tactic language of Coq. A proof is a sequence of tactics. Tactics act upon the hypotheses and goals of a proof. As Coq is interactive, the user is presented after each tactic with the new goals and the new hypothesis at hand. Tactics can generate new hypotheses, new goals, apply previously proved theorems, apply a proof technique such as structural induction. For instance, one could write the following theorem to prove a property related to logical propositions `A` and `B`:

```
Theorem example: forall A B: Prop, A → ((A → B) → B).
```

where → is the standard implication. The proof can be started with the tactic `intros A B HA HAB.`, which tells the assistant to introduce the propositions `A` and `B` quantified over, as well as the two hypotheses `A` and `A → B`.

At this point, the new goal to prove is simply `B`, with the hypothesis `HA : A` and `HAB : A → B`. One can move forward by applying the tactic `apply HAB.`, telling the assistant to apply our hypothesis. Since the goal is the conclusion of this implication, the new generated goal is its hypothesis: `A`. And of course, `A` holds as it is exactly our hypothesis `HA`. One can then conclude with the tactic `exact HA.`, and the theorem is proved. One can also prove theorems about programs, as seen in Figure 4. A Coq development then includes both a functional program and proofs about it.

Coq is a trusted software which rigorously checks that each proof has no mistake and can be conducted without forgetting minor details. Since one can write and reason about a program in Coq, it is particularly useful to design a reliable program in Coq. A program in Coq can then be *extracted* to an equivalent program in OCaml or Haskell [21].

2.2 Formal Verification of Compilers

Verifying compilers is a crucial step for obtaining reliable software. It is important to be sure that a program can be compiled without the compilation process introducing any bug or unwanted behaviors. Most verified compilers use proof assistants.

CompCert [20] is a fully verified compiler for the C programming language, written and proved in Coq. CompCert has been a landmark in compiler verification. It formally defines both C and assembly semantics for several architectures (x86, ARM, PowerPC and RiscV), and ensures that the source and compiled program have equivalent behavior. CompCert includes many optimizations, such as function inlining and dead code elimination. The program passes through several intermediate representations, resulting in an elaborate proof of correctness. CompCertSSA [7] is an extension of CompCert that adds an SSA middle-end in the compilation process. SSA (Static Single Assignment) is another intermediate representation where each variable is assigned exactly once, making it particularly convenient for multiple optimizations. Vellvm [29] is a verified formal framework for the intermediate representation of LLVM, a tool to design compilers. Many compilers use LLVM, including ones for Python, Rust, Scala and many others. Vellvm itself is written and proved in Coq. CakeML [18] is a functional language with a verified compiler backend. It includes a verified type system, and a verified garbage collector. CakeML is written and proved in HOL, but has the ability of boot-strapping itself, meaning that compiling CakeML with itself produces a verified machine-code implementation of the compiler.

To prove its correctness, CompCert defines C, assembly, and every intermediate language semantics with operational semantics. For each language, there is an inductive predicate `step` that represents the transition between configurations. To prove that the target program is equivalent to the source program, one could try to prove a *bisimulation*. A bisimulation is a relation between configurations of the source and target semantics such that each step in the source semantics is matched to a step in the target semantics. However, this definition is too strong for many simple optimizations. Instead, CompCert proves a weaker notion, but adapted to compiler verification: a *backward simulation*. Informally, it means that every time the target program takes a step, the source program also takes a step to a related state. With a backward simulation, we can claim that the compiled program behavior is a refinement of the source behavior. Indeed, the source language of CompCert has a non-deterministic semantics, that is faithful to the C standard (for instance, the evaluation order of expressions is not specified by the standard, each order is a valid behavior). Then, one can define in a symmetric way *forward simulations*: each time the source makes a step, the target makes a matching one (in practice, the definition also allows more complex cases, where the target makes zero or several steps). Forward simulations are usually easier to prove than backward simulations. Then, CompCert proves that, if the target language is *determinate* (a weaker version of determinism), and there is a forward simulation between a source and a target language, we can build a backward simulation. Finally, the correctness proof consists in proving a forward simulation for each compilation pass (including optimizations), compose them, and use the determinacy of the assembly language to build a backward simulation between the C program and the compiled one. Our correctness result also relies on various notions of simulations. These are more formally described in section 5.2.

Having a formally verified compiler means that a well-designed source program, once compiled, will behave as expected. Moreover, it means that properties proved at the source level still hold at the assembly level.

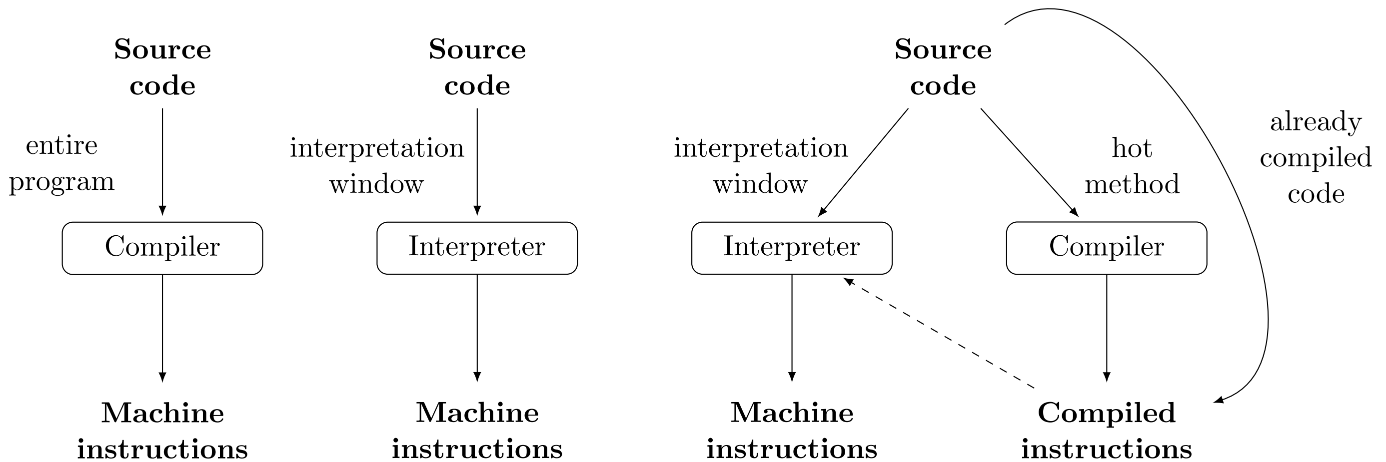

Figure 5: Side-by-side comparison of a compiler, an interpreter and a JIT compiler

3 Just-in-Time Compilers

3.1 Presentation

Compilers and interpreters are always confronted to the same problem: given a source program, translate it into machine-readable code with the same observable behavior. With standard compilers, such translation is done beforehand, and the compiled code can then be run anytime. With interpreters, the translation happens at run-time, usually instruction by instruction. The interpreter reads the next source code line, translates it, runs it then proceeds to the next one.

The main idea behind JIT compilers is to get the best of both worlds. In practice, this means using an interpreter, but compiling on-the-fly loops and frequently run code. Figure 5 shows a comparison of compilers, interpreters and JIT compilers. Note that, except for standard compilers, each translation is done dynamically. In the case of just-in-time, some methods can be sent to a compiler for them to be optimized. Then, upon seeing them again, the compiled code can be directly executed. In some cases like speculative optimization, one needs a way to go back to the interpreter.

In general, optimizations in JIT compilers are often similar to those used in standard static compilers, with two main differences. Firstly, depending on the granularity of the code to dynamically compile, some information might be missing, such as variable scope, meaning that seemingly useless code cannot always be removed. Secondly, using run-time information and assumptions often gives better optimizations. In comparison, static compilers must capture every possible behavior of the program, while compiling in a JIT setting can focus on the most likely path.

Mixing interpretation with run-time compilation had already been introduced in the 60s. In his history of just-in-time [5], Aycock describes the evolution of JIT techniques since then. Initially, the main advantage of interpreted programs was the size of the programs: interpreted programs are written and stored in high-level languages. Nowadays, JIT compilation focuses much more on execution and startup time. As time goes by, a consensus has been made to translate and compile

bigger blocks of code. Two main timeless issues of JIT compilers are profiling, i.e. finding the "hot spots", the parts of the code that might be executed more than the others (and thus would benefit greatly from optimizations with a limited overhead), as well as choosing when and how to optimize code. JIT compilers have reached maturity with Java. Code optimization has been made the norm, and the focus is on striking a balance between speed of optimization algorithm execution and speed of generated code.

3.2 Trace-based Just-in-Time Compilers

A first trend among JIT compilers is trace-based compilers, introduced by Bala *et al* [6]. A trace usually refers to a sequence (finite or not) of program instructions. Rather than deciding to optimize a hot method or function, trace-based JIT compilers first detect portions of source code that are frequently run. One typical example is iterations of a loop. They then record the sequence of instructions (sometimes at a lower level than the source) of an iteration. This sequence can be compiled and optimized for later uses. It is often useful to make assumptions that have been true so far in the previous executions, and use them to optimize the code even more. In that case, guards are added to the recorded trace, and the JIT compiler must provide a *side-exit* (or *bail-out*) mechanism to go back to interpreted code.

Making assumptions about code to optimize is often crucial for performance. For instance, in dynamically typed languages such as JavaScript, many tests are performed at run-time to check if some object has a given type. Indeed, operations such as an addition can be overloaded and behave differently depending on the type of its arguments. Basic interpretation must then perform the same tests for each addition. However, objects do not typically change type across iterations of a loop. If a trace-based compiler sees a loop being executed multiple times with some object `a` being an integer, it can start optimizing the loop iteration assuming that `a` is an integer (and thus removing all tests inside the loop). It then becomes much faster to check the type of `a` only once per iteration, when calling the optimized trace. If the type matches the assumption, then the iteration will be executed with the optimized version. Otherwise, the compiler *bails-out* and the iteration is simply interpreted. This specific example is sometimes called type specialization, but assumptions can also be about values, control-flow or anything that could speed up the optimized trace.

Vandercammen *et al.* [25] formalize trace-based JIT compilers by defining a tracing machine, a finite-state automata that can either record traces, run normal interpretation, optimize traces or execute optimized traces. The tracing machine successfully summarizes the spirit of trace-based JIT compilers, and can be seen in Figure 6. The execution of a program goes through four stages: interpretation and execution of compiled code, as well as recording and optimizing traces. When interpreting a program, if a loop is found and has not been optimized yet, it gets recorded. If it had already been optimized, then the compiled trace can be directly executed instead. On the second iteration of a recorded trace, the tracing machine shifts to the optimization stage, where the trace is optimized and stored for later executions. They insist on decoupling these stages in their formalization, to obtain a more general framework. The method is mostly language-independent and tested on a LISP-like language, but should work on any interpreter implementing their *tracing interface*. For instance, this interface contains the *step* function. This function takes as input the program state, and tells the interpreter to perform a few interpretation steps. The function then returns the updated state, and the sequence of executed instructions (a trace). It is then up to the tracer (the part of the tracing machine in charge of recording traces) to decide whether to

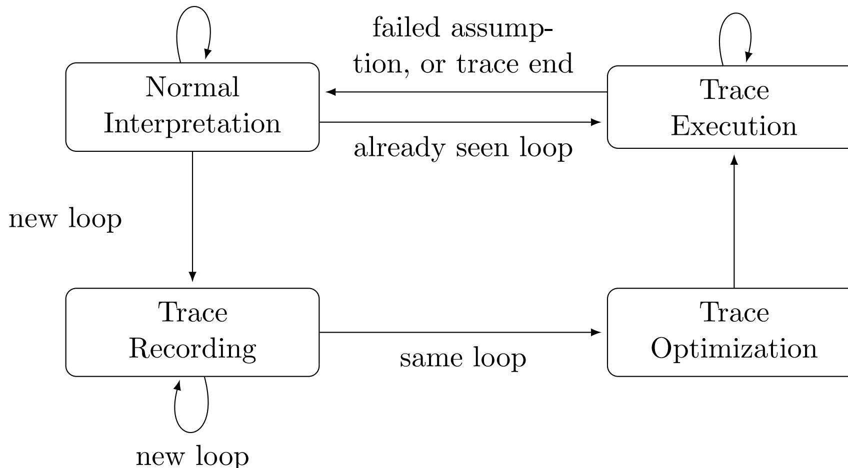

Figure 6: The four stages of a tracing machine, adapted from [25]

record the trace, optimize it once it corresponds to the trace of an entire loop iteration, or run the optimized version if it exists. This interface allows complete modularity of the different parts of the JIT compiler.

Gal *et al.* [13] go as far as generalizing the notion of trace. Their Tracing Monkey JIT compiler for JavaScript is based on an existing interpreter, SpiderMonkey. They focus on loop iteration traces as well, but try to cover more complex control flow. First, they allow exits inside traces: if some exit is frequently taken in a compiled trace, because of a failed assumption for instance, they add a test inside the recorded trace to cover both cases. This way, the compiled versions can cover every hot path of the loop without going back to the interpreter. However, using this definition would lead to poor performance with nested loops. Indeed, a standard trace-based compiler would first consider the inner loop to be hot. The inner iteration trace would be recorded. But as the inner loop terminates, the exit to the outer loop would be recorded as a frequent exit, and the tracer would start creating a branching trace and trace the outer loop inside the trace corresponding to the inner one. If the inner loop always exits the same way, then this would not be an issue. But if it has several exits, then the code of the outer loop would be added at the end of each of these exits. To solve this, they define *trace trees*. When recording the inner loop, going back to the outer loop is not recorded as a branching anymore. Instead, a new trace is recorded, starting at the beginning of the outer loop. When this trace reaches the inner loop once again, it calls the inner loop's trace, then resumes tracing the outer loop.

For instance, consider a program whose control-flow is informally depicted in Figure 7. The inner loop has two different exits. If the trace-based compiler only used branching traces, the trace monitor would record the trace depicted in Figure 8, where the code corresponding to the outer loop is needlessly copied for each exit of the inner loop. With trace trees, the compiler would record the two traces on Figure 9, one for each loop. This approach has proved to be very efficient on SunSpider, an industry standard JavaScript benchmark suite.

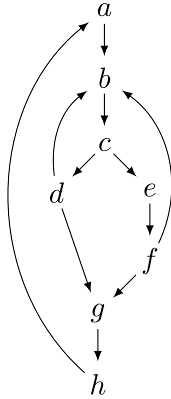

Figure 7: Nested loops

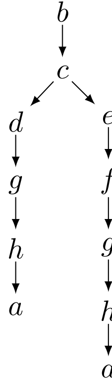

Figure 8: Branching trace

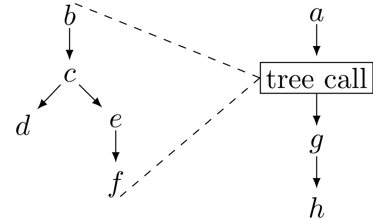

Figure 9: Trace tree

3.3 Partial Evaluation

The idea of mixing compilation and interpretation, for JIT compilation or not, has been in the spotlight for a few years. For instance, Würthinger *et al.* [27] suggest a new approach for dynamic languages, such as JavaScript, Ruby and R. The approach consists in defining the semantics of a language only by implementing a reference interpreter. A compiler can then be derived from specializing the interpreter by means of *partial evaluation*, as in the first Futamura projection [12]. This work is not JIT compilation, but may provide good insight for designing and formalizing JIT compilers today.

In this setting, partial evaluation takes as input a method of the source program, and some profiling information gathered by the interpreter so far. This profiling information can include types or values for some arguments of the method. Some simplifications can then be made. For instance, an if-statement whose condition is known can be simplified to a single branch. Once simplified, we can derive the intermediate representation of the target function corresponding to the input method. This way, we derive an optimized version of some method, which now only requires the remaining values of its arguments to be executed, much like standard JIT compilers create a compiled, optimized version of some methods. The main idea is using the run-time information on the inputs of the method to create a version optimized for a restricted set of inputs that the method is likely to use. As in trace-based compilers, the compilation step can make assumptions on the code to optimize, and a bail-out mechanism to the interpreter is provided. To be as efficient as possible, partial evaluation goes through all methods called from its input method, as some simplifications may occur in these calls too. However, this traversal has to stop at one point to keep the compiled code small enough. This method allows the implementer to mark any method with a boundary annotation, that informs partial evaluation not to simplify it. The authors highlight the difficulty of automatically finding good partial evaluation boundaries.

3.4 A Real Just-in-Time Compiler: Graal

The Graal project extends the Java Virtual Machine to many other languages, such as JavaScript, Python, C, C++ and many others. It aims at defining a virtual machine for many languages, and thus includes the Truffle framework, to define languages for Graal. Languages are defined and implemented by writing an interpreter with Truffle. Finally, the Graal Compiler [24] is designed to

Figure 10: From Java source code to machine code

Figure 11: The sea-of-nodes graph for `-(-x)+y` Figure 12: The optimized sea-of-nodes graph

compile and run programs written on Truffle-based languages. This approach is mostly language independent. The compiler itself is written in Java.

Recall the standard way to execute a Java program, as shown on Figure 10. The source code is first compiled to bytecode using a static compiler such as `javac`. Then, the JVM implementation has to translate the bytecode to machine code. This translation used to be interpretation, but more recent JVM implementations, such as OpenJ9 or Solaris JVM, decide to use a JIT compiler (from bytecode to machine code) to compile some hot methods. In that case, the compiler is not in charge of finding hot spots and choosing between interpretation and compilation (the JVM implementation is), but simply acts as a compiler from bytecode to machine code, with access to dynamic information collected by the interpreter of the JVM. This approach was used by the OpenJDK implementation of the JVM, with two JIT compilers being called. The Graal compiler replaces them, relying on the JVMCI (JVM compiler interface), which enables a compiler written in Java to be used as a dynamic compiler for the JVM. Graal is written in Java, and now provides a more universal compiler, for any languages running on the Graal virtual machine. It takes as input some bytecode, along with additional information from the interpreter. It first integrates a graph builder. The bytecode is represented as a sea-of-node graph (as introduced by Click and Cooper [8]), which includes both data-flow information and the order of execution of each method. Optimizations are then performed directly on this graph. They include redundant code removal or lock elimination. Figure 11 shows the graph of a simple program returning `-(-x) + y`. The dashed line represents the order of method calls (meaning that `getX()` should be called before `getY()`). Figure 12 shows the optimized graph. Finally, the resulting graph is transformed into machine code by simply generating code for each node in the order specified by the graph.

3.5 Formalizing Just-in-Time Compilers

Even if some ideas have been introduced since the 60s, JIT compilers have been of common use fairly recently compared to standard compilers or interpreters. Nevertheless, some works have already approached formal verification of JIT compilers.

3.5.1 Formalizing Trace-Based Just-in-Time compilers

In particular, Guo and Palsberg [16] focus on the soundness of trace optimizations in trace-based JIT compilers. Sound trace optimizations may differ from standard ones. Indeed, when optimizing a trace, the rest of the program is not known to the optimizer. As such, optimizations such as dead store elimination are unsound when optimizing a trace: a store might seem useless in the trace itself, but actually impacts the semantics of the rest of the program. Similarly, new optimizations can be done in this context: free variables of the trace can be considered constant for the entire trace. To characterize sound and unsound optimizations, the authors define formal semantics for trace recording and bail-out mechanism for speculative optimization. Finally, they define a bisimulation-based criterion for sound optimization. Checking for soundness then reduces to checking that the criterion is satisfied. Then, a theorem proves that using such sound optimizations, program semantics are observationally equivalent whether tracing is performed or not.

3.5.2 Formalizing deoptimization

When compiling dynamic languages, speculative optimizations seem necessary for performance. But there might come an execution where the assumptions do not hold. In that case, the generated code must be discarded and we must fall back to a less optimized and more general version of the code. This might even happen during the execution of the version compiled with wrong assumptions. This process is called deoptimization, and generalizes the side-exits of trace-based JIT compilers, which simply fall back on the interpreter. Deoptimization is a special case of *on-stack-replacement*, where some function must be replaced dynamically, during its execution. This raises many technical issues, especially in the case of deoptimizations, where the wrong version might have taken out some parts of the program that should not have been. For instance, if an internal call to a function has been inlined, then when deoptimizing a new stack frame corresponding to this function needs to be created. Flückiger *et al.* investigate dynamic deoptimization and provide formal tools for speculative optimization [11]. In particular, representing the assumptions directly in an intermediate representation with a new instruction `assume` (as in our example Figure 3) can be useful for both checking the validity of assumptions before executing the optimized version, and contains enough information to correctly deoptimize if the assumptions are not valid.

They design Sourir, an intermediate representation inspired by RIR, an IR for the R language. Sourir adds an assume instruction for speculative optimization. This instruction contains a condition (the assumption) and a deoptimization target. When executing it, if the condition evaluates to true, the program proceeds to the next instruction. Otherwise, it deoptimizes to the deoptimization target, a more general version of the same function. This is more than just a branch for two reasons. First, the deoptimization target is more than just a label. It contains information to synthesize missing stackframes (in case of a deoptimization from an inlined function) and update the registers (in case some assignments had been removed). Second, this deoptimization target is invisible to the optimizer. This means that an analysis can be local to the version being optimized. This helps keep optimization time as small as possible.

In this setting, Flückiger *et al.* prove the correctness of various optimizations, some standard (constant propagation, unreachable code elimination, function inlining) and some specific to speculation, like moving or composing assume instructions. The correctness proof relies on invariants about different versions of the same functions: all versions should be observationally equivalent, and adding assumptions to a version should not alter its semantics. The equivalence is maintained by means of weak bisimulations (see section 5.2).

This intermediate representation seems well adapted to a JIT setting. Speculative optimization is possible and can be reasoned on. An interpreter for sourir has been implemented, but this is still far from being an actual JIT compiler. This work does not handle profiling and creating new versions, but simply addresses the optimization of programs with multiple versions and speculations already inserted. The proof techniques themselves seem to require a lot of work, compared to the forward simulations of CompCert.

3.5.3 A verified Just-in-Time compiler for x86

Myreen presents a fully verified JIT compiler to x86 [22]. Its input language is a small stack-based bytecode, with instructions such as `push i, pop` to handle the stack and conditional jumps. The language is given an operational semantics $\xrightarrow{next}$. Semantic states are tuples (xs, l, p, cs) where xs is the data stack, l the available space on that stack, p is a program counter and cs the bytecode program. Treating the program counter as a separate register simplifies the reasoning about pointers and jumps. One example of semantics rule is the `swap` rule. If `fetch p cs = some swap` (meaning that the instruction at program counter p in the program cs is a swap, the instruction that switches the two top elements of the stack), then we have the following transitions:

$$\forall x, y, xs, l, \quad (x :: y :: xs, l, p, cs) \xrightarrow{next} (y :: x :: xs, l, p+1, cs)$$

Myreen points out that one of the main challenges of verifying a JIT compiler is dealing with code as data. Indeed, in standard compilers, the code can be considered as living in a disjoint memory space than the program data, and will not interfere. But in this case, some code is produced dynamically as the output of the compiler and has to be put in the memory, and accessed whenever needed. Thus, he defines operational semantics $\xrightarrow{x86}$ for self-modifying x86 code. The memory is modeled by a function m, and he also models an instruction cache i. Instructions to execute are fetched in i, data is stored and read in m, and i can be updated with values of m. In this semantics $\xrightarrow{x86}$, the execution of the code in the cache can only affect m, not i itself. The cache is only changed with a dedicated transition for cache updating.

For verification purposes, a Hoare Logic is defined to reason about these semantics. This logic includes a separating operator to describe the memory contents. It is not the standard separation operator of separation logic. The author makes a few design choices that allow the frame rule to be applied on both code and data. Hoare Triples are defined as statements about the x86 semantics. Each triple has to be proved using this semantics. Informally $\{p\}c\{q\}$ means that, starting from a state where p is true on some part of the memory, and the code of c is correctly represented separately in the instruction cache, any possible execution sequence will reach a state where q is true and the code of c is still correctly represented in the cache. These Hoare triples are a more convenient formalism than simply using another separation operator. Indeed, the code c in $\{p\}c\{q\}$ is treated as an invariant: it is the code in the instruction cache, that cannot be modified by its

execution. When executing instructions of the instruction cache, m changes and i stays the same. In $\{p\}c\{q\}$, p and q represent the changes to m, while c represents the content of i.

Take for instance the instruction `jmp eax`, an x86 instruction to jump to the value stored in register `eax`. This instruction is encoded as `FFE0`. One could prove the following Hoare triple:

$$\{eax\ v\ *\ pc\ p\}\ \texttt{FFE0}\ \{eax\ v\ *\ pc\ v\}$$

In the precondition, we assume that register eax contains value v and the program counter pc some value p. If the code `FFE0` is correctly represented in the instruction cache, then after executing it, pc will also hold the value v. The code in the cache has not changed, but the program counter has.

The correctness of the JIT compiler is proved with the help of an invariant, *jit_inv*, which relates a state of the bytecode semantics with an equivalent state of the x86 semantics. $jit_inv\ s\ a$ is a separation logic predicate which means that an x86 semantic state, corresponding to the bytecode semantic state s, is correctly represented in the memory at address a. He proves that this invariant is preserved by the semantic step of the bytecode. More formally,

$$\forall s, t, a, \quad s \xrightarrow{next} t \implies \{jit_inv\ s\ a\}\ \emptyset\ \{jit_inv\ t\ a\}$$

This allows to prove that any execution of the bytecode is computed by the x86 code hidden in *jit_inv*. As the code invariant is $\emptyset$, the theorem holds for any content of the cache.

To ensure that the JIT produces x86 code that complies with the invariant, he uses a previously developed synthesis tool for proof-producing compilation [23]. This proof producing compilation tool takes as input a HOL function, and generates machine code, along with a certificate HOL theorem that proves that the code correctly implements the function. As the input is a function in the native language of HOL, one can easily prove in the proof assistant that the function is correct, and the generated code will be correct too. In this case, the JIT compiler is written inside HOL, where it can be proved that it correctly generates x86 code. Finally, using the proof-producing compilation tool, we get an x86 JIT compiler, reading bytecode and producing x86 code. This compiler was tested on real bytecode programs. When the bytecode is already optimized, this approach performs as well as a compiled C version with `gcc -O3`, but this JIT does not perform any optimization itself.

4 A JIT in Coq

Our work consists in implementing and verifying in Coq a JIT compiler. Projects such as CompCert have proved that formally verified realistic compilers are feasible. Optimizing compilers are such complex software that a formal proof offers a guarantee that no kind of testing can. For instance, Yang *et al.* developed a tool to find bugs in C compilers [28]. While all other tested compilers (LLVM, GCC, CIL, TCC, Open64) had middle-end bugs and despite 6 CPU-years devoted to finding some, the tool did not find any in CompCert, whose middle-end is formally verified.

JIT compilation is being used increasingly, for many languages. As complex software, JIT compilers implementations are error-prone, and their correctness cannot be taken lightly. However, compilation at runtime requires different proof techniques than static compilation. Our objective is to show how these can be adapted to fit the model of JIT compilation. In general, ahead-of-time compilers can simply apply the same optimization passes to every program. But in just-in-time compilation, the functions to optimize and the optimizations themselves are different depending on the current execution. For verification purposes, this means that we cannot compare formally the input program and some compiled program: the program evolves dynamically.

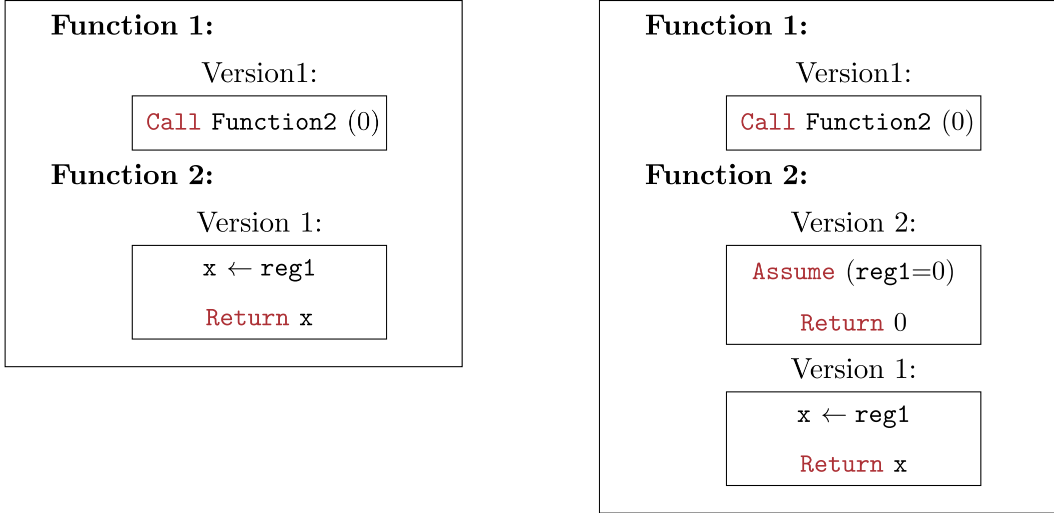

Figure 13: A simplified view of an input JIT program on the left, without assumptions and with one version per function. During the JIT execution, it can become the program on the right, where assumptions have been inserted, and new versions created.

4.1 The JIT IR language

To avoid being overwhelmed by the size of the semantics of realistic languages, we chose to design our own input and target language. This allows us to focus on the characteristic features of JIT compilation. The proofs are not tied to a particular implementation or language, and could be adapted. While simple, our language is still close from other intermediate representations. In particular, it is reminiscent of Sourir [11] and RTL. RTL (*register transfer language*), is a standard three-address code CFG (control-flow graph) intermediate representation, used in many compilers such as GCC or CompCert.

We decided to use the same language for both the input programs and the compiled functions, with two exceptions. The difference can be seen in Figure 13. Firstly, as the program gets progressively compiled and optimized, new versions of functions are added. Initially, an input program has only one version per function. During the JIT execution, its internal program will start as an input program, then some functions will receive new versions when optimizing. Multiple versions for one function with different assumptions and different optimizations can coexist. This allows us to have some heavily optimized versions that rely on assumptions that might not always be correct, and still have some less specific versions in case the assumptions do not hold. Secondly, as in [11], our language includes an `Assume` instruction (to check assumptions and deoptimize if needed). An input program should not have any of these instructions. However, any optimization using speculation should insert them in the target program with the information needed to deoptimize.

One usually expects JIT compilers to produce native code directly, and yet we compile to the same language. Having only one language allows simpler definitions in our Coq development. For instance, there is no fundamental difference between the interpretation of original code and the execution of optimized code: they both use the same interpreter. However, no correction result relies on the two languages being the same. If one wanted to compile to a different language, one would simply need a new interpreter for this language (and its correctness proof), so that the JIT

can execute compiled code.

Our `Assume` instruction is directly inspired from Sourir. Other JIT compilers may proceed differently. For instance, D'Elia and Demetrescu describe a framework for on-stack replacement [9] where speculative optimization can be implemented with *OSR conditions*. When falling back to another version, some *compensation code* is executed, to adapt the current state to the new version (restoring allocations that had been removed for instance). With the `Assume` instruction from Sourir, this adaptation is made possible with *deoptimization metadata*, which syntactically contains the deoptimization target (where to deoptimize to), the stackframes to synthesize, and a map to update the registers states.

Other JIT compilers prefer a graph-based intermediate representation with SSA, as in Graal [10]. However, SSA semantics are not trivial to formalize in Coq [7] and would have hindered our correctness proofs without being a characteristic feature of JIT compilation.

4.1.1 Syntax

Our language, JIT IR, is a CFG (control-flow graph) intermediate representation, as seen in many compilers. The complete syntax can be seen Figure 14. A program is a list of functions. Functions are composed of versions. A version contains several instructions, each associated with a label. Each instruction contains the labels of its successors.

As in Sourir, the `Assume` instruction is used to check a list of speculative assumptions. It also includes the identifier of the version to deoptimize to, if the assumptions do not hold. One can also include more deoptimization information, such as additional mappings that might be needed, or stackframes to synthesize (useful for optimizations such as function inlining). For instance, writing `Assume` (`x`=0, `y`=0) (`Fun1`,`Ver3`,`lbl12`) {(`z`,2) (`x`,0)} [] `next` informally means that one should evaluate the list of expressions (`x`=0,`y`=0). If all tests hold, move to the instruction labeled `next`. Otherwise, move to the deoptimization target (`Fun1`,`Ver3`,`lbl12`), *i.e.* the function whose identifier is `Fun1`, in version `Ver3`, to label `lbl12`. In that case, one must update the register states with the content of the *varmap* {(`z`,2) (`x`,0)}: register `z` receives value 2 and register `x` receives value 0.

As in CompCert RTL [19], the instructions operate on abstract *registers* (with no bounds on the number of registers). Instructions include basic operations on registers, memory store and load, printing values and function calls. The expressions used in the syntax are non-recursive. Such design choices are typical of compiler backend intermediate representations.

An example of JIT IR program can be seen Figure 15. For readability purposes, the syntax has been slightly modified. For instance, we only print the labels when they are referred to in an instruction. This program computes and prints the first 20 values of the factorial function. `Fun2` takes one argument (`reg1`) and computes its factorial recursively. `Fun1`, the main function, has a loop with the register `reg1` increasing. At each iteration, `Fun2` is called and the result is printed.

4.1.2 Small-step Semantics

Our language is given formal small-step semantics, in Coq. Small-step semantics allow us to reason locally about one computation of the JIT. This will prove useful as our JIT interleaves steps of the interpreter (following steps of the semantics) and optimizations.

States of the semantics should include enough information to execute the current instruction. States (s, v, pc, rm, ms) contain a stack, the current version, the current label, the state of the registers (a map from registers to values) and the state of the memory (a map from addresses to

op	::=		Operands
	|	*reg*	Registers
	|	*value*	Values
expr	::=		Expressions
	|	**Binexpr** *binop op op*	Binary expressions
	|	**Unexpr** *unop op*	Unary Expressions
instr	::=		Instructions
	|	**Nop** *label*	Moves to next label
	|	*reg* ← *expr label*	Assignment
	|	*reg* ← **Call** *fun_id expr* label*	Call function with arguments
	|	**Return** *expr*	Return value
	|	**Cond** *expr label label*	Branching statement
	|	Mem[*expr*] ← *expr label*	Store in memory
	|	*reg* ← Mem[*expr*] *label*	Load from memory
	|	**Print** *expr*	Print value of expression
	|	**Assume** *expr* target synth* label*	Assume
binop	::=	**Plus** | **Minus** | **Mult** | **Gt** | **Lt** | **Eq**	Binary operations
unop	::=	**UMinus** | **Neg**	Unary operations
target	::=	*fun_id ver_id label varmap*	Deoptimizing target and varmap
synth	::=	*fun_id ver_id label reg varmap*	Synthesized Stackframe
code	::=	*(label instr)**	Mapping from labels to instructions
version	::=	*ver_id code label*	Identifier, code and entry label
function	::=	*fun_id reg* version* ver_id*	Parameters, versions and current version
program	::=	*fun_id function**	Main function identifier and functions

Figure 14: The JIT IR syntax

```
[Main: Fun1]
Function Fun1:
       reg1 ← 0
<lbl2> reg2 ← Call Fun2 (reg1)
       Print reg2
       reg1 ← Plus reg1 1
       Cond (Gt reg1 20) lbl6 lbl2
<lbl6> Return 0
```

```
Function Fun2:
Parameters: (reg1)
       Cond reg1 lbl3 lbl2
<lbl2> Return 1
<lbl3> reg2 ← Minus reg1 1
       reg2 ← Call Fun2 (reg2)
       reg2 ← Mult reg1 reg2
       Return reg2
```

Figure 15: An example of JIT IR program that computes the first 20 values of factorial.

$$\text{Nop}\ \frac{v \,!\, pc \;=\; \texttt{Nop}\ next}{(s, v, pc, rm, ms) \to (s, v, next, rm, ms)}$$

$$\text{Op}\ \frac{v \,!\, pc \;=\; reg \leftarrow\ expr\ next \qquad (expr, rm) \downarrow\ val}{(s, v, pc, rm, ms) \to (s, v, next, rm\#reg \leftarrow val, ms)}$$

$$\text{Cond True}\ \frac{v \,!\, pc \;=\; \texttt{Cond}\ expr\ iftrue\ iffalse \qquad (expr, rm) \downarrow\ true}{(s, v, pc, rm, ms) \to (s, v, iftrue, rm, ms)}$$

$$\text{Cond False}\ \frac{v \,!\, pc \;=\; \texttt{Cond}\ expr\ iftrue\ iffalse \qquad (expr, rm) \downarrow\ false}{(s, v, pc, rm, ms) \to (s, v, iffalse, rm, ms)}$$

$$\text{Call}\ \frac{v \,!\, pc \;=\; \texttt{Call}\ f\ args\ retreg\ next \qquad \texttt{current_version}\ f \;=\; v' \qquad \texttt{init_regs}\ args\ rm\ f \;=\; rm'}{(s, v, pc, rm, ms) \to ((retreg, f, next, rm) :: s, v', entry(v'), rm', ms)}$$

$$\text{Return}\ \frac{v \,!\, pc \;=\; \texttt{Return}\ expr \qquad (expr, rm) \downarrow\ val}{((retreg, v', next, rm') :: s, v, pc, rm, ms) \to (s, v', next, rm'\#retreg \leftarrow val, ms)}$$

$$\text{Return Final}\ \frac{v \,!\, pc \;=\; \texttt{Return}\ expr \qquad (expr, rm) \downarrow\ val}{([], v, pc, rm, ms) \to (Final(val, ms))}$$

$$\text{Print}\ \frac{v \,!\, pc \;=\; \texttt{Print}\ expr\ next \qquad (expr, rm) \downarrow\ val}{(s, v, pc, rm, ms) \xrightarrow{val} (s, v, next, rm, ms)}$$

$$\text{Store}\ \frac{v \,!\, pc \;=\; \texttt{Store}\ expr_1\ expr_2\ next \qquad (expr_1, rm) \downarrow\ val \qquad (expr_2, rm) \downarrow\ addr}{(s, v, pc, rm, ms) \to (s, v, next, rm, ms\#addr \leftarrow val)}$$

$$\text{Load}\ \frac{v \,!\, pc \;=\; \texttt{Load}\ expr\ reg\ next \qquad (expr, rm) \downarrow\ val}{(s, v, pc, rm, ms) \to (s, v, next, rm\#reg \leftarrow val, ms)}$$

$$\text{Assume Holds}\ \frac{v \,!\, pc \;=\; \texttt{Assume}\ le\ (fa, va, la)\ rom\ sl\ next \qquad (le, rm) \Downarrow\ true}{(s, v, pc, rm, ms) \to (s, v, next, rm, ms)}$$

$$\text{Assume Fails}\ \frac{\begin{array}{c} v \,!\, pc \;=\; \texttt{Assume}\ le\ (fa, va, la)\ rom\ sl\ next \qquad (le, rm) \Downarrow\ false \\ \texttt{update_regmap}\ rom\ rm \;=\; rm' \qquad \texttt{synthesize_frame}\ rm\ sl \;=\; synth \end{array}}{(s, v, pc, rm, ms) \to (synth{+}{+}s, va, la, rm', ms)}$$

Figure 16: JIT IR small-step semantics

values). A stack is made out of stackframes (r, v, l, rm) containing the register to hold the result of the call, the calling version to return to after the call, the label to return to, and the registers state to restore. States can also be final, in case the main function has returned, and then only contain the final returned value and the memory state.

```
Inductive state: Type :=      (* Semantic states *)
| State: forall (s:stack)          (* call stack *)
                (v:version)        (* current version being executed *)
                (pc:label)         (* current label *)
                (rm:reg_map)       (* state of the registers *)
                (ms:mem_state),    (* state of the memory *)
                state
| Final: forall (v:value)          (* after returning from function main *)
                (ms:mem_state),
                state.
```

Finally, the small-step semantics is implemented as a predicate. An informal version is depicted in Figure 16. For readability purposes, we use version identifiers instead of versions, but in the actual semantics, the version has to be found in the program. The smallstep relation is thus parametrized by a program. $(v \; ! \; pc)$ represents the code of version v at label pc, and $(rm\#reg \leftarrow val)$ represents the updated register map rm where the register reg has received value val. The semantics relies on the evaluation of expressions, given the state of the registers, omitted in this report. We note $(expr, rm) \downarrow val$ when the expression $expr$ evaluates to the value val under the register state rm. Similarly, lists of expressions can be evaluated as *true* or *false*, noted $(le, rm) \Downarrow b$. `current_version` returns the current version of a function. `entry` returns the entry label of a version. To initialize the register state when calling functions, `init_regs` *args rm f* takes a list of expressions *args*, evaluates them under *rm*, and creates a new register states where each function argument of *f* receives its corresponding value. When deoptimizing, we might need to update the register state with the mapping of the instruction. This is done with function `update_regmap`. Finally, `synthesize_frame` creates the stackframes to synthesize during deoptimization.

We implement this semantics as a predicate. (`step p s1 e s2`) means that in program `p`, the semantic state `s1` can step to state `s2` with event `e` (the printed value, or `E0` for a silent step). The program is used as a parameter of the small-step semantics and is used for calls and assumes, where the current version might change. The `step` predicate consists of the semantic rules. For instance, the rule that evaluates an operation assignment `reg ← expr next` is:

```
| exec_Op:
    forall p s f pc rm ms expr reg next v,
      (ver_code f)!pc = Some (Op expr reg next) → eval_expr expr rm v →
      step p (State s f pc rm ms) E0 (State s f next (rm # reg ← v) ms)
```

In that case, if the current instruction is `reg ← expr next`, then `expr` is evaluated, and in the next state the register mapping has been updated with the new value for `reg`.

Our language JIT IR is deterministic. A deterministic target language allows for forward reasoning when proving optimization correctness (see section 5.2), and JIT IR is also our target language. If we had a different language for the input and optimized code, the input semantics could be non-deterministic.

We defined an interpreter that the JIT uses for executing code. For a given program and state, it computes the next semantic state.

```
(* Interpreter step *)
Definition step (p:program) (s:state): res (state * event) :=
```

Given a program and a semantic state, it returns the next state and the event produced by the step (for instance, an output). It finds the rule of the semantics that can be applied, if it exists, and applies it to the current state. `res` is a monadic encoding of errors: either the interpreter returns the next state and event, or it returns an error with a message. Examples of errors can be trying to call a function that does not exist or accessing unassigned memory addresses.

We proved its correctness with respect to the small-step semantics:

```
Theorem interpreter_sound:
  forall p s1 s2 e, interpreter.step p s1 = OK(s2,e) → ir.step p s1 e s2.

Theorem interpreter_progress:
  forall p s1 e s2, ir.step p s1 e s2 → exists e', exists s2', interpreter.step p s1 = OK(s2',e').
```

The first theorem states that if the interpreter computes with no error the next state and the event produced, then this step is indeed a step of the small-step semantics. The second theorem states that if a program has non-blocking semantics in a state (*i.e.* there exists a step from this state), then the interpreter can compute a next state too. Note that in our case, the JIT IR semantics is deterministic. One can prove that this step taken by the interpreter is exactly the same as the one of the semantics (same event and next state). We could thus prove that `interpreter.step p s1 = OK(s2,e) ↔ ir.step p s1 e s2`. However, we will see in section 6 that we might want to add non-determinism in the semantics. As the interpreter only returns one of the possible steps, these two theorems guarantee that if a program has one or more behaviors in the semantics, then using the interpreter will return one of these behaviors.

4.2 The Coq JIT compiler

Our JIT compiler contains several components.

- A parser, to get a JIT IR program from a text file.
- An interpreter, to execute the program (both input code and optimized).
- A profiler, which gathers runtime information at each step. It can suggest optimizations to the JIT (for instance, once it detects a hot method). The profiler takes as input the state of the interpreter (the current configuration of the execution), and updates some profiler state. In our implementation for instance, the profiler state includes a counter for each function. When the interpreter state shows that some function has been called, the profiler increases its corresponding counter. This helps to track the hot functions. We also record the values of function arguments for speculative optimizations.
- An optimizer, including several optimizations. When the JIT decides to optimize, it replaces its own program with an optimized version. The optimizations use information from the profiler.
- Finally, a function which loops the `jit_step` function until a final state is reached, and prints the outputs.

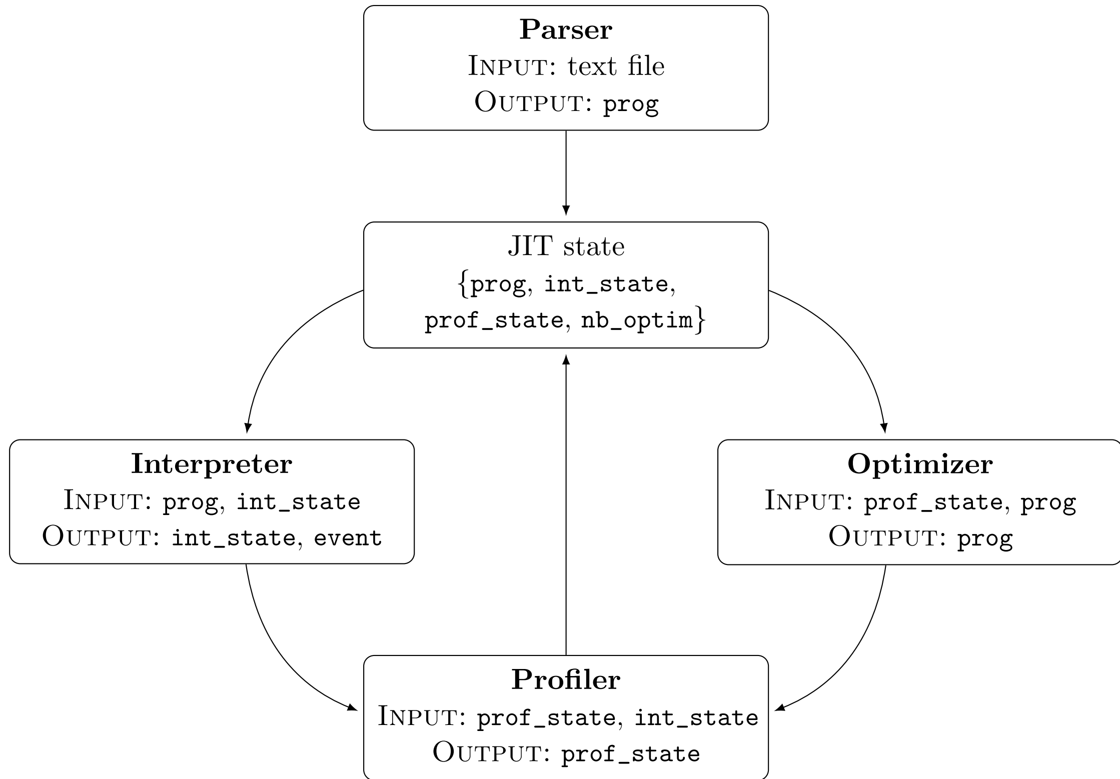

Figure 17: Modifying the JIT state through all components of the JIT.

The `jit_step` function operates on a `jit_state`, and returns the modified `jit_state` and the produced event. This `jit_state` type contains the current program (initially, the input program, but it gets modified through the execution), an interpreter state (same as a semantic state), a profiler state (defined in the profiler), and a counter.

```
Record jit_state: Type := mk_jitstate {
  prog: program;
  int_state: state;
  prof_state: profiler_state;
  nb_optim: nat
}.
```

In order to limit the number of optimizations, this counter decreases each time an optimization is performed. When it reaches 0, the JIT cannot make any more optimizations, even when the profiler tells it to. As the profiler is external and cannot be trusted (see section 4.3), if the JIT was to follow its recommendations, it might get stuck if the profiler wants to optimize at each step. Such a counter ensures that the JIT will eventually stop optimizing and execute its program, thus making progress in the execution. The different parts of the JIT state are modified as depicted in Figure 17.

Finally, the `jit_step` function is the following:

```
(* The JIT step function to be looped *)
(* Returns the next state and the event processed during the step *)
Definition jit_step (js:jit_state): res (jit_state * event) :=
  match next_status (prof_state js) (nb_optim js) with
  | Exe ⇒     (* Executing *)
      do (news,e) ← interpreter.step (prog js) (int_state js);
      do newps ← OK (profiler (prof_state js) news);
      OK (mk_jitstate news newps (prog js) (nb_optim js), e)
  | Opt ⇒     (* Optimizing *)
      do newprog ← OK(optimizer.safe_optimize (prof_state js) (prog js));
      do newps ← OK(profiler (prof_state js) (int_state js));
      OK(mk_jitstate (int_state js) newps newprog ((nb_optim js)  1), E0)
  end.
```

The `do x ← y; z` notation allows us to chain computations and return the first error message encountered in case something was to fail. `OK` is used every time the computation does not fail and cannot return an error. Errors can only happen during interpretation, as explained below the JIT should not stop unless a runtime error is encountered. It first chooses between execution of its current program or optimization. This choice (`next_status`) is determined by the profiler and the number of optimizations left (when this reaches 0, the JIT performs execution regardless of what the profiler says). When the JIT does an executing step, it changes its interpreter state by calling the interpreter on its current program. When the JIT does an optimizing step, it updates its program using the optimizer. In both cases, the profiler is called and updates the profiler state. The `jit_step` function returns an event. In case of optimization, this is a silent event, as no output has been produced. In the case of execution, the event is the one returned by the interpreter.

This JIT compiler is not a tracing JIT. It optimizes functions by creating new versions of them, and does not record traces. For the correctness proof, this has the advantage of not having to trust the profiler at all. Indeed, in our case the profiler simply sends to the JIT compiler the identifier of the function to optimize, and the speculation. Using an incorrect profiler can result in a slower execution (for example if it optimizes functions that are not hot, or with speculations that do not happen frequently), but the behavior of the JIT will still match the behavior of the source program. In a tracing JIT, one would need to make sure that the trace recorder does not introduce any mistake. However, Flückiger *et al.* argue that inserting `Assume` instructions speculating on branching targets can simulate a tracing JIT [11]. By that they mean that, with the right optimizations, one can derive optimized versions corresponding to optimized traces.

4.2.1 Optimizations

So far, two optimizations have been implemented: constant propagation, and speculative calls. At each function call, our profiler records the number of calls so far, and the values of function arguments. After reaching some threshold, the function is considered hot. The profiler sends this information to the JIT compiler. If the JIT decides to optimize the function, it first looks at the recorded values.

If in the last calls, some argument has had the same value, the **speculative calls** optimization creates a new version by inserting an `Assume` instruction at the beginning of the current version. The assumptions are a list of equalities between the parameters and the recorded values. The target of the `Assume` is the entry of the previous version. This is an example of speculative optimization:

the next time the function will be called, if the values are the same again, the `Assume` will hold. So far the rest of the version is exactly the same as before, but making the assumptions explicit allows the next optimization pass to fully exploit the speculation.

Then, the **constant propagation** optimization performs standard constant propagation on this new version. In particular, it can infer from the `Assume` some information about function arguments. Constant propagation can be implemented using the Kildall Algorithm [17, 15].

For instance, if some function has been repeatedly called with 0 as an argument for parameter r, once it gets optimized it will have a new version with `Assume (Eq r 0)`. Then, in this new version, every use of r can be replaced by 0 (unless r is re-assigned). For the next calls, either r is still equal to 0: in that case, the new version (specialized for this value) will be run. Or its value has changed: in that case, the `Assume` fails and the JIT deoptimizes to the previous version, without assumption.

In the case of static compilers, the compilation success is not a matter of correctness, but of implementation. This means that a compiler failing to produce any output (a compiler that always crashes for instance) is correct as the correction theorem only says something about the compiled program if it exists. In the case of runtime compilation though, even if some compilation passes were to fail and optimized code was not produced, the JIT itself should not stop the execution of the program, as it also acts as an interpreter. To this end, if some optimization fails, our JIT optimizer (`optimizer.safe_optimize`) goes back to interpretation of the previous program (without new versions). This ensures that progress is preserved by the JIT compiler: if the original program is safe, then so is its JIT execution.

4.2.2 Running the JIT

Consider the following program:

```
[Main: Fun1]
Function Fun1:
        reg1 ← 0
<lbl2>  reg2 ← Call Fun2 (reg1, 0)
        Print reg2
        reg1 ← Plus reg1 1
        Cond (Gt reg1 5) lbl6 lbl2
<lbl6>  reg1 ← Call Fun2 (reg1, 100)
        Print reg1
        Return 0
```

```
Function Fun2:
Parameters: (reg1, reg2)
Version Ver1:
<lbl1> Return (Plus reg1 reg2)
```

In this program, `Fun2` simply computes the sum of its two arguments, while `Fun1` calls `Fun2` for increasing values of `reg1` with 0 as a second argument. When the loop finishes, it calls `Fun2` one last time, with 100 as a second argument.

Running this program through the JIT compiler gives the following output:

```
OUTPUT: 0
OUTPUT: 1
OUTPUT: 2
DEBUG:  Optimizing Fun2
OUTPUT: 3
OUTPUT: 4
OUTPUT: 5
```

```
DEBUG:  Deoptimizing
OUTPUT: 106
```

We added debug messages to our JIT compiler to be able to see when are functions optimized and deoptimized. After three calls to `Fun2`, the JIT compilers tries to optimize it. The first optimization pass, **speculative calls**, creates a new version of the function with an `Assume` instruction. Since the profiler has recorded that, for all previous calls, the second argument was 0, it creates the following version:

```
Version Ver2:
[Entry: lbl2]
<lbl2> Assume (Eq reg2 0)  Fun2.Ver1.lbl1 {} [] lbl1
<lbl1> Return Plus reg1 reg2
```

When reading the `Assume` instruction, the interpreter will test if `reg2` is indeed 0. If it is, it proceeds to `lbl1`. Otherwise, it deoptimizes to `Fun2.Ver1.lbl1`, the entry of the original version. In that case, there is no stackframe to synthesize ([]) and no update to the register states to do ({}).

Then, the second optimization pass, **constant propagation**, optimizes this new version. As the `Return` instruction of this version is only reachable after successfully passing the `Assume`, it deduces that `reg2` has to be equal to 0. It then creates the new version:

```
Version Ver3:
[Entry: lbl2]
<lbl2> Assume (Eq reg2 0)  Fun2.Ver1.lbl1 {} [] lbl1
<lbl1> Return reg1
```

In the final program, `Fun2` has 3 versions, and the current version is `Ver3`. In the last call to `Fun2`, the second argument is not 0. The `Assume` instruction fails, and the JIT successfully deoptimizes to the original version `Ver1`.

4.3 Implementation

Most of the JIT compiler has been written in Coq, and is then extracted to OCaml to get an executable version. Using this approach, it is possible to not write every part of the JIT in Coq. It is possible to leave some functions as *external parameters*, and then implement them in OCaml (see Figure 18). For instance, we can assume that there exists some profiling function which takes as input some profiler state and the current interpreter state, and returns the updated profiler state.

```
Parameter profiler: profiler_state → state → profiler_state.
```

The type `profiler_state` itself can be an external parameter:

```
Parameter profiler_state: Type.
```

When writing proofs in Coq, we cannot assume anything about these parameters, except their type. We could write a validator, a Coq function that checks if a parameter returns a correct result. However, our correctness proof does not need it. A proof of correctness of the JIT compiler then means that as long as we provide a `profiler_state` type and a well-typed `profiler` function, the JIT compiler is correct.

The case of the parser is different. Our parser has been implemented in OCaml too, with Menhir, a LR(1) parser generator [4]. This parser first transforms a text file into an AST (abstract syntax tree), which syntax resembles closely the syntax of JIT IR, with a few exceptions (the datatype

to represent values, labels, function identifiers are different for instance). The AST program is then transformed, by an OCaml function, into a JIT IR program. JIT IR does not use OCaml integers to represent values and identifiers. Instead, it uses the `positive` type, implemented in Coq for CompCert. This type has a good computational complexity and many theorems are already proved in its library. The correctness theorems only deal with this JIT IR program. We do not reason about the parsing stage, only the middle-end of our JIT compiler is verified. So far, no bugs have been found during parsing, but we have no formal guarantee. Formally verified parsers are yet another problem, that we do not address here.

Finally, the function that loops the `jit_step` function is written in OCaml. It loops `jit_step` until a final state is reached and prints outputs::

```
(* Looping the jit step *)
let rec jit_main (s:jit_state) =
  match (Jit.jit_step s) with
  | Error message -> raise (RunTimeErr message)
  | OK (nexts, e) ->
     let _ = print_event e in
     match (jit_final_value nexts) with
     | None -> jit_main nexts   (* recursive call *)
     | Some v -> raise (Return (v, Jit.jit_program nexts))
```

This function is not yet verified either. It may not terminate, as we want our JIT to run indefinitely if its input program has a diverging behavior. In conclusion, the interpreter is written and verified in Coq. The optimizer is written and verified in Coq (one optimization has not been proved yet: see section 5.4). The JIT step itself is written in Coq. It uses an external profiler, written in OCaml. For any well-typed profiling function, the JIT is proved correct. The `jit_main` and `print_event` OCaml functions are not verified. A summary can be seen Figure 18.

5 Proving the JIT compiler

Our work includes not only the JIT compiler itself, but also a proof of its correctness. By itself the JIT is still a prototype, and cannot compete yet in terms of performance with bigger projects, with much more optimization passes. Comparing our JIT to others would be difficult to do now, as it only works on our own language. However, its novelty is its correctness proof. As our JIT design is typical of modern JIT compilers, it shows that verified realistic JIT compilation is feasible.

This section shows how we adapted standard compiler verification techniques (simulation relations) to the JIT setting. The result is a correctness proof for the JIT in Coq. This proof is modular: the interpreter proof does not depend on the optimizer proof for instance, and optimization passes are proved separately. We first remind standard verification techniques for static compilers, then show our version of JIT simulations.

5.1 Compiler Correctness

5.1.1 Observational semantics

To prove that a compiler is correct, one must prove that the behavior of the compiled program is similar to the original program behavior. This similarity can be expressed through *observational semantics*. The idea is to formally express, on top of a program semantics, the part of its behavior

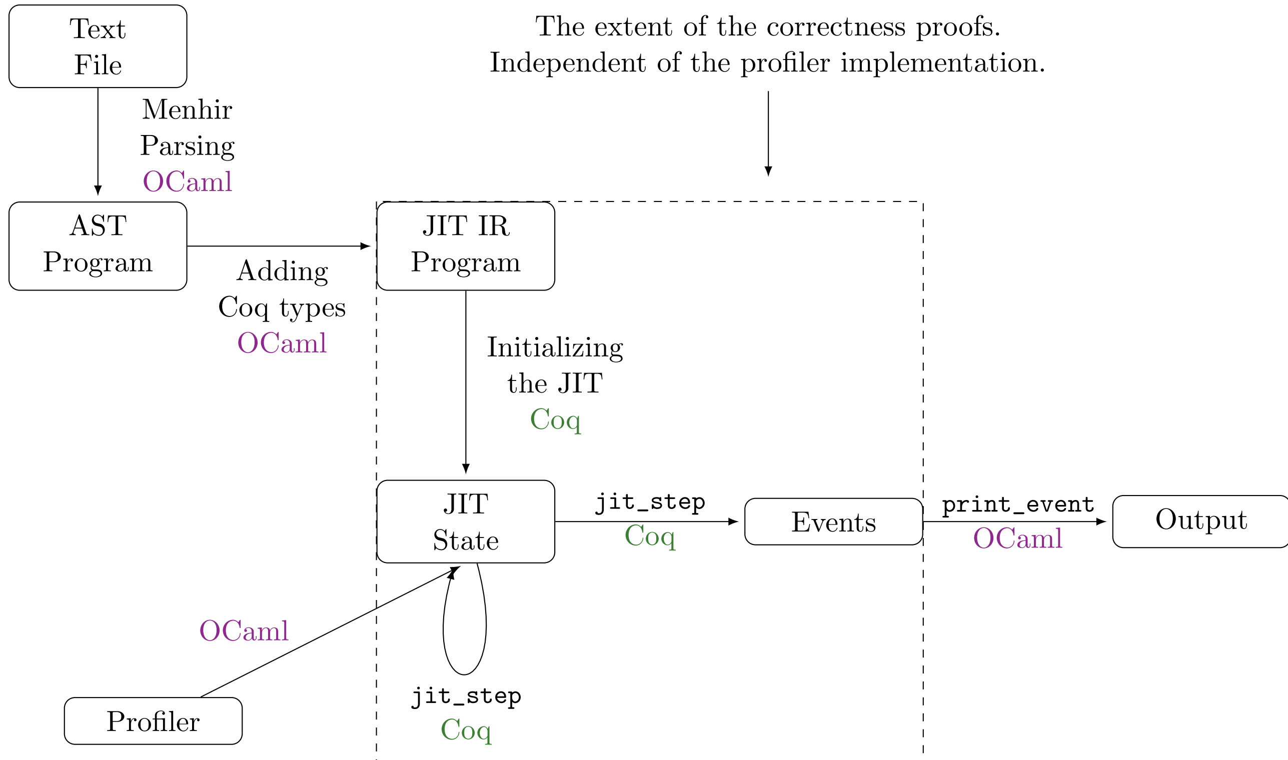

Figure 18: The Coq JIT and its verified middle-end

that should be preserved. Similarly, when executing a program through a JIT compiler, one should observe the same behavior as its semantics.

In our case, we want two equivalent programs to output the exact same values. We thus define an `event` type, which is either nothing (`E0`), or a printed value. `E0` is called a *silent event*, such events are not preserved by the JIT compiler.

```
(* No event or some printed value *)
Inductive event:Type :=
| E0: event
| Valprint: value → event.
```

Remember that our small-step semantics returns events: each step produces an event. In most cases this is a silent event, except when the current instruction is a `Print`.

```
(* Small-step semantics *)
(* [step p s1 e s2] means that in program [p], state [s1] transitions to [s2] with event [e] *)
Inductive step: program → state → event → state → Prop := ...
```

Proving that two programs are *observationally equivalent* amounts to proving that their semantic steps produce the exact same sequence of non-silent events (also called a *trace*). There are different kinds of behaviors: programs can terminate, diverge or go wrong. The first two kinds are defined behaviors, and correspond to finite or infinite traces, while errors correspond to no trace. Two observationally equivalent programs correspond to the same trace, and more precisely,

the compiled program can have strictly more (or strictly less) steps than the original one, as long as these additional (or skipped) steps are silent and the other events are preserved. Observations can be defined with more or less information, to correspond to a stronger or weaker equivalence relation. For instance, CompCert observes which external function calls are made, and volatile global variables.

5.1.2 Behavior refinement

Many semantics are non-deterministic, meaning that a single program can have different behaviors (and thus different possible observations). This is often the case for source languages, where some details are not specified (*e.g.* the order of evaluation of subexpressions in C). Low-level languages such as assembly are more often deterministic. Let us note $\mathit{Beh}(P)$ the set of all observational behaviors of the program P. In our case, a set of traces.

If P_c is the compiled program, one could ask the compiler to guarantee that $\mathit{Beh}(P) = \mathit{Beh}(P_c)$. However, this requirement is often too strong, either because a weaker one would be sufficient, or because this one cannot be proved. For instance, if the target language is deterministic while the input language is not, this equality could not be enforced. In C, a compiler must choose an evaluation order, while several are allowed by the C standard.

For a compiler, a sufficient requirement is $\mathit{Beh}(P_c) \subseteq \mathit{Beh}(P)$, meaning that every behavior of the compiled program was indeed a behavior of the original one. We then say that the compiler *refines* the behaviors of the original program.

This requirement is still too strong in most cases. Indeed, compilers perform *semantics improving*, where errors are removed from the original program during optimizations. For instance, if the instruction `x:=y` appears in the program but `y` is undefined, the original program has no behavior. However, if `x` is never used in the program later, some optimization removing dead allocations could completely remove this instruction from the program. The resulting program could have a behavior, while the original one did not.

In general, the specification of a compiler includes behavior refinement only for program with behaviors. If a program has an undefined behavior, then the compiler is free to produce any output. Noting $\mathit{Safe}(P)$ the fact that a program can always progress without blocking, a more realistic requirement is thus:

$$\textbf{Correction requirement:} \quad \forall P, \mathit{Safe}(P) \Rightarrow \mathit{Beh}(P_c) \subseteq \mathit{Beh}(P)$$

5.2 Simulations for static compilers

To prove that a compiler meets this requirement, many formal works use the notion of *simulations*. To prove that two programs P_1 and P_2 are simulated, one must prove that there exists a *relation* $\mathcal{R} \subseteq \Sigma_1 \times \Sigma_2$, associating semantic states of P_1 with states of P_2, and satisfying some properties, depending on the *simulation scheme* used. We now introduce a few simulation schemes and discuss their uses and advantages.

5.2.1 Backward Lockstep simulation

A backward lockstep simulation between P_1 and P_2 means that each time P_2 takes a step, P_1 takes a matching state. Then,

$$\forall s_1 s_2 s_2' e, \quad s_1 \mathcal{R} s_2 \Rightarrow \mathit{step}\ P_2\ s_2\ e\ s_2' \Rightarrow \quad \exists s_1', \quad \mathit{step}\ P_1\ s_1\ e\ s_1' \wedge s_1' \mathcal{R} s_2'$$

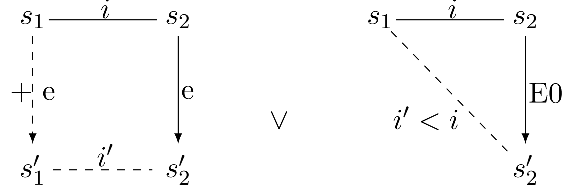

Figure 19: A backward simulation diagram, where solid lines represent hypotheses and dashed lines are conclusions. On the left of each diagram is the source program and its current state s_1, and target program and its current state s_2 on the right. Horizontal lines represent the simulation relation, and vertical lines the semantic steps.

This means that if initial states are matched, and P_2 has some behavior, then P_1 is guaranteed to have this same behavior as well (matching steps must produce the same event, and are thus observationally equivalent). As a result, if there is a backward lockstep simulation between the original program and the compiled program, then the compiler meets the correction requirement.

Unfortunately, many optimizations produce programs that are not lockstep-simulated with the source program. For instance, any optimization adding or removing steps cannot be proved with a lockstep simulation. We must define a weaker simulation scheme in order to prove correct a compiler with optimizations such as dead code elimination.

5.2.2 Backward simulation

For a backward simulation, relations are taken in $\mathcal{I} \times \Sigma_1 \times \Sigma_2$, where $\mathcal{I}$ is a set of indexes over which there exists some well-founded order. Informally, a backward simulation between P_1 and P_2 means that each time P_2 takes a step, either P_1 takes one or more steps to a matching state, or P_1 does not progress, but its current state is matched to the new state of P_2 for a new index that has decreased (for a well-founded order). In the first case, this notion of simulation generalizes the lockstep simulation by allowing P_1 to take several steps. This is especially useful for optimizations that remove steps.

The second case is useful for optimizations that add steps in the compiled program P_2. While these new steps are processed, the source P_1 does not progress. This is called *stuttering*. The order decrease ensures that the compiled program cannot indefinitely stutter without eventually corresponding to some progress in the source program.

A diagram can be seen in Figure 19. In CompCert, this simulation is described as follows:

```
bsim_simulation:
    forall s2 t s2', Step L2 s2 t s2' →
    forall i s1, match_states i s1 s2 → safe L1 s1 →
    exists i', exists s1',
       (Plus L1 s1 t s1' ∨ (Star L1 s1 t s1' ∧ order i' i))
       ∧ match_states i' s1' s2'
```

where `Star` describes an arbitrary number of steps, `Plus` is a strictly positive number of steps and `match_states` corresponds to the simulation relation $\mathcal{R}$. Notice how the `safe` hypothesis allows semantics improving: if the source program has no behavior (and thus is not `safe`), then the compiler can produce any output and the theorem still holds.

Finally, if one can prove a backward simulation between the source program and the compiled

program, then the compiler meets its correction requirement. Backward simulations can be composed: if there exists a backward simulation between P_1 and P_2 and another one between P_2 and P_3, then we can construct a new backward simulation relation (for a new set of indexes and a new order, constructed from those of these two backward simulations), for which there is a backward simulation between P_1 and P_3. This can be useful to chain optimizations or compiler passes: prove a backward simulation for each pass of the compiler, then compose them to deduce a backward simulation between the source and the final result.

However, backward simulations are often hard to prove directly. Indeed, they require reasoning on steps of the compiled program, while most optimizations are defined by matching on the source program. In some optimizations, proving the backward simulation directly would require building a decompilation function to find which source programs could have led to the compiled one.

5.2.3 Forward simulation

To overcome this, one can prove forward simulations. Informally, a forward simulation between P_1 and P_2 means that any step of P_1 will be matched by one or several steps of P_2, or P_2 does not progress but then some well-founded order decreases (just like in backward simulations, but with the roles of P_1 and P_2 reversed). This means that behaviors of the source program are also behaviors of the compiled program. This is often easier to prove than a backward simulation. Forward simulations can be composed too.

However, in the general case, proving a forward simulation is not enough to ensure the correction requirement. Indeed, this does not prevent the compiled program from having multiple behaviors, some of which not matched by any source behavior.

But if the target language is deterministic, then the compiled program can have at most one behavior. Using this intuition, it is proved in CompCert that, for a *determinate* target language (a weaker version of determinism) and a *receptive* source language (progress is independent of the external environment), one can deduce a backward simulation from a forward one:

```
Lemma forward_to_backward_simulation:
  forall L1 L2,
  forward_simulation L1 L2 → receptive L1 → determinate L2 →
  backward_simulation L1 L2.
```

Proving CompCert's correctness consists in proving a forward simulation for each optimization pass [19]. Then, all these simulations are composed into one big forward simulation. Finally, they use the previous theorem to deduce a backward simulation between the source and compiled programs, which ensures that the correction requirement is met.

5.3 Simulations for a JIT compiler

This section details the simulation theorems we have defined for our JIT compiler. They extend the simulation theorems of verified static compilers such as CompCert.

5.3.1 The JIT correctness theorem

In the case of a static compiler like CompCert, one can know in advance what optimizations will be applied: the program always goes through all the passes, always in the same order.

However, in JIT compilation, optimizations are suggested by the profiler at runtime, and different programs will receive different optimizations. In other words, the program of a JIT evolves dynamically when performing optimizations, and there is no way to compare the final JIT program to the source one like in the static case for two reasons. First, this final JIT program is unknown. Proving a simulation in CompCert's case requires two programs. Second, this final JIT program has not been executed all along the JIT execution. At first, the original program was interpreted. Then, after some optimizations, an intermediate program with new versions was being executed. The JIT execution is not simply the execution of the final JIT program, but the execution of all intermediate programs.

To prove our JIT correct, we still want an observational equivalence between the JIT execution of a program, and the program semantics. This too can be proved by a backward simulation: we can prove that there exists some relation between states of the source program semantics and JIT states (`match_states`), such that if the source program is safe (`safe p s`), then any step of the JIT (`jit.jit_step js`) can be matched by zero or several steps of the source program. And in the case of *stuttering* (when the source programs takes no step but the compiled takes one or more steps), some well-founded order strictly decreases (`jit_order ji' ji`). In the case of stuttering, the steps taken by the compiled program must be *silent*, otherwise the traces would not be preserved.

Our theorem is the following:

```
(* Any JIT behavior, if it exists, refines the semantics of the original safe program *)
Theorem jit_simulation:
  forall (p:program) (s:state) (js:jit_state) (ji:jit_index) (e:event) (js':jit_state),
    match_states p s js ji →
    safe p s →
    jit.jit_step js = OK (js', e) →
    (exists s', exists ji',     (* source progress *)
          plus p s (traceof e) s' ∧ match_states p s' js' ji') ∨
    (exists ji',                (* the jit order decreases *)
          match_states p s js' ji' ∧ jit_order ji' ji ∧ silent e).
```

where `p` is the original program. `traceof` builds a trace (a list of output values) out of an event. This definition resembles closely the backward simulation definition of CompCert.

This theorem is not enough in itself. As in CompCert, one must also show that the initial state of the source semantics is matched with the initial JIT state:

```
(* When starting the jit on input program p,
   the jit initial state is related to the initial state of p's semantics *)
Theorem init_jit_correctness:
  forall (p:program) (s:state) (js:jit_state),
    initial_jit_state p = OK(js) →
    ir.initial_state p s →
    safe p s →
    match_states p s js init_jit_index.
```

And that progress of the source program is preserved (otherwise, a JIT that would always get stuck would be considered correct):

```
(* If the original program has a behavior (safe), then the JIT also has one *)
Theorem jit_progress:
  forall (p:program) (s:state) (js:jit_state) (ji:jit_index),
    match_states p s js ji →
```

```
safe p s →
exists js', exists e,
jit.jit_step js = OK(js', e).
```

These three theorems express our correctness proof. For a safe source program, the initial JIT state will be matched with `match_states`. As the program is safe, and progress is preserved, the JIT program can make a step. Thanks to the backward simulation, we know that this step is matched by some steps of the source program. And after these steps, the source program and the JIT program are still matched with `match_states`.

Note that the simulation relation `match_states`, the JIT index type `jit_index` and its order `jit_order` must still be defined. `match_states` takes as arguments the original program `p`, one of its semantic states `s`, a JIT state `js` and a JIT index `ji`.

```
(* Relating interpreter states of the original program and jit states *)
Definition match_states (p:program) (s:state) (js:jit_state) (ji:jit_index): Prop := ...
```

5.3.2 The JIT simulation relation

`match_states` must be described as an invariant of the JIT execution, and such that the JIT index decreases in case of stuttering.

First, one needs to state in this invariant that the original program and the current JIT program are related in some way. This is needed to ensure that every time the JIT takes an executing step (in its current program), the source program takes a matching one. This is best described with a backward simulation, between the source program and the JIT program.

Note that in our definitions, there are two kinds of backward simulations, as shown in Figure 20:

- An *external* backward simulation (`jit_simulation`) between the source program semantic states and the JIT states: each time the JIT makes a step (executing or optimizing), the source makes matching steps.
- An *internal* backward simulation (`bwd_sim`) between the source program semantic states and the JIT program semantic states: executing the current JIT program corresponds to matching steps in the source. The JIT program may be optimized. In this simulation we only match its execution steps with steps of the original program.

Simulation relations for internal simulations have the following type:

```
Definition relation {index:Type}:Type := index → state → state → Prop.
```

which differs from the type of `match_states`, the external simulation relation. Internal simulations take as arguments two programs, an order and a relation.

To prove the *external* simulation, we need an *internal* simulation as an invariant. The simulations are named this way because the internal simulation is part of `match_states`, the relation of the external simulation. If the JIT step consists in executing the current JIT program, then we can use the internal simulation to find the next matching state of the source program. If the JIT step is optimizing, then the execution of the source program stays in the same state, and we must show that the invariant still holds. In other words, there is still an internal simulation between the source program and the new JIT program, which in essence corresponds to proving the correction of our optimization.

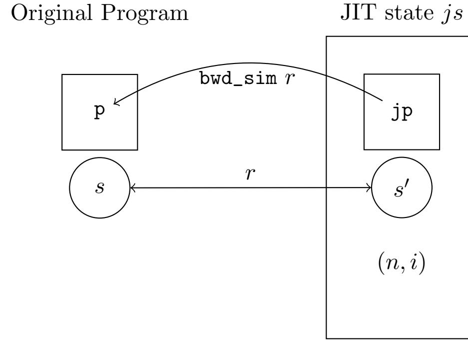

Figure 20: Internal and external simulations. With the internal simulation relation r, s is matched with s'. In the external simulation relation `match_states`, s is matched with the entire JIT state (`jp`, s', (n, i)). The internal backward simulation `bwd_sim` is part of the relation for the external simulation.

This proof relies on two main results: optimizer correction, and composition of backward simulations. The first one resembles the correction theorem of a static compiler: when optimizing a program, there is a backward (internal) simulation between the input and the result of the optimization process.

```
Theorem optimization_correctness:
  forall p ps newp,
    optimize ps p = OK (newp) →
    exists index, exists order, exists (r:relation (index:=index)),
          bwd_sim p newp order r ∧ reflexive r.
```

Note that the index type, the order and the relation are existentially quantified over, as one cannot know in advance which optimization will be performed.

The second one proves that backward simulations can be composed, by constructing the correct index type, order and simulation relation:

```
(* Backward simulation composition *)
Theorem compose_bwd_sim:
  forall p p1 p2 (i1:Type) o1 (r1:relation (index:=i1)) (i2:Type) o2 (r2:relation (index:=i2)),
    bwd_sim p p1 o1 r1 →
    bwd_sim p1 p2 o2 r2 →
    bwd_sim p p2 (compose_order o1 o2) (compose_rel r1 r2).
```

The next step is composing the current internal backward simulation of the `match_states` invariant with the internal backward simulation of `optimization_correctness`. Before the optimization, the invariant holds, and thus there exists a backward simulation between `p`, the original program, and `jp`, the current JIT program at the time of the optimization. After optimization, `jp` is replaced by `optimize jp` in the JIT state, and there is a backward simulation between `jp` and `optimize jp`. Finally, one can use `compose_bwd_sim` to prove that there is an internal backward simulation between `p` and `optimize jp`. Thus, the invariant is maintained. A diagram of all internal backward simulations can be seen on Figure 21.

In `optimization_correctness`, we require the new simulation relation `r` to be *reflexive* (meaning that for any state `s`, there exists some index `i` such that `r i s s`). Indeed, we always want (as part

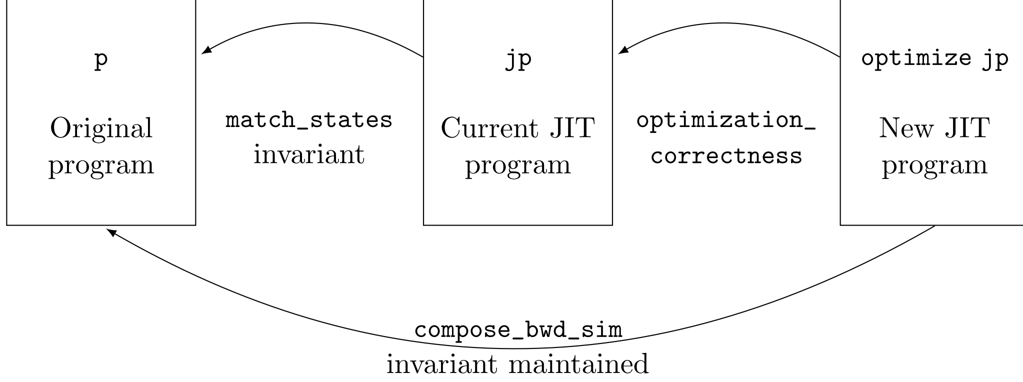

Figure 21: Maintaining an internal backward simulation when optimizing.

of the invariant) the semantic state of the original program to stay matched with the current JIT interpreter state, by the internal simulation relation. When optimizing, if the original program is in state `s`, then by invariant, there exists a simulation `r` between `jp` and `p` and an index `j` such that (`r j s js`), where `js` is the current JIT interpreter state. After the optimization, the JIT program has changed but its interpreter state is still `js`. The new internal simulation uses the relation (`compose_rel r ropt`) where `ropt` is the simulation relation of `optimization_correctness`. As `ropt` is reflexive, it follows that there exists `i` such that (`ropt i js js`), and then we can prove that (`compose_rel r ropt`) (`j`,`i`) `s js`: the original program state `s` is matched with the JIT interpreter state `js` for the new internal simulation relation: the invariant is maintained.

Our definition of `match_states` is thus:

```
(* Relating interpreter states of the original program and jit states *)
Definition match_states (p:program) (s:state) (js:jit_state) (ji:jit_index): Prop :=
  bwd_sim p (prog js) (jwft ji) (jrel ji) ∧
  (jrel ji) (jindex ji) s (int_state js) ∧
  (joptim ji) = nb_optim js.
```

The first proposition is the internal backward simulation (`bwd_sim`). The second one expresses that the current state of the original semantics (`s`) is matched with the current interpreter state of the JIT (`int_state js`), with the internal simulation relation (`jrel ji`). Finally, the `jit_index` type also includes a maximum number of optimizations to perform (`joptim ji`), to prevent infinite stuttering (see section 5.3.3).

In conclusion, the external backward simulation ensures that the behaviors of the JIT matches behaviors of the source program. As the JIT progresses, its program changes. But at all times, its program is related to the original program via an internal backward simulation. Whenever the JIT decides to optimize its program, this internal backward simulation also changes. Thanks to our composition theorem, to add a new optimization in the JIT, it suffices to prove an internal backward simulation for this optimization (`optimization_correctness`).

5.3.3 The JIT decreasing order

There are two cases where stuttering can happen:

- The JIT decides to perform an optimization. During this step, the JIT state evolves (its current program changes), yet this corresponds to no step in the source semantics.
- While executing the current JIT program, which has been obtained via optimizations, the execution takes a step that corresponds to no step in the source semantics. For instance, if some `Assume` instruction was inserted, the step that evaluates the list of expression and moves to the next instruction corresponds to no step of the source semantics.

Infinite stuttering should never happen. Otherwise, the JIT could get stuck doing stuttering steps while the original program would have a finite behavior.

In the second case, when executing an optimized program, we can be sure that infinite stuttering does not happen. Indeed, we know that there is an internal backward simulation, and such a simulation ensures that there is no infinite stuttering if we keep executing the current JIT program. However, such an argument does not hold if we interleave optimizations with execution of stuttering steps. Each time the internal simulation changes, its decreasing order might also change.

To ensure progress, we must then restrict the JIT so that it cannot always optimize. In our case, we decided to add a counter of optimizations inside the JIT state. Each time an optimization is performed, it decreases. This corresponds to real strategies in actual JIT compilers, where the time spent optimizing is limited.

Finally, our `jit_index` type contains both the counter of optimizations left, as well as the order, relation and index of the internal simulation. Note that the order, typed `wft_order`, contains an index type, which may change as we perform optimizations, a proof of well-foundness and a proof of transitivity. Then, our `jit_order` is a lexicographic order, with a special feature: the second order may change, but only when the first one decreases. Because we only change it to well-founded orders, this lexicographic order is well-founded as well.

```
(* The number of optimizations left *)
Definition optim_index: Type := nat.
Definition optim_order: optim_index → optim_index → Prop := lt.

(* The decreasing order, during the execution, given by the current backward simulation *)
Record exec_index : Type := mkindex {
  wft: wft_order;
  rel: @relation (index wft);
  index: index wft }.

(* This order decreases only if the order and relation stay the same *)
Inductive exec_order: exec_index → exec_index → Prop :=
| exec_ord:
    forall (wft:wft_order) rel i i',
      (ord wft) i' i →
      exec_order (mkindex wft rel i') (mkindex wft rel i).

(* The order that decreases each time the JIT takes a stuttering step *)
Definition jit_index: Type := optim_index * exec_index.
Definition jit_order: jit_index → jit_index → Prop :=
  lex_ord lt exec_order.
```

The `optim_index` is simply a natural integer. The usual *less-than* (`lt`) relation on naturals serves as the first well-founded order of the lexicographical order.

Then, our `exec_index` type represents the index, the order and the relation of the internal simulation relation. As the index type might change during the execution (each time a new optimization is added), the record also includes it. Our `exec_order` relation is an order over any `exec_index`. It only decreases when the order of the internal simulation decreases, meaning that the index type, the order itself and the relation must stay unchanged.

Finally, `jit_order` is the lexicographic order over pairs of `optim_index` and `exec_index`. We thus have two decreasing cases:
$$\begin{array}{cccc} \forall\ n\ n'\ i\ i', & n' < n & \rightarrow & (n', i) < (n, i') \\ \forall\ n\ i\ i', & i' < i & \rightarrow & (n, i') < (n, i) \end{array}$$
where n and n' are typed `optim_index`, i and i' are typed `exec_index`. By $i' < i$, we mean that we have `exec_order i' i` and thus the index type, the order and simulation relation of i and i' are the same.

In summary, during execution, the first order (number of optimizations left) stays unchanged, while the second order (specific to the current internal simulation) decreases. When an optimization is performed, the first order decreases, and the second one changes to another well-founded order. As the lexicographic order is well-founded, this ensures that the JIT itself cannot stutter indefinitely, and its progress will eventually correspond to some source progress.

5.3.4 The JIT simulation

Figure 22 depicts the external backward simulation. JIT states are composed of a program, a state and an index of type `jit_index`. In practice, it also includes some profiler information, but this is not depicted in this diagram, as it plays no role in the simulation.

The hypotheses are pictured in solid lines. In each case, we assume that the JIT is in a JIT state js, matched with the source program at some program state. We also assume that the JIT steps to a new JIT state js'. The `match_states` invariant is represented by the two horizontal arrows: the internal backward simulation (for r, some internal simulation relation), and the matching of the states with r and some index i.

The conclusion is pictured in dashed lines. Either we can find a new original program state for which the invariant holds, or the new JIT state is still matched to the old original state, but with the JIT index decreasing. In both stuttering cases, the JIT index lexicographically decreases: either the internal index decreases ($(n, i') < (n, i)$ because $i' < i$), or the number of optimizations ($(n-1, i) < (n, i')$).

5.4 Proving optimizations correct

With this proof scheme, one can add optimization passes, as long as one proves a backward simulation for each one. Instead of directly proving the backward simulation, one can use another simulation scheme that implies the backward one.

For instance, we proved the correctness of the speculative calls optimization (introduced in section 4.2.1) with a special forward simulation scheme, which can be seen Figure 23. There are two cases. Either we have a forward lockstep: if the source program takes a step, then the compiled takes a matching step, with the same event. In the other case, the compiled program is allowed to take a step with a silent event while the source does not progress, but some measure decreases.

Recall that the speculative calls optimization adds a new version to a function, with an `Assume` instruction at the top. Our internal simulation relation relates states of some program and states

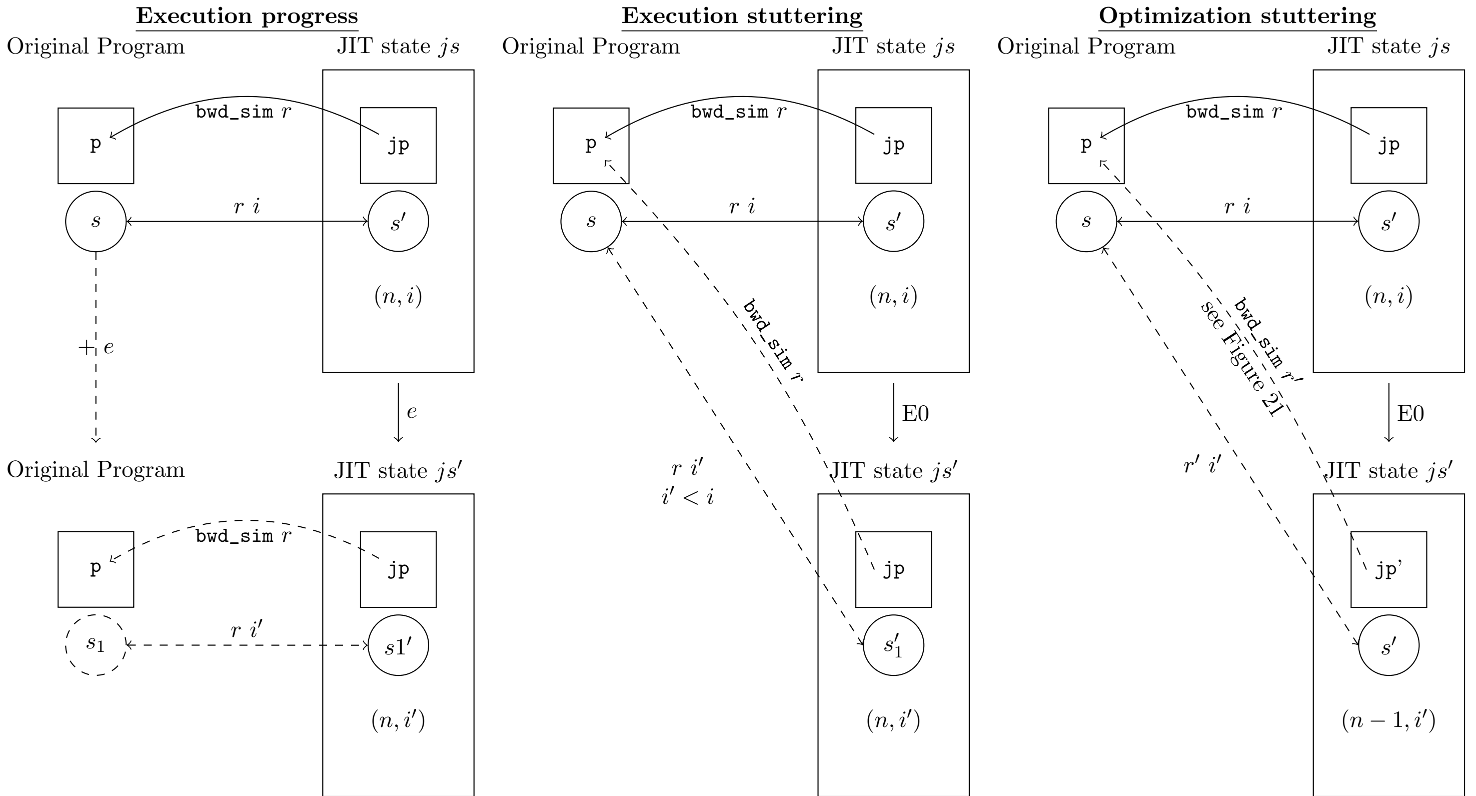

Figure 22: The 3 cases of JIT external simulation

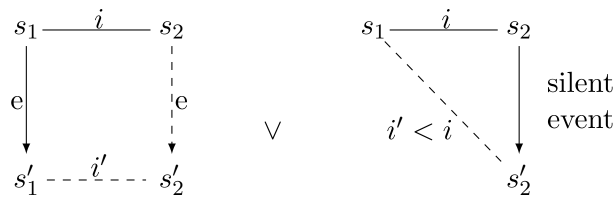

Figure 23: The forward simulation scheme for speculative calls

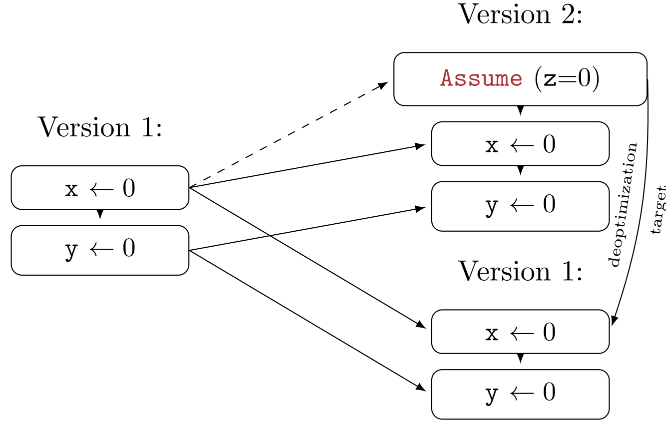

Figure 24: The internal simulation relation for speculative calls. The optimized program is on the right. The dashed line has a bigger index than solid lines.

of the optimized program. In particular, it relates all states with themselves. The state with the `Assume` instruction is related with the entry of the unoptimized version. Other states of the optimized version are related to corresponding states of the unoptimized version. The relation can be seen in Figure 24.

We can then prove the simulation scheme: in most versions, the code is unchanged, and we can use the lockstep. At the top of the optimized version, one `Assume` instruction has been inserted. The simulation relation matches this `Assume` with the start of the unoptimized version. As one instruction has been added, the optimized version takes one more step, which is either passing or failing the `Assume`: a silent event. Our internal simulation relation is such that in both cases, we are matched with a smaller index to the start of the unoptimized version: on Figure 24 for instance, the $\texttt{x} \leftarrow 0$ instruction on the left is matched with both $\texttt{x} \leftarrow 0$ instructions on the right. We can thus prove the second part of the simulation scheme.

We must also prove that inserting the assumptions cannot block the execution. The semantics of the `Assume` are non-blocking as long as the condition can evaluate safely (there is no unassigned registers used) and the deoptimization target exists. In this case, the condition does evaluate safely because all registers used are parameters of the function, and as such are all initialized when calling it. The deoptimization target is the unoptimized version, and can be found in the new program as well.

Finally, we proved that this simulation scheme implies a backward simulation. The speculative calls optimization is then correct, for any `fidoptim` (identifier of the function to optimize) and `spec_args` (the values of the parameters suggested by the profiler).

```
Theorem spec_calls_correctness:
  forall p fidoptim spec_args newp, optimize_spec_calls fidoptim spec_args p = OK (newp) →
    exists order, exists (r:relation), bwd_sim p newp order r ∧ reflexive r.
```

So far, by lack of time, the constant propagation optimization has not yet been verified in Coq. However, we are confident that a correctness proof can be done. Constant propagation can be proved with a forward simulation (see section 5.2), as seen in CompCert. Compared to CompCert RTL, JIT IR has the new `Assume` instruction. When optimizing a version, this means that the

execution can leave the version, and that versions can be entered at any location, not always at the entry point of the version.

One of the advantages of JIT compilation is that optimizations can be done locally, without looking at all possible versions. When proving constant propagation correct, the execution leaving the current version is not an issue. Indeed, if the source step leaves because it failed an `Assume`, then the target should make the same step: during constant propagation, the `Assume` instruction has been copied. Some registers may have been replaced in the assumption, but one invariant of proving constant propagation is that if a register r has been replaced by a value v at some instruction, then r was mapped to v at this point in the source program execution. This ensures that the two speculation conditions evaluate to the same result in the source and the compiled version. As the deoptimization target is simply copied, the compiled version will take a step matching the source one.

The remaining time of our internship will be spent formalizing and proving in Coq this constant propagation analysis.

6 Future Work: Transparency

So far, our definitions are enough to prove the correctness of the JIT compiler, including some optimizations. As Flückiger *et al.* showed [11], some optimizations specific to speculation can be proved with the *transparency invariant*. Such optimizations include moving or merging the `Assume` instructions. This invariant, while not needed for our previous proofs, seems well adapted to prove these other optimizations.

Informally, the transparency invariant holds if upon executing an `Assume`, one can always deoptimize, even if the speculative condition holds. And deoptimizing does not change the observational semantics, *i.e.* the JIT execution will still be correct with respect to the source semantics. An important corollary is that one can always modify the program to make an `Assume` condition stronger. If the new assumption was to fail, then the JIT execution would still be correct because deoptimizing does not change the semantics.

Proving such an invariant is challenging. With the semantics described in section 4.1.2, one would need to reason about behaviors that do not exist (as the deterministic semantics only deoptimize if the condition does not hold). Our plan to handle transparency is to define an alternate non-deterministic semantics where, even if the condition holds, deoptimizing upon encountering an `Assume` is always a valid behavior. This consists in adding a new semantic rule to those of Figure 16:

$$\text{Deoptimize}\ \frac{\begin{array}{c} v\ !\ pc\ =\ \texttt{Assume}\ le\ (fa, va, la)\ rom\ sl\ next \qquad (le, rm)\ \Downarrow\ true \\ \texttt{update_regmap}\ rom\ rm\ =\ rm' \quad \texttt{synthesize_frame}\ rm\ sl\ =\ synth \end{array}}{(s, v, pc, rm, ms) \to (synth{+}{+}s, va, la, rm', ms)}$$

This rule is identical to the one where we deoptimize because the condition is false, except that this time the condition evaluates to true. This does not mean that the interpreter will deoptimize when the condition is true. Just as before, the interpreter can decide to use the previous rule of the semantics and go to the next instruction when the condition holds. Our interpreter correctness theorem only requires that the interpreter picks one of the possible behaviors. If the JIT is proved correct with respect to this semantics, then the transparency invariant holds: deoptimizing is always a valid behavior.

However, this non-determinism breaks the proof that a forward simulation implies a backward simulation in the general case. In CompCert proofs, one can create a backward simulation from a forward even if the language is not deterministic. But this non-determinism is restricted. Informally, the proof works if non-deterministic steps produce events that are non-silent (*i.e.* that are part of the trace). One could add such non-silent events to the rules Deoptimize and Assume Holds of the semantics. But recall that for correctness, the source program and its execution through the JIT must have the same trace, which prevents inserting new `Assume` instructions when optimizing. For standard compiler optimizations, such as constant propagation, the `Assume` instructions are neither added, moved nor removed, thus using non-silent deoptimizations is a solution: one can still prove a forward simulation and deduce a backward one. Some of these proofs have only been done on paper so far.

For other optimizations, deoptimizations are silent, and one must find new simulation schemes to prove the backward simulation. For the speculative calls optimization, we defined a new scheme that resembles Figure 23. In addition, you also need to prove that, when encountering an `Assume`, the states you end up to when deoptimizing or staying in the optimized version are matched to the same source states. The scheme also implies a backward simulation in the non-deterministic case. This optimization has been proved in Coq.

Overall, we are confident that with the right simulation schemes, one can prove JIT correctness for the non-deterministic semantics. Then, the next step would be to use the transparency invariant to prove more speculative optimizations. Such changes to the semantics require new unusual simulation schemes, specific to each optimization (for instance, the scheme for our speculative calls optimization does not work for other speculative optimizations). Further work could also consist in trying to find a more general simulation scheme.

7 Conclusion

There are many ways, including semantics and program logics, to formally reason about programs. Proof assistants allow a mechanization of proofs. Proof assistants such as Coq have reached enough maturity to prove complex theorems such as compiler correctness. CompCert is an example of verified compiler, with various optimizations and performance on par with modern compilers, which shows that formally verified realistic compilers are feasible.

JIT compilers are complex software, with many components. There are many different JIT compilers, from trace-based compilers to compilers with complex deoptimization techniques. Other techniques, such as partial evaluation, resemble JIT compilation in some ways. With so many differences, it is hard to give an exact definition of JIT compilation. For instance, earlier versions of V8 (a modern JavaScript engine) did not even have an interpreter, and everything ended up being compiled, without producing any intermediate code.

Few works have tried formalizing and verifying JIT compilers so far, compared to standard compilers and interpreters. As JIT compilation is more and more prevalent, the need for reliable JIT compilers is growing. Myreen [22] presents a fully-verified JIT compiler. To the best of our knowledge, this is the only mechanized proof of just-in-time correctness. However, the compiler itself is not very typical of modern JIT compilers, in terms of design and optimizations (none are done). It successfully showcases the challenges of self-modifying code semantics and the usability of proof-producing compilation, but self-modifying code is not the only challenge of proving a realistic modern JIT compiler.

Instead, we suggest using the simulation techniques to prove the correctness of a small JIT compiler in Coq, in an approach similar to CompCert. We designed a small language, used as both input and output, a parser, an interpreter, a profiler, an optimizer, and finally the JIT itself. Our correctness theorem resembles a standard backward simulation, and guarantees that a JIT-compiled program behaves as prescribed by its semantics. The dynamic nature of optimizations required substantial changes to the simulation proofs. However, we have laid out a proof that easily could be augmented with new optimizations: one has to prove a standard backward simulation to add a new optimizing pass. This should be less work than the weak bisimulations of [11], or the bisimulations of [16].

We made several design choices for our JIT compiler. This is not a tracing JIT. For now, optimizing entire functions is simpler, as we can use the same optimizations as in standard static compilers. One could imagine a similar approach to ours for a tracing JIT, but such modification would not be trivial (and we know that optimizations have to be different in a tracing JIT [16]). Our input and output language was made for this work, and does not have all the features of more realistic languages. Yet JIT IR is still closely resembling CompCert RTL with an `Assume` instruction, and extending it should not be an issue.

We provided machine-checked proofs for our theorems. Even if not many optimizations have been implemented yet, this is still more realistic than the only previous mechanized proof of Myreen [22]. Our different components, while simple, resemble the architecture of modern JIT compilers. We believe that this work successfully shows how standard compiler verification techniques can be adapted to JIT compilers. And that verified realistic JIT compilation should be feasible.

Future work should focus on implementing the transparency invariant, and then use it to prove more JIT-specific optimizations.

References

[1] HOL Interactive Theorem Prover. `https://hol-theorem-prover.org/`.

[2] The Coq Proof Assistant. `https://coq.inria.fr/`.

[3] The Isabelle Proof Assistant. `https://isabelle.in.tum.de/index.html`.

[4] The Menhir parser generator for the OCaml programming language. `http://gallium.inria.fr/~fpottier/menhir/`.

[5] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 2003.

[6] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Programming Language Design and Implementation (PLDI)*, 2000.

[7] Gilles Barthe, Delphine Demange, and David Pichardie. A formally verified ssa-based middle-end - static single assignment meets compcert. In *ESOP*, 2012.

[8] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.*, 1995.

[9] Daniele Cono D'Elia and Camil Demetrescu. Flexible on-stack replacement in LLVM. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization.*

[10] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, New York, USA, 2013. ACM.

[11] Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek. Correctness of speculative optimizations with dynamic deoptimization. *PACMPL*, 2018.

[12] Yoshihiko Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 1999.

[13] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of PLDI*, 2009.

[14] Georges Gonthier. The four colour theorem: Engineering of a formal proof. In *Computer Mathematics, 8th Asian Symposium, ASCM*, 2007.

[15] Ben Greenman. Short Presentation of Kildall's algorithm. `http://www.ccs.neu.edu/home/types/resources/notes/kildall/kildall.pdf`.

[16] Shu-yu Guo and Jens Palsberg. The essence of compiling with traces. In *Proceedings of the Symposium on Principles of Programming Languages, POPL*, 2011.

[17] Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages, 1973.*

[18] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. Cakeml: a verified implementation of ML. In *Proceedings of POPL*, 2014.

[19] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason. 2009.*

[20] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of POPL*, 2006.

[21] Pierre Letouzey. Extraction in Coq: An Overview. In *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, 2008.*

[22] Magnus O. Myreen. Verified just-in-time compiler on x86. In *Proceedings of the Symposium on Principles of Programming Languages, POPL*, 2010.

[23] Magnus O. Myreen, Konrad Slind, and Michael J. C. Gordon. Extensible proof-producing compilation. In *Compiler Construction, CC*, 2009.

[24] Chris Seaton. Understanding How Graal Works - a Java JIT Compiler Written in Java. `https://chrisseaton.com/truffleruby/jokerconf17/`.

[25] M. Vandercammen, J. Nicolay, S. Marr, J. De Koster, T. D'Hondt, and C. De Roover. A formal foundation for trace-based JIT compilers. In *Proceedings of the 13th International Workshop on Dynamic Analysis*, 2015.

[26] Glynn Winskel. *The formal semantics of programming languages - an introduction.* Foundation of computing series. MIT Press, 1993.

[27] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of PLDI*, 2017.

[28] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011.*

[29] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the Symposium on Principles of Programming Languages, POPL*, 2012.