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

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 for implementing optimizations as guarded rewrite rules. 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 optimizations implemented in this language. Our technique requires only a small set of optimization-specific proof obligations to be discharged for each optimization by an automatic theorem prover. We have written a variety of forward and backward intraprocedural dataflow optimizations in our language, including constant propagation and folding, branch folding, (partial) redundancy elimination, (partial) dead assignment elimination, and simple forms of points-to analysis. We have implemented our proof strategy with the Simplify automatic theorem prover, and we have used this implementation to automatically prove our optimizations correct. Our system also found many subtle bugs during the course of developing our optimizations.

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 and other 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 input 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.

Credible compilers [25, 24] and translation validation [18] improve on this testing approach by having the compiler automatically check whether or not the optimized version of an input program is semantically equivalent to the original program. The compiler 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 the compiler reports that it cannot validate the programmer's compiled program. Furthermore, these approaches can have a substantial impact on the time to run an optimization. For example, Necula mentions that translation validation of an optimization pass takes about four times longer than the optimization pass itself [18].

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, 14, 12, 6, 22, 3, 9]. 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 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. Our language 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 our language 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 our language to express a variety of intraprocedural forward and backward dataflow optimizations, including constant propagation and folding, copy propagation, common subexpression elimination, dead assignment elimination, branch folding, partial redundancy elimination, partial dead code elimination, and loop-invariant code motion. We have also used our language to express several simple intraprocedural pointer analyses, whose results we have exploited in the above optimizations.
- We present a strategy for automatically proving the soundness of optimizations and analyses expressed in our language. 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 non-inductive theorems to prove about individual program states.
- We have implemented our correctness checking strategy using Simplify [26, 21], the automatic theorem prover used in the Extended Static Checker for Java [5]. 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.

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 our language for expressing optimizations by example and sketches our strategy for automatically proving soundness of such optimizations. Sections 3 and 4 formally define our optimization language and automatic proof strategy, respectively. Section 5 evaluates our work. Section 6 discusses our current and future work, including an extension to support interprocedural optimizations. Section 7 discusses related work, and section 8 offers our conclusions. The optional appendices contain definitions of all the optimizations and analyses we have written in our language.

2. OVERVIEW

In this section, we informally describe our language for defining optimizations and our technique for proving those optimizations sound.

2.1 Forward Transformation Patterns

2.1.1 Semantics

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

$$\psi_1 \textbf{\textit{ followed by }} \psi_2 \textbf{\textit{ until }} s \Rightarrow s' \textbf{\textit{ with witness }} \mathcal{P}$$

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.

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 no intervening (innocuous) statement modifies 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. 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. This sequence of events is expressed by the following transformation pattern (the witness is discussed in more detail in section 2.1.2):*

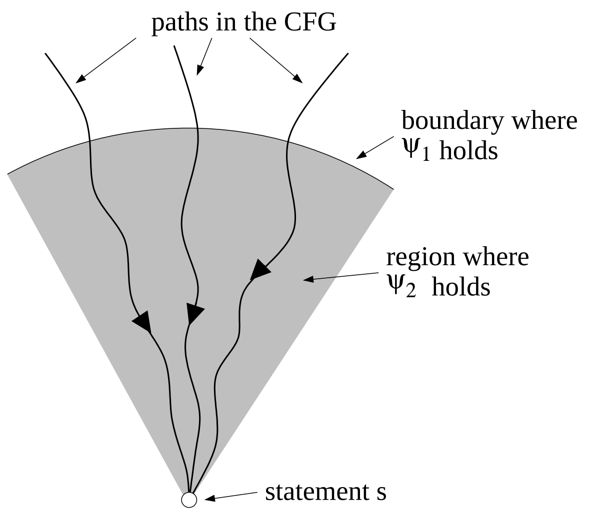

Figure 1: CFG paths leading to a statement s which can be transformed to s' by the transformation pattern ψ_1 ***followed by*** ψ_2 ***until*** $s \Rightarrow s'$ ***with witness*** $\mathcal{P}$. The shaded region can only be entered through a statement satisfying ψ_1, and all statements within the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

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

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'. Then any execution trace of the procedure that contains s' will at some point execute an enabling statement, then 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 (i.e., statement) in a procedure's CFG is *labeled* with properties that are true at that node, such as $mayDef(\mathtt{y})$ or $stmt(\mathtt{x := 5})$. The formulas ψ_1 and ψ_2 in an optimization are propositional boolean expressions over these labels, which may reference pattern variables like Y and C. Our framework provides a single pre-defined label, $stmt(s)$, which is true at a node if and only if that node is the statement s. Users can then define their own labels in two ways: *syntactic* labels and *semantic* labels.

A syntactic label is defined only in terms of the *stmt* label and other syntactic labels of the current statement. For example, $synDef(Y)$, which stands for syntactic definition of Y, can be defined as:

$$synDef(Y) \triangleq stmt(Y := \ldots)$$

Then the syntactic (and conservative) version of the $mayDef(Y)$ label from example 1 can be defined as:

$$mayDef(Y) \triangleq synDef(Y) \vee stmt(*X := \ldots) \vee stmt(\ldots := P(\ldots))$$

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

In contrast to syntactic labels, a semantic label of a node can incorporate information about the node's surrounding context in the CFG. For example, a $doesNotPointTo(X, Y)$ label, which says that the contents of X is definitely not the address of Y, is a semantic label: its truth value at a node depends on the execution paths leading to the node. Section 2.4 shows how semantic labels are defined and how they can be used to make the *mayDef* label 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 ***until*** $s \Rightarrow s'$ ***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

[1] The 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.

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 our language by the following backward transformation pattern:*

$(\mathit{synDef}(X) \vee \mathit{stmt}(\texttt{return}\ldots)) \wedge \neg \mathit{mayUse}(X)$
preceded by
$\neg \mathit{mayUse}(X)$
until
$X := E \;\Rightarrow\; \texttt{skip}$
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, indicated by $\mathit{synDef}(X)$, 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 re-definition of X or a `return` *statement causes the execution states of the two traces to become identical.*

2.3 Profitability Heuristics

If an optimization's transformation pattern is proven sound, then all matching occurrences of that pattern are legal to be transformed. 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*.

[2]An execution engine would not actually insert such `skip`s.

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 the procedure being optimized, *choose* returns the subset of the transformations in Δ that should actually be performed. A complete optimization in our language therefore has the following form, where O_{pat} is a transformation pattern:

O_{pat} ***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: $\mathit{choose}_{all}(\Delta, p) = \Delta$. This profitability heuristic is the default if none is specified explicitly. Below we give an example of an optimization with a non-trivial *choose* function:

Example 3. *Consider the implementation of partial redundancy elimination (PRE) [13, 8] in our optimization language. 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 our language. 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 our language as the following backward optimization:

$\mathit{stmt}(X := E) \wedge \mathit{unchanged}(E)$

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

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 code duplication. A sample choose function for PRE is shown in the appendix.*

2.4 Pure Analyses

In addition to optimizations, our language 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 semantic label, and the result of the analysis is a labeling of the given CFG. For instance, the does-not-point-to analysis (a definition of which is shown in the appendix) results in nodes of the CFG being annotated with labels of the form *doesNotPointTo*(X, Y). These labels can then be used by other optimizations in their guards.

A pure analysis is similar to a forward optimization, except that it does not contain a rewrite rule or a profitability heuristic.[4] Instead, it has a ***defines*** clause that gives a name to the new semantic label. A 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 semantic 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 analysis defines the notTainted label:*

$\quad$ *stmt*(`decl` X)
followed by
$\quad$ $\neg$*stmt*$(\ldots := \&X)$
defines
$\quad$ *notTainted*(X)
with witness
$\quad$ *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 notPointedTo(X, η) is a first-order predicate defined by the user (and shown in the appendix) that ensures that 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 neither pointer stores nor procedure calls can affect variables that are untainted:

$$\begin{aligned} \mathit{mayDef}(Y) \triangleq\ & \mathit{synDef}(Y) \vee \\ & (\mathit{stmt}(*X := \ldots) \wedge \neg\mathit{notTainted}(Y)) \vee \\ & (\mathit{stmt}(\ldots := P(\ldots)) \wedge \neg\mathit{notTainted}(Y)) \end{aligned}$$

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

3. LANGUAGE FOR DEFINING OPTIMIZATIONS

This section provides a formal definition of our optimization language and the intermediate language that optimizations manipulate. The full formal details can be found in our technical report [11].

3.1 Intermediate Language

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

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

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 each procedure ends with a `return` statement. Statements are indexed consecutively from 0, and *stmtAt*(π, ι) returns the statement with index ι in π. Expressions include constants,

[3] An execution engine for optimizations would conceptually insert `skip`s dynamically as needed to perform insertions.
[4] Our language currently has no notion of backward analyses. In addition, we currently only allow the results of a forward analysis to be used in a forward optimization, or in another forward analysis.

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* $\rightarrow_\pi$. 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 $\rightarrow_\pi$ is standard and is given in our accompanying technical report [11]. We also define an *intraprocedural state transition function* $\hookrightarrow_\pi$. This function acts like $\rightarrow_\pi$ except when the statement to be executed is a procedure call. In that case, $\hookrightarrow_\pi$ 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 Optimization Language

In this section, we first specify the syntax of a rewrite rule's original and transformed statements s and s'. Then we define the language 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:

$$\begin{array}{llll} \textit{Exprs} & e & ::= & \cdots \mid E \\ \textit{Vars} & x & ::= & \cdots \mid X \mid Y \mid Z \mid \ldots \\ \textit{Consts} & c & ::= & \cdots \mid C \end{array}$$

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 is described by the following grammar:

$$\psi \quad ::= \quad \textit{true} \mid \textit{false} \mid pr \mid \neg\psi \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2$$

In this grammar, pr stands for atomic predicates, each of which is formed from the pre-defined *stmt* label or from any user-defined label, with parameters drawn from the *extended* intermediate language. For example, pr includes predicates such as $\textit{mayDef}(X)$ and $\textit{unchanged}(E)$, where X and E are pattern variables.

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 atomic predicates pr that do not contain pattern variables. For example, a node could be labeled with $\textit{stmt}(\mathtt{x} := 3)$ and $\textit{mayDef}(\mathtt{x})$.

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 $\iota \models^p_\theta \psi$ to indicate that the node with index ι satisfies ψ in the labeled CFG of p under substitution θ. The definition of $\iota \models^p_\theta \psi$ is straightforward, with the base case being $\iota \models^p_\theta pr \iff \theta(pr) \in L_p(\iota)$. The complete definition of $\models^p_\theta$ is in our technical report [11].

3.2.3 Semantics of Optimizations

We define the semantics of analyses and optimizations 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:*

$$\begin{array}{l} [\![O_{guard}]\!](p) = \{(\iota, \theta) \mid \\ \quad \textit{for all paths } \iota_1, \ldots, \iota_j, \iota \textit{ in } p\textit{'s CFG} \\ \quad \textit{such that } \iota_1 \textit{ is the index of } p\textit{'s entry node} \\ \qquad \exists k.(1 \le k \le j \wedge \iota_k \models^p_\theta \psi_1 \wedge \forall i.(k < i \le j \Rightarrow \iota_i \models^p_\theta \psi_2))\} \end{array}$$

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 $\iota, \iota_j, \ldots, \iota_1$ 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 LTL and CTL.

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

$$[\![O_{pat}]\!](p) = \{(\iota, \theta) \mid (\iota, \theta) \in [\![O_{guard}]\!](p) \text{ and } n \models^p_\theta \textit{stmt}(s)\}$$

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

$$\begin{array}{l} [\![O]\!](p) = \textbf{\textit{let}}\ \Delta := [\![O_{pat}]\!](p)\ \textbf{\textit{in}} \\ \qquad\qquad\qquad app(s', p, \textit{choose}(\Delta, p) \cap \Delta) \end{array}$$

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

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 $[\![O_{guard}]\!](p)$, the node with index ι is additionally labeled with $\theta(\textit{label})$.

4. PROVING SOUNDNESS AUTOMATICALLY

In this section we describe in detail our technique for automatically proving soundness of optimizations. We say that an intermediate-language program π' is a *semantically equivalent transformation* of π if, whenever $\mathtt{main}(v_1)$ 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, $\pi[p \mapsto [\![O]\!](p)]$ is a semantically equivalent transformation of π.

The first subsection describes our technique for proving optimizations sound, which requires an automatic theorem prover to discharge only a small set of simple proof obligations. The second subsection describes how these obligations are implemented and automatically discharged with the Simplify theorem prover.

4.1 Soundness of Optimizations

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 $\Delta \subseteq [\![O_{pat}]\!](p)$, $\pi[p \mapsto app(s', p, \Delta)]$ 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 of these is known to be 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.

4.1.1 Forward Transformation Patterns

Consider a forward transformation pattern of the following form:

$$\psi_1 \textit{ followed by } \psi_2 \textit{ until } s \Rightarrow s' \textit{ with witness } \mathcal{P}$$

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 in this context. 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 non-inductive obligations, which pertain to individual execution states rather than entire execution traces. We have proven (see Theorems 1 and 2 below) 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 $index(\eta) \models^p_\theta \psi_1$, then $\theta(\mathcal{P})(\eta')$.

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

F3. If $\theta(\mathcal{P})(\eta)$ and $\eta \hookrightarrow_\pi \eta'$ and $\iota = index(\eta)$ and $stmtAt(\pi, \iota) = \theta(s)$ and $stmtAt(\pi', \iota) = \theta(s')$, 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 $index(\eta) \models^p_\theta stmt(Y := C)$, 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, which is given in our technical report [11], 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. Our proof also handles the case when multiple transformations, with possibly overlapping witnessing regions, are performed.

4.1.2 Backward Transformation Patterns

Consider a backward transformation pattern of the following form:

$$\psi_1 \textit{ preceded by } \psi_2 \textit{ until } s \Rightarrow s' \textit{ with witness } \mathcal{P}$$

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 $\iota = index(\eta)$ 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 $\iota_{old} \models^\pi_\theta \psi_2$ 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 $\iota_{old} \models^\pi_\theta \psi_1$ and $stmtAt(\pi, \iota_{old}) = stmtAt(\pi', \iota_{new})$, 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.[5] Condition B2 ensures that the backward witness is preserved through the innocuous statements. Condition B3 ensures that the two traces become

[5]This 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 our technical report [11].

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.*

4.2 Implementation with Simplify

We have implemented our proof strategy with the Simplify automatic theorem prover. For each optimization, we ask Simplify to prove the three associated optimization-specific obligations. To do so, Simplify requires background information in the form of a set of axioms. These axioms, which simply encode the semantics of our intermediate language and of the *stmt* label, are optimization-independent: they need not be modified in order to prove new optimizations sound.

We introduce function symbols to 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 *evalLExpr* which computes the location of a *lhs* expression given a program state. Finally, we provide axioms for the *stepIndex*, *stepEnv*, *stepStore*, *stepStack*, and *stepMem* functions, which together define the state transition function $\rightarrow_\pi$ 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:

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

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

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. The $\hookrightarrow_\pi$ function is then defined in terms of the $\rightarrow_\pi$ function.

We have implemented and automatically proven sound a dozen optimizations and analyses in our language (which are given in the appendix). On a modern workstation, the time taken by Simplify to discharge the optimization-specific obligations for these optimizations ranges from 2 to 89 seconds, with an average of 22 seconds.

5. DISCUSSION

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

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, the restrictions of our optimization language are not as onerous as they may first appear. First, much of the complexity of an optimization can be factored out 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 and partial redundancy elimination, can in fact be decomposed into several simpler transformations, each of which fits our constraint of making a single transformation at the end of a witnessing region. The PRE example in section 2.3 illustrates all three of these points. PRE is a complex code-motion optimization [13, 8], and yet it can be expressed in our language 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, our current language does have limitations. For example, it cannot express interprocedural optimizations or one-to-many transformations. As mentioned in section 6, our ongoing work is addressing 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 our language to be rejected. In these cases, optimizations can be written outside of our framework, perhaps verified using translation validation. Optimizations written in our optimization language 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 $Y := \ldots$ 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 automatic theorem prover, the manual proofs done as part of our framework, and the run-time 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 6, the run-time engine can be implemented as a single dataflow analysis common to all user-defined optimizations. This means that the trustworthiness of the run-time 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 [20]. 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 [11] to increase confidence. We could also use an interactive theorem prover such as PVS [23] to validate these proofs.

6. CURRENT AND FUTURE WORK

Our current work is focused in two directions. First, we are implementing the dynamic semantics of our optimization language as an analysis in the Whirlwind compiler, a successor to Vortex [4]. This analysis stores at every program point a set of substitutions, each substitution representing a potential witnessing region. Consider a forward optimization:

$$\begin{array}{c}\psi_1 \textbf{\textit{ followed by }} \psi_2 \textbf{\textit{ until }} s \;\Rightarrow\; s' \\ \textbf{\textit{with witness }} \mathcal{P} \textbf{\textit{ filtered through }} \mathit{choose}\end{array}$$

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, 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 $\theta(\mathit{stmt}(s))$ true, and the *choose* function selects this statement, then the statement is transformed to $\theta(s')$.

For example, in constant propagation we have $\psi_1 = \mathit{stmt}(Y := C)$ and $\psi_2 = \neg\mathit{mayDef}(Y)$. The following program fragment shows the dataflow facts propagated after each statement:

$$\begin{array}{ll}\mathtt{S1} : \mathtt{a} := 2; & [Y \mapsto \mathtt{a}, C \mapsto 2] \\ \mathtt{S2} : \mathtt{b} := 3; & [Y \mapsto \mathtt{a}, C \mapsto 2], [Y \mapsto \mathtt{b}, C \mapsto 3] \\ \mathtt{S3} : \mathtt{c} := \mathtt{a}; & \end{array}$$

`S1` satisfies ψ_1, and so its outgoing dataflow fact contains the substitution $[Y \mapsto \mathtt{a}, C \mapsto 2]$. `S2` satisfies ψ_2 under this substitution, and so the substitution is propagated; `S2` also satisfies ψ_1 and so $[Y \mapsto \mathtt{b}, C \mapsto 3]$ is added to the outgoing dataflow fact. In fact, the dataflow information after `S2` is very similar to the regular constant propagation dataflow fact $\{\mathtt{a} \mapsto 2, \mathtt{b} \mapsto 3\}$. At fixed point, the statement $\mathtt{c} := \mathtt{a}$ can be transformed to $\mathtt{c} := 2$ because the incoming dataflow fact contains the map $[Y \mapsto \mathtt{a}, C \mapsto 2]$. Note that this implementation evaluates all "instances" of the constant propagation optimization pattern simultaneously. (We also plan to explore potentially more efficient implementation techniques, such as generating specialized code to run each optimization [27].)

Our analysis is being implemented using our earlier framework for composing optimizations in Whirlwind [10]. 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 our language will therefore also be 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 will likewise be conferred on optimizations written in our language. For example, a recursive version of dead-assignment elimination would allow $X := E$ to be removed even if X is used before being re-defined, as long as it is only used by other dead assignments (including itself).

The other direction of our current work is in extending the language to handle interprocedural optimizations. One approach would extend the scope of analysis from a single procedure to the whole program's control-flow supergraph. One technical challenge here is the need to express the witness $\mathcal{P}$ in a way that makes sense 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 variable 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.

There are also many directions for future work. First, the optimization language currently supports only 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 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.

Finally, an important consideration that we have not addressed is the interface between the optimization writer and our automatic soundness prover. It will be critical to provide useful error messages when an optimization is unable to 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.

7. RELATED WORK

Our work is inspired by that of Lacey et al. [9]. 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 semantic labels 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 that we support may not be expressible by Lacey.

As mentioned in the introduction, much other work has been done on manually proving optimizations correct [12, 14, 1, 2, 6, 22, 3]. Transformations have also been proven correct mechanically, but not automatically: the transformation is proven sound using an interactive theorem prover, which increases one's confidence but requires user interaction. For example, Young [29] has proven a code generator correct using the Boyer-Moore theorem prover enhanced with an interactive interface [7].

Instead of proving that the compiler is always correct, credible compilation [25, 24] and translation validation [18] both attack the problem of checking the correctness of a given compilation run. Therefore, a bug in an optimization will only appear 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, while credible compilation and translation validation typically do not.

Proof-carrying code [17], certified compilation [19], typed intermediate languages [28], and typed assembly languages [15, 16] have all been used to prove properties of programs generated by a compiler. However, the kind 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.

8. 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 for writing optimizations that is both reasonably expressive and amenable to automated correctness reasoning. Using our technique we have proven correct our 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.

9. 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, 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, 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*, 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] 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.

[6] J. Guttman, J. Ramsdell, and M. Wand. VLISP: a verified implementation of Scheme. *Lisp and Symbolic Compucation*, 8(1-2):33–110, 1995.

[7] 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.

[8] 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.

[9] 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*, January 2002.

[10] 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*, January 2002.

[11] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. Technical Report UW-CSE-02-11-02, University of Washington, November 2002. `http://www.cs.washington.edu/homes/lerns/UW-CSE-02-11-02.ps`.

[12] 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.

[13] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[14] 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*, January 1973.

[15] 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, USA, May 1999.

[16] 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.

[17] George C. Necula. Proof-carrying code. In *Conference Record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1997.

[18] 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.

[19] 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*, June 1998.

[20] 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.

[21] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[22] D. P. Oliva, J. Ramsdell, and M. Wand. The VLISP verified PreScheme compiler. *Lisp and Symbolic Computation*, 8(1-2):111–182, 1995.

[23] 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.

[24] Martin Rinard. Credible compilation. Technical Report MIT-LCS-TR-776, Massachusetts Institute of Technology, March 1999.

[25] Martin Rinard and Darko Marinov. Credible compilation. In *Proceedings of the FLoC Workshop Run-Time Result Verification*, July 1999.

[26] Simplify automatic theorem prover home page, `http://research.compaq.com/SRC/esc/Simplify.html`.

[27] Bernhard Steffen. Data flow analysis as model checking. In A.R. Meyer T. Ito, editor, *Theoretical Aspects of Computer Science (TACS'91), Sendai (Japan)*, volume 526 of *Lecture Notes in Computer Science (LNCS)*, pages 346–364, Heidelberg, Germany, September 1991. Springer-Verlag.

[28] 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*, May 1996.

[29] William D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5(4):493–518, December 1989.

APPENDIX

A. ADDITIONAL OPTIMIZATIONS

This optional appendix shows additional examples of optimizations and analyses defined in our language, all of which have been proven sound automatically using our strategy. This appendix also contains definitions of all the labels used in the paper.

A.1 Optimizations

Copy propagation

$\quad stmt(Y := Z)$
followed by
$\quad \neg mayDef(Z) \wedge \neg mayDef(Y)$
until
$\quad X := Y \;\Rightarrow\; X := Z$
with witness
$\quad \eta(Y) = \eta(Z)$

Constant propagation

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

$\quad stmt(X := C)$
followed by
$\quad \neg mayDef(X)$
until
$\quad$ `if` X `goto` P_1 `else` $P_2 \;\Rightarrow\;$ `if` C `goto` P_1 `else` P_2
with witness
$\quad \eta(X) = C$

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

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

Constant folding

$\quad true$
followed by
$\quad true$
until
$\quad X := op\ C_1\ C_2 \;\Rightarrow\; X := C \qquad (C = op(C_1, C_2))$
with witness
$\quad true$

The above guard, *true* **followed by** *true*, holds at all nodes.

Branch folding

In the following optimizations, we use `goto` P_1 as sugar for `if` *true* `goto` P_1 `else` P_1.

$\quad true$
followed by
$\quad true$
until
$\quad$ `if` *true* `goto` P_1 `else` $P_2 \;\Rightarrow\;$ `goto` P_1
with witness
$\quad true$

$\quad true$
followed by
$\quad true$
until
$\quad$ `if` C `goto` P_1 `else` $P_2 \;\Rightarrow\;$ `goto` $P_2 \qquad (C \neq true)$
with witness
$\quad true$

Common subexpression elimination

$\quad unchanged(E) \wedge stmt(Z := E)$
followed by
$\quad \neg mayDef(Z) \wedge unchanged(E)$
until
$\quad X := E \;\Rightarrow\; X := Z$
with witness
$\quad \eta(Z) = \eta(E)$

Load removal

$\quad stmt(Y := \&Z)$
followed by
$\quad \neg mayDef(Y) \wedge \neg stmt($`decl` $Z)$
until
$\quad X := *Y \;\Rightarrow\; X := Z$
with witness
$\quad \eta(Y) = \eta(\&Z)$

Dead assignment elimination

$\quad (synDef(X) \vee stmt($`return`$\ldots)) \wedge \neg mayUse(X)$
preceded by
$\quad \neg mayUse(X)$
until
$\quad X := E \;\Rightarrow\;$ `skip`
with witness
$\quad \eta_{old}/X = \eta_{new}/X$

Code hoisting (PRE)

This is the PRE optimization from section 2.3, with pseudocode for a sample *choose* function. The *choose* function shown here is Steffen's optimal computation placement condition [27], which intuitively tries to place duplicates as early as possible. A duplicate is placed at node n if placing the duplicate any earlier would either not be legal, or it would cover different redundancies from those covered by the placement at n. This is just one possible *choose* function, given here for illustrative purposes.

$stmt(X := E) \land unchanged(E)$
preceded by
$unchanged(E) \land \neg mayDef(X) \land \neg mayUse(X)$
until
`skip` $\Rightarrow X := E$
with witness
$\eta_{old}/X = \eta_{new}/X$
filtered through
let $\psi_1 = stmt(X := E) \land unchanged(E)$
let $\psi_2 = unchanged(E) \land \neg mayDef(X) \land \neg mayUse(X)$
let $\psi_3 = \neg unchanged(E) \lor mayUse(X) \lor (mayDef(X) \land \neg stmt(X := E))$
in
$\{(\iota, \theta) \in \Delta|$ forall paths in the CFG leading to ι
there exists a node at which ψ_3 holds
followed by nodes at which $\neg legal$ holds
followed by the node with index $\iota\}$
where *legal* holds at a node with index ι' if
forall paths in the CFG from ι'
there exists a node at which ψ_1 holds
preceded by nodes at which ψ_2 holds
preceded by the node with index ι'

Code sinking (PDE)

We show a version of code sinking that performs partial dead code elimination. We perform code sinking by inserting copies of certain computations downstream of where they occur in the CFG, and then running dead-assignment elimination to remove the original computations. Our code sinking optimization is symmetrical to the code hoisting optimization described above, and the intuition for how it works is the same.

$stmt(X := E) \land unchanged(E)$
followed by
$unchanged(E) \land \neg mayDef(X) \land \neg mayUse(X)$
until
`skip` $\Rightarrow X := E$
with witness
$\eta(X) = \eta(E)$
filtered through
let $\psi_1 = stmt(X := E) \land unchanged(E)$
let $\psi_2 = unchanged(E) \land \neg mayDef(X) \land \neg mayUse(X)$
let $\psi_3 = \neg unchanged(E) \lor mayUse(X) \lor (mayDef(X) \land \neg stmt(X := E))$
in
$\{(\iota, \theta) \in \Delta|$ forall paths in the CFG from ι
there exists a node at which ψ_3 holds
preceded by nodes at which $\neg legal$ holds
preceded by the node with index $\iota\}$
where *legal* holds at a node with index ι' if
forall paths in the CFG leading to ι'
there exists a node at which ψ_1 holds
followed by nodes at which ψ_2 holds
followed by the node with index ι'

A.2 Analyses

Tainted-variable analysis

$stmt($`decl` $X)$
followed by
$\neg stmt(\ldots := \&X)$
defines
$notTainted(X)$
with witness
$notPointedTo(X, \eta)$

where *notPointedTo* is defined as:

$$notPointedTo(X, \eta) \triangleq \forall l \in domain(store(\eta)).\eta(*l) \neq \eta(\&X)$$

Simple points-to analysis

The following is a simple points-to analysis. In section A.3, we show how this label is combined with the *notTainted* label to define the *doesNotPointTo* label, which in turn is used to define labels such as *mayDef*, *mayUse* and *unchanged*. This optimization uses *npMayDef*, which is a version of *mayDef* that does not use pointer information.

$stmt(X := \&Z) \land \neg synUse(Y) \land hasBeenDeclared(Y)$
followed by
$\neg npMayDef(X)$
defines
$simpleNotPntTo(X, Y)$
with witness
$\eta(X) \neq \eta(\&Y)$

Has-been-declared analysis

$stmt($`decl` $X)$
followed by
$true$
defines
$hasBeenDeclared(X)$
with witness
$X \in domain(env(\eta))$

A.3 Label definitions

We now define the labels used in this paper. Labels that start with *np* (e.g. *npMayDef*, *npMayUse*, *npUnchanged*) are versions of the labels that don't use pointer information (and thus are more conservative).

$$
\begin{array}{lcl}
\mathit{synDef}(Y) & \triangleq & \mathit{stmt}(Y := \ldots) \\
\mathit{synUse}(Y) & \triangleq & \mathit{stmt}(\ldots := \ldots Y \ldots) \lor \\
 & & \mathit{stmt}(*Y := \ldots) \lor \\
 & & \mathit{stmt}(\texttt{if}\ Y\ \ldots) \\
\mathit{npMayDef}(Y) & \triangleq & \mathit{synDef}(Y) \lor \\
 & & \mathit{stmt}(*X := \ldots) \lor \\
 & & \mathit{stmt}(\ldots := P(\ldots)) \\
\mathit{npMayUse}(Y) & \triangleq & \mathit{synUse}(Y) \lor \\
 & & \mathit{stmt}(\ldots := *X) \lor \\
 & & \mathit{stmt}(\ldots := P(\ldots)) \\
\mathit{doesNotPointTo}(X,Y) & \triangleq & \mathit{simpleNotPntTo}(X,Y) \lor \\
 & & \mathit{notTainted}(Y) \\
\mathit{mayPointTo}(X,Y) & \triangleq & \neg \mathit{doesNotPointTo}(X,Y) \\
\mathit{mayDef}(Y) & \triangleq & \mathit{synDef}(Y) \lor \\
 & & (\mathit{stmt}(*X := \ldots) \land \mathit{mayPointTo}(X,Y)) \lor \\
 & & (\mathit{stmt}(\ldots := P(\ldots)) \land \neg \mathit{notTainted}(Y)) \\
\mathit{mayUse}(Y) & \triangleq & \mathit{synDef}(Y) \lor \\
 & & (\mathit{stmt}(\ldots := *X) \land \mathit{mayPointTo}(X,Y)) \lor \\
 & & \mathit{stmt}(\ldots := P(\ldots))
\end{array}
$$

The *unchanged* label is defined as follows. When E is not $*X$, $\mathit{unchanged}(E)$ is defined as the conjunction of $\neg \mathit{mayDef}$ for all the variables in E. For $*X$, $\mathit{unchanged}(*X)$ is defined as follows:

$$
\begin{array}{lcl}
\mathit{unchanged}(*X) & \triangleq & \neg \mathit{mayDef}(X) \land \\
 & & \neg \mathit{stmt}(*Y := \ldots) \land \\
 & & \neg \mathit{stmt}(\ldots := P(\ldots)) \land \\
 & & (\mathit{stmt}(Y := \ldots) \implies \\
 & & \qquad \mathit{doesNotPointTo}(X,\, Y))
\end{array}
$$

npUnchanged is a version of *unchanged* that does not use pointer information. For $*X$, $\mathit{npUnchanged}(*X)$ is *false*. When E is not $*X$, it is defined as the conjunction of $\neg \mathit{npMayDef}$ for all the variables in E.