

Automatically Proving the Correctness of Compiler Optimizations

Sorin Lerner Todd Millstein Craig Chambers
Department of Computer Science and Engineering
University of Washington
{lerns,todd,chambers}@cs.washington.edu

Technical Report UW-CSE-02-11-02

Abstract

We describe a technique for automatically proving compiler optimizations *sound*, meaning that their transformations are always semantics-preserving. We first present a domain-specific language, called Cobalt, for implementing optimizations as guarded rewrite rules. Cobalt optimizations operate over a C-like intermediate representation including unstructured control flow, pointers to local variables and dynamically allocated memory, and recursive procedures. Then we describe a technique for automatically proving the soundness of Cobalt optimizations. Our technique requires an automatic theorem prover to discharge a small set of simple, optimization-specific proof obligations for each optimization. We have written a variety of forward and backward intraprocedural dataflow optimizations in Cobalt, including constant propagation and folding, branch folding, full and partial redundancy elimination, full and partial dead assignment elimination, and simple forms of points-to analysis. We implemented our soundness-checking strategy using the Simplify automatic theorem prover, and we have used this implementation to automatically prove our optimizations correct. Our checker found many subtle bugs during the course of developing our optimizations. We also implemented an execution engine for Cobalt optimizations as part of the Whirlwind compiler infrastructure.

1 Introduction

Compilers are an important part of the infrastructure relied upon by programmers. If a compiler is faulty, then so are potentially all programs compiled with it. Unfortunately, compiler errors can be difficult for programmers to detect and debug. First, because the compiler's output cannot be easily inspected, problems can often be found only by running a compiled program. Second, the compiler may appear to be correct over many runs, with a problem only manifesting itself when a particular compiled program is run with a particular input. Finally, when a problem does appear, it can be difficult to determine whether it is an error in the compiler or in the source program that was compiled.

For these reasons, it is very useful to develop tools and techniques that give compiler developers and programmers confidence in their compilers. One way to gain confidence in the correctness of a compiler is to run it on various programs and check that the optimized version of each program produces correct results on various inputs. While this method can increase confidence, it cannot provide any guarantees: it does not guarantee the absence of bugs in the compiler, nor does it even guarantee that any one particular optimized program is correct on all inputs. It also can be tedious to assemble an extensive test suite of programs and program inputs.

Translation validation [25, 19] and credible compilation [27, 26] improve on this testing approach by automatically checking whether or not the optimized version of an input program is semantically equivalent to the original program. These techniques can therefore guarantee the correctness of certain optimized programs, but the compiler itself is still not guaranteed to be bug-free: there may exist programs for which the compiler produces incorrect output. There is little recourse for a programmer if a compiled program cannot be validated. Furthermore, these approaches can have a substantial impact on the time to run an optimization.

The best solution would be to prove the compiler *sound*, meaning that for any input program, the compiler always produces an equivalent output program. Optimizations, and sometimes even complete compilers, have been proven sound by hand [1, 2, 15, 13, 8, 23, 3, 11]. However, manually proving large parts of a compiler sound requires a lot of effort and theoretical skill on the part of the compiler writer. In addition, these proofs are usually done for optimizations as written on paper, and bugs may still arise when the algorithms are implemented from the paper specification.

We present a new technique for proving the soundness of compiler optimizations that combines the benefits from the last two approaches: our approach is fully automated, as in credible compilers and translation validation, but it also proves optimizations correct once and for all, for *any* input program. We achieve this goal by providing the compiler writer with a domain-specific language for implementing optimizations that is both flexible enough to express a variety of optimizations and amenable to automated correctness reasoning.

The main contributions of this paper are as follows:

- We present a language, called Cobalt, for defining optimizations over programs expressed in a C-like intermediate language including unstructured control flow, pointers to local variables and dynamically allocated memory, and recursive procedures. To implement an optimization (i.e., an analysis plus a code transformation), users provide a rewrite rule along with a guard describing the conditions that must hold for the rule to be triggered at some node of an input program’s control-flow graph (CFG). The optimization also includes a small predicate over program states, which captures the key “insight” behind the optimization that justifies its correctness. Cobalt also allows users to express pure analyses, such as pointer analysis. Pure analyses can be used both to verify properties of interest about a program and to provide information to be consumed by later transformations. Optimizations and pure analyses written in Cobalt are directly executable by a special dataflow analysis engine written for this purpose; they do not need to be reimplemented in a different language to be run.
- We have used Cobalt to express a variety of intraprocedural forward and backward dataflow optimizations, including constant propagation and folding, copy propagation, common subexpression elimination, branch folding, partial redundancy elimination, partial dead assignment elimination, and loop-invariant code motion. We have also used Cobalt to express several simple intraprocedural pointer analyses, whose results we exploited in the above optimizations.
- We present a strategy for automatically proving the soundness of optimizations and analyses expressed in Cobalt. The strategy requires an automatic theorem prover to discharge a small set of proof obligations for each optimization. We have manually proven that if these obligations hold for any particular optimization, then that optimization is sound. The manual proof takes care of the necessary induction over program execution traces, which is difficult to automate. As a result, the automatic theorem prover is given only noninductive theorems to prove about individual program states.
- We have implemented our correctness checking strategy using Simplify [30, 22], the automatic theorem prover used in the Extended Static Checker for Java [6]. We have written a general set of axioms that are used by Simplify to automatically discharge the optimization-specific proof obligations generated by our strategy. The axioms simply encode the semantics of programs in our intermediate language.

New optimization programs can be written and proven sound without requiring any modifications to Simplify’s axiom set.

- We have used our correctness checker to automatically prove correct all of the optimizations and pure analyses listed above. The correctness checker uncovered a number of subtle problems with earlier versions of our optimizations that might have eluded manual testing for a long time.
- We have implemented an execution engine for Cobalt optimizations as part of the Whirlwind compiler infrastructure, and we have used it to successfully execute all of our optimizations.

By providing greater confidence in the correctness of compiler optimizations, we hope to provide a foundation for *extensible compilers*. An extensible compiler would allow users to include new optimizations tailored to their applications or domains of interest. The extensible compiler can protect itself from buggy user optimizations by verifying their correctness using our strategy; any bugs in the resulting extended compiler can be blamed on other aspects of the compiler’s implementation, not on the user’s optimizations. Extensible compilers could also be a good vehicle for research into new compiler optimizations.

The next section introduces Cobalt by example and sketches our strategy for automatically proving soundness of Cobalt optimizations. Sections 3 and 4 formally define Cobalt and our automatic proof strategy, respectively. Section 5 discusses our implementation of Cobalt’s execution engine and correctness checker. Section 6 evaluates our work, and section 7 discusses future work. Section 8 describes related work, and section 9 offers our conclusions.

2 Overview

In this section, we informally describe Cobalt and our technique for proving Cobalt optimizations sound through a number of examples. Appendix B contains the complete definitions of all the optimizations and analyses we have written in Cobalt.

2.1 Forward Transformation Patterns

2.1.1 Semantics

The heart of a Cobalt optimization is its *transformation pattern*. For a forward optimization, a transformation pattern has the following form:

ψ_1 **followed by** ψ_2 **until** $s \Rightarrow s'$ **with witness** \mathcal{P}

A transformation pattern describes the conditions under which a statement s may be transformed to s' . The formulas ψ_1 and ψ_2 , which are properties of a statement such as “ x is defined and y is not used,” together act as the guard indicating when it is legal to perform this transformation: s can be transformed to s' if on all paths in the CFG from the start of the procedure being optimized to s , there exists a statement satisfying ψ_1 , followed by zero or more statements satisfying ψ_2 , followed by s . Figure 1 shows this scenario pictorially.

Forward transformation patterns codify a scenario common to many forward dataflow analyses: an *enabling* statement establishes the conditions necessary for a transformation to be performed downstream, and any intervening statements are *innocuous*, i.e., do not invalidate the conditions. The formula ψ_1 captures the properties that make a statement enabling, and ψ_2 captures the properties that make a statement innocuous. The *witness* \mathcal{P} captures the conditions established by the enabling statement that allow the transformation to be safely performed. Witnesses have no effect on the semantics of an optimization; they will be discussed more below in the context of our strategy for automatically proving optimizations sound.

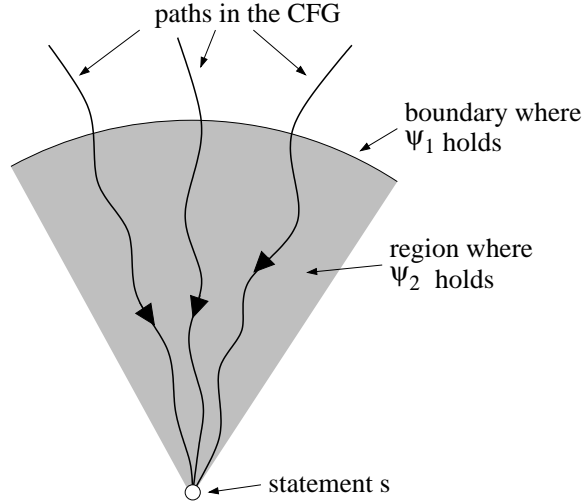


Figure 1: CFG paths leading to a statement s which can be transformed to s' by the transformation pattern ψ_1 **followed by** ψ_2 **until** $s \Rightarrow s'$ **with witness** \mathcal{P} . The shaded region can only be entered through a statement satisfying ψ_1 , and all statements within the region satisfy ψ_2 . The statement s can only be reached by first passing through this shaded region.

Example 1 A simple form of constant propagation replaces statements of the form $X := Y$ with $X := C$ if there is an earlier (enabling) statement of the form $Y := C$ and each intervening (innocuous) statement does not modify Y . The enabling statement ensures that variable Y holds the value C , and this condition is not invalidated by the innocuous statements, thereby allowing the transformation to be safely performed downstream. This sequence of events is expressed by the following transformation pattern (the witness is discussed in more detail in section 2.1.2):

$stmt(Y := C)$
followed by
 $\neg mayDef(Y)$
until
 $X := Y \Rightarrow X := C$
with witness
 $\eta(Y) = C$

The “pattern variables” X and Y may be instantiated with any variables of the procedure being optimized, while the pattern variable C may be instantiated with constants in the procedure.

2.1.2 Soundness

A transformation pattern is *sound*, i.e., correct, if all the transformations it allows are semantics-preserving. Forward transformation patterns have a natural approach for understanding their soundness. Consider a statement s transformed to s' . Any execution trace of the procedure that contains s' will at some point execute an enabling statement, followed by zero or more innocuous statements, before reaching s' . As

mentioned earlier, executing the enabling statement establishes some conditions at the subsequent state of execution. These conditions are then preserved by the innocuous statements. Finally, the conditions imply that s and s' have the same effect at the point where s' is executed. As a result, the original program and the transformed program have the same semantics.

Our automatic strategy for proving optimizations sound is based on the above intuition. As part of the code for a forward transformation pattern, optimization writers provide a *forward witness* \mathcal{P} , which is a (possibly first-order) predicate over an execution state, denoted η . The witness plays the role of the conditions mentioned in the previous paragraph and is the intuitive reason why the transformation pattern is correct. Our strategy attempts to prove that the witness is established by the enabling statement and preserved by the innocuous statements, and that it implies that s and s' have the same effect.¹ We call the region of an execution trace between the enabling statement and the transformed statement the *witnessing region*. In figure 1, the part of a trace that is inside the shaded area is its witnessing region.

In example 1, the forward witness $\eta(Y) = C$ denotes the fact that the value of Y in execution state η is C . Our implementation proves automatically that the witness $\eta(Y) = C$ is established by the statement $Y := C$, preserved by statements that do not modify the contents of Y , and implies that $X := Y$ and $X := C$ have the same effect. Therefore, the constant propagation transformation pattern is automatically proven to be sound.

2.1.3 Labels

Each node in a procedure's CFG is *labeled* with properties that are true at that node, such as $stmt(x := 5)$ or $mayDef(y)$. The formulas ψ_1 and ψ_2 in an optimization are boolean expressions over these labels.

Users can define a new kind of label by giving a predicate over a statement, referred to in the predicate's body using the distinguished variable $currStmt$. As a trivial example, the $stmt(S)$ label, which denotes that the statement at the current node is S , can be defined as:

$$stmt(S) \triangleq currStmt = S$$

As another example, $syntacticDef(Y)$, which stands for syntactic definition of Y , can be defined as:

$$\begin{aligned} syntacticDef(Y) \triangleq & \text{ case } currStmt \text{ of} \\ & \text{ decl } X \quad \Rightarrow \quad X = Y \\ & X := \dots \quad \Rightarrow \quad X = Y \\ & \text{ else} \quad \Rightarrow \quad false \\ & \text{ endcase} \end{aligned}$$

The label $syntacticDef(Y)$ holds at a node if and only if the current statement is a declaration of or an assignment to Y . The “**case**” predicate is a convenience that provides a form of pattern matching, but it is easily desugared into an ordinary logical expression. Similarly, pattern variables and ellipses get desugared into ordinary quantified variables.

Given the definition of $syntacticDef$, a conservative version of the $mayDef$ label from example 1 can be defined as:

$$\begin{aligned} mayDef(Y) \triangleq & \text{ case } currStmt \text{ of} \\ & *X := Z \quad \Rightarrow \quad true \\ & X := P(Z) \quad \Rightarrow \quad true \\ & \text{ else} \quad \Rightarrow \quad syntacticDef(Y) \\ & \text{ endcase} \end{aligned}$$

¹The correctness of our approach does not depend on the correctness of the witness, since our approach independently verifies that the witness has the required properties.

In other words, a statement may define variable Y if the statement is either a pointer store (since our intermediate language allows taking the address of a local variable), a procedure call (since the procedure may be passed pointers from which the address of Y is reachable), or otherwise a syntactic definition of Y .

In addition to defining labels using predicates, users can also define labels using the results of an analysis. Section 2.4 shows how such labels are defined and how they can be used to make *mayDef* less conservative in the face of pointers.

2.2 Backward Transformation Patterns

A backward transformation pattern is similar to a forward one, except that the direction of the flow of analysis is reversed:

ψ_1 **preceded by** ψ_2 **since** $s \Rightarrow s'$ **with witness** \mathcal{P}

The backward transformation pattern above says that s may be transformed to s' if on all paths in the CFG from s to the end of the procedure, there exists a statement satisfying ψ_1 , *preceded* by zero or more statements satisfying ψ_2 , *preceded* by s . The witnessing region of a program execution trace consists of the states between the transformed statement and the statement satisfying ψ_1 ; \mathcal{P} is called a *backward witness*.

As with forward transformation patterns, the backward witness plays the role of an invariant in the witnessing region. However, in a backward transformation the witnessing region occurs *after*, rather than before, the point where the transformed statement has been executed. Therefore, in general a backward witness must be a predicate that relates two execution states η_{old} and η_{new} , representing corresponding execution states in the witnessing region of traces in the original and transformed programs. Our automatic proof strategy attempts to prove that the backward witness is established by the transformation and preserved by the innocuous states. Finally, we prove that after the enabling statement is executed, the witness implies that the original and transformed execution states become identical, implying that the transformation is semantics-preserving.

Example 2 *Dead assignment elimination may be implemented in Cobalt by the following backward transformation pattern:*

$(\text{stmt}(X := \dots) \vee \text{stmt}(\text{return} \dots)) \wedge \neg \text{mayUse}(X)$
preceded by
 $\neg \text{mayUse}(X)$
since
 $X := E \Rightarrow \text{skip}$
with witness
 $\eta_{old}/X = \eta_{new}/X$

We express statement removal by replacement with a **skip** statement.² The removal of $X := E$ is enabled by either a later assignment to X or a **return** statement, which signals the end of the procedure. Preceding statements are innocuous if they don't use the contents of X .

The backward witness $\eta_{old}/X = \eta_{new}/X$ says that η_{old} and η_{new} are equal “up to” X : corresponding states in the witnessing region of the original and transformed programs are identical except for the contents of variable X . This invariant is established by the removal of $X := E$ and preserved throughout the region because X is not used. The witness implies that a redefinition of X or a **return** statement causes the execution states of the two traces to become identical.

²An execution engine for optimizations would not actually insert such **skips**.

2.3 Profitability Heuristics

If an optimization’s transformation pattern is proven sound, then it is legal to transform all matching occurrences of that pattern. For some optimizations, including our two examples above, all legal transformations are also *profitable*. However, in more complex optimizations, such as code motion and optimizations that trade off time and space, many transformations may preserve program behavior while only a small subset of them improve the code. To address this distinction between legality and profitability, an optimization is written in two pieces. The transformation pattern defines only which transformations are legal. An optimization separately describes which of the legal transformations are also profitable and should be performed; we call this second piece of an optimization its *profitability heuristic*.

An optimization’s profitability heuristic is expressed via a *choose* function, which can be arbitrarily complex and written in a language of the user’s choice. Given the set Δ of the legal transformations determined by the transformation pattern, and given the procedure being optimized, *choose* returns the subset of the transformations in Δ that should actually be performed. A complete optimization in Cobalt therefore has the following form, where O_{pat} is a transformation pattern:

$$O_{pat} \textit{ filtered through choose}$$

This way of factoring optimizations into a transformation pattern and a profitability heuristic is critical to our ability to prove optimizations sound automatically, since only an optimization’s transformation pattern affects soundness. Transformation patterns tend to be simple even for complicated optimizations, with the bulk of an optimization’s complexity pertaining to profitability. Profitability heuristics can be written in any language, thereby removing any limitations on their expressiveness. Without profitability heuristics, the extra complexity added to guards to express profitability information would prevent automated correctness reasoning.

For the constant propagation and dead assignment elimination optimizations shown earlier, the *choose* function returns all instances: $choose_{all}(\Delta, p) = \Delta$. This profitability heuristic is the default if none is specified explicitly. Below we give an example of an optimization with a nontrivial *choose* function.

Example 3 Consider the implementation of partial redundancy elimination (PRE) [14, 10] in Cobalt. One way to perform PRE is to first insert copies of statements in well-chosen places in order to convert partial redundancies into full redundancies, and then to eliminate the full redundancies by running a standard common subexpression elimination (CSE) optimization expressible in Cobalt. For example, in the following code fragment, the computation $x := a + b$ at the end is partially redundant, since it is redundant only when the true leg of the branch is executed:

```
b := ...;
if (...) {
  a := ...;
  x := a + b;
} else {
  ... // don't define a, b, or x, and don't use x.
}
x := a + b;
```

This partial redundancy can be eliminated by making a copy of the assignment $x := a + b$ in the false leg of the branch. Now the assignment after the branch is fully redundant and can be removed by running CSE followed by self-assignment removal (removing assignments of the form $x := x$).

The criterion that determines when it is legal to duplicate a statement is relatively simple. Most of the complexity in PRE involves determining which of the many legal duplications are profitable, so that partial

redundancies will be converted to full redundancies at minimum cost. The first, “code duplication” pass of PRE can be expressed in Cobalt as the following backward optimization:

```

    stmt( $X := E$ )  $\wedge$   $\neg$ mayUse( $X$ )
preceded by
    unchanged( $E$ )  $\wedge$   $\neg$ mayDef( $X$ )  $\wedge$   $\neg$ mayUse( $X$ )
since
    skip  $\Rightarrow X := E$ 
with witness
     $\eta_{old}/X = \eta_{new}/X$ 
filtered through
    ...

```

Analogous to statement removal, we express statement insertion as replacement of a `skip` statement.³ The label `unchanged(E)` is defined (by the optimization writer, as described in section 2.1.3) to be true at a statement s if s does not redefine the contents of any variable mentioned in E . The transformation pattern for code duplication allows the insertion if, on all paths in the CFG from the `skip`, $X := E$ is preceded by statements that do not modify E and X and do not use X , which are preceded by the `skip`. In the code fragment above, the transformation pattern allows $x := a + b$ to be duplicated in the `else` branch, as well as other (unprofitable) duplications. This optimization’s choose function is responsible for selecting those legal code insertions that also are the latest ones that turn all partial redundancies into full redundancies and do not introduce any partially dead computations. This condition is rather complicated, but it can be implemented in a language of the user’s choice and can be ignored when verifying the soundness of PRE.

2.4 Pure Analyses

In addition to optimizations, Cobalt allows users to write pure analyses that do not perform transformations. These analyses can be used to compute or verify properties of interest about a procedure and to provide information to be consumed by later transformations. A pure analysis defines a new label, and the result of the analysis is a labeling of the given CFG. The new label can then be used by other analyses, optimizations, or label definitions.

A forward pure analysis is similar to a forward optimization, except that it does not contain a rewrite rule or a profitability heuristic. Instead, it has a *defines* clause that gives a name to the new label. A forward pure analysis has the form

ψ_1 *followed by* ψ_2 *defines label with witness* \mathcal{P}

The new label can be added to a statement s if on all paths to s , there exists an (enabling) statement satisfying ψ_1 , followed by zero or more (innocuous) statements satisfying ψ_2 , followed by s . The given forward witness should be established by the enabling statement and preserved by the innocuous statements. If so, the witness provides the new label’s meaning: if a statement s has label *label*, then the corresponding witness \mathcal{P} is true of the program state just before execution of s .

The following example shows how a pure analysis can be used to compute a simple form of pointer information:

Example 4 *We say that a variable is tainted at a program point if its address may have been taken prior to that program point. The following pure analysis defines the notTainted label:*

³An execution engine for optimizations would conceptually insert `skips` dynamically as needed to perform insertions.

```

    stmt(decl X)
followed by
    ¬stmt(... := &X)
defines
    notTainted(X)
with witness
    notPointedTo(X, η)

```

The analysis says that a variable is not tainted at a statement if on all paths leading to that statement, the variable was declared, and then its address was never taken. The witness $\text{notPointedTo}(X, \eta)$ is a first-order predicate defined by the user that holds when no memory location in η contains a pointer to X .

The notTainted label can be used to define a more precise version of the mayDef label from earlier examples, which incorporates the fact that pointer stores and procedure calls cannot affect variables that are not tainted:

```

mayDef(Y)  $\triangleq$ 
  case currStmt of
    *X := Z    $\Rightarrow$   ¬notTainted(Y)
    X := P(Z)  $\Rightarrow$   X = Y  $\vee$  ¬notTainted(Y)
    else      $\Rightarrow$   syntacticDef(Y)
  endcase

```

With this new definition, optimizations using mayDef become less conservative in the face of pointer stores and calls.

Cobalt currently has no notion of backward pure analyses. Although we anticipate no technical barrier to introducing such a notion, additional mechanisms would be required in order to define the semantics of a label introduced by a backward analysis. So far we have not encountered a need for backward analyses.

Cobalt also currently only allows the results of a forward analysis to be used in a forward optimization, or in another forward analysis. Allowing a forward analysis to be used in a backward optimization may result in *interference*, whereby a transformation triggered by the backward optimization invalidates the results of the forward analysis. This issue is discussed in more detail in section 4.1.

3 COBALT

This section provides a formal definition of Cobalt and of the intermediate language that Cobalt optimizations manipulate. The full formal details can be found in appendix A.

3.1 Intermediate Language

A program π in our (untyped) intermediate language is described by the following grammar:

<i>Progs</i>	π	$::=$	$pr \dots pr$
<i>Procs</i>	pr	$::=$	$p(x) \{s; \dots ; s\}$
<i>Stmts</i>	s	$::=$	$\mathbf{decl} \ x \mid \mathbf{skip} \mid lhs := e \mid x := \mathbf{new} \mid$ $x := p(b) \mid \mathbf{if} \ b \ \mathbf{goto} \ \iota \ \mathbf{else} \ \iota \mid$ $\mathbf{return} \ x$
<i>Exprs</i>	e	$::=$	$b \mid *x \mid \&x \mid op \ b \ \dots \ b$
<i>Locatables</i>	lhs	$::=$	$x \mid *x$
<i>Base Exprs</i>	b	$::=$	$x \mid c$
<i>Ops</i>	op	$::=$	various operators with arity ≥ 1
<i>Vars</i>	x	$::=$	$\mathbf{x} \mid \mathbf{y} \mid \mathbf{z} \mid \dots$
<i>Proc Names</i>	p	$::=$	$\mathbf{p} \mid \mathbf{q} \mid \mathbf{r} \mid \dots$
<i>Consts</i>	c	$::=$	constants
<i>Indices</i>	ι	$::=$	$0 \mid 1 \mid 2 \mid \dots$

A program π is a sequence of procedures, and each procedure is a sequence of statements. We assume a distinguished procedure named `main`. Statements include local variable declarations, assignments to local variables and through pointers, heap memory allocation, procedure calls and returns, and conditional branches (unconditional branches can be simulated with conditional branches). We assume that no procedure declares the same local variable more than once. We assume that each procedure ends with a `return` statement. Statements are indexed consecutively from 0, and $stmtAt(\pi, \iota)$ returns the statement with index ι in π . Expressions include constants, local variable references, pointer dereferences, taking the addresses of local variables, and n -ary operators over non-pointer values.

A *state* of execution of a program is a tuple $\eta = (\iota, \rho, \sigma, \xi, \mathcal{M})$. The index ι indicates which statement is about to be executed. The environment ρ is a map from variables in scope to their locations in memory, and the store σ describes the contents of memory by mapping locations to values (constants and locations). The dynamic call chain is represented by a stack ξ , and \mathcal{M} is the memory allocator, which returns fresh locations as needed.

The states of a program π transition according to the *state transition function* \rightarrow_π . We denote by $\eta \rightarrow_\pi \eta'$ the fact that η' is the program state that is “stepped to” when execution proceeds from state η . The definition of \rightarrow_π is standard and is given in appendix A. We also define an *intraprocedural state transition function* \hookrightarrow_π . This function acts like \rightarrow_π except when the statement to be executed is a procedure call. In that case, \hookrightarrow_π steps “over” the call, returning the program state that will eventually be reached when control returns to the calling procedure.

We model run-time errors through the absence of state transitions: if in some state η program execution would fail with a run-time error, there won’t be any η' such that $\eta \rightarrow_\pi \eta'$ is true. Likewise, if a procedure call does not return successfully, e.g., because of infinite recursion, there won’t be any η' such that $\eta \hookrightarrow_\pi \eta'$ is true.

3.2 Cobalt

In this section, we first specify the syntax of a rewrite rule’s original and transformed statements s and s' . Then we define the syntax used for expressing ψ_1 and ψ_2 . Finally, we provide the semantics of optimizations. The witness \mathcal{P} does not affect the (dynamic) semantics of optimizations.

3.2.1 Syntax of s and s'

Statements s and s' are defined in the syntax of the *extended intermediate language*, which augments the intermediate language with a form of free variables called *pattern variables*. Each production in the grammar of the original intermediate language is extended with a case for a pattern variable. A few examples are shown below:

$$\begin{array}{lll} \text{Exprs} & e & ::= \dots \mid E \\ \text{Vars} & x & ::= \dots \mid X \mid Y \mid Z \mid \dots \\ \text{Consts} & c & ::= \dots \mid C \end{array}$$

Statements in the extended intermediate language are *instantiated* by substituting for each pattern variable a program fragment of the appropriate kind from the intermediate-language program being optimized. For example, the statement $X := E$ in the extended intermediate language contains two pattern variables X and E , and this statement can be instantiated to form an intermediate-language statement assigning any expression occurring in the intermediate program to any variable occurring in the intermediate program.

3.2.2 Syntax and Semantics of ψ_1 and ψ_2

The syntax for ψ , and also for label definitions, is described by the following grammar:

$$\begin{aligned} \psi & ::= \text{true} \mid \text{false} \mid \neg\psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid \\ & \quad l(t_1, \dots, t_n) \mid t = t \mid \\ & \quad \text{case } t \text{ of } t \Rightarrow \psi \ \dots \ t \Rightarrow \psi \ \text{else } \Rightarrow \psi \ \text{endcase} \end{aligned}$$

In the above grammar, l ranges over label names and t ranges over *terms*, which are elements drawn from the extended intermediate language as well as the distinguished term *currStmt*. The grammar consists of propositional logic augmented with label predicates, term equality, and the case predicate.

The semantics of a formula ψ is defined with respect to a labeled CFG. Each node n in the CFG for procedure p is labeled with a finite set $L_p(\iota)$, where ι is n 's index. $L_p(\iota)$ includes labels $l(t_1, \dots, t_n)$ where the terms do not contain pattern variables. For example, a node could be labeled with $\text{stmt}(x := 3)$ and $\text{mayDef}(x)$.

The meaning of a formula ψ at a node depends on a substitution θ mapping the pattern variables in ψ to fragments of p . We extend substitutions to formulas and program fragments containing pattern variables in the usual way. We write $\iota \models_{\theta}^p \psi$ to indicate that the node with index ι satisfies ψ in the labeled CFG of p under substitution θ . The definition of $\iota \models_{\theta}^p \psi$ is straightforward, with the base case being $\iota \models_{\theta}^p l(t_1, \dots, t_n) \iff \theta(l(t_1, \dots, t_n)) \in L_p(\iota)$. The complete definition of \models_{θ}^p is in appendix A.

3.2.3 Semantics of Optimizations

We define the semantics of optimizations and analyses in several pieces. First, the meaning of a *forward guard* ψ_1 **followed by** ψ_2 is a function that takes a procedure and returns a set of matching indices with their corresponding substitutions:

Definition 1 *The meaning of a forward guard O_{guard} of the form ψ_1 **followed by** ψ_2 is as follows:*

$$\begin{aligned} \llbracket O_{\text{guard}} \rrbracket(p) &= \{(\iota, \theta) \mid \\ & \quad \text{for all paths } \iota_1, \dots, \iota_j, \iota \text{ in } p\text{'s CFG} \\ & \quad \text{such that } \iota_1 \text{ is the index of } p\text{'s entry node} \\ & \quad \exists k. (1 \leq k \leq j \wedge \iota_k \models_{\theta}^p \psi_1 \wedge \forall i. (k < i \leq j \Rightarrow \iota_i \models_{\theta}^p \psi_2))\} \end{aligned}$$

The above definition formalizes the description of forward guards from Section 2. The meaning of a backward guard ψ_1 **preceded by** ψ_2 is identical, except that the guard is evaluated on CFG paths $\iota, \iota_j, \dots, \iota_1$ that start, rather than end, at ι , where ι_1 is the index of the procedure’s exit node. Guards can be seen as a restricted form of temporal logic formula, expressible in variants of both Linear Temporal Logic (LTL) [7] and Computation Tree Logic (CTL) [5].

Next we define the semantics of transformation patterns. A forward (backward) transformation pattern $O_{pat} = O_{guard}$ **until (since)** $s \Rightarrow s'$ **with witness** \mathcal{P} simply filters the set of nodes matching its guard to include only those nodes of the form s :

$$\llbracket O_{pat} \rrbracket(p) = \{(\iota, \theta) \mid (\iota, \theta) \in \llbracket O_{guard} \rrbracket(p) \text{ and } \iota \models_{\theta}^p \text{stmt}(s)\}$$

The meaning of an optimization is a function that takes a procedure p and returns the procedure produced by applying to p all transformations selected by the *choose* function.

Definition 2 *Given an optimization O of the form O_{pat} **filtered through** *choose*, where O_{pat} has rewrite rule $s \Rightarrow s'$, the meaning of O is as follows:*

$$\llbracket O \rrbracket(p) = \text{let } \Delta := \llbracket O_{pat} \rrbracket(p) \text{ in } \\ \text{app}(s', p, \text{choose}(\Delta, p) \cap \Delta)$$

where $\text{app}(s', p, \Delta')$ returns the procedure identical to p but with the node with index ι transformed to $\theta(s')$, for each (ι, θ) in Δ' .⁴

Finally, the meaning of a pure analysis O_{guard} **defines label with witness** \mathcal{P} applied to a procedure p is a new version of p ’s CFG where for each pair (ι, θ) in $\llbracket O_{guard} \rrbracket(p)$, the node with index ι is additionally labeled with $\theta(\text{label})$.

4 Proving Soundness Automatically

In this section we describe our technique for automatically proving soundness of Cobalt optimizations. The full details, including the proofs of the theorems, are in appendix A.

We say that an intermediate-language program π' is a *semantically equivalent transformation* of π if, whenever $\text{main}(v_1)$ returns v_2 in π , for some values v_1 and v_2 , then it also does in π' . Let $\pi[p \mapsto p']$ denote the program identical to π but with procedure p replaced by p' . An optimization O is *sound* if for all intermediate-language programs π and procedures p in π , $\pi[p \mapsto \llbracket O \rrbracket(p)]$ is a semantically equivalent transformation of π .

To prove a Cobalt optimization sound, we prove the soundness of its associated transformation pattern. We say that a transformation pattern O_{pat} with rewrite rule $s \Rightarrow s'$ is sound if, for all intermediate-language programs π and procedures p in π , for all subsets $\Delta \subseteq \llbracket O_{pat} \rrbracket(p)$, $\pi[p \mapsto \text{app}(s', p, \Delta)]$ is a semantically equivalent transformation of π . If a transformation pattern is sound, then *any* optimization O with that transformation pattern is sound, since the optimization will select some subset of the transformation pattern’s suggested transformations, and each subset is known to result in a semantically equivalent transformation of π . Therefore, we need not reason at all about an optimization’s profitability heuristic in order to prove that the optimization is sound.

First we discuss a property of Cobalt that simplifies the obligations necessary for proving a transformation pattern sound. Then we describe these obligations for forward and backward optimizations, respectively.

⁴If there are multiple pairs in Δ' with the same index ι , then one of them is chosen nondeterministically.

4.1 Noninterference

As described above, for a transformation pattern to be sound, it must be possible to apply any subset of the suggested transformations without changing a procedure’s semantics. Therefore, to prove a transformation pattern sound, we must argue that its suggested transformations cannot *interfere* with one another. Interference occurs when multiple transformations that are semantics-preserving in isolation cause a procedure’s semantics to change when performed together.

In general it is possible for an optimization to interfere with itself. For example, consider an optimization that performs both dead assignment elimination and redundant assignment elimination. On the following program fragment

```

...
S1:  x := 5;
S2:  x := 5;
...

```

our hypothetical optimization will suggest both S1 and S2 for removal: S1 is dead and S2 is redundant. Performing either removal is correct, but performing both removals changes the program’s semantics.

Fortunately, it is possible to show that a Cobalt transformation pattern cannot interfere with itself: if each transformation from a set of suggested transformations is correct in isolation, then performing any subset of the transformations is correct. The optimization above cannot be directly written in Cobalt. Instead, it must be written as two separate optimizations, one forward and one backward.⁵

Because of Cobalt’s noninterference property, the optimization-specific obligations to be discharged as part of our proof strategy need only pertain to a single transformation. The theorems described below validate the sufficiency of these obligations for proving Cobalt optimizations sound.

4.2 Forward Transformation Patterns

Consider a forward transformation pattern of the following form:

ψ_1 *followed by* ψ_2 *until* $s \Rightarrow s'$ *with witness* \mathcal{P}

As discussed in section 2, our proof strategy entails showing that the forward witness \mathcal{P} holds throughout the witnessing region and that the witness implies s and s' have the same semantics. This can naturally be shown by induction over the states in the witnessing region of an execution trace leading to a transformed statement. In general, it is difficult for an automatic theorem prover to determine when proof by induction is necessary and to perform such a proof with a strong enough inductive hypothesis. Therefore we instead require an automatic theorem prover to discharge only noninductive obligations, which pertain to individual execution states rather than entire execution traces. We have proven that if these obligations hold for any particular optimization, then that optimization is sound.

We use *index* as an accessor on states: $index((\iota, \rho, \sigma, \xi, \mathcal{M})) = \iota$. The optimization-specific obligations, to be discharged by an automatic theorem prover, are as follows, where $\theta(\mathcal{P})$ is the predicate formed by applying θ to each pattern variable in the definition of \mathcal{P} :

F1. If $\eta \hookrightarrow_{\pi} \eta'$ and $index(\eta) \models_{\theta}^{\mathcal{P}} \psi_1$, then $\theta(\mathcal{P})(\eta')$.

F2. If $\theta(\mathcal{P})(\eta)$ and $\eta \hookrightarrow_{\pi} \eta'$ and $index(\eta) \models_{\theta}^{\mathcal{P}} \psi_2$, then $\theta(\mathcal{P})(\eta')$.

⁵The example illustrates a potential unsoundness from combining forward and backward transformation patterns. This is the reason that we currently disallow employing a forward pure analysis in a backward transformation. We can, however, prove that a forward transformation pattern cannot interfere with any forward pure analysis.

F3. If $\theta(\mathcal{P})(\eta)$ and $\eta \hookrightarrow_{\pi} \eta'$ and $\iota = \text{index}(\eta)$ and $\text{stmtAt}(\pi, \iota) = \theta(s)$ and $\text{stmtAt}(\pi', \iota) = \theta(s')$, then $\eta \hookrightarrow_{\pi'} \eta'$.

Condition F1 ensures that the witness holds at any state following the execution of an enabling statement (one satisfying ψ_1). Condition F2 ensures that the witness is preserved by any innocuous statement (one satisfying ψ_2). Finally, condition F3 ensures that s and s' have the same semantics when executed from a state satisfying the witness.

As an example, consider condition F1 for the constant propagation optimization from example 1. The condition looks as follows: If $\eta \hookrightarrow_{\pi} \eta'$ and $\text{index}(\eta) \models_{\theta}^p \text{stmt}(Y := C)$, then $\theta(\eta'(Y)) = C$. The condition is easily proven automatically from the semantics of assignments and the *stmt* label.

The following theorem validates the optimization-specific proof obligations.

Theorem 1 *If O is a forward optimization satisfying conditions F1, F2, and F3, then O is sound.*

The proof of this theorem uses conditions F1 and F2 as part of the base case and the inductive case, respectively, in an inductive argument that the witness holds throughout a witnessing region. Condition F3 is then used to show that s and s' have the same semantics in this context.

A pure analysis ψ_1 **followed by** ψ_2 **defines label with witness** \mathcal{P} is proven sound similarly. We require conditions F1 and F2 to be satisfied; F3 has no analogue. These conditions allow us to show that *label* indeed has the semantics of the witness \mathcal{P} .

4.3 Backward Transformation Patterns

Consider a backward transformation pattern of the following form:

ψ_1 **preceded by** ψ_2 **since** $s \Rightarrow s'$ **with witness** \mathcal{P}

The optimization-specific obligations are similar to those for a forward transformation pattern, except that the ordering of events in the witnessing region is reversed:

B1. If $\eta \hookrightarrow_{\pi} \eta_{old}$ and $\eta \hookrightarrow_{\pi'} \eta_{new}$ and $\iota = \text{index}(\eta)$ and $\text{stmtAt}(\pi, \iota) = \theta(s)$ and $\text{stmtAt}(\pi', \iota) = \theta(s')$, then $\theta(\mathcal{P})(\eta_{old}, \eta_{new})$.

B2. If $\theta(\mathcal{P})(\eta_{old}, \eta_{new})$ and $\eta_{old} \hookrightarrow_{\pi} \eta'_{old}$ and $\iota_{old} = \text{index}(\eta_{old})$ and $\iota_{new} = \text{index}(\eta_{new})$ and $\iota_{old} \models_{\theta}^{\pi} \psi_2$ and $\text{stmtAt}(\pi, \iota_{old}) = \text{stmtAt}(\pi', \iota_{new})$, then there exists some η'_{new} such that $\eta_{new} \hookrightarrow_{\pi'} \eta'_{new}$ and $\theta(\mathcal{P})(\eta'_{old}, \eta'_{new})$.

B3. If $\theta(\mathcal{P})(\eta_{old}, \eta_{new})$ and $\eta_{old} \hookrightarrow_{\pi} \eta$ and $\iota_{old} = \text{index}(\eta_{old})$ and $\iota_{new} = \text{index}(\eta_{new})$ and $\iota_{old} \models_{\theta}^{\pi} \psi_1$ and $\text{stmtAt}(\pi, \iota_{old}) = \text{stmtAt}(\pi', \iota_{new})$, then $\eta_{new} \hookrightarrow_{\pi'} \eta$.

Condition B1 ensures that the backward witness holds between the original and transformed programs, after s and s' are respectively executed.⁶ Condition B2 ensures that the backward witness is preserved through the innocuous statements. Condition B3 ensures that the two traces become identical again after executing the enabling statement (and exiting the witnessing region).

Analogous to the forward case, the following theorem validates the optimization-specific proof obligations for backward optimizations.

Theorem 2 *If O is a backward optimization satisfying conditions B1, B2, and B3, then O is sound.*

⁶This condition assumes that s' does not get “stuck” by causing a run-time error. That assumption must actually be *proven*, but for simplicity we elide this issue here. It is addressed by requiring a few additional obligations to be discharged that imply that s' cannot get stuck if the original program does not get stuck. Details are in appendix A.

5 Implementing COBALT

We have implemented a tool that automatically checks the correctness of Cobalt optimizations as well as an execution engine for running them. Section 5.1 describes our correctness checker, and section 5.2 describes our execution engine.

5.1 Correctness Checker

We have implemented our strategy for automatically proving Cobalt optimizations sound with the Simplify automatic theorem prover. For each optimization, we ask Simplify to prove the three associated optimization-specific obligations given a set of background axioms. There are two kinds of background axioms: optimization-independent ones and optimization-dependent ones. The optimization-independent axioms simply encode the semantics of our intermediate language and they need not be modified in order to prove new optimizations sound. The optimization-dependent axioms encode the semantics of user-defined labels and are generated automatically from the Cobalt label definitions. Our correctness checker translates label definitions into Simplify axioms by expanding case expressions into ordinary boolean expressions and performing a few simple transformations to produce axioms in a form accepted by Simplify.

To encode the Cobalt intermediate language in Simplify, we introduce function symbols that represent term constructors for each kind of expression and statement. For example, the term $assgn(var(x), deref(var(y)))$ represents the statement $x := *y$. Next we formalize the representation of program states. Simplify has built-in axioms about a *map* data structure, with associated functions *select* and *update* to access elements and (functionally) update the map. This is useful for representing many components of a state. For example, an environment is a map from variables to locations, and a store is a map from locations to values.

Given our representation for states, we define axioms for a function symbol $evalExpr$, which evaluates an expression in a given state. The $evalExpr$ function represents the function $\eta(\cdot)$ used in section 2. We also define axioms for a function $evalLEExpr$ which computes the location of a *lhs* expression given a program state. Then we provide axioms for the $stepIndex$, $stepEnv$, $stepStore$, $stepStack$, and $stepMem$ functions, which together define the state transition function \rightarrow_π from section 3.1. These functions take a state and a program and return the new value of the state component being “stepped.” As an example, the axioms for stepping an index and a store through an assignment $lhs := e$ are as follows:

$$\begin{aligned} \forall \eta, \pi, lhs, e. \\ stmtAt(\pi, index(\eta)) = assgn(lhs, e) \Rightarrow \\ stepIndex(\eta, \pi) = index(\eta) + 1 \end{aligned}$$

$$\begin{aligned} \forall \eta, \pi, lhs, e. \\ stmtAt(\pi, index(\eta)) = assgn(lhs, e) \Rightarrow \\ stepStore(\eta, \pi) = update(store(\eta), evalLEExpr(\eta, lhs), \\ evalExpr(\eta, e)) \end{aligned}$$

The first axiom says that the new index is the current index incremented by one. The second axiom says that the new store is the same as the old one, but with the location of *lhs* updated to the value of *e*.

Finally, the \hookrightarrow_π function is defined in terms of the \rightarrow_π function. In the context of intraprocedural analysis, we do not have access to the bodies of called procedures. Therefore, we conservatively model the semantics of stepping over a procedure call by a set of axioms that hold for *any* call. The primary axiom says that the store after a call preserves the values of local variables in the caller whose locations are not pointed to before the call. This axiom encodes the fact that locals not reachable from the store cannot be modified by a call.

We have implemented and automatically proven sound a dozen Cobalt optimizations and analyses (which are given in the appendix B). On a modern workstation, the time taken by Simplify to discharge the optimization-specific obligations for these optimizations ranges from 3 to 104 seconds, with an average of 28 seconds.

5.2 Execution Engine

To run Cobalt optimizations without first rewriting them in some other language, we have implemented an execution engine for Cobalt as an analysis in the Whirlwind compiler, a successor to Vortex [4].

This analysis stores at each program point a set of substitutions, with each substitution representing a potential witnessing region. Consider a forward optimization:

*ψ_1 followed by ψ_2 until $s \Rightarrow s'$
with witness \mathcal{P} filtered through choose*

The flow function for our analysis works as follows. First, if the statement being processed satisfies ψ_1 , then the flow function adds to the outgoing dataflow fact the substitution that caused ψ_1 to be true. Also, for each substitution θ in the incoming dataflow fact, the flow function checks if $\theta(\psi_2)$ is true at the current statement. If it is, then θ is propagated to the outgoing dataflow fact, and otherwise it is dropped. Finally, merge nodes simply take the intersection of the incoming dataflow facts. After the analysis has reached a fixed point, if a statement has a substitution θ in its incoming dataflow fact that makes $\theta(stmt(s))$ true and the *choose* function selects this statement, then the statement is transformed to $\theta(s')$.

For example, in constant propagation we have $\psi_1 = stmt(Y := C)$ and $\psi_2 = \neg mayDef(Y)$. Below we show the dataflow facts propagated after a few example statements:

S1 : $a := 2$; $[Y \mapsto a, C \mapsto 2]$
S2 : $b := 3$; $[Y \mapsto a, C \mapsto 2], [Y \mapsto b, C \mapsto 3]$
S3 : $c := a$;

S1 satisfies ψ_1 , so its outgoing dataflow fact contains the substitution $[Y \mapsto a, C \mapsto 2]$. S2 satisfies ψ_2 under this substitution, so the substitution is propagated; S2 also satisfies ψ_1 so $[Y \mapsto b, C \mapsto 3]$ is added to the outgoing dataflow fact. In fact, the dataflow information after S2 is very similar to the regular constant propagation dataflow fact $\{a \mapsto 2, b \mapsto 3\}$. At fixed point, the statement $c := a$ can be transformed to $c := 2$ because the incoming dataflow fact contains the map $[Y \mapsto a, C \mapsto 2]$. Note that this implementation evaluates all “instances” of the constant propagation transformation pattern simultaneously.

Our analysis is implemented using our earlier framework for composable optimizations in Whirlwind [12]. This framework allows optimizations to be defined modularly and then automatically combines all-forward or all-backward optimizations in order to gain mutually beneficial interactions. Analyses and optimizations written in Cobalt are therefore also composable in this way. Furthermore, Whirlwind’s framework automatically composes an optimization with itself, allowing a recursively defined optimization to be solved in an optimistic, iterative manner; this property is likewise conferred on Cobalt optimizations. For example, a recursive version of dead-assignment elimination allows $X := E$ to be removed even if X is used before being redefined, as long as it is only used by other dead assignments (possibly including itself).⁷

6 Discussion

In this section, we evaluate our system along three dimensions: expressiveness of Cobalt, debugging value, and reduced trusted computing base.

⁷Although Cobalt optimizations can be composed, we have not yet proved that the flow function of our Cobalt engine satisfies the properties required in [12] for the composition to be sound. We plan to investigate this in future work.

Expressiveness. One of the key choices in our approach is to restrict the language in which optimizations can be written, in order to gain automatic reasoning about soundness. However, Cobalt’s restrictions are not as onerous as they may first appear. First, much of the complexity of an optimization can be factored into the profitability heuristic, which is unrestricted. Second, the pattern of a witnessing region — beginning with a single enabling statement and passing through zero or more innocuous statements before reaching the statement to be transformed — is common to many forward intraprocedural dataflow analyses, and similarly for backward intraprocedural dataflow analyses. Third, optimizations that traditionally are expressed as having effects at multiple points in the program, such as various sorts of code motion, can in fact be decomposed into several simpler transformations, each of which fits Cobalt’s transformation pattern syntax.

The PRE example in section 2.3 illustrates all three of these points. PRE is a complex code-motion optimization [14, 10], and yet it can be expressed in Cobalt using simple forward and backward passes with appropriate profitability heuristics. Our way of factoring complicated optimizations into smaller pieces, and separating the part that affects soundness from the part that doesn’t, allows users to write optimizations that are intricate and expressive yet still amenable to automated correctness reasoning.

Even so, the current version of Cobalt does have limitations. For example, it cannot express interprocedural optimizations or one-to-many transformations. As mentioned in section 7, our future work will address these limitations. Also, optimizations and analyses that build complex data structures to represent their dataflow facts may be difficult to express. Finally, it is possible for limitations in either our proof strategy or in the automatic theorem prover to cause a sound optimization expressible in Cobalt to be rejected. In all these cases, optimizations can be written outside of our framework, perhaps verified using translation validation. Optimizations written in Cobalt and proven correct can peacefully co-exist with optimizations written “the normal way.”

Debugging benefit. Writing correct optimizations is difficult because there are many corner cases to consider, and it is easy to miss one. Our system in fact found several subtle problems in previous versions of our optimizations. For example, we have implemented a form of common subexpression elimination (CSE) that eliminates not only redundant arithmetic expressions, but also redundant loads. In particular, this optimization tries to eliminate a computation of $*X$ if the result is already available from a previous load. Our initial version of the optimization precluded pointer stores from the witnessing region, to ensure that the value of $*X$ was not modified. However, a failed soundness proof made us realize that even a direct assignment $Y := \dots$ can change the value of $*X$, because X could point to Y . Once we incorporated pointer information to make sure that direct assignments in the witnessing region were not changing the value of $*X$, our implementation was able to automatically prove the optimization sound. Without the static checks to find the bug, it could have gone undetected for a long time, because that particular corner case may not occur in many programs.

Reduced trusted computing base. The trusted computing base (TCB) ordinarily includes the entire compiler. In our system we have moved the compiler’s optimization phase, one of the most intricate and error-prone portions, outside of the TCB. Instead, we have shifted the trust in this phase to three components: the correctness checker, including the automatic theorem prover, the manual proofs done as part of our framework, and the engine that executes optimizations. Because all of these components are optimization-independent, new optimizations can be incorporated into the compiler without enlarging the TCB. Furthermore, as discussed in section 5, the execution engine is implemented as a single dataflow analysis common to all user-defined optimizations. This means that the trustworthiness of the execution engine is akin to the trustworthiness of a single optimization pass in a traditional compiler.

Trust can be further enhanced in several ways. First, we could use an automatic theorem prover that generates proofs, such as the prover in the Touchstone compiler [21]. This would allow trust to be shifted from the theorem prover to a simpler proof checker. The manual proofs of our framework are made public for peer review in appendix A to increase confidence. We could also use an interactive theorem prover such as PVS [24] to validate these proofs.

7 Future Work

There are many directions for future work. We plan to extend Cobalt to handle interprocedural optimizations. One approach would extend the scope of analysis from a single procedure to the whole program’s control-flow supergraph. A technical challenge for this approach is the need to express the witness \mathcal{P} in a way that is robust across procedure calls. For example, the predicate $\eta(Y) = C$ does not make sense once a call is stepped into, because Y has gone out of scope. We intend to extend the syntax for the witness to be more precise about which location is being talked about. A different approach to interprocedural analysis would use pure analyses to define summaries of procedures, which could be used in intraprocedural optimizations of callers.

Currently Cobalt only supports transformations that replace a single statement with a single statement. It should be relatively straightforward to generalize the framework to handle one-to-many statement transformations, allowing optimizations like inlining to be expressed. Supporting many-to-many statement transformations, including various kinds of loop restructuring optimizations, would also be interesting.

We plan to try inferring the witnesses, which are currently provided by the user. It may be possible to use some simple heuristics to guess a witness from the given transformation pattern. As a simple example, in the constant propagation example of section 2, the appropriate witness, that Y has the value C , is simply the strongest postcondition of the enabling statement $Y := C$. Many of the other forward optimizations that we have written also have this property.

Our current notion of a semantically equivalent transformation reasons only about computations in the original program that terminate without an error. It would be straightforward to reason about computations that end in a run-time error by extending the \rightarrow_π function to step to an explicit *error* state in these situations. We would also like to extend the notion of semantic equivalence to allow reasoning about nonterminating computations.

We plan to explore more efficient implementation techniques for the Cobalt execution engine, such as generating specialized code to run each optimization [31]. Another direction for improving efficiency would be to allow analyses to be defined over a sparse representation such as a dataflow graph.

Finally, an important consideration that we have not addressed is the interface between the optimization writer and our automatic correctness checker. It will be critical to provide useful error messages when an optimization cannot be proven sound. When Simplify cannot prove a given proposition, it returns a *counterexample context*, which is a state of the world that violates the proposition. An interesting approach would be to use this counterexample context to synthesize a small intermediate-language program that illustrates a potential unsoundness of the given optimization.

8 Related Work

Temporal logic has previously been used to express dataflow analyses and reason about them by hand [31, 32, 28, 29, 11]. Our language is inspired by recent work in this direction by Lacey et al. [11]. Lacey describes a language for writing optimizations as guarded rewrite rules evaluated over a labeled CFG, and our transformation patterns are modeled on this language. Lacey’s intermediate language lacks several constructs found in realistic languages, including pointers, dynamic memory allocation, and procedures. Lacey describes a general strategy, based on relating execution traces of the original and transformed programs, for manually proving the soundness of optimizations in his language. Three example optimizations are shown and proven sound by hand using this strategy. Unfortunately, the generality of this strategy makes it difficult to automate.

Lacey’s guards may be arbitrary CTL formulas, while our guard language can be viewed as a strict subset of CTL that codifies a particularly common idiom. However, we are still able to express more precise versions

of Lacey’s three example optimizations (as well as many others) and to prove them sound automatically. Further, Lacey’s optimization language has no notion of labels defined by pure analyses nor of profitability heuristics. Therefore, expressing optimizations that employ pointer information (assuming Lacey’s language were augmented with pointers) or optimizations like PRE would instead require writing more complicated guards, and some optimizations we support may not be expressible by Lacey.

As mentioned in the introduction, much other work has been done on manually proving optimizations correct [13, 15, 1, 2, 8, 23, 3]. Transformations have also been proven correct mechanically, but not automatically: the transformation is proven sound using an interactive theorem prover, which requires user involvement. For example, Young [34] has proven a code generator correct using the Boyer-Moore theorem prover enhanced with an interactive interface [9].

Instead of proving that the compiler is always correct, translation validation [25, 19] and credible compilation [27, 26] both attack the problem of checking the correctness of a given compilation run. Therefore, a bug in an optimization only appears when the compiler is run on a program that triggers the bug. Our work allows optimizations to be proven correct before the compiler is even run once. However, to do so we require optimizations to be written in a special-purpose language. Our approach also requires the Cobalt execution engine to be part of the TCB, while translation validation and credible compilation do not require trust in any part of the compiler.

Proof-carrying code [18], certified compilation [20], typed intermediate languages [33], and typed assembly languages [16, 17] have all been used to prove properties of programs generated by a compiler. However, the kinds of properties that these approaches have typically guaranteed are type safety and memory safety. In our work, we prove the stronger property of semantic equivalence between the original and resulting programs.

9 Conclusion

We have presented an approach for automatically proving the correctness of compiler optimizations. Our technique provides the optimization writer with a domain-specific language, called Cobalt, for writing optimizations. Cobalt is both reasonably expressive and amenable to automated correctness reasoning. Using our technique we have proven correct implementations of several optimizations over a realistic intermediate language. We believe our approach is a promising step toward the goal of reliable and user-extensible compilers.

Acknowledgments

This research is supported in part by NSF grants CCR-0073379 and ACI-0203908, a Microsoft Graduate Fellowship, an IBM Faculty Development Award, and by gifts from Sun Microsystems. We would also like to thank Keunwoo Lee, Andrew Petersen, Mark Seigle and the anonymous reviewers for their useful suggestions on how to improve the paper.

References

- [1] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles CA, January 1977.
- [2] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages*, pages 269–282, San Antonio TX, January 1979.

- [3] Patrick Cousot and Radhia Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Conference Record of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland OR, January 2002.
- [4] Jeffrey Dean, Greg DeFouw, Dave Grove, Vassily Litvinov, and Craig Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Proceedings of the 1996 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 83–100, San Jose CA, October 1996.
- [5] E.M. Clarke and E.A. Emerson. Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In *Logics of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*, Yorktown Heights, New York, May 1981. Springer-Verlag.
- [6] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*, June 2002.
- [7] Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, Las Vegas, Nevada, 1980.
- [8] J. Guttman, J. Ramsdell, and M. Wand. VLISP: a verified implementation of Scheme. *Lisp and Symbolic Computation*, 8(1-2):33–110, 1995.
- [9] M. Kauffmann and R.S. Boyer. The Boyer-Moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications*, 29(2):27–62, 1995.
- [10] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.
- [11] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Conference Record of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland OR, January 2002.
- [12] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. In *Conference Record of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland OR, January 2002.
- [13] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In T. J. Schwartz, editor, *Proceedings of Symposia in Applied Mathematics*, January 1967.
- [14] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.
- [15] F. Lockwood Morris. Advice on structuring compilers and proving them correct. In *Conference Record of the 1st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Boston MA, January 1973.
- [16] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta GA, May 1999.
- [17] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.
- [18] George C. Necula. Proof-carrying code. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, January 1997.
- [19] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 83–95, Vancouver, Canada, June 2000.
- [20] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

- [21] George C. Necula and Peter Lee. Proof generation in the Touchstone theorem prover. In *Proceedings of the International Conference on Automated Deduction*, pages 25–44, Pittsburgh, Pennsylvania, June 2000. Springer-Verlag LNAI 1831.
- [22] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.
- [23] D. P. Oliva, J. Ramsdell, and M. Wand. The VLISP verified PreScheme compiler. *Lisp and Symbolic Computation*, 8(1-2):111–182, 1995.
- [24] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.
- [25] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS '98*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166, 1998.
- [26] Martin Rinard. Credible compilation. Technical Report MIT-LCS-TR-776, Massachusetts Institute of Technology, March 1999.
- [27] Martin Rinard and Darko Marinov. Credible compilation. In *Proceedings of the FLoC Workshop Run-Time Result Verification*, July 1999.
- [28] David A. Schmidt. Dataflow analysis is model checking of abstract interpretations. In *Conference Record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego CA, January 1998.
- [29] David A. Schmidt and Bernhard Steffen. Data flow analysis as model checking of abstract interpretations. In Giorgio Levi, editor, *Proceedings of the 5th International Symposium on Static Analysis (SAS)*, volume 1503 of *Lecture Notes in Computer Science (LNCS)*, pages 351–380. Springer-Verlag, September 1998.
- [30] Simplify automatic theorem prover home page, <http://research.compaq.com/SRC/esc/Simplify.html>.
- [31] Bernhard Steffen. Data flow analysis as model checking. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Science (TACS), Sendai (Japan)*, volume 526 of *Lecture Notes in Computer Science (LNCS)*, pages 346–364. Springer-Verlag, September 1991.
- [32] Bernhard Steffen. Generating dataflow analysis algorithms for model specifications. *Science of Computer Programming*, 21(2):115–139, 1993.
- [33] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia PA, May 1996.
- [34] William D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5(4):493–518, December 1989.

A Formalization

A.1 Semantics of the Intermediate Language

A.1.1 Preliminaries

The set of indices of program π is denoted $Indices_\pi$. The set of indices of procedure p is denoted $Indices_p$. The formal argument name of procedure p in program π is denoted $formal_p$. The index of the first statement in procedure p of program π is denoted $start_p$. We assume WLOG that no **if** statements in a procedure p refer to indices not in $Indices_p$, nor to the index $start_p$.⁸

The arity of an operator op is denoted $arity(op)$. We assume a fixed interpretation function for each n -ary operator symbol op : $\llbracket op \rrbracket : Consts^n \rightarrow Consts$.

We assume an infinite set $Locations$ of memory locations, with metavariable l ranging over the set. We assume that the set $Consts$ is disjoint from $Locations$ and contains the distinguished elements $true$ and $uninit$. Then the set of values is defined as $Values = (Locations \cup Consts)$

An *environment* is a partial function $\rho : Vars \rightarrow Locations$; we denote by *Environments* the set of all environments. A *store* is a partial function $\sigma : Locations \rightarrow Values$; we denote by *Stores* the set of all stores. The domain of an environment ρ is denoted $dom(\rho)$, and similarly for the domain of a store. The notation $\rho[x \mapsto l]$ denotes the environment identical to ρ but with variable x mapping to location l ; if $x \in dom(\rho)$, the old mapping for x is shadowed by the new one. The notation $\sigma[l \mapsto v]$ is defined similarly. The notation $\sigma/\{l_1, \dots, l_i\}$ denotes the store identical to σ except that all pairs $(l, v) \in \sigma$ such that $l \in \{l_1, \dots, l_i\}$ are removed.

The current dynamic call chain is represented by a stack. A *stack frame* is a triple $f = (\iota, l, \rho) : Indices \times Locations \times Environments$. Here ι is the index of the first statement following the call currently being executed, l is the location in which to put the return value from the call, and ρ is the current lexical environment at the point of the call. We denote by *Frames* the set of all stack frames. A *stack* $\xi = \langle f_1 \dots f_n \rangle : Frames^*$ is a sequence of stack frames. The set of all stacks is denoted *Stacks*. Stacks support two operations defined as follows:

$$\begin{aligned} push &: (Frames \times Stacks) \rightarrow Stacks \\ push(f, \langle f_1 \dots f_n \rangle) &= \langle f \ f_1 \dots f_n \rangle \\ \\ pop &: Stacks \rightarrow (Frames \times Stacks) \\ pop(\langle f_1 \ f_2 \dots f_n \rangle) &= (f_1, \langle f_2 \dots f_n \rangle), \text{ where } n > 0 \end{aligned}$$

Finally, a memory allocator \mathcal{M} is an infinite stream $\langle l_1, l_2, \dots \rangle$ of locations. We denote the set of all memory allocators as *MemAllocs*.

A *state* of execution of a program π is a quintuple $\eta = (\iota, \rho, \sigma, \xi, \mathcal{M})$ where $\iota \in Indices$, $\rho \in Environments$, $\sigma \in Stores$, $\xi \in Stacks$, and $\mathcal{M} \in MemAllocs$. We denote the set of program states by *States*. We refer to the corresponding index of a state η as $index(\eta)$, and we similarly define accessors *env*, *store*, *stack*, and *mem*.

⁸This last restriction ensures that the entry node of p 's CFG will have no incoming edges.

A.1.2 State Transition Functions

The evaluation of an expression e in a program state η , where $env(\eta) = \rho$ and $store(\sigma) = \rho$, is given by the function $\eta(e) : (States \times Exprs) \rightarrow Values$ defined by:

$$\begin{aligned}
\eta(x) &= \sigma(\rho(x)) \\
&\text{where } x \in dom(\rho), \rho(x) \in dom(\sigma) \\
\eta(c) &= c \\
\eta(*x) &= \sigma(\sigma(\rho(x))) \\
&\text{where } x \in dom(\rho), \rho(x) \in dom(\sigma), \sigma(\rho(x)) \in dom(\sigma) \\
\eta(\&x) &= \rho(x) \\
&\text{where } x \in dom(\rho) \\
\eta(opb_1 \dots b_n) &= \llbracket op \rrbracket(\eta(b_1), \dots, \eta(b_n)) \\
&\text{where } arity(op) = n \text{ and } \forall 1 \leq j \leq n. (\eta(b_j) \in Consts)
\end{aligned}$$

Similarly, the evaluation of a locatable expression lhs in a state η , where $env(\eta) = \rho$ and $store(\sigma) = \rho$, is given by the function $\eta^l(lhs) : (States \times Locatables) \rightarrow Locations$ defined by:

$$\begin{aligned}
\eta^l(x) &= \rho(x) \\
&\text{where } x \in dom(\rho) \\
\eta^l(*x) &= \sigma(\rho(x)) \\
&\text{where } x \in dom(\rho), \rho(x) \in dom(\sigma), \sigma(\rho(x)) \in Locations
\end{aligned}$$

Definition 3 Given a program π , the state transition function $\rightarrow_\pi \subseteq States \times States$ is defined by:

- If $stmtAt(\pi, \iota) = \mathbf{decl} \ x$ then $(\iota, \rho, \sigma, \xi, \langle l, l_1, l_2, \dots \rangle) \rightarrow_\pi (\iota + 1, \rho[x \mapsto l], \sigma[l \mapsto uninit], \xi, \langle l_1, l_2, \dots \rangle)$
where $l \notin dom(\sigma)$
- If $stmtAt(\pi, \iota) = \mathbf{skip}$ then $(\iota, \rho, \sigma, \xi, \mathcal{M}) \rightarrow_\pi (\iota + 1, \rho, \sigma, \xi, \mathcal{M})$
- If $stmtAt(\pi, \iota) = (lhs := e)$ then $(\iota, \rho, \sigma, \xi, \mathcal{M}) \rightarrow_\pi (\iota + 1, \rho, \sigma[\eta^l(lhs) \mapsto \eta(e)], \xi, \mathcal{M})$
where $\eta = (\iota, \rho, \sigma, \xi, \mathcal{M})$
- If $stmtAt(\pi, \iota) = x := \mathbf{new}$ then $(\iota, \rho, \sigma, \xi, \langle l, l_1, l_2, \dots \rangle) \rightarrow_\pi (\iota + 1, \rho, \sigma[\rho(x) \mapsto l][l \mapsto uninit], \xi, \langle l_1, l_2, \dots \rangle)$
where $x \in dom(\rho)$, $l \notin dom(\sigma)$
- If $stmtAt(\pi, \iota) = x := p_0(b)$ then $(\iota, \rho, \sigma, \xi, \langle l, l_1, l_2, \dots \rangle) \rightarrow_\pi (\iota_0, \{(y, l)\}, \sigma[l \mapsto \eta(b)], push(f, \xi), \langle l_1, l_2, \dots \rangle)$
where $\eta = (\iota, \rho, \sigma, \xi, \langle l, l_1, l_2, \dots \rangle)$, $\iota_0 = start_{p_0}$, $y = formal_{p_0}$, $l \notin dom(\sigma)$, $x \in dom(\rho)$, $f = (\iota + 1, \rho(x), \rho)$
- If $stmtAt(\pi, \iota) = (\mathbf{if} \ b \ \mathbf{goto} \ \iota_1 \ \mathbf{else} \ \iota_2)$ then $(\iota, \rho, \sigma, \xi, \mathcal{M}) \rightarrow_\pi (\iota_1, \rho, \sigma, \xi, \mathcal{M})$
where $(\iota, \rho, \sigma, \xi, \mathcal{M})(b) = true$
- If $stmtAt(\pi, \iota) = (\mathbf{if} \ b \ \mathbf{goto} \ \iota_1 \ \mathbf{else} \ \iota_2)$ then $(\iota, \rho, \sigma, \xi, \mathcal{M}) \rightarrow_\pi (\iota_2, \rho, \sigma, \xi, \mathcal{M})$
where $(\iota, \rho, \sigma, \xi, \mathcal{M})(b) \neq true$
- If $stmtAt(\pi, \iota) = \mathbf{return} \ x$ then $(\iota, \rho, \sigma, \xi, \mathcal{M}) \rightarrow_\pi (\iota_0, \rho_0, \sigma_0, \xi_0, \mathcal{M})$
where $pop(\xi) = ((\iota_0, l_0, \rho_0), \xi_0)$, $dom(\rho) = \{x_1, \dots, x_i\}$, $\sigma_0 = (\sigma / \{\rho(x_1), \dots, \rho(x_i)\})[l_0 \mapsto (\iota, \rho, \sigma, \xi, \mathcal{M})(x)]$

We denote the reflexive, transitive closure of \rightarrow_π as \rightarrow_π^* .

Definition 4 Given a program π , the intraprocedural state transition function $\hookrightarrow_\pi \subseteq \text{States} \times \text{States}$ is defined by:

- If $\text{stmtAt}(\pi, \iota)$ is not a procedure call, then $\eta \hookrightarrow_\pi \eta'$ where $\eta \rightarrow_\pi \eta'$
- If $\text{stmtAt}(\pi, \iota)$ is a procedure call, then $\eta \hookrightarrow_\pi \eta'$ where $\eta \rightarrow_\pi \eta'' \rightarrow_\pi^* \eta'$ and η' is the first state on the trace between η and η' such that $\text{stack}(\eta') = \text{stack}(\eta)$

We denote the reflexive, transitive closure of \hookrightarrow_π as \hookrightarrow_π^* .

We say that $\eta \not\hookrightarrow_\pi$ if there does not exist some η' such that $\eta \hookrightarrow_\pi \eta'$.

□

A.1.3 Programs and Program Transformations

Definition 5 The semantic function of a program π is the partial function $\llbracket \pi \rrbracket : \text{Consts} \times \text{MemAllocs} \rightarrow \text{Values}$ defined by: $\llbracket \pi \rrbracket(c, \langle l, l_1, l_2, \dots \rangle) = \sigma(l)$ where $(\iota, \{(x, l)\}, \{(l, \text{uninit})\}, \langle \rangle, \langle l_1, l_2, \dots \rangle) \rightarrow_\pi^* (\iota + 1, \{(x, l)\}, \sigma, \langle \rangle, \mathcal{M})$ and $\iota \notin \text{Indices}_\pi$ and $\text{stmtAt}(\pi, \iota)$ is defined to be $x := \text{main}(c)$.

Definition 6 We say that π' is a semantically equivalent transformation of π if for all c, \mathcal{M} such that $\llbracket \pi \rrbracket(c, \mathcal{M}) = v$, it is the case that $\llbracket \pi' \rrbracket(c, \mathcal{M}) = v$.

A.2 Semantics of Optimizations

Let $\text{stmtAt}(p, \iota)$ denote the statement at index ι of procedure p .

Definition 7 The control flow graph (CFG) of a procedure p is a graph $(\text{Indices}_p, \rightarrow_{\text{cfg}})$, where $\rightarrow_{\text{cfg}} \subseteq \text{Indices}_p \times \text{Indices}_p$ and $\iota_1 \rightarrow_{\text{cfg}} \iota_2$ if and only if:

$$\begin{aligned} & (\text{stmtAt}(p, \iota_1) \in \{\text{decl } x, \text{lhs} := e, \text{skip}, x := \text{new}, x := p(b), \text{return } x\} \wedge \iota_2 = \iota_1 + 1) \\ \vee & (\text{stmtAt}(p, \iota_1) = \text{if } b \text{ goto } \iota \text{ else } \iota' \wedge (\iota_2 = \iota \vee \iota_2 = \iota')) \end{aligned}$$

□

Before giving the definition of $\iota \models_\theta^p \psi$, we first define the complete grammar for ψ :

$$\begin{aligned} \psi & ::= \text{true} \mid \text{false} \mid \neg\psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid l(t, \dots, t) \mid t = t \mid \text{case } t \text{ of cases } \text{endcase} \\ \text{cases} & ::= \epsilon \mid \text{case cases} \\ \text{case} & ::= t \Rightarrow \psi \mid \text{decl vars in } t \Rightarrow \psi \\ \text{vars} & ::= \text{patvar} : \tau \mid \text{patvar} : \tau, \text{vars} \\ \text{patvar} & ::= X \mid Y \mid Z \mid E \mid \dots \\ \tau & ::= \text{Stmt} \mid \text{Expr} \mid \text{Call} \mid \text{NonCallExpr} \mid \text{LHS} \mid \text{BaseExpr} \mid \text{Const} \mid \text{Var} \mid \text{Deref} \mid \text{Ref} \mid \text{Fun} \mid \text{Index} \end{aligned}$$

The full language for ψ differs from the one presented in section 3 in two ways. First it requires pattern variables to be explicitly declared and typed. Second, it does not provide an **else** construct, instead requiring the user to enumerate all possible cases. The **else** construct can easily be added to the full language; it would be handled by desugaring it automatically into the unstated cases.

The **case** statement is desugared into ordinary quantifiers. This not only makes the definition of $\iota \models_{\theta}^p \psi$ simpler, but it is also required in order to transform **case** statements into Simplify's first-order predicate language. The language for ψ after **case** statements have been desugared is described by the following grammar, which is the same as above, but with the *cases* and *case* productions removed and the **case** predicate replaced by the \forall predicate:

$$\begin{aligned}
\psi &::= \text{true} \mid \text{false} \mid \neg\psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid l(t, \dots, t) \mid t = t \mid \forall \text{vars. } \psi \\
\text{vars} &::= \text{patvar} : \tau \mid \text{patvar} : \tau, \text{vars} \\
\text{patvar} &::= X \mid Y \mid Z \mid E \mid \dots \\
\tau &::= \text{Stmt} \mid \text{Expr} \mid \text{Call} \mid \text{NonCallExpr} \mid \text{LHS} \mid \text{BaseExpr} \mid \text{Const} \mid \text{Var} \mid \text{Deref} \mid \text{Ref} \mid \text{Fun} \mid \text{Index}
\end{aligned}$$

The **case** statement can be translated into the above desugared language as follows (where we use $\psi_1 \Rightarrow \psi_2$ as sugar for $\neg\psi_1 \vee \psi_2$):

$$\begin{array}{l}
\text{case } t \text{ of} \\
\quad \text{decl } \overrightarrow{X_1} \text{ in } t_1 \Rightarrow \psi_1 \\
\quad \dots \\
\quad \text{decl } \overrightarrow{X_n} \text{ in } t_n \Rightarrow \psi_n \\
\text{endcase}
\end{array}
\iff
\begin{array}{l}
\forall \overrightarrow{X_1}. (t = t_1 \Rightarrow \psi_1) \quad \wedge \\
\quad \dots \quad \wedge \\
\forall \overrightarrow{X_n}. (t = t_n \Rightarrow \psi_n)
\end{array}$$

Finally, we define the semantics of $\iota \models_{\theta}^p \psi$ only for the desugared language. The definition of $\iota \models_{\theta}^p \psi$, which evaluates a formula ψ at index ι in the CFG of procedure p under substitution θ , is as follows:

$$\begin{array}{l}
\iota \models_{\theta}^p \text{true} \iff \text{true} \\
\iota \models_{\theta}^p \text{false} \iff \text{false} \\
\iota \models_{\theta}^p \neg\psi \iff \text{not } \iota \models_{\theta}^p \psi \\
\iota \models_{\theta}^p \psi_1 \vee \psi_2 \iff \iota \models_{\theta}^p \psi_1 \text{ or } \iota \models_{\theta}^p \psi_2 \\
\iota \models_{\theta}^p \psi_1 \wedge \psi_2 \iff \iota \models_{\theta}^p \psi_1 \text{ and } \iota \models_{\theta}^p \psi_2 \\
\iota \models_{\theta}^p \forall X : \tau. \psi \iff \forall t : \tau. \iota \models_{\theta[X \mapsto t]}^p \psi \\
\iota \models_{\theta}^p l(t_1, \dots, t_n) \iff l(\theta[t_1]_{\iota}^p, \dots, \theta[t_n]_{\iota}^p) \in L_p(\iota) \\
\iota \models_{\theta}^p t_1 = t_2 \iff \theta[t_1]_{\iota}^p = \theta[t_2]_{\iota}^p
\end{array}$$

where the semantics of a term t at index ι in the CFG of procedure p under substitution θ is defined by:

$$\theta[t]_{\iota}^p = \begin{cases} \text{stmtAt}(p, \iota) & \text{if } t = \text{currStmt} \\ \theta(t) & \text{otherwise} \end{cases}$$

A.3 Proof Obligations

A.3.1 Forward Optimizations

Consider a forward transformation pattern of the following form:

ψ_1 **followed by** ψ_2 **until** $s \Rightarrow s'$ **with witness** \mathcal{P}

The optimization-specific obligations, to be discharged by an automatic theorem prover, are as follows:

- F1. If $\eta \hookrightarrow_{\pi} \eta'$ and $\text{index}(\eta) \models_{\theta}^p \psi_1$, then $\theta(\mathcal{P})(\eta')$.
- F2. If $\theta(\mathcal{P})(\eta)$ and $\eta \hookrightarrow_{\pi} \eta'$ and $\text{index}(\eta) \models_{\theta}^p \psi_2$, then $\theta(\mathcal{P})(\eta')$.
- F3. If $\theta(\mathcal{P})(\eta)$ and $\eta \hookrightarrow_{\pi} \eta'$ and $\iota = \text{index}(\eta)$ and $\text{stmtAt}(\pi, \iota) = \theta(s)$ and $\text{stmtAt}(\pi', \iota) = \theta(s')$, then $\eta \hookrightarrow_{\pi'} \eta'$.

A.3.2 Backward Optimizations

Consider a backward transformation pattern of the following form:

ψ_1 **preceded by** ψ_2 **since** $s \Rightarrow s'$ **with witness** \mathcal{P}

We require that the witness \mathcal{P} match program points in the original and transformed state, or in other words that $\mathcal{P} \Longrightarrow (index(\eta_{old}) = index(\eta_{new}))$. The optimization-specific obligations, to be discharged by an automatic theorem prover, are as follows:

- B1. If $\eta \hookrightarrow_{\pi} \eta_{old}$ and $\eta \hookrightarrow_{\pi'} \eta_{new}$ and $\iota = index(\eta)$ and $stmtAt(\pi, \iota) = \theta(s)$ and $stmtAt(\pi', \iota) = \theta(s')$, then $\theta(\mathcal{P})(\eta_{old}, \eta_{new})$.
- B2. If $\theta(\mathcal{P})(\eta_{old}, \eta_{new})$ and $\eta_{old} \hookrightarrow_{\pi} \eta'_{old}$ and $\iota_{old} = index(\eta_{old})$ and $\iota_{new} = index(\eta_{new})$ and $\iota_{old} \models_{\theta}^{\pi} \psi_2$ and $stmtAt(\pi, \iota_{old}) = stmtAt(\pi', \iota_{new})$, then there exists some η'_{new} such that $\eta_{new} \hookrightarrow_{\pi'} \eta'_{new}$ and $\theta(\mathcal{P})(\eta'_{old}, \eta'_{new})$.
- B3. If $\theta(\mathcal{P})(\eta_{old}, \eta_{new})$ and $\eta_{old} \hookrightarrow_{\pi} \eta$ and $\iota_{old} = index(\eta_{old})$ and $\iota_{new} = index(\eta_{new})$ and $\iota_{old} \models_{\theta}^{\pi} \psi_1$ and $stmtAt(\pi, \iota_{old}) = stmtAt(\pi', \iota_{new})$, then $\eta_{new} \hookrightarrow_{\pi'} \eta$.

In rule B1, we assume that both programs can step. However, we in fact need to prove that the transformed program steps if the original one does, in order to show that the transformed program is semantically equivalent to the original one. Unfortunately, it is not possible to prove this for B1 using only local knowledge. Therefore, we allow B1 to assume that the transformed program steps, and we separately prove the property using some additional obligations.

We introduce the notion of an *error predicate* $\epsilon(\eta)$. Intuitively, the error predicate says what the state of the *original* program must look like at the point in the trace where a transformation is allowed, if that transformation would get stuck. We then show that the error predicate would continue to hold on the original program throughout the witnessing region, eventually implying that the original program itself will get stuck. So we will have shown that the transformed program gets stuck only if the original one does.

We currently *infer* the error predicate: it is simply a predicate stating the conditions under which the transformed statement s' is “stuck” — it cannot take a step. This inference has been sufficient to automatically prove soundness of all the backward optimizations we have written. However, in our obligations below, we allow an arbitrary error predicate to be specified.

- B1'. If $\eta \hookrightarrow_{\pi} \eta_{old}$ and $\eta \not\hookrightarrow_{\pi'}$ and $\iota = index(\eta)$ and $stmtAt(\pi, \iota) = \theta(s)$ and $stmtAt(\pi', \iota) = \theta(s')$, then $\theta(\epsilon)(\eta_{old})$.
- B2'. If $\theta(\epsilon)(\eta)$ and $\eta \hookrightarrow_{\pi} \eta'$ and $\iota = index(\eta)$ and $\iota \models_{\theta}^{\pi} \psi_2$, then $\theta(\epsilon)(\eta')$.
- B3'. If $\theta(\epsilon)(\eta)$ and $\iota = index(\eta)$ and $\iota \models_{\theta}^{\pi} \psi_1$, then $\eta \not\hookrightarrow_{\pi}$.

A.3.3 Analyses

Consider a pure analysis of the following form:

ψ_1 **followed by** ψ_2 **defines label with witness** \mathcal{P}

The optimization-specific obligations, to be discharged by an automatic theorem prover, are as follows:

- A1. If $\eta \hookrightarrow_{\pi} \eta'$ and $index(\eta) \models_{\theta}^p \psi_1$, then $\theta(\mathcal{P})(\eta')$.
- A2. If $\theta(\mathcal{P})(\eta)$ and $\eta \hookrightarrow_{\pi} \eta'$ and $index(\eta) \models_{\theta}^p \psi_2$, then $\theta(\mathcal{P})(\eta')$.

A.4 Metatheory

A.4.1 Forward Optimizations

Theorem 1 If O is a forward optimization with transformation pattern ψ_1 **followed by** ψ_2 **until** $s \Rightarrow s'$ **with witness** \mathcal{P} satisfying conditions F1, F2, and F3, then O is sound.

Proof:

Let π be an intermediate-language program, p be a procedure in π , $\Delta \subseteq \llbracket O_{pat} \rrbracket(p)$, and let π_Δ be the program identical to π but with p replaced by $app(s', p, \Delta)$. It suffices to show that π_Δ is a semantically equivalent transformation of π . Let c be a constant and \mathcal{M} be a memory allocator such that $\llbracket \pi \rrbracket(c, \mathcal{M}) = v$. By definition 6 we must show that also $\llbracket \pi_\Delta \rrbracket(c, \mathcal{M}) = v$.

Since $\llbracket \pi \rrbracket(c, \mathcal{M}) = v$, by definition 5 we have that $(\iota, \{(x, l)\}, \{(l, \text{uninit})\}, \langle \rangle, \langle l_1, l_2, \dots \rangle) \rightarrow_\pi^* (\iota + 1, \{(x, l)\}, \sigma, \langle \rangle, \mathcal{M}')$ and $stmtAt(\pi, \iota)$ is defined to be $x := \text{main}(c)$ and $\iota \notin \text{Indices}_\pi$ and $v = \sigma(l)$, where $\mathcal{M} = \langle l, l_1, l_2, \dots \rangle$. Define \rightsquigarrow_π to act like \rightarrow_π ordinarily, but to act like \hookrightarrow_π when executing a state at some node with index ι_0 such that some $(\iota_0, \theta) \in \Delta$. Then we have that

$$(\iota, \{(x, l)\}, \{(l, \text{uninit})\}, \langle \rangle, \langle l_1, l_2, \dots \rangle) \rightsquigarrow_\pi \eta_1 \rightsquigarrow_\pi \eta_2 \rightsquigarrow_\pi \dots \rightsquigarrow_\pi \eta_{k-1} \rightsquigarrow_\pi (\iota + 1, \{(x, l)\}, \sigma, \langle \rangle, \mathcal{M}')$$

for some $\eta_1, \dots, \eta_{k-1}$. To prove that $\llbracket \pi_\Delta \rrbracket(c, \mathcal{M}) = v$ we will show that also

$$(\iota, \{(x, l)\}, \{(l, \text{uninit})\}, \langle \rangle, \langle l_1, l_2, \dots \rangle) \rightsquigarrow_{\pi_\Delta} \eta_1 \rightsquigarrow_{\pi_\Delta} \eta_2 \rightsquigarrow_{\pi_\Delta} \dots \rightsquigarrow_{\pi_\Delta} \eta_{k-1} \rightsquigarrow_{\pi_\Delta} (\iota + 1, \{(x, l)\}, \sigma, \langle \rangle, \mathcal{M}')$$

where $stmtAt(\pi_\Delta, \iota)$ is defined to be $x := \text{main}(c)$.

Let $\eta_k = (\iota + 1, \{(x, l)\}, \sigma, \langle \rangle, \mathcal{M}')$. We show by induction on k that every prefix of the trace in π up to η_j , for all $1 \leq j \leq k$, is mirrored in π_Δ .

- Case $j=1$. Since $stmtAt(\pi, \iota) = stmtAt(\pi_\Delta, \iota) = x := \text{main}(c)$ and $(\iota, \{(x, l)\}, \{(l, \text{uninit})\}, \langle \rangle, \langle l_1, l_2, \dots \rangle) \rightsquigarrow_\pi \eta_1$, by definition of \rightsquigarrow_π we have $(\iota, \{(x, l)\}, \{(l, \text{uninit})\}, \langle \rangle, \langle l_1, l_2, \dots \rangle) \rightarrow_\pi \eta_1$. Therefore also $(\iota, \{(x, l)\}, \{(l, \text{uninit})\}, \langle \rangle, \langle l_1, l_2, \dots \rangle) \rightarrow_{\pi_\Delta} \eta_1$, so $(\iota, \{(x, l)\}, \{(l, \text{uninit})\}, \langle \rangle, \langle l_1, l_2, \dots \rangle) \rightsquigarrow_{\pi_\Delta} \eta_1$.
- Case $1 < j \leq k$. By induction we have that $(\iota, \{(x, l)\}, \{(l, \text{uninit})\}, \langle \rangle, \langle l_1, l_2, \dots \rangle) \rightsquigarrow_{\pi_\Delta} \eta_1 \rightsquigarrow_{\pi_\Delta} \dots \rightsquigarrow_{\pi_\Delta} \eta_{j-1}$. Let $index(\eta_{j-1}) = \iota_{j-1}$. We have two sub-cases:

– $\neg \exists \theta. ((\iota_{j-1}, \theta) \in \Delta)$. Then by the definition of π_Δ we have $stmtAt(\pi, \iota_{j-1}) = stmtAt(\pi_\Delta, \iota_{j-1})$. Therefore, since $\eta_{j-1} \rightsquigarrow_\pi \eta_j$, by definition of \rightsquigarrow_π we have $\eta_{j-1} \rightarrow_\pi \eta_j$. Then also $\eta_{j-1} \rightarrow_{\pi_\Delta} \eta_j$, so $\eta_{j-1} \rightsquigarrow_{\pi_\Delta} \eta_j$ and the result follows.

– $\exists \theta. ((\iota_{j-1}, \theta) \in \Delta)$. Then by the definition of π_Δ , there is some θ such that $stmtAt(\pi_\Delta, \iota_{j-1}) = \theta(s')$. Then also $(\iota_{j-1}, \theta) \in \llbracket O_{pat} \rrbracket(p)$, so we have $\iota_{j-1} \models_\theta^p stmt(i)$, so $stmtAt(\pi, \iota_{j-1}) = \theta(s)$.

By definition of \rightsquigarrow_π we know that $(\iota, \{(x, l)\}, \{(l, \text{uninit})\}, \langle \rangle, \langle l_1, l_2, \dots \rangle) \rightarrow_\pi^* \eta_{j-1}$, and we also have $stmtAt(\pi, \iota) = x := \text{main}(c)$ and $\iota_{j-1} \in \text{Indices}_p$. Assume

$$(\iota, \{(x, l)\}, \{(l, \text{uninit})\}, \langle \rangle, \langle l_1, l_2, \dots \rangle) \rightarrow_\pi \eta'_1 \rightarrow_\pi \dots \rightarrow_\pi \eta'_v$$

where $\eta'_v = \eta_{j-1}$. Then there must be some t such that $1 \leq t < v$ and $index(\eta'_t) = start_p$, representing the first statement executed on the same invocation of p as in η_{j-1} . Then let $\eta''_w, \dots, \eta''_1$ be identical to the sequence η'_v, \dots, η'_t , but with all states that are not in the same invocation of p as in η_{j-1} removed. Let $index(\eta''_x) = \iota''_x$ for all $1 \leq x \leq w$. Then $\iota''_w = \iota_{j-1}$. It is easy to show that $\eta''_w \hookrightarrow_\pi \dots \hookrightarrow_\pi \eta''_1$. Also, by the definition of an intraprocedural CFG we have that $\eta''_w, \dots, \eta''_1$ represents a backward path in the CFG of p to the entry node. Therefore, since $(\iota_{j-1}, \theta) \in \llbracket O_{pat} \rrbracket(p)$, it follows that there exists some r such that $1 \leq r < w$ and $\iota''_r \models_\theta^p \psi_1$, and for all q such that $r < q < w$ we have $\iota''_q \models_\theta^p \psi_2$.

First we prove $\forall q. (r < q \leq w) \Rightarrow \theta(\mathcal{P})(\eta''_q)$. We prove it by induction on q :

- * (base case) $q = r + 1$. So we have $\eta_r'' \hookrightarrow_\pi \eta_q''$ and $index(\eta_r'') \models_\theta^p \psi_1$, and the result follows from condition F1.
- * (inductive case) $q > r + 1$. By the inductive hypothesis we have $\theta(\mathcal{P})(\eta_{q-1}'')$. We also know that $\eta_{q-1}'' \hookrightarrow_p \eta_q''$ and, since $r < q - 1 < w$, $index(\eta_{q-1}'') \models_\theta^p \psi_2$. Then the result follows from condition F2.

So we have shown in particular that $\theta(\mathcal{P})(\eta_{j-1})$ holds. We saw above that $\eta_{j-1} \rightsquigarrow_\pi \eta_j$, and by definition of \rightsquigarrow_π that means $\eta_{j-1} \hookrightarrow_\pi \eta_j$. We also know that $stmtAt(\pi, \iota_{j-1}) = \theta(s)$ and $stmtAt(\pi_\Delta, \iota_{j-1}) = \theta(s')$. Then by condition F3 we have $\eta_{j-1} \hookrightarrow_{\pi_\Delta} \eta_j$, so also $\eta_{j-1} \rightsquigarrow_{\pi_\Delta} \eta_j$ and the result follows.

A.4.2 Backward Optimizations

Lemma 1 *Let O be a backward optimization with transformation pattern ψ_1 **preceded by** ψ_2 **since** $s \Rightarrow s'$ **with witness** \mathcal{P} and error predicate ϵ such that B1'-B3' hold. Let p be a procedure, π be a program containing p , $\iota \in Indices_p$, and $stmtAt(\pi, \iota) = \theta(s)$. Let η be a state such that $index(\eta) = \iota$. Let π' be a program such that $stmtAt(\pi', \iota) = \theta(s')$ and $\eta \not\hookrightarrow_{\pi'}$. If*

$$\eta \hookrightarrow_\pi \eta_1 \hookrightarrow_\pi \dots \hookrightarrow_\pi \eta_k$$

and for all $1 \leq j < k$ we have $index(\eta_j) \models_\theta^p \psi_2$ and $index(\eta_k) \models_\theta^p \psi_1$, then $\eta_k \not\hookrightarrow_\pi$.

Proof: We will first prove by induction on k that $\theta(\epsilon(\eta_j))$ holds for all $1 \leq j \leq k$.

- Case $j = 1$. Since $\eta \rightarrow_\pi \eta_1$ and $\eta \not\hookrightarrow_{\pi'}$ and $stmtAt(\pi, \iota) = \theta(s)$ and $stmtAt(\pi', \iota) = \theta(s')$, the result follows from B1'.
- Case $1 < j \leq k$. By induction, assume that $\theta(\eta_{j-1})$ holds. We are given that $\eta_{j-1} \rightarrow_\pi \eta_j$, and since $(j - 1) \leq k$ we also have that $index(\eta_{j-1}) \models_\theta^p \psi_2$. Then the result follows from B2'.

So in particular we have shown that $\theta(\eta_k)$ holds. We are given that $index(\eta_k) \models_\theta^p \psi_1$, so by B3' we have $\eta_k \not\hookrightarrow_\pi$.

Theorem 2 If O is a backward optimization with transformation pattern ψ_1 **preceded by** ψ_2 **since** $s \Rightarrow s'$ **with witness** \mathcal{P} with error predicate ϵ and satisfying conditions B1, B2, B3, B1', B2', and B3', then O is sound.

Proof: Let π be an intermediate-language program, p be a procedure in π , $\Delta \subseteq \llbracket O_{pat} \rrbracket(p)$, and let π_Δ be the program identical to π but with p replaced by $app(s', p, \Delta)$. It suffices to show that π_Δ is a semantically equivalent transformation of π .

We define an infinite family of *generalized* intermediate-language programs as follows. Let π_Δ^j denote the program that acts like π_Δ for the first j states but henceforth acts like π . Formally, we define the transition relation of π_Δ^j directly as a relation $\rightarrow_{\pi_\Delta^j}$ on prefixes of execution traces, rather than as a relation on states.

Let $T = [\eta_1 \dots \eta_r]$ denote a partial trace of π_Δ^j such that $index(\eta_1) \notin Indices_{\pi_\Delta^j}$ and $s_{index(\eta_1)}^{\pi_\Delta^j}$ is a call to **main**. We say that $T \rightarrow_{\pi_\Delta^j} T'$ if and only if $T' = [\eta_1 \dots \eta_{r+1}]$, where

- $r \leq j \Rightarrow \eta_r \rightarrow_{\pi_\Delta} \eta_{r+1}$
- $r > j \Rightarrow \eta_r \rightarrow_\pi \eta_{r+1}$

Let $\rightarrow_{\pi_\Delta^j}^*$ denote the reflexive, transitive closure of $\rightarrow_{\pi_\Delta^j}$. We also define an intraprocedural version $\hookrightarrow_{\pi_\Delta^j}$ in the identical way that \hookrightarrow_π is defined for \rightarrow_π . Finally, we define the semantic function of π_Δ^j by the straightforward modification of Definition 5.

We prove that for all $j \geq 1$, π_Δ^j is a semantically equivalent transformation of π . Since $\pi_\Delta = \pi_\Delta^\infty$ and the semantic equivalence relation is transitive, it then follows easily that π_Δ is a semantically equivalent transformation of π . The proof proceeds by induction on j .

For the base case, $j = 1$. Let c be a constant and $\mathcal{M} = \langle l, l_1, l_2, \dots \rangle$ such that $\llbracket \pi \rrbracket(c, \mathcal{M}) = v$. By Definition 6 we must show that $\llbracket \pi_\Delta^1 \rrbracket(c, \mathcal{M}) = v$. By Definition 5 we have that $v = \sigma(l)$ and $(\iota, \{(x, l)\}, \{(l, \text{uninit})\}, \langle \rangle, \langle l_1, l_2, \dots \rangle) \xrightarrow{\ast}_\pi (\iota + 1, \{(x, l)\}, \sigma, \langle \rangle, \mathcal{M})$ and $\text{stmtAt}(\pi, \iota)$ is defined to be $x := \text{main}(c)$ and $\iota \notin \text{Indices}_\pi$. Therefore assume that

$$(\iota, \{(x, l)\}, \{(l, \text{uninit})\}, \langle \rangle, \langle l_1, l_2, \dots \rangle) \rightarrow_\pi \eta_2 \rightarrow_\pi \dots \rightarrow_\pi \eta_k \rightarrow_\pi (\iota + 1, \{(x, l)\}, \sigma, \langle \rangle, \mathcal{M})$$

Let $\eta_1 = (\iota, \{(x, l)\}, \{(l, \text{uninit})\}, \langle \rangle, \langle l_1, l_2, \dots \rangle)$ and $\eta_{k+1} = (\iota + 1, \{(x, l)\}, \sigma, \langle \rangle, \mathcal{M})$. Also let $s_n^{\pi_\Delta}$ be $x := \text{main}(c)$. Then we claim that

$$[\eta_1] \rightarrow_{\pi_\Delta^1} [\eta_1, \eta_2] \rightarrow_{\pi_\Delta^1} \dots \rightarrow_{\pi_\Delta^1} [\eta_1, \dots, \eta_k, \eta_{k+1}]$$

If we can prove this, then the result follows. We prove inductively that each transition in the above sequence of transitions holds.

- Base Case. We must show that $[\eta_1] \rightarrow_{\pi_\Delta^1} [\eta_1, \eta_2]$. We're given that $\eta_1 \rightarrow_\pi \eta_2$ and $\text{stmtAt}(\pi, \iota) = \text{stmtAt}(\pi_\Delta^1, \iota) = x := \text{main}(c)$. Then $\eta_1 \rightarrow_{\pi_\Delta} \eta_2$, so by the definition of $\rightarrow_{\pi_\Delta^1}$ the result follows.
- Inductive Case. By induction we have $[\eta_1] \rightarrow_{\pi_\Delta^1} [\eta_1, \eta_2] \rightarrow_{\pi_\Delta^1} \dots \rightarrow_{\pi_\Delta^1} [\eta_1, \dots, \eta_q]$, for some $1 < q \leq k$. We're given that $\eta_q \rightarrow_\pi \eta_{q+1}$. Then by the definition of $\rightarrow_{\pi_\Delta^1}$ we have that $[\eta_1, \dots, \eta_q] \rightarrow_{\pi_\Delta^1} [\eta_1, \dots, \eta_{q+1}]$.

For the inductive case, $j > 1$ and π_Δ^{j-1} is a semantically equivalent transformation of π . We will prove that π_Δ^j is a semantically equivalent transformation of π_Δ^{j-1} . Let c be a constant and $\mathcal{M} = \langle l, l_1, l_2, \dots \rangle$ such that $\llbracket \pi \rrbracket(c, \mathcal{M}) = v$. It suffices to show that $\llbracket \pi_\Delta^j \rrbracket(c, \mathcal{M}) = v$. Since π_Δ^{j-1} is a semantically equivalent transformation of π , we know that $\llbracket \pi_\Delta^{j-1} \rrbracket(c, \mathcal{M}) = v$. Then we have that $v = \sigma(l)$ and $[(\iota, \{(x, l)\}, \{(l, \text{uninit})\}, \langle \rangle, \langle l_1, l_2, \dots \rangle) \xrightarrow{\ast}_{\pi_\Delta^{j-1}} (\iota, \{(x, l)\}, \{(l, \text{uninit})\}, \langle \rangle, \langle l_1, l_2, \dots \rangle), \eta_2, \dots, \eta_k, (\iota + 1, \{(x, l)\}, \sigma, \langle \rangle, \mathcal{M})]$ and $\text{stmtAt}(\pi, \iota) = \text{stmtAt}(\pi_\Delta^{j-1}, \iota)$ is defined to be $x := \text{main}(c)$ and $\iota \notin \text{Indices}_\pi$. Therefore assume that

$$[\eta_1] \rightarrow_{\pi_\Delta^{j-1}} [\eta_1, \eta_2] \rightarrow_{\pi_\Delta^{j-1}} \dots \rightarrow_{\pi_\Delta^{j-1}} [\eta_1, \dots, \eta_{k+1}]$$

where $\eta_1 = (\iota, \{(x, l)\}, \{(l, \text{uninit})\}, \langle \rangle, \langle l_1, l_2, \dots \rangle)$ and $\eta_{k+1} = (\iota + 1, \{(x, l)\}, \sigma, \langle \rangle, \mathcal{M})$.

For each $1 \leq t \leq k + 1$, let $\Phi(t, \theta)$ denote the following predicate:

$$\begin{aligned} & t > j \wedge \\ & (\text{index}(\eta_j), \theta) \in \Delta \wedge \\ & \text{stmtAt}(\pi, \text{index}(\eta_j)) = \theta(s) \wedge \\ & \text{stmtAt}(\pi_\Delta, \text{index}(\eta_j)) = \theta(s') \wedge \\ & \forall m. ((j < m < t \wedge \eta_j \xrightarrow{\ast}_\pi \eta_m) \Rightarrow \text{index}(\eta_m) \not\models_\theta^\pi \psi_1) \end{aligned}$$

Define $\rightsquigarrow_{\pi_\Delta^{j-1}}$ as a view on the above execution trace, in the following way: $\rightsquigarrow_{\pi_\Delta^{j-1}}$ acts like $\rightarrow_{\pi_\Delta^{j-1}}$ ordinarily, but it acts like $\xrightarrow{\ast}_{\pi_\Delta^{j-1}}$ when it is either at a state η_t such that $\Phi(t, \theta)$ for some θ , or it is at the state η_j , where $(\text{index}(\eta_j), \theta) \in \Delta$. Then we have

$$[\eta'_1] \rightsquigarrow_{\pi_\Delta^{j-1}} [\eta'_1, \eta'_2] \rightsquigarrow_{\pi_\Delta^{j-1}} \dots \rightsquigarrow_{\pi_\Delta^{j-1}} [\eta'_1, \dots, \eta'_z]$$

where $\eta_1 = \eta'_1$ and $\eta_{k+1} = \eta'_z$.

Then we claim that

$$[\eta_1''] \rightsquigarrow_{\pi_\Delta^j} [\eta_1'', \eta_2''] \rightsquigarrow_{\pi_\Delta^j} \cdots \rightsquigarrow_{\pi_\Delta^j} [\eta_1'', \eta_2'', \dots, \eta_z'']$$

where $\rightsquigarrow_{\pi_\Delta^j}$ acts like $\rightarrow_{\pi_\Delta^j}$ ordinarily, but acts like $\hookrightarrow_{\pi_\Delta^j}$ when it is either at a state η_y'' such that $\eta_y' = \eta_t$ and $\Phi(t, \theta)$ for some θ , or it is at a state η_y'' such that $\eta_y' = \eta_j$, where $(\text{index}(\eta_j), \theta) \in \Delta$. Further, each η_y'' is defined as follows. For each $1 \leq y \leq z$:

- If there exists θ such that $\Phi(t, \theta)$, where $1 \leq t \leq k+1$ and $\eta_y' = \eta_t$, then $\theta(\mathcal{P})(\eta_y', \eta_y'')$.
- Else $\eta_y' = \eta_y''$.

If we can prove this, then we have that $\eta_z'' = \eta_z' = \eta_{k+1}$, and the result follows.

We prove inductively that each of the partial traces in the sequence above exists.

- For the base case, $y = 1$. We saw above that $\eta_1 = \eta_1'$. Since $j > 1$, we have $1 \not\asymp j$, so $\forall \theta. \neg \Phi(1, \theta)$. Therefore we must prove that $\eta_1' = \eta_1''$. We're given that $\eta_1 = (\iota, \{(x, l)\}, \{(l, \text{uninit})\}, \langle \rangle, \langle l_1, l_2, \dots \rangle)$ and $\text{stmtAt}(\pi, \iota) = x := \text{main}(c)$. Therefore $[\eta_1']$ is a valid partial trace for π_Δ^j .
- For the inductive case, $y > 1$ and $[\eta_1'', \dots, \eta_{y-1}'']$ is a valid partial trace for π_Δ^j with each component state meeting the definition above. We must show that there exists η_y'' meeting the definition above such that $[\eta_1'', \dots, \eta_y'']$ is a valid partial trace for π_Δ^j . Let t be the integer between 1 and $k+1$ such that $\eta_{y-1}' = \eta_t$. There are several cases.

– $t < j$. Since $t \not\asymp j$, by definition of Φ we have that $\eta_{y-1}' = \eta_{y-1}'$. By the definition of $\rightsquigarrow_{\pi_\Delta^{j-1}}$, $\eta_{y-1}' \rightarrow_{\pi_\Delta} \eta_y'$. Then by definition of $\rightsquigarrow_{\pi_\Delta^j}$ we have $[\eta_1'', \dots, \eta_{y-1}''] \rightsquigarrow_{\pi_\Delta^j} [\eta_1'', \dots, \eta_{y-1}'', \eta_y']$. Since $t+1 \not\asymp j$, by definition of Φ we must show that $\eta_y'' = \eta_y'$, so the result follows.

– $t = j$. Then by definition of Φ we have that $\eta_{y-1}' = \eta_{y-1}'$. There are two sub-cases.

* $\neg \exists \theta. (\text{index}(\eta_j), \theta) \in \Delta$. Then by definition of $\rightsquigarrow_{\pi_\Delta^{j-1}}$, we have $\eta_{y-1}' \rightarrow_\pi \eta_y'$. Also $\text{stmtAt}(\pi, \text{index}(\eta_j)) = \text{stmtAt}(\pi_\Delta, \text{index}(\eta_j))$, so we have $\eta_{y-1}' \rightarrow_{\pi_\Delta} \eta_y'$. Therefore by definition of $\rightsquigarrow_{\pi_\Delta^j}$ we have $[\eta_1'', \dots, \eta_{y-1}''] \rightsquigarrow_{\pi_\Delta^j} [\eta_1'', \dots, \eta_{y-1}'', \eta_y']$. Since $\neg \exists \theta. (\text{index}(\eta_j), \theta) \in \Delta$, by definition of Φ we must show that $\eta_y' = \eta_y''$, so the result follows.

* $\exists \theta. (\text{index}(\eta_j), \theta) \in \Delta$. Then by definition of π_Δ , there is some θ such that $(\text{index}(\eta_j), \theta) \in \Delta$ and $\text{stmtAt}(\pi_\Delta, \text{index}(\eta_j)) = \theta(s')$. Then by definition of $\rightsquigarrow_{\pi_\Delta^{j-1}}$, we have $\eta_{y-1}' \hookrightarrow_\pi \eta_y'$. Further, since $\Delta \subseteq \llbracket O_{pat} \rrbracket$, we have $\text{stmtAt}(\pi, \text{index}(\eta_j)) = \theta(s)$. Let $\eta_y' = \eta_{t'}$, for some $j < t' \leq k+1$. Since $\eta_{y-1}' \hookrightarrow_\pi \eta_y'$, we vacuously have that $\forall m. ((j < m < t' \wedge \eta_j \hookrightarrow_\pi^* \eta_m) \Rightarrow \text{index}(\eta_m) \not\models_{\theta}^\pi \psi_1)$. Therefore we have shown $\Phi(t', \theta)$, so by definition of Φ and $\rightsquigarrow_{\pi_\Delta^j}$ we have to show that there exists η_y'' such that $\eta_{y-1}' \hookrightarrow_{\pi_\Delta} \eta_y''$ and $\theta(\mathcal{P})(\eta_y', \eta_y'')$.

We have two cases. Suppose there exists η_y'' such that $\eta_{y-1}' \hookrightarrow_{\pi_\Delta} \eta_y''$. Then by condition B1 the result follows.

Now suppose there does not exist η_y'' such that $\eta_{y-1}' \hookrightarrow_{\pi_\Delta} \eta_y''$, so that $\eta_{y-1}' \not\hookrightarrow_{\pi_\Delta}$. We are given that $(\text{index}(\eta_j), \theta) \in \Delta$. By definition of π_Δ^{j-1} we know that π_Δ^{j-1} acts like π from η_j on in the sequence $[\eta_1, \dots, \eta_{k+1}]$. Further, $\eta_{k+1} = (\iota + 1, \{(x, l)\}, \sigma, \langle \rangle, \mathcal{M})$, where $\iota + 1 \notin \text{Indices}_\pi$. Therefore one of the states between η_j and η_{k+1} exclusive must represent the **return** node from the same invocation of p as η_j . Therefore we have that there exists some $j < r \leq k$ such that $\text{index}(\eta_r) \models_{\theta}^\pi \psi_1$ and for all $t \leq q < r$ such that η_q is in the same invocation of p as η_t , we have $\text{index}(\eta_q) \not\models_{\theta}^\pi \psi_2$. Then by Lemma 1 we have $\eta_r \not\hookrightarrow_\pi$, and we have a contradiction.

– $t > j$. There are two sub-cases.

- * $\neg\exists\theta.\Phi(t, \theta)$. By definition of $\rightsquigarrow_{\pi_{\Delta}^{j-1}}$, we have $\eta'_{y-1} \rightarrow_{\pi} \eta'_y$. Therefore $\eta''_{y-1} = \eta'_{y-1}$, and for all θ we have that either $(index(\eta_j), \theta) \notin \Delta$ or $stmtAt(\pi, \eta_j) \neq \theta(s)$ or $stmtAt(\pi_{\Delta}, \eta_j) \neq \theta(s')$ or $\exists m.(j < m < t \wedge \eta_j \hookrightarrow_{\pi}^* \eta_m \wedge index(\eta_m) \not\models_{\theta}^{\pi} \psi_1)$. Then also $\neg\exists\theta.\Phi(t', \theta)$, where $\eta'_y = \eta_{t'}$, so by the definition of $\rightsquigarrow_{\pi_{\Delta}^j}$ we must show that $\eta''_{y-1} \rightarrow_{\pi} \eta''_y$, where $\eta'_y = \eta''_y$. Since $\eta'_{y-1} \rightarrow_{\pi} \eta'_y$, the result follows.
- * $\exists\theta.\Phi(t, \theta)$. Therefore $\theta(\mathcal{P})(\eta'_{y-1}, \eta''_{y-1})$ and by definition of $\rightsquigarrow_{\pi_{\Delta}^{j-1}}$, we have $\eta'_{y-1} \hookrightarrow_{\pi} \eta'_y$. We have two sub-cases.
 - $index(\eta_t) \not\models_{\theta}^{\pi} \psi_1$. Then since $\eta'_{y-1} \rightarrow_{\pi} \eta'_y$, we have $\forall m.((j < m < t' \wedge \eta_j \hookrightarrow_{\pi}^* \eta_m) \Rightarrow index(\eta_m) \not\models_{\theta}^{\pi} \psi_1)$, where $\eta'_y = \eta_{t'}$, so $\Phi(t', \theta)$. Then we must show that $\eta''_{y-1} \hookrightarrow_{\pi} \eta''_y$, where $\theta(\mathcal{P})(\eta'_y, \eta''_y)$. Since $\exists\theta.\Phi(t, \theta)$, we have $(index(\eta_j), \theta) \in \Delta$. We know that $\eta_{j+1} \rightarrow_{\pi} \dots \rightarrow_{\pi} \eta_t$. Therefore either $index(\eta_w) \models_{\theta}^{\pi} \psi_2$ for all $j+1 \leq w < t$ such that η_w is a state in the same invocation of p as η_j , or there exists $j+1 \leq w < t$ such that $index(\eta_w) \models_{\theta}^{\pi} \psi_1$ and η_w is a state in the same invocation of p as η_j . Since we saw above that $\forall m.((j < m < t' \wedge \eta_j \hookrightarrow_{\pi}^* \eta_m) \Rightarrow index(\eta_m) \not\models_{\theta}^{\pi} \psi_1)$, it must be the case that $index(\eta_t) \models_{\theta}^{\pi} \psi_2$. Therefore the result follows from condition B2.
 - $index(\eta_t) \models_{\theta}^{\pi} \psi_1$. Then by the definition of Φ we have $\neg\Phi(t', \theta)$, where $\eta'_y = \eta_{t'}$. Since θ is the unique substitution such that $stmtAt(\pi, index(\eta_j)) = \theta(s)$ and $stmtAt(\pi_{\Delta}, index(\eta_j)) = \theta(s')$, we have $\neg\exists\theta.\Phi(t', \theta)$. Therefore by the definition of $\rightsquigarrow_{\pi_{\Delta}^j}$ we must show that $\eta''_{y-1} \hookrightarrow_{\pi} \eta''_y$. The result follows from condition B3.

A.4.3 Pure Analyses

Let ψ_1 **followed by** ψ_2 **defines label with witness** \mathcal{P} be a pure analysis. We say that the analysis is *sound* if for all programs π , all procedures p in π , all indices ι in $Indices_p$, and all substitutions θ , the following condition holds: If the analysis puts a label of the form $\theta(label)$ on the node of p 's CFG with index ι , then $\theta(\mathcal{P})$ holds at all program states η of all execution traces of π such that $index(\eta) = \iota$.

Theorem 3 *If ψ_1 followed by ψ_2 defines label with witness \mathcal{P} is a pure analysis satisfying conditions A1 and A2, then the analysis is sound.*

Proof: Identical to the argument used in the proof of Theorem 1 to show that $\theta(\mathcal{P})$ holds at any execution's program state just before a transformed statement is executed. (Note that conditions A1 and A2 are the same as F1 and F2.)

B Source code of optimizations we have expressed in Cobalt

```

-----
--
-- We first define the optimizations, and the label definitions
-- follow. The following is a simple self-assignment removal
-- optimization.
--
-----

opt self_assign_elim =
  decl X:Var in
    TRUE
  followed_by
    FALSE

```

```

until
  X := X ==> skip
with_witness
  TRUE
-----
--
-- Copy propagation
-----

opt copy_prop =
  decl X:Var,Y:Var,Z:Var in
    stmt(Y := Z)
  followed_by
    AND(NOT(mayDef(Y)), NOT(mayDef(Z)))
  until
    X := Y ==> X := Z
  with_witness
    EQ(evalExpr(eta,Y),evalExpr(eta,Z))
-----
--
-- Contant propagation.
-----

opt const_prop1 =
  decl X:Var,Y:Var,C:Const in
    stmt(Y := C)
  followed_by
    NOT(mayDef(Y))
  until
    X := Y ==> X := C
  with_witness
    -- We use evalExpr(eta,C) instead of just C because otherwise
    -- there would be a type mismatch. The pattern variable C ranges
    -- over AST nodes of type constant, and the AST node contains
    -- the actual constant value. For example, C could be bound to
    -- an AST node Const(i), where i is the constant value stored at
    -- the AST node. However, the lhs of the equality is
    -- evalExpr(eta,Y), which evaluates to a value, not an AST
    -- node. So we need to convert the AST node C into the constant
    -- it stores, and evalExpr(eta,C) does exactly that. It would be
    -- easy to automatically insert such conversions.
    EQ(evalExpr(eta,Y), evalExpr(eta,C))

opt const_prop2 =
  decl X:Var,Y:Var,B:BaseExpr,C:Const in
    stmt(Y := C)
  followed_by
    NOT(mayDef(Y))
  until
    X := +(B,Y) ==> X := +(B,C)
  with_witness
    EQ(evalExpr(eta,Y), evalExpr(eta,C))

opt const_prop3 =
  decl X:Var,Y:Var,B:BaseExpr,C:Const in
    stmt(Y := C)

```

```

followed_by
  NOT(mayDef(Y))
until
  X := +(Y,B) ==> X := +(C,B)
with_witness
  EQ(evalExpr(eta,Y), evalExpr(eta,C))

opt const_prop4 =
  decl Y:Var,I1:Index,I2:Index,C:Const in
    stmt(Y := C)
  followed_by
    NOT(mayDef(Y))
  until
    if Y goto I1 else I2 ==> if C goto I1 else I2
  with_witness
    EQ(evalExpr(eta,Y), evalExpr(eta,C))

-----
--
-- Constant folding
--
-----

opt const_fold =
  TRUE
  followed_by
    FALSE
  until
    -- semanticPlus, which is a primitive, represents [[+]], the
    -- semantic function for the + symbol. The backquote ` operator
    -- says that semanticPlus(C1,C2) must be evaluated first, and
    -- then the resulting constant is used to create the rhs of the
    -- transformed statement. If we did not use backquote, the
    -- transformed statement would be interpreted as an assignment
    -- whose rhs is a call to semanticPlus.
    X := +(C1,C2) ==> X := `semanticPlus(C1,C2)`
  with_witness
    TRUE

-----
--
-- branch folding
--
-----

opt branch_folding1 =
  decl I1:Index,I2:Index in
    TRUE
  followed_by
    FALSE
  until
    if true goto I1 else I2 ==> if true goto I1 else I1
  with_witness
    TRUE

opt branch_folding2 =
  decl I1:Index,I2:Index in
    TRUE
  followed_by

```

```

    FALSE
  until
    if false goto I1 else I2 ==> if false goto I2 else I2
  with_witness
    TRUE
-----
--
-- Common subexpression elimination
--
-----

opt cse =
  decl E:NonCallExpr, Z:Var, X:Var in
    AND(stmt(Z := E),syntacticUnchanged(E))
  followed_by
    AND(NOT(mayDef(Z)),unchanged(E))
  until
    X := E ==> X := Z
  with_witness
    EQ(evalExpr(eta,Z), evalExpr(eta,E))
-----
--
-- Dead assignment elimination. We use the non-pointer versions of
-- labels (e.g. npMayUse) because in backwards optimizations, we cannot
-- use the result of a forward analysis.
--
-----

opt dead_assign_elim =
  decl X:Var,E:NonCallExpr,Y:Var in

    AND(OR(syntacticDef(X),stmt(return Y)),
        NOT(npMayUse(X)))

  preceded_by
    NOT(npMayUse(X))

  until
    X := E ==> skip

  with_witness
    equalUpTo(X,eta1,eta2)
-----
--
-- Code hoisting. We use the non-pointer versions of labels
-- (e.g. npMayUse, npMayDef) because in backwards optimizations, we cannot use
-- the result of a forward analysis.
--
-- With an appropriate profitability heuristic, this transformation
-- pattern can be used to implement PRE.
--
-----

opt code_dup_hoisting =
  decl X:Var,E:NonCallExpr in

```

```

    AND(stmt(X := E),
        NOT(npMayUse(X)))

preceded_by
    AND(npUnchanged(E),
        NOT(npMayUse(X)),
        NOT(npMayDef(X)))

until
    skip ==> X := E

with_witness
    equalUpTo(X,eta1,eta2)
-----
--
-- Code sinking. With an appropriate profitability
-- heuristic, this transformation pattern can be used
-- to implement partial dead code elimination.
--
-----

opt code_dup_sinking =
  decl X:Var,E:NonCallExpr in

    AND(stmt(X := E),
        syntacticUnchanged(E))

  followed_by
    AND(unchanged(E),
        NOT(mayUse(X)),
        NOT(mayDef(X)))

  until
    skip ==> X := E

  with_witness
    EQ(evalExpr(eta,X),evalExpr(eta,E))

-----
--
-- Removal of redundant loads.
--
-----

opt load_removal =
  decl X:Var,Y:Var,Z:Var in
    stmt(Y := &Z)
  followed_by
    NOT(mayDef(Y))
  until
    X := *Y ==> X := Z
  with_witness
    EQ(evalExpr(eta,Y),evalExpr(eta,&Z))

-----
--
-- We now define labels. We start with the stmt label

```

```

--
-----
define stmt(S:Stmt) = EQ(S,currStmt)
-----
--
-- syntacticDef label
--
-----

define syntacticDef(Z:Var) =
  CASE currStmt OF
    skip => FALSE
    decl X in decl X => EQ(Z,X)
    decl X:Var,I1:Index,I2:Index in if X goto I1 else I2 => FALSE
    decl C:Const,I1:Index,I2:Index in if C goto I1 else I2 => FALSE
    decl X:Var,C:Const in X := C => EQ(Z,X)
    decl X:Var,Y:Var in X := Y => EQ(Z,X)
    decl X:Var,Y:Var in *X := Y => FALSE
    decl X:Var,Y:Var in X := *Y => EQ(Z,X)
    decl X:Var,Y:Var in X := &Y => EQ(Z,X)
    decl X:Var in X := new => EQ(Z,X)
    decl X:Var,A:Var,B:Var in X := +(A,B) => EQ(Z,X)
    decl X:Var,A:Var,C:Const in X := +(A,C) => EQ(Z,X)
    decl X:Var,C:Const,B:Var in X := +(C,B) => EQ(Z,X)
    decl X:Var,C1:Const,C2:Const in X := +(C1,C2) => EQ(Z,X)
    decl X:Var,Y:Var,P:Fun in X := P(Y) => EQ(Z,X)
    decl X:Var in return X => FALSE
  ENDCASE
-----
--
-- syntacticUse label
--
-----

define syntacticUse(Z:Var) =
  CASE currStmt OF
    skip => FALSE
    decl X in decl X => FALSE
    decl X:Var,I1:Index,I2:Index in if X goto I1 else I2 => EQ(Z,X)
    decl C:Const,I1:Index,I2:Index in if C goto I1 else I2 => FALSE
    decl X:Var,C:Const in X := C => FALSE
    decl X:Var,Y:Var in X := Y => EQ(Z,Y)
    decl X:Var,Y:Var in *X := Y => OR(EQ(Z,Y), EQ(Z,X))
    decl X:Var,Y:Var in X := *Y => EQ(Z,Y)
    decl X:Var,Y:Var in X := &Y => EQ(Z,Y)
    decl X:Var in X := new => FALSE
    decl X:Var,A:Var,B:Var in X := +(A,B) => OR(EQ(Z,A),EQ(Z,B))
    decl X:Var,A:Var,C:Const in X := +(A,C) => EQ(Z,A)
    decl X:Var,C:Const,B:Var in X := +(C,B) => EQ(Z,B)
    decl X:Var,C1:Const,C2:Const in X := +(C1,C2) => FALSE
    decl X:Var,Y:Var,P:Fun in X := P(Y) => EQ(Z,Y)
    decl X:Var in return X => EQ(Z,X)
  ENDCASE
-----
--
-- syntacticUnchanged label

```

```

--
-----
define syntacticUnchanged(E:NonCallExpr) =
CASE E OF
  decl Z:Var in          Z          => NOT(syntacticDef(Z))
  decl Z:Var in          &Z        => NOT(syntacticDef(Z))
  decl C:Const in       C          => TRUE
  decl C1:Const,C2:Const in +(C1,C2) => TRUE
  decl X:Var,Y:Var in   +(X,Y)    => AND(NOT(syntacticDef(X)),NOT(syntacticDef(Y)))
  decl X:Var,C:Const in +(X,C)    => NOT(syntacticDef(X))
  decl X:Var,C:Const in +(C,X)    => NOT(syntacticDef(X))
  decl Z:Var in         *Z        => NOT(syntacticDef(Z))
ENDCASE

```

```

--
-----
-- npMayDef label: conservative version of mayDef that does not
-- use pointer information.
--
-----

```

```

define npMayDef(Z:Var) =
CASE currStmt OF
  skip                    => syntacticDef(Z)
  decl X:Var in          decl X                    => syntacticDef(Z)
  decl X:Var,I1:Index,I2:Index in if X goto I1 else I2 => syntacticDef(Z)
  decl C:Const,I1:Index,I2:Index in if C goto I1 else I2 => syntacticDef(Z)
  decl X:Var,C:Const in   X := C                    => syntacticDef(Z)
  decl X:Var,Y:Var in     X := Y                    => syntacticDef(Z)
  decl X:Var,Y:Var in     *X := Y                  => TRUE
  decl X:Var,Y:Var in     X := *Y                  => syntacticDef(Z)
  decl X:Var,Y:Var in     X := &Y                  => syntacticDef(Z)
  decl X:Var in           X := new                  => syntacticDef(Z)
  decl X:Var,A:Var,B:Var in X := +(A,B)             => syntacticDef(Z)
  decl X:Var,A:Var,C:Const in X := +(A,C)           => syntacticDef(Z)
  decl X:Var,C:Const,B:Var in X := +(C,B)           => syntacticDef(Z)
  decl X:Var,C1:Const,C2:Const in X := +(C1,C2)     => syntacticDef(Z)
  decl X:Var,Y:Var,P:Fun in   X := P(Y)             => TRUE
  decl X:Var in             return X                 => syntacticDef(Z)
ENDCASE

```

```

--
-----
-- npMayUse: conservative version of mayUse that does not use
-- pointer information.
--
-----

```

```

define npMayUse(Z:Var) =
CASE currStmt OF
  skip                    => syntacticUse(Z)
  decl X:Var in          decl X                    => syntacticUse(Z)
  decl X:Var,I1:Index,I2:Index in if X goto I1 else I2 => syntacticUse(Z)
  decl C:Const,I1:Index,I2:Index in if C goto I1 else I2 => syntacticUse(Z)
  decl X:Var,C:Const in   X := C                    => syntacticUse(Z)
  decl X:Var,Y:Var in     X := Y                    => syntacticUse(Z)
  decl X:Var,Y:Var in     *X := Y                  => syntacticUse(Z)
  decl X:Var,Y:Var in     X := *Y                  => TRUE

```

```

    decl X:Var,Y:Var in          X := &Y          => syntacticUse(Z)
    decl X:Var in                X := new          => syntacticUse(Z)
    decl X:Var,A:Var,B:Var in    X := +(A,B)      => syntacticUse(Z)
    decl X:Var,A:Var,C:Const in  X := +(A,C)    => syntacticUse(Z)
    decl X:Var,C:Const,B:Var in  X := +(C,B)    => syntacticUse(Z)
    decl X:Var,C1:Const,C2:Const in X := +(C1,C2) => syntacticUse(Z)
    decl X:Var,Y:Var,P:Fun in    X := P(Y)       => TRUE
    decl X:Var in                return X         => syntacticUse(Z)
  ENDCASE

-----

--
-- npUnchanged label: conservative version of unchanged that does not
-- use pointer information.
--
-----

define npUnchanged(E:NonCallExpr) =
  CASE E OF
    decl Z:Var in                Z          => NOT(npMayDef(Z))
    decl Z:Var in                &Z        => NOT(npMayDef(Z))
    decl C:Const in              C          => TRUE
    decl C1:Const,C2:Const in    +(C1,C2) => TRUE
    decl X:Var,Y:Var in          +(X,Y)    => AND(NOT(npMayDef(X)),NOT(npMayDef(Y)))
    decl X:Var,C:Const in        +(X,C)    => NOT(npMayDef(X))
    decl X:Var,C:Const in        +(C,X)    => NOT(npMayDef(X))
    decl Z:Var in                *Z        => FALSE
  ENDCASE

-----

--
-- mayDef label
--
-----

define mayDef(Z:Var) =
  CASE currStmt OF
    skip                          => syntacticDef(Z)
    decl X                          => syntacticDef(Z)
    decl X:Var,I1:Index,I2:Index in if X goto I1 else I2 => syntacticDef(Z)
    decl C:Const,I1:Index,I2:Index in if C goto I1 else I2 => syntacticDef(Z)
    decl X:Var,C:Const in          X := C          => syntacticDef(Z)
    decl X:Var,Y:Var in            X := Y          => syntacticDef(Z)
    decl X:Var,Y:Var in            *X := Y        => OR(syntacticDef(Z),mayPointTo(X,Z))
    decl X:Var,Y:Var in            X := *Y        => syntacticDef(Z)
    decl X:Var,Y:Var in            X := &Y        => syntacticDef(Z)
    decl X:Var in                  X := new        => syntacticDef(Z)
    decl X:Var,A:Var,B:Var in      X := +(A,B)    => syntacticDef(Z)
    decl X:Var,A:Var,C:Const in    X := +(A,C)    => syntacticDef(Z)
    decl X:Var,C:Const,B:Var in    X := +(C,B)    => syntacticDef(Z)
    decl X:Var,C1:Const,C2:Const in X := +(C1,C2) => syntacticDef(Z)
    decl X:Var,Y:Var,P:Fun in      X := P(Y)     => OR(syntacticDef(Z),NOT(isNotTainted(Z)))
    decl X:Var in                  return X       => syntacticDef(Z)
  ENDCASE

-----

--
-- mayUse label
--
-----

```

```

define mayUse(Z:Var) =
  CASE currStmt OF
    skip => syntacticUse(Z)
    decl X in decl X => syntacticUse(Z)
    decl X:Var,I1:Index,I2:Index in if X goto I1 else I2 => syntacticUse(Z)
    decl C:Const,I1:Index,I2:Index in if C goto I1 else I2 => syntacticUse(Z)
    decl X:Var,C:Const in X := C => syntacticUse(Z)
    decl X:Var,Y:Var in X := Y => syntacticUse(Z)
    decl X:Var,Y:Var in *X := Y => syntacticUse(Z)
    decl X:Var,Y:Var in X := *Y => OR(syntacticUse(Z),mayPointTo(Y,Z))
    decl X:Var,Y:Var in X := &Y => syntacticUse(Z)
    decl X:Var in X := new => syntacticUse(Z)
    decl X:Var,A:Var,B:Var in X := +(A,B) => syntacticUse(Z)
    decl X:Var,A:Var,C:Const in X := +(A,C) => syntacticUse(Z)
    decl X:Var,C:Const,B:Var in X := +(C,B) => syntacticUse(Z)
    decl X:Var,C1:Const,C2:Const in X := +(C1,C2) => syntacticUse(Z)
    decl X:Var,Y:Var,P:Fun in X := P(Y) => TRUE
    decl X:Var in return X => syntacticUse(Z)
  ENDCASE

```

```

-----
--
-- unchanged label
--
-----

```

```

define unchanged(E:NonCallExpr) =
  CASE E OF
    decl Z:Var in Z => NOT(mayDef(Z))
    decl Z:Var in &Z => NOT(mayDef(Z))
    decl C:Const in C => TRUE
    decl C1:Const,C2:Const in +(C1,C2) => TRUE
    decl X:Var,Y:Var in +(X,Y) => AND(NOT(mayDef(X)),NOT(mayDef(Y)))
    decl X:Var,C:Const in +(X,C) => NOT(mayDef(X))
    decl X:Var,C:Const in +(C,X) => NOT(mayDef(X))
    decl Z:Var in *Z =>
      CASE currStmt OF
        skip => NOT(mayDef(Z))
        decl X in decl X => NOT(mayDef(Z))
        decl X:Var,I1:Index,I2:Index in if X goto I1 else I2 => NOT(mayDef(Z))
        decl C:Const,I1:Index,I2:Index in if C goto I1 else I2 => NOT(mayDef(Z))
        decl X:Var,C:Const in X := C => AND(NOT(mayDef(Z)),doesNotPointTo(Z,X))
        decl X:Var,Y:Var in X := Y => AND(NOT(mayDef(Z)),doesNotPointTo(Z,X))
        decl X:Var,Y:Var in *X := Y => FALSE
        decl X:Var,Y:Var in X := *Y => AND(NOT(mayDef(Z)),doesNotPointTo(Z,X))
        decl X:Var,Y:Var in X := &Y => AND(NOT(mayDef(Z)),doesNotPointTo(Z,X))
        decl X:Var in X := new => AND(NOT(mayDef(Z)),doesNotPointTo(Z,X))
        decl X:Var,A:Var,B:Var in X := +(A,B) => AND(NOT(mayDef(Z)),doesNotPointTo(Z,X))
        decl X:Var,A:Var,C:Const in X := +(A,C) => AND(NOT(mayDef(Z)),doesNotPointTo(Z,X))
        decl X:Var,C:Const,B:Var in X := +(C,B) => AND(NOT(mayDef(Z)),doesNotPointTo(Z,X))
        decl X:Var,C1:Const,C2:Const in X := +(C1,C2) => AND(NOT(mayDef(Z)),doesNotPointTo(Z,X))
        decl X:Var,Y:Var,P:Fun in X := P(Y) => FALSE
        decl X:Var in return X => NOT(mayDef(Z))
      ENDCASE
  ENDCASE

```

```

-----
--
-- fnCall label

```

```

--
-----
define fnCall() =
  CASE currStmt OF
    skip => FALSE
    decl X:Var in decl X => FALSE
    decl X:Var,I1:Index,I2:Index in if X goto I1 else I2 => FALSE
    decl C:Const,I1:Index,I2:Index in if C goto I1 else I2 => FALSE
    decl X:Var,C:Const in X := C => FALSE
    decl X:Var,Y:Var in X := Y => FALSE
    decl X:Var,Y:Var in *X := Y => FALSE
    decl X:Var,Y:Var in X := *Y => FALSE
    decl X:Var,Y:Var in X := &Y => FALSE
    decl X:Var in X := new => FALSE
    decl X:Var,A:Var,B:Var in X := +(A,B) => FALSE
    decl X:Var,A:Var,C:Const in X := +(A,C) => FALSE
    decl X:Var,C:Const,B:Var in X := +(C,B) => FALSE
    decl X:Var,C1:Const,C2:Const in X := +(C1,C2) => FALSE
    decl X:Var,Y:Var,P:Fun in X := P(Y) => TRUE
    decl X:Var in return X => FALSE
  ENDCASE

```

```

--
-- addrTaken label
--
-----

```

```

define addrTaken(Z:Var) =
  CASE currStmt OF
    skip => FALSE
    decl X:Var in decl X => FALSE
    decl X:Var,I1:Index,I2:Index in if X goto I1 else I2 => FALSE
    decl C:Const,I1:Index,I2:Index in if C goto I1 else I2 => FALSE
    decl X:Var,C:Const in X := C => FALSE
    decl X:Var,Y:Var in X := Y => FALSE
    decl X:Var,Y:Var in *X := Y => FALSE
    decl X:Var,Y:Var in X := *Y => FALSE
    decl X:Var,Y:Var in X := &Y => EQ(Z,Y)
    decl X:Var in X := new => FALSE
    decl X:Var,A:Var,B:Var in X := +(A,B) => FALSE
    decl X:Var,A:Var,C:Const in X := +(A,C) => FALSE
    decl X:Var,C:Const,B:Var in X := +(C,B) => FALSE
    decl X:Var,C1:Const,C2:Const in X := +(C1,C2) => FALSE
    decl X:Var,Y:Var,P:Fun in X := P(Y) => FALSE
    decl X:Var in return X => FALSE
  ENDCASE

```

```

--
-- Pointer information
--
-- We define two simple points-to analyses (simpleNotPntTo and
-- isNotTainted), and then combine them together to get the
-- doesNotPointTo and mayPointTo labels. The simpleNotPntTo analysis
-- uses npMayDef, which is a version of mayDef that does not use
-- pointer information (since we can't assume pointer information
-- while we're computing it).
--

```

```

-----

analysis isNotTainted(X:Var) =
  stmt(decl X)
  followed_by
    NOT(addrTaken(X))
  defines
    isNotTainted(X:Var)
  with_witness
    -- This witness says that for any location L, the result of
    -- evaluating *L (using evalLocDeref, which is a primitive, much
    -- like evalExpr) is not equal to &X. In other words, no store
    -- location points to X.
    FORALL L . IMPLIES(isLoc(L),
      NEQ(evalLocDeref(eta,L),
        evalExpr(eta,&X)))

analysis simpleNotPntTo(X:Var,Y:Var) =
  decl Z:Var in
    AND(stmt(X := &Z),hasBeenDeclared(Y),NOT(syntacticUse(Y)))
  followed_by
    NOT(npMayDef(X))
  defines
    simpleNotPntTo(X:Var,Y:Var)
  with_witness
    NEQ(evalExpr(eta,X),evalExpr(eta,&Y))

analysis hasBeenDeclared(X:Var) =
  stmt(decl X)
  followed_by
    TRUE
  defines
    hasBeenDeclared(X:Var)
  with_witness
    -- This witness says that X is not stuck in the machine state
    -- eta, which basically means that X is in the domain of the
    -- environment component of eta.
    isExprNotStuck(eta,X)

define doesNotPointTo(X:Var,Y:Var) = OR(simpleNotPntTo(X,Y),
  isNotTainted(Y))
define mayPointTo(X:Var,Y:Var) = NOT(doesNotPointTo(X,Y))

```