Speculative Optimizations without Fear
Correct Compiler Transformations in the Context of Assumptions

ANONYMOUS AUTHOR(S)

Just-in-time (JIT) language implementations make heavy use of speculative optimizations, based on assumptions that might become invalid. This requires the implementation to support a bailout mechanism to undo invalidated optimizations. The interaction between assumptions and optimizations is a major source of complexity for JIT implementers: how should optimizations preserve bailout information? What are the trade-offs when adding more assumptions?

We demonstrate how to reason about correctness of speculative optimization and deoptimization. We present sourir, an intermediate representation with explicit assumption checkpoints, designed to formally study speculative optimizations in JIT systems. Sourir models core difficulties of speculative optimization, such as interleaving assumptions and optimizations, and the interaction with inlining. Our formalization stays at the IR level, thus abstracting away orthogonal aspects of JIT implementations, such as dynamic code generation and self mutation. We describe a set of common optimizations (constant folding, unreachable code elimination, and function inlining) and prove them correct in presence of speculative assumptions.

CCS Concepts:
• Software and its engineering → Just-in-time compilers;

Additional Key Words and Phrases: speculative optimization, compiler correctness, deoptimization

ACM Reference format:
https://doi.org/10.1145/nnnnnn.nnnnnnn

1 INTRODUCTION
Dynamic languages pose a problem for classical compiler optimizations. Very few facts are static and the semantics often have little regard for efficiency. The dominant approach to efficiently evaluate programs in these languages is just-in-time (JIT) compilation.

A JIT compiler starts from an unoptimized representation of the program, and interleaves its execution with a profiling mechanism to detect performance-critical fragments. For those an optimized representation is compiled and executed instead. JIT compilers may perform speculative optimizations, where the code is optimized based on assumptions that may be invalidated in the future: “the traversed elements of this array were all integers” (JavaScript), “this virtual call resolves to that method” (Python), “no subclass of this class has been loaded so far” (Java), “the rebindable if identifier really binds to the conditional control primitive we expect” (R), etc. To guard an assumption a checkpoint is added, an instruction that contains a dynamic check that the assumption holds and enough information to bail out to the unoptimized version, if the guard fails.

Thanks to speculative optimization, JITs can remove the overhead of dynamic language features. For instance, efficient numerical computations are possible in JavaScript due to type specialization [Hackett and Guo 2012], even though numerical operations [ECMA International 2016, sec. 7.1.3] are notoriously complex and expensive. But JITs also require massive engineering efforts: since many of the trade-offs involved in an efficient implementation are specific to a given language,
this cost has to be paid again for each programming language. Research on the principles of JIT compilers can help develop a shared understanding and orient implementation choices.

Myreen [2010] formally verifies a JIT compiler from a simple bytecode language into x86 assembly. Many of the difficulties are covered by this impressive formalization work: dynamic code generation, in-place code mutation, as well as the interaction between a high- and a low-level language. However, this only covers the mechanism to enable just-in-time optimizations.

The formalization of speculative optimizations, and in particular the idea of bailing out of incorrect optimizations, remains unexplored. For example, there is a tension between aggressive optimization and the need to retain enough information to be able to bail out of an invalidated version. Do optimizations have to be restricted in presence of checkpoints? When is it correct to hoist a write above a checkpoint? How should checkpoints be treated by inlining?

In the present work, we introduce sourir, a language in the style of an intermediate compiler representation (IR), to study the correctness of speculative optimization and deoptimization. The presentation is based on an OCaml prototype we used to experiment with optimizations. The motivation for our work comes out of an ongoing effort to implement an optimizing JIT for the R programming language, which Morandat et al. [2012] showed to be particularly difficult to reason, analyze, and optimize.

A sourir program contains several versions that make different optimization assumptions, and checkpoint instructions that may bail out from one version into another when an assumption fails. We demonstrate that sourir can model common approaches for speculation, present several optimizations and their interaction with checkpoints, and formally prove the optimizations correct.

To focus on the correctness of speculation, we abstract away three orthogonal issues of JIT implementations. First, instead of having both a high-level language for the reference program and one or more low-level languages for the optimized fragments, we use a single IR to represent all optimization levels. Second, instead of explicitly modeling dynamic code generation and code mutation, we work on a single optimized program that is, conceptually, the unfolding of all code generation and mutation operations that have happened during the execution of a program. Finally, we do not model the JIT profiling mechanism—for correctness, we can treat it as an oracle.

As a simple example of sourir, consider the following snippet, a fragment of an unoptimized program doing a dynamic type dispatch:

\[
V_{source} \rightarrow \\
L_{before} : \ldots \\
L_{test} : \text{branch (tag = NUM) } L_{isnum} \ L_{next} \\
L_{isnum} : \ldots \\
L_{next} : \ldots \\
\]

If this branch is taken many times, the profiler may decide to speculate on it. This corresponds to creating a new version \(V_{specul}\) with a checkpoint to guard the assumption and bail out to the old version \(V_{source}\) if it fails. It is an instance of a generic transformation that we formally proved correct, and results in the following equivalent program; version \(V_{source}\) is unchanged:

\[
V_{specul} \rightarrow \\
L_{before} : \ldots \\
L_{checkpoint} : \text{assume } [(tag = NUM)] \text{ else } \langle F_{main}.V_{source}.L_{test} [tag = tag] \rangle \\
L_{test} : \text{branch (tag = NUM) } L_{isnum} \ L_{next} \\
L_{isnum} : \ldots \\
L_{next} : \ldots \\
V_{source} \rightarrow \ldots \\
\]
Finally, we defined constant folding and unreachable code elimination optimizations that would transform this into the following proved-equivalent program:

$$V_{specul} \rightarrow L_{before} : \ldots$$

$$L_{checkpoint} : \text{assume } ([\text{tag} = \text{NUM}]) \text{ else } (f_{\text{main}}.V_{\text{source}}.L_{\text{test}} [\text{tag} = \text{tag}])$$

$$L_{isnum} : \ldots$$

$$V_{\text{source}} \rightarrow \ldots$$

The branch is gone and the unreachable code at label $L_{next}$ is removed. This corresponds to the state that a JIT compiler would have reached after warm-up. The $[\text{tag} = \text{tag}]$ mapping remembers the environment necessary to bail out, and in particular prevents optimizations from discarding the $\text{tag}$ variable unless it can be statically reconstructed.

The contributions of this paper are:

- A simple intermediate representation, $\text{sourir}$, that formalizes the interaction between optimizations and speculative assumptions, while abstracting away many important but orthogonal aspects of JIT implementations.
- Correctness proofs for a set of program transformations (inserting new assumptions and hoisting assumptions), and several standard compiler optimizations (constant folding, unreachable code elimination, and function inlining), adapted to account for speculative assumptions already in the program.

This set of program transformations can serve as a toolbox for building optimization strategies out of proved-correct components. More importantly, we think of $\text{sourir}$ as a high-level specification of a JIT implementation. We hope our simple presentation can give implementers ideas for reasoning about the correctness of their own implementations.

Section 2 introduces the $\text{sourir}$ IR. Section 3 presents our program transformations: how assumptions are added, used, and maintained by optimizations. Section 4 presents formalization and correctness proofs. Section 5 discusses more advanced examples and further support for our design decisions. Section 6 discusses related work, we present ideas for future work in Section 7 and conclude in Section 8.

2 SOURIR IN A NUTSHELL

$\text{sourir}$ is an untyped bytecode-like language with lexically-scoped mutable variables and first-class functions. The two speculation-specific features are $\text{versions}$, the $\text{sourir}$ counterpart of dynamically-generated code fragments, and assumption checkpoints represented by the $\text{assume}$ instruction.

For example, in version $V_{\text{luck}}$ of Figure 1, we believe that $c$ is constant and optimize based on that assumption. If the assumption becomes invalid, we bail out to the less optimized version $V_{\text{tough}}$ by reconstructing the original environment. We discuss checkpoints in more detail in Section 2.3; for now, we introduce the more standard instructions of $\text{sourir}$.

$$F_{\text{fun}}(c) \rightarrow V_{\text{luck}} \rightarrow L_0 : \text{assume } ([c = 41]) \text{ else } (F_{\text{fun}}.V_{\text{tough}}.L_1 [c = c, o = 1])$$

$$L_1 : \text{print } 42$$

$$V_{\text{tough}} \rightarrow L_0 : \text{var } o = 1$$

$$L_1 : \text{print } (c + o)$$

Fig. 1. If we are lucky, $c$ stays constant.
2.1 Standard Instructions

The syntax of sourir instructions is shown in Figure 2. We support defining a local variable, removing a variable from scope, variable assignment, creating arrays (by length or from a list of elements), array assignment, unstructured control flow (labels $L$, conditional branches, and goto), input and output, function calls and returns, and checkpoints.

Literals (lit) are integers, booleans and nil. Together with variables and function references (functions live in a separate namespace), they form simple expressions (se). Finally, an expression (e) is either a simple expression or an operation: array access, array length, or primitive operations (arithmetic, comparison, and logic operations). The arguments of operations must be simple expressions, which forbids complex nested expressions—this is common in bytecode languages or intermediate compiler representations such as A-normal form [Sabry and Felleisen 1992].

Control flow instructions give explicit target labels, but other instructions advance to the next instruction in the sequence. The following instruction sequence reads a number $N$ and initializes an array with the sequence $0, \ldots, N - 1$.

\[
\begin{align*}
L_0 & : \text{ var } N = \text{ nil} \\
L_1 & : \text{ read } N \\
L_2 & : \text{ array } t[N] \\
L_3 & : \text{ var } k = 0 \\
L_4 & : \text{ goto } L_{\text{loop}} \\
L_{\text{loop}} & : \text{ branch } (k < N) \ L_{\text{body}} L_{\text{end}} \\
L_{\text{body}} & : t[k] \leftarrow k \\
L_5 & : k \leftarrow (k + 1) \\
L_6 & : \text{ goto } L_{\text{loop}} \\
L_{\text{end}} & : \text{ drop } k
\end{align*}
\]
A sourir program \( P \) is a set of declarations of functions. The body of a function is not just an instruction sequence \( I \), but a list of versions indexed by a version label, where each version is an instruction sequence. The first instruction sequence in the list (the active version) is executed when the function is called.

Versions model the dynamic code fragments generated by a JIT implementation. Initially, a program has just one version per function, but the optimizer will create additional versions by speculation. All versions should behave as the original function, but may have been generated from different sets of assumptions.

We use the metavariable \( F \) to range over function names, \( V \) for version labels, and \( L \) for instruction labels. An absolute reference to an instruction of the program is thus a triple \( F.V.L \). The only instruction that explicitly references versions is \( \text{assume} \), described next.

### 2.3 Assumption Checkpoints

A checkpoint instruction \( \text{assume} [e^*] \text{ else } \tilde{\xi} \bar{\xi}^* \) contains both assumptions \( (e^*) \) and bailout information. The assumptions are guarded by dynamic checks; if all assumptions hold the checkpoint behaves as a no-op, otherwise execution will bail out to a less optimized version.

The bailout information \( \tilde{\xi} ::= (F.V.L [x_1 = e_1, \ldots, x_n = e_n]) \) contains the bailout target triple and a varmap. To bail out, a fresh environment for the target scope is created according to the varmap \([x_1 = e_1, \ldots, x_n = e_n]\). Each expression \( e_i \) is evaluated in the old environment and bound to \( x_i \) in the new environment. Also, the program counter is updated to the bailout target \( F.V.L \). If multiple frames need to be created, the specification is encoded by \( \bar{\xi}^* \); we discuss this situation and inlining in Section 3.

The purpose of \( \tilde{\xi} \) is twofold. First, as described, it provides the necessary information for bailing out to the context of the target version. Second, its presence in the instruction stream allows the optimizer to keep the mapping between different versions up-to-date.

### Example

We conclude this section with an example optimization in the presence of speculative assumptions. Consider the function \( F_{\text{size}} \) in Figure 3 which computes the size of a vector, given by the argument \( \text{obj} \). In version \( V_{\text{base}} \), the input \( \text{obj} \) is either nil or an array that represents a vector object, with its length stored at index 0. The optimized version \( V_{\text{opt}} \) assumes that the input is never nil. Classical compiler optimizations can rely on this assumption, such as unreachable code removal which

![Fig. 3. A speculative optimization of size where we assume \( \text{obj} \) is not \text{nil}.](image-url)
prunes the unused $L_{\text{isnil}}$ branch. An interesting detail is how the constant folding pass also updates the mention of the local variable $SZ$ in the checkpoint. Thus, it is ensured that the target scope of the checkpoint is correct, even though the optimized environment was reduced.

3 SPECULATION AND OPTIMIZATION

In this section, we discuss how the optimizer interacts with assumption checkpoints. We explain when and how new versions are introduced in Section 3.1, then in Section 3.2 we present classical compiler optimizations extended to support assumptions. The formal proofs of the transformations in those two sections follow in Section 4. Section 3.3 introduces additional optimizations for the checkpoints, and we conclude with two case studies in Section 3.4.

The transformations in this section preserve the following invariants:

Version Equivalence Every new version of a function is observationally equivalent to the existing ones, and optimizations preserve the equivalence.

Checkpoint Invariant It is always correct to take the bailout transition more than necessary, i.e., the post-bailout state is equivalent to the pre-bailout state.

As an example in $F_{\text{fun}}$ that prints the argument $x$, $V_{\text{opt}}$ respects both invariants. On the other hand, $V_{\text{broken}}$, while being equivalent, violates the checkpoint invariant:

$$
\begin{align*}
F_{\text{fun}}(x) & \rightarrow \\
V_{\text{opt}} & \rightarrow \\
L_0 & : \ \text{assume \ [false \ else \ (}F_{\text{fun}} \cdot V_{\text{base}}.L_0 \ [x = x] \}) \\
V_{\text{broken}} & \rightarrow \\
L_0 & : \ \text{assume \ [ ] else \ (}F_{\text{fun}} \cdot V_{\text{base}}.L_0 \ [x = 42] \}) \\
L_1 & : \ \text{print \ } x \\
V_{\text{base}} & \rightarrow \\
L_0 & : \ \text{print \ } x
\end{align*}
$$

The checkpoint invariant is necessary to ensure that adding more assumptions does not alter the program semantics.

3.1 Optimization Versions

A version represents the minimal granularity for undoing optimizations. Multiple assumptions can be added to one version. We can picture a new version and the version it is based on, executing in lockstep; at every checkpoint the metadata provides a complete mapping from the new version to the base version’s state. This simulation relation between the two versions is our correctness argument. As Béra et al. [2016] show the relation can also be used for testing an implementation.

Creating a Fresh Version and Adding Checkpoints. A new version $V'$ is created from a currently active version $V$, by copying all instructions and updating the checkpoints. Existing assumptions need to be preserved, but the checkpoints are supposed to bail out to version $V$. Therefore, the bailout target is set to the identical label in $V$ and as varmap we choose the identity map for the scope at that label. For example, if the target scope contains the variables $x$ and $z$, then the mapping is $[x = x, z = z]$. Additionally, we can add new checkpoints and remove empty ones. This new version preserves the version and checkpoint invariants by construction.

As an example, in the following program we created the new version $V'$, which is a copy of $V$; the checkpoint at $L_0$ was added, the one at $L_1$ was updated, and the one at $L_3$ was removed.
Updating existing checkpoints is not a necessity for correctness. But the idea with a new version is that it captures a set of assumptions that can be undone independently from the existing assumptions. Thus, the deoptimization is supposed to bail out one version at a time. In a JIT implementation, versions might, for example, correspond to optimization tiers. This approach might lead to a cascade of deoptimizations if an early assumption fails; we discuss this situation in Section 3.3.

There is a secondary invariant regarding versions: we can only modify versions if they are not bailout targets, otherwise we might break the invariants for the incoming deoptimizee. We say that a version with incoming deoptimizations is sealed. In sourir active versions can be optimized, but all others are sealed.

Starting from version $V_{base}$ from Figure 3, we generate the following new version (for compactness we only add a subset of all possible checkpoints):

$$F_{size}(obj) \rightarrow \quad V_{opt} \rightarrow$$

$$L_{cpp0} : \text{assume } [] \text{ else } (F_{size},V_{base},L_{1}[obj = obj])$$
$$L_{cpp1} : \text{assume } [] \text{ else } (F_{size},V_{base},L_{2}[SZ = SZ, obj = obj])$$
$$L_{vector} : \text{branch } (obj = \text{nil}) L_{isnil} L_{vector}$$
$$L_{isnil} : \text{return } (len \ast SZ)$$
$$V_{base} \rightarrow \ldots$$

Adding Assumptions. Our approach to speculative optimizations is to first add an assumption and then use it for "static" optimizations. In contrast, the traditional approach is to apply an unsound optimization and then recover by adding a guard (see for example Duboscq et al. [2013]). The end result is the same, but the different perspective helps with reasoning about correctness.

Assumptions are boolean predicates, similar to user provided assertions. For example, to speculate on a branch target, the assumption is the branch condition or its negation.

Checkpoints in sourir are skipped if all assumptions evaluate to true, otherwise they bail out. It is therefore correct to assume that all conditions hold at the instruction immediately following the checkpoint; a data-flow analysis may infer that they are preserved further down.

Inserting a new checkpoint in an existing program is difficult in general, as we do not know where to bail to or how to reconstruct the target environment. On the other hand, it is always correct to add an additional assumption to an existing checkpoint: thanks to the checkpoint invariant, bailing
out more often preserves the program behavior. To introduce a new assumption, we therefore add
it to the nearest dominating checkpoint.

For instance, in \texttt{assume \{(x \neq \texttt{nil}), (x > 10)\}} \texttt{else} \ldots the assumption \texttt{(x \neq \texttt{nil})} was further
narrowed down to \texttt{(x > 10)}. We can imagine a profiler that records different measurements of the
program execution, such as types, value ranges, or branch targets, and feeds stable measurements
back to the compiler in the form of assumptions.

### 3.2 Classical Optimizations

At this point we have shown how to create new versions, establish the deoptimization invariants, and
add assumptions. The next step is to optimize a version based on assumptions, i.e., to speculatively
optimize it. The optimizations in this chapter are not exhaustive, but rather illustrative of the
approach.

**Constant Propagation.** We provide a simple constant propagation and folding pass, that finds
constant variables and then updates all uses. The optimization context of the constant propagation
pass is a mapping from variable names to constants or \texttt{unknown}. The context is computed for every
position in the instruction stream using a data-flow analysis [Kildall 1973].

For the analysis, the \texttt{update} function adds and removes constants to the optimization context.
For example \texttt{var x = 2, or x ← 2} adds the mapping \texttt{x → 2}. It also considers the contents of the
existing context. The instruction \texttt{var y = (x + 1) under the previous context adds the mapping
y → 3}. Finally, \texttt{drop x} removes a mapping. The \texttt{join} function for intersecting two contexts is
straightforward. Mappings which agree are preserved, while others are set to \texttt{unknown}.

In a second step expressions that can be evaluated statically under the optimization context are
replaced. Additionally, now unused variable declarations are removed.

No additional care needs to be taken to make this pass correct in the presence of checkpoints. In
\texttt{sourir}, expressions needed to bail out are mentioned in the varmap part of the \texttt{assume} instruction.
For example, in \texttt{assume \{ \} else (F.V.L \{ x = (y + z) \}, the mentioned variables y and z should be
treated the same as in \texttt{call h = `F_{foo}((y + z))}. They can be replaced and a checkpoint will not
artificially keep constant variables alive.

**Speculative Constant Propagation.** Constant propagation as introduced up to this point is correct
in the presence of checkpoints. As a next step, we want to extend it to be speculative.

After executing \texttt{assume \{(x = 1)\}} \texttt{else} \ldots \texttt{; var y = (x + x)}, we know that \texttt{x} must be 1 and \texttt{y}
must be 2. Therefore, we can add those facts to the static optimization context. This is the only
extension required for speculative constant propagation.

As an example, in the case where we speculate on a type check

\begin{verbatim}
L_0 : assume \{(tag = \texttt{INT})\} else . . . 
L_1 : branch \{tag = \texttt{INT}\} L_{\text{int}} L_{\text{notint}}
\end{verbatim}

the optimization context is \texttt{tag → INT} at \texttt{L_1}. Evaluating the branch condition under this context
yields \texttt{true}, and a further optimization opportunity presents itself.

**Unreachable Code Elimination.** As shown above, the right assumption coupled with speculative
constant folding leads to branches becoming deterministic. \texttt{Unreachable code elimination} as we
present it profits from that fact.

Our approach is a two step algorithm. The first pass replaces \texttt{branch e L_1 L_2} with \texttt{goto L_1} if \texttt{e}
is a tautology and with \texttt{goto L_2} if \texttt{e} is a contradiction. Then the second pass removes unreachable
instructions.
In our running example from Figure 3, we chose to add the assumption \((\text{obj} \neq \text{nil})\) right before the branch. The *speculative constant propagation* pass "statically" proved that the branch takes the \(L_{\text{vector}}\) target, *unreachable code elimination* removed the dead \(L_{\text{isnil}}\) target, and vanilla *constant propagation* removed \(SZ\), by also replacing its mention in the varmap of the checkpoint:

\[
F_{\text{size}}(\text{obj}) \rightarrow \\
V_{\text{opt}} \rightarrow \\
L_{\text{cp}_0} : \text{assume } [\text{else } (F_{\text{size}}V_{\text{base}.L_1}[\text{obj} = \text{obj}]) \\
L_{\text{cp}_1} : \text{assume } [(\text{obj} \neq \text{nil}) \text{ else } (F_{\text{size}}V_{\text{base}.L_2}[SZ = 32, \text{obj} = \text{obj}])] \\
L_{\text{vec}} : \text{var len = obj}[0] \\
L_3 : \text{return } (\text{len} \ast 32) \\
V_{\text{base}} \rightarrow \ldots
\]

Only the slice for the \(L_{\text{vector}}\) case survives in the optimized version.

**Inlining.**Inlining is the most involved optimization, since checkpoints inherited from the inlinee need to stay correct. The inlining itself is standard, using name mangling to separate the caller and callee environments.

The following example shows an inlining of the above function \(F_{\text{size}}\) into a function \(F_{\text{main}}:\n
\[
F_{\text{main}}() \rightarrow \\
V_{\text{inlined}} \rightarrow \\
L_0 : \text{array } pl = [1, 2, 3, 4] \\
L_1 : \text{array } vec = [\text{length}(pl), pl] \\
L_2 : \text{var } res = \text{nil} \\
L_3 : \text{var } obj = \text{vec} \\
L_{\text{cp}_1} : \text{assume } [(\text{obj} \neq \text{nil}) \text{ else } \ldots] \\
L_5 : \text{var len = obj}[0] \\
L_6 : \text{res } \leftarrow (\text{len} \ast 32) \\
L_7 : \text{drop } len \\
L_8 : \text{drop } obj \\
L_9 : \text{var } size = \text{res} \\
L_{10} : \text{drop } res \\
L_{\text{ret}} : \text{print } size \\
V_{\text{base}} \rightarrow \ldots
\]

\[
F_{\text{main}}() \rightarrow \\
V_{\text{base}} \rightarrow \\
L_0 : \text{array } pl = [1, 2, 3, 4] \\
L_1 : \text{array } vec = [\text{length}(pl), pl] \\
L_2 : \text{call } size = 'F_{\text{size}}(vec) \\
L_{\text{ret}} : \text{print } size \\
F_{\text{size}}(\text{obj}) \rightarrow \\
V_{\text{opt}} \rightarrow \\
L_{\text{cp}_1} : \text{assume } [(\text{obj} \neq \text{nil}) \text{ else } \ldots] \\
L_{\text{vec}} : \text{var len = obj}[0] \\
L_3 : \text{return } (\text{len} \ast 32) \\
V_{\text{base}} \rightarrow \ldots
\]

The function \(F_{\text{size}}\) is speculatively optimized. Naively inlining this version without updating the checkpoints will result in an incorrect transition when the inherited guard triggers. The unabbreviated checkpoint from \(F_{\text{size}}\) reads:

\[
L_{\text{cp}_1} : \text{assume } [(\text{obj} \neq \text{nil}) \text{ else } (F_{\text{size}}V_{\text{base}.L_2}[SZ = 32, \text{obj} = \text{obj}])]
\]

Using this checkpoint to bail out from \(V_{\text{inlined}}\), we would jump to \(F_{\text{size}}V_{\text{base}.L_2}\) with no continuation on the stack. However, version \(V_{\text{base}}\) of function \(F_{\text{size}}\) certainly expects to be able to return to \(F_{\text{main}}\). Also, \(F_{\text{main}}\)’s part of the environment is discarded by the bailout and permanently lost.

We solve the issue by extending the checkpoint with an additional list of frames to synthesize. In our example, we would append \((F_{\text{main}}V_{\text{base}.L_{\text{ret}} size}[pl = pl, vec = vec])\). This creates an additional stack frame that returns to \(F_{\text{main}}\) after the call, with the entire caller portion of the environment reconstructed.
442 \( F_{\text{main}}() \rightarrow \)
443 \( V_{\text{inline}} \rightarrow \)
444 \( L_0 : \ldots \)
445 \( \text{assume } [(\text{obj} \neq \text{nil})] \text{ else } \langle F_{\text{size}}.V_{\text{base}}.L_2 [\text{SZ} = 32, \text{obj} = \text{obj}] \rangle \)
446 \( \langle F_{\text{main}}.V_{\text{base}}.L_{\text{ret}} \text{ size } [\text{pl} = \text{pl}, \text{vec} = \text{vec}] \rangle \)
447 \( L_6 : \ldots \)
448 \( V_{\text{base}} \rightarrow \ldots \)

Overall, after a bailout, it appears as if version \( V_{\text{base}} \) of \( F_{\text{main}} \) had called version \( V_{\text{base}} \) of \( F_{\text{size}} \). Multiple continuations can be added at the end for further levels of inlining. The inlining needs to be applied bottom up: for the next level of inlining, e.g., to inline \( V_{\text{inline}} \) into an outer caller, renamings must also be applied to the extra continuation frames, since they refer to local variables in \( V_{\text{inline}} \).

### 3.3 Optimizing the Assumptions

So far, we have shown some classical optimizations, how they interact with the checkpoints, and how they can consume assumptions. We want to end this exploration with two examples of optimizations specifically geared towards checkpoints.

**Hoisting Assumptions.** A checkpoint represents non-local control flow and the idea is to “forget” about the version at the bailout target. This means we have to be careful with instruction reordering. For example, hoisting a side-effecting instruction over a checkpoint is invalid, because if we bail out the effect happens twice. Removing a local variable is equally not possible if its value is needed to reconstruct the target environment.

As a simple alternative to moving checkpoints, we can hoist assumptions. To understand why, let us decompose the approach in two steps. Given a checkpoint \( a \) that dominates a second checkpoint \( b \), we copy an assumption from \( b \) to \( a \). This is valid since the checkpoint invariant allows adding more assumptions. Then, a data-flow analysis can determine if the new assumption from \( a \) is available at \( b \). In case the intermediate instructions do not interfere with it, the assumption can be removed from \( b \).

In our running example, we can see that the last optimized version \( V_{\text{opt}} \) on page 8 has two checkpoints next to each other and one assumption. We can trivially hoist the assumption \((\text{obj} \neq \text{nil})\), since there are no intermediate instructions, thus the checkpoint with the larger scope could be removed. More interestingly, in the case of a loop-invariant assumption, we can hoist the assumption out of the loop.

**Composing Checkpoints.** As we have argued in Section 3.1, it is beneficial to bail out to the previous version and undo as few assumptions as possible. On the other hand, bailing on an assumption added in an early version cascades through all the later versions. To regain both advantages, we show that checkpoints are *composable*.

If a checkpoint \( a \) bails out to a target \( b \) that is a checkpoint bailing to \( c \), then we can combine the bailout metadata to take both steps at once. Due to the checkpoint invariant, the state at \( a \) is equivalent to the after-bailout state at \( b \), which is equivalent to the after-bailout state at \( c \). Even if the assumptions are not the same, it is safe to trigger the second checkpoint too.

For example, the checkpoint \( \text{assume } [e_2] \text{ else } \langle F.V_1.L \ [x = 1] \rangle \) that bails out to the checkpoint \( L: \text{assume } [e_1] \text{ else } \langle F.V_0.L \ [y = x] \rangle \) can be combined as \( \text{assume } [e_2] \text{ else } \langle F.V_0.L \ [y = 1] \rangle \). This new unified checkpoint skips the intermediate version and bails out to \( V_0 \) directly.

This is an interesting property for multi-tier JITs: after the system stabilizes, the intermediate versions are rarely used and we might want to discard them.
3.4 Case Studies

We conclude this section with two example program optimizations to demonstrate the kind of transformations possible.

Type Specialization. In dynamic languages such as JavaScript and R, the lack of static type information means that code often dispatches based on runtime type information. If the type information were known, then the code could be specialized to a specific type, resulting in faster code with fewer checks and branches.

As an example, consider implementing a generic binary addition function in 

\[
\text{add}(x, y) \rightarrow \\
\text{branch}(x = \text{nil}) \ L_{\text{error}}, L_{\text{nil}} \\
\text{branch}(y = \text{nil}) \ L_{\text{error}}, L_{\text{int}} \\
\text{var} \ x_t = x[0] \\
\text{var} \ y_t = y[0] \\
\text{branch}(x_t \neq \text{INTTPE}) \ L_{\text{error}}, L_{\text{int}} \\
\text{branch}(y_t \neq \text{INTTPE}) \ L_{\text{error}}, L_{\text{ok}} \\
\text{x} = \ldots
\]

In this example we represent objects as arrays, which contain a dynamic type tag at the first index. The function takes two variables, \(x\) and \(y\). No static information is available; they could be integer objects, objects of some other type, or even nil values. Therefore, multiple checks need to happen before the addition. In this implementation, four separate dynamic checks are required.

Suppose we have profiling information that indicates we never take an error branch. We specialize the function for non-nil integer objects, by speculatively pruning the branches:

\[
\text{add}(x, y) \rightarrow \\
\text{branch}(x = \text{nil}) \ L_{\text{error}}, L_{\text{nil}} \\
\text{branch}(y = \text{nil}) \ L_{\text{error}}, L_{\text{int}} \\
\text{var} \ x_t = x[0] \\
\text{var} \ y_t = y[0] \\
\text{var} \ z_v = (x_v + y_v) \\
\text{call} \ z = \text{\texttt{Fboxint}}(z_v) \\
\text{return} \ z
\]

The optimized code is shorter and contains no branches. As opposed to version \(V_{\text{base}}\), the function now has one exit point and one result value \(z\). As such, it is a much better candidate for function inlining.

Reordering Checks. In certain cases, sourir’s transformations can make it appear as though checks have been reordered. Consider a variation of the previous example, where we want to add two non-nil, nonzero integers:

\[
\text{add}(x, y) \rightarrow \\
\text{assume} \ [(x \neq \text{nil}), (y \neq \text{nil})] \text{ else } \langle F_{\text{add}} \cdot V_{\text{base}}, L_{\text{1}} [x = x, y = y] \rangle \\
\text{var} \ x_t = x[0] \\
\text{var} \ y_t = y[0] \\
\text{assume} \ [(x_t = \text{INTTPE}), (y_t = \text{INTTPE})] \text{ else } \langle F_{\text{add}} \cdot V_{\text{base}}, L_{\text{2}} [x = x, \ldots] \rangle \\
\text{var} \ x_v = x[1] \\
\text{var} \ y_v = y[1] \\
\text{var} \ z_v = (x_v + y_v) \\
\text{var} \ z = \text{\texttt{Fboxint}}(z_v) \\
\text{return} \ z
\]
\[ F_{\text{add}}(x, y) \rightarrow \]
\[ V_{\text{base}} \rightarrow \]
\[ L_{\text{nil}, x} : \] \text{branch} \((x = \text{nil}) L_{\text{error}, L_{\text{nil}, y}} \]
\[ L_{\text{nil}, y} : \] \text{branch} \((y = \text{nil}) L_{\text{error}, L_{\text{zero}, x}} \]
\[ L_{\text{zero}, x} : \] \text{branch} \((x = 0) L_{\text{error}, L_{\text{zero}, y}} \]
\[ L_{\text{zero}, y} : \] \text{branch} \((y = 0) L_{\text{error}, L_{\text{ok}}} \]
\[ L_{\text{ok}} : \] \text{return} \((x + y) \]
\[ L_{\text{error}} : \ldots \]

Suppose we are only able to speculate on \(x\), but not on \(y\). The assumption \((y \neq 0)\) can be hoisted and combined with the assumption \((y \neq \text{nil})\):

\[ F_{\text{add}}(x, y) \rightarrow \]
\[ V_{\text{opt}} \rightarrow \]
\[ L_{\text{cp}_0} : \] \text{assume} \([y \neq \text{nil}), (y \neq 0)] \text{else} (F_{\text{add}} V_{\text{base}} L_{\text{nil}, x} [x = x, y = y]) \]
\[ L_{\text{nil}, x} : \] \text{branch} \((x = \text{nil}) L_{\text{error}, L_{\text{nil}, y}} \]
\[ L_{\text{zero}, x} : \] \text{branch} \((x = 0) L_{\text{error}, L_{\text{ok}}} \]
\[ L_{\text{ok}} : \] \text{return} \((x + y) \]
\[ L_{\text{error}} : \ldots \]
\[ V_{\text{base}} \rightarrow \ldots \]

In the optimized version, we perform both checks on \(y\) first and then the ones on \(x\), whereas in the unoptimized version they are interleaved. By ruling out an exception early, it is possible to perform the checks in a more efficient order.

4 CORRECTNESS

In this section we give a small-step operational semantics for sourir and prove correctness of the program transformations presented in Section 3.

4.1 Valid sourir programs

As introduced in Section 2, a sourir program contains several functions, each of which can have multiple versions. This high-level structure is described in Figure 4. The first version is considered the currently active version and will be picked by the semantics on a call. Each version consists of a stream of labeled instructions. The grammar for instructions is shown in Section 2 in Figure 2.

\[
\begin{align*}
P & ::= (F(x^*) \rightarrow D_F)^* & \text{program: a list of named functions} \\
D_F & ::= (V \rightarrow I)^* & \text{function definition: a list of versioned instruction streams} \\
I & ::= (L : i)^* & \text{instruction stream with labeled instructions}
\end{align*}
\]

Fig. 4. Sourir program syntax.

Besides grammatical and scoping validity, we impose some well-formedness requirements for sourir programs to ease analysis and reasoning.

Single declaration. We forbid having two variable declarations for the same name in a given instruction stream \(I\). This restriction simplifies reasoning, by letting us use variable names to non-ambiguously track information depending on the declaration site. Note that two different versions have separate scopes and can have names in common.
An operational semantics is a formal description of the execution behavior of a programming language. It is executable, so it can be directly implemented; one can also use it to specify the behavior expected of a more efficient implementation.

In Figure 5 we first define the semantics of sourir expressions. The evaluation of an expression \( e \) returns a value \( v \), which may be either a literal \( \text{lit} \), a function, or if the expression returns an array, an address \( a \). Arrays are represented by addresses into a shared memory \( M \), formalized as a mapping from addresses to blocks of \( n \) values \( [v_1, \ldots, v_n] \). To evaluate an expression, one needs to know the current memory, and the lexical environment \( E \), that is a mapping from the variables of the current scope to their values. Evaluation is defined by a relation \( M E e \rightarrow v \) under memory \( M \) and environment \( E \), the expression \( e \) evaluates to the value \( v \). This definition in term relies on a relation \( E se \rightarrow v \) defining evaluation of simple expressions \( se \), which need not access the array memory.

The evaluation rules are standard. We use the notation \([\text{primop}]\) to denote, for each primitive operation \( \text{primop} \), a partial function (at the meta-level) on values. Arithmetic operators and arithmetic comparison operators are only defined when their arguments are numbers. Equality and inequality are defined for all values.

Validity of function reference. If a function reference \( 'F \) is used in the program source, we require that the function \( F \) indeed exists in the program.

Scoping discipline. We ask that all control-flow transitions going to the same instruction have the same scope, the same set of currently declared variables. When jumping to a label \( L \), this requires removing all variables (drop \( x \)) that were not in scope at \( L \). This restriction simplifies reasoning by making it trivial to determine the lexical environment at any point in the program.

We found that it is fairly natural to respect this restriction.

\[
\begin{align*}
L_{\text{loop}} : & \quad \text{var } x = e \\
L_{\text{body}} : & \quad \text{print } x \\
L_{\text{test}} : & \quad \text{branch } e' L_{\text{loop}} L_{\text{end}} \\
L_{\text{end}} : & \quad \ldots \\
L_{\text{loop}} : & \quad \text{var } x = e \\
L_{\text{body}} : & \quad \text{print } x \\
L_{\text{test}} : & \quad \text{branch } e' L_{\text{continue}} L_{\text{end}} \\
L_{\text{continue}} : & \quad \text{drop } x \\
L_{\text{continue1}} : & \quad \text{goto } L_{\text{loop}} \\
L_{\text{end}} : & \quad \ldots 
\end{align*}
\]

The version on the left is invalid because the branch instruction has \( x \) in scope when it jumps to \( L_{\text{loop}} \), where \( x \) does not exist. Instead we write the equivalent program on the right.

4.3 Operational semantics: instructions and programs

For sourir programs we define a small-step, labeled operational semantics by defining a notion of machine state, or configuration, that represents the dynamic state of a sourir program being executed, and a reduction relation, a transition relation between configurations that specifies the execution of a single instruction. A sourir configuration is a six-component tuple \( \langle P I L K^* M E \rangle \) described in Figure 6.

Call continuations \( K \) are tuples of the form \( \langle I L x E \rangle \), storing the information needed to correctly return to a caller function. On a call instruction \text{call } x = se(se_1, \ldots, se_n), the continuation pushed
\[ v ::= \text{values} \]
\[ | \text{lit} \]
\[ | F \]
\[ | \text{addr} \]
\[ M ::= (a \rightarrow [v_1, \ldots, v_n])^* \quad \text{shared memory} \]
\[ E ::= (x \rightarrow v)^* \quad \text{lexical environment} \]

\[
\begin{array}{lll}
\text{[LITERAL]} & \text{[FUNREF]} & \text{[LOOKUP]} \\
E \text{\textit{lit}} \rightarrow \text{\textit{lit}} & E' \text{\textit{F}} \rightarrow F & E x \rightarrow E(x) \\
\end{array}
\]

\[
\begin{array}{l}
\text{[V\text{ECLEN}]} \\
E \text{\textit{se}} \rightarrow a & M(a) = [v_1, \ldots, v_n] \\
M E \text{\textit{length(se)}} \rightarrow n & E \text{\textit{se}} \rightarrow n \quad 0 \leq n \leq m \\
\end{array}
\]

\[
\begin{array}{l}
\text{[PR\text{IMOP}]} \\
E \text{\textit{se}}_1 \rightarrow v_1 \quad ... \quad E \text{\textit{se}}_n \rightarrow v_n \\
M E \text{\textit{primop(se}_1, \ldots, \text{se}_n) \rightarrow \text{\textit{primop}}(v_1, \ldots, v_n)} \\
\end{array}
\]

Fig. 5. Evaluation \( M E e \rightarrow v \) of expressions and \( E \text{\textit{se}} \rightarrow v \) of simple expressions.

on the stack contains the current lexical environment \( E \) and instruction stream \( I \) (to be restored on return), the label \( L \) of the next instruction after the call (the return label), and the variable \( x \) to name the returned result. For the details, see the reduction rules for \textbf{call} and \textbf{return} in Figure 8.

\[
\begin{array}{l}
\text{configuration} \\
P \quad \text{running program} \\
I \quad \text{current instruction stream} \\
L \quad \text{next instruction label} \\
K^* ::= (K_1, \ldots, K_n) \quad \text{call stack} \\
M \quad \text{array memory} \\
E \quad \text{lexical environment} \\
\end{array}
\]

\[
\begin{array}{l}
\text{call continuation} \\
I \quad \text{code of the calling function} \\
L \quad \text{return label} \\
x \quad \text{return variable} \\
E \quad \text{lexical environment at the call site} \\
\end{array}
\]

Fig. 6. Sourir abstract machine state

Our \textit{reduction relation} \( C \xrightarrow{A_{\tau}} C' \) specifies that executing the next instruction of \( C \) may result in the configuration \( C' \). The \textit{action} \( A_{\tau} \) is either an input/output action \textbf{read lit} or \textbf{print lit}, or a \textit{silent} label, when no input/output happened, traditionally written \( \tau \) ("tau").

We write \( C \xrightarrow{T} C' \) where there are zero or more steps from \( C \) to \( C' \). The reduction \textit{trace} \( T \) is a list of non-silent actions, collected in the order in which they appeared. For example, if we have

\[
\begin{array}{l}
C_1 \xrightarrow{\text{read} 1} C_2 \xrightarrow{\tau} C_3 \xrightarrow{\text{print} 2} C_4, \text{then we have} \ C_1 \xrightarrow{\text{read} 1} C_2 \xrightarrow{\tau} C_3 \xrightarrow{\text{print} 2} C_4. \\
\end{array}
\]

Actions are defined in Figure 7, and the full reduction relation is given in Figure 8.

Most reduction rules read the current instruction \( I(L) \), perform an operation, and advance to the next label in the stream. The shorthand \( (L+1) \) refers to the next label.
We use the standard proof technique of weak bisimulation to prove equivalence between source configurations. The idea is to define, for each program transformation we want to prove correct, a correspondence relation \( R \) between configurations over the source program and configurations over the transformed program. We show that related configurations will behave in the same way: they have the same observable behavior (they can perform the same input/output actions), and reducing them results in configurations that are themselves related. Two programs are equivalent if their starting configurations are related.

**Definition 4.1 (Weak Bisimulation).** Given two programs \( P_1 \) and \( P_2 \) and a relation \( R \) between the configurations of \( P_1 \) and of \( P_2 \), that \( R \) is a weak simulation if for any related states \( (C_1, C_2) \in R \) and any reduction \( C_1 \xrightarrow{A_{r_1}^*} C'_1 \) over \( P_1 \), there exists a reduction \( C_2 \xrightarrow{A_{r_2}^*} C'_2 \) over \( P_2 \) such that \( (C'_1, C'_2) \) are themselves related by \( R \).

Notice that the reduction over \( P_2 \) is allowed to take zero, one or several steps, but not to change the trace: the extra steps may only be silent transitions.

![Fig. 7. Actions and traces](image-url)
\[ I(L) = \text{var} \ x = e \ M \ E \ e \rightarrow v \quad \text{[DECL]} \quad \begin{aligned} I(L) &= \text{drop} \ x \ M(L) \ E \rightarrow (P\ I\ (L+1)\ K^*\ M\ E[x \leftarrow v]) \end{aligned} \]

\[ I(L) = \text{array} \ x = [e_1, \ldots, e_n] \ M \ E \ e_1 \rightarrow v_1 \ldots M \ E \ e_n \rightarrow v_n \quad \text{[ARRAY]} \quad \begin{aligned} a \ \text{fresh} &\quad M' \overset{\text{def}}{=} M[a \leftarrow [v_1, \ldots, v_n]] \end{aligned} \]

\[ I(L) = [\text{read} \ x \rightarrow (P\ I\ (L+1)\ K^*\ M'\ E[x \leftarrow a])] \quad \text{[READ]} \]

\[ I(L) = \text{read} \ x \quad \begin{aligned} \text{read}\_\text{lit} &\quad \rightarrow (P\ I\ (L+1)\ K^*\ M\ E[x \leftarrow \text{lit}]) \end{aligned} \quad \text{[READ]} \]

\[ I(L) = \text{print} \ e \quad \begin{aligned} \text{print}\_\text{lit} &\quad \rightarrow (P\ I\ (L+1)\ K^*\ M'\ E) \end{aligned} \quad \text{[READ]} \]

\[ I(L) = \text{goto} \ L' \quad \begin{aligned} \text{goto}\_\text{lit} &\quad \rightarrow (P\ I\ L'\ K^*\ M\ E) \end{aligned} \quad \text{[READ]} \]

\[ I(L) = \text{branch} e \ L_1 \ L_2 \quad \begin{aligned} \text{branch}\_\text{lit} &\quad \rightarrow (P\ I\ L_1\ K^*\ M\ E) \end{aligned} \quad \text{[READ]} \]

\[ I(L) = \text{call} \ x = e(e_1, \ldots, e_n) \quad \begin{aligned} \text{call}\_\text{lit} &\quad \rightarrow (P\ I\ L'\ K^*\ M\ E) \end{aligned} \quad \text{[READ]} \]

\[ I(L) = \text{assume} \ \xi = \xi' \ 
\forall m, M \ E \ e_m \rightarrow \text{true} \quad \begin{aligned} \text{assume}\_\text{lit} &\quad \rightarrow (P\ I\ (L+1)\ K^*\ M\ E) \end{aligned} \quad \text{[READ]} \]

\[ I(L) = \text{return} \ e \quad \begin{aligned} \text{return}\_\text{lit} &\quad \rightarrow (P\ I\ (L+1)\ K^*\ M'\ E[x \leftarrow v]) \end{aligned} \quad \text{[READ]} \]

\[ I(L) = \text{bailout}((P\ I\ L'\ K^*\ M, \xi, \xi')) \quad \begin{aligned} \text{bailout} &\quad \rightarrow \text{bailout}((P\ I\ L'\ K^*\ M, \xi, \xi')) \end{aligned} \quad \text{[READ]} \]

\[ M \ E \ e_1 \rightarrow v_1 \ldots M \ E \ e_n \rightarrow v_n \quad \begin{aligned} \text{EVAL\_ENV} &\quad \rightarrow (P\ I\ (L+1)\ K^*\ M[x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n]) \end{aligned} \quad \text{[READ]} \]

Fig. 8. Reduction relation \[ \overset{r}{\rightarrow} \] for sourir IR. Syntax from Figure 2 and Figure 6.
In other words, the diagram on the left below can always be completed into the diagram on the right.

We say that \( R \) is a weak bisimulation if it is a weak simulation and the symmetric relation \( R^{-1} \) also is. Finally, we say that two configurations are weakly bisimilar if there exists a bisimulation up to expansion \( R \) that relates them.

In the rest of this document we may omit the adjective weak, but it is always implied.

The following result is standard, and essential to compose the correctness proof of subsequent transformation passes.

**Lemma 4.2 (Transitivity).** If \( R_{12} \) is a weak bisimulation between \( P_1 \) and \( P_2 \), and \( R_{13} \) is a weak bisimulation between \( P_2 \) and \( P_3 \), then the composed relation \( R_{13} \) is a weak bisimulation between \( P_1 \) and \( P_3 \).

**Definition 4.3 (Version bisimilarity).** Let \( V_1, V_2 \) be two versions of a function \( F \) in a program \( P \), and let \( I_1 \) and \( I_2 \) be two instructions. We say that \( V_1 \) and \( V_2 \) are (weakly) bisimilar if \( \langle P, I_1 \rangle \) and \( \langle P, I_2 \rangle \) are weakly bisimilar for any \( K^*, M, E \).

**Definition 4.4 (Program equivalence).** We say that \( P_1 \) and \( P_2 \) are equivalent if \( \text{Start}(P_1) \) and \( \text{Start}(P_2) \) are weakly bisimilar.

### 4.5 Deoptimization invariants, formalized

We can now give a formal definition of the two invariants of Section 3.

**Version Equivalence** Any two versions \((V_1, V_2)\) of a function \( F \) are bisimilar as defined in 4.3.

**Checkpoint Invariant** For any configuration \( C \) at an instruction of the form \( \text{assume} \ [e^*] \ \text{else} \ [\xi] \ [\bar{\xi}] \), \( C \) is bisimilar to bailout(\( C, \xi, \bar{\xi}^* \)).

### 4.6 Creation of new versions, checkpoints and assumptions

**Definition 4.5.** We say that the configuration \( C \) is over the location \( F.V.L \) when it is of the form \( \langle P, P(F, V) \ L K^* \ M E \rangle \). In that case we write \( C[F.V.L \leftarrow F', V', L'] \) for the configuration \( \langle P, P(F', V') \ L' K^* \ M E \rangle \).

More generally, we use the notation \( C[X \leftarrow Y] \) to replace various components of \( C \). For example, \( C[P_1 \leftarrow P_2] \) updates the program component of \( C \); if only the versions change between two locations \( F.V.L \) and \( F.V'.L \), we may write \( C[V \leftarrow V'] \) instead of repeating the locations, etc.

**Theorem 4.6.** Creating a new copy of the currently active version of a function, possibly adding new checkpoints (see Section 3.1), returns an equivalent program.

**Lemma 4.7.** Adding a new assumption \( e' \) to an existing checkpoint \( \text{assume} \ [e^*] \ \text{else} \ [\xi] \ [\bar{\xi}^*] \) of \( P_1 \) returns an equivalent program \( P_2 \).

The proofs can be found in Section A.1 in the appendix.
4.7 Classical optimizations

The proof of the classical optimizations are sensibly easier than the transformations above (although they seem, as program transformations, to be more elaborate). This comes from the fact that they rewrite a version on the fly, instead of introducing a new version that is related to an older version, and interact little with the bailout mechanisms.

Constant propagation.

Definition 4.8. Given a program version \( V \), a static environment \( SE \) for the label \( L \) is a mapping from a subset of the variables in scope at \( L \) to values. A static environment is valid, written \( SE \vdash L \), if for any configuration \( C \) over \( L \) reachable from the starting label of \( V \) we have that \( SE \) is a subset of the lexical environment \( E \) of \( C \) – they agree on all variables on which \( SE \) is defined.

Our implementation of constant propagation uses a classic work-queue data-flow algorithm to compute a valid static environment \( SE \) at each label \( L \). It then replaces, in the instruction at label \( L \), each expression or simple expression that can be evaluated in \( SE \) (all the variable it uses are statically known) by its value.

This constant propagation is speculative in the sense that assumptions of the form \( x = \text{lit} \) populate the static environment with the binding \( x \rightarrow \text{lit} \). In general, a richer abstract domain may be used to store constraints on values rather than just equalities, but this would not change the shape of the following correctness argument.

Lemma 4.9. For any program version \( V_1 \), let \( V_2 \) be the result of constant propagation. \( V_1 \) and \( V_2 \) are bisimilar.

Proof. As in the proof of creation of a new version, we use the notational convention that \( C'_1 \) is always over \( P_1 \) and \( C'_2 \) is always \( C'_1[V_1 \leftarrow V_2] \), etc.

The relation \( R \) to use here for bisimulation is the one that relates each reachable \( C_1 \) in \( \text{Reachable}(P_1) \) to the corresponding state \( C_2 \) in \( \text{Reachable}(P_2) \) — following our notational convention, \( C_2 \overset{\text{def}}{=} C_1[V_1 \leftarrow V_2] \).

Consider two related \( C_1, C_2 \) over the label \( L \), and \( SE \) be the valid static environment at \( L \) inferred by our constant propagation algorithm. Reducing the next instruction of \( C_1 \) and \( C_2 \) will produce the same result, given that they only differ by substitutions of subexpressions by values that are valid under the static environment \( SE \), and thus under their lexical environment \( E \). If \( C_1 \overset{A_{S}}{\rightarrow} C'_1 \) then \( C_2 \overset{A_{S}}{\rightarrow} C'_2 \), and conversely. \( \square \)

Note that, in this case, the restriction of our bisimulation \( R \) to reachable configurations introduced is crucial for the proof to work. Indeed, a configuration that is not reachable may not respect the static environment \( SE \). Consider the following example, with \( V_1 \) on the left and \( V_2 \) on the right.

\[
\begin{align*}
L_0 & : \text{var } x = 1 & L_0 & : \text{var } x = 1 \\
L_1 & : \text{print } (x + x) & L_1 & : \text{print } 2 \\
L_2 & : \text{return } 3 & L_2 & : \text{return } 3
\end{align*}
\]

Now consider a pair of configurations at label \( L_1 \) with the binding \( x \rightarrow L_0 \) in the lexical environment.

\[
\begin{align*}
C_1 & \overset{\text{def}}{=} \langle P_1 P_2(F, V_1) L_1 K^* M [x \rightarrow 0] \rangle & C_2 & \overset{\text{def}}{=} \langle P_2 P_2(F, V_2) L_1 K^* M [x \rightarrow 0] \rangle
\end{align*}
\]

They are related by the relation \( R \) used by the proof, yet they are not bisimilar: we have \( C_1 \overset{\text{print}_0}{\rightarrow} C'_1 \) as the only transition of \( C_1 \) in \( V_1 \), and \( C_2 \overset{\text{print}_2}{\rightarrow} C'_2 \) as the only transition of \( C_2 \) in \( V_2 \).
Unreachable code elimination.

**Lemma 4.10.** Replacing \( \text{branch true } L_1 L_2 \) by \( \text{goto } L_1 \) or \( \text{branch false } L_1 L_2 \) by \( \text{goto } L_2 \) results in an equivalent program version.

**Lemma 4.11.** Removing an unreachable label results in an equivalent program version.

In those two cases the correctness proof is trivial: the simple version-change mapping between configurations on the two version is clearly a bisimulation. In the first case, this comes from the case that \( \text{branch true } L_1 L_2 \) and \( \text{goto } L_1 \) reduce in the example same way. In the second case unreachable configurations are not even considered by the proof.

**Inlining.** In this section, we assume that the function \( F \) has active version \( V_{\text{callee}} \). If the new version contains a direct call \( \text{call } x = 'F(s_1, .., s_n) \) to \( F \) with return label \( L_0 \) (the label after the call), our inlining pass removes the this call instruction and instead:

* it declares a mutable return variable \( \text{var } x = \text{nil} \)
* for the formal variables \( x_1, .., x_n \) of \( F \), it defines the argument variables \( \text{var } x_1 = s_1, .. \text{var } x_n = s_n \).
* then it inserts the instruction stream from \( V_{\text{callee}} \), replacing each instruction \( \text{return } e \) by the sequence

\[
\begin{align*}
L_0 & : x \leftarrow e \\
L_{\text{envStart}} & : \text{drop } x_1 \\
\ldots & : \\
L_{\text{envStop}} & : \text{drop } x_n \\
L_{\text{ret}} & : \text{goto } L
\end{align*}
\]

**Theorem 4.12.** The inlining transformation presented in Section 3.2 returns a version equivalent to the caller version.

**Proof.** The key idea of the proof is that any lexical environment \( E \) in the inlined instruction stream can be split into two disjoint parts: a lexical environment corresponding to the caller function, \( E_{\text{caller}} \), and a lexical environment corresponding to the callee, \( E_{\text{callee}} \).

To build the bisimulation, we relate the inlined version, on one hand, with the callee on the other hand, when the callee was called by the called at the inlined call point. This takes two form:

* If a configuration is currently executing in the callee, and has the caller on the top of the call stack with the expected return address, we relate it to a configuration in the inlined version (at the same position in the callee). The lexical environment of the inlined version is exactly the union of the callee environment (the environment of the configuration) and the caller environment (found on the call stack).
* If the call stack contains a caller frame above a callee frame, we relate this to a single frame in the inlined version; again, there is a bidirectional correspondence between inlined environment and a pair of a caller and callee environment.

To check that this relation is a bisimulation, there are three sorts of interesting case:

* If a transition is purely within the callee’s code on one side, and within the inlined version of the callee on the other, it suffices to check that the environment decomposition is preserved. During the execution of inlinee, \( E_{\text{caller}} \) never changes, given that the instruction coming from the callee do not have the caller’s variable in scope – and thus cannot mutate them.
* If the transition is a call of the callee from the caller on one side, and the entry into the declaration of the return variable \( \text{var } x = \text{nil} \) on the other, we step through the silent transitions that bind the call parameters \( \text{var } x_1 = e_1, .. \text{var } x_n = e_n \) and get to a state in the inlined function corresponding to the start label of the callee.
If the transition is a `return` of the callee to the caller on one side, and the entry into the result assignment \( x \leftarrow e \) on the other, we similarly step through the `drop` \( x \) for each \( x \) in the callee’s environment, and get to related state on the label \( L \) following the function call.

4.8 Optimizing the Assumptions

Section 3.3 introduces two optimizations regarding the assumption checkpoints. Hoisting assumptions and composing bailout metadata.

**Hoisting Assumptions.** Hoisting assumptions takes a version \( V_1 \), an expression \( e \) and two labels \( L_1, L_2 \), such that the instruction at \( L_1, L_2 \) are both checkpoints and \( e \) is an assumption of the checkpoint at \( L_1 \). The pass copies the assumption \( e \) from \( L_1 \) to \( L_2 \), if all variables mentioned in \( e \) are in scope at \( L_2 \). If, after this step the assumption \( e \) can be constant folded to `true` at \( L_1 \), by the optimization from Section 3.2, then it is removed from \( L_1 \), otherwise the whole version stays unchanged.

**Lemma 4.13.** Let \( V_2 \) be the result of hosting assumption \( e \) from \( L_1 \) to \( L_2 \) in \( V_1 \). \( V_1 \) and \( V_2 \) are bisimilar.

**Proof.** Copying is bisimilar due to Theorem 4.7 and the constant folded version is bisimilar due to Theorem 4.9.

5 DISCUSSION

Our formalization of speculative optimizations raises new questions and makes apparent certain design choices. In this section, we present advanced applications of `sourir` as well as insights into the design space for JIT implementations.

5.1 The Cost of Checkpoints

Having more checkpoints is useful for hoisting assumptions, but they are also optimization barriers. One obvious constraint is that all variables referenced by a checkpoint must be kept alive. Consider the following example, where we have a checkpoint at the end of a loop:

If \( y \) is never used in this version, we can safely remove it, since the varmap can synthesize it at the bailout target. On the other hand, we have no hope of synthesizing \( x \) out of thin air, because it might be the result of a side-effecting call. Even if \( x \) is never used in the loop body after line 4, we have to preserve its value in case we deoptimize. This dilemma is similar to when compiler writers need to decide how much of the local state is still available when debugging an optimized binary.

Given a powerful enough deoptimization mechanism, it is possible to synthesize many kinds of values on bailout. For example, while our formalism does not allow us to sink array allocations, this is an artificial limitation and we could extend the varmap with array constructors. However,
certain values cannot be reconstructed, such as results of possibly side-effecting calls or function arguments.

Checkpoints also restrict code motion in two cases. First, side-effecting code cannot be moved across a checkpoint, since it might be re-executed at the bailout target. Second, we cannot move code that modifies variables mentioned in the varmap. For instance, we cannot hoist a write $x \leftarrow 1$ above a checkpoint that needs $x$ to bail out, without preserving the original value.

Anecdotally, we have found it far more effective to hoist the guards within a checkpoint to another checkpoint (see Section 3.3). In the above example, if the expression $e$ is unchanged inside the loop, we could hoist it to a checkpoint above the loop and remove the checkpoint at $L_5$.

5.2 Lazy Deoptimization and Dependencies

There is also a runtime cost to checkpoints, which is compounded if the guard is inside a loop. Suppose we speculate that the contents of $x$ remain the same during the execution of a loop. Such a guard would have to check every single element of the array. This eager strategy, where we check if the assumption still holds and bail out otherwise, is needlessly wasteful. It would be more efficient for an external event, like a write to the array, to invalidate the assumption, a strategy sometimes known as lazy deoptimization.

In sourir, we could implement lazy deoptimization by separating assumptions from runtime checks. Specifically, we let GUARDS$[13] = \text{true}$ be the runtime check, where the global array GUARDS is a collection of all remote assumptions that can be invalidated by external events, such as an array assignment to $x$. In terms of correctness, both eager and lazy deoptimization are similar; however, we would need to prove correctness of the dependency mechanism that modifies the global array.

5.3 Inlining

Another interesting design consideration involves the interaction between speculative optimizations and function inlining. As we discussed in Section 3.2, we have to be able to recreate the caller context when bailing out of inlined code.

One option is to require a checkpoint immediately after the call instruction, which describes the caller environment, and append its varmap of the inlined checkpoints as extra frames. However, this is not strictly necessary: in sourir, the static scope at the return label is known. Therefore, our inlining pass does not require a checkpoint in the caller.

Another question is where the additional continuation should return to. It seems the deoptimized inlinee could return to either the optimized or the base version of the caller. In the example from Section 3.2, this would be $F_{\text{main}}, V_{\text{inlined}}-L_{\text{ret}}$. However, if we bailed out of the inlined code, it is precisely because some of its assumptions are invalid. It would be incorrect to bail out to code that was optimized under those assumptions.

5.4 Bailing Into Optimized Code

Most of our discussions concern bailing out of optimized code and into a less optimized version. We can also consider the inverse transition, where we bail into more optimized code from a less optimized version. Consider executing a long running loop in an unoptimized function:
The value of \textit{debug} is constant, yet we are stuck in this hot loop where we must branch on each iteration. The JIT can compile an optimized version of \textit{F} \textit{loop} \textit{y} that prunes the unused \textit{L}\textit{slow} branch, but now the problem is to transfer execution while \textit{V}_\textit{base} is still running, an operation known as \textit{hot loop transfer}. Specifically, the next time we reach \textit{L}\textit{body} of the unoptimized code, we want to transfer to an equivalent location in the optimized version. To do so, continuation-passing style can be used to compile a continuation function from the beginning of the loop, which takes arguments \textit{debug} and \textit{x}, the variables of the local state at \textit{F} \textit{loop} \textit{y}. \textit{V}_\textit{base}. \textit{L}\textit{body}. The optimized continuation we bail into might look like the following:

\begin{verbatim}
F_continuation(debug, x) \rightarrow
V_{opt} \rightarrow
L_loop : branch (x < 1000000) L_body L_done
L_body : goto L_fast
L_fast : ...
L_end : goto L_loop
L_done : ...
\end{verbatim}

In some sense, bailing in is easier than bailing out, because we strengthen our assumptions instead of weakening them. In other words, all the information needed to reconstruct the state in the target version is already available.

To simulate how a JIT bails into the optimized continuation, we can use the following version:

\begin{verbatim}
F_loop_y() \rightarrow
V_{bailin} \rightarrow
L_0 : ...
L_loop : branch (x < 1000000) L_body L_done
L_body : branch HOT L_transfer L_cont
L_transfer : call res = F continuation(debug, x)
L_ret : return res
L_cont : ...
V_{base} \rightarrow ...
\end{verbatim}

If the runtime profiler, which we treat as an oracle, determines the code is hot, then we perform a hot loop transfer by calling the optimized continuation. Otherwise, we proceed with execution as normal.

### 5.5 Fine-Grained Deoptimization

Instead of blindly removing all assumptions of a version when a checkpoint fails, it is possible to undo only failing assumptions while preserving the rest.
speculative Optimizations without Fear 1:23

\[ F_{undo}() \rightarrow \]
\[ V_{spec123} \rightarrow \]
\[ L_0 : assume [e_1, e_2, e_3] else (F_{undo} \cdot V_{spec123} \cdot L_0 [\ldots]) \]
\[ V_{spec12} \rightarrow \]
\[ L_0 : assume [e_1, e_2] else (F_{undo} \cdot V_{spec1} \cdot L_0 [\ldots]) \]
\[ V_{spec1} \rightarrow \]
\[ L_0 : assume [e_1] else (F_{undo} \cdot V_{base} \cdot L_0 [\ldots]) \]
\[ V_{base} \rightarrow \ldots \]

Fig. 9. To undo assumption \( e_2 \), we deoptimize and then re-optimize with rest of the assumptions enabled.

As shown in Figure 9, if assumption \( e_2 \) fails in version \( V_{spec123} \) we bail out to the last version that did not rely on this assumption. By deoptimizing to version \( V_{spec1} \), we are forced to also discard assumption \( e_3 \). However, \( e_1, e_3 \) still hold, so we would like to preserve optimizations based on those assumptions. Using the technique mentioned in Section 5.4 we continue executing in a version \( L_{spec13} \) that reintroduces \( e_3 \). The overall effect is that we remove only the invalidated assumption and its optimizations. We are not aware of an existing implementation that explores such a strategy.

5.6 Simulating a Tracing JIT

A tracing JIT [Bala et al. 2000; Gal et al. 2009] records the sequence of instructions that are executed at runtime, called a trace. Typically, a trace corresponds to a path through a frequently executed loop. On subsequent runs, the JIT implementation must ensure that execution follows the same path again, otherwise execution must bail out of the trace and into the original version of the program.

Guo and Palsberg [2011] develop a framework for reasoning about optimizations applied to traces. One of their results is that dead store elimination is unsound, because the trace is only a partial view of the entire program. For example, a variable \( z \) might be assigned to within a trace, but never used. However, it is unsound to remove the assignment, because \( z \) might be used outside the trace.

We can simulate their tracing formalism in sourir. Consider a variant of a running example in their paper, a trace of the loop while \( e \) (\( x \leftarrow 0; \ldots \)) embedded in a larger context:

\[ L_{begin} : \ldots \]
\[ L_{loop} : \textbf{branch} e L_{body} L_{done} \]
\[ L_{body} : x \leftarrow 0 \]
\[ L_3 : \ldots \]
\[ L_4 : \textbf{goto} L_{loop} \]
\[ L_{done} : \ldots \]

Instead of a runtime that records a sequence of instructions, we assume a runtime that records which branch targets are taken. For this example, suppose we record the two targets \( L_{body} \) and \( L_{done} \), which means we executed the loop body once and then exited. In other words, the loop condition \( e \) was \( \texttt{true} \) the first time we checked it and \( \texttt{false} \) the second time. We could have a loop unrolling to unroll the loop once and assume \( e \) for the first iteration and \( \neg e \) for the second iteration:
Now we apply unreachable code elimination to get the following result, which resembles a trace:

Since we optimize the entire function, we can determine if $x$ is used after $L_{body_0}$. However, $x$ might be used in an older version, if a guard fails and we bail out. In sourir, it becomes obvious why dead store elimination of $x$ is unsound: the varmap at checkpoint $L_{cp_1}$ indicates that $x$ is needed at the bailout target. We can only remove the store if we reconstruct its effect when we bail out. In this specific example, a constant propagation pass updates the varmap and allows us to materialize 0 at the second checkpoint.

However, before the code can be reduced, loop unrolling might result in intermediate versions that are much larger than the original program. In contrast, tracing JITs can handle this case without the drastic expansion in code size [Gal et al. 2009].

6 RELATED WORK

This paper touches topics from different fields. Deoptimization and on stack replacement (OSR), pioneered by the Self VM [Hölzle and Ungar 1994], have been used extensively for speculative optimizations [Almasi and Padua 2000; Fink and Qian 2003; Paleczny et al. 2001].

There are formalizations for tracing compilers [Dissegna et al. 2014; Guo and Palsberg 2011], but we are unaware of any other formalization effort for speculative optimizations in general. Having bailout metadata and/or assumptions as first-class entities in the IR has been tried out experimentally in a number of settings.

Implementations of Speculative Optimization

Soman and Krintz [2006] present an implementation of deoptimization for speculative optimization. In their implementation, a data structure called VARMAP corresponds roughly to the bailout information in our assume instruction. They show that more optimizations can be unlocked if the optimizer can update this mapping. In the paper, three different updates are possible: transferVarForOsr(x,y), replaceVarWithExpression(x, e) and removeVarForOsr(x). In sourir the corresponding bailout metadata is $[x = y]$, $[x = e]$ and $[x = nil]$. Soman and Krintz [2006] claim that the additional optimizations are unlocked due to removing the deoptimization metadata from the instruction stream and storing it offline. As we have shown with sourir, keeping the metadata inline does
not mean it cannot be updated. In many cases, our approach is even superior. For example, a constant propagation pass can be written in a way that it handles updating uses of variables in any instruction, including **assume**. Their paper has some experimental support for the claim that separating the technical challenges of deoptimization and its application for speculation in the optimizer into two separate problems also benefits the implementation.

Duboscq et al. [2013] propose a similar approach. Their implementation features a graph based IR with **framestate** and **guard** nodes. The former represent the bailout metadata and the later the dynamic checks. For implementation reasons, it seems advantageous to keep the two parts separated and associate guards to a framestate in a later scheduling phase. This has a similar effect as our hoist assumption phase. The framestate nodes can be updated and they support, for example, sinking object allocation by what they call **virtual objects**, that are only materialized on bailout. The paper presents the IR and its speculation facilities, but does not discuss how speculation interacts with the optimizer.

Béra et al. [2016] present a way to test their bytecode-to-bytecode JIT compiler. The approach involves symbolic execution of optimized and unoptimized versions of the bytecode and verifying the deoptimization mapping for every checkpoint.

Odaira and Hiraki [2005] investigate exception reordering by hoisting guards. Their implementation handles programs with interleaved exceptions, like our example from Section 3.4.

**Complementary Work**

The **sourir** IR uses a single language to represent all versions of the program, optimized or not. Recent work suggests it is practical to describe deoptimization at an IR level. For example, Lameed and Hendren [2013] add support for bailout to LLVM and D’Elia and Demetrescu [2016] show how to generalize the concept to support more expressive variable mappings. Alternatively, the micro virtual machine is a proposed compile target for dynamic languages that supports stack introspection and conditional checkpoints [Wang et al. 2015]. The idea of this work is to design an architecture that supports common runtime code modification idioms of JITs.

We do not consider how to efficiently represent the speculative assumptions and bailout information. Both Schneider and Bolz [2012] and Duboscq et al. [2014] present techniques for production implementations to reduce the size requirements.

**Formalizing compiler optimizations**

There is now a rich literature on formalizing compiler optimizations, even mechanized formalizations, that cover many more optimizations. The CompCert work [Leroy and Blazy 2008] for example implements many more optimization, and contains detailed proof arguments for the data-flow optimization used for constant folding that is extremely similar to ours – in fact, CompCert’s RTL language is close to **sourir** without versions and checkpoints, which suggests that adding them may be interesting future work to connect speculation verification to a realistic compiler backend.

While bisimulation is a good support to intuition—we encourage JIT implementers can think in terms of state correspondence—in our experience direct bisimulation proofs can be bothersome. So lighter proof techniques would be necessary to make the effort scale. Schneider et al. [2016] propose inductive arguments to obtain simulation results that could be a source of inspiration.

7 **FUTURE WORK**

The formalization presented in this paper will serve as a foundation for future work. We consider to further explore the following areas.

**Experimental Validation.** We would like to validate the design by implementing an optimizing JIT, that follows the design principles laid out in this paper from the ground. We consider this as the next step and we expect a positive feedback loop between testing the ideas in practice and formalizing new issues that might become relevant.

**Bidirectional Bailout Metadata.** There is a potential for unification of hot loop transfer and de-optimization. A checkpoint is a function exit point and its bailout target is an entry point. The metadata associated with the checkpoint describes the transition from the former to the latter. If this mapping is bidirectional then every entry point is also a valid exit point and vice versa. It is an open question how bijective checkpoints would interact with optimizations and if they would be useful in practice. It seems that more fine-grained composition of more and less aggressively optimized fragments would be possible. For example, if usage of an assumption could be restricted to a region of optimized code, then when the assumption fails that region could be replaced by a fragment of less optimized code surrounded by glue code that projects the local state one version down and then up again.

**Self Modification.** For some features of a JIT implementation self modification would need to be reintroduced. A possible direction would be to follow the approach laid out by Wang et al. [2015] and allow only a controlled set of self mutation. This would enable the extension of the correctness argument to speculation on external events as discussed in Section 5.2, and to the deoptimization implementation itself.

8 CONCLUSIONS

Speculative optimizations are key to just-in-time optimization of dynamic languages. While there are formalizations of JITs and runtime code generation, the formalization of speculation, and bailing out of incorrect optimizations, was unexplored. We formalize correctness of speculative optimizations by abstracting away orthogonal issues of JIT implementations. We introduce an intermediate representation for a compiler with explicit assumption checkpoints and show how to build correct speculative optimization passes. We formally prove standard compiler optimizations (constant folding, unreachable code elimination, and function inlining) in the presence of assumptions. Our model of JIT compilation suggests future research directions, such as fine-grained, per-assumption deoptimization and bidirectional checkpoints.

REFERENCES


ECMA International. 2016. ECMA Script 2016 Language Specification (7.0 ed.).


David Schneider and Carl Friedrich Bolz. 2012. The efficient handling of guards in the design of RPython’s tracing JIT. In Proceedings of the sixth ACM workshop on Virtual machines and intermediate languages. ACM, 3–12. 25


A APPENDIX

A.1 Proofs

Theorem A.1. Creating a new copy of the currently active version of a function, possibly adding new checkpoints (see Section 3.1), returns an equivalent program.

Proof. Consider a program \( P_1 \) with a function \( F \) having an active version \( V_1 \). Creating a new version, possibly adding checkpoints results in a program \( P_2 \) that has all the functions and versions of \( P_1 \), plus a new active version \( V_2 \) of \( F \) such that:

- any label \( L \) of \( V_1 \) also exists in \( V_2 \); the instruction at \( L \) in \( V_1 \) and in \( V_2 \) are identical, except for checkpoints that are updated: if \( L \) points to \textbf{assume} \( [e^*] \) else \( \xi \xi^* \) in \( V_1 \), then it points to \textbf{assume} \( [e^*] \) else \( (F.V_1.L \delta) \) in \( V_2 \), where \( \delta \) is the identity mapping over the lexical environment at \( L \).
- \( V_2 \) may contain extra empty checkpoints: for any instruction \( i \) at \( L \) in \( V_1 \), \( V_2 \) may contain a checkpoint of the form \textbf{assume} \( [] \) else \( (F.V_1.L \delta) \), where \( \delta \) is the identity mapping over the lexical environment at \( L \), followed by \( i \) at a fresh label \( L' \).

In the whole proof, we respect the following notational convention: a configuration with index 1, such as \( C'_1 \), is over the program \( P_1 \), and the configuration of the same name but with index 2, \( C'_2 \) in our example, is the exact same configuration in \( P_2 \): \( C[P_1 \leftarrow P_2] \). Note that a configuration of the form \( C_2 \) cannot be over the version \( V_2 \), which does not exist in \( P_1 \).

We define a relation \( R \) as the smallest relation such that:

- For any reachable configuration \( C_1 \) that is not over \( F.V_1.L \) for any \( L \), \( R \) relates \( C_1 \) to \( C_2 \).
- For any reachable configuration \( C_1 \) over a \( F.V_1.L \), \( R \) relates \( C_1 \) to both \( C_2 \) and \( C_2[V_1 \leftarrow V_2] \).
- For any checkpoint added in \( V_2 \) at label \( L \), with the next instruction at the fresh label \( L' \), for any reachable configuration \( C_1 \) over \( F.V_1.L \), \( R \) additionally relates \( C_1 \) to \( C_2[F.V_1.L \leftarrow F.V_2.L'] \).

It remains to prove that \( R \) is a bisimulation.

A consequence of our definition of \( R \) is that any reachable configuration \( C_1 \) is related to the corresponding relation \( C_2 \) in \( P_2 \).

This implies that \( R \) is a simulation: for any reduction \( C_1 \xrightarrow{A_r} C'_1 \) with \( (C_1, C_2) \in R \), we have \( C_2 \xrightarrow{A_r} C'_2 \) in \( P_2 \) with \( (C'_1, C'_2) \in R \). This also implies that \( R^{-1} \) is a simulation for configurations that are not over the new version \( V_2 \): for a reduction \( C_2 \xrightarrow{A_r} C'_2 \), we similarly have \( C_1 \xrightarrow{A_r} C'_1 \).

Now let us consider a reduction \( C_3 \xrightarrow{A_r} C'_3 \) over \( P_2 \), such that either \( C_3 \) or \( C'_3 \) are over the new version \( V_2 \), with \( C_3 \) related to some \( C \) over \( P_1 \).

If \( C_3 \) is not over \( V_2 \) but \( C'_3 \) is, then \( C_3 \) is a \( C_2 \) for some \( C_1 \) in \( P_1 \) and we have \( C_2 \xrightarrow{A_r} C'_3 \). The transition is either a function call or return (\( V_2 \) being a new version, it is not a bailout target). We distinguish the two cases:

- Function call: in \( P_1 \), a call to the function \( F \) goes to the active version of \( F \), that is \( V_1 \); by construction of \( R \), the start labels of \( V_1 \) and \( V_2 \) are related, so the transition \( C_2 \xrightarrow{A_r} C_3 \) is matched by \( C_1 \xrightarrow{A_r} C_3[V_2 \leftarrow V_1] \).
- Function return: this case is actually impossible. If \( C'_3 \) is in \( V_2 \), it means that \( C_3 \) had the version \( V_2 \) on its call stack. But we assumed that there exists \( C \) over \( P_1 \) such that \( (C, C_3) \in R \), and our relation \( R \) always relates configuration with exactly the same call stack. \( C \) cannot have a return address in \( V_2 \) on the call stack, so \( C_3 \) cannot either.

If \( C_3 \) is over \( V_2 \), then by construction of \( R \) the related configuration \( C \) in \( P_1 \) must be over \( V_1 \), and more precisely we have \( C = C_3[V_2 \leftarrow V_1] \).
The reduction behavior of $C_3$ is determined by its next instruction: either it is an instruction coming from $V_1$, or it is a new checkpoint:

- If the next instruction of $C_3$ comes from $V_1$, then $C_3$ and $C$ reduce in the exact same way: from $C_3 \xrightarrow{\text{Ar}} \xi^*$ and $C = C_3[V_2 \leftarrow V_1]$ we can deduce that $C \xrightarrow{\text{Ar}} C' \xi^*, C_3'[V_2 \leftarrow V_1]$, and $C_3'[V_2 \leftarrow V_1]$ is related to $C'$ as expected.

- If the next instruction of $C_3$ is one of the new checkpoints: it is a label $L$, the instruction is of the form \text{assume } [\text{else } \langle F, V_1, L \delta \rangle], with $\delta$ the identity environment over the current lexical scope. Because the assumption list is empty, this checkpoint always passes, and reduces to $C'$ at label $(L+1)$. This configuration points to the instruction that was present before the checkpoint was inserted; by construction of $R$ (third case), it is $R^{-1}$-related with $C_3[V_2 \leftarrow V_1]$, that is $C$.

We have established that $R$ is a bisimulation.

Finally, we remark that our choice of $R$ also proves that the Checkpoint Invariant is respected by the new version. A new checkpoint at label $L$ in $V_2$ is of the form \text{assume } [\text{else } \langle F, V_1, L \delta \rangle], with $\delta$ the identity environment. Any configuration $C$ over $F, V_2, L$ is in relation, by $R^{-1}$, with $C[F, V_2, L \leftarrow F, V_1, L]$, which is equal to bailout($C, \langle F, V_2, L \delta \rangle, \emptyset$). These two configurations are related by $R^{-1}$, and $R^{-1}$ is a bisimulation, so they are bisimilar.  \hfill $\square$

\textsc{Lemma A.2.} Adding a new assumption $e'$ to an existing checkpoint \text{assume } $e^*$ else $\xi \xi^*$ of $P_1$ returns an equivalent program $P_2$.

\textsc{Proof.} This is a consequence of the Checkpoint Invariant.

Let $R$ be the bisimilarity relation and $F, V, L$ be the location of the modified checkpoint. By the Checkpoint Invariant, any configuration $C$ over $F, V, L$ is related by $R$ to the post-bailout configuration bailout($C, \xi, \xi^*$).

$R$ is a relation between reachable configurations of $P_1$, but $P_2$ contains no new functions, version or label, so well-formed configurations of $P_1$ are also valid for $P_2$. $R$ can be restricted into a relation between reachable configurations of $P_1$ and reachable configurations of $P_2$. We prove that this restriction is a bisimulation.

Consider the reduction behavior of two related configurations $C_1, C_2$ over $P_1$ and $P_2$. The interesting (non-immediate) case is when both $C_1$ and $C_2$ are over the checkpoint being modified, the previous assumptions $e'$ all passed in $C_1$ and $C_2$, but the new assumption $e'$ fails in $C_2$—in all other cases the reductions in $P_1$ and $P_2$ are exactly identical. We now prove the interesting case.

Let us pose $\xi_1 \equiv C_1[L \leftarrow (L+1)], \xi_2 \equiv C_2[L \leftarrow (L+1)], \xi_3 \equiv \text{bailout}(C_2, \xi, \xi^*)$ and $\xi_3' \equiv C_3[L \leftarrow (L+1)]$. In $P_1$ we have $C_2 \xrightarrow{r^*} C_2'$, in $P_2$ we have $C_2 \xrightarrow{r^*} C_3$, and in both we have $C_1 \xrightarrow{r^*} C_1'$ and $C_3 \xrightarrow{r^*} C_3'$.

To show that $R$ is a bisimulation for $C_1$ in $P_1$ and $C_2$ in $P_2$, we have to show that $C_1'$ and $C_3$ silently reduce into $R$-related states.

We assumed that $C_1$ and $C_2$ are related by $R$, so they are bisimilar in $P_1$. Therefore, $C_1'$ and $C_2'$ are silently reduce into $R$-related states.

By the Checkpoint Invariant for $P_1$, $C_2$ and $C_4 = \text{bailout}(C_4, \xi, \xi^*)$ are also bisimilar in $P_1$. Therefore, their one-step reduct $C_2'$ and $C_4'$ silently reduce into $R$-related states.

By transitivity, we have that $C_1'$ and $C_3'$ silently reduce into $R$-related states. Given that $C_3 \xrightarrow{r^*} C_3'$, we can conclude that $C_1'$ and $C_3$ silently reduce into $R$-related states. \hfill $\square$