High-performance dynamic language implementations make heavy use of speculative optimizations to achieve speeds close to statically compiled languages. These optimizations are typically performed by a just-in-time compiler that generates code under a set of assumptions about the state of the program and its environment. In certain cases, a program may execute code compiled under assumptions that are no longer valid. The implementation must then deoptimize the program on-the-fly; this entails finding semantically equivalent code that does not rely on invalid assumptions, translating program state to that expected by the target code, and transferring control. This paper looks at the interaction between optimization and deoptimization, and shows that reasoning about speculation is surprisingly easy when assumptions are made explicit in the program representation. This insight is demonstrated on a compiler intermediate representation, named sourir, modeled after the high-level representation for a dynamic language. Traditional compiler optimizations such constant folding, dead code elimination, and function inlining are shown to be correct in the presence of assumptions. Furthermore, the paper establishes the correctness of compiler transformations specific to deoptimization: namely unrestricted deoptimization, predicate hoisting, and assume composition.
the assumption it is operating on integers, the body of the loop shrinks down to 13 instructions. As another example, most Java implementations assume that non-final methods are not over-ridden, and speculating on this fact allows compilers to avoid emitting dispatch code [Ishizaki, Kawahito, Yasue, Komatsu, and Nakatani 2000]. Newly loaded classes are monitored, and any time a method is overridden, the virtual machine invalidates code that contains devirtualized calls to that method. The validity of speculations is expressed as a predicate on the program state. If some program action, like loading a new class, falsifies that predicate, the generated code must be discarded. To undo an assumption, an implementation must ensure that functions compiled under that assumption are retired. This entails replacing affected code with a version that does not depend on the invalid predicate and, if a function currently being executed is found to contain invalid code, that function needs to be replaced on-the-fly. In such a case, it is necessary to transfer control to a different version of the function, and in the process, it may be necessary to materialize portions of the state that were optimized away and perform other recovery actions. In particular, if the invalid function was inlined into another function, it is necessary to synthesize a new stack frame for the caller. This is referred to as deoptimization, or on-stack-replacement, and is found in most industrial-strength compilers.

Speculative optimization gives rise to a large and multi-dimensional design space that lies mostly unexplored. First, compiler writers must decide how to obtain information about program state. This can be done ahead-of-time by profiling, just-in-time by sampling or instrumenting code. Next, they must select what facts to record. This can range from information about the program, its class hierarchy, which packages were loaded, to information about the value of a particular mutable location in the heap. Finally, they must decide how to efficiently monitor the validity of speculations. While some points in this space have been explored empirically, existing systems have done it in an ad hoc manner that is often both language- and implementation-specific, and thus difficult to apply broadly.

This paper has a focused goal. We aim to demystify the interaction between compiler transformations and deoptimization. When are two versions compiled under different assumptions equivalent? How should traditional optimizations be adapted when operating on code containing deoptimization points? In what ways does deoptimization inhibit optimizations? To do this in a way that is independent of the specific language being targeted and of implementation details relative to a particular compiler infrastructure, we have designed a high-level compiler intermediate representation (IR), named sourir, that is adequate for many dynamic languages without being tied to any one in particular.

Sourir is inspired by our work on RIR, an IR for the R language. A sourir program is made up of functions, and each function can have multiple versions. We equip the IR with a single instruction, named assume, specific to speculative optimization. This instruction has the role of describing what assumptions are being used to perform speculative optimization and what information must be preserved for deoptimization. It tests if those assumptions hold, and in case they do not, transfers control to another, less optimized, version of the code. Reifying assumptions in the IR makes the interaction with compiler transformations explicit and simplifies reasoning. The assume instruction is more than a branch: when deoptimizing it replaces the current stack frame with a stack frame that has the variables and values expected by the target version, and, in case the function was inlined, it synthesizes missing stack frames. Furthermore, unlike a branch, its deoptimization target is not followed by the compiler during analysis and optimization. The code executed in case of deoptimization is invisible to the optimizer. This simplifies optimizations and reduces compile
time as analysis remains local to the version being optimized and the deoptimization metadata is considered to be a stand-in for the target version.

As an example consider the function from Figure 1. A possible translation to sourir is shown in Figure 2 (less relevant code elided). Vbase contains the original version. We let the helper functions get and store implement JavaScript (JS) array semantics, and the function add implement JS addition. Version Vnative contains only primitive sourir instructions. This version is optimized under the assumption that the variable a is an array of primitive numbers, which is represented by the first assume instruction. Further, JS arrays can be sparse and contain holes, in which case access might need to be delegated to a getter function. For this example we use HL to denote such a hole. The second assume instruction reifies the compiler’s speculation that the array has no holes, by asserting the predicate t ≠ HL. It also contains the associated deoptimization metadata. In case the predicate does not hold, we deoptimize to a related position in the base version by recreating the variables in the target scope. As can be seen in the second assume, local variables are mapped as $[i = i, j = i + 1]$; the current value of i is carried over into the target frame’s i, whereas variable j has to be recomputed.

We prove the correctness of a selection of traditional compiler optimizations in the presence of speculation; these are constant propagation, unreachable code elimination, and function inlining. The main challenge for correctness is that the transformations operate on one version in isolation and therefore only see a subset of all possible control flows. We show how to split the work to prove correctness between the pass that establishes a version-to-version correspondence and the actual optimizations. Furthermore we prove the correctness of three optimizations specific to speculation, namely unrestricted deoptimization, predicate hoisting, and assume composition.

Our work makes several simplifying assumptions. We use the same IR for optimized and unoptimized code. We ignore the issue of generation of versions: we study optimizations operating on a program at a certain point of time, on a set of versions created before that time. We do not model the low-level details of code generation. Correctness of runtime code generation and code modification within a JIT compiler has been addressed by Myreen [2010]. Sourir is not designed for implementation, but to give a reasoning model for existing JIT implementations. We do not intend to implement a new JIT engine. Instead, we evaluated our work by discussing it with JIT implementers; the V8 team [Chromium 2017] confirmed that intuitions and correctness arguments could be ported from sourir to their setting.
2 RELATED WORK

The SELF virtual machine pioneered dynamic deoptimization [Hölzle, Chambers, and Ungar 1992]. The SELF compiler implemented many optimizations, one of which was aggressive inlining, yet the language designers wanted to give end users the illusion that they were debugging source code. They achieved this by replacing optimized code and the corresponding stack frames with non-optimized code and matching stack frames. When deoptimizing code that had been inlined, the SELF compiler synthesized stack frames. The HotSpot compiler followed from the work on SELF by introducing the idea of speculative optimizations [Paleczny, Vick, and Click 2001]. HotSpot supported very specific assumptions related to the structure of the class hierarchy and instrumented the class loader to trigger invalidation. When an invalidation occurred affected functions were rolled forward to a safe point and control was transferred from native code to an interpreter frame. The Jikes RVM adopted these ideas to avoid compiling uncommon code paths [Fink and Qian 2003].

One drawback of the early work was that deoptimization points were barriers around which optimizations were not allowed. Odaira and Hiraki [2005] were the first to investigate exception reordering by hoisting guards. They remarked that checking assumptions early might improve code. In Soman and Krintz [2006] the optimizer is allowed to update the deoptimization metadata. In particular they support eliding duplicate variables in the mapping and lazily reconstructing values when transferring control. This unlocks further optimizations, which were blocked in previous work. The paper also introduces the idea of being able to transfer control at any point. We support both the update of metadata and unconstrained deoptimization.

Modern virtual machines have all incorporated some degree of speculation and support deoptimization. These include implementations of Java (HotSpot, Jikes RVM), JavaScript (WebKit Core, Chromium V8, Truffle/JS, Firefox), Ruby (Truffle/Ruby), and R (FastR), among others. Anecdotal evidence suggests that the representation adopted in this work is representative of the instructions found in the IR of production VMs: the TurboFan IR from V8 [Chromium 2017] represents assume with three distinct nodes. First a checkpoint, holding the deoptimization target, marks a stable point, to where execution can be rolled back. In sourir this corresponds to the original location of an assume. A framestate node records the layout of, and changes to, the local frame, roughly the varmap in sourir. Assumption predicates are guarded by conditional deoptimization nodes, such as deoptimizelf. Graal [Duboscq, Würthinger, Stadler, Wimmer, Simon, and Mössenböck 2013] also has an explicit representation for assumptions and associated metadata as guard and framestate nodes in their high-level IR. In both cases guards are associated with the closest dominating checkpoint. Lowering deoptimization metadata is described in Duboscq, Würthinger, and Mössenböck [2014]; Schneider and Bolz [2012]. A detailed empirical evaluation of deoptimization appears in Zheng, Bulej, and Binder [2017]. The implementation of control-flow transfer is not modeled here as it is not relevant to our results. For one particular implementation, we refer readers to D’Elia and Demetrescu [2016] which builds on LLVM. Alternatively, Wang, Lin, Blackburn, Norrish, and Hosking [2015] propose an IR that supports restricted primitives for hot-patching code in a JIT.

There is a rich literature on formalizing compiler optimizations. The CompCert project [Leroy and Blazy 2008] for example implements many optimizations, and contains detailed proof arguments for a data-flow optimization used for constant folding that is similar to ours. In fact, sourir is close to CompCert’s RTL language without versions or assumptions. There are formalizations for tracing compilers [Dissegna, Logozzo, and Ranzato 2014; Guo and Palsberg 2011], but we are unaware of any other formalization effort for speculative optimizations in general. Béra, Miranda, Denker, and Ducasse [2016] present a verifier for a bytecode-to-bytecode optimizer. By symbolically executing optimized and unoptimized code, they verify that the deoptimization metadata produced by their optimizer correctly maps the symbolic values of the former to the latter at all deoptimization points.
3 SOURIR: SPECULATIVE COMPILATION UNDER ASSUMPTIONS

This section introduces our IR and its design principles. We first present the structure of programs and the assume instruction. Then, Section 3.2 and following explain how sourir maintains multiple equivalent versions of the same function, each with a different set of assumptions. This enables the speculative optimizations presented in Section 4. All concepts introduced in this section are formalized in Section 5.

3.1 Sourir in a Nutshell

Sourir is an untyped language with lexically scoped mutable variables and first-class functions. As an example the function in Figure 3 reads queries a number n from the user and initializes an array with values from 0 to n-1. By design, sourir is a cross between a compiler representation and a high-level language. We have equipped it with sufficient expressive power so that it is possible to write interesting programs in a style reminiscent of dynamic languages. The only features that are critical to our result are versions and assumptions. Versions are the counterpart of dynamically generated code fragments. Assumptions, represented by the assume instruction, support dynamic deoptimization of speculatively compiled code. The syntax of sourir instructions is shown in Figure 4.

Sourir supports defining a local variable, removing a variable from scope, variable assignment, creating arrays, array assignment, (unstructured) control flow, input and output, function calls and returns, assumptions, and terminating execution. Control-flow instructions take explicit labels, which are compiler-generated symbols but we sometimes give them meaningful names for clarity of exposition. Literals are integers, booleans, and nil. Together with variables and function references, they form simple expressions. Finally, an expression is either a simple expression or an operation: array access, array length, or primitive operation (arithmetic, comparison, and logic operation). Expressions are not nested—this is common in intermediate representations such as A-normal form [Sabry and Felleisen 1992]. We do allow bounded nesting in instructions for brevity.

A program P is a set of function declarations. The body of a function is 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. We use 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. Every instruction is labeled, but for brevity we omit unused labels.

Versions model the speculative optimizations performed by the compiler. The only instruction that explicitly references versions is assume. It has the form assume e{} else \(\xi\) \(\xi^*\) with a list of predicates \(e^*\) and deoptimization metadata \(\xi\) and \(\xi^*\). When executed, assume evaluates its predicates; if they hold execution skips to the next instruction. Otherwise, deoptimization occurs according to the metadata. The format of \(\xi\) is F.V.L \(\{x_1 = e_1, \ldots, x_n = e_n\}\), which contains a target F.V.L and a varmap \(\{x_1 = e_1, \ldots, x_n = e_n\}\). To deoptimize, a fresh environment for the target is created according to the varmap. Each expression \(e_i\) is evaluated in the old environment and bound to \(x_i\) in the new environment. The environment specified by \(\xi\) replaces the current one. Deoptimization might also need to create additional continuations (i.e., activation records), if assume occurs in an inlined

---

1 An implementation of a sourir interpreter and of the optimizations presented here is available at https://github.com/reactorlabs/sourir.

function. In this case multiple $\xi$ of the form $F.V.L. x$ \[x_1 = e_1, \ldots, x_n = e_n\] can be appended. Each one synthesizes a continuation with an environment constructed according to the varmap, a return target $F.V.L$, and the name $x$ to hold the returned result—we discuss this situation and inlining in Section 4.3. The purpose of deoptimization metadata is twofold. First, it provides the necessary information for jumping to the target version. Second, its presence in the instruction stream allows the optimizer to keep the mapping between different versions up-to-date.

**Example.** Consider the function size in Figure 5 which computes the size of a vector $x$. In version $V_b$, $x$ is either nil or an array with its length stored at index 0. The optimized version $V_o$ expects that the input is never nil. Classical compiler optimizations can leverage this fact: unreachable code removal prunes the unused branch. Constant propagation replaces the use of $el$ with its value and updates the varmap so that it restores the deleted variable, when we deoptimize to the base version $V_b$.

```
size(x)
Vo
  assume x != nil else size.Vb.L2 [el = 32, x = x]
  var l = x[0]
  return l * 32
Vb
  L1 var el = 32
  L2 branch x = nil L3 L4
  L3 var l = x[0]
  return l * el
  L4 return 0
```

Fig. 5. Speculation on $x$. 
3.2 Deoptimization Invariants

A version is the unit of optimization and deoptimization. Thus we expect that each function will have one original version and possibly many optimized versions. Versions are constructed such that they preserve two crucial invariants: (1) version equivalence and (2) assumption transparency. By the first invariant all versions of a function are observationally equivalent. The second invariant ensures that even if the assumption predicates do hold, deoptimizing to the target should be correct. Thus one could execute an optimized version and its base in lockstep; at every assume the varmap provides a complete mapping from the new version to the base. This simulation relation between versions is our correctness argument. The transparency invariant allows us to add assumption predicates without fear of altering program semantics. Consider a function show in Figure 6 which prints its argument x. Version V₀ respects both invariants: any value for x will result in the same behavior as the base version and deoptimizing is always possible. On the other hand, Vₓ, which is equivalent because it will never deoptimize, violates the second invariant: if it were to deoptimize, the value of x would be set to 42, which is almost always incorrect. We present a formal treatment of the invariants and the correctness proofs in Section 5.4 and following.

3.3 Creating Fresh Versions

We expect that versions are chained. A compiler will create a new version, say V₁, from an existing version V₀ by copying all instructions from the original version and chaining their deoptimization targets. The latter is done by updating the target and varmap of assume instructions such that all targets refer to V₀ at the same label as the current instruction. As the new version starts out as a copy, the varmap is the identity function. For instance, if the target contains the variables x and y, then the mapping is [x = x, z = z]. Additional assume instructions can be added; assume instructions that bear predicates (i.e., the predicate list is either empty or just tautologies) can be removed while preserving equivalence. As an example in Figure 7, the new version V₂ is a copy of V₁; the instruction at L₀ was added, the instruction at L₁ was updated, and the one at L₂ was removed.
Updating assume instructions is not required for correctness. But the idea with a new version is that it captures a set of assumptions that can be undone independently from the previously existing assumptions. Thus, we want to be able to undo one version at a time. In an implementation, versions might, for example, correspond to optimization tiers. This approach can lead to a cascade of deoptimizations if an inherited assumption fails; we discuss this in Section 4.6. In the following sections we use the base version Vb of Figure 5 as our running example. As a first step, we generate the new version Vdup with two fresh assume instructions shown in Figure 8. Initially the predicates are true and the assume instructions never fire. Version Vb stays unchanged.

### 3.4 Injecting Assumptions

We advocate an approach where the compiler—after establishing the correspondence between two versions with assume instructions—first injects predicates, and then uses those predicates in optimizations. In contrast, earlier work would apply an unsound optimization and then recover by adding a guard (see, for example, Duboscq et al. [2013]). While the end result is the same, 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. It is therefore correct for the compiler to expect that the predicate holds immediately following an assume. Inserting a fresh assume in a function is difficult in general, as we must determine where to transfer control to or how to reconstruct the target environment. On the other hand, it is always correct to add a predicate to an existing assume. Thanks to the assumption transparency invariant it is always safe to deoptimize more often to the target. For instance, in assume x ≠ nil, x > 10 else ... the predicate x ≠ nil was narrowed down to x > 10.

### 4 OPTIMIZATION WITH ASSUMPTIONS

In the previous section we introduced our approach for establishing a fresh version of a function that lends itself to speculative optimizations. Next, we introduce classical compiler optimizations that are exemplary of our approach. Then we give additional transformations for the assume in Section 4.4 and following, and conclude with a case study in Section 4.7. All transformations introduced in this section are proved correct in Section 6.

#### 4.1 Constant Propagation

Consider a simple constant propagation pass that finds constant variables and then updates all uses. This pass maintains a map from variable names to constant expressions or unknown. The map is computed for every position in the instruction stream using a data-flow analysis. Following the approach by Kildall [1973], the analysis has an update function to add and remove constants to the map. For example analyzing var x = 2, or x ← 2 adds the mapping x → 2. The instruction

---

\[\text{size(x)}\]

Vdup

<table>
<thead>
<tr>
<th>L1</th>
<th>assume true else size.Vb.L1 [x = x]</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2</td>
<td>var el = 32</td>
</tr>
<tr>
<td>L3</td>
<td>assume true else size.Vb.L2 [el = el, x = x]</td>
</tr>
<tr>
<td>L4</td>
<td>branch x = nil L5 L6</td>
</tr>
<tr>
<td>L5</td>
<td>var l = x[0]</td>
</tr>
<tr>
<td>L6</td>
<td>return l + el</td>
</tr>
<tr>
<td>L7</td>
<td>return 0</td>
</tr>
</tbody>
</table>

Vb ...

Fig. 8. A fresh copy of the base version of size.

2A common strategy for VMs is to have different kind of optimizing compilers with different compilation speed versus code quality trade-offs. The more a code fragment is executed, the more powerful optimizations will be applied to it.
\( \text{var } y = x + 1 \) adds \( y \rightarrow 3 \) to the previous map. Finally, \( \text{drop } x \) removes a mapping. Control-flow
merges rely on a join function for intersecting two maps; mappings which agree are preserved, while
others are set to \( \text{unknown} \). In a second step, expressions that can be evaluated to values are replaced
and unused variables are removed. No additional care needs to be taken to make this pass correct
in the presence of assumptions. This is because in \text{sour'1r}, the expressions needed to reconstruct
environments appear in the varmap of the assume and are thus visible to the constant propagation
pass. Additionally, the pass can update them, for example, in \text{assume true else } F.V.L \ [x = y + z],
the variables \( y \) and \( z \) are treated the same as in \text{call } h = \text{foo}(y + z). \) They can be replaced and will
not artificially keep constant variables alive.

Constant propagation can become speculative. After the instruction \text{assume } x = 0 \ else \ldots, \), we
know that \( x \) is 0. Therefore, we can add \( x \leftarrow 0 \) to the state map. This is the only extension required
for speculative constant propagation. As an example, in the case where we speculate on a nil check
\[
\begin{align*}
& \ldots \\
& L2 \quad \text{assume } x \neq \text{nil else } \text{size}.V.b.L2 \ [el = el, x = x] \\
& \text{branch } x = \text{nil } L5 \ L6 \\
& \ldots
\end{align*}
\]
the map is \( x \rightarrow \text{¬nil} \) after L2. Evaluating the branch condition under this context yields \( \text{¬nil} == \text{nil} \),
and a further optimization opportunity presents itself.

4.2 Unreachable Code Elimination
As shown above, an assumption coupled
with constant folding leads to branches
becoming deterministic. Unreachable code
elimination benefits from that. We consider
a two step algorithm: the first pass replaces
\text{branch } e L1 L2 with \text{goto } L1 if \( e \) is a tautology and with \text{goto } L2 if it is a contradiction.
The second pass removes unreachable
instructions. In our running example from
Figure 8, we add the predicate \( x \neq \text{nil} \) to
the empty assume at L2. Constant propagation
shows that the branch always goes to
L5, and unreachable code elimination removes the dead statement at L6 and branch. This creates
the version shown in Figure 9. Additionally, constant propagation can replace \( el \) by 32. By also
replacing its mention in the varmap of the assume at L2, \( el \) becomes unused and can be removed
from the optimized version. This yields version \( V_0 \) in Figure 5 at the top.

4.3 Function Inlining
Function inlining is our most involved optimization, since assume instructions inherited from the
inlinee need to remain correct. The inlining itself is standard. Name mangling is used to separate
the caller and callee environments. As an example Figure 10 shows the inlining of size into a
function main. Naïvely inlining without updating the metadata of the assume at L1 will result in
an incorrect deoptimization, as we would jump to \text{size}.V.b.L2 with no way to return to the main
function. Also, main’s part of the environment is discarded in the transfer and permanently lost.
The solution is that we must synthesize a new stack frame. As shown in the figure, the assume at
in the optimized main is thus extended with \text{main}.V.b.Lret s [pl = pl, vec = vec].
This creates an additional stack frame that returns to the base version of main, and stores the result in s with the entire caller portion of the environment reconstructed. It is always possible to compute the continuation, since the original call site must have a label and the scope at this label is known. Overall, after deoptimization, it appears as if version Vb of main had called version Vb of size. Note, we should not create a continuation that returns to the optimized version of the caller Vinl. If we deoptimized from the inlined code, it is precisely because some of its assumptions are invalid. Multiple continuations can be appended for further levels of inlining. The inlining needs to be applied bottom up: for the next level of inlining, e.g., to inline Vinl into an outer caller, renamings must also be applied to the extra continuation frames, since they refer to local variables in Vinl.

\begin{figure}[h]
\centering
\begin{lstlisting}
main() Vinl
  array pl = [1, 2, 3, 4]
  array vec = [length(pl), pl]
  var s = nil
  var x = vec
  assume x ≠ nil else size.Vb.L2 [el = 32, x = x] main.Vb.Lret s [pl = pl, vec = vec]
   var l = x[0]
   s ← l * 32
   drop l
   drop x
   goto Lret
   print s
   stop
Lret
Vb
  array pl = [1, 2, 3, 4]
  array vec = [length(pl), pl]
  \call s = size(vec)
Lret
  print s
  stop
size(x) Vo
  \assume x ≠ nil else size.Vb.L2 [el = 32, x = x]
  var l = x[0]
  return l * 32
Vb ...
\end{lstlisting}
\caption{An inlining of size into a main.}
\end{figure}

### 4.4 Unrestricted Deoptimization

The assume instructions are expensive: they create dependencies on live variables and are barriers for moving instructions. Hoisting a side-effecting instruction over an assume is invalid, because if we deoptimize the effect happens twice. Removing a local variable is also not possible if its value is needed to reconstruct the target environment. Thus it makes sense to insert as few assume instructions as possible. On the other hand we would like to “deoptimize everywhere”—checking assumptions in the basic block in which they are used can avoid unnecessary deoptimization—so there is a tension between speculation and optimization. Reaching an assume marks a stable state in the execution of the program that we can fall back to, similar to a transaction. Implementations, like Duboscq et al. [2013], separate deoptimization points and the associated guards into two separate instructions, to be able to deoptimize more freely. As long as the effects of instructions performed since the last deoptimization point are not observable, it is valid to throw away intermediate results and resume control from there. Effectively, in sourir this corresponds to moving an assume instruction forward in the instruction stream, while keeping its deoptimization target fixed. An assume can be moved over another instruction if that instruction:

1. has no side-effects and is not a call instruction,
2. does not interfere with the varmap or predicates, and
3. has the assume as its only predecessor instruction.
The first condition prevents side-effects from happening twice. The second condition can be enabled by copying the affected variables at the original assume instruction location (i.e., taking a snapshot of the required part of the environment). The last condition prevents capturing traces incoming from other basic blocks where (1) and (2) do not hold for all intermediate instructions since the original location. This is not the weakest condition, but a reasonable, sufficient one. Let us consider a modified version of our running example in Figure 11 on the left. Again, we have an assume before the branch, but would like to place a guard inside one of the branches.

![Diagram](image)

**Fig. 11.** Moving an assume forward in the instruction stream.

There is an interfering instruction at L5 that modifies x. By creating a temporary variable to hold the value of x at the original assume location we can resolve the interference. Now it is possible to move the assume inside the branch and then add a predicate on the updated x, as can be seen on the right side of the figure. Note that the target is unchanged. This approach allows for the (logical) separation between the deoptimization point and the position where assumption predicates are introduced. In the transformed example a stable deoptimization point is established at the beginning of the function by storing the value of x, but then the assumption is checked only in one branch. The intermediate states are ephemeral and can be safely discarded when deoptimizing. For example the variable el is not mentioned in the vmap here, we say it is not captured by the assume. Instead it is recomputed by the original code at the deoptimization target size.Vb.L1. To be able to deoptimize from any position it is sufficient to have an assume after every side-effecting instruction, call, and control-flow merge.

### 4.5 Predicate Hoisting

Moving an assume backwards in the code would require replaying the moved-over instructions in the case of deoptimization. Hoisting `assume true else size.Vb.L1 [x = x]` is allowed if the vmap is changed to `[el = el, . . .] ` above `var el = 32` is allowed if the vmap is changed to `[el = 32, . . .] ` to compensate for the lost definition. However this approach is tricky and does not work for instructions with multiple predecessors as it could lead to conflicting compensation code. But as a simple alternative to hoisting assume, we can hoist a predicate from one assume to a previous one. To understand why, let us decompose the approach in two steps. Given an assume at L1 that dominates a second one at L2, we copy a predicate from the latter to the former. This is valid since the assumption transparency invariant allows strengthening predicates. A data-flow analysis can determine if the copied predicate from L1 is available at L2, in which case it can be removed from the original instruction. In our running example, version Vpruned in Figure 9 has two assume instructions and one predicate. We can trivially hoist `x ≠ nil`,

---

3In an SSA based IR this step is not necessary for SSA variables, since the captured ones are guaranteed to stay unchanged.
since there are no interfering instructions. This allows us to discard the assume with the larger scope. More interestingly, in the case of a loop-invariant assumption, we can hoist predicates out of the loop.

### 4.6 Assume Composition

As we have argued in Section 3.3, it is beneficial to undo as few assumptions as possible. On the other hand, deoptimizing an assumption added in an early version cascades through all the later versions. To be able to remove chained assume instructions, we show that assumptions are *composable*. If an assume in version V3 transfers control to a target V2.La that is itself an assumption with V1.Lb as target, then we can combine the metadata to take both steps at once. By the assumption transparency invariant, the pre- and post-deoptimization states are equivalent: even if the assumptions are not the same, it is correct to conservatively trigger the second deoptimization. For example, an instruction `assume e else F.V2.La [x = 1]` that jumps to `assume e' else F.V0.Lb [y = x]` can be combined as `assume e, e' else F.V0.Lb [y = 1]`. This new unified assume skips the intermediate version V2 and goes to V0 directly. This could be an interesting approach for multi-tier JITs: after the system stabilizes, the intermediate versions are rarely used and we might want to discard them.

### 4.7 Case Study

We conclude with an example. In dynamic languages code is often dispatched on runtime types. If types were known, code could be specialized, resulting in faster code with fewer checks and branches. Consider implementing a generic binary division function that expects two values and their type tags:

\[
\text{div}(\text{tag}x, x, \text{tag}y, y)
\]

\[
\begin{align*}
V_{\text{base}} & \quad \text{branch} \tagx \neq \text{NUM} \quad \text{Lslow} \quad \text{L2} \\
L1 & \quad \text{branch} \tagy \neq \text{NUM} \quad \text{Lslow} \quad \text{L3} \\
L2 & \quad \text{branch} \quad x = 0 \quad \text{Lerror} \quad \text{L4} \\
L3 & \quad \text{return} \quad y/x \\
\text{Lslow} & \quad \ldots
\end{align*}
\]

No static information is available; the arguments could be any type. Therefore, multiple checks are needed before the division, for example the slow branch will require even more checks on the exact value of the type tag. Suppose we have profiling information that indicates we should expect numbers. We specialize the function by speculatively pruning the branches:

\[
\begin{align*}
\text{assume} & \quad \tagx = \text{NUM}, \tagy = \text{NUM} \quad \text{else} \quad \text{div.Vb.L1} \quad \ldots \\
\text{branch} & \quad x = 0 \quad \text{Lerror} \quad \text{L4} \\
L4 & \quad \text{return} \quad y/x \\
\ldots
\end{align*}
\]

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 speculate on x, but not y:

\[
\begin{align*}
\text{assume} & \quad \tagx = \text{NUM}, x \neq 0 \quad \text{else} \quad \text{div.Vb.L1} \quad \ldots \\
\text{branch} & \quad \tagy \neq \text{NUM} \quad \text{Lslow} \quad \text{L4} \\
L4 & \quad \text{return} \quad y/x \\
\ldots
\end{align*}
\]

In this version, we perform both checks on x first and then the ones on y, 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. The fully speculated on version contains only the integer division and the required assumptions.

```plaintext
assume tagx = NUM, tagy = NUM, x ≠ 0 else div
return y/x
```

This version has no more branches and is for example a good inlining candidate.

## 5 SOURIR: SPECULATIVE COMPILATION FORMALIZED

A sourir program contains several functions, each of which can have multiple versions. This high-level structure is described in Figure 12. The first version is considered the currently active version and will be picked by a call operation. Each version consists of a stream of labeled instructions. We use an indentation-based syntax that directly reflects this structure and omit unreferenced labels. Besides grammatical and scoping validity, we impose some well-formedness requirements to ease analysis and reasoning.

### 5.1 Operational Semantics: Expressions

In Figure 13 we define the semantics of expressions. The evaluation of an expression \( e \) returns a value \( v \), which may be either a literal \( lit \), a function, or an address \( a \). Arrays are represented by addresses into memory \( M \), formalized as a mapping from addresses to blocks of 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 variables in the current scope to their values. Evaluation is defined by a relation \( M \ E \ e \rightarrow v \): under memory \( M \) and environment \( E \), \( e \) evaluates to \( v \). This definition in turn relies on a relation \( E \ se \rightarrow v \) defining evaluation of simple expressions \( se \), which need not access arrays. The evaluation rules are standard. We use the notation \([primop]\) to denote, for each primitive operation \( primop \), a partial function on values. Arithmetic operators and arithmetic

---

**Proceedings of the ACM on Programming Languages, Vol. 2, No. POPL, Article 49. Publication date: January 2018.**
v ::= values
| lit
| F
| a

addr ::= a
M ::= (a → [v₁, . . . , vn])
E ::= (x → v)

[LITERAL]
E lit → lit

[FUNCTION]
E F → F
E x → E(x)

[LOOKUP]

[SIMPLE EXPRESSION]
E se → v
M E se → v

M E primop(se₁, . . . , sn) → [[primop]](v₁, . . . , vn)

[VECTOR LENGTH]
E se → a
M(a) = [v₁, . . . , vn]
M E length(se) → n

M E x[se] → vn

Fig. 13. Evaluation M E e → v of expressions and E se → v of simple expressions.

comparison operators are only defined when their arguments are numbers. Equality and inequality are defined for all values. The relation M E e → v, when seen as a function from M, E, e to v, is partial: it is not defined on all inputs. For example, there is no v such that the relation M E x[se] → v holds if E(x) is not an address a, if a is not bound in M, if se does not reduce to a number n, or if n is out of bounds.

5.2 Operational Semantics: Instructions and 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 program being executed, and a reduction relation, a transition relation between configurations, that specifies the execution of a single instruction. A configuration is a six-component tuple \((PILK^*M)\) described in Figure 14. Call continuations \(K\) are tuples of the form \((I,L,x,E)\), storing the information needed to correctly return to a caller function. On a call \(\text{call } x = e(e₁, . . . , en)\), the continuation pushed on the stack contains the current instruction stream \(I\) (to be restored on return), the label \(L\) of the next instruction after the call (the return label), the variable \(x\) to name the returned result, and the lexical environment \(E\). For the details, see the reduction rules for \(\text{call}\) and \(\text{return}\) in Figure 16.

Our reduction relation \(C \xrightarrow{A_r} C'\) specifies that executing the next instruction of \(C\) may result in the configuration \(C'\). The action \(A_r\) indicates whether this reduction is observable: it is either the silent action, traditionally written \(\tau\), an input/output action read \(\text{lit}\) or print \(\text{lit}\), or stop. We write \(C \xrightarrow{T} C'\) when there are zero or more steps from \(C\) to \(C'\). The reduction trace \(T\) is a list of non-silent actions, collected in the order in which they appeared. For example, if we have \(C_1 \xrightarrow{\text{read } 1} C_2 \xrightarrow{\tau} C_3 \xrightarrow{\text{print } 2} C_4\), then we have \(C_1 \xrightarrow{\text{read } 1} \xrightarrow{\text{print } 2} C_4\). Actions are defined in Figure 15, and the full reduction relation is given in Figure 16.

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. The read \(\text{lit}\) and print \(\text{lit}\)
actions represent observable input and output operations. They are emitted by Read and Print in Figure 16. In particular, the action read lit on the read x transition may be any literal value. This is the only reduction rule that is non-deterministic, as an arbitrary value may be passed by the environment. For all other rules, for a given $C$, there is at most one $A_\tau$ and one $C'$ such that $C \xrightarrow{A_\tau} C'$ holds. In particular, the relation $C \rightarrow^* C'$ (with the empty trace), containing only sequences of silent reductions, is deterministic. The stop reduction emits the stop transition, and also produces a configuration with an empty instruction sequence $\emptyset$; this is a technical device to ensure that the resulting configuration is stuck—it cannot reduce further. Thanks to the “Explicit stop” a program with a silent loop (without input/output) has a different trace from a program that halts.

Given a program $P$, we define $\text{start}(P)$ as its starting configuration, and $\text{reachable}(P)$ as the set of configurations reachable from it; they are all the states that may be encountered during a valid run of $P$.

$$
\text{start}(P) \overset{\text{def}}{=} (P I L K^* M E)
$$

$$
\text{reachable}(P) \overset{\text{def}}{=} \{ C \mid \exists T, \text{start}(P) \xrightarrow{T} C \}
$$

Fig. 15. Actions and traces.
\[ I(L) = \text{var } x = e \quad M E e \rightarrow v \]  
\[ \langle P I L K^* M E \rangle \xrightarrow{e} \langle P I (L+1) K^* M E[x \leftarrow v] \rangle \]  
\[ \text{[DECL]} \]

\[ I(L) = \text{drop } x \]  
\[ \langle P I L K^* M E \rangle \xrightarrow{e} \langle P I (L+1) K^* M E \rangle \xrightarrow{\{x\}} \]  
\[ \text{[DROP]} \]

\[ I(L) = \text{array } x = [e_1, \ldots, e_n] \quad M E e_i \rightarrow v_i \quad \ldots \quad M E e_n \rightarrow v_n \]  
\[ a \text{ fresh } M' \overset{\text{def}}{=} M[a \leftarrow [v_1, \ldots, v_n]] \]  
\[ \langle P I L K^* M E \rangle \xrightarrow{e} \langle P I (L+1) K^* M'E[x \leftarrow a] \rangle \]  
\[ \text{[ARRAY]} \]

\[ I(L) = x \leftarrow e \quad x \in \text{dom}(E) \quad M E e \rightarrow v \]  
\[ \langle P I L K^* M E \rangle \xrightarrow{e} \langle P I (L+1) K^* M E[x \leftarrow v] \rangle \]  
\[ \text{[UPDATE]} \]

\[ I(L) = \text{read } x \]  
\[ \langle P I L K^* M E \rangle \xrightarrow{\text{read } x} \langle P I (L+1) K^* M E[x \leftarrow \text{lit}] \rangle \]  
\[ \text{[READ]} \]

\[ I(L) = \text{branch } e \mid L_1 \mid L_2 \quad M E e \rightarrow \text{true} \]  
\[ \langle P I L K^* M E \rangle \xrightarrow{e} \langle P I L_1 K^* M E \rangle \]  
\[ \text{[BRANCH T]} \]

\[ I(L) = \text{goto } L' \]  
\[ \langle P I L K^* M E \rangle \xrightarrow{e} \langle P I L' K^* M E \rangle \]  
\[ \text{[GOTO]} \]

\[ I(L) = \text{call } x = (e_1, \ldots, e_n) \quad M E e \rightarrow F \]  
\[ P(F) = F(x_1, \ldots, x_n) : \text{Def } x' \overset{\text{def}}{=} P(F, \text{active}) \]  
\[ L' \overset{\text{def}}{=} \text{start}(F') \quad M E'[x_1 = e_1, \ldots, x_n = e_n] \rightarrow E' \]  
\[ \langle P I L K^* M E \rangle \xrightarrow{e} \langle P I L' (K^*(I(L + 1) x E)) M E' \rangle \]  
\[ \text{[CALL]} \]

\[ I(L) = \text{assume } e \text{ else } \xi \xi' \quad \forall m \quad M E e_m \rightarrow \text{true} \]  
\[ \langle P I L K^* M E \rangle \xrightarrow{e} \langle P I (L+1) K^* M E \rangle \]  
\[ \text{[ASSUME PASS]} \]

\[ I(L) = \text{return } e \quad M E e \rightarrow v \]  
\[ \langle P I L (K^*, (I'L' x E')) M E \rangle \xrightarrow{e} \langle P I L' K^* M'E'[x \leftarrow v] \rangle \]  
\[ \text{[RETURN]} \]

\[ I(L) = \text{assume } e \text{ else } \xi \xi' \quad \text{~}(\forall m, M E e_m \rightarrow \text{true}) \]  
\[ \langle P I L K^* M E \rangle \xrightarrow{e} \text{deoptimize}((P I L K^* M E), \xi, \xi') \]  
\[ \text{[DEOPTIMIZE] \]

\[ \text{deoptimize}((P I L K^* M E), F', V', L' VA, \tilde{\xi}_1, \ldots, \tilde{\xi}_r) \overset{\text{def}}{=} (P I'L' (K^*, K_1, \ldots, K_r) M'E') \]  
\[ \text{[DEOPTIMIZE]} \]

\[ M E v A \rightarrow E' \quad l' \overset{\text{def}}{=} P(F', V') \]  
\[ \forall q \in 1, \ldots, r, \]  
\[ \tilde{\xi}_q = F_q V_q L_q X_q VA_q \]  
\[ M E v A \rightarrow E_q \quad l_q \overset{\text{def}}{=} P(F_q V_q) \quad K_q \overset{\text{def}}{=} (L_q X_q V_q) \]  
\[ \text{deoptimize}((P I L K^* M E), F', V', L' VA, \tilde{\xi}_1, \ldots, \tilde{\xi}_r) \overset{\text{def}}{=} (P I'L' (K^*, K_1, \ldots, K_r) M'E') \]  
\[ \text{[DEOPTIMIZE]} \]

\[ [\text{EVALENV}] \]

\[ M E e_1 \rightarrow v_1 \quad \ldots \quad M E e_n \rightarrow v_n \]  
\[ M E [x_1 = e_1, \ldots, x_n = e_n] \rightarrow [x_1 \rightarrow v_1, \ldots, x_n \rightarrow v_n] \]  

Fig. 16. Reduction relation \( C \xrightarrow{\tau} C' \) for sour IR.

5.3 Equivalence of Configurations: Bisimulation

We use the standard proof technique of \textit{weak bisimulation} to prove equivalence between 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, or both terminate), and reducing them results in configurations that are themselves related. Two programs are equivalent if their starting configurations are related.

\begin{definition}[Weak Bisimulation] Given two programs \( P_1 \) and \( P_2 \) and a relation \( R \) between the configurations of \( P_1 \) and of \( P_2 \), we say that \( R \) is a weak simulation if for any related states \((C_1, C_2) \in R\) and any reduction \( C_1 \xrightarrow{A} C_1' \) over \( P_1 \), there exists a reduction \( C_2 \xrightarrow{A*} C_2' \) over \( P_2 \) such that \((C_1', C_2')\) are themselves related by \( R \). 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. In other words, the diagram on the left below can always be completed into the diagram on the right.

\[
\begin{array}{c}
C_1 \xrightarrow{A} C_1' \\
\downarrow R \\
C_2
\end{array}
\quad\quad
\begin{array}{c}
C_1 \xrightarrow{A} C_1' \\
\downarrow R \\
C_2 \xrightarrow{A*} C_2'
\end{array}
\]

We say that \( R \) is a weak bisimulation if it is a weak simulation and the symmetric relation \( R^{-1} \) also is—a reduction from \( C_2 \) can be matched by \( C_1 \). Finally, we say that two configurations are \textit{weakly bisimilar} if there exists a weak bisimulation \( R \) 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.

\begin{lemma}[Transitivity] If \( R_{12} \) is a weak bisimulation between \( P_1 \) and \( P_2 \), and \( R_{23} \) is a weak bisimulation between \( P_2 \) and \( P_3 \), then the composed relation \( R_{13} \) defined as \( (R_{12}; R_{23}) \) is a weak bisimulation between \( P_1 \) and \( P_3 \).
\end{lemma}

\begin{definition}[Version bisimilarity] Let \( V_1, V_2 \) be two versions of a function \( F \) in a program \( P \), and let \( I_1 \overset{\text{def}}{=} P(F, V_1) \) and \( I_2 \overset{\text{def}}{=} P(F, V_2) \). We say that \( V_1 \) and \( V_2 \) are \textit{(weakly) bisimilar} if \( \langle P I_1 \text{ start}(I_1) K* M E \rangle \) and \( \langle P I_2 \text{ start}(I_2) K* M E \rangle \) are weakly bisimilar for all \( K*, M, E \).
\end{definition}

\begin{definition}[Program equivalence] We say that \( P_1 \) and \( P_2 \) are \textit{equivalent} if \( \text{start}(P_1) \) and \( \text{start}(P_2) \) are weakly bisimilar.
\end{definition}

5.4 Deoptimization Invariants

We can now give a formal definition of our two invariants from Section 3.2:

\textbf{Version Equivalence} Any two versions \((V_1, V_2)\) of a function \( F \) are bisimilar as defined in 5.3.

\textbf{Assumption Transparency} For any configuration \( C \) at an \texttt{assume} \( e^* \) else \( \xi \bar{\xi}^* \), \( C \) is bisimilar to \texttt{deoptimize}(\( C, \xi, \bar{\xi}^* \)), as defined in Figure 16, \texttt{DEOPTIMIZECONF}.
5.5 Creating Fresh Versions and Injecting Assumptions

Definition 5.5. We say that the configuration \(C\) is over the location \(F.V.L\) when it is of the form \(\langle P P(F, V) LK^* ME \rangle\), where \(P(F, V)\) denotes the instruction stream at version \(V\) of function \(F\) in program \(P\). We write \(C[F.V.L \leftarrow F'.V'.L']\) for the configuration \(\langle P P(F', V') L'K^* ME \rangle\).

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 5.6. Creating a new copy of the currently active version of a function, possibly adding new assume instructions (see Section 3.3 and Section 3.4), 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 assume instructions, 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 assume instructions that are updated: if \(L\) points to assume \(e^*\) else \(\xi\) \(\xi^*\) in \(V_1\), then it points to assume \(e^*\) else \(F.V_1.L\) \(\text{Id}\) in \(V_2\), where \(\text{Id}\) is the identity mapping over the lexical environment at \(L\).
- \(V_2\) may contain extra empty assume instructions: for any instruction \(i\) at \(L\) in \(V_1\), \(V_2\) may contain an assume of the form assume true else \(F.V_1.L\) \(\text{Id}\), where \(\text{Id}\) is the identity mapping over the lexical environment at \(L\), followed by \(i\) at a fresh label \(L'\).

The bisimulation relation. Let us write \(I_1\) for the instruction sequence of \(V_1\), and \(I_2\) for the instruction sequence of \(V_2\). We add extra assume instructions.

For a stack \(K_1^*\), we say that a stack \(K_2^*\) is a replacement of \(K_1^*\) if it is obtained from \(K_1^*\) by replacing continuation frames of the form \(\langle I_1 L x E \rangle\) (which return into \(V_1\)) by frames of the form \(\langle I_2 L x E \rangle\) (which return into \(V_2\)). Note that this replacement is a device used in the proof and does not correspond with a modification by any of the reduction rules.

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

1. For any configuration \(C_1\) over \(P_1\), \(R\) relates \(C_1\) to \(C_1[P_1 \leftarrow P_2]\).
2. For any configuration \(C_1\) over a \(F.V_1.L\) such that \(L\) in \(V_2\) is not a newly added assume instruction, \(R\) relates \(C_1\) to \(C_1[P_1 \leftarrow P_2][V_1 \leftarrow V_2]\).
3. For any configuration \(C_1\) over a \(F.V_1.L\) such that \(L\) in \(V_2\) is a newly added assume instruction with next label \(L'\), \(R\) relates \(C_1\) to both (a) \(C_1[F.V_1.L \leftarrow F.V_2.L]\) and (b) \(C_1[F.V_1.L \leftarrow F.V_2.L']\).
4. For any related pair \((C_1, C_2)\) \(\in R\), where \(K_1^*\) is the call stack of \(C_2\), for any replacement \(K_2^*\), the pair \((C_1, C_2[K_1^* \leftarrow K_2^*])\) is also in \(R\).

The proof proceeds by showing that \(R\) is a bisimulation.

Definition: if a related pair \((C_1, C_2)\) \(\in R\) comes from the cases (1), (2) or (3) of the definition of \(R\), we say that it is a base pair. If the pair comes from the case (4), we define its base pair as follows. A pair \((C_1, C_2)\) in case (4) is defined from another pair \((C_1, C_2)\) \(\in R\), such that the call stack of \(C_2\) is a replacement of the stack of \(C_2\). If \((C_1, C_2)\) \(\in R\) is a base pair, we say that it is the base pair of \((C_1, C_2)\). Otherwise, we say that the base pair of \((C_1, C_2)\) is the base pair of \((C_1, C_2)\).

Bisimulation proof: generalities. To prove that \(R\) is a bisimulation, we consider all related pairs \((C_1, C_2)\) \(\in R\) and show that a reduction from \(C_1\) can be matched by \(C_2\) and conversely.

Without loss of generality, we can also assume that \(C_2\) is not a newly added assume instruction – that the base pair of \((C_1, C_2)\) is not in the case (3,b) of the definition of \(R\). The proof of
the case (3,b) follows from proof of the case (3,a). In the case (3,b), $C_2$ is a newly added assume instruction assume true else ... at label $L$ with next label $L'$. $C_2$ can only reduce silently into $C_2' \overset{\text{def}}{=} C_2[L \leftarrow L']$, which is related to $C_1$ by the case (3,a). The empty reduction sequence from $C_1$ matches this reduction from $C_2$. Conversely, if we assume the result in the case (3,a), then any reduction of $C_1$ can be matched from $C_2'$, and thus matched from $C_2$ by prepending the silent reduction $C_2 \xrightarrow{r} C_2'$ to the matching reduction sequence. Finally, if $(C_1, C_2)$ comes from case (4) and has a base pair $(C_1, C_2')$ from (3,b), and $C_2$ has label $L$ and next label $L'$, then the bisimulation property for $(C_1, C_2) \in R$ comes from the one of $(C_1, C_2[L \leftarrow L']) \in R$ by the same reasoning.

_Bisimulation proof: easy cases._ The easy cases of the proof are the reductions $C_1 \xrightarrow{A'} C_1'$ where neither $C_1$ nor $C_1'$ are over $V_1$, and the reductions $C_2 \xrightarrow{A'} C_2'$ where neither $C_2$ nor $C_2'$ are over $V_2$. For $C_1 \xrightarrow{A'} C_1'$, we can define $C_2'$ as $C_2'[P_1 \leftarrow P_2]$, and we have both $C_2 \xrightarrow{A'} C_2'$ and $(C_1', C_2') \in R$ as expected. The $C_2 \xrightarrow{A'} C_2'$ case is symmetric, defining $C_1'$ as $C_1'[P_2 \leftarrow P_1]$.

_Bisimulation proof: harder cases._ The harder cases are split in two categories: version-change reductions (deoptimizations, functions call and returns), and same-version reductions within $V_1$ in $P_1$ or $V_2$ in $P_2$. We will consider same-version reductions first.

Without loss of generality, we can assume that the pair $(C_1, C_2) \in R$ is a base pair, that is a pair related by the cases (2) or (3) of the definition of $R$, but not (4) – the case that changes the call stack of the configuration. Indeed, if we have a pair $(C_1, C_2') \in R$ coming from (4), the only difference between this pair and its base pair $(C_1, C_2) \in R$ is in the call stack of $C_2$ and $C_2'$. This means that $C_2$ and $C_2'$ have the exact same reduction behavior for non-version-change reductions. As long as the proof that the related configurations $C_1$ and $C_2$ match each other does not use version-change reductions (a property that holds for the proofs of the non-version-change cases below), it also applies to $C_1$ and $C_2'$.

If we have a reduction $C_2 \xrightarrow{A'} C_2'$ that is not a version-change reduction (deoptimization, call or return), we prove that it can be matched from $C_1$ by reasoning on whether $C_2$ or $C_2'$ are assume instructions, coming from $V_1$ or newly added.

- If none of them are assume instructions, then they are both in the case (2) of the definition of $R$, they are equal to $C_1[V_1 \leftarrow V_2]$ and $C_1'[V_1 \leftarrow V_2]$ respectively, so we have $C_1 \xrightarrow{A'} C_1'$ and $(C_1', C_2') \in R$ as expected.
- If $C_2$ or $C_2'$ are assume instructions coming from $V_1$, the same reasoning holds – the problematic case where the assume is $C_2$ and the guards do not pass is not considered here, as we assumed that the reduction is not a deoptimization.
- If $C_2'$ is a newly added assume in $V_2$ at label $L$ with next label $L'$ (we already eliminated the case where $C_2$ may be), we know that $C_2$ is an instruction of $V_2$ copied from $V_1$, so $(C_1, C_2)$ are in the case (2) of the definition of $R$ and $C_1$ is $C_1[V_2 \leftarrow V_1]$. The reduction from $C_2$ corresponds to a reduction $C_1 \xrightarrow{A'} C_1'$ in $P_1$ with $C_1' \overset{\text{def}}{=} C_2'[V_2 \leftarrow V_1]$, and we have $(C_1', C_2') \in R$ by the case (3,a) of the definition of $R$.

The reasoning for transitions $C_1 \xrightarrow{A'} C_1'$ that have to be matched from $C_2$ and are not version-change transitions (deoptimization, function calls or return) is similar. $C_2$ cannot be a new assume instruction, so we have $C_2 \xrightarrow{A'} C_2'$, and either $C_2'$ is not a new assume instruction and matches $C_1$ by case (2) of the definition of $R$, or it is a new assume instruction and it matches it by the case (3,a).
**Bisimulation proof: final cases.** The cases that remain are the hard cases of version-change reductions: function call, return and deoptimization.

If $C_1 \xrightarrow{A} C'_1$ is a deoptimization reduction, then $C_1$ is over a location $F.V_1.L$ in $P_1$, and its instruction is of the form $\text{assume } e^* \text{ else } \xi \xi'\xi^*$, and $C_1'$ is equal to deoptimize($C_1, \xi, \xi'\xi^*$). $C_2$ is over the copied instruction $\text{assume } e^* \text{ else } F.V_1.L \text{ Id}$, where Id is the identity mapping. $C_2$ also deoptimizes, given that the tests give the same results in the same environment, so we have $C_2 \xrightarrow{\tau} C'_2$ for $C'_2 \overset{\text{def}}{=} \text{deoptimize}(C_2, F.V_1.L \text{ Id}, \emptyset)$. $C'_2$ is over $F.V_1.L$, that is the same assume instruction as $C_1$, so it also deoptimizes, to $C''_2 \overset{\text{def}}{=} \text{deoptimize}(C'_2, \xi, \xi'\xi^*)$. We show that $C'_1$ and $C''_2$ are related by $R$:

- If $(C_1, C_2) \in R$ is a base pair, then we know that $C_1$ is $C_2[V_2 \leftarrow V_1]$. In particular, the two configurations have the same environment, and $C''_2$, which is identical to $C_2$ except it is over $F.V_1.L$, is thus identical to $C_1$. As a consequence, $C'_1$ and $C''_2$, which are obtained from $C_1$ and $C'_2$ by the same deoptimization reduction, are the same configurations, and thus related in $R$.

- If $C_1$ and $C_2$ are related by the case (4) of the definition of $R$, the stack of $C_2$ is a replacement of the stack of $C_1$. The same reasoning as in the previous case shows that the configurations $C'_1$ and $C''_2$ are not identical, but that the stack of $C''_2$ is a replacement of the stack of $C_1$, and the configuration are otherwise identical: they are related by the case (4) of the definition of $R$.

Conversely, if $C_2 \xrightarrow{A'} C'_2$ is a deoptimization instruction then, by the same reasoning as in the proof of matching a deoptimization of $C_1$, we have that $C'_2$ is identical to $C_1$ (modulo replaced stacks). This means that the empty reduction sequence from $C_1$ matches the reduction of $C_2$.

If $C_1 \xrightarrow{A} C'_1$ is a function call transition

$$\langle P_1 I_1 L K^*_1 M E \rangle \xrightarrow{\tau} \langle P_1 I'_1 L' (K^*, (I_1 (L+1) \times E)) M E' \rangle$$

we know that $C_2$ is on the same call instruction with the same arguments, so it takes a transition

$$\langle P_2 I_2 L K^*_2 M E \rangle \xrightarrow{\tau} \langle P_2 I'_2 L' (K^*, (I_2 (L+1) \times E)) M E' \rangle$$

The stack of $C'_2$ a replacement of the stack of $C'_1$: assuming that $K^*_2$ is a replacement of $K^*_1$, the difference in the new stack frame is precisely the definition replacing stacks. Also, the new instruction streams $I'1$ and $I'2$ are either identical (if the function is not $F$ itself) or equal to $I1$ and $I2$ respectively, so we do have $(C'_1, C'_2) \in R$ as expected. Note that this is precisely for this proof case to work that we needed to introduce the case (4) in the definition of $R$. The proof of the symmetric case, matching a function call from $C_2$, is identical.

If $C_1 \xrightarrow{A_r} C'_1$ is a function return transition

$$\langle P_1 I_1 L (K^*, (I'_1 L' x E')) M E \rangle \xrightarrow{\tau} \langle P_1 I'_1 L' K^*_1 M E'[x \leftarrow v] \rangle$$

then $C_2 \xrightarrow{A_r} C'_2$ is also a function return transition

$$\langle P_2 I_2 L (K^*, (I'_2 L' x E')) M E \rangle \xrightarrow{\tau} \langle P_2 I'_2 L' K^*_2 M E'[x \leftarrow v] \rangle$$

We have to show that $C'_1$ and $C'_2$ are related by $R$. The environment and heap of the two configurations are identical; we have to reason on their instruction sequence and call stacks. We know that the stack of $C_2$ is a replacement of the stack of $C_1$, which means that $K^*_2$ a replacement of $K^*_1$, and that either $I'_1$ and $I'_2$ are identical (both transitions are returning to the same place), or they are respectively equal to $I_1$ and $I_2$ (the first returns to $V_1$, and the second was replaced with a return to
V_2). In either case, C'_1 and C'_2 are related by R. The proof of the symmetric case, matching a function return from C_2, is identical.

We have established that R is a bisimulation.

Finally, we remark that our choice of R also proves that assumption transparency is respected by the new version. A new assume at label L in V_2 is of the form \textbf{assume true} else F.V_1.L Id, with Id the identity environment. Any configuration C over F.V_2.L is in relation, by R^{-1}, with C[F.V_2.L ← F.V_1.L], which is equal to deoptimize(C, F.V_2.L Id, ∅). These two configurations are related by R^{-1}, and R^{-1} is a bisimulation, so they are bisimilar.

\[ \text{Lemma 5.7. Adding a new predicate } e' \text{ to an existing assume instruction } \textbf{assume } e^* \text{ else } \xi \tilde{e}' \text{ of } P_1 \text{ returns an equivalent program } P_2. \]

\[ \text{Proof. This is a consequence of the invariant of assumption transparency. Let } R_{P_1} \text{ be the bisimilarity relation for configurations over } P_1, \text{ and } F.V.L \text{ be the location of the modified assume. Let us define the relation } R \text{ between } P_1 \text{ and } P_2 \text{ by} \]

\[ (C_1, C_2) \in R \iff (C_1, C_2[P_2 ← P_1]) \in R_{P_1} \]

We show that R is a bisimulation.

Consider (C_1, C_2) ∈ R. If C_2 is not over F.V.L, the reductions of C_2 (in P_2) and C_2[P_2 ← P_1] (in P_1) are identical, and the latter configuration is, by assumption, bisimilar to C_1, so it is immediate that any reduction from C_1 can be matched by C_2 and conversely.

If C_2 is over F.V.L, we can compare its reduction behavior (in P_2) with the one of C_2[P_2 ← P_1] (in P_1). The first configuration deoptimizes when one of the e', e' is not true in the environment of C_2, while the second deoptimizes when one of the e' is not true – in the same environment. If C_2 gives the same boolean value to both series of test, then the two configurations have the same reduction behavior, and (C_1, C_2) match each other by the same reasoning as in the previous paragraph. The only interesting case is the configurations C_2 that pass all the tests in e'^*, but fail e'. Let us show that, even in that case, the reductions of C_1 and C_2 match each other.

The following diagram will be useful to follow the proof below:

\[ \begin{aligned}
C_1 &\xrightarrow{A_r} C'_1 \\
&\downarrow R \\
C_2 &\xrightarrow{τ} \text{deoptimize}(C_2, \tilde{ξ}, \tilde{ξ}^*) &\xrightarrow{A_r} C''_1 \\
&\downarrow R \\
&\xrightarrow{τ} \text{deoptimize}(C_2, \tilde{ξ}, \tilde{ξ}^*) &\xrightarrow{A_r} C''_1
\end{aligned} \]

Let us first show that the reductions of C_2 can be matched by C_1. The only possible reduction from C_2, given our assumptions, is \( C_2 \xrightarrow{τ} \text{deoptimize}(C_2, \tilde{ξ}, \tilde{ξ}^*) \). We claim that the empty reduction sequence from C_1 matches it, that is, that \((C_1, \text{deoptimize}(C_2, \tilde{ξ}, \tilde{ξ}^*)) \in R\). By definition of R, this goal means that C_1 and deoptimize(C_2, ξ, ξ^*)[P_2 ← P_1] are bisimilar in P_1. But the latter configuration is the same as deoptimize(C_2[P_2 ← P_1], ξ, ξ^*), which is bisimilar to C_2 by the invariant of assumption transparency, and thus to C_1 by transitivity.

Conversely, we show that the reductions of C_1 can be matched by C_2. Suppose that we have a reduction C_1 \xrightarrow{A_r} C'_1. We know that deoptimize(C_2, ξ, ξ^*)[P_2 ← P_1] is bisimilar to C_1 (same reasoning as in the previous paragraph), so there is a matching state C''_1 such that deoptimize(C_2, ξ, ξ^*)[P_2 ← P_1] \xrightarrow{A_r} C''_1 in P_1 with (C'_1, C''_1) ∈ R_{P_1}. We can transpose this reduction in P_2: we have deoptimize(C_2, ξ, ξ^*) \xrightarrow{A_r} C''_1[P_1 ← P_2] in P_2, and thus C_2 \xrightarrow{A_r} C''_1[P_1 ← P_2].
This matches the reduction of $C_1$, given that our assumption $(C'_1, C''_1) \in R_P$ exactly means that $(C'_1, C''_1[P_1 \leftarrow P_2]) \in R$.

\section{Optimization Correctness}

The proofs of the classical optimizations from Section 4 are easier than the proofs for the deoptimization invariants in the previous Section 5.4 (although, as program transformations, they seem more elaborate). This comes from the fact that the classical optimizations rewrite a version on the fly, instead of introducing a new version that is related to an older version, and interact little with deoptimization.

\subsection{Constant Propagation}

\textbf{Definition 6.1.} 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 $SE \models L$, 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 $SE$ by its value. This constant propagation is speculative in the sense that assumption predicates 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.

\textbf{Lemma 6.2.} For any program version $V_1$, let $V_2$ be the result of constant propagation. $V_1$ and $V_2$ are bisimilar.

\textbf{Proof.} 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 $C_2 \stackrel{\text{def}}{=} C_1[V_1 \leftarrow V_2]$ in $\text{reachable}(P_2)$. 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 $C_1 \xrightarrow{A_L} C'_1$ then $C_2 \xrightarrow{A_L} C'_2$, and conversely.

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{verbatim}
L1 var x = 1
  print x + x
  return 3

L1 var x = 1
  print 2
  return 3
\end{verbatim}

Now consider a pair of configurations at label L1 with the binding $x \rightarrow 0$ in the lexical environment.

$C_1 \stackrel{\text{def}}{=} \langle P \ P(F, V_1) \ L_1 \ K^* \ M \ [x \rightarrow 0] \rangle$ \hspace{1cm} $C_2 \stackrel{\text{def}}{=} \langle P \ P(F, V_2) \ L_1 \ K^* \ M \ [x \rightarrow 0] \rangle$

They would be related by the relation $R$ used by the proof, yet they are not bisimilar: we have $C_1 \rightarrow C'_1$ as the only transition of $C_1$ in $V_1$, and $C_2 \rightarrow C'_2$ as the only transition of $C_2$ in $V_2$. 

6.2 Unreachable Code Elimination

**Lemma 6.3.** Replacing \textbf{branch true} \(L_1\ L_2\) by \textbf{goto} \(L_1\) or \textbf{branch false} \(L_1\ L_2\) by \textbf{goto} \(L_1\) results in an equivalent program version.

**Lemma 6.4.** 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 \textbf{branch true} \(L_1\ L_2\) and \textbf{goto} \(L_1\) reduce in the example same way. In the second case, unreachable configurations are not even considered by the proof.

6.3 Function Inlining

In this section, we assume that the function \(F\) has active version \(V_{callee}\). If the new version contains a direct call \textbf{call} \(res = F(e_1, \ldots, e_n)\) to \(F\) with return label \(L_{ret}\) (the label after the call), our inlining pass removes the call and instead:

- it declares a fresh mutable return variable \textbf{var} \(res = \text{nil}\)
- for the formal variables \(x, \ldots\) of \(F\), it defines the argument variables \textbf{var} \(x_1 = se_1, \ldots, \textbf{var} x_n = se_n\).
- then it inserts the instruction stream from \(V_{callee}\), replacing each instruction \textbf{return} \(e\) by the sequence: \(res \leftarrow e; \textbf{drop} x_1; \ldots; \textbf{drop} x_n; \textbf{goto} L_{ret}\)

**Theorem 6.5.** The inlining transformation presented in Section 4.3 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, \(E_{caller}\), and a lexical environment corresponding to the callee, \(E_{callee}\). 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 forms:

- If a configuration is currently executing in the callee, and has the caller on 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 cases:

- 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, \(E_{caller}\) 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 \textbf{var} \(res = \text{nil}\) on the other, we step through the silent transitions that bind the call parameters \textbf{var} \(x_1 = e_1, \ldots, \textbf{var} x_n = e_n\) and get to a state in the inlined function corresponding to the start label of the callee.
- If the transition is a \textbf{return} \(e\) of the callee to the caller on one side, and the entry into the result assignment \(res \leftarrow e\) on the other, we similarly step through the \textbf{drop} \(x\) for each \(x\) in the callee’s environment, and get to related state on the label \(ret\) following the function call.

\(\square\)
6.4 Unrestricted Deoptimization

Consider a program $P_1$ containing a program fragment consisting of an assume instruction at label $L_1$, followed by an instruction $i_m$ at label $L_2 \overset{\text{def}}{=} (L_1 + 1)$ such that $i_m$ has a unique successor, is the unique predecessor of $L_1$, and is not a function call, has no side-effect (read or write), does not modify the array memory (array write or creation), and does not modify the variables mentioned in the assume instruction (tests, environment mapping and extra frames).

Under these conditions, we can move the assume instruction at $L_1$ immediately after its successor instruction $i_m$ at $L_2$. Let us name $P_2$ the program modified in this way.

**Lemma 6.6.** Given a program $P_1$, and $P_2$ obtained by permuting an assume instruction $L_1$ after $i_m$ at $L_2$ under the conditions above, $P_1$ and $P_2$ are bisimilar.

**Proof.** The restrictions on when the transformation may take place are specific enough that we can reason precisely about the structure of reductions around the permuted instructions.

Consider a configuration $C_1$ over the assume instruction at $L_1$ in version $P_1$, and $C_2 \overset{\text{def}}{=} C_1[P_1 \leftarrow P_2][L_1 \leftarrow L_2]$ the corresponding configuration over $L_2$ in $P_2$.

We assumed that $i_m$, the next instruction of $C_2$, has a single successor, so there is only one possible reduction rule that applies. And $i_m$ is not an input-output instruction, it must be a silent action. Hence there is a unique $C_2'$ such that $C_2 \xrightarrow{A_r} C_2'$ holds, and furthermore $A_r$ is $\tau$.

The configurations $C_1$ and $C_2'$ are over the same assume instruction. Let $E_1$ and $E_2$ be the lexical environment of $C_1$ and $C_2'$ respectively, and $E'$ be their common sub-environment that contain only the variables mentioned in the assume instruction (the test and the environment construction); it carries the same values in both configurations as we assumed that $i_m$ does not modify its variables.

If all tests in the assume instruction are true under $E'$, then $C_1$ and $C_2'$ silently reduce to the next instruction; let us call $C_1''$ and $C_2''$ the resulting configurations. $C_1''$ is over $i_m$ at $L_2$, so it can take a unique reduction $C_1'' \xrightarrow{\tau} C_1'''$; notice that $C_1''$ and $C_2''$ are over the labels $(L_2 + 1)$ in $P_1$ and $(L_1 + 1)$ in $P_2$, which we assumed to be equal.

If not all tests of the assume are true under $E'$, then both $C_1$ and $C_2'$ deoptimize. The deoptimized configurations are the same:

- their function, version and label are the same: the assume’s deoptimization target
- they have the same call stack: it only depends on the call stack of $C_1$ and the interpretation of the assume’s extra frames under $E'$
- they have the same array memory, as we assumed that $i_m$ does not modify the array memory
- they have the same deoptimized environment: it only depends on $E'$

Let us call $C_0$ the configuration resulting from either deoptimization transitions.

We establish bisimilarity using the relation $R$ defined as the smallest relation such that:

1. For any $C_1$ and $C_2$ as above, $C_1$ and $C_1'$ are related to $C_2$.
2. For any $C_1$ and $C_2$ as above such that $C_1$ passes the assume tests (does not deoptimize), both $C_1'$ and $C_2'$ are related to $C_1''$.
3. For any $C$ over $P_1$ that is over neither $L_1$ nor $L_2$, $C$ and $C[P_1 \leftarrow P_2]$ are related.

We now prove that $R$ is a bisimulation.
The following diagrams of \( R \) are useful to follow the definition above and the proofs below:

\[
\begin{align*}
L_1 : C_1 & \xrightarrow{\text{assume}} L_2 : C'_1 \xrightarrow{\text{im}} C''_1 & L_1 : C_1 & \xrightarrow{\text{deoptimize}} C_0 \\
\end{align*}
\]

Any pair of configurations that are not over either \( L_1 \) or \( L_2 \) come from the case (3), so they are identical and it is immediate that they match each other. The interesting cases are for matching pairs of configurations over \( L_1 \) or \( L_2 \).

In the case where no deoptimization happens, the reductions in \( P_2 \) are either \( C_2 \xrightarrow{\tau} C'_2 \), where both configurations are related to \( C_1 \), or \( C'_2 \xrightarrow{\tau} C''_2 \) which is matched by \( C_1 \xrightarrow{\tau} C'_1 \). The reductions in \( P_1 \) are either \( C_1 \xrightarrow{\tau} C'_1 \), which is matched by \( C_2 \xrightarrow{\tau} C'_2 \xrightarrow{\tau} C''_2 \) and \( C'_2 \xrightarrow{\tau} C''_2 \), or \( C'_1 \xrightarrow{\tau} C''_1 \), which are both related to \( C''_2 \).

In the case where a deoptimization happens, the only reduction in \( P_1 \) is \( C_1 \xrightarrow{\tau} C_0 \), which is matched by the empty reduction on \( C_1 \) and \( C'_2 \xrightarrow{\tau} C_0 \) are matched by \( C_1 \xrightarrow{\tau} C_0 \).

Finally, we show preservation of the assumption transparency invariant. We have to establish the invariant for \( P_2 \), assuming the invariant for \( P_1 \). We have to show that \( C_0 \) and \( C'_2 \) are bisimilar. \( C_0 \) is bisimilar to \( C_1 \) (this is the transparency invariant on \( P_1 \)), and \( C_1 \) and \( C'_2 \) are bisimilar because they are related by the bisimulation \( R \).

\[\square\]

6.5 Predicate Hoisting

Hoisting predicates 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 assume instructions and \( e \) is a part of the predicate list at \( L_1 \). The pass copies \( e \) from \( L_1 \) to \( L_2 \), if all variables mentioned in \( e \) are in scope at \( L_2 \). If, after this step the \( e \) can be constant folded to true at \( L_1 \) by the optimization from Section 4.1, then it is removed from \( L_1 \), otherwise the whole version stays unchanged.

**Lemma 6.7.** Let \( V_2 \) be the result of hoisting \( e \) from \( L_1 \) to \( L_2 \) in \( V_1 \). \( V_1 \) and \( V_2 \) are bisimilar.

**Proof.** Copying is bisimilar due to the assumption transparency invariant and to the fact that the constant-folded version is bisimilar due to Lemma 6.2. \[\square\]

6.6 Assume Composition

Let \( V_1, V_2, V_3 \) be three versions of a function \( F \) with instruction streams \( I_1, I_2, I_3 \) and \( L_1, L_2, L_3 \) labels, such that \( I_1(L_1) = \text{assume } e_1 \text{ else } F.V_2.L_2 \text{ VA}_1 \text{ and } I_2(L_2) = \text{assume } e_2 \text{ else } F.V_3.L_3 \text{ VA}_2 \).

The composition pass creates a new program \( P_2 \) from \( P_1 \) identical but the assume \( P_2(F.V_1.L_1) \) is replaced by \( \text{assume } e_1, e_2 \text{ else } F.V_3.L_3 \text{ VA}_2 \circ \text{VA}_1 \), where \( ([x_1 = e_1, \ldots, x_n = e_n] \circ \text{VA}) \) is defined as
\[\begin{align*}
x_1 &= e_1 \frac{V_1(y)}{y} \forall y \in \text{VA}, \ldots, x_n &= e_n \frac{V_1(y)}{y} \forall y \in \text{VA}.
\end{align*}\]

**Lemma 6.8.** Let \( P_2 \) be the result of composing assume instructions at \( L_1 \) and \( L_2 \). \( P_1 \) and \( P_2 \) are bisimilar.

**Proof.** For \( C_1 \xrightarrow{\tau} C'_1 \), \( C_2 \xrightarrow{\tau} C'_2 \) over \( L_1 \) in \( P_1, P_2 \), we distinguish four cases:

1) If \( e_1 \) and \( e_2 \) both hold, the assume does not deoptimize in \( P_1 \) and \( P_2 \) and they behave identically.

2) If \( e_1 \) and \( e_2 \) both fail, the original program deoptimizes twice; the modified \( P_2 \) only once.

Assuming deoptimizing under the combined varmap \( MEVA_2 \circ VA_1 \leadsto E'' \) produces an
environment equivalent to $M E VA_1 \leadsto E'$ and $M E' VA_2 \leadsto E''$ the final configuration is identical. Since the extra intermediate step is silent, both programs are bisimilar.

(3) If $e_1$ fails and $e_2$ holds, we deoptimize to $V_3$ in $P_2$, but to $V_2$ in $P_1$. As we have shown in case (2) the deoptimized configuration $C'_2$ over $L_3$ is equivalent to a post-deoptimization configuration of $C'_1$, which, due to assumption transparency is bisimilar to $C'_1$ itself.

(4) If $e_1$ holds and $e_2$ fails, we deoptimize to $V_3$ in $P_2$ but not in $P_1$. Again $C'_2$ is equivalent to a post-deoptimization state, which is, transitively, bisimilar to $C'_1$.

Since a well-formed assume has only unique names in the deoptimization metadata, it is simple to show the assumption in (2) with a substitution lemma. □

7 DISCUSSION

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

7.1 The Cost of Assuming

Assumptions restrict optimizations. Obviously, variables referenced by the deoptimization metadata must be kept alive. Consider the following example in Figure 17, where we have an assume at the end of a loop. If $y$ is never used in this version, we can safely remove it, since we have enough information to synthesize it if needed. On the other hand, we have no hope of synthesizing $x$ out of thin air. We cannot reconstruct the result of a possibly side-effecting call, or function arguments. 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. Additionally, the assume instructions restrict code motion in two cases.

First, side-effecting code cannot be moved over an assume. Second, we cannot hoist assume instructions over instructions that modify variables mentioned in the metadata. It is not hard to move assume forward, since data dependencies can be resolved by taking a snapshot of the environment at the original location. For the reverse effect, we support hoisting the predicate from one assume to another (see Section 4.5). Moving assume instructions up is tricky and also unnecessary, since in combination those two primitives allow moving checks to any position. In the above example, if $e$ is invariant in the loop body and there is an assume before Lloop, the predicate can be hoisted out of the loop. If the predicate is only relevant for a subset of the instructions after the current location, it can be moved down as a whole.

7.2 Lazy Deoptimization and Dependencies

There is also a runtime cost to an assume imposed by checking the predicates, which is compounded if it is inside a loop. Suppose we speculate that the contents of an array 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 deoptimize otherwise, is needlessly wasteful. It would be more efficient for an external event, such as a write to the array, to invalidate the

![Fig. 17. Deoptimization points keep variables alive. Loop-invariant assumptions should be hoisted, to remove assume instructions in loops.](image-url)
assumption, a strategy sometimes known as lazy deoptimization. We could implement dependencies by separating assumptions from runtime checks. Specifically, we let GUARDS[13] = 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. 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.

7.3 Jumping Into Optimized Code

Most of our discussions concerns transferring control out of optimized code and into a less optimized version. We can also consider the inverse transition, where we jump into more optimized code from a less optimized version. Consider executing a long running loop in an unoptimized function, such as in Figure 18. The value of debug is constant inside the loop, yet we are stuck inside the long running function and must branch on each iteration.

A JIT can compile an optimized version that speculates on debug, but it can only be used on the next invocation. Ideally we would like to transfer with execution still in the first invocation, while stuck in the loop; this is an operation known as hot loop transfer. Specifically, the next time we reach Lo of the unoptimized code, we want to transfer to an equivalent location in the optimized version. To do so, continuation-passing style (CPS) can be used to compile a staged continuation function from the beginning of the loop where debug is known to be false. The optimized continuation we jump into might look like continuation in Figure 19. In some sense, this is easier than deoptimization because we strengthen our assumptions instead of weakening them and all the values needed to construct the state at the target version are readily available. Another approach, that should be explored in future work, would be to support some kind of bidirectional deoptimization metadata, such that transfer of control is always possible in both directions.

7.4 Fine-Grained Deoptimization

Instead of blindly removing all assumptions of a version on deoptimization, it is possible to undo only failing assumptions while preserving the rest. As shown in the example in Figure 20, if e2 fails in version Vspec123, we jump to the last version that did not rely on this assumption predicate. By deoptimizing to version Vspec1, we are forced to also discard assumption e3. However, e1, e3 still hold, so we would like to preserve optimizations based on those assumptions. Using the technique mentioned in Section 7.3 we continue executing in a version Lspec13 that reintroduces e3. 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.
7.5 Simulating a Tracing JIT

A tracing JIT [Bala, Duesterwald, and Banerjia 2000; Gal, Eich, Shaver, Anderson, Mandelin, Haghighat, Kaplan, Hoare, Zbarsky, Orendorff, Ruderman, Smith, Reitmaier, Bebenita, Chang, and Franz 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 trace is executed instead. The JIT implementation must ensure that execution follows the same path again, otherwise we must deoptimize out of the trace and into the original version of the program. In this context Guo and Palsberg [2011] develop a framework for reasoning about optimizations applied to traces.

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 \( x \) might be assigned to within a trace, but never used. However, it is unsound to remove the assignment, because \( x \) might be used outside the trace. We can simulate their tracing formalism in Sourir. Consider a variant of their running example shown in Figure 21, a trace of the loop

\[ \text{while } e (x \leftarrow 0; \ldots) \]

embedded in a larger context.

Instead of a runtime that records a sequence of instructions (i.e., a tracing JIT), we expect a runtime that records which branch targets are taken. For this example, suppose we recorded the two targets Lbody and Ldone, which means we executed the loop body once and then exited. In other words, the loop condition \( e \) was true the first time we checked it and false the second time. Therefore we unroll the loop twice and assert \( e \) for the first iteration and \( \neg e \) for the second iteration (left). Then we apply unreachable code elimination to get the following result, which resembles a trace (right):

\[ \ldots \]

\[ \begin{array}{c}
\text{assume } e \text{ else F.Vbase.Lloop } [x = x, \ldots] \\
\text{branch } e \text{ Lbody} \_0 \text{ Ldone} \\
\text{Lbody} \_0 \quad x \leftarrow 0 \\
\ldots
\end{array} \]

\[ \begin{array}{c}
\text{assume } \neg e \text{ else F.Vbase.Lloop } [x = x, \ldots] \\
\text{branch } e \text{ Lbody} \_1 \text{ Ldone} \\
\text{Lbody} \_1 \quad x \leftarrow 0 \\
\ldots
\end{array} \]

\[ \begin{array}{c}
\text{goto } \text{Lloop} \\
\text{Ldone} \quad \ldots
\end{array} \]
Say x is not accessed after the store in this optimized version. In sour ir, it is obvious why dead store elimination of \( x \) would be unsound: the deoptimization metadata indicates that x is still needed when we deoptimize and we can only remove the store if we can replay it then. In this specific example, a constant propagation pass could update the metadata to materialize the write of 0, only when deoptimizing at the second assume. But, 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], but lose more information about instructions outside of the trace.

8 CONCLUSIONS

Speculative optimizations are key to just-in-time optimization of dynamic languages. While there is previous work on formalizations of JITs and on runtime code generation, the formalization of speculation and deoptimization 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 assumptions and show how to build correct speculative optimization passes. We formalize deoptimization invariants between multiple versions of the same function and show that they enable very simple proofs for standard compiler optimizations, constant folding, unreachable code elimination, and function inlining, in the presence of assumptions. We also prove correct three optimizations that are specifically dealing with deoptimizations, namely unrestricted deoptimization, predicate hoisting, and assume composition. Our model of JIT compilation suggests future research directions, such as fine-grained, per-assumption deoptimization and bidirectional deoptimization metadata.

ACKNOWLEDGMENTS

We thank Jean-Marie Madiot for providing guidance on the use and limitations of various notions of bisimulation. In particular, he suggested adding a non-silent stop transition to recover equi-termination from weak bisimilarity. We thank Francesco Zappa Nardelli for his valuable inputs regarding the motivation and presentation of our work. We are greatful to Sewell, Nardelli, Owens, Peskine, Ridge, Sarkar, and Strniša [2007] for providing Ott.

This material is based upon work supported by the National Science Foundation under Grants CCF–1544542, CCF–1318227, CCF–1618732, ONR award 503353, and the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement 695412). Any opinions, findings, and conclusions or recommendations expressed in this material may be those of the authors and likely do not reflect the views of our funding agencies.
REFERENCES


David Schneider and Carl Friedrich Bolz. 2012. The efficient handling of guards in the design of RPython’s tracing JIT. In Workshop on Virtual Machines and Intermediate Languages (VMIL). https://doi.org/10.1145/2414740.2414743


