Verification of a Just-In-Time compiler

Roméo La Spina

Abstract. For dynamic languages, using Just-In-Time compilation allows to execute programs faster compared to interpretation. They rely on speculations that are made, at code generation time, to specialize the code generated for the most common cases. Verifying a JIT compiler raises new important challenges which doesn't appear in verified batch compilers.

For these reasons, verified JIT compilers are still under study. For example, an architecture of verified JIT compiler has been developed by Aurèle Barrière et al. This work presents two improvements of the optimization passes previously deployed in this JIT compiler, and formally verifies their correctness in Coq. The first one consists in the improvement of an inlining pass, which consists in relaxing a condition that must be true for inlining to trigger at compile time. As a result, inlining can be applied more often. We have checked on a benchmark that, with this improvement, inlining can happen more often. The second improvement is an improvement to the way we introduce speculations, which uses a liveness analysis. When we introduce a speculation, we must be able to rebuild the environment in case we have to deoptimize because the speculation doesn't hold. The liveness analysis allows to reduce the size of this environment. We have finally shown, with the same benchmark as with inlining, that this improvement reduced the execution time.

Contents

1 Introduction **2**
1.1 Context . . . 2
1.2 Properties of the semantics of SpecIR . . . 2
1.3 Aim of our project . . . 2

2 First improvement : Relaxation of the condition to trigger inlining **3**
2.1 Presentation . . . 3
2.2 Verification . . . 4
2.2.1 Overall principles . . . 4
2.2.2 Invariants . . . 5
2.2.3 Main steps of the proof . . . 6

3 Second improvement: Introduction of a liveness analysis for `Framestate` insertion **7**
3.1 Presentation . . . 7
3.2 Verification . . . 8

4 Performance analysis **10**
4.1 With improved inlining . . . 10
4.2 With liveness analysis . . . 10

5 Conclusion **10**

Acknowledgements

I would like to express my sincere gratitude to my supervisor David Pichardie, from the CELTIQUE team at IRISA, for giving me the golden opportunity to work on this project which significantly improved my knowledge in formal proofs and verified compilation, and for all the advices he gave me over the course of this project.

I would also like to deeply thank Aurèle Barrière for his valuable guidance and support from the beginning to the end of the project, which were essential for me to get in the right direction towards the achievement of this project. His help was also very precious for the redaction of the present report.

1 Introduction

1.1 Context

The work presented in the present report is a project that is part of Aurèle Barrière's PhD thesis, whose goal is to design a verified Just-In-Time (JIT) compiler with the proof assistant Coq [1]. This compiler, and especially the libraries it uses, are partially inspired by CompCert, a C verified compiler which is a main reference in the domain of verified compilation [2]. This JIT compiler introduces SpecIR, its intermediate representation whose semantics are enhanced with special instructions that allows speculation, the specific feature of JIT compilers. Then, our compiler performs several optimization passes on the SpecIR program, such as inlining or constant propagation, and the resulting program is finally converted to LLVM, a well-known backend.

1.2 Properties of the semantics of SpecIR

SpecIR features some classical instructions, like assignment, binary operations, memory store/load, but also two particular instructions: `Framestate` and `Assume`. The latter allows the compiler to make one or more speculations on the current environment. If the speculation is true, we can execute an optimized version of the program since we have some guarantees on the environment, so as to curb the execution time. If the speculation is not true, the compiler gets back to a base version of the program, and restore a correct environment to pursue the execution of the program. This is called a deoptimization. For `Assume` instructions to be inserted, one first needs to insert a `Framestate` instruction. It expresses the possibility of a deoptimization. In later optimization passes, speculation can be inserted next to any `Framestate` instruction in the form of an `Assume`.

1.3 Aim of our project

Our project consisted in improving two optimization passes, firstly the inlining pass provided by the compiler, and secondly the pass that insert `Framestate` instructions in order to be able to insert `Assume` instructions later. The second optimization is now based on a liveness analysis, a technique that is frequently used in compilation. Moreover we have proven the correctness of these improvements by adapting the Coq proofs of these two optimization passes to their improved implementation.

2 First improvement : Relaxation of the condition to trigger inlining

2.1 Presentation

Whenever a speculation fails, we must restore a correct environment for the original version of the function being executed. If a speculation fails in inlined code, we must not only restore the environment of the inlined function, but also the environment of the function in which the function was inlined. To know how to reconstruct that environment, the inlining pass checks that there is a `Framestate` instruction next to the Call being inlined.

To make this proof easier, a strong condition on the form of the `Framestate` was introduced. Therefore, the inlining was not triggering very often in practice. Our improvement consisted in relaxing this condition to allow inlining more often.

A `Framestate` is defined by :

- A label pointing to an instruction of the base version, where we must jump to when deoptimizing
- A mapping of the registers (varmap) and a list of stackframes, to restore the correct environment.

Previously, the varmap of the inserted `Framestate` had to map the return register `retreg` of the called function to its current value before all the other values. We relaxed the condition so as to only need that the varmap contains once and only once the assignment ($\texttt{retreg} \leftarrow \texttt{retreg}$), possibly in the middle of the other assignments. To sum up, before the improvement, we had to have a varmap of the form

$$[(\texttt{retreg} \leftarrow \texttt{retreg}), (r_{x_1} \leftarrow e_{x_1}), \ldots, (r_{x_n} \leftarrow e_{x_n})]$$

whereas in the new version of inlining, a varmap of the form

$$[(r_{x_1} \leftarrow e_{x_1}), \ldots, (r_{x_k} \leftarrow e_{x_k}), (\texttt{retreg} \leftarrow \texttt{retreg}), (r_{x_{k+1}} \leftarrow e_{x_{k+1}}), \ldots (r_{x_n} \leftarrow e_{x_n})]$$

for some k is accepted, where all the r_{x_i} differs from `retreg` and none of the e_{x_i} use `retreg`.

To relax this condition, we have to modify, in the Coq implementation of the inlining pass, the definition of a function named `check_vm`, which checks the form of the given varmap, and returns a unit type element if the varmap is well-formed in the sense of the previous criteria, or returns an error if the varmap does not satisfy these criteria.

Previously, `check_vm` was checking if the varmap is not empty and if its first element is indeed ($\texttt{retreg} \leftarrow \texttt{retreg}$). We give below a simplified version of its implementation in Coq (skipping the additional checks)

```
Definition check_vm (vm:varmap) (retreg:reg) (abs:def_abs_dr): res unit :=
  match vm with
  | nil ⇒ Error "the varmap does not capture the return register"
  | a:: vm' ⇒
    match (andb (check_reconstructed retreg a) (varmap_no_use retreg vm')) with
    | true ⇒ OK tt
    | false ⇒ Error "The varmap doesn't have the desired form"
    end
  end.
```

Our changes include two new functions, `check_varmap_capture_retreg` and `check_varmap_only_once`, which respectively check that

- the varmap indeed contains the assignment (`retreg` ← `retreg`)
- the varmap, excepted at most one eventual assignment (`retreg` ← `retreg`), never use `retreg`.

The new version of `check_vm` is given below.

```
Definition check_vm (vm:varmap) (retreg:reg) (abs:def_abs_dr): res unit :=
  match check_varmap_capture_retreg vm retreg with
  | true ⇒
    match check_varmap_only_once vm retreg with
    | true ⇒ OK tt
    | false ⇒ Error "The varmap doesn't have the desired form"
    end
  | false ⇒ Error "the varmap does not capture the return register"
  end.
```

2.2 Verification

2.2.1 Overall principles

First of all, we shall explicit what we mean by "the optimization is correct" when we refer to an optimization pass from a source SpecIR program to an optimized SpecIR program. It actually means that for any source program, there is a *backward simulation* between the source program and the resulting optimized program. This concept of backward simulation is actually the very essence of the correctness proof of CompCert.

Briefly, a backward simulation expresses the fact that every possible behavior of the optimized program is a *correct* behavior of the source program (before the optimization), where correct means that the behavior is included in the specification of the source program given by its semantics.

To prove a backward simulation, we have to define a matching relation between states of the source program and states of the optimized program, that guarantees that a set of invariants is preserved between these states, then prove that this relation is preserved at each step of the execution.

Formally, if two states s_1 and s_2 are matched ($\approx$ in the diagrams below), and s_2 steps to another state s_2', then we must have for the simulation to be true :

- either s_1 steps to a state s_1' after a finite non-zero number of steps, and s_1' is matched with s_2', and both the successive steps from s_1 to s_1' and the step from s_2 to s_2' have the same observable behavior (trace t)

$$\begin{tikzcd}
s_1 \arrow[r, dash, "\approx"] \arrow[d, dashed, "+" ', "t"] & s_2 \arrow[d, "t"] \\
s_1' \arrow[r, dash, dashed, "\approx"] & s_2'
\end{tikzcd}$$

- either, under some condition, s_1 is directly matched with s_2', and $s_2 \to s_2'$ has no observational behavior

$$\begin{tikzcd}
s_1 \arrow[r, dash, "\approx"] \arrow[dr, dashed] & s_2 \arrow[d, "\emptyset"] \\
 & s_2'
\end{tikzcd}$$

All the challenge lies in defining such a matching relation, which must be strong enough to guarantee that both programs will have the same behavior, but not too strong in order to be preserved at each step. Then we must prove that backward simulation holds with this matching relation.

2.2.2 Invariants

In our JIT compiler, the environments that contain the values of all the registers are represented by mappings that associate a register number to a value. This structure is called a *regmap*. They are internally represented with binary search trees

In the initial proof, it was shown that if we assume that the varmap of the inserted `Framestate` is well-formed (initially, with (`retreg` ← `retreg`) at the beginning), when we deoptimize from the inlined code, the final regmap is the same as the initial regmap (before the call) with `retreg` receiving the return value of the inlined function, whatever the result of the inlined function is.

With the new condition on the inserted `Framestate`, however, this is no longer guaranteed. Indeed, the regmaps are not exactly the same in the sense of Coq equality, but they are *equivalent* i.e. they have the same values on all registers, despite being internally represented differently. Therefore, we have to change the previous set of invariants so as to only require for regmaps in two matched states to be equivalent, and to adapt the proof in order to prove that this new set of invariants is sufficient to have the backward simulation.

In Coq, the equivalence between two regmaps is denoted by the following property, where `rm # r` is the value of the register `r` in the regmap `rm` :

```
Definition regmap_eq (rm1 rm2:reg_map): Prop :=
  ∀ r:reg, rm1 # r = rm2 # r.
```

Before introducing the match relation between states, first of all, we have an inductive property `match_stack` expressing the fact that two stacks are similar. The definition of this property is composed of 5 cases, only the 2 cases that are different from the original definition are given below for the sake of simplicity.

```
Inductive match_stack (p:program) (caller:version) (callee:version) (call_lbl:label)
(args:list expr) (retreg:reg) (next:label) (tgt:deopt_target) (vm:varmap)
(sl:list synth_frame) (params:list reg) (abs:def_abs_state): stack → stack → Prop :=
| ms_same: ∀ s1 s2 r v lbl rm rm'
   (MS: (match_stack p caller callee call_lbl args retreg next tgt vm sl params abs) s1 s2)
   (EQ: (regmap_eq rm rm')),
      (match_stack p caller callee call_lbl args retreg next tgt vm sl params abs)
         ((Stackframe r v lbl rm)::s1) ((Stackframe r v lbl rm')::s2)

| ms_deopt: ∀ s1 s2 tgtver rm rmdeopt synth
   (MS: (match_stack p caller callee call_lbl args retreg next tgt vm sl params abs) s1 s2)
   (UPDATE: ∀ retval, ∃ rmdeopt',
      update_regmap vm (rm # retreg← retval) rmdeopt' ∧
      regmap_eq rmdeopt' (rmdeopt # retreg← retval))
   (SYNTH: ∀ retval, synthesize_frame p (rm # retreg← retval) sl synth)
   (FINDBASE: find_base_version (fst tgt) p = Some tgtver),
      (match_stack p caller callee call_lbl args retreg next tgt vm sl params abs)
         ((Stackframe retreg caller next rm)::s1)
         ((Stackframe retreg tgtver (snd tgt) rmdeopt)::synth ++ s2).
```

The interesting changes to notice are indeed in the two cases susmentioned : in `ms_same`, instead of having the same regmap on both stackframes at the top of the two stacks, we have equivalent regmaps. In the `ms_deopt` inductive case, it is the `UPDATE` hypothesis which is different. This is due to the fact that if (`retreg` ← `retreg`) appears in the middle of the varmap, the registers are not assigned in the same order. Since regmaps are represented as binary search trees, if we assign keys in a different order, we obtain a different internal representation, even if each key holds the same value.

Finally, we have the definition of our fundamental property `match_states` that matches two similar states. For this definition, only one case differs. Below is given the concerned case. It previously required to have the same regmap in both states, now it only requires equivalent regmaps.

```
Inductive match_states (p:program) (caller:version) (callee:version) (call_lbl:label)
(args:list expr) (retreg:reg) (next:label) (tgt:deopt_target) (vm:varmap)
(sl:list synth_frame) (params:list reg) (abs:def_abs_state): unit → state → state → Prop :=
| refl_match: ∀ stk stk' v l rm rm' ms
   (MS: (match_stack p caller callee call_lbl args retreg next tgt vm sl params abs) stk stk')
   (EQ : regmap_eq rm rm'),
      (match_states p caller callee call_lbl args retreg next tgt vm sl params abs) tt
         (State stk v l rm ms)
         (State stk' v l rm' ms)
```

2.2.3 Main steps of the proof

First of all, the new proof consists in showing that if the new hypotheses on the inserted `Framestate` are true, the varmap associated is always as we want, namely of the form `vm1 ++ (retreg ← retreg)::vm2` where `vm1` and `vm2` never refer to `retreg`.

In the previous version of inlining, the following lemma was part of the proof:

```
Lemma update_deopt_regmap_first:
   ∀ vm rm_r rm_deopt r v,
      update_regmap ((r, Unexpr Assign (Reg r))::vm) rm_r rm_deopt →
      varmap_no_use r vm = true →
      update_regmap ((r, Unexpr Assign (Reg r))::vm)
                    (rm_r # r ← v) (rm_deopt # r ← v).
```

The main challenge when relaxing the needed condition to inline lies in this particular point of the proof. Indeed, we would like to prove instead:

```
Lemma update_deopt_regmap_middle:
   ∀ vm1 vm2 rm_r rm_deopt r v,
      update_regmap (vm1 ++ (r, (Unexpr Assign (Reg r)))::vm2) rm_r rm_deopt →
      varmap_no_use r vm1 && varmap_no_use r vm2 = true →
      update_regmap (vm1 ++ (r, (Unexpr Assign (Reg r)))::vm2)
                     (rm_r # r ← v)  (rm_deopt # r ← v).
```

but this theorem is not true. Actually, as we mentioned in the previous section, the resulting regmap is equivalent but not equal, therefore we had to prove the following lemma:

```
Lemma update_deopt_regmap_middle:
   ∀ vm1 vm2 rm_r rm_deopt r v,
      update_regmap (vm1 ++ (r, (Unexpr Assign (Reg r)))::vm2) rm_r rm_deopt →
      varmap_no_use r vm1 = true →
      varmap_no_use r vm2 = true →
      ∃ rm_deopt',
         update_regmap (vm1 ++ (r, (Unexpr Assign (Reg r)))::vm2) (rm_r # r ← v) rm_deopt' ∧
         regmap_eq rm_deopt' (rm_deopt # r ← v).
```

To prove this lemma, we actually transform our varmap with (`retreg` $\leftarrow$ `retreg`) in the middle to a varmap with (`retreg` $\leftarrow$ `retreg`) first, in order to be able to apply the lemma `update_deopt_regmap_first`. We get then back to our original varmap, nonetheless without preserving the strict equality but rather an equivalence.

Finally, in the proof of the backward simulation, we had to show that regmap equivalence is sufficient for basic properties to hold, such as the fact that evaluating the same expression in equivalent regmaps returns the same value.

3 Second improvement: Introduction of a liveness analysis for `Framestate` insertion

3.1 Presentation

Each `Framestate` comes with data that allows the compiler to restore a correct environment if deoptimization occurs, and among others, a varmap to restore the contents of the registers. When inserting a `Framestate`, the just-in-time compiler used to perform a definition analysis, and to restore all the registers that are defined in the program including useless ones, thus making the environment carried by the `Framestate` very big. Having as few registers as possible to restore intuitively reduce register pressure, therefore it might be interesting to try to only assign the registers whose value is significant for the rest of the execution of the program.

A liveness analysis allows us to know, at code generation time, which registers are actually important to restore. Therefore, we can improve our way of inserting `Framestate` instructions. Our second improvement consisted in adding a liveness analysis to our definition analysis, in order to only assign the live registers (holding a significant value) in the varmap.

The implementation of our liveness analysis is based on the one of CompCert, which we adapted so as to be able to use it in our JIT compiler. This liveness analysis consists in a backward data-flow analysis that gives for each label of the program, the set of the registers that are live after this label. This data-flow analysis is performed using the implementation of Kildall's algorithm provided by CompCert.

In order to apply Kildall's algorithm, we had to define a transfer function, `live_dr_transf`, which computes the set of live registers before an instruction at a given label in a function, provided the set of live registers after the instruction. For instance, if `r1` and `r2` are live after an instruction `r2 <- r1 + r3`, then `r1` will still be live before (since it is used by the instruction, its value is significant), `r3` will be live too, but `r2` will no longer be live because its value is erased when processing the instruction. In conclusion, if a code `c` has the instruction `r2 <- r1 + r3` at label `l`, with `after` being a set containing exactly `r1` and `r2`, then `live_dr_transf c l after` will be a set containing exactly `r1` and `r3`.

Next, our contribution also consisted in integrating this liveness analysis to the previously implemented `Framestate` insertion. Before our improvement, the varmap carried by the inserted `Framestate` was the *identity* varmap on the set of the defined registers at the label of the `Framestate`, where the identity varmap on a set $S = \{r_{i_1}, \dots, r_{i_n}\}$ is the varmap $[(r_{i_1} \leftarrow r_{i_1}), \dots, (r_{i_n} \leftarrow r_{i_n})]$. The key difference with our improvement is that instead of taking the identity varmap on defined registers, we take the identity varmap on the *intersection* between defined and live registers. Actually, we cannot simply take the identity varmap on live registers, because there can be live registers that are not defined in ill-formed programs. For instance, let us consider the following program:

```
r1 <- 1
r3 <- r1 + r2
```

After the second instruction, no registers are live. Therefore, before the second instruction, `r1` and `r2` are live. However, `r2` is not defined at the second instruction.

We must not assign such registers in the varmap of the `Framestate` in order to preserve the backward simulation, since in case of failure, the program needs to fail at matching states of the base and the optimized version (when using the value of these registers).

In the example, `r2` will also be live before the first instruction. As a consequence, if we insert a `Framestate` before the first instruction which assigns all the live registers, it will contain the assignment `r2 <- r2`, and at deoptimization, it will directly result in a failure (since `r2 <- r2` will need the value of `r2` which is undefined), while it will only fail at the second instruction in the base version (because the second instruction references `r2`). The optimized version must also fail at the instruction `r3 <- r1 + r2` in order to keep the backward simulation.

3.2 Verification

To verify the new `Framestate` insertion pass, we followed the same process as with inlining: we had to find a set of invariants preserved at each execution step.

Once we performed the liveness analysis, one of the invariants must express the fact that when we deoptimize back to the base version of the program, the two resulting states are such that live registers have the same value in both states. Other registers aren't significant.

Therefore, we have first to define a property, named `agree`, which states that two regmaps *agree* on a set of registers, which means that they have the same value on all the registers of this set. In Coq, we gave this definition:

```
Definition agree (rm1 rm2:reg_map) (adr:live_abs_dr) :=
  ∀ r:reg, PositiveSet.In r adr → rm1 # r = rm2 # r.
```

The new `match_states` function is similar to the previously implemented one for `Framestate` insertion, excepted that there is an additional case. Indeed, the values on dead registers may be different when we deoptimize from an inserted `Framestate` since we don't restore them.

This case of the matching relation is used after a deoptimization from an inserted `Framestate`. In this new case, we only require that `rmo` (the environment obtained after deoptimization to the original version) agrees with `rms` (the environment if we had simply executed directly the original version) on the set of live registers at the current instruction. This new case is given below:

```
| deopt_match:
  ∀ s s' pc rms rmo ms
    (MATCHSTACK: (match_stack v fid lbllist live def) s s')
    (AGREE: agree rms rmo (live_dr_transf (ver_code v) pc (live_absstate_get pc live))),
      (match_states p v fid lbllist live def) Zero (State s v pc rms ms) (State s' v pc rmo ms)
```

In the proof of the `Framestate` insertion, a property, `match_stackframe`, expresses the fact that two stackframes are similar. A stack is basically a list of stackframes. `match_stack` then states that the stackframes of two stacks are side-by-side matched by `match_stackframe`.

When doing a Call from such states matched with `deopt_match`, the environments that are saved in the execution stack may be different. However, after returning from the call, they should still agree on the live registers. In our new proof, we had to add the following case in the definition of `match_stackframe`, used in the invariants to relate the execution stacks of the source and optimized program.

```
| frame_deopt:
  ∀ r lbl rms rmo
    (AGREE: ∀ retval, agree (rms # r ← retval) (rmo # r ← retval)
    (live_dr_transf (ver_code v) lbl (live_absstate_get lbl live))),
        (match_stackframe v fid l live def) (Stackframe r v lbl rms) (Stackframe r v lbl rmo)
```

In addition, we had to prove the following theorems, whose statements were different in the previous version:

```
Theorem defined_deopt_subset_agree:
  ∀ rm rs rs',
    defined rm (DefFlatRegset.Inj rs) →
    PositiveSet.Subset rs' rs →
      ∃ rmdeopt,
        update_regmap (identity_varmap rs') rm rmdeopt ∧
       agree rm rmdeopt rs'.

Lemma deopt_subset_agree:
  ∀ rm rs rs' newrm,
    defined rm (DefFlatRegset.Inj rs) →
    PositiveSet.Subset rs' rs →
    update_regmap (identity_varmap rs') rm newrm →
    agree rm newrm rs'.
```

These new theorems are actually weaker versions of the previous ones. They state respectively that:

- if we deoptimize from a given regmap with an identity varmap on a *subset* of its defined registers (not necessarily all of them), then the resulting regmap will *agree* on this subset with the given regmap (instead of being equivalent). Previously, `rs'` was exactly `rs` and `rm` and `rmdeopt` were equivalent.
- if two regmaps are such that deoptimizing from the first with an identity varmap on a *subset* of its defined registers result in the second, then they agree on this subset. Previously, `rs'` was exactly `rs` and `rm` and `newrm` were equivalent.

The proof of the first one follows the same ideas as in the previous `Framestate` insertion. The second one is a direct consequence of the first one.

Finally we had to consider the new `deopt_match` case in the main proof of the backward simulation. Therefore we had to prove some basic properties on `agree`, which expresses the fact that agreeing on live registers is always sufficient : we couldn't use what was proved for `regmap_eq` since regmaps were supposed to agree on live registers instead of being equivalent in this case. For instance, we had to prove the following lemma, where `expr_live` is a function that add the registers used by an expression to a set of registers :

```
(* If two regmaps agree on the registers used for the evaluation of e,
then e evaluates to the same value in both regmaps *)
Lemma agree_eval_expr:
  ∀ e v rs rms rmo ,
    agree rms rmo (expr_live e rs) →
    eval_expr e rms v →
    eval_expr e rmo v.
```

This lemma states that only the value of the registers used by an expression matters to evaluate correctly this expression. For example, only the value of `r1` and `r2` matters to evaluate `r1 + r2`.

4 Performance analysis

To assess the impact of our improvements on the performance of the JIT compiler, we used a previously implemented Bash script, which runs the JIT with a Lua frontend on a benchmark of various Lua programs, such as bubble sort, or various ways of computing the beginning of the Fibonacci sequence. One execution of this script lasts for a few seconds, therefore in order to have more reliable results we ran multiples times this script in a row.

4.1 With improved inlining

Unfortunately, for this improvement, we couldn't see any significant gain on the overall execution time with our script. Nonetheless, there is still a positive point to highlight. Indeed, we added a counter in the extracted (in OCaml) version of the Coq inlining function, and we noticed that inlining was triggered significantly more often: on one execution of our Bash script, inlining happened 21 times with our improvement while it happened only 7 times with the previous version of inlining.

4.2 With liveness analysis

However, the good news is that liveness analysis cut the execution time by nearly 8% over 30 executions of our benchmark. This confirms our intuition that reducing register pressure at speculative instructions benefits the execution time. A liveness analysis could also be useful for other works on the JIT, such as implementing a backend for SpecIR.

5 Conclusion

As a matter of conclusion, we have improved an existing JIT compiler using some popular techniques in the domain of optimization, such as a liveness analysis. Inlining is particularly difficult with speculative instructions since we have to synthesize stackframes to cope with eventual deoptimizations. Inlining is only possible under some restrictive conditions, and it is important to know when we are able to inline. Our project relaxed these conditions so as to inline more often. Furthermore, the speculative instructions that we encounter with JIT compilers allows the specialization of functions for a better efficiency. Therefore they must be compiled themselves as efficiently as possible. Our previous version of insertion of speculative instructions had an unnecessary register pressure. Thanks to our project we managed to capture an environment as minimal as possible for a correct deoptimization.

References

[1] Aurèle Barrière, Sandrine Blazy, Olivier Flückiger, David Pichardie, and Jan Vitek. Formally verified speculation and deoptimization in a JIT compiler. *Proceedings of the ACM on Programming Languages*, 5(POPL):26, January 2021.

[2] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009.