The Essence of Compiling with Traces

Shu-yu Guo Jens Palsberg
UCLA Computer Science Department
University of California, Los Angeles, USA
{shu,palsberg}@cs.ucla.edu

Abstract

The technique of trace-based just-in-time compilation was introduced by Bala et al. and was further developed by Gal et al. It currently enjoys success in Mozilla Firefox's JavaScript engine. A trace-based JIT compiler leverages run-time profiling to optimize frequently-executed paths while enabling the optimized code to "bail out" to the original code when the path has been invalidated. This optimization strategy differs from those of other JIT compilers and opens the question of *which trace optimizations are sound*. In this paper we present a framework for reasoning about the soundness of trace optimizations, and we show that some traditional optimization techniques are sound when used in a trace compiler while others are unsound. The converse is also true: some trace optimizations are sound when used in a traditional compiler while others are unsound. So, traditional and trace optimizations form incomparable sets. Our setting is an imperative calculus for which tracing is explicitly spelled out in the semantics. We define optimization soundness via a notion of bisimulation, and we show that sound optimizations lead to confluence and determinacy of stores.

Categories and Subject Descriptors D.2.4 [*Program Verification*]: Correctness proofs, formal methods; D.3.4 [*Processors*]: Compilers; F.3.2 [*Semantics of Programming Languages*]: Operational semantics

General Terms Languages, Theory

Keywords just-in-time compilation, compiler correctness, bisimulation

1. Introduction

With the advent of "Web 2.0", the web browser has become a platform that delivers rich interactive applications. The technology central to this transformation of the web browser is JavaScript. JavaScipt's dynamic nature has since then become a performance bottleneck. The performance of dynamic languages is much worse than statically typed languages, and JavaScript is no exception. Moreover, traditional just-in-time (JIT) compilation techniques designed for static, typed languages are ill-fitted for JavaScript.

The work of Bala et al. [1] was adapted as a novel JIT compilation technique called trace compilation [2–4, 7, 8]. A trace-based JIT compiler uses run-time profiling to approximate the "hot" execution paths (loops) in the program and compiles only those paths [4, 7, 8]. The rarely executed bits of code are interpreted. The idea is quite intuitive: if there is a repeatedly executed section of the code, that section should be top priority for compiling to native code. For example, a micro-blogging web application might take many rows of data and transform them into a news feed format. This loop would be where the program spends the majority of its time; a trace-enabled JIT detects that this loop is a hot execution path and compiles it to native code.

Tracing JIT compilers are amenable to JavaScript and enjoy their greatest success in Mozilla Firefox's JavaScript engine (TraceMonkey). It is available in versions 3.5 and greater, and Mozilla metrics report that approximately 94 million people in the world are using the tracing JIT [13].[1]

Tracing JIT compilers differ greatly in technique from many other JIT techniques. It opens the following question:

Which trace optimizations are sound?

We distill the essence of trace compilation to a simple imperative calculus with an operational semantics. This allows us to formally investigate notions of correctness of trace-based JIT compilers and the properties that trace optimizations must satisfy to be sound. We present a bisimulation-based soundness criterion for trace optimizations, and we prove a determinism theorem: whether one traces or not, the final store will be the same.

Our framework is modular in two ways. First, an optimization designer needs only prove that a given optimization satisfies our correctness criterion; the determinism theorem then follows. Second, the composition of two sound optimizations is itself sound. We leverage the first kind of modularity to easily prove soundness of the folding of free loop variables and dead branch elimination. Proving optimizations unsound is equally simple. We show that dead store elimination is unsound with an easy-to-check counterexample. Readers can easily proceed like we did to prove additional trace optimizations sound.

Our proof of the determinism theorem has the following coarse steps. First we prove that an unoptimized, recorded trace of the loop is "behaviorally correct", or bisimilar, to the original loop. We then prove that the original program with the new trace stitched in place of the old loop is bisimilar to the original program. This then sets the stage for sound optimizations: sound optimizations are those that do not invalidate this behavioral correctness guarantee had from bisimilarity. Finally, we put the pieces together and prove confluence and determinacy of stores via a diamond lemma and a strip lemma.

Our framework shows, surprisingly, that "traditional" whole-function optimizations and tracing JIT optimizations *do not stand in a subset relation in either direction*. In one direction, it is clear

POPL'11, January 26–28, 2011, Austin, Texas, USA.

[1] The exact average of daily usage from January 1, 2010 until July 12, 2010 of versions 3.5, 3.6, 3.7, and 4.0 is 93,977,941.

that traditional optimizations are not subsumed by trace optimizations. Informally, trace optimizations are not obliged to be correct for all possible executions and contexts. They are obliged to be correct only for a *particular* execution and a *particular* context. For instance, in the paper we prove folding of variables that are not assigned to be sound for tracing but unsound in general in a traditional setting. In the other direction, trace optimizations are also not subsumed by traditional optimizations. The reason for this is more subtle: the domain of trace optimizations is restricted to *only* the trace. The code surrounding the trace is unavailable to the optimizer. Traditional optimizations, on the other hand, are privy to both the prefix and the suffix of the trace in that their domain *is* the entire procedure. In other words, those optimizations can prove properties on entire procedures while trace optimizations cannot. For one, whole-function optimizations know that their local variables are dead after the function exits. Trace optimizations cannot make the same assumption about their local variables after the trace exits. In the paper we show dead store elimination to be unsound as a trace optimization.

To expand upon the incomparability of the two sets of optimizations, it is illuminating to spell out the differences between our work and recent prominent works in compiler correctness [11, 12]. The program equivalence condition (Lemma 1) found in Lacey et al. [11] basically states classical bisimilarity as the condition under which to judge the correctness of optimizations. An optimized program must be bisimilar to the original and their respective final stores must contain the same values for all variables. In their framework, optimizations are formulated as rewrite rules with side conditions expressed in temporal logic. Despite this difference, we can make the following fruitful comparisons. We write $m \approx n$ *for* σ to mean that the program m is bisimilar to n when their initial stores are σ. Suppose there is an optimization function O. Their correctness criterion describes traditional optimizations and is extensional: for all p, $p \approx O(p)$ *for all stores*. Our correctness criterion describes tracing optimizations and is intensional. Suppose the program p decomposes into two components, w (the traced loop) and k (the rest of the program). The criterion is then: for all w, k, σ, $w\ k \approx O(w, \sigma)\ k$ *for* σ. Note how our optimization function takes a state σ in addition to a program to produce an optimized program guaranteed to be correct when the initial store is fixed to be that store and the rest of the program is fixed to be k. This succinctly captures that trace optimizations may not be straightforwardly used in a classical setting.

The correctness criterion in Chambers et al. [12] raises yet more differences between classical optimization soundness and trace optimization soundness. At the heart of their formulation of soundness is contextual equivalence. However, note that we have said that trace optimization correctness is intensionalized to a particular computation suffix. This necessarily precludes trace optimization correctness from being contextual equivalence. Classical optimizations speculate about the behavior of a program to substitute optimized portions for the original portions *before* execution. Trace optimizations, on the other hand, know exactly the behavior for which they need to optimize and substitute optimized portions *during* execution. This departure also highlights that the input to trace optimizations is mercurial: we do not know *a priori* what loops are hot. The input to classical optimization on the other hand is fixed. In Chambers et al. and Lacey et al. [11, 12], this input is the entire program. As mentioned before, the domains of the two kinds of optimizations are simply different.

The remainder of this paper is organized as follows. In section 2 we introduce our language, its operational semantics, and discuss certain properties of compiling with traces. In section 3 we show that the operational semantics of our language is correct up to weak bisimulation and relate the results to confluence. In section 4 we explore various optimizations, both provably correct and provably incorrect, pluggable into the framework. In section 5 we discuss related work. Section 6 concludes.

2. Compiling with Traces

What is essential to the trace compilation technique? What features must our calculus contain? At its core, it is a method of compiling often-executed loops. We must therefore have loops. More specifically, it is a technique of recording loop bodies at run-time and optimizing them. The second essential part must thus be the ability to record execution. What is optimized, then, is not the text of the loop but a run-time execution path *through* the loop. In other words, we are optimizing some fixed execution. So, when the execution diverges from the recorded path, there must exist a mechanism to return us to the original program. The third and final component is this bail-out mechanism. In the literature of trace compilation, these are called side-exits [8].

We forego modeling the many other features of the technique that exist in implementations. For instance, we shall not model the heuristics in how one actually detects a hot path of execution, we will simply build in the ability to record a path of execution nondeterministically. This nondeterminism will be realized via overlapping reduction rules. Nor shall we model implementation details, such as trace trees and their interactions [7]. We aim to keep the calculus minimal yet high-level, safe—and even desirable—for human consumption.

2.1 The Language and Its Baseline Execution

We first present the syntax and small-step operational semantics for a simple imperative language with two of the three essential ingredients: loops and the ability to "bail out" modeled by continuations.

The syntax of our language, inspired by the calculus presented in Moll [9], is shown in figure 1. Concretely, traces are a subset of normal statements. They are meant to be straightline sections of code with side-exits, so there are only no-ops, assignments, and side-exits.

We use x to range over variables, i to range over non-negative integers, σ, ρ to range over stores, and l, m, n, k, s, p to range over statements throughout the rest of the paper.

The baseline transition rules are shown in figure 2. Assume $\oplus$ is the "real" addition operator on integers. We use a labeled transition system where labels correspond to store updates. We assume the reader is familiar with such systems as they are used in the literature of concurrency [14]. The only observable transitions are store updates, which are labeled by the "store delta". All other transitions are silent, labeled τ, and are unobservable. The subscript $_B$ denotes baseline transition rules. The subscript $_A$ denotes a strict subset of the baseline transition rules that will be used in the upcoming proofs. The subscript $_T$ denotes tracing transition rules. The baseline rules in figure 2 are common to both.

The baseline rules do the usual things. BailTrue is the rule that applies continuations in the **bail**s. It says to clobber the current reduct with the packaged continuation s.

2.2 Recording Traces

We extend the baseline execution with the ability to record traces. The set of baseline rules is a proper subset of the tracing rules, i.e. $\longrightarrow_B \subset \longrightarrow_T$. The abstract syntax is the same between the two languages. The additional transition rules are shown in figure 3.

Starting Traces We start a trace at the beginning of a **while** loop. For technical reasons for the proof of correctness, we record when we have already unfolded at least one iteration of the loop.

Also note that Trace puts the reduction rules in *recording mode*, which is represented syntactically as 4-tuples. The components are,

$$
\begin{array}{ll}
e ::= n \mid x + 1 & \text{expressions} \\
b ::= x = 0 \mid x \neq 0 & \text{boolean expressions} \\
w ::= \textbf{while}\ b\ \textbf{do}\ s & \text{loops} \\
s ::= \epsilon \mid c\ s & \text{statements} \\
c ::= \textbf{skip};\ \mid x := e;\ \mid w \mid \textbf{if}\ b\ \textbf{then}\ s \mid \textbf{bail}\ b\ \textbf{to}\ s & \text{commands} \\
 & \\
t ::= \epsilon \mid c_t\ t & \text{traces} \\
c_t ::= \textbf{skip};\ \mid x := e;\ \mid \textbf{bail}\ b\ \textbf{to}\ s & \text{recorded commands}
\end{array}
$$

Figure 1. Syntax of the Simple Imperative Language and Traces

$$
\hat{\sigma}(e) = \begin{cases} n & \text{if } e = n \\ \sigma(x) \oplus 1 & \text{if } e = x + 1 \end{cases}
\qquad
\hat{\sigma}(b) = \begin{cases} \mathit{true} & \text{if } b \text{ is } x = 0 \wedge \sigma(x) = 0 \\ \mathit{false} & \text{if } b \text{ is } x = 0 \wedge \sigma(x) \neq 0 \\ \mathit{true} & \text{if } b \text{ is } x \neq 0 \wedge \sigma(x) \neq 0 \\ \mathit{false} & \text{if } b \text{ is } x \neq 0 \wedge \sigma(x) = 0 \end{cases}
$$

$$
\begin{array}{ll}
\delta ::= x/i \mid x/\textbf{true} \mid x/\textbf{false} & \text{store updates} \\
\alpha ::= \tau \mid \delta & \text{actions}
\end{array}
$$

$$
\begin{array}{rll}
\langle \sigma, x := e; k\rangle \xrightarrow{\delta}_{T,B,A} \langle \sigma[x/\hat{\sigma}(e)], k\rangle & \text{where } \delta = x/\hat{\sigma}(e) & \text{(Assign)} \\
\langle \sigma, \textbf{skip}; k\rangle \xrightarrow{\tau}_{T,B,A} \langle \sigma, k\rangle & & \text{(Seq)} \\
\langle \sigma, (\textbf{if}\ b\ \textbf{then}\ s)\ k\rangle \xrightarrow{\tau}_{T,B,A} \langle \sigma, k\rangle & \text{if } \hat{\sigma}(b) = \mathit{false} & \text{(IfFalse)} \\
\langle \sigma, (\textbf{if}\ b\ \textbf{then}\ s)\ k\rangle \xrightarrow{\tau}_{T,B,A} \langle \sigma, s\ k\rangle & \text{if } \hat{\sigma}(b) = \mathit{true} & \text{(IfTrue)} \\
\langle \sigma, (\textbf{while}\ b\ \textbf{do}\ s)\ k\rangle \xrightarrow{\tau}_{T,B,A} \langle \sigma, (\textbf{if}\ b\ \textbf{then}\ (s\ \textbf{while}\ b\ \textbf{do}\ s))\ k\rangle & & \text{(While)} \\
\langle \sigma, (\textbf{bail}\ b\ \textbf{to}\ s)\ k\rangle \xrightarrow{\tau}_{T,B,A} \langle \sigma, k\rangle & \text{if } \hat{\sigma}(b) = \mathit{false} & \text{(BailFalse)} \\
\langle \sigma, (\textbf{bail}\ b\ \textbf{to}\ s)\ k\rangle \xrightarrow{\tau}_{T,B} \langle \sigma, s\rangle & \text{if } \hat{\sigma}(b) = \mathit{true} & \text{(BailTrue)}
\end{array}
$$

Figure 2. Shared Transition Rules

$$
\neg b = \begin{cases} x = 0 & \text{if } b \text{ is } x \neq 0 \\ x \neq 0 & \text{if } b \text{ is } x = 0 \end{cases}
$$

$$
\begin{array}{rll}
\langle \sigma, (\textbf{if}\ b\ \textbf{then}\ (s\ (\textbf{while}\ b\ \textbf{do}\ s)))\ k\rangle \xrightarrow{\tau}_{T} \langle \sigma, (\textbf{while}\ b\ \textbf{do}\ s)\ k, \epsilon, s\ (\textbf{while}\ b\ \textbf{do}\ s)\ k\rangle & \text{if } \hat{\sigma}(b) = \mathit{true} & \text{(Trace)} \\
\langle \sigma, k_w, t, x := e; k\rangle \xrightarrow{\delta}_{T} \langle \sigma[x/\hat{\sigma}(e)], k_w, t\ (x := e;), k\rangle & \text{where } \delta = x/\hat{\sigma}(e) & \text{(RecordAssign)} \\
\langle \sigma, k_w, t, \textbf{skip}; k\rangle \xrightarrow{\tau}_{T} \langle \sigma, k_w, t\ (\textbf{skip};), k\rangle & & \text{(RecordSeq)} \\
\langle \sigma, k_w, t, (\textbf{if}\ b\ \textbf{then}\ s)\ k\rangle \xrightarrow{\tau}_{T} \langle \sigma, k_w, t\ (\textbf{bail}\ b\ \textbf{to}\ (s\ k)), k\rangle & \text{if } \hat{\sigma}(b) = \mathit{false} & \text{(RecordIfFalse)} \\
\langle \sigma, k_w, t, (\textbf{if}\ b\ \textbf{then}\ s)\ k\rangle \xrightarrow{\tau}_{T} \langle \sigma, k_w, t\ (\textbf{bail}\ \neg b\ \textbf{to}\ k), s\ k\rangle & \text{if } \hat{\sigma}(b) = \mathit{true} & \text{(RecordIfTrue)} \\
\langle \sigma, k_w, t, (\textbf{while}\ b\ \textbf{do}\ s)\ k\rangle \xrightarrow{\tau}_{T} \langle \sigma, k_w, t\ (\textbf{skip};), (\textbf{if}\ b\ \textbf{then}\ (s\ \textbf{while}\ b\ \textbf{do}\ s))\ k\rangle & \text{if } k_w \neq (\textbf{while}\ b\ \textbf{do}\ s)\ k & \text{(RecordWhile)} \\
\langle \sigma, k_w, t, (\textbf{while}\ b\ \textbf{do}\ s)\ k\rangle \xrightarrow{\tau}_{T} \langle \sigma, O(\textbf{while}\ b\ \textbf{do}\ t, \sigma)\ k\rangle & \text{if } k_w = (\textbf{while}\ b\ \textbf{do}\ s)\ k & \text{(Stitch)} \\
\langle \sigma, k_w, t, k\rangle \xrightarrow{\alpha}_{T} \langle \sigma', k'\rangle & \text{if } \langle \sigma, k\rangle \xrightarrow{\alpha}_{T} \langle \sigma', k'\rangle \wedge k_w \neq k & \text{(Abort)}
\end{array}
$$

Figure 3. Tracing Transition Rules

in order, the store, the stopping point of the trace, the trace thus far, and the current program being reduced.

Recording Traces The recording rules record one command at a time and concatenate it to the end of the trace. Concatenation is simple juxtaposition. The trace itself is a straightline section of code, so we install side-exits (pieces of code that jump back to untraced code when the condition we traced no longer holds) when we record conditionals.

To ease the task of proving correctness, RecordWhile appends a **skip** to the trace while unrolling the loop. Its side condition is to ensure that we are recording an *inner* loop inside the current loop we are tracing, and that we have not come full circle and finished tracing. The work for finishing up a trace is done in Stitch, whose side condition is mutually exclusive with that of RecordWhile.

Ending Well-Behaved Traces We end the trace and stitch it back into the program using Stitch when we finish tracing the body of the loop. We know we have finished when we come back to reducing the same loop that started the trace.

We "compile" the loop that was traced into the same language.[2] The actual optimization is immaterial to the semantics; we assume that there is a sound optimization function $O : (\textit{Statement} \times \textit{Store}) \rightarrow \textit{Statement}$. What soundness entails here will be made precise when we investigate correctness. Informally, soundness means that the output of the O function "does the same thing" as the original code, as far as observable behavior (store updates) goes.

Ending Badly-Behaved Traces We are not guaranteed to finish tracing the body of the loop. That loop body might never terminate! Consider the following example; assume s_2 never changes b to 0.

```
1  a := 1;
2  b := 1;
3  while a ≠ 0 do
4      s1
5      while b ≠ 0 do
6          s2
```

If we start tracing the *outer* loop, once we start executing the *inner* loop we will *never finish the outer loop body*, and thus never finish tracing. Implementations of trace compilation, then, must use heuristics to end the trace if it is continuing on for too long.

In our semantics, we model this by introducing another nondeterministic rule that prematurely stops the trace, Abort. This rule shares the same premises with *all* Record_ rules, where "_" is a wildcard. Note that there are no axioms for recording **bail**s—what this means is that instead of the semantics getting stuck when trying to trace a trace, we abort the trace (that is, we do not model higher-order tracing). Also note that Abort's[3] side condition is mutually exclusive with Stitch, which is intuitively the "good" situation of a successful trace. In this way the rule models the semantics of bailing out of tracing mode for all "bad" situations.

2.3 Example Trace Recording

To help illustrate the tracing rules and to build some concrete intuition, consider the following contrived example.

Example Input

```
1  x := 0;
2  while x = 0 do
3      y := 0;
4      while y = 0 do
5          y := 1;
6      z := 1;
7      b := a + 1;
```

There are two loops; the inner loop only iterates once. The variable a is computed at some earlier point in the program. We give a rule-by-rule walkthrough of tracing the outer loop. We build up the trace in tandem with our walking through of the reduction rules; each snippet that the Record_ rules append to the trace is displayed one by one.

To start, line 1 of the input is matched by Assign, so we reduce by Assign. Line 2 is a **while** loop, which we reduce by While. While converts the loop into an **if** statement testing the condition $x = 0$. This is indeed true by how we mutated the store in line 1, so we can reduce by IfTrue or Trace. In the interest of demonstrating tracing, we reduce by Trace. The trace built thus far is empty, or ϵ. We've only entered recording mode, but we haven't actually recorded any commands yet.

Line 3 in the input is an assignment, which is matched by RecordAssign. RecordAssign appends the assignment itself onto the trace:

Example Trace

```
1  y := 0;
```

Line 4 in the input is the inner loop, and we will now see how the tracing rules deal with recording loops. The loop itself will first reduce to an **if** via RecordWhile, which appends a no-op **skip;** to the trace. In reducing the resulting **if**, we are testing the condition $y = 0$. It is true, so we reduce using RecordIfTrue. The result is that we append a side-exit as a **bail** to the trace. The computation that would have been executed *had the condition been false* gets packaged up as a continuation and gets put into the body of the **bail** (shown indented in the listing):

```
2  skip;
3  bail y ≠ 0 to
4      z := 1;
5      b := a + 1;
6      while x = 0 do
7          y := 0;
8          while y = 0 do
9              y := 1;
10         z := 1;
11         b := a + 1;
```

Now that we have installed the side-exit for entering into the inner loop, we trace the body of the inner loop as straightline code. Line 5 in the input is another assignment, which we record using RecordAssign.

```
12 y := 1;
```

After the body of the inner loop we attempt to reduce the next iteration of that loop. Again, the loop will first reduce to an **if** by RecordWhile. This appends a **skip;**. Unlike the last time, however, the condition $y = 0$ is now false, so we instead reduce using RecordIfFalse. We append another side-exit as before, but the packaged continuation is different. Since the condition was false in the actual execution, we need to include the statement that would have been executed if the condition were true. After that statement we package the rest of the iteration of the outer loop and append it:

```
13 skip;
14 bail y = 0 to
15     y := 1;
16     while y = 0 do
```

[2] Note that this is a simplification in our model. In actual tracing JITs, the compiled code is in machine language.

[3] The rule is modeled as presented instead of the viable alternative of $\langle \sigma, k_w, t, k \rangle \xrightarrow{\tau}_T \langle \sigma, k \rangle$ if $k_w \neq k$ for a cleaner proof of correctness.

$$FV : \textit{Statement} \to \textit{Variables}$$
$$FV(s) = \{x \mid x \text{ is free in } s\}$$

$$F : ((\textit{Expression} + \textit{Statement} + \textit{Command}) \times \textit{Store} \times \mathcal{V}) \to \textit{Statement}$$
$$F(e,\sigma,v) = \begin{cases} n & \text{if } e = n \\ \sigma(x) \oplus 1 & \text{if } e = x + 1 \wedge x \in v \\ e & \text{if } e = x + 1 \wedge x \notin v \end{cases}$$
$$F(s,\sigma,v) = \begin{cases} s & \text{if } s = \epsilon \\ F(c,\sigma,v)\ F(s_1,\sigma,v) & \text{if } s = c\ s_1 \end{cases}$$
$$F(c,\sigma,v) = \begin{cases} x := F(e,\sigma,v) & \text{if } c = x := e \\ c & \text{otherwise} \end{cases}$$

$$O : (\textit{Statement} \times \textit{Store} \times \mathcal{V}) \to \textit{Statement}$$
$$O(s,\sigma) = \begin{cases} \textbf{while}\ b\ \textbf{do}\ F(s_1,\sigma,FV(s_1)) & \text{if } s = \textbf{while}\ b\ \textbf{do}\ s_1 \\ s & \text{otherwise} \end{cases}$$

Figure 4. Variable Folding O

```
17          y := 1;
18      z := 1;
19      b := a + 1;
20      while x = 0 do
21          y := 0;
22          while y = 0 do
23              y := 1;
24          z := 1;
25          b := a + 1;
```

Finally, we apply RecordAssign twice to lines 6–7 and append the assignments to the trace.

```
26  z := 1;
27  b := a + 1;
```

Having successfully traced an iteration of the loop, we now reduce by Stitch to stitch the trace back into the program using the identity as the O function. Abbreviating the continuations for the side-exits as k_i, the final stitched traced loop is as follows.

Abbreviated Stitched and Traced Example Loop

```
1  while x = 0 do
2      y := 0;
3      skip;
4      bail y ≠ 0 to k1
5      y := 1;
6      skip;
7      bail y = 0 to k2
8      z := 1;
9      b := a + 1;
```

2.4 Example O

We have seen the output of tracing, but we obviously want to do more than that. We want to optimize. Consider optimization shown in figure 4 that folds away variables that we never assign to inside a traced loop. First we define a function that calculates the "free" (in the sense of never-assigned-to) variables of a statement. It is assumed to be defined in the usual way. Next we define a helper function F that does the actual optimization. $\mathcal{V}$ is the set of free variables. Finally the O function is just a wrapper around F that calculates and passes in the free variables.

If we apply it to our running example where $a \mapsto 41$, we fold the assignment to b on line 9 of the abbreviated stitch example:

```
9       b := 42;
```

The main benefit of run-time optimization is that we can be more aggressive than with ahead-of-time optimization. Here we presented a simple conservative folding of free loop variables. The idea is that free variables in the loop bodycan be treated as constants and folded until we break out of the loop. We cannot be so bold with a static version of this kind of folding, as we can only do so if we know that the variables we want to fold are constants for the entirety of program execution. Here, however, we only need to know that the variable's value does not change *until the loop is finished*.[4]

3. Correctness

What does it mean for a trace to be correct? First, correctness of the traced code is behavioral correctness—the trace has to "do the same thing" as the original code. Attempting to prove confluence of the program text such as in Pfenning [16] is unfruitful, as there are no guarantees in trace compilation of the traced code converging back to the same text as the original program. In a reactive user-interface, for instance, we might trace-compile many inner loops, and those compiled inner loops might execute forever, waiting for user input. But even in those cases of infinite execution we still want to reason about correctness. The need for infinite executions suggests the tool of bisimilarity.

Second, correctness of the traced code is intensional correctness. Unlike ahead-of-time compiler correctness, we cannot say that an optimized trace is observationally equivalent, or has the same sequence of observable reductions, to the original loop in the traditional, extensional sense. Specifically, an optimized trace need *not* be observationally equivalent to the original loop under all stores. Consider the following version of our little example:

```
1  x := 0;
2  while x = 0 do
3      b := a + 1;
```

A reasonable trace-based optimization if $a \mapsto 41$, as we have seen, would be to replace the loop body with $b := 42$. But this code is

[4] In our simple model, the trace is effectively discarded after the loop exits. There is no way to re-enter a traced loop once it exits. This is not the case in practice, where constructs such as methods allow compiled traces to be called multiple times. In those cases the tracing JIT has to add in more guards and side-exits to guard the folded values. We omit this complexity.

most definitely not observationally equivalent to the original: the original has a free variable, a, and the optimized code $b := 42$ does not. Side-exits are also problematic. How do we ensure that we jump back to the right place in the original code?

We retain the familiar notion of observational equivalence, but parameterize it over stores and computation suffixes. Namely, a trace is correct if it is observationally equivalent to the original loop *for the store that the original loop is currently reducing under* and *for the rest of the program that the original loop would have reduced under*.

To formalize these intuitions, we model correctness using intensionalized bisimulations over stores. Intensionalizing to a particular suffix will be made formal in the definition of O soundness in section 3.3. Bisimulation techniques see popular use in process calculi [14]. There is also existing work in the analysis and correctness proofs of program transformation [6, 21, 22].

The definitions here are built upon, but slightly different from, the standard notions found in the concurrency literature [14], as they are defined over a store. Observational equivalence also becomes formally defined as the notion of bisimilarity. Let the set of labels be defined as follows.

$$\mathit{Act} = \{\delta \mid \delta \text{ is a store update}\} \cup \{\tau\}$$

Definition. If $r \in \mathit{Act}^*$, then $\hat{r}$ is the sequence whereby all occurrences of τ are removed.

Definition. If $r = \alpha_1 \cdots \alpha_n \in \mathit{Act}^*$, we write $m \stackrel{r}{\Longrightarrow} m'$ to mean

$$m \xrightarrow{\tau}{}^* \cdot \xrightarrow{\alpha_1} \cdot \xrightarrow{\tau}{}^* \cdots \xrightarrow{\tau}{}^* \cdot \xrightarrow{\alpha_n} \cdot \xrightarrow{\tau}{}^* m'$$

That is, there may be any number of intervening silent transitions between the observable sequences. In this particular system, the primary observable entity is the store itself, so the intuitive meaning of a program becomes the sequence of store updates it performs.

We only concern ourselves with closed program-store pairs in this paper, where the definition of *closed* is as follows.

Definition. For a store σ and a program m, we say m is σ-closed if for all variables that appear in m, $\hat{\sigma}(x)$ is defined.

For the rest of the paper, when we say "for any store" or "for all stores", we mean for all stores that form closed program-store pairs with the programs under consideration.

Definition (Bisimulation)**.** A *bisimulation* for two reduction relations X, Y is a relation $\mathcal{R}$ such that $\mathcal{R}(\sigma, m, n)$ implies

1. Whenever $\langle \sigma, m\rangle \xrightarrow{\alpha}_X \langle \sigma', m'\rangle$ then, for some n', $\langle \sigma, n\rangle \stackrel{\hat{\alpha}}{\Longrightarrow}_Y \langle \sigma', n'\rangle$ and $\mathcal{R}(\sigma', m', n')$
2. Whenever $\langle \sigma, n\rangle \xrightarrow{\alpha}_Y \langle \sigma', n'\rangle$ then, for some m', $\langle \sigma, m\rangle \stackrel{\hat{\alpha}}{\Longrightarrow}_X \langle \sigma', m'\rangle$ and $\mathcal{R}(\sigma', m', n')$

In the above definition we abuse notation and let m, m', n, and n' range over both statements and triples of statements. That is, since it does not add to the discussion to distinguish between 2-tuples and 4-tuples in the definition, we use a single metavariable to range over both.

The traditional notion of bisimilarity is a special case of this one: two programs are bisimilar in the traditional sense if they are bisimilar for all stores.

Definition. m is said to be *bisimilar* to n under reduction relations X, Y for a store σ, written $m \ {}_X{\approx}_Y \ n$ *for* σ, if $\mathcal{R}(\sigma, m, n)$ for some bisimulation $\mathcal{R}$ on X, Y. In other words,

$${}_X{\approx}_Y \ = \bigcup \{\mathcal{R} \mid \mathcal{R} \text{ is a bisimulation for } X, Y\}$$

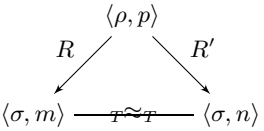
Lemma 3.1. Bisimilarity is an equivalence relation.

Before stating the main lemma, we note that all nondeterministic rules in our system step to the same store. We prove this later in lemma 3.7. For simplicity in stating the main lemma, we simply say that the two branching stores are always the same.

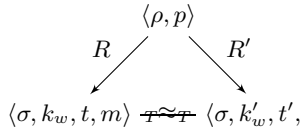
We are now ready to state the main lemma. In the literature, diamond lemmas are usually single diamonds. The trace calculus, however, has a modal flavor with 2-tuples as one mode and 4-tuples as the other mode. As such, our calculus has six diamonds up to symmetry.

Lemma 3.2 (Diamond Lemma)**.** All of the following hold. For diamonds 4–6, $\langle \rho, k_w, t, p\rangle$ is well-formed, a notion we will expound upon in section 3.1.

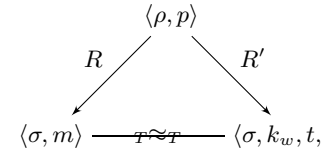
1. If $R : \langle \rho, p\rangle \xrightarrow{\alpha}_T \langle \sigma, m\rangle$ and $R' : \langle \rho, p\rangle \xrightarrow{\alpha}_T \langle \sigma, n\rangle$, then $m \ {}_T{\approx}_T \ n$ for σ.



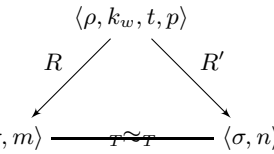
2. If $R : \langle \rho, p\rangle \xrightarrow{\alpha}_T \langle \sigma, k_w, t, m\rangle$ and $R' : \langle \rho, p\rangle \xrightarrow{\alpha}_T \langle \sigma, k'_w, t', n\rangle$, then $\langle k_w, t, m\rangle \ {}_T{\approx}_T \ \langle k'_w, t', n\rangle$ for σ.



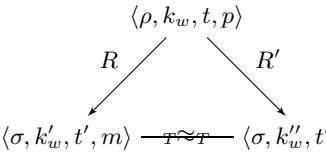
3. If $R : \langle \rho, p\rangle \xrightarrow{\alpha}_T \langle \sigma, m\rangle$ and $R' : \langle \rho, p\rangle \xrightarrow{\alpha}_T \langle \sigma, k_w, t, n\rangle$, then $m \ {}_T{\approx}_T \ \langle k_w, t, n\rangle$ for σ.



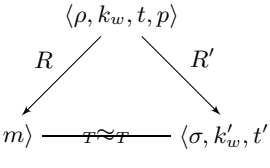
4. If $R : \langle \rho, k_w, t, p\rangle \xrightarrow{\alpha}_T \langle \sigma, m\rangle$ and $R' : \langle \rho, k_w, t, p\rangle \xrightarrow{\alpha}_T \langle \sigma, n\rangle$, then $m \ {}_T{\approx}_T \ n$ for σ.



5. If $R : \langle \rho, k_w, t, p\rangle \xrightarrow{\alpha}_T \langle \sigma, k'_w, t', m\rangle$ and $R' : \langle \rho, k_w, t, p\rangle \xrightarrow{\alpha}_T \langle \sigma, k''_w, t'', n\rangle$, then $\langle k'_w, t', m\rangle \ {}_T{\approx}_T \ \langle k''_w, t'', n\rangle$ for σ.



6. If $R : \langle \rho, k_w, t, p\rangle \xrightarrow{\alpha}_T \langle \sigma, m\rangle$ and $R' : \langle \rho, k_w, t, p\rangle \xrightarrow{\alpha}_T \langle \sigma, k'_w, t', n\rangle$, then $m \ {}_T{\approx}_T \ \langle k'_w, t', n\rangle$ for σ.



This lemma says that should execution branch into two branches, both branches will do the same thing, at least observationally. We aim to use the main lemma to arrive at a more familiar place: confluence of stores, namely corollary 3.10.

The rest of this section is organized as follows. Section 3.1 introduces the idea of well-formedness for the 4-tuples, or the tracing rules. Section 3.2 introduces the correctness criterion of the unoptimized trace. Section 3.3 proves the main lemma. Section 3.4 explores the relationship between confluence and our bisimulation result.

Many proofs in this section are omitted for brevity. The reader may find them in the full version of the paper at `http://www.cs.ucla.edu/~palsberg/paper/popl11.pdf`.

3.1 Well-Formedness of 4-Tuples

When the calculus decides to initiate a trace, it steps to a configuration in the shape of a 4-tuple. The four components are, in order, the store, the point in the original code when we started tracing, the trace so far, and the statement currently being reduced. Not all 4-tuples are created equal, however, as not all 4-tuples are *well-formed*. Intuitively, well-formedness is something like an incremental version of correctness. Only well-formed 4-tuples eventually become fully correct unoptimized traces. Thus, we want it to be an invariant of the computation.

Well-formedness is a tight and intricate relationship between the original loop, the trace thus far, and the current reduct. Informally, we need the trace thus far to be a recording of all the steps that the original loop took just before it reached the current reduct. Before we formally define well-formedness, we formalize what it means to be a "trace thus far".

Definition (Partial Trace Relation)**.** A *partial trace relation* is a relation $\mathcal{T}$ such that $\mathcal{T}(\sigma, t, l)$ implies that whenever $\langle \sigma, t\rangle \xrightarrow{\alpha}_B \langle \sigma', t'\rangle$ then, for some l', $\langle \sigma, l\rangle \xrightarrow{\alpha}_B \langle \sigma', l'\rangle$ and

1. If t stepped by $\mathsf{BailTrue}$, $t' = l'$
2. Otherwise, $\mathcal{T}(\sigma', t', l')$

The constituents are as follows: t is the trace and l is the original code. Recall that both t and l are just statements. The formalization is a variation on the standard simulation relation and captures the two properties that a partially constructed trace intuitively satisfies. First, $\mathsf{BailTrue}$ models jumping back to the original code, so we expect the descendants to be exactly equal. Second, the partial trace is *partial*, so it can terminate before the original code does, signifying that the rest has not yet been traced.

Definition. We call t a *partial trace* to l for a store σ, written $t \precapprox l$ for σ, if $\mathcal{T}(\sigma, t, l)$ for some partial trace relation $\mathcal{T}$. In other words,

$$\precapprox = \bigcup \{\mathcal{T} \mid \mathcal{T} \text{ is a partial trace relation}\}$$

For the definition of well-formedness and subsequent lemmas we will be working with the reduction relation A, which we have not used yet, as well as a notion of being "stuck". Recall that the reduction relation $A = B \setminus \{\mathsf{BailTrue}\}$.

Definition. For a reduction relation X, we say a configuration $\langle \sigma, m\rangle$ is *not X-stuck* iff $m = \epsilon$ or $\langle \sigma, m\rangle \xrightarrow{\alpha}_X \langle \sigma', m'\rangle$ for some σ', m'.

Definition (Well-Formedness)**.** A 4-tuple $\langle \sigma_0, k_w, t, m\rangle$ is *well-formed* iff all the following hold.

1. $k_w = (\textbf{while } b_0 \textbf{ do } l_0)\ k_0$
2. For all σ, either
$$\langle \sigma, t\rangle \xrightarrow{r}{}^*_A \langle \sigma', \epsilon\rangle \text{ and } \langle \sigma, l_0\ k_w\rangle \xrightarrow{r}{}^*_A \langle \sigma', m\rangle$$
or for some t',
$$\langle \sigma, t\rangle \xrightarrow{r'}{}^*_A \langle \sigma'', t'\rangle \text{ and } t' \text{ is } A\text{-stuck but not } B\text{-stuck}$$
3. For all σ, $t \precapprox l_0\ k_w$ for σ.

This property formalizes the invariant we wish computation to preserve. First, k_w must be the loop where we started tracing. Second, the trace t must do one of two things. It must either "go far enough" by terminating in the same reduction sequence that body of the original loop undergoes to reduce to the current statement, m, or it must eventually step to some descendant that can only reduce by $\mathsf{BailTrue}$. Third, t must be a partial trace to the original loop for all stores.

We now prove that the reduction relation T preserves this invariant. We will do this in two steps. First, we prove that the tracing rules themselves—the rules that step from a 4-tuple to another 4-tuple—preserve well-formedness.

Lemma 3.3. Let $p = \langle \sigma_0, k_w, t, m\rangle$. If p is well-formed and $p \xrightarrow{\alpha}_T p'$ such that p' is a 4-tuple, then p' is also well-formed.

Proof. The first conjunct holds trivially because no rules change k_w, and we proceed to prove the next two conjuncts together by case analysis on the structure of the reduction relation. □

Second, we prove that whenever we initiate a trace—whenever we step from a 2-tuple to a 4-tuple—the resulting 4-tuple is well-formed.

Lemma 3.4. If $\langle \sigma, m\rangle \xrightarrow{\alpha}_T p'$ where p' is a 4-tuple, then p' is well-formed.

Proof. By inversion the only rule that results in a 4-tuple is Trace. Let $w = \textbf{while } b \textbf{ do } s$. We have

$$\langle \sigma, (\textbf{if } b \textbf{ then } (s\ w))\ k\rangle \xrightarrow{\tau}_T \langle \sigma, w\ k, \epsilon, s\ w\ k\rangle$$

The first two conjuncts are clearly satisfied. It remains to prove conjunct 3, that $\epsilon \precapprox s\ w\ k$ for all σ. It suffices to exhibit a partial trace relation $\mathcal{T}$ such that for all σ, $\mathcal{T}(\sigma, \epsilon, s\ w\ k)$. Since $t = \epsilon$, we exhibit the empty relation $\emptyset$ as one such $\mathcal{T}$. □

Lemmas 3.3 and 3.4 lead us to a more general lemma about the transitive, reflexive closure of the T reduction relation. This lemma is not used in the rest of the section, but does clearly convey that well-formedness is a property preserved by computation in our calculus.

Lemma 3.5. If $\langle \sigma, m\rangle \xrightarrow{r}{}^*_T p'$ where p' is a 4-tuple, then p' is well-formed.

3.2 Correctness of the Unoptimized Trace

Recall that the intuition for well-formedness is that it is an incremental correctness. With it we can now build up a bisimulation relation.[5]

Lemma 3.6 (Stitch Lemma)**.** For some t, let $w = \textbf{while } b_0 \textbf{ do } l_0$ and $w' = \textbf{while } b_0 \textbf{ do } t$. If for some k, $t \precapprox l_0\ w\ k$ for all σ and (*) holds of $t, l_0\ w\ k, \sigma$ then, $w'\ k\ {}_B{\approx}_B\ w\ k$ for all σ.

(*) For all σ, either

$$\langle \sigma, t\rangle \xrightarrow{r}{}^*_A \langle \sigma', \epsilon\rangle \text{ and } \langle \sigma, l_0\ w\ k\rangle \xrightarrow{r}{}^*_A \langle \sigma', w\ k\rangle$$

[5] The unoptimized trace is in fact strongly bisimilar to the original code. Since we are simply recording some execution path command-for-command, it shouldn't be surprising that the resulting trace is exactly equivalent to the original path. In the interest of less mechanism and since weak bisimilarity subsumes strong bisimilarity, we will directly prove weak bisimilarity.

or for some t',

$$\langle \sigma, t\rangle \xrightarrow{r'}{}^*_A \langle \sigma'', t'\rangle \text{ and } t' \text{ is } A\text{-stuck but not } B\text{-stuck}$$

This lemma is the correctness property we want to express of unoptimized traces. In prose, w is the original loop, and w' is the new loop with the trace stitched in. The $t \lessapprox l_0\ w\ k$ for all σ and (*) conditions are what hold of $t, l_0\ w\ k, \sigma$ per well-formedness at the point of stitching.

The lemma says once we come full circle and stitch the recorded trace into the original program, that trace is actually equivalent to the original loop. The reader should bear in mind that this is an almost extensional equivalence, a stronger notion than intensional equivalence. The insight here is since we have proven a more restrictive property than we need, we can relax it. For fruitful optimization we need to make the relation larger, relaxing extensional bisimilarity to the intensional version.

Proof. By exhibition of a bisimulation relation $\mathcal{R}$ under the relations B, B such that $\mathcal{R}(\sigma, w'\ k, w\ k)$ for all σ. □

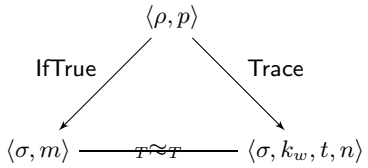
3.3 Proof of the Diamond Lemma

With the tool of bisimilarity under our belt, we can now make precise the notion of the soundness of O. This definition will be crucial for the proof of the diamond lemma.

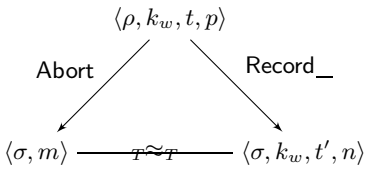
Definition (O-Soundness)**.** An O function is *sound* iff for any w, w', k, σ such that $w'\ k\ {}_B{\approx}_B\ w\ k$ for all stores, $O(w', \sigma)\ k\ {}_B{\approx}_B\ w\ k$ for σ.

Proof of the diamond lemma. If R and R' are the same, then we are done as $m = n$ or $\langle k_w, t, n\rangle = \langle k'_w, t', n'\rangle$. It is straightforward to verify that $R = R'$ for diamonds 1, 2, 4, and 5, which are deterministic, so we only need to prove diamonds 3 and 6. We can rewrite these diamonds more precisely below.

3. The nondeterministic rules are IfTrue and Trace. We have $m = n$ by inversion. Further, by lemma 3.4 $\langle \sigma, k_w, t, n\rangle$ is well-formed.
   If $R : \langle \rho, p\rangle \xrightarrow{\alpha}_T \langle \sigma, m\rangle$ and $R' : \langle \rho, p\rangle \xrightarrow{\alpha}_T \langle \sigma, k_w, t, n\rangle$, then $m\ {}_T{\approx}_T\ \langle k_w, t, n\rangle$ for σ.



6. The nondeterministic rules are Record_ and Abort. We have $k_w = k'_w$ and $m = n$ by inversion. Similarly, by lemma 3.3 $\langle \sigma, k_w, t', n\rangle$ is well-formed.
   If $R : \langle \rho, w, t, p\rangle \xrightarrow{\alpha}_T \langle \sigma, m\rangle$ and $R' : \langle \rho, k_w, t, p\rangle \xrightarrow{\alpha}_T \langle \sigma, k'_w, t', n\rangle$, then $m\ {}_T{\approx}_T\ \langle k'_w, t', n\rangle$ for σ.



To show that ${}_T{\approx}_T$ holds for both diamonds, it suffices to exhibit a bisimulation relation $\mathcal{R}$ under the reductions T, T such that $\mathcal{R}(\sigma, m, \langle k_w, t, n\rangle)$ and $\mathcal{R}(\sigma, m, \langle k_w, t', n\rangle)$ hold.

We claim the following relation is a bisimulation for any $m, n, u, k_v, k_w, t, \sigma$.

$$\begin{aligned}\mathcal{R} = \{&(\sigma, m, n) && \mid m\ {}_B{\approx}_B\ n \text{ for } \sigma\}\\ \cup\ \{&(\sigma, m, \langle k_v, u, n\rangle) && \mid m\ {}_B{\approx}_B\ n \text{ for } \sigma \text{ and}\\ & && \quad \langle \sigma, k_v, u, n\rangle \text{ is well-formed }\}\\ \cup\ \{&(\sigma, \langle k_w, t, m\rangle, n) && \mid m\ {}_B{\approx}_B\ n \text{ for } \sigma \text{ and}\\ & && \quad \langle \sigma, k_w, t, m\rangle \text{ is well-formed }\}\end{aligned}$$

The rest is omitted for brevity. □

3.4 From Bisimulation to Confluence

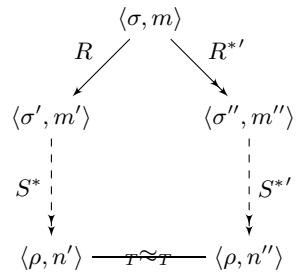
This section aims to be the interface between bisimulation and confluence. As correctness is often studied in terms of determinacy and confluence, we seek here to prove something akin to confluence of stores to show the adequacy of our operational semantics. Traditionally, confluence theorems are proven from the bottom up using a diamond lemma, iterating that diamond lemma to build a strip lemma, and finally using the strip lemma to construct confluence [16]. Bisimilarity, however, allows us to skip the iteration of the single-step diamond lemma. Indeed, there is no analog to an iterable diamond lemma here. We instead use bisimilarity to directly obtain a strip lemma. Nevertheless, the techniques and diagrams in this section are strongly influenced by the clear and readable approach of Pfenning [16].

First we prove the assumption needed for all cases of the diamond lemma, that nondeterministic branching always branches to configurations with the same store.

Lemma 3.7. If $\langle \sigma, m\rangle \xrightarrow{\alpha}_T \langle \sigma', m'\rangle$ and $\langle \sigma, m\rangle \xrightarrow{\alpha'}_T \langle \sigma'', m''\rangle$, then $\sigma' = \sigma''$ and $\alpha = \alpha'$.

Proof. Straightforward case analysis. □

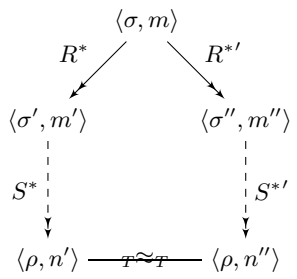
Lemma 3.8 (Strip Lemma)**.** If $R : \langle \sigma, m\rangle \longrightarrow_T \langle \sigma', m'\rangle$ and $R^{*\prime} : \langle \sigma, m\rangle \longrightarrow^*_T \langle \sigma'', m''\rangle$, then for some ρ, n', n'', $\langle \sigma', m'\rangle \longrightarrow^*_T \langle \rho, n'\rangle$ and $\langle \sigma'', m''\rangle \longrightarrow^*_T \langle \rho, n''\rangle$ such that $n'\ {}_T{\approx}_T\ n''$ for ρ.



Proof. By case analysis on the structure of R^* and the definition of ${}_T{\approx}_T$. □

Now we can prove the diamond property for stores on the multistep reduction, which we call *store confluence*.

Theorem 3.9 (Store Confluence)**.** If $R^* : \langle \sigma, m\rangle \longrightarrow^*_T \langle \sigma', m'\rangle$ and $R^{*\prime} : \langle \sigma, m\rangle \longrightarrow^*_T \langle \sigma'', m''\rangle$, then for some ρ, n', n'', $\langle \sigma', m'\rangle \longrightarrow^*_T \langle \rho, n'\rangle$ and $\langle \sigma'', m''\rangle \longrightarrow^*_T \langle \rho, n''\rangle$ such that $n'\ {}_T{\approx}_T\ n''$ for ρ.



Proof. By induction on the structure of R^*. □

Finally, theorem 3.9 implies the familiar notion of store determinacy for terminating programs.

Corollary 3.10. If $\langle \sigma, m\rangle \longrightarrow_T^* \langle \sigma', \epsilon\rangle$ and $\langle \sigma, m\rangle \longrightarrow_T^* \langle \sigma'', \epsilon\rangle$, then $\sigma' = \sigma''$.

4. Sound and Unsound Optimizations

We have achieved our project of proving the essence of trace compilation correct, yet at the same time that result is largely interesting due to its modularity with respect to the O function. In this section we show the example O from section 2.4 to be sound and explore which kinds of O functions are sound and which are not sound.

4.1 Soundness of Variable Folding

Let $F, FV,$ and O be the ones presented in figure 4. For brevity we assume that $FV(s)$ is defined in the usual way and correctly generates the set of free variables for s. That is, for some store σ, $FV(s)$ the set of variables which s never writes to in σ during reduction.

Lemma 4.1. $\longrightarrow_B^*$ is deterministic.

Proof. $\longrightarrow_B$ has no points of nondeterminism. □

Lemma 4.2. O is sound.

Proof. Assuming we have for some w, w', k such that $w'\ k\ {}_B{\approx}_B\ w\ k$ for all stores, we want to show that $O(w', \sigma)\ k\ {}_B{\approx}_B\ w\ k$ for σ.

Our technique will be showing that $O(w', \sigma)\ k\ {}_B{\approx}_B\ w'\ k$, and then obtaining the desired result via transitivity of ${}_B{\approx}_B$.

We proceed by case analysis on the O function.

Case: $s = \textbf{while}\ b\ \textbf{do}\ s_1$.

We want to show that

$$(\textbf{while}\ b\ \textbf{do}\ F(s_1, \sigma, FV(s_1)))\ k\ {}_B{\approx}_B\ (\textbf{while}\ b\ \textbf{do}\ s_1)\ k$$

It suffices to exhibit a bisimulation relation $\mathcal{R}$ such that

$$\mathcal{R}(\sigma, (\textbf{while}\ b\ \textbf{do}\ F(s_1, \sigma, FV(s_1)))\ k, (\textbf{while}\ b\ \textbf{do}\ s_1)\ k)$$

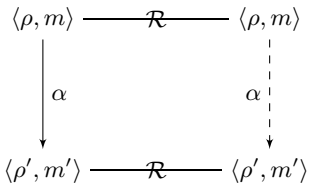
Let $s_1' = F(s_1, \sigma, FV(s_1))$. We claim the following relation is a bisimulation relation for any m, ρ, σ'. Note that σ and the loops are fixed from the assumption.

$$\begin{aligned}
\mathcal{R} = &\{(\rho, m, m)\} \\
&\cup \{(\sigma, (\textbf{while}\ b\ \textbf{do}\ s_1')\ k, \\
&\qquad (\textbf{while}\ b\ \textbf{do}\ s_1)\ k)\} \\
&\cup \{(\sigma, (\textbf{if}\ b\ \textbf{then}\ (s_1'\ \textbf{while}\ b\ \textbf{do}\ s_1'))\ k, \\
&\qquad (\textbf{if}\ b\ \textbf{then}\ (s_1\ \textbf{while}\ b\ \textbf{do}\ s_1))\ k)\} \\
&\cup \{(\sigma', (F(n, \sigma, FV(s_1))\ \textbf{while}\ b\ \textbf{do}\ s_1')\ k, \\
&\qquad (n\ \textbf{while}\ b\ \textbf{do}\ s_1)\ k) \\
&\qquad \mid \sigma'(x) = \sigma(x) \text{ for all free variables in } s_1\}
\end{aligned}$$

We proceed by case analysis on the left-side reduction step.

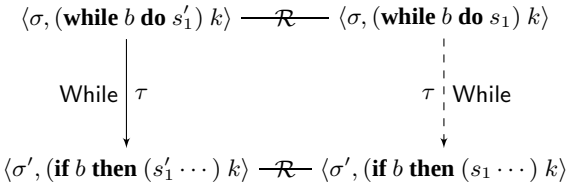
Subcase: The left side and right side are the same.

By lemma 4.1 B is deterministic, both sides step using the same rule, producing the same descendants. But then they are in $\mathcal{R}$ by construction.



Subcase: The left side is $(\textbf{while}\ b\ \textbf{do}\ s_1')\ k$ and the right side is $(\textbf{while}\ b\ \textbf{do}\ s_1)\ k$. Both sides reduce by way of While.

Their descendants are in $\mathcal{R}$ by construction.



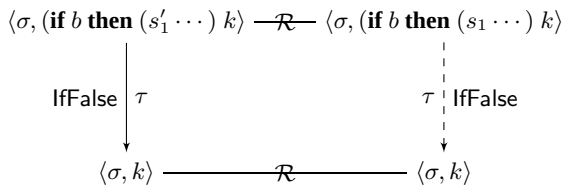
Subcase: The left side is

$$\langle \sigma, (\textbf{if}\ b\ \textbf{then}\ (s_1'\ \textbf{while}\ b\ \textbf{do}\ s_1'))\ k\rangle$$

and the right side is

$$\langle \sigma, (\textbf{if}\ b\ \textbf{then}\ (s_1\ \textbf{while}\ b\ \textbf{do}\ s_1))\ k\rangle$$

Suppose the left side reduce by IfFalse, then both sides step to k, which is already in $\mathcal{R}$ by way of the first subrelation.



Subcase: The left side is

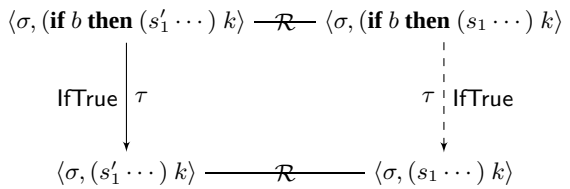
$$\langle \sigma, (\textbf{if}\ b\ \textbf{then}\ (s_1'\ \textbf{while}\ b\ \textbf{do}\ s_1'))\ k\rangle$$

and the right side is

$$\langle \sigma, (\textbf{if}\ b\ \textbf{then}\ (s_1\ \textbf{while}\ b\ \textbf{do}\ s_1))\ k\rangle$$

Suppose the left side reduce by IfTrue, we can then fill out the diagram as follows. The descendants are in $\mathcal{R}$ by way of the fourth subrelation, as

$$s_1' = F(s_1, \sigma, FV(s_1))$$



Subcase: The left side is

$$(F(n, \sigma, FV(s_1))\ \textbf{while}\ b\ \textbf{do}\ s_1')\ k$$

and the right side is

$$(n\ \textbf{while}\ b\ \textbf{do}\ s_1)\ k$$

The left side steps by way of Assign. By inversion, $n = c\ n'$ and

$$F(n, \sigma, FV(s_1)) = F(c, \sigma, FV(s_1))\ F(n', \sigma, FV(s_1))$$

Further, $c = x := e$. For brevity let $n'' = F(n', \sigma, FV(s_1))$.

By case analysis on e we have two subcases. In the case where $e = n$, we have an identity. In the case where $e = x' + 1 \wedge x' \in v$, $e = \sigma(x') \oplus 1$. We know that $\sigma(x')$ is defined from assumption that the left side steps at all. This means the left side step looks like the following. The call to F is abbreviated due to space.

$$\langle \sigma', (c'\ n''\ \cdots)\ k\rangle \xrightarrow{\delta}_B \langle \sigma'[x/\sigma(x') \oplus 1], (n'' \cdots)\ k\rangle$$
$$\text{where } \delta = x/\sigma(x') \oplus 1$$

By inversion then we see that the right side, starting with c, also steps by Assign. By the definition of $\hat{\sigma'}$ we have the following reduction for the right side

$$\langle \sigma', (c\ n' \cdots)\ k\rangle \xrightarrow{\delta'}_B \langle \sigma'[x/\sigma'(x') \oplus 1], (n' \cdots)\ k\rangle$$
$$\text{where } \delta' = x/\sigma'(x') \oplus 1$$

For these two descendants to be in $\mathcal{R}$, we need $\sigma'(x') = \sigma(x')$. We know this to hold for all free variables in s_1, as their freeness guarantees them to be never written to during s_1's reduction. We assumed that FV correctly generates the set of free variables for a statement. It is easy to see that c is a descendant of s_1, thus $x \in FV(s_1)$ and $\sigma'(x') = \sigma(x')$ holds.

Since FV is correct, x is not free in s_1. Therefore,

$$\sigma'[x/\sigma'(x') \oplus 1](y) = \sigma(y)$$

for all y free in s_1. This finally gives us

$$\mathcal{R}(\sigma'[x/\sigma'(x') \oplus 1], n'', n')$$

which holds by way of the fourth subrelation.

For the diagram below, let $v = \sigma(x') \oplus 1$.

$$\begin{array}{ccc}
\langle \sigma', (F(n, \sigma, FV(s_1)) \cdots)\ k\rangle & \not\mathcal{R} & \langle \sigma', (n \cdots)\ k\rangle \\
\Big\downarrow \text{Assign}\ \delta & & \delta\ \text{Assign} \Big\downarrow \\
\langle \sigma'[x/v], (F(n', \sigma, FV(s_1)) \cdots)\ k\rangle & \not\mathcal{R} & \langle \sigma'[x/v], (n' \cdots)\ k\rangle
\end{array}$$

All other cases (where s is something other than a **while** loop) are identities. The proofs for the converses are symmetric.

We have shown $O(w', \sigma)\ k\ {}_B{\approx}_B\ w'\ k$ for σ. By transitivity, we have the desired result of $O(w', \sigma)\ k\ {}_B{\approx}_B\ w\ k$. □

The most interesting part of the proof is that in every subcase we relied on the right side to be able to mirror the left side's move exactly in a single step. This is a stronger property than required by the bisimulation definition, which says only *visible* moves need to be mirrored.

4.2 Soundness of Dead Branch Elimination

What kinds of optimizations only mirror visible moves? One can imagine that during tracing we may generate many spurious side-exits. Suppose we extend our variable folding example to also eliminate "dead", or always-*false* **bail**s. The modifications needed for F are shown in figure 5.

For example, considered the following example trace with a dead side-exit. Clearly x is free in the body of the traced loop, and the boolean expression $x \neq 0$ is always going to be *false*.

Example Trace with Dead Bail

```
1  while x = 0 do
2      bail x ≠ 0 to k1
3      z := 1;
```

Plugging the above example into the extended O function will output the following.

Example Trace with Dead Bail Optimized Away

```
1  while x = 0 do
2      z := 1;
```

Such an optimization does not generate code that exactly mirrors the original. This fails to hold if we wholly excise dead conditionals, as the original code would still need to take a step to evaluate the conditional to false before skipping it. To show that this new optimization is still bisimilar, let us extend lemma 4.2 with the proof sketch of a new subcase and its converse.

New subcase and its converse for lemma 4.2.

Subcase: The left side is

$$(F(n, \sigma, FV(s_1))\ \textbf{while}\ b\ \textbf{do}\ s_1')\ k$$

and the right side is

$$(n\ \textbf{while}\ b\ \textbf{do}\ s_1)\ k$$

The left side takes some step. Let $n = c\ n'$ and

$$F(n, \sigma, FV(s_1)) = F(n', \sigma, FV(s_1))$$

We are concerned with the case when $c = \textbf{bail}\ b'\ \textbf{to}\ k' \wedge F(b', \sigma, FV(s_1)) = \textit{false}$, all other cases for c are identities. For brevity let $n'' = F(n', \sigma, FV(s_1))$.

The left side step looks like the following for some n'''. The call to F is abbreviated due to space.

$$\langle \sigma', (n''\ \cdots)\ k\rangle \xrightarrow{\alpha}_B \langle \sigma'', (n''' \cdots)\ k\rangle$$

By inversion we know that $\hat{\sigma}(b') = \textit{false}$. Since we assumed that FV correctly generates the set of free variables for s_1 and c is a s_1-descendant, so $\hat{\sigma'}(b') = \textit{false}$. By inversion then we see that the right side, starting with c, steps by BailFalse.

$$\langle \sigma', (c\ n' \cdots)\ k\rangle \xrightarrow{\tau}_B \langle \sigma', (n' \cdots)\ k\rangle$$

It remains to show that n' can take a step to match the left side step that $F(n', \sigma, FV(s_1))$ took. We again decompose n' into its first command and continuation. We iteratively apply the same reasoning we just underwent until the first command is not **bail** b'' **to** $k'' \wedge F(b'', \sigma, FV(s_1)) = \textit{false}$. For these other cases F acts as an identity for the first command and as congruence for the continuation, so clearly it will take the same α step.

In the diagram below, let $^+$ mean "1 or more times".

$$\begin{array}{ccc}
\langle \sigma', (F(n', \sigma, FV(s_1)) \cdots)\ k\rangle & \not\mathcal{R} & \langle \sigma', (n \cdots)\ k\rangle \\
\Big\downarrow \alpha & & \tau\ \text{BailFalse}^+ \Big\downarrow \\
 & & \langle \sigma', (n' \cdots)\ k\rangle \\
 & & \Big\downarrow \alpha \\
\langle \sigma'', (F(n'', \sigma, FV(s_1)) \cdots)\ k\rangle & \not\mathcal{R} & \langle \sigma'', (n'' \cdots)\ k\rangle
\end{array}$$

$$F(b,\sigma,v) = \begin{cases} \textit{true} & \text{if } b = x = 0 \wedge x \in v \wedge \sigma(x) = 0 \\ \textit{false} & \text{if } b = x = 0 \wedge x \in v \wedge \sigma(x) \neq 0 \\ \textit{true} & \text{if } b = x \neq 0 \wedge x \in v \wedge \sigma(x) \neq 0 \\ \textit{false} & \text{if } b = x \neq 0 \wedge x \in v \wedge \sigma(x) = 0 \\ \textit{undef} & \text{otherwise} \end{cases}$$

$$F(c,\sigma,v) = \begin{cases} \cdots & \\ \epsilon & \text{if } c = \textbf{bail}\ b\ \textbf{to}\ s_1 \wedge F(b,\sigma,v) = \textit{false} \\ \cdots & \end{cases}$$

Figure 5. Variable Folding extended with Dead Branch Elimination

$$F(c,\sigma,v) = \begin{cases} \cdots & \\ \epsilon & \text{if } c = x := e \wedge x \text{ has no use sites in the trace} \\ \cdots & \end{cases}$$

Figure 6. Variable Folding extended with Dead Branch and Dead Store Elimination

The converse is considerably simpler. We have the case where the right side steps by BailFalse. By the same reasoning above concerning free variables, we see that the left side would have had its **bail** optimized away into ϵ, thus we can complete the diagram by using *Id*.

$$\begin{array}{ccc} \langle \sigma', (F(n', \sigma, FV(s_1)) \cdots)\ k\rangle & \not\mathcal{R} & \langle \sigma', (n \cdots)\ k\rangle \\ \Big\downarrow Id & & \Big\downarrow \tau\ \mathsf{BailFalse} \\ \langle \sigma', (F(n', \sigma, FV(s_1)) \cdots)\ k\rangle & \not\mathcal{R} & \langle \sigma', (n' \cdots)\ k\rangle \end{array}$$

All other cases are still identities. □

4.3 Unsoundness of Dead Store Elimination

Finally, we want to explore what kinds of optimizations are simply unsafe in the tracing framework. Put formally, we want to ask what kind of optimizations do not produce bisimilar code. Continuing with our existing O function, suppose we were to extend it with dead store elimination. That is, suppose variables that we assign to but have no use sites inside the trace body are simply excised. This is shown informally in figure 6.

For example, consider the following example trace with a dead assignment. The variable z is assigned but never used.

Example Trace with Dead Assignment

```
1  while x = 0 do
2      z := 1;
```

Plugging the above example into the extended O function will output the following.

Example Trace with Dead Assignment Optimized Away

```
1  while x = 0 do
2      ϵ
```

Intuitively this is unsafe because even though z is dead inside the trace, there very well may be use sites of z after the trace! This intuition is reflected formally. Taking our example above, we need to show that $z := 1;$ takes a step that can be mirrored by ϵ. For some σ, by inversion $z := 1$ can step only by Assign: $\langle \sigma, z := 1\rangle \xrightarrow{\tau}_B \langle \sigma[z/1], \epsilon\rangle$. ϵ needs to be able to match this move, but $\langle \sigma, \epsilon\rangle \stackrel{\hat{\tau}}{\not\Longrightarrow}_B \langle \sigma[z/1], s\rangle$ for any s. In fact, it does not step at all.

In this fashion this optimization does not output bisimilar code, and is not safe for use inside the tracing framework.

4.4 Soundness of Composition

One property that correct optimizations enjoy in our framework is that the composition of two correct optimizations also yield a correct optimization. We give the following two lemmas to demonstrate this property.

Lemma 4.3. Let $I : (\textit{Statement} \times \textit{Store}) \to \textit{Statement}$ be the identity function on its first argument. I is sound.

Proof. Trivial by the definition of O-soundness. □

Lemma 4.4. Let $F, G : (\textit{Statement} \times \textit{Store}) \to \textit{Statement}$ be two sound optimizations. Let their composition be defined as

$$F \circ G = \lambda(s,\sigma).F(G(s,\sigma),\sigma)$$

$F \circ G$ is sound.

Proof. We want to show that for any w, w', k such that $w'\ k\ {}_B{\approx}_B\ w\ k$ for all stores, $(F \circ G)(w', \sigma)\ k\ {}_B{\approx}_B\ w\ k$ for σ.

By soundness of G on w, w', k we know that

$$G(w', \sigma)\ k\ {}_B{\approx}_B\ w\ k$$

By soundness of F on $w, G(w', \sigma), k$ we then know that

$$F(G(w', \sigma), \sigma)\ k\ {}_B{\approx}_B\ w\ k$$

But $F(G(w', \sigma), \sigma)\ k = (F \circ G)(w', \sigma)$, so we are done. □

5. Related Work

The work carried out in this paper depends on both compiler-correctness and concurrency techniques. Though the corpora of both communities are large, there is a dearth of truly relevant papers that explore purely operational compiler correctness of JIT compilers from as a high-level as ours. Nevertheless, we have taken inspiration as well as fruitful comparisons with several works.

Relevant is Wand's work on parallel compiler correctness [20]. We believe Wand's enterprise to be, though also employing bisimulations to prove compiler-correctness, of a different flavor than our own. His approach is the combination of (syntax-directed) denotational semantics and essentially β-convertibility. His picture is also

closer to the traditional picture of compiler correctness [5]—that is, the compilation process preserves denotation up to bisimulation—than ours, as his compiler is an ahead-of-time compiler. The elegance of Wand's work is that he recognized that β-convertibility *induces* bisimilarity; in the conclusion he admits that almost all the required reasoning is done in the λ-caculus and as such, he can re-use work already done in sequential compiler correctness.

Unlike Wand's work, our vision of correctness is purely syntax-directed: the translation itself (if the JIT tracing can be seen as such) becomes a non-instantaneous process since we have to spell it out in the operational semantics. This is what makes the enterprise non-trivial. Our notion of convertibility informally becomes something akin to "store-convertiblity", but this is much less powerful than β-convertibility as it does not directly imply bisimilarity. We also do not have the luxury of bringing to bear the entirety of the λ-calculus machinery, so our technique here, while still using bisimulations, is at once more basic and less elegant.

There is also a breadth of literature exploring using bisimulation to show program equivalence by way of contextual equivalence in Pierce et al., Lassen et al., and Wand et al. [10, 17–19]. Though the topics of their specific investigations differ, they all concern themselves with using bisimulation as a more tractable proof technique to prove contextual equivalence without having to universally quantify over all contexts. Their setting is higher-ordered, modeled within the λ-calculus. Pierce et al. [19], for instance, aims to prove bisimiliarity sound and complete with respect to contextual equivalence for a modified λ-calculus with recursive types. Wand [10] aims to improve the proof technique and reasons about a λ-calculus extended with explicit stores. These works are basic investigations into the nature of the proof technique. Ours is an application of the technique to prove equivalence of a dynamically transformed program. We also arrived at bisimiliarity by an entirely different motivation, that of proving determinism of a JIT compiler that performs the dynamic transform. We are not met with the difficulty of universally quantifying contexts; in fact, we fix our correctness to hold for one context only.

Myreen's method of creating formally correct JIT compilers for x86 [15] is at the much lower level of abstraction: machine language. They use Hoare logic, and so still retain a flavor of the denotational. We are much farther from the "bare metal" than they are.

6. Conclusions and Future Work

We have demonstrated a paradigm for high-level, purely operational correctness of the tracing JIT compilation technique via bisimulation and confluence. Unlike traditional ahead-of-time compiler correctness where the translation process from the source language to the target language is an opaque function, trace compiler-correctness requires the translation—the tracing—to be spelled out explicitly. We overcome this difficulty by using bisimulations, though we strive to maintain continuity with existing purely operational correctness approaches by returning to confluence.

We hope that the theoretical framework we have provided will prove useful in reasoning about trace compilers at a high level. We hope that we have opened up a wealth of possible future research in the foundational differences betweens traditional and trace optimizations. Though a different problem, we also feel applying the trace compilation technique to an applicative setting, namely the λ-calculus, will be a worthy venture. It is also interesting to further explore O and the question of just what exactly is observable in computation. We also hope to look at deriving tools from the techniques described here in the future.

Acknowledgements. We thank Michael Bebenita and the Mozilla JavaScript team for enlightening discussions about the implementation of trace compilers. We also thank Jonathan Lee, Oren Freiberg, Kannan Goudan, Dave Herman, Dimitris Vardoulakis, and the anonymous reviewers for draft reading and helpful discussions.

References

[1] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *PLDI '00*, pages 1–12. ACM, 2000.

[2] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. SPUR: A trace-based JIT compiler for CIL. In *OOPSLA '10*, 2010.

[3] Michael Bebenita, Mason Chang, Gregor Wagner, Christian Wimmer, Andreas Gal, and Michael Franz. Trace-based compilation in execution environments without interpreters. In *PPPJ '10*, 2010.

[4] Mason Chang, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Tracing for web 3.0: trace compilation for the next generation web applications. In *VEE*, pages 71–80, 2009.

[5] Joëlle Despeyroux. Proof of translation in natural semantics. In *LICS*, pages 193–205, 1986.

[6] Cormac Flanagan and Matthias Felleisen. The semantics of future and its use in program optimization. In *POPL '95*, pages 209–220. ACM, 1995.

[7] Andreas Gal. *Efficient bytecode verification and compilation in a virtual machine*. PhD thesis, 2006. Adviser: Michael Franz.

[8] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI '09*, pages 465–478. ACM, 2009.

[9] A. J. Kfoury, Michael A. Arbib, and Robert N. Moll. *A Programming Approach to Computability*. Springer-Verlag, 1982.

[10] Vasileios Koutavas and Mitchell Wand. Small bisimulations for reasoning about higher-order imperative programs. In *POPL '06*, pages 141–152. ACM, 2006.

[11] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *POPL '02*, pages 283–294. ACM, 2002.

[12] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *PLDI '03*, pages 220–231. ACM, 2003.

[13] Mozilla Metrics. Firefox usage: https://metrics.mozilla.com/.

[14] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1995.

[15] Magnus O. Myreen. Verified just-in-time compiler on x86. In *POPL '10*, pages 107–118. ACM, 2010.

[16] Frank Pfenning. A proof of the Church-Rosser theorem and its representation in a logical framework. *Journal of Automated Reasoning*, 1993.

[17] Kristian Støvring and Soren B. Lassen. A complete, co-inductive syntactic theory of sequential control and state. In *POPL '07*, pages 161–172. ACM, 2007.

[18] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for dynamic sealing. In *POPL '04*, pages 161–172. ACM, 2004.

[19] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for type abstraction and recursion. In *POPL '05*, pages 63–74. ACM, 2005.

[20] Mitchell Wand. Compiler correctness for parallel languages. In *FPCA*, pages 120–134, 1995.

[21] Mitchell Wand and William D. Clinger. Set constraints for destructive array update optimization. *Journal of Functional Programmng*, 11(3):319–346, 2001.

[22] Mitchell Wand and Igor Siveroni. Constraint systems for useless variable elimination. In *POPL '99*, pages 291–302. ACM, 1999.