Credible Compilation with Pointers

Martin C. Rinard and Darko Marinov
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{rinard,marinov}@lcs.mit.edu

Abstract

This paper presents the formal foundations and architectural design of a *credible compiler*, or a compiler that, in addition to a transformed program, produces a proof that the transformed program correctly implements the original input program. In our design, programs are represented using a standard low-level intermediate form based on control-flow graphs. The compiler is structured as a set of components. Each component performs a specific transformation and produces a proof that the transformation is correct. Typically, the correctness proof will consist of two subproofs: a subproof that the analysis of the input program produced a correct result, and a subproof that establishes a simulation relation between the original and transformed programs. The paper presents two logics, one for each kind of subproof, and shows that the logics are sound.

A novel and important feature of our framework is its simultaneous support for both formal reasoning and sophisticated compiler transformations that deal with the program and the target machine at a very low level. In particular, our logics allow the compiler to prove the correctness of low-level optimizations such as register allocation and instruction scheduling even in the presence of potentially aliased pointers into the memory of the machine.

1 Introduction

Today, compilers are black boxes. The programmer gives the compiler a program, and the compiler spits out an inscrutable bunch of bits. Until the programmer runs the program, he or she has no idea if the compiler has compiled the program correctly. Even running the program offers no guarantees — compiler errors may show up only for certain inputs. So the programmer must simply trust the compiler.

We propose instead to build a compiler that automatically generates a proof that it compiled the program correctly. The compiler is structured as a set of components; each component transforms the program to bring it closer to the final compiled result. Whenever a component runs, it generates a proof that the transformed program produces the same result as the program that it was given. A compiler driver links these components together to obtain the full compiler. After each transformation, the driver uses a simple automated proof checker to verify that the transformation preserved the semantics of the program. We call a compiler that generates these proofs a *credible* compiler, because it produces verifiable evidence that it is operating correctly.

This paper presents the formal foundations and architectural design of a credible compiler for standard imperative languages such as C. It defines a standard program representation that the different components can use to communicate, and introduces logics that the components can use to prove the correctness of their analysis results and transformations. A novel and important feature of our framework is its simultaneous support for both formal reasoning and sophisticated compiler transformations that deal with the program and the target machine at a very low level. In particular, our logics allow the compiler to prove the correctness of low-level optimizations such as register allocation and instruction scheduling even in the presence of potentially aliased pointers into the memory of the machine.

This low-level support is required if the compiler is to generate good code for languages with low-level constructs such as C pointers. But we believe that it is essential even for more disciplined languages such as ML and Java. Even though these languages provide a safe, restricted memory model at the source level, standard compiler optimizations such as induction variable elimination produce code that accesses memory directly in an unstructured way. So compilers for these languages must deal with a low-level memory model, even if the language presents a high level memory model to the programmer.

2 Compiler Development Problems

At first glance, it may seem that credible compilation is designed to solve the problem of unreliable compilers. But the fact is that production-quality compilers are very reliable. They very infrequently generate incorrect code, and are some of the most reliable software tools available. This reliability is understandable given the consequences of an incorrect compilation. Because the output is unreadable, failures are silent — in practice, the programmer can recognize a failed compilation only by observing an incorrect program execution. Because a programmer developing a program is typically struggling with his or her own programming errors, compiler errors can make it almost impossible to debug the program — every time the program executes incorrectly, the programmer must contend with the possibility that the error is not in his or her program at all! Programmers who encounter one of the relatively few errors in existing compilers have been known to spend days trying to isolate the cause of a single incorrect execution.

The real problem is the large development time and effort required to deliver a compiler with the necessary extreme level of reliability. The result is that production-quality compilers are almost always many years old and close to obsolete. They fail to incorporate the results of recent research and typically support only old versions of old languages. Consider, for example, the lag time for the development of Java compilers. Even though the language definition and interpreters have been widely available for years, reasonable compilers are only now starting to become available. And the quality of the generated code is far below that of mature compilers for previous-generation languages such as C, even though the mature compilers do not use recently developed compiler techniques that would significantly increase the performance of the generated code.

The difficulty of compiler development has also significantly limited progress in computer architecture. The latest Intel microprocessors are still built to execute code generated by compilers that are many years old, even though it is clear that the combination of an advanced VLIW architecture and sophisticated compiler would both perform better and use less silicon area. In fact, architects have produced technically superior microprocessors such as the Intel i860 only to see them fail in the marketplace because of a lack of compiler support. More recently, the Intel IA-64 is reportedly delayed, in part, because of the difficulty of developing an effective compiler for the architecture. Finally, the lack of effective compiler support for advanced architectural features such as the Intel MMX also limits the ability of the computer industry to deliver the benefits of advanced hardware.

3 Impact

Credible compilation can fundamentally change the way compilers are built. This change will eliminate many of the problems that make production-quality compilers lag so far behind other kinds of software. Here is why:

- **Buggier Compilers:** A credible compiler flags its errors immediately, even before the program finishes compiling. Because the compiler fails noisily, compiler errors do not interfere with the process of debugging the application — they are cleanly separated from errors in the application. Programmers can therefore use much less reliable compilers without significantly complicating the development of the application programs.

  Compiler developers will respond by reducing the reliability of their compilers in return for a faster development cycle. Compilers will be released much sooner and with much less testing than would otherwise be possible, and the frequency of new releases will be significantly increased. Developers will also be much more aggressive in their modifications to existing compilers. Shipped compilers will therefore track advances in languages, compiler technology, and computer architecture much more closely than they do now.

- **Open Source Development:** Before a developer can safely integrate a component into a compiler, there must be some evidence that the component will work. But there is currently no way to verify the correctness of compiler components. Developers are therefore typically reduced to relying on the reputation of the person that produced the component, rather than on the trustworthiness of the code itself. In practice, this means that the entire compiler is typically built by a small, cohesive group of people in a single organization. The compiler is closed in the sense that these people must coordinate any contribution to the compiler.

  Credible compilation eliminates the need for developers to trust each other. Anyone can take any component, integrate into their compiler, and use it. If a component operates incorrectly, it is immediately apparent, and the compiler can discard the transformation. There is no need to trust anyone. The compiler is now open and anyone can contribute. Instead of relying on a small group of people in one organization, the effort, energy, and intelligence of every compiler developer in the world can be productively applied to the development of one compiler.

- **Compiler Development:** Credible compilation will make it easier to develop the compiler — errors will be immediately visible because the purported correctness proof will not check out. The result will be faster development of an acceptably correct compiler.

4 Example

In this section we present an example that shows how a credible compiler can prove that it performed a translation correctly. Figure 1 presents the example program represented as a control flow graph. The program contains several assignment nodes; for example the node $5 : i \leftarrow i + x + y$ at label 5 assigns the value of the expression $i + x + y$ to the variable i and the node $6 : *p \leftarrow 2 * i$ at label 6 assigns the value of the expression $2 * i$ to the variable pointed to by p. There is also a conditional branch node $4 : \text{br } i < 24$. Control flows from this node through its outgoing left edge to the assignment node at label 5 if $i < 24$, otherwise control flows through the right edge to the exit node at label 7.

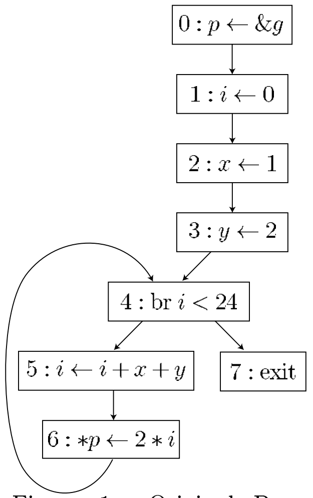

Figure 1: Original Program

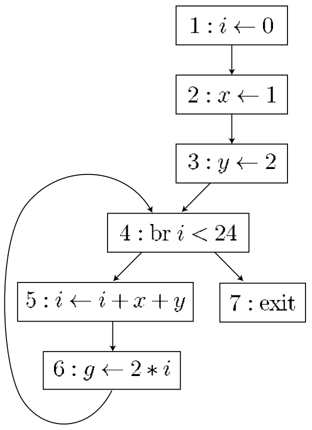

Figure 2: Program After Pointer Elimination

Figure 2 presents the program after pointer elimination. The compiler has replaced the node $6 : *p \leftarrow 2 * i$ at label

6 with the node $6 : g \leftarrow 2 * i$. The goal is to prove that this particular transformation on this particular program preserves the semantics of the original program. The goal is *not* to prove that the compiler will always transform an arbitrary program correctly.

To perform this optimization, the compiler did two things:

- **Analysis:** The compiler determined that p always points to g at the program point before node 6 executes.
- **Transformation:** The compiler used the analysis information to transform the program so that it generates the same result while (hopefully) executing in less time or space or consuming less power. In our example, the compiler replaces $*p$ with g.[1]

Our approach to proving optimizations correct supports this basic two-step structure. The compiler first proves that the analysis is correct, then uses the analysis results to prove that the original and transformed programs generate the same result. Here is how this approach works in our example.

4.1 Proving Analysis Results Correct

In our approach, the compiler uses a generalization of Floyd-Hoare logic to prove analysis results correct [4, 2].

In our example, the key invariant is that at the point just before the program executes node 6, it is always true that $p = \&g$. We represent this invariant as $\langle p = \&g \rangle 6$. Section 5.3 presents a logic that the compiler can use to prove such invariants. In effect, this logic allows the compiler to construct proofs by induction on the length of the partial executions of the program.

The simplest way for the compiler to generate a proof of $\langle p = \&g \rangle 6$ is for it to generate a set of invariants that represent the analysis results, then use the logic to prove that all of the invariants hold. In our example, the compiler generates the set of invariants that p points to g at every point in the program except 0.

Conceptually, the compiler proves this set of invariants by tracing execution paths. The proof is by induction on the structure of the partial executions of the program. For each invariant, the compiler first assumes that the invariants at all preceding nodes in the control flow graph are true. It then traces the execution through each preceding node to verify the invariant at the next node. We next present an outline of the proofs for several key invariants. The compiler can use the logic presented in Section 5.3 to produce machine-verifiable versions of these proofs.

- $\langle p = \&g \rangle 1$ because the only preceding node, node 0, sets p to $\&g$.
- To prove $\langle p = \&g \rangle 3$, first assume $\langle p = \&g \rangle 2$. The only way for control to reach node 3 is from node 2 and node 2 does not change the value of p. Therefore $\langle p = \&g \rangle 3$.
- To prove $\langle p = \&g \rangle 4$, assume $\langle p = \&g \rangle 3$ and $\langle p = \&g \rangle 6$. Then consider the two preceding nodes, nodes 3 and 6. Because $\langle p = \&g \rangle 3$ and node 3 does not affect the value of p, $\langle p = \&g \rangle 4$. Also, $\langle p = \&g \rangle 6$ so $p \neq \&p$ at node 6, i.e. p does not point to itself, and node 6 cannot change the value of p. Therefore $\langle p = \&g \rangle 4$.

[1]In the example, we also eliminate the assignment of $\&g$ to p at node 0. In practice, this transformation would be performed by the dead assignment elimination transformation discussed in Section 7.2.

In this proof we have assumed that the compiler generates an invariant at almost all of the nodes in the program. More traditional approaches use fewer invariants, typically one invariant per loop, then produce proofs that trace paths consisting of multiple nodes. The logic presented in Section 5.3 supports both styles of proofs.

4.2 Proving Transformations Correct

When a compiler transforms a program, there are typically some externally observable effects that it must preserve. A standard requirement, for example, is that the compiler must preserve the input/output relation of the program. In our framework, we assume that the compiler is operating on a compilation unit such as procedure or method, and that there are externally observable variables such as global variables or object instance variables. The compiler must preserve the final values of these variables. All other variables are either parameters or local variables, and the compiler is free to do whatever it wants to with these variables so long as it preserves the final values of the observable variables. The compiler may also assume that the initial values of the observable variables and the parameters are the same in both cases.

In our example, the only requirement is that the transformation must preserve the final value of the global variable g. The compiler proves this property by proving a simulation correspondence between the original and transformed programs. To present the correspondence, we must be able to refer, in the same context, to variables and node labels from the two programs. We adopt the convention that all entities from the original program P will have a subscript of P, while all entities from the transformed program T will have a subscript of T. So i_P refers to the variable i in the original program, while i_T refers to the variable i in the transformed program.

In our example, the compiler proves that the transformed program implements the original program in the following sense: for every execution of the transformed program T that reaches the final node 7_T, there exists an execution of the original program P that reaches the final node 7_P such that $(g_P \text{ at } 7_P) = (g_T \text{ at } 7_T)$. We call such a correspondence a *simulation invariant*, and write it as $\langle g_P \rangle 7_P \lhd \langle g_T \rangle 7_T$. In Section 5.4 we present a logic that the compiler can use to prove simulation invariants.

The compiler typically generates a set of simulation invariants, then uses the logic to construct a proof of the correctness of all of the simulation invariants. The proof is by induction on the length of the partial executions of the original program. We next outline how the compiler can use this approach to prove $\langle g_P \rangle 7_P \lhd \langle g_T \rangle 7_T$. First, the compiler is given that $\langle g_P \rangle 0_P \lhd \langle g_T \rangle 0_T$ — in other words, the values of g_P and g_T are the same at the start of the two programs. The compiler then generates the following simulation invariants:

- $\langle g_P \rangle 1_P \lhd \langle g_T \rangle 1_T$
- $\langle (g_P, i_P) \rangle 2_P \lhd \langle (g_T, i_T) \rangle 2_T$
- $\langle (g_P, i_P, x_P) \rangle 3_P \lhd \langle (g_T, i_T, x_T) \rangle 3_T$
- $\langle (g_P, i_P, x_P, y_P) \rangle 4_P \lhd \langle (g_T, i_T, x_T, y_T) \rangle 4_T$
- $\langle (g_P, i_P, x_P, y_P) \rangle 5_P \lhd \langle (g_T, i_T, x_T, y_T) \rangle 5_T$
- $\langle (g_P, i_P, x_P, y_P) \rangle 6_P \lhd \langle (g_T, i_T, x_T, y_T) \rangle 6_T$
- $\langle g_P \rangle 7_P \lhd \langle g_T \rangle 7_T$

The key invariants are $\langle g_P \rangle 7_P \lhd \langle g_T \rangle 7_T$ at the end of the program, $\langle (g_P, i_P, x_P, y_P) \rangle 6_P \lhd \langle (g_T, i_T, x_T, y_T) \rangle 6_T$ and $\langle (g_P, i_P, x_P, y_P) \rangle 4_P \lhd \langle (g_T, i_T, x_T, y_T) \rangle 4_T$. We next outline the proofs of these invariants. The compiler can use the logic presented in Section 5.4 to produce machine-verifiable versions of these proofs. The proof relies on the property that since p_P points to g_P, $*p_P = g_P$ at 6_P.

- To prove $\langle (g_P, i_P, x_P, y_P) \rangle 4_P \lhd \langle (g_T, i_T, x_T, y_T) \rangle 4_T$, first assume $\langle (g_P, i_P, x_P, y_P) \rangle 6_P \lhd \langle (g_T, i_T, x_T, y_T) \rangle 6_T$ and $\langle (g_P, i_P, x_P) \rangle 3_P \lhd \langle (g_T, i_T, x_T) \rangle 3_T$. There are two paths to 4_T:
  - Control flows from 6_T to 4_T, with $(g_T \text{ at } 4_T) = (2 * i_T \text{ at } 6_T)$. The corresponding path in P is from 6_P to 4_P, with $(*p_P \text{ at } 4_P) = (2 * i_P \text{ at } 6_P)$. The analysis proofs showed that $(p_P \text{ at } 6_P) = \&g$, so $(*p_P \text{ at } 6_P) = (g_P \text{ at } 6_P)$. The assumed simulation invariant $\langle (g_P, i_P, x_P, y_P) \rangle 6_P \lhd \langle (g_T, i_T, x_T, y_T) \rangle 6_T$ allows us derive $(2*i_P \text{ at } 6_P) = (2 * i_T \text{ at } 6_T)$ and hence verify a correspondence between the values of $(g_P \text{ at } 4_P)$ and $(g_T \text{ at } 4_P)$; namely that they are equal. Because 6_P does not change i_P, x_P, or y_P and 6_T does not change i_T, x_T, or y_T, corresponding variables at 4_P and 4_T also have the same value.
  - Control flows from 3_T to 4_T. The corresponding path in P is from 3_P to 4_P, so we can apply the assumed simulation invariant $\langle (g_P, i_P, x_P) \rangle 3_P \lhd \langle (g_T, i_T, x_T) \rangle 3_T$ to derive $(g_P \text{ at } 4_P) = (g_T \text{ at } 4_T)$, $(i_P \text{ at } 4_P) = (i_T \text{ at } 4_T)$, and $(x_P \text{ at } 4_P) = (x_T \text{ at } 4_T)$. Also, nodes 3_P and 3_T set the variables y_P and y_T, respectively, to the same value, namely 2, and so $(y_P \text{ at } 4_P) = (y_T \text{ at } 4_T)$.
- To prove $\langle (g_P, i_P, x_P, y_P) \rangle 6_P \lhd \langle (g_T, i_T, x_T, y_T) \rangle 6_T$, assume $\langle (g_P, i_P, x_P, y_P) \rangle 5_P \lhd \langle (g_T, i_T, x_T, y_T) \rangle 5_T$. The only path to 6_T goes from 5_T, with $(i_T \text{ at } 6_T) = (i_T \text{ at } 5_T) + (x_T \text{ at } 5_T) + (y_T \text{ at } 5_T)$. The corresponding path in P goes from 5_P to 6_P, with $(i_P \text{ at } 6_P) = (i_P \text{ at } 5_P) + (x_P \text{ at } 5_P) + (y_P \text{ at } 5_P)$. We can apply the assumed $\langle (g_P, i_P, x_P, y_P) \rangle 5_P \lhd \langle (g_T, i_T, x_T, y_T) \rangle 5_T$ to derive $(i_P \text{ at } 5_P) + (x_P \text{ at } 5_P) + (y_P \text{ at } 5_P) = (i_T \text{ at } 5_T) + (x_T \text{ at } 5_T) + (y_T \text{ at } 5_T)$, and therefore $(i_P \text{ at } 6_P) = (i_T \text{ at } 6_T)$. Since 5_P does not change g_P, x_P, or y_P and 5_T does not change g_T, x_T, or y_T we can derive $(g_P \text{ at } 6_P) = (g_T \text{ at } 6_T)$, $(x_P \text{ at } 6_P) = (x_T \text{ at } 6_T)$ and $(y_P \text{ at } 6_P) = (y_T \text{ at } 6_T)$.
- To prove $\langle g_P \rangle 7_P \lhd \langle g_T \rangle 7_T$, first assume that simulation invariant $\langle (g_P, i_P, x_P, y_P) \rangle 4_P \lhd \langle (g_T, i_T, x_T, y_T) \rangle 4_T$ holds. For each path to 7_T in T, we must find a corresponding path in P to 7_P such that the values of g_P and g_T are the same in both paths. The only path to 7_T goes from 4_T to 7_T when $i_T \geq 24$. The corresponding path in P goes from 4_P to 7_P when $i_P \geq 24$. Because $\langle (g_P, i_P, x_P, y_P) \rangle 4_P \lhd \langle (g_T, i_T, x_T, y_T) \rangle 4_T$, control flows from 4_P to 7_P whenever control flows from 4_T to 7_T. The simulation invariant $\langle (g_P, i_P, x_P, y_P) \rangle 4_P \lhd \langle (g_T, i_T, x_T, y_T) \rangle 4_T$ also implies that the values of g_P and g_T are the same in both cases.

5 Logical Foundations

In this section we present the logical foundations of credible compilation. We formally define a program representation based on control flow graphs and define an operational semantics for this representation. We present the logic used to prove standard invariants and show that this logic is sound. We also present the logic used to prove simulation invariants and show that this logic is sound.

5.1 Program Representation

In this section we define a simple standard program representation based on control flow graphs. We use it to present the major ideas and concepts in the remainder of the paper. We expect that a practical implementation would require a more elaborate representation, although the representation would be quite similar to the one we present here. One additional issue is that some components may need to operate on program representations that are specialized for the specific analysis and transformation. These components would contain translators between the standard and specialized representations.

We start with expressions e and conditions c. For simplicity we assume the program computes on integer values; we denote the set of integers by $z \in \mathcal{Z}$. We also assume a set of variables V with addresses chosen from a set of locations $\mathcal{L} \subseteq \mathcal{Z}$.

We use C-style syntax $\&v$ to denote the address of variable $v \in V$. The expression v denotes the value of v, i.e. $v \overset{def}{=} *\&v$, the value at the address $\&v$. Similarly, $*e$ stands for the value stored at memory location e. Note that we use variables in two different ways, depending on the context. In the expression $\&v$, v identifies the memory location whose address is taken. In all other expressions, v denotes the value stored at that location. The correct meaning should be clear from the context.

The abstract syntax in Figure 3 defines the set of expressions e. α, ρ and λ represent, respectively, arithmetical, relational and logical operators, and σ ranges over all binary operators used.

$$e ::= z | \&v | v | e \alpha e | \Leftrightarrow e | * e | \text{true} | \text{false} | e \rho e | e \lambda e | \neg e$$

$$\alpha ::= + | \Leftrightarrow | * | / | \% \qquad \rho ::= = | \neq | > | \leq | < | \geq$$
$$\lambda ::= \wedge | \vee | \Rightarrow | \Leftrightarrow \qquad \sigma ::= \alpha | \rho | \lambda$$

Figure 3: Grammar for Expressions

In some cases, we interpret an expression e as a condition c whose value is true or false. We adopt the C convention that a condition is true if its value is not zero, and false if its value is zero. In the expression grammar above, true is 1 and false is 0.

Each control flow graph is composed of a set of nodes. Each node has its own label; these labels are used to determine the flow of control between nodes. Each node is one of the following types:

- **Assignment:** An assignment node $s : w \leftarrow e \; t$ has its label s, left hand side w, an expression e and a label t. The left hand side w can be either a variable v or an expression $*e'$, where e' evaluates to a memory location. The notation $\&w$ denotes the syntactic elimination of one level of dereferencing. If $w \equiv v$, then $\&w$ is defined to be $\&v$; if $w \equiv *e'$, then $\&w$ is defined to be e'. Here $\equiv$ denotes syntactic equivalence. When the assignment node executes, it evaluates $\&w$ and e and stores the value of e in the memory location $\&w$. Execution continues at the node whose label is t.

- **Conditional Branch:** A conditional branch node s : br c t_1 t_2 has its label s, a condition c and two labels t_1 and t_2. When the node executes, it evaluates c. If c is true, execution continues at the node whose label is t_1. Otherwise, execution continues at the node whose label is t_2.

- **Nop:** A nop node s : nop t has its label s and another label t. When the node executes, execution continues at the node whose label is t.

- **Exit:** The exit node s_x : exit is the last node in the graph.

There is a unique entry node with label s_0 and a unique exit node with label s_x. We require that there be a path from the entry node to the exit node, and that no two distinct nodes have the same label.

We use the notation that $s : w \leftarrow e\ t$ is true if there exists an assignment node with label s, left hand side w, expression e and label t in the control flow graph, and false otherwise. Also, s : br c t_1 t_2 is true if there is a conditional branch node in the control flow graph with label s, condition c, and labels t_1 and t_2 in the program, and false otherwise, and similarly for nop and exit nodes. We use this notation to define the set of predecessors of a node in the control flow graph:

Definition 1 *Given a label t, the set of predecessors of t is the set of all labels of nodes from which control may flow directly to t:*

$$\begin{aligned}\text{pred}(t) \quad = \quad & \{s | s : w \leftarrow e\ t\} \cup \{s | s : \text{nop}\ t\} \cup \\ & \{s | s : \text{br}\ c\ t\ t'\} \cup \{s | s : \text{br}\ c\ t'\ t\}\end{aligned}$$

We require that the entry node s_0 have no predecessors, i.e., $\text{pred}(s_0) = \emptyset$. Also note that the exit node has no successors, i.e. for all s, $s_x \notin \text{pred}(s)$.

5.2 Operational Semantics

We next present a simple operational semantics for control flow graphs. The semantics uses configurations $\langle s, m\rangle$, which consist of the label s of the next node to execute and a memory $m : \mathcal{L} \to \mathcal{Z}$ that maps memory locations to their values.

The environment function $a : V \to \mathcal{L}$ maps variables to the addresses in memory where their values are stored. We impose two restrictions on this function. First, for all v, $a(v) \neq 0$, i.e., none of the variables is mapped to location 0.[2] Second, we require the environment function a to be injective, i.e., different variables are stored in different locations. This requirement ensures that $\&v_1 = \&v_2$ if and only if v_1 and v_2 are syntactically equal, i.e., $v_1 \equiv v_2$.

Given an expression e and a memory m, $\overline{m}(e)$ denotes the value of e when evaluated in the context of the memory m. The rules in Figure 4 define the expression evaluation function $\overline{m}(e)$. We assume that the expression $*e$ always evaluates to a valid value, i.e. $\overline{m}(e) \in \mathcal{L}$.

The operational semantics is defined using a transition function $\to$ which maps each configuration $\langle s, m\rangle$ to its successor configuration $\langle s', m'\rangle$. The successor configuration is obtained by executing the node at label s in the context of memory m. Figure 5 presents the rules that define the transition function. We assume two disjoint sets of variables: local variables and externally observable variables. In the initial memory m_0, local variables have value 0 and observable variables have arbitrary values. The initial value of variable v is denoted v_0, i.e. $m_0(\&v) = v_0$.

[2] To make the flow-insensitive pointer analyses in Section 6.3.1 produce meaningful results, location 0 must have the special behavior that reading from location 0 always gives 0, while writing to location 0 does not change the memory.

$$\begin{gathered}\overline{m}(\&v) = a(v)\\ \overline{m}(\& * e) = \overline{m}(e)\\ \overline{m}(v) = m(a(v))\\ \overline{m}(*e) = m(\overline{m}(e))\\ \overline{m}(z) = z\\ \overline{m}(\text{true}) = \text{true}\\ \overline{m}(\text{false}) = \text{false}\\ \overline{m}(\Leftrightarrow e) = \Leftrightarrow \overline{m}(e)\\ \overline{m}(\neg e) = \neg \overline{m}(e)\\ \overline{m}(e_1 \sigma e_2) = \overline{m}(e_1) \lambda \overline{m}(e_2)\end{gathered}$$

Figure 4: Evaluating Expressions in Memory m

$$\frac{s : w \leftarrow e\ t, \overline{m}(\&w) \neq 0}{\langle s, m\rangle \to \langle t, m[\overline{m}(\&w) \mapsto \overline{m}(e)]\rangle}$$

$$\frac{s : w \leftarrow e\ t, \overline{m}(\&w) = 0}{\langle s, m\rangle \to \langle t, m\rangle}$$

$$\frac{s : \text{nop}\ t}{\langle s, m\rangle \to \langle t, m\rangle}$$

$$\frac{s : \text{br}\ c\ t_1\ t_2, \overline{m}(c)}{\langle s, m\rangle \to \langle t_1, m\rangle}$$

$$\frac{s : \text{br}\ c\ t_1\ t_2, \neg\overline{m}(c)}{\langle s, m\rangle \to \langle t_2, m\rangle}$$

Figure 5: Operational Semantics

We use the operational semantics to define the concept of a *partial execution* of a control flow graph. A partial execution starts at the entry node in the graph, and executes part of the computation.

Definition 2 *A partial execution of a control flow graph is a sequence of configurations $\langle s_0, m_0\rangle \to \cdots \to \langle s_n, m_n\rangle$ in which each configuration $\langle s_{i+1}, m_{i+1}\rangle$ is the successor of the preceding configuration $\langle s_i, m_i\rangle$ in the sequence.*

5.3 Standard Invariants

We next present the logic that the compiler uses to prove that its analysis results are correct. The logic consists of a set of proof rules; these rules are a version of Floyd-Hoare proof rules adapted for control flow graphs. The rules operate on several types of invariants:

- $\langle i\rangle s$: the condition i is always true at the program point before the execution of the node whose label is s.

- $s\langle i\rangle t$: the condition i is always true at the program point before the execution of the node whose label is t, if control flowed directly to t from s.

- $\langle i\rangle s \cdot t$: the condition i is always true at the program point before the execution of the node whose label is s, if control will flow next to t.

The proof rules assume a set I of invariants; we require that invariants of the form $s\langle i\rangle t$ or $\langle i\rangle s \cdot t$ do not appear in I. We also assume the existence of a logic for proving standard relationships between integers such as $z < z+1$ and $x < 4 \wedge y < 3 \Rightarrow x + y < 7$.

5.3.1 Substitution in the Presence of Pointers

In standard proof systems for reasoning about programs, the inference rule for assignment statements uses a simple substitution. If our programs did not contain pointers, the proof rules would model assignment statements by simply replacing the variable on the left hand side of the assignment with the expression on the right hand side.

This approach fails in the presence of pointers because of the possibility of aliasing. We therefore define a generalization of substitution which may produce multiple results. Each result is conditioned on an aliasing context that characterizes the relevant aliasing relationships. The substitution relation $i\{e/w\}|c \doteq i'$ states that when w is replaced by e in the invariant i, the result is a new invariant i' if the set of aliasing relationships in the condition c hold. The aliasing relationships in c are expressed using points-to conditions of the form $p = \&x$, which states that p must point to x for the substitution result to be valid. Conversely, the condition $p \neq \&x$ states that p must not point to x for the result to be valid. Figure 6 presents the inference rules that define the substitution relation. The relation is defined inductively on the structure of expressions.

The main rules are 9, 10 and 11. The first two are rules for substitution if the points-to condition is satisfied in an aliasing context and conversely if it is not satisfied. Rule 11 is used for case analysis, which allows results from different aliasing contexts to be combined in a single expression that lists the result for each context. We call this new kind of expression a *partial conditioned expression*. The grammar for partial conditioned expressions is:

$$l ::= c?e | l;l$$

Note that the conditions in a partial conditioned expression may not cover all of the possible cases, i.e. for $c_1?e_1;\ldots;c_n?e_n$, $c_1 \vee \ldots \vee c_n$ may not be equivalent to true. We define the value of a partial conditioned expression in memory m as:

$$\overline{m}(c?e) = \begin{cases} \overline{m}(e) & : \quad \overline{m}(c) \\ \perp & : \quad otherwise \end{cases}$$

$$\overline{m}(l_1;l_2) = \begin{cases} \overline{m}(l_1) & : \quad \overline{m}(l_1) \neq \perp \\ \overline{m}(l_2) & : \quad \overline{m}(l_1) = \perp \end{cases}$$

where $\perp$ represents undefined value. Partial conditioned expressions are generated by substitution rules, and do not appear in assignment nodes in the program. We also do not allow partial conditioned expressions to be used in the invariants and aliasing contexts. Rules 12 and 13 define the substitution relation for partial conditioned expressions. We also define notation for the common case of a two-element list with inverted conditions:

$$c?e_1 : e_2 \overset{def}{=} c?e_1; \neg c?e_2$$

Binary operators distribute over the cases in partial conditioned expressions:

$$(c?e')\sigma e = c?(e'\sigma e)$$
$$(l_1;l_2)\sigma e = (l_1\sigma e);(l_2\sigma e)$$

In this section, we presented the basic substitution rules. Section 6.1 presents several derived rules, which are customized for specific cases of w (which is of one of two forms: v or $*e$) and the aliasing condition (which either holds or does not hold). These derived rules are clearer and enable shorter substitution proofs.

$$\frac{}{z\{e/w\}|c \doteq z} \quad (1)$$

$$\frac{}{\mathrm{true}\{e/w\}|c \doteq \mathrm{true}} \quad (2)$$

$$\frac{}{\mathrm{false}\{e/w\}|c \doteq \mathrm{false}} \quad (3)$$

$$\frac{}{\&v\{e/w\}|c \doteq \&v} \quad (4)$$

$$\frac{e_1\{e/w\}|c \doteq e_1'}{\Leftrightarrow e_1\{e/w\}|c \doteq \Leftrightarrow e_1'} \quad (5)$$

$$\frac{e_1\{e/w\}|c \doteq e_1'}{\neg e_1\{e/w\}|c \doteq \neg e_1'} \quad (6)$$

$$\frac{e_1\{e/w\}|c \doteq e_1', e_2\{e/w\}|c \doteq e_2'}{e_1\sigma e_2\{e/w\}|c \doteq e_1'\sigma e_2'} \quad (7)$$

$$\frac{*\&v\{e/w\}|c \doteq e_1'}{v\{e/w\}|c \doteq e_1'} \quad (8)$$

$$\frac{e_1\{e/w\}|c \doteq e_1', c \Rightarrow (e_1' = \&w \wedge e_1' \neq 0)}{*e_1\{e/w\}|c \doteq e} \quad (9)$$

$$\frac{e_1\{e/w\}|c \doteq e_1', c \Rightarrow (e_1' \neq \&w \vee e_1' = 0)}{*e_1\{e/w\}|c \doteq *e_1'} \quad (10)$$

$$\frac{e'\{e/w\}|c_1 \doteq e_1, e'\{e/w\}|c_2 \doteq e_2}{e'\{e/w\}|c_1 \vee c_2 \doteq c_1?e_1; c_2?e_2} \quad (11)$$

$$\frac{c_1\{e/w\}|c \doteq c_1', e_1\{e/w\}|c \doteq e_1'}{c_1?e_1\{e/w\}|c \doteq c_1'?e_1'} \quad (12)$$

$$\frac{l_1\{e/w\}|c \doteq l_1', l_2\{e/w\}|c \doteq l_2'}{l_1;l_2\{e/w\}|c \doteq l_1';l_2'} \quad (13)$$

Figure 6: Proof Rules for Substitution

5.3.2 Proof Rules for Standard Invariants

Figure 7 presents the proof rules for standard invariants. Because the proof rules work with control flow graphs instead of structured programs, they propagate expressions across edges in the graph instead of through structured control flow statements.

5.3.3 Aliasing Invariants

In general, the correctness of code transformations may depend on the aliasing relationships at various points in the program. The compiler must therefore prove that its pointer

$$\frac{\mathrm{pred}(t) \neq \emptyset, \forall s \in \mathrm{pred}(t).\, I \vdash s\langle i\rangle t}{I \vdash \langle i\rangle t} \quad (14)$$

$$\frac{s : \mathrm{nop}\ t, I \vdash \langle i\rangle s \cdot t}{I \vdash s\langle i\rangle t} \quad (15)$$

$$\frac{\begin{array}{c} s : w \leftarrow e\ t, i\{e/w\}|c \doteq i', \\ I \vdash \langle c\rangle s \cdot t, I \vdash \langle i'\rangle s \cdot t \end{array}}{I \vdash s\langle i\rangle t} \quad (16)$$

$$\frac{s : \mathrm{br}\ c\ t_1\ t_2, I \vdash \langle c \Rightarrow i\rangle s \cdot t_1}{I \vdash s\langle i\rangle t_1} \quad (17)$$

$$\frac{s : \mathrm{br}\ c\ t_1\ t_2, I \vdash \langle \neg c \Rightarrow i\rangle s \cdot t_2}{I \vdash s\langle i\rangle t_2} \quad (18)$$

$$\frac{I \vdash \langle i\rangle s}{I \vdash \langle i\rangle s \cdot t} \quad (19)$$

$$\frac{\langle i'\rangle s \in I, i' \Rightarrow i}{I \vdash \langle i\rangle s \cdot t} \quad (20)$$

$$\frac{i}{I \vdash \langle i\rangle s} \quad (21)$$

Figure 7: Proof Rules for Standard Invariants

analysis results are correct if it is to prove that the transformation is correct.

To prove that the pointer analysis results are correct, the compiler first generates invariants that represent the pointer analysis results. For example, an invariant of the form $p = \&x$ states that p must point to x, while an invariant of the form $p = \&x \vee p = \&y$ states that p must point to either x or y. Similarly, $p = \&x \wedge q = \&y$ states that p must point to x and q must point to y. Aliasing invariants, as well as the other invariants, can be arbitrary expressions as long as they do not include partial conditioned expressions. The compiler then uses the proof rules in Figure 7 to prove that these invariants are correct.

There are many research results that deal with proving properties of programs that use pointers. The basic idea of our approach is similar to that of Morris [7]. The difference in that our logic allows the compiler to generate proofs only for those aliasing contexts that actually hold in the program. This property reduces the size of the proofs that the compiler must generate.

5.3.4 Soundness of Proof Rules for Substitution

We next show that the proof rules for substitution in Figure 6 are sound. More precisely, we show that if c holds before the assignment statement $w \leftarrow e$ and $i\{e/w\}|c \doteq i'$, then i' has the same value before the statement as i does after. The proof is omitted for brevity of presentation.

Theorem 1 *Assume a proof of $i\{e/w\}|c \doteq i'$ and a partial execution $\langle s_0, m_0\rangle \to \cdots \to \langle s, m'\rangle \to \langle t, m\rangle$ such that $s : w \leftarrow e\ t$. Then $\overline{m'}(c)$ is* true *implies $\overline{m'}(i') = \overline{m}(i)$.*

5.3.5 Soundness of Proof Rules for Standard Invariants

We next present a key soundness theorem: that if there exists a proof of all of the invariants in I, then the invariants correctly reflect the relationships during the execution of the program. We state a lemma used in the theorem and the theorem itself without proofs.

Lemma 1 *Assume for all $\langle i_1\rangle s_1 \in I$, $I \vdash \langle i_1\rangle s_1$. Also assume a proof of $I \vdash \langle i\rangle s \cdot t$ and a partial execution $\langle s_0, m_0\rangle \to \cdots \to \langle s, m\rangle$ such that $I \vdash \langle i'\rangle s$ implies $\overline{m}(i')$ is* true. *Then $\overline{m}(i)$ is* true.

Theorem 2 *Assume for all standard invariants $\langle i\rangle s \in I$, $I \vdash \langle i\rangle s$. Then $I \vdash \langle i\rangle t$ and $\langle s_0, m_0\rangle \to \cdots \to \langle t, m\rangle$ implies $\overline{m}(i)$ is* true.

5.4 Simulation Invariants

We next present the logic that the compiler uses to prove simulation invariants between two programs P and T. For purposes of presentation, we adopt the convention that P is the original program and T is the transformed program, although of course the logic imposes no constraint on the origin of the two programs. We assume that P and T are two disjoint control flow graphs with entry nodes s_0^P and s_0^T and initial memories m_0^P and m_0^T, respectively.

We also assume sets of memory locations corresponding to externally observable variables $\{o_1^P, \ldots, o_n^P\}$ of P and $\{o_1^T, \ldots, o_n^T\}$ of T. Corresponding variables have the same values at the start of the program — i.e., $m_0^P(o_i^P) = m_0^T(o_i^T)$ for $1 \leq i \leq n$. We require the data layout of the observable variables to be the same in both programs. More precisely, we define $a(o_i^P) = a(o_i^T)$ for $1 \leq i \leq n$.

Simulation invariants consist of two partial simulation invariants that together express a simulation relationship between the partial executions of the programs. For example, $\langle c_1, e_1\rangle s_1 \lhd \langle c_2, e_2\rangle s_2$ is true if for all partial executions of T that reach s_2 with the condition c_2 true, there exists a partial execution of P that reaches s_1 with c_1 true such that $e_1 = e_2$. Like the logic for standard invariants presented in Section 5.3, the logic for simulation invariants uses multiple labels to express how the flow of control affects relationships between the two programs.

Definition 3 *A partial simulation invariant p has the form $\langle c, e\rangle t$, $s\langle c, e\rangle t$ or $\langle c, e\rangle s \cdot t$, where c is a condition and e is an expression.*

We adopt the convention that a partial simulation invariant of the form $\langle e\rangle t$, $s\langle e\rangle t$, or $\langle e\rangle s \cdot t$ denotes, respectively, $\langle \mathrm{true}, e\rangle t$, $s\langle \mathrm{true}, e\rangle t$, or $\langle \mathrm{true}, e\rangle s \cdot t$.

Definition 4 *A simulation invariant has the form $p_1 \lhd p_2$, where p_1 and p_2 are partial simulation invariants.*

Figures 8, 9, and 10 present the proof rules. Each proof propagates the partial simulation invariants against the flow of control through the two programs. Eventually, the partial simulation invariants reach program points where it is possible to terminate the proof by applying rule 22 or rule 23. The rules in Figure 9 propagate the partial simulation invariant from the original program; the rules in Figure 10 propagate the partial simulation invariant from the transformed program.

The proof rules all refer to a set I of invariants. In general, this set will contain both standard invariants of the form $\langle c\rangle s$ and simulation invariants of the form $\langle c_1, e_1\rangle s_1 \lhd \langle c_2, e_2\rangle s_2$. We require that it does not contain simulation invariants whose partial simulation invariants are of the form $s\langle c, e\rangle t$ or $\langle c, e\rangle s \cdot t$.

The proof rules illustrate a key difference between the treatment of the original and transformed programs. Rule 29 requires that the simulation invariant hold on all paths in the transformed program. Rule 24 requires only that the simulation invariant hold on one path in the original program. This difference reflects the asymmetry in the implicit quantifiers of the simulation invariant, which is true if for all paths in the transformed program, there exists a path in the original program that satisfies the appropriate conditions.

5.4.1 The Simulation Condition

To prove that the transformed program implements the original program, the compiler generates a set of invariants I and a proof of each invariant. We require one of the invariants to state that the transformed program preserves the values of the externally observable variables. We formalize this concept as follows:

Definition 5 *A transformed program T implements an original program P if there exists a set of invariants I such that*

- *for all standard invariants $\langle i\rangle s \in I$, $I \vdash \langle i\rangle s$,*
- *for all simulation invariants $\langle c_1, e_1\rangle s_1 \lhd \langle c_2, e_2\rangle s_2 \in I$, $I \vdash \langle c_1, e_1\rangle s_1 \lhd \langle c_2, e_2\rangle s_2$, and*
- *simulation invariant $\langle (o_1^P, \ldots, o_n^P)\rangle s_x^P \lhd \langle (o_1^T, \ldots, o_n^T)\rangle s_x^T \in I$, where $\{o_1^P, \ldots, o_n^P\}$ and $\{o_1^T, \ldots, o_n^T\}$ are sets of corresponding externally observable variables, s_x^P is the exit node in P, and s_x^T is the exit node in T.*

5.4.2 Standard Form Proofs of Simulation Invariants

We next introduce the concept of a standard form for proofs of simulation invariants. This standard form simplifies the presentation of the soundness proofs. We also expect the compiler to generate proofs in standard form, although there is of course no requirement that it do so.

A standard form proof has the following structure. Each leaf in the proof tree is a use of rule 22 or 23. Along each path in the proof tree from the leaves towards the root, the proof first uses rules 24 through 28 to propagate the partial simulation invariant from the original program through the program. Note that in this phase of the proof tree, each rule use has exactly one child. Next, uses of rules 29 through 35 appear on the path. These uses propagate the partial simulation invariant from the transformed program T. Because the proof must verify the simulation invariant for all paths in the transformed program, uses of rule 29 will have one child for each predecessor of the corresponding node in the control flow graph.

Definition 6 *A proof of a simulation invariant is in standard form if all uses of rules 24 through 28 precede all uses of rules 29 through 35.*

Theorem 3 *If $I \vdash p_1 \lhd p_2$, then there exists a proof of $I \vdash p_1 \lhd p_2$ that is in standard form.*

5.4.3 Soundness of Proof Rules for Simulation Invariants

We next show that the proof rules for simulation invariants are sound. We first state two lemmas, then the theorem.

Lemma 2 *Assume that for all $\langle i\rangle s \in I$, $I \vdash \langle i\rangle s$ and for all $\langle c_1, e_1\rangle s_1 \lhd \langle c_2, e_2\rangle s_2 \in I$, $I \vdash \langle c_1, e_1\rangle s_1 \lhd \langle c_2, e_2\rangle s_2$. Assume a standard form proof of $I \vdash \langle c_1, e_1\rangle s_1 \lhd p$, whose last rule is one of 22, 23 or 29, and $p = \langle c_2, e_2\rangle s_2$ or $p = \langle c_2, e_2\rangle s_2 \cdot t$. Also assume a partial execution $\langle s_0^T, m_0^T\rangle \to \cdots \to \langle s_2, m_2\rangle$ such that $\overline{m_2}(c_2)$ is* true. *If $p = \langle c_2, e_2\rangle s_2 \cdot t$, also assume that $I \vdash \langle c, e\rangle s \lhd \langle c', e'\rangle s_2$ and $\overline{m_2}(c')$ is* true *implies that there exists a partial execution $\langle s_0^P, m_0^P\rangle \to \cdots \to \langle s, m\rangle$ such that $\overline{m}(c)$ is* true *and $\overline{m}(e) = \overline{m_2}(e')$. Then there exists a partial execution $\langle s_0^P, m_0^P\rangle \to \cdots \to \langle s_1, m_1\rangle$ such that $\overline{m_1}(c_1)$ is* true *and $\overline{m_1}(e_1) = \overline{m_2}(e_2)$.*

Lemma 3 *Assume that for all $\langle i\rangle s \in I$, $I \vdash \langle i\rangle s$ and for all $\langle c_1, e_1\rangle s_1 \lhd \langle c_2, e_2\rangle s_2 \in I$, $I \vdash \langle c_1, e_1\rangle s_1 \lhd \langle c_2, e_2\rangle s_2$. Assume a standard form proof of $I \vdash \langle c_1, e_1\rangle s_1 \lhd \langle c_2, e_2\rangle s_2 \cdot t$, and a partial execution $\langle s_0^T, m_0^T\rangle \to \cdots \to \langle s_2, m_2\rangle$ such that $m_2(c_2)$ is* true. *Also assume that $I \vdash \langle c, e\rangle s \lhd \langle c', e'\rangle s_2$ and $\overline{m_2}(c')$ is* true *implies that there exists a partial execution $\langle s_0^P, m_0^P\rangle \to \cdots \to \langle s, m\rangle$ such that $\overline{m}(c)$ is* true *and $\overline{m}(e') = \overline{m_2}(e)$. Then there exists a partial execution $\langle s_0^P, m_0^P\rangle \to \cdots \to \langle s_1, m_1\rangle$ such that $\overline{m_1}(c_1)$ is* true *and $\overline{m_1}(e_1) = \overline{m_2}(e_2)$.*

Theorem 4 *Assume that for all $\langle i\rangle s \in I$, $I \vdash \langle i\rangle s$ and for all $\langle c_1, e_1\rangle s_1 \lhd \langle c_2, e_2\rangle s_2 \in I$, $I \vdash \langle c_1, e_1\rangle s_1 \lhd \langle c_2, e_2\rangle s_2$. Then for all standard form proofs of $I \vdash \langle c_1, e_1\rangle s_1 \lhd \langle c_2, e_2\rangle s_2$ and for all partial executions $\langle s_0^T, m_0^T\rangle \to \cdots \to \langle s_2, m_2\rangle$ such that $\overline{m_2}(c_2)$ is* true, *there exists a partial execution $\langle s_0^P, m_0^P\rangle \to \cdots \to \langle s_1, m_1\rangle$ such that $\overline{m_1}(c_1)$ is* true *and $\overline{m_1}(e_1) = \overline{m_2}(e_2)$.*

6 Proving Aliasing Invariants

This section describes several methods that use the given logics to prove aliasing invariants. We also present an example that shows a proof of an invariant which is not an aliasing invariant, but depends on the results of the pointer analysis. We start by introducing the following additional notation for points-to relationships. Let p be an arbitrary expression that evaluates to a memory location, v be a variable and $S = \{v_1, \ldots, v_n\}$ be a set of variables. We then define the following notation for points-to relationships:

$$\begin{aligned} p \mapsto v &\overset{def}{=} p = \&v \\ p \mapsto S &\overset{def}{=} p \mapsto v_1 \vee \ldots \vee p \mapsto v_n \\ p \not\mapsto S &\overset{def}{=} \neg(p \mapsto S) \end{aligned}$$

We also extend the substitution rules to the new type of conditions as follows:

$$\frac{p\{e/w\}|c \doteq p'}{p \mapsto S\{e/w\}|c \doteq p' \mapsto S}$$

6.1 Derived Rules

Figure 6 presents the basic rules for substitution in the presence of pointers. To shorten the substitution proofs, we derive several rules from the basic rules. We first add new rules to the logics for substitution, standard invariants and simulation invariants. Figure 11 shows the new rules for rewriting expressions. The logics remain provably sound after adding these rules. Figure 12 presents additional substitution rules. All of these rules are derived from the basic rules for substitution and the new rules for rewriting. There are two forms of each rule: one for $w \equiv v$ and one for $w \equiv *e$. The derivation proofs of these rules are generated in a straightforward

$$\frac{((o_1^P,\ldots,o_n^P) = (o_1^T,\ldots,o_n^T) \wedge c_2) \Rightarrow (c_1 \wedge e_1 = e_2)}{I \vdash \langle c_1, e_1\rangle s_0^P \lhd \langle c_2, e_2\rangle s_0^T} \qquad \textit{base} \qquad (22)$$

$$\frac{\begin{array}{c} I \vdash \langle i_1\rangle s_1, I \vdash \langle i_2\rangle s_2, \langle c_1', e_1'\rangle s_1 \lhd \langle c_2', e_2'\rangle s_2 \in I, \\ (i_2 \wedge c_2) \Rightarrow c_2', (i_1 \wedge i_2 \wedge c_2 \wedge e_1' = e_2') \Rightarrow (c_1 \wedge e_1 = e_2) \end{array}}{I \vdash \langle c_1, e_1\rangle s_1 \lhd \langle c_2, e_2\rangle s_2 \cdot t} \qquad \textit{induction} \qquad (23)$$

Figure 8: Simulation Invariant Base and Induction Proof Rules

$$\frac{\exists s \in \mathrm{pred}(t).\ I \vdash s\langle c, e\rangle t \lhd p}{I \vdash \langle c, e\rangle t \lhd p} \qquad \textit{orig-pred} \qquad (24)$$

$$\frac{s : \mathrm{nop}\ t, I \vdash \langle c, e\rangle s \lhd p}{I \vdash s\langle c, e\rangle t \lhd p} \qquad \textit{orig-nop} \qquad (25)$$

$$\frac{\begin{array}{c} s : w \leftarrow e'\ t, c\{w/e'\}|c_1' \doteq c_1, e\{w/e'\}|c_2' \doteq e_1, \\ I \vdash \langle c_1' \wedge c_2'\rangle s, I \vdash \langle c_1, e_1\rangle s \lhd p \end{array}}{I \vdash s\langle c, e\rangle t \lhd p} \qquad \textit{orig-assign} \qquad (26)$$

$$\frac{s : \mathrm{br}\ c'\ t_1\ t_2, I \vdash \langle c \wedge c', e\rangle s \lhd p}{I \vdash s\langle c, e\rangle t_1 \lhd p} \qquad \textit{orig-brtrue} \qquad (27)$$

$$\frac{s : \mathrm{br}\ c'\ t_1\ t_2, I \vdash \langle c \wedge \neg c', e\rangle s \lhd p}{I \vdash s\langle c, e\rangle t_2 \lhd p} \qquad \textit{orig-brfalse} \qquad (28)$$

Figure 9: Proof Rules for the Original Program P

$$\frac{\mathrm{pred}(t) \neq \emptyset, \forall s \in \mathrm{pred}(t).\ I \vdash p \lhd s\langle c, e\rangle t}{I \vdash p \lhd \langle c, e\rangle t} \qquad \textit{trans-pred} \qquad (29)$$

$$\frac{s : \mathrm{nop}\ t, I \vdash p \lhd \langle c, e\rangle s \cdot t}{I \vdash p \lhd s\langle c, e\rangle t} \qquad \textit{trans-nop} \qquad (30)$$

$$\frac{\begin{array}{c} s : w \leftarrow e'\ t, c\{w/e'\}|c_1' \doteq c_1, e\{w/e'\}|c_2' \doteq e_1, \\ I \vdash \langle c_1' \wedge c_2'\rangle s, I \vdash p \lhd \langle c_1, e_1\rangle s \cdot t \end{array}}{I \vdash p \lhd s\langle c, e\rangle t} \qquad \textit{trans-assign} \qquad (31)$$

$$\frac{s : \mathrm{br}\ c'\ t_1\ t_2, I \vdash p \lhd \langle c \wedge c', e\rangle s \cdot t_1}{I \vdash p \lhd s\langle c, e\rangle t_1} \qquad \textit{trans-brtrue} \qquad (32)$$

$$\frac{s : \mathrm{br}\ c'\ t_1\ t_2, I \vdash p \lhd \langle c \wedge \neg c', e\rangle s \cdot t_2}{I \vdash p \lhd s\langle c, e\rangle t_2} \qquad \textit{trans-brfalse} \qquad (33)$$

$$\frac{I \vdash p \lhd \langle c_1, e\rangle s \cdot t, I \vdash p \lhd \langle c_2, e\rangle s \cdot t, c \Rightarrow c_1 \vee c_2}{I \vdash p \lhd \langle c, e\rangle s \cdot t} \qquad \textit{trans-case} \qquad (34)$$

$$\frac{I \vdash p \lhd \langle c, e\rangle s}{I \vdash p \lhd \langle c, e\rangle s \cdot t} \qquad \textit{trans-step} \qquad (35)$$

Figure 10: Proof Rules for the Transformed Program T

way; an example is shown in Figure 13. We write the inference rules used in the proof trees to the right of the line separating antecedents and consequents. The numbers refer to the presented rules. L stands for logic for proving standard relationships between integers, A means by assumption, and D stands for one of the derived, unlabeled rules.

$$\frac{e_1\{e/w\}|c \doteq e_1'', c \Rightarrow e_1' = e_1''}{e_1\{e/w\}|c \doteq e_1'} \tag{36}$$

$$\frac{e_1\{e/w\}|c' \doteq e_1', c \Rightarrow c'}{e_1\{e/w\}|c \doteq e_1'} \tag{37}$$

$$\frac{I \vdash \langle i' \rangle s, i' \Rightarrow i}{I \vdash \langle i \rangle s} \tag{38}$$

Figure 11: Rules for Rewriting Expressions

6.2 Example

Figure 14 shows a program that we use to illustrate the logics. Observe that, no matter what the condition at node 0 is, $x + y = 8$ at node 7. To find this out, the compiler must perform a relational attributes points-to analysis, which tracks points-to relationships between groups of pointers. At node 5, we have the precise information that $(p \mapsto x \wedge q \mapsto y) \vee (p \mapsto y \wedge q \mapsto x)$. The alternative is to use an independent attributes analysis, which records the points-to relationships for each pointer independently of other pointers. An independent attributes analysis would generate the result that $(p \mapsto x \vee p \mapsto y) \wedge (q \mapsto x \vee q \mapsto y)$ at node 5. Note that this result does not provide enough information to determine that $x + y = 8$ at node 7. Our logics are powerful enough to prove results generated by both kinds of analysis.

Figure 15 presents invariant that a relational attributes analysis generates for this example. The first invariant, $\langle (p \mapsto x \wedge q \mapsto y) \vee (p \mapsto y \wedge q \mapsto x) \rangle 5$, is a pure aliasing invariant. Figures 16 through 18 show the proof tree for this invariant. We omit the proof trees denoted π_3 and π_4; these proof trees are identical to the proof trees denoted π_1 and π_2, respectively, except that x and y are switched. Note that details of some steps involving rewrite rules are omitted from this tree.

The second invariant $\langle x + y = 8 \rangle 7$ does not refer to pointers directly, but the presence of pointers complicates its proof. Figure 22 shows the proof tree for this invariant. In one of the leaves of this tree, we use the aliasing invariant to apply the induction step rule. It is possible to prove $\langle x + y = 8 \rangle 7$ without separating out the aliasing information. However, dividing the analysis into two steps, the first of which deals only with aliasing, is a natural way for the compiler to reason in the presence of pointers.

6.3 Proving Results of Points-To Analyses

In the previous section, we showed how to prove the results of a relational attributes analysis correct. Such a result can be generated with a flow-sensitive (but potentially quite inefficient) dataflow analysis. Independent attributes analyses provide a more efficient but less precise alternative.

In this section we present a general schema for proving results of points-to analyses correct. We first introduce a normal form for writing the aliasing invariants. The normal

$$\frac{c \Rightarrow v_1 \equiv v_2}{v_1\{e/v_2\}|c \doteq e}$$

$$\frac{c \Rightarrow v_1 \not\equiv v_2}{v_1\{e/v_2\}|c \doteq v_1}$$

$$\frac{c \Rightarrow e' \mapsto v}{v\{e/{*e'}\}|c \doteq e}$$

$$\frac{c \Rightarrow e' \not\mapsto v}{v\{e/{*e'}\}|c \doteq v}$$

$$\frac{e_1\{e/v\}|c \doteq e_1', c \Rightarrow e_1' \mapsto v}{{*e_1}\{e/v\}|c \doteq e}$$

$$\frac{e_1\{e/v\}|c \doteq e_1', c \Rightarrow e_1' \not\mapsto v}{{*e_1}\{e/v\}|c \doteq {*e_1'}}$$

$$\frac{e_1\{e/{*e_2}\}|c \doteq e_1', c \Rightarrow (e_1' = e_2 \wedge e_1' \neq 0)}{{*e_1}\{e/{*e_2}\}|c \doteq e}$$

$$\frac{e_1\{e/{*e_2}\}|c \doteq e_1', c \Rightarrow (e_1' \neq e_2 \vee e_1' = 0)}{{*e_1}\{e/{*e_2}\}|c \doteq {*e_1'}}$$

$$\frac{c \Rightarrow v_1 \equiv v_2}{v_1 \mapsto S\{e/v_2\}|c \doteq e \mapsto S}$$

$$\frac{c \Rightarrow v_1 \not\equiv v_2}{v_1 \mapsto S\{e/v_2\}|c \doteq v_1 \mapsto S}$$

$$\frac{c \Rightarrow e' \mapsto v}{v \mapsto S\{e/{*e'}\}|c \doteq e \mapsto S}$$

$$\frac{c \Rightarrow e' \not\mapsto v}{v \mapsto S\{e/{*e'}\}|c \doteq v \mapsto S}$$

Figure 12: Derived Rules for Substitution

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\&v\{e/{*e'}\}|c \doteq \&v}\,4 \quad \dfrac{\dfrac{}{c \Rightarrow e' \not\mapsto v}\,\mathrm{A}}{c \Rightarrow (e' \neq \&v \vee e' = 0)}\,\mathrm{L}}{{*\&v}\{e/{*e'}\}|c \doteq {*\&v}}\,9 \quad \dfrac{}{c \Rightarrow {*\&v} = v}\,\mathrm{L}}{{*\&v}\{e/{*e'}\}|c \doteq v}\,36}{v\{e/{*e'}\}|c \doteq v}\,8$$

Figure 13: Example Proof of Derived Rule

$$\dfrac{\dfrac{\text{true} \Rightarrow p \equiv p}{p \mapsto x\{\&x/p\}|p \equiv p \doteq \&x \mapsto x}\,\text{D} \quad \text{true} \Rightarrow \text{true} = \&x \mapsto x}{p \mapsto x\{\&x/p\}|\text{true} \doteq \text{true}}\,36$$

Figure 16: Proof Tree π_1 for $p \mapsto x\{\&x/p\}|\text{true} \doteq \text{true}$

$$\dfrac{\dfrac{p\mapsto x\{\&y/q\}|\text{true} \doteq p\mapsto x \quad q\mapsto y\{\&y/q\}|\text{true}\doteq\text{true}}{p\mapsto x \wedge q\mapsto y\{\&y/q\}|\text{true}\doteq p\mapsto x}\,7 \quad \dfrac{p\mapsto y\{\&y/q\}|\text{true}\doteq p\mapsto y \quad q\mapsto x\{\&y/q\}|\text{true}\doteq\text{false}}{p\mapsto y\wedge q\mapsto x\{\&y/q\}|\text{true}\doteq\text{false}}\,7}{(p\mapsto x\wedge q\mapsto y)\vee(p\mapsto y\wedge q\mapsto x)\{\&y/q\}|\text{true}\doteq p\mapsto x}\,7$$

Figure 17: Proof Tree π_2 for $(p\mapsto x\wedge q\mapsto y)\vee(p\mapsto y\wedge q\mapsto x)\{\&y/q\}|\text{true}\doteq p\mapsto x$

$$\dfrac{\dfrac{\pi_2 \quad \dfrac{\dfrac{\text{true}}{I\vdash\langle\text{true}\rangle 2}\,21}{I\vdash\langle\text{true}\rangle 2\cdot 5}\,14 \quad \dfrac{\dfrac{\dfrac{\pi_1 \quad I\vdash\langle\text{true}\rangle 1\cdot 2 \quad I\vdash\langle\text{true}\rangle 1\cdot 2}{I\vdash 1\langle p\mapsto x\rangle 2}\,16}{I\vdash\langle p\mapsto x\rangle 2}\,14}{I\vdash\langle p\mapsto x\rangle 2\cdot 5}\,19}{I\vdash 2\langle(p\mapsto x\wedge q\mapsto y)\vee(p\mapsto y\wedge q\mapsto x)\rangle 5}\,16 \quad \dfrac{\pi_4 \quad \dfrac{\dfrac{\text{true}}{I\vdash\langle\text{true}\rangle 4}\,21}{I\vdash\langle\text{true}\rangle 4\cdot 5}\,14 \quad \dfrac{\dfrac{\dfrac{\pi_3 \quad I\vdash\langle\text{true}\rangle 3\cdot 4 \quad I\vdash\langle\text{true}\rangle 3\cdot 4}{I\vdash 3\langle p\mapsto y\rangle 4}\,16}{I\vdash\langle p\mapsto y\rangle 4}\,14}{I\vdash\langle p\mapsto y\rangle 4\cdot 5}\,19}{I\vdash 4\langle(p\mapsto x\wedge q\mapsto y)\vee(p\mapsto y\wedge q\mapsto x)\rangle 5}\,16}{I\vdash\langle(p\mapsto x\wedge q\mapsto y)\vee(p\mapsto y\wedge q\mapsto x)\rangle 5}\,14$$

Figure 18: Proof Tree for $I\vdash\langle(p\mapsto x\wedge q\mapsto y)\vee(p\mapsto y\wedge q\mapsto x)\rangle 5$

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}p\mapsto x\{3/*q\}\\|q\mapsto x\doteq p\mapsto x\end{array} \quad \begin{array}{c}y=3\{3/*q\}\\|q\mapsto x\doteq y=3\end{array}}{\begin{array}{c}p\mapsto x?y=3\{3/*q\}\\|q\mapsto x\doteq p\mapsto x?y=3\end{array}} \quad \dfrac{\begin{array}{c}p\mapsto y\{3/*q\}\\|q\mapsto x\doteq p\mapsto y\end{array} \quad \begin{array}{c}x=3\{3/*q\}\\|q\mapsto x\doteq \text{true}\end{array}}{\begin{array}{c}p\mapsto y?x=3\{3/*q\}\\|q\mapsto x\doteq p\mapsto y?\text{true}\end{array}}}{\begin{array}{c}p\mapsto x?y=3;p\mapsto y?x=3\{3/*q\}\\|q\mapsto x\doteq p\mapsto x?y=3;p\mapsto y?\text{true}\end{array}} \quad \dfrac{\dfrac{\begin{array}{c}p\mapsto x\{3/*q\}\\|q\mapsto y\doteq p\mapsto x\end{array} \quad \begin{array}{c}y=3\{3/*q\}\\|q\mapsto y\doteq \text{true}\end{array}}{\begin{array}{c}p\mapsto x?y=3\{3/*q\}\\|q\mapsto y\doteq p\mapsto x?\text{true}\end{array}} \quad \dfrac{\begin{array}{c}p\mapsto y\{3/*q\}\\|q\mapsto y\doteq p\mapsto y\end{array} \quad \begin{array}{c}x=3\{3/*q\}\\|q\mapsto y\doteq x=3\end{array}}{\begin{array}{c}p\mapsto y?x=3\{3/*q\}\\|q\mapsto y\doteq p\mapsto y?x=3\end{array}}\,12}{\begin{array}{c}p\mapsto x?y=3;p\mapsto y?x=3\{3/*q\}\\|q\mapsto y\doteq p\mapsto x?\text{true};p\mapsto y?x=3\end{array}}\,13}{p\mapsto x?y=3;p\mapsto y?x=3\{3/*q\}|q\mapsto x\vee q\mapsto y\doteq q\mapsto x?(p\mapsto x?y=3;p\mapsto y?\text{true});q\mapsto y?(p\mapsto x?\text{true};p\mapsto y?x=3}\,11$$

Figure 19: Proof Tree π_1 for $p\mapsto x?y=3;p\mapsto y?x=3\{3/*q\}|q\mapsto y\vee q\mapsto x\doteq c$

$$\dfrac{\dfrac{\dfrac{x\{5/*p\}|p\mapsto x\doteq 5 \quad y\{5/*p\}|p\mapsto x\doteq y}{x+y\{5/*p\}|p\mapsto x\doteq 5+y}\,7 \quad 8\{5/*p\}|p\mapsto x\doteq 8}{x+y=8\{5/*p\}|p\mapsto x\doteq y=3}\,7 \quad \dfrac{\dfrac{x\{5/*p\}|p\mapsto y\doteq x \quad y\{5/*p\}|p\mapsto y\doteq 5}{x+y\{5/*p\}|p\mapsto y\doteq x+5}\,7 \quad 8\{5/*p\}|p\mapsto y\doteq 8}{x+y=8\{5/*p\}|p\mapsto y\doteq x=3}\,7}{x+y=8\{5/*p\}|p\mapsto x\vee p\mapsto y\doteq p\mapsto x?y=3;p\mapsto y?x=3}\,11$$

Figure 20: Proof Tree π_2 for $x+y=8\{5/*p\}|p\mapsto x\vee p\mapsto y\doteq p\mapsto x?y=3;p\mapsto y?x=3$

$$\dfrac{\dfrac{\dfrac{p\mapsto x\vee p\mapsto y\{3/*q\}|\text{true}\doteq p\mapsto x\vee p\mapsto y \quad I\vdash\langle\text{true}\rangle 5\cdot 6 \quad \dfrac{\begin{array}{c}\langle(p\mapsto x\wedge q\mapsto y)\vee(p\mapsto y\wedge q\mapsto x)\rangle 5\in I,\\((p\mapsto x\wedge q\mapsto y)\vee(p\mapsto y\wedge q\mapsto x))\Rightarrow p\mapsto x\vee p\mapsto y\end{array}}{I\vdash\langle p\mapsto x\vee p\mapsto y\rangle 5\cdot 6}\,20}{I\vdash 5\langle p\mapsto x\vee p\mapsto y\rangle 6}\,16}{I\vdash\langle p\mapsto x\vee p\mapsto y\rangle 6}\,14}{I\vdash\langle p\mapsto x\vee p\mapsto y\rangle 6\cdot 7}\,19$$

Figure 21: Proof Tree π_3 for $I\vdash\langle p\mapsto x\vee p\mapsto y\rangle 6\cdot 7$

$$\dfrac{\dfrac{\pi_2 \quad \pi_3 \quad \dfrac{\dfrac{\dfrac{\pi_1 \quad \dfrac{\begin{array}{c}\langle(p\mapsto x\wedge q\mapsto y)\vee(p\mapsto y\wedge q\mapsto x)\rangle 5\in I,\\((p\mapsto x\wedge q\mapsto y)\vee(p\mapsto y\wedge q\mapsto x))\Rightarrow\\q\mapsto x\vee q\mapsto y\end{array}}{I\vdash\langle q\mapsto x\vee q\mapsto y\rangle 5\cdot 6} \quad \dfrac{\begin{array}{c}\langle(p\mapsto x\wedge q\mapsto y)\vee(p\mapsto y\wedge q\mapsto x)\rangle 5\in I,\\((p\mapsto x\wedge q\mapsto y)\vee(p\mapsto y\wedge q\mapsto x))\Rightarrow\\q\mapsto x?(p\mapsto x?y=3;p\mapsto y?\text{true});q\mapsto y?(p\mapsto x?\text{true};p\mapsto y?x=3)\end{array}}{I\vdash\langle q\mapsto x?(p\mapsto x?y=3;p\mapsto y?\text{true});q\mapsto y?(p\mapsto x?\text{true};p\mapsto y?x=3)\rangle 5\cdot 6}}{I\vdash 5\langle p\mapsto x?y=3;p\mapsto y?x=3\rangle 6}\,20}{I\vdash\langle p\mapsto x?y=3;p\mapsto y?x=3\rangle 6}\,14}{I\vdash\langle p\mapsto x?y=3;p\mapsto y?x=3\rangle 6\cdot 7}\,19}{I\vdash 6\langle x+y=8\rangle 7}\,16}{I\vdash\langle x+y=8\rangle 7}\,14$$

Figure 22: Proof Tree for $I\vdash\langle x+y=8\rangle 7$

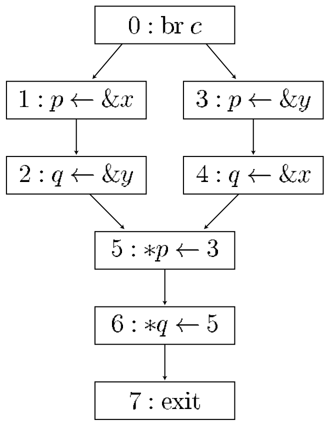

Figure 14: Example Program with Pointers

$I{=}\{\langle (p \mapsto x \wedge q \mapsto y) \vee (p \mapsto y \wedge q \mapsto x)\rangle 5, \langle x + y = 8\rangle 7\}$

Figure 15: Invariants for Example Program with Pointers

form is defined by the following grammar:

$$n ::= v \mapsto S | \&v \mapsto S | v \equiv v' | \text{true} | \text{false} | \neg n | n \wedge n | n \vee n | n?n; n?n$$

The key property of this form is the absence of explicit dereferencing — all invariants state the points-to relationships using expressions of the form $v \mapsto S$.

To simplify the proof rules, we assume that the control-flow graph has been converted to a simplified form in which all pointer assignments are of one of the following four basic assignment types:

$v' \leftarrow \&v''$	(address-of assign)
$v' \leftarrow v''$	(copy assign)
$v' \leftarrow *v''$	(load assign)
$*v' \leftarrow v''$	(store assign)

More complex assignments can be transformed, using additional temporary variables, into sequences of these basic assignments.

Figure 23 presents the proof rules for proving points-to invariants for the four basic assignment types. The primary issue is ensuring that all of the points-to invariants are maintained in normal form during substitution. The potential problem occurs when substituting an expression of the form $*v''$ in for a variable v in $v \mapsto S$. The standard substitution would generate an invariant with dereferencing. The rules avoid this outcome by constructing two levels of points-to expressions. The first level is of the form $v'' \mapsto \{v_1, \ldots, v_n\}$ and specifies all the variables to which v'' may point. The next level specifies, for each variable v_i, the variables to which v_i may point. Figure 24 presents sketches of the derivations of the rules in Figure 23.

6.3.1 Flow-Insensitive Analyses

The flow-sensitive analyses discussed so far produce an analysis result for each point in the program. Flow-insensitive analyses such as Steensgaard's or Andersen's analysis [10, 1], on the other hand, produce a single analysis result that is valid at all points in the program. Flow-insensitive analyses are typically more efficient than flow-sensitive analyses, but provide less precise results.

It is possible to use the rules from the last section to prove the results of flow-insensitive pointer analyses correct, but the correctness proofs are longer than they need to be. This section presents a set of derived rules that can be used to shorten the correctness proofs. In both Steensgaard's and Andersen's analysis, the result can be represented using the following subset of the normal form:

$$n' ::= v \mapsto S | \&v \not\mapsto S | v \equiv v' | v \not\equiv v' | \text{true} | \text{false} | n' \wedge n'$$

The difference from the full normal form is that there is no explicit disjunction, i.e. we have either a condition that evaluates to true or false or a conjunct of the form $v_1 \rightarrow S_1 \wedge \ldots \wedge v_n \mapsto S_n$. The implicit disjunction is present in $v \mapsto S$. Note that the results of a flow-sensitive, but independent attributes analysis, can also be written in this form.

We first show one alternative for flow-insensitive analyses. The main idea is that since a flow-insensitive result is the same at all program points, each program point can have the same invariant i as its analysis result. We then use induction on the structure of the control flow graph to derive the following rule:

$$\frac{I \vdash \langle i\rangle s_0, \forall s' : w \leftarrow e\ t.((i\{e/w\}|c \doteq i'), i \Rightarrow c, i \Rightarrow i')}{I \vdash \langle i\rangle s}$$

This rule states that an invariant i holds at every program point s if it holds at the entry node s_0 and all assignment nodes preserve the invariant. To prove that an invariant i holds at the entry node, we use the following rule:

$$\frac{i\{v_{1,0}, v_{2,0}, \ldots, v_{n,0}/v_1, v_2, \ldots, v_n\}|c \doteq i', c, i'}{I \vdash \langle i\rangle s_0}$$

This rule allows us to simultaneously substitute all the observable variables with their initial values, and then show that the condition generated is provably true.

The two previous rules are an extension to the logic for the standard invariants, and they do not affect the substitution logic. We next present several substitution rules that are specialized for proving results of flow-insensitive pointer analyses. Figure 25 presents the rules and Figure 26 shows derivations of some of the rules.

The last rule in Figure 25 is designed to eliminate case analysis for the kind of invariants that Steensgaard's analysis produces. More specifically, for an assignment of the form $*v' \leftarrow v''$ in a context in which v' may point to v, Steensgaard's analysis produces a result in which v and v'' point to the same set of variables.

For Andersen's analysis, we need the following meta rule involving set inclusion for the last rule in Figure 25 to be generally useful.

$$\frac{p \mapsto S', S' \subseteq S}{p \mapsto S}$$

Note that because the proof rules always produce sets S that consist of a finite set of variables, it is always possible for the proof checker to determine set inclusion by examining the representations of the sets.

We next discuss a problem that arises in proving the results of flow-insensitive analyses. The main issue is that flow-insensitive analyses generate the same result for all program points, including the initial program point. The analysis result must therefore include an initial value for all pointers. These initial values introduce the following problem. According to the analysis result, any pointer assignment may access the initial value of the pointer. But it is not

$$\frac{}{v \mapsto S\{\&v''/v'\}|c \doteq v' \equiv v?\&v'' \mapsto S : v \mapsto S}$$

$$\frac{}{v \mapsto S\{v''/v'\}|c \doteq v' \equiv v?v'' \mapsto S : v \mapsto S}$$

$$A \overset{def}{=} v'' \mapsto \{v_1, \ldots, v_n\} \wedge ((v'' \mapsto v_1 \wedge v_1 \mapsto S) \vee \ldots \vee (v'' \mapsto v_n \wedge v_n \mapsto S))$$
$$\frac{c \Rightarrow A}{v \mapsto S\{*v''/v'\}|c \doteq A \wedge v' \equiv v?\text{true}; v' \not\equiv v?v \mapsto S}$$

$$\frac{}{v \mapsto S\{v''/*v'\}|c \doteq v' \mapsto v?v'' \mapsto S : v \mapsto S}$$

$$\frac{c \Rightarrow (v' \mapsto S' \wedge \&v \not\mapsto S')}{v \mapsto S\{v''/*v'\}|c \doteq v \mapsto S}$$

$$\frac{c \Rightarrow v'' \mapsto S}{v \mapsto S\{v''/*v'\}|c \doteq v' \mapsto v?\text{true} : v \mapsto S}$$

Figure 23: Basic Pointer Assignment Rules

$$\frac{v \mapsto S\{\&v''/v'\}|v' \equiv v \doteq \&v'' \mapsto S, v \mapsto S\{\&v''/v'\}|v' \not\equiv v \doteq v \mapsto S, c \Rightarrow (v' \equiv v \vee v' \not\equiv v)}{v \mapsto S\{\&v''/v'\}|c \doteq v' \equiv v?\&v'' \mapsto S : v \mapsto S}$$

$$\frac{v \mapsto S\{v''/v'\}|v' \equiv v \doteq v'' \mapsto S, v \mapsto S\{v''/v'\}|v' \not\equiv v \doteq v \mapsto S, c \Rightarrow (v' \equiv v \vee v' \not\equiv v)}{v \mapsto S\{v''/v'\}|c \doteq v' \equiv v?v'' \mapsto S : v \mapsto S}$$

$$A \overset{def}{=} v'' \mapsto \{v_1, \ldots, v_n\} \wedge ((v'' \mapsto v_1 \wedge v_1 \mapsto S) \vee \ldots \vee (v'' \mapsto v_n \wedge v_n \mapsto S))$$
$$\frac{v \mapsto S\{*v''/v'\}|A \wedge v' \equiv v \doteq *v'' \mapsto S, v \mapsto S\{*v''/v'\}|v' \not\equiv v \doteq v \mapsto S, A \Rightarrow ((A \wedge v' \equiv v) \vee v' \not\equiv v), (A \wedge v' \equiv v) \Rightarrow *v'' \mapsto S = true}{v \mapsto S\{*v''/v'\}|A \doteq A \wedge v' \equiv v?\text{true}; v' \not\equiv v?v \mapsto S}$$

$$\frac{v \mapsto S\{v''/*v'\}|v' \mapsto v \doteq v'' \mapsto S, v \mapsto S\{v''/*v'\}|v' \not\mapsto v \doteq v \mapsto S, c \Rightarrow (v' \mapsto v \vee v' \not\mapsto v)}{v \mapsto S\{v''/*v'\}|c \doteq v' \mapsto v?v'' \mapsto S : v \mapsto S}$$

$$\frac{v \mapsto S\{v''/*v'\}|v' \not\mapsto v \doteq v \mapsto S, (v' \mapsto S' \wedge \&v \not\mapsto S') \Rightarrow v' \not\mapsto v}{v \mapsto S\{v''/*v'\}|v' \mapsto S' \wedge \&v \not\mapsto S' \doteq v \mapsto S}$$

$$\frac{\begin{array}{c} v \mapsto S\{v''/*v'\}|v' \mapsto v \wedge v'' \mapsto S \doteq v'' \mapsto S, v \mapsto S\{v''/*v'\}|v' \not\mapsto v \doteq v \mapsto S, \\ v'' \mapsto S \Rightarrow ((v'' \mapsto S \wedge v' \mapsto v) \vee v' \not\mapsto v), (v' \mapsto v \wedge v'' \mapsto S) \Rightarrow v'' \mapsto S = \text{true}, \\ v'' \mapsto S \Rightarrow (v' \mapsto v \wedge v'' \mapsto S?\text{true}; v' \mapsto v?v \mapsto S) = (v' \mapsto v?\text{true} : v \mapsto S) \end{array}}{v \mapsto S\{v''/*v'\}|v'' \mapsto S \doteq v' \mapsto v?\text{true} : v \mapsto S)}$$

Figure 24: Derivations of Basic Pointer Assignment Rules

$$\frac{c \Rightarrow v' \not\equiv v}{v \mapsto S\{\&v''/v'\}|c \doteq v \mapsto S}$$

$$\frac{c \Rightarrow v' \equiv v}{v \mapsto S\{\&v''/v'\}|c \doteq \&v'' \mapsto S}$$

$$\frac{c \Rightarrow v' \not\equiv v}{v \mapsto S\{v''/v'\}|c \doteq v \mapsto S}$$

$$\frac{c \Rightarrow v' \equiv v}{v \mapsto S\{v''/v'\}|c \doteq v'' \mapsto S}$$

$$\frac{c \Rightarrow v' \not\equiv v}{v \mapsto S\{*v''/v'\}|c \doteq v \mapsto S}$$

$$A' \stackrel{def}{=} v'' \mapsto \{v_1, \ldots, v_n\} \wedge v_1 \mapsto S \wedge \ldots \wedge v_n \mapsto S$$
$$\frac{c \Rightarrow v' \equiv v \wedge A'}{v \mapsto S\{*v''/v'\}|c \doteq \text{true}}$$

$$\frac{c \Rightarrow v' \mapsto S' \wedge \&v \not\mapsto S' \wedge v \mapsto S}{v \mapsto S\{v''/*v'\}|c \doteq \text{true}}$$

$$\frac{c \Rightarrow v \mapsto S \wedge v'' \mapsto S}{v \mapsto S\{v''/*v'\}|c \doteq \text{true}}$$

Figure 25: Rules for Flow-Insensitive Pointer Analyses

$$A' \stackrel{def}{=} v'' \mapsto \{v_1, \ldots, v_n\} \wedge v_1 \mapsto S \wedge \ldots \wedge v_n \mapsto S$$
$$\dfrac{\dfrac{v \mapsto S\{*v''/v'\}|v' \equiv v \doteq *v'' \mapsto S \quad v' \equiv v \wedge (A' \Rightarrow v' \equiv v)}{v \mapsto S\{*v''/v'\}|v' \equiv v \wedge A' \doteq *v'' \mapsto S}{}_{37} \quad (v' \equiv v \wedge A') \Rightarrow *v'' \mapsto S = \text{true}}{v \mapsto S\{*v''/v'\}|v' \equiv v \wedge A' \doteq \text{true}}{}_{36}$$

$$\dfrac{\dfrac{v \mapsto S\{v''/*v'\}|v' \not\mapsto v \doteq v \mapsto S \quad (v' \mapsto S' \wedge \&v \not\mapsto S' \wedge v \mapsto S) \Rightarrow v' \not\mapsto v}{v \mapsto S\{v''/*v'\}|v' \mapsto S' \wedge \&v \not\mapsto S' \wedge v \mapsto S \doteq v \mapsto S}{}_{37} \quad (v' \mapsto S' \wedge \&v \not\mapsto S' \wedge v \mapsto S) \Rightarrow v \mapsto S = \text{true}}{v \mapsto S\{v''/*v'\}|v' \mapsto S' \wedge \&v \not\mapsto S' \wedge v \mapsto S \doteq \text{true}}{}_{36}$$

$$\dfrac{\dfrac{\dfrac{v \mapsto S\{v''/*v'\}|v' \not\mapsto v \doteq v \mapsto S \quad v \mapsto S\{v''/*v'\}|v' \mapsto v \doteq v'' \mapsto S}{v \mapsto S\{v''/*v'\}|v' \not\mapsto v \vee v' \mapsto v \doteq v' \not\mapsto v ? v \mapsto S : v'' \mapsto S}{}_{11} \quad \begin{array}{c}(v \mapsto S \wedge v'' \mapsto S) \Rightarrow \\ v' \not\mapsto v \vee v' \mapsto v\end{array}}{v \mapsto S\{v''/*v'\}|v \mapsto S \wedge v'' \mapsto S \doteq v' \not\mapsto v ? v \mapsto S : v'' \mapsto S}{}_{37} \quad \begin{array}{c}(v \mapsto S \wedge v'' \mapsto S) \Rightarrow \\ (v' \not\mapsto v ? v \mapsto S : v'' \mapsto S) = \text{true}\end{array}}{v \mapsto S\{v''/*v'\}|v \mapsto S \wedge v'' \mapsto S \doteq \text{true}}{}_{36}$$

Figure 26: Derivations of Rules for Flow-Insensitive Pointer Analyses

clear where this initial value points. Steensgaard's and Andersen's analyses both assume that either the initial value will never be dereferenced, or that dereferencing the initial value will never affect any other value in the program. In the absence of any information about the effect of dereferencing uninitialized pointers, these analyses only provide information about where pointers *may* point, not information information about where pointers *may not* point. But to prove an analysis result, our rules require information about where pointers *may not* point.

One way to eliminate this problem is to exploit the special behavior of the memory location 0. Namely, a read from 0 always returns 0, and writes to 0 do not change the content of the location. It is actually practical to deliver an efficient implementation on a real machine with this meaning. If each variable is initialized to 0, we can obtain the required information about where variables may not point by adding the possibility that each pointer points to the location 0. If we extend the definition of S to allow $0 \in S$ by defining $v \mapsto 0 \overset{def}{=} v = 0$ and $0 \mapsto S \overset{def}{=} 0 \in S$, we can use the proof rules presented above to verify the results of Steensgaard's and Andersen's analyses. This extension also enables the verification of flow-sensitive analysis results for programs that may dereference pointers that are not explicitly initialized.

7 Optimization Schemas

We next present examples that illustrate how to prove the correctness of a variety of standard optimizations.[3] Our goal is to establish a general schema for each optimization. The compiler would then use the schema to produce a correctness proof that goes along with each optimization.

7.1 Constant Propagation and Constant Folding

If a variable always has a constant value at a given program point, the compiler can replace the variable with the constant at that point. This transformation is called constant propagation. Constant propagation may allow the compiler to perform additional expression simplifications; these transformations are called constant folding. Figures 27 and 28 present an example that we use to illustrate the schema. This example continues the example introduced in Section 4. Figure 29 presents the invariants that the compiler generates for this example.

The key invariant that enables these optimizations is $\langle x_P = 1 \wedge y_P = 2\rangle 5_P$. It can be proven using the logic for standard invariants. Figure 30 presents the proof tree for simulation invariant $\langle(g_P, i_P)\rangle 6_P \lhd \langle(g_T, i_T)\rangle 6_T$. At the leaf of this proof tree, standard invariant $\langle x_P = 1 \wedge y_P = 2\rangle 5_P$ is used in the application of the induction step rule, rule 23.

7.2 Dead Assignment Elimination

The compiler can eliminate an assignment to a local variable if that variable is not used after the assignment. The proof schema is relatively simple: the compiler simply generates simulation invariants that assert the equality of corresponding live variables at corresponding points in the program. Figures 31 and 32 present an example that we use to illustrate the schema. Figure 33 presents the invariants that the compiler generates for this example.

[3] Because none of the examples in this section use pointers, we use direct expression substitution rather than substitution with partial conditioned expressions.

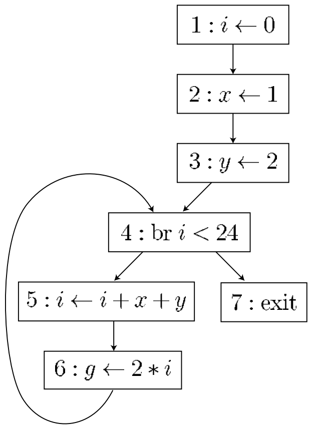

Figure 27: Original Program P

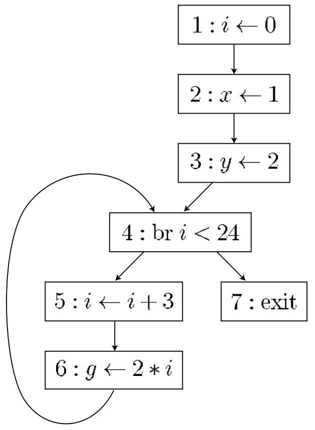

Figure 28: Program T After Constant Propagation and Constant Folding

$$I = \{\langle x_P = 1 \wedge y_P = 2\rangle 5_P, \langle(g_P, i_P)\rangle 5_P \lhd \langle(g_T, i_T)\rangle 5_T, \\ \langle(g_P, i_P)\rangle 6_P \lhd \langle(g_T, i_T)\rangle 6_T, \langle g_P\rangle 7_P \lhd \langle g_T\rangle 7_T\}$$

Figure 29: Invariants for Constant Propagation and Constant Folding

$$\begin{array}{cl}
I \vdash \langle x_P = 1 \wedge y_P = 2\rangle 5_P, \langle(g_P, i_P)\rangle 5_P \lhd \langle(g_T, i_T)\rangle 5_T \in I, & \\
x_P = 1 \wedge y_P = 2 \wedge (g_P, i_P) = (g_T, i_T) \Rightarrow & \\
(g_P, i_P + x_P + y_P) = (g_T, i_T + 3) & \\
\hline
I \vdash \langle(g_P, i_P + x_P + y_P)\rangle 5_P \lhd \langle(g_T, i_T + 3)\rangle 5_T \cdot 6_T & 23 \\
\hline
I \vdash 5_P\langle(g_P, i_P)\rangle 6_P \lhd \langle(g_T, i_T + 3)\rangle 5_T \cdot 6_T & 26 \\
\hline
I \vdash \langle(g_P, i_P)\rangle 6_P \lhd \langle(g_T, i_T + 3)\rangle 5_T \cdot 6_T & 24 \\
\hline
I \vdash \langle(g_P, i_P)\rangle 6_P \lhd 5_T\langle(g_T, i_T)\rangle 6_T & 31 \\
\hline
I \vdash \langle(g_P, i_P)\rangle 6_P \lhd \langle(g_T, i_T)\rangle 6_T & 29
\end{array}$$

Figure 30: Proof Tree for $I \vdash \langle(g_P, i_P)\rangle 6_P \lhd \langle(g_T, i_T)\rangle 6_T$

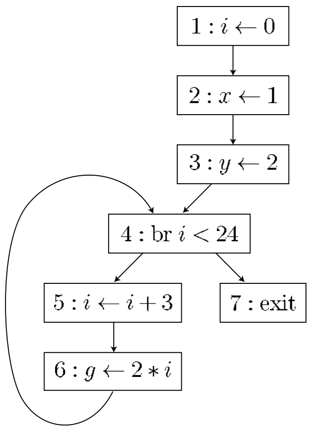

Figure 31: Program P Before Dead Assignment Elimination

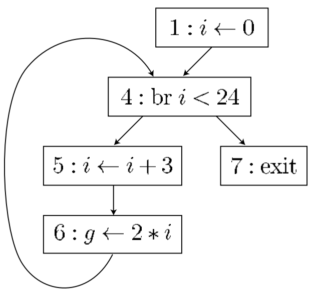

Figure 32: Program T After Dead Assignment Elimination

$$I = \{\langle(g_P, i_P)\rangle 4_P \lhd \langle(g_T, i_T)\rangle 4_T, \langle i_P\rangle 5_P \lhd \langle i_T\rangle 5_T, \\ \langle i_P\rangle 6_P \lhd \langle i_T\rangle 6_T, \langle g_P\rangle 7_P \lhd \langle g_T\rangle 7_T\}$$

Figure 33: Invariants for Dead Assignment Elimination

Note that the set I of invariants contains no standard invariants. In general, dead assignment elimination requires only simulation invariants. The proofs of these invariants are simple; the only complication is the need to skip over dead assignments. Figure 36, which contains the proof tree for $\langle (g_P, i_P)\rangle 4_P \lhd \langle (g_T, i_T)\rangle 4_T$, illustrates this situation. Figures 34 and 35 present the subtrees of the proof.

$$\begin{array}{cl}
(g_P) = (g_T) \Rightarrow (g_P, 0) = (g_T, 0) & \\ \hline
I \vdash \langle (g_P, 0)\rangle 1_P \lhd \langle (g_T, 0)\rangle 1_T & {}^{22} \\ \hline
I \vdash 1_P\langle (g_P, i_P)\rangle 2_P \lhd \langle (g_T, 0)\rangle 1_T & {}^{26} \\ \hline
I \vdash \langle (g_P, i_P)\rangle 2_P \lhd \langle (g_T, 0)\rangle 1_T & {}^{24} \\ \hline
I \vdash 2_P\langle (g_P, i_P)\rangle 3_P \lhd \langle (g_T, 0)\rangle 1_T & {}^{26} \\ \hline
I \vdash \langle (g_P, i_P)\rangle 3_P \lhd \langle (g_T, 0)\rangle 1_T & {}^{24} \\ \hline
I \vdash 3_P\langle (g_P, i_P)\rangle 4_P \lhd \langle (g_T, 0)\rangle 1_T & {}^{26} \\ \hline
I \vdash \langle (g_P, i_P)\rangle 4_P \lhd \langle (g_T, 0)\rangle 1_T & {}^{24} \\ \hline
I \vdash \langle (g_P, i_P)\rangle 4_P \lhd \langle (g_T, 0)\rangle 1_T \cdot 4_T & {}^{35} \\ \hline
I \vdash \langle (g_P, i_P)\rangle 4_P \lhd 1_T\langle (g_T, i_T)\rangle 4_T & {}^{31}
\end{array}$$

Figure 34: Proof Tree π_1 for $I \vdash \langle (g_P, i_P)\rangle 4_P \lhd 1_T\langle (g_T, i_T)\rangle 4_T$

$$\begin{array}{cl}
\langle i_P\rangle 6_P \lhd \langle i_T\rangle 6_T \in I, & \\
i_P = i_T \Rightarrow (2 * i_P, i_P) = (2 * i_T, i_T) & \\ \hline
I \vdash \langle (2 * i_P, i_P)\rangle 6_P \lhd \langle (2 * i_T, i_T)\rangle 6_T \cdot 4_T & {}^{23} \\ \hline
I \vdash 6_P\langle (g_P, i_P)\rangle 4_P \lhd \langle (2 * i_T, i_T)\rangle 6_T \cdot 4_T & {}^{26} \\ \hline
I \vdash \langle (g_P, i_P)\rangle 4_P \lhd \langle (2 * i_T, i_T)\rangle 6_T \cdot 4_T & {}^{24} \\ \hline
I \vdash \langle (g_P, i_P)\rangle 4_P \lhd 6_T\langle (g_T, i_T)\rangle 4_T & {}^{31}
\end{array}$$

Figure 35: Proof Tree π_2 for $I \vdash \langle (g_P, i_P)\rangle 4_P \lhd 6_T\langle (g_T, i_T)\rangle 4_T$

$$\begin{array}{cl}
\pi_1 \quad \pi_2 & \\ \hline
I \vdash \langle (g_P, i_P)\rangle 4_P \lhd \langle (g_T, i_T)\rangle 4_T & {}^{29}
\end{array}$$

Figure 36: Proof Tree for $I \vdash \langle (g_P, i_P)\rangle 4_P \lhd \langle (g_T, i_T)\rangle 4_T$

7.3 Branch Movement

Our next optimization moves a conditional branch from the top of a loop to the bottom. The optimization is legal if the loop always executes at least once. This optimization is different from all the other optimizations we have discussed so far in that it changes the control flow. Figure 37 presents the program before branch movement; Figure 38 presents the program after branch movement. Figure 39 presents the set of invariants that the compiler generates for this example.

Figure 42 presents the proof tree for $I \vdash \langle i_P\rangle 5_P \lhd \langle i_T\rangle 5_T$. One of the paths that the proof must consider is the path in the transformed program T from 1_T to 5_T. The corresponding path in P that is used to prove $I \vdash \langle i_P\rangle 5_P \lhd \langle i_T\rangle 5_T$ is the path from 1_P through 4_P to 5_P. The fact that the loop executes at least once shows up as a true condition in the partial simulation invariant for P that is propagated from 5_P back to 1_P. This enables the use of rule 22 at the leaf of the proof tree. Figure 41 presents the branch of the proof tree for this path.

7.4 Induction Variable Elimination

Our next optimization eliminates the induction variable i from the loop, replacing it with g. The correctness of this transformation depends on the invariant $\langle g_P = 2 * i_P\rangle 4_P$. Figure 43 presents the program before induction variable

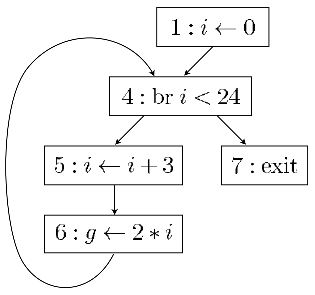

Figure 37: Program P Before Branch Movement

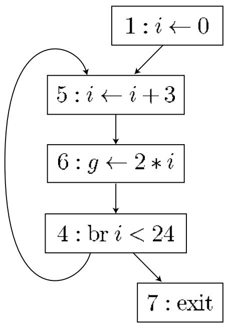

Figure 38: Program T After Branch Movement

$$I = \{\langle i_P\rangle 5_P \lhd \langle i_T\rangle 5_T, \langle i_P\rangle 6_P \lhd \langle i_T\rangle 6_T, \langle g_P\rangle 7_P \lhd \langle g_T\rangle 7_T\}$$

Figure 39: Invariants for Branch Movement

$$\begin{array}{cl}
\langle i_P\rangle 6_P \lhd \langle i_T\rangle 6_T \in I, & \\
i_T < 24 \land i_P = i_T \Rightarrow (i_P < 24 \land i_P = i_T) & \\ \hline
I \vdash \langle i_P < 24, i_P\rangle 6_P \lhd \langle i_T < 24, i_T\rangle 6_T \cdot 4_T & {}^{23} \\ \hline
I \vdash 6_P\langle i_P < 24, i_P\rangle 4_P \lhd \langle i_T < 24, i_T\rangle 6_T \cdot 4_T & {}^{26} \\ \hline
I \vdash \langle i_P < 24, i_P\rangle 4_P \lhd \langle i_T < 24, i_T\rangle 6_T \cdot 4_T & {}^{24} \\ \hline
I \vdash 4_P\langle i_P\rangle 5_P \lhd \langle i_T < 24, i_T\rangle 6_T \cdot 4_T & {}^{27} \\ \hline
I \vdash \langle i_P\rangle 5_P \lhd \langle i_T < 24, i_T\rangle 6_T \cdot 4_T & {}^{24} \\ \hline
I \vdash \langle i_P\rangle 5_P \lhd 6_T\langle i_T < 24, i_T\rangle 4_T & {}^{31} \\ \hline
I \vdash \langle i_P\rangle 5_P \lhd \langle i_T < 24, i_T\rangle 4_T & {}^{29} \\ \hline
I \vdash \langle i_P\rangle 5_P \lhd \langle i_T < 24, i_T\rangle 4_T \cdot 5_T & {}^{35} \\ \hline
I \vdash \langle i_P\rangle 5_P \lhd 4_T\langle i_T\rangle 5_T & {}^{32}
\end{array}$$

Figure 40: Proof Tree π_1 for $I \vdash \langle g_P\rangle 5_P \lhd 4_T\langle g_T\rangle 5_T$

$$\begin{array}{cl}
(g_P) = (g_T) \Rightarrow 0 < 24 \land 0 = 0 & \\ \hline
I \vdash \langle 0 < 24, 0\rangle 1_P \lhd \langle 0\rangle 1_T & {}^{22} \\ \hline
I \vdash 1_P\langle i_P < 24, i_P\rangle 4_P \lhd \langle 0\rangle 1_T & {}^{26} \\ \hline
I \vdash \langle i_P < 24, i_P\rangle 4_P \lhd \langle 0\rangle 1_T & {}^{24} \\ \hline
I \vdash 4_P\langle i_P\rangle 5_P \lhd \langle 0\rangle 1_T & {}^{27} \\ \hline
I \vdash \langle i_P\rangle 5_P \lhd \langle 0\rangle 1_T & {}^{24} \\ \hline
I \vdash \langle i_P\rangle 5_P \lhd \langle 0\rangle 1_T \cdot 5_T & {}^{35} \\ \hline
I \vdash \langle i_P\rangle 5_P \lhd 1_T\langle i_T\rangle 5_T & {}^{31}
\end{array}$$

Figure 41: Proof Tree π_2 for $I \vdash \langle g_P\rangle 5_P \lhd 1_T\langle g_T\rangle 5_T$

$$\begin{array}{cl}
\pi_1 \quad \pi_2 & \\ \hline
I \vdash \langle i_P\rangle 5_P \lhd \langle i_T\rangle 5_T & {}^{29}
\end{array}$$

Figure 42: Proof Tree for $I \vdash \langle g_P\rangle 5_P \lhd \langle g_T\rangle 5_T$

elimination; Figure 44 presents the program after induction variable elimination. Figure 45 presents the set of invariants that the compiler generates for this example. These invariants characterize the relationship between the eliminated induction variable i_P from the original program and the variable g_T in the transformed program. Figure 46 presents the proof tree for $I \vdash \langle 2*i_P\rangle 4_P \lhd \langle g_T\rangle 4_T$; Figure 47 presents the proof tree for $I \vdash \langle g_P\rangle 7_P \lhd \langle g_T\rangle 7_T$.

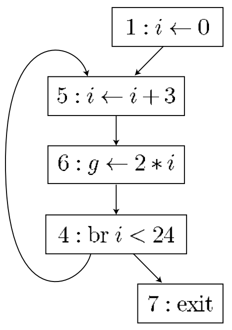

Figure 43: Program P Before Induction Variable Elimination

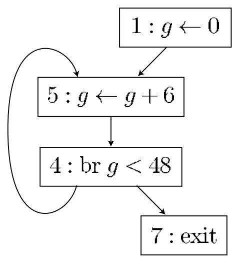

Figure 44: Program T After Induction Variable Elimination

$$\begin{aligned}I=\{&\langle g_P = 2*i_P\rangle 4_P, \langle 2*i_P\rangle 5_P \lhd \langle g_T\rangle 5_T,\\ &\langle 2*i_P\rangle 4_P \lhd \langle g_T\rangle 4_T, \langle g_P\rangle 7_P \lhd \langle g_T\rangle 7_T\}\end{aligned}$$

Figure 45: Invariants for Induction Variable Elimination

$$\begin{array}{cl}
\langle 2*i_P\rangle 5_P \lhd \langle g_T\rangle 5_T \in I, 2*i_P = g_T \Rightarrow 2*(i_P+3) = g_T+6 & \\ \hline
I \vdash \langle 2*(i_P+3)\rangle 5_P \lhd \langle g_T+6\rangle 5_T \cdot 4_T & 23 \\ \hline
I \vdash 5_P\langle 2*i_P\rangle 6_P \lhd \langle g_T+6\rangle 5_T \cdot 4_T & 26 \\ \hline
I \vdash \langle 2*i_P\rangle 4_P \lhd \langle g_T+6\rangle 5_T \cdot 4_T & 24 \\ \hline
I \vdash \langle 2*i_P\rangle 4_P \lhd 5_T\langle g_T\rangle 4_T & 31 \\ \hline
I \vdash \langle 2*i_P\rangle 4_P \lhd \langle g_T\rangle 4_T & 29
\end{array}$$

Figure 46: Proof Tree for $I \vdash \langle 2*i_P\rangle 4_P \lhd \langle g_T\rangle 4_T$

$$\begin{array}{cl}
I \vdash \langle g_P = 2*i_P\rangle 4_P, \langle 2*i_P\rangle 4_P \lhd \langle g_T\rangle 4_T \in I, & \\
g_P = 2*i_P \wedge g_T \geq 48 \wedge 2*i_P = g_T \Rightarrow (i_P \geq 24 \wedge g_P = g_T) & \\ \hline
I \vdash \langle i_P \geq 24, g_P\rangle 4_P \lhd \langle g_T \geq 48, g_T\rangle 4_T \cdot 7_T & 23 \\ \hline
I \vdash 4_P\langle g_P\rangle 7_P \lhd 4_T\langle g_T \geq 48, g_T\rangle 4_T \cdot 7_T & 26 \\ \hline
I \vdash \langle g_P\rangle 7_P \lhd \langle g_T \geq 48, g_T\rangle 4_T \cdot 7_T & 24 \\ \hline
I \vdash \langle g_P\rangle 7_P \lhd 4_T\langle g_T\rangle 7_T & 31 \\ \hline
I \vdash \langle g_P\rangle 7_P \lhd \langle g_T\rangle 7_T & 29
\end{array}$$

Figure 47: Proof Tree for $I \vdash \langle g_P\rangle 7_P \lhd \langle g_T\rangle 7_T$

7.5 Loop Unrolling

The next optimization unrolls the loop once. Figure 48 presents the program before loop unrolling; Figure 49 presents the program after unrolling the loop. Note that the loop unrolling transformation preserves the loop exit test; this test can be eliminated by the branch elimination optimization discussed in Section 7.6.

Figure 50 presents the set of invariants that the compiler generates for this example. Note that, unlike the simulation invariants in previous examples, these simulation invariants have conditions. The conditions are used to separate different executions of the same node in the original program.

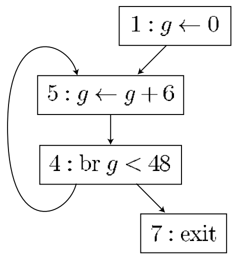

Figure 48: Program P Before Loop Unrolling

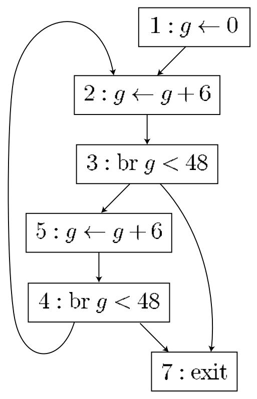

Figure 49: Program T After Loop Unrolling

$$\begin{aligned}I=\{&\langle g_P\%12 = 0 \vee g_P\%12 = 6\rangle 4_P, \langle g_T\%12 = 0\rangle 4_T,\\ &\langle g_P\%12 = 0, g_P\rangle 5_P \lhd \langle g_T\rangle 2_T, \langle g_T\%12 = 6\rangle 3_T,\\ &\langle g_P\%12 = 6, g_P\rangle 4_P \lhd \langle g_T\rangle 3_T,\\ &\langle g_P\%12 = 6, g_P\rangle 5_P \lhd \langle g_T\rangle 5_T,\\ &\langle g_P\%12 = 0, g_P\rangle 4_P \lhd \langle g_T\rangle 4_T\}\end{aligned}$$

Figure 50: Invariants for Loop Unrolling

Some of the time, the execution at node 4_P corresponds to the execution at node 4_T, and other times to the execution at node 3_T. The conditions in the simulation invariants identify when, in the execution of the transformed program, each correspondence holds. For example, when $g_P\%12 = 0$, the execution at 4_P corresponds to the execution at 4_T; when $g_P\%12 = 6$, the execution at 4_P corresponds to the execution at 3_T.

Figure 51 presents the proof tree for $I \vdash \langle g_P\rangle 7_P \lhd \langle g_T\rangle 7_T$. Loop unrolling is a transformation which replicates code. In an inverse transformation that would shrink code, the key component of the correctness proofs would be the case analysis rule, rule 34.

7.6 Branch Elimination

We continue with our example by eliminating the branch in the middle of the loop at node 3_P. Figure 52 presents the program before the branch is eliminated. The key property that allows the compiler to remove the branch is that $g_P\%12 = 6 \wedge g_P \leq 48$ at 3_P, which implies that $g_P < 48$ at 3_P. In other words, the condition in the branch is always true. Figure 53 presents the program after the branch is eliminated. Figure 54 presents the set of invariants that the compiler generates for this example.

Figure 55 presents the proof tree for $I \vdash \langle i_P\rangle 5_P \lhd \langle i_T\rangle 5_T$. The path that proof must consider in the transformed program T is from 2_P to 5_P. The corresponding path in P that is used to prove $I \vdash \langle i_P\rangle 5_P \lhd \langle i_T\rangle 5_T$ is the path from 2_T to 3_T to 5_T. The loop in the original program P always exits from 4_P, not 3_P. This fact shows up because the standard invariant $\langle g_P\%12 = 0 \wedge g_P < 48\rangle 2_P$ implies the condition $g_P + 6 < 48$ from the partial simulation invariant for P at 2_P.

8 Termination Anomalies

Throughout the paper so far, we have required that the transformed program simulate the original program in the

$$\dfrac{\begin{array}{cl} \langle g_P\%12 = 0, g_P\rangle 4_P \lhd \langle g_T\rangle 4_T \in I, & \\ g_T \geq 48 \wedge g_P = g_T \Rightarrow g_P \geq 48 \wedge g_P = g_T & \\ \hline I \vdash \langle g_P \geq 48, g_P\rangle 4_P \lhd \langle g_T \geq 48, g_T\rangle 4_T \cdot 7_T & {\scriptstyle 23} \\ \hline I \vdash 4_P\langle g_P\rangle 7_P \lhd \langle g_T \geq 48, g_T\rangle 4_T \cdot 7_T & {\scriptstyle 28} \\ \hline I \vdash \langle g_P\rangle 7_P \lhd \langle g_T \geq 48, g_T\rangle 4_T \cdot 7_T & {\scriptstyle 24} \\ \hline I \vdash \langle g_P\rangle 7_P \lhd 4_T\langle g_T\rangle 7_T & {\scriptstyle 33} \end{array} \quad \begin{array}{cl} \langle g_P\%12 = 6, g_P\rangle 4_P \lhd \langle g_T\rangle 3_T \in I, & \\ g_T \geq 48 \wedge g_P = g_T \Rightarrow g_P \geq 48 \wedge g_P = g_T & \\ \hline I \vdash \langle g_P \geq 48, g_P\rangle 4_P \lhd \langle g_T \geq 48, g_T\rangle 3_T \cdot 7_T & {\scriptstyle 23} \\ \hline I \vdash 4_P\langle g_P\rangle 7_P \lhd \langle g_T \geq 48, g_T\rangle 3_T \cdot 7_T & {\scriptstyle 28} \\ \hline I \vdash \langle g_P\rangle 7_P \lhd \langle g_T \geq 48, g_T\rangle 3_T \cdot 7_T & {\scriptstyle 24} \\ \hline I \vdash \langle g_P\rangle 7_P \lhd 3_T\langle g_T\rangle 7_T & {\scriptstyle 33} \end{array}}{I \vdash \langle g_P\rangle 7_P \lhd \langle g_T\rangle 7_T}\,{\scriptstyle 29}$$

Figure 51: Proof Tree for $I \vdash \langle g_P\rangle 7_P \lhd \langle g_T\rangle 7_T$

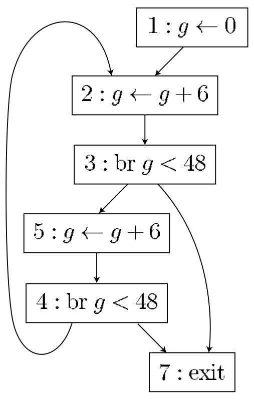

Figure 52: Program P Before Branch Elimination

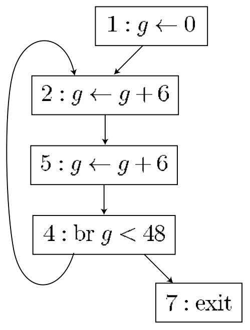

Figure 53: Program T After Branch Elimination

$$\begin{aligned} I = \{ & \langle g_P\%12 = 0 \wedge g_P < 48\rangle 2_P, \langle g_P\%12 = 6 \wedge g_P \leq 48\rangle 3_P, \\ & \langle g_P\%12 = 6 \wedge g_P < 48\rangle 5_P, \langle g_P\%12 = 0 \wedge g_P \leq 48\rangle 4_P, \\ & \langle g_P\rangle 2_P \lhd \langle g_T\rangle 2_T, \langle g_P\rangle 5_P \lhd \langle g_T\rangle 5_T, \langle g_P\rangle 3_P \lhd \langle g_T\rangle 5_T, \\ & \langle g_P\rangle 4_P \lhd \langle g_T\rangle 4_T, \langle g_P\rangle 7_P \lhd \langle g_T\rangle 7_T \} \end{aligned}$$

Figure 54: Invariants for Branch Elimination

$$\begin{array}{cl} I \vdash \langle g_P\%12 = 0 \wedge g_P < 48\rangle 2_P, \langle g_P\rangle 2_P \lhd \langle g_T\rangle 2_T \in I, & \\ g_P\%12 = 0 \wedge g_P < 48 \wedge g_P = g_T \Rightarrow & \\ g_P + 6 < 48 \wedge g_P + 6 = g_T + 6 & \\ \hline I \vdash \langle g_P + 6 < 48, g_P + 6\rangle 2_P \lhd \langle g_T + 6\rangle 2_T & {\scriptstyle 23} \\ \hline I \vdash 2_P\langle g_P < 48, g_P\rangle 3_P \lhd \langle g_T + 6\rangle 2_T & {\scriptstyle 26} \\ \hline I \vdash \langle g_P < 48, g_P\rangle 3_P \lhd \langle g_T + 6\rangle 2_T & {\scriptstyle 24} \\ \hline I \vdash 3_P\langle g_P\rangle 5_P \lhd \langle g_T + 6\rangle 2_T & {\scriptstyle 27} \\ \hline I \vdash \langle g_P\rangle 5_P \lhd \langle g_T + 6\rangle 2_T & {\scriptstyle 24} \\ \hline I \vdash \langle g_P\rangle 5_P \lhd \langle g_T + 6\rangle 2_T \cdot 5_T & {\scriptstyle 35} \\ \hline I \vdash \langle g_P\rangle 5_P \lhd 2_T\langle g_T\rangle 5_T & {\scriptstyle 31} \\ \hline I \vdash \langle g_P\rangle 5_P \lhd \langle g_T\rangle 5_T & {\scriptstyle 29} \end{array}$$

Figure 55: Proof Tree for $I \vdash \langle g_P\rangle 5_P \lhd \langle g_T\rangle 5_T$

sense that for every execution in the transformed program that reaches the exit node, there exists an execution in the original program that reaches the exit node such that the values of the observable variables are the same. There is, however, an anomaly associated with this notion of simulation. What happens if the transformed program contains an infinite loop? Then the transformed program implements *any* program. One can imagine that programmers might like to have stronger guarantees; in particular they might like the guarantee that if the original program terminates, then so does the transformed program.

One option is to require also that the original program simulate the transformed program. If the two programs simulate each other, the transformed program terminates if and only if the original program terminates. And if they terminate, they terminate with identical values in corresponding observable values. We anticipate that this will be a good solution in practice.

There is, however, a potential anomaly associated with this approach. The logics for proving simulation invariants are based on notions of partial correctness. For some programs, it is impossible to use the logic to prove that they simulate each other, even if they both terminate with the same result. Consider the two programs in Figures 57 and 56 that compute $g = 48$. Using the logic presented in Section 5.4, it is not possible to prove that the closed form program in Figure 57 implements the iterative program in Figure 56. Roughly speaking, the problem is that the logic cannot prove that the loop in the iterative program terminates.

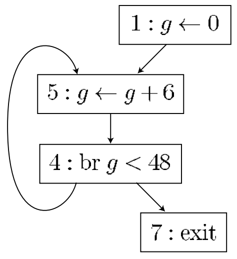

Figure 56: Iterative Program to Compute $g = 48$

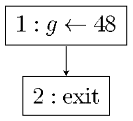

Figure 57: Closed Form Program to Compute $g = 48$

We do not anticipate that this anomaly will prove to be a problem in practice, because the overwhelming majority of compiler transformations do not eliminate or introduce loops. If it does turn out to be a problem in practice, the solution is to augment the logic so that it can prove that loops terminate.

9 Code Generation

In principle, we believe that it is possible to produce a proof that the final object code correctly implements the original program. For engineering reasons, however, we designed the proof system to work with a standard intermediate format based on control flow graphs. The parser, which produces the initial control flow graph, and the code generator, which generates object code from the final control flow graph, are therefore potential sources of uncaught errors. We believe it should be straightforward, for reasonable languages, to produce a standard parser that is not a serious source of errors. It is not so obvious how the code generator can be made simple enough to be reliable.

Our goal is make the step from the final control flow graph to the generated code be as small as possible. Ideally, each node in the control flow graph would correspond to a single instruction in the generated code. To achieve this goal, it must be possible to express the result of complicated, machine-specific code generation algorithms (such as register allocation and instruction selection) using control flow graphs. After the compiler applies these algorithms, the final control flow graph would be structured in a stylized way appropriate for the target architecture. The code generator for the target architecture would accept such a control flow graph as input and use a simple translation algorithm to produce the final object code.

With this approach, we anticipate that code generators can be made approximately as simple as proof checkers. We therefore anticipate that it will be possible to build standard code generators with an acceptable level of reliability for most users. However, we would once again like to emphasize that it should be possible to build a framework in which the compilation is checked from source code to object code.

In the following two sections, we first present an approach for a simple RISC instruction set, then discuss an approach for more complicated instruction sets.

9.1 A Simple RISC Instruction Set

For a simple RISC instruction set, the key idea is to introduce special variables that the code generator interprets as registers. The control flow graph is then transformed so that each node corresponds to a single instruction in the generated code. We first consider assignment nodes.

- If the destination variable is a register variable, the source expression must be one of the following:
  - A non-register variable. In this case the node corresponds to a load instruction.
  - A constant. In this case the node corresponds to a load immediate instruction.
  - A single arithmetic operation with register variable operands. In this case the node corresponds to an arithmetic instruction that operates on the two source registers to produce a value that is written into the destination register.
  - A single arithmetic operation with one register variable operand and one constant operand. In this case the node corresponds to an arithmetic instruction that operates on one source register and an immediate constant to produce a value that is written into the destination register.
- If the destination variable of an assignment node is a non-register variable, the source expression must consist of a register variable, and the node corresponds to a store instruction.

It is possible to convert assignment nodes with arbitrary expressions to this form. The first step is to flatten the expression by introducing temporary variables to hold the intermediate values computed by the expression. Additional assignment nodes transfer these values to the new temporary variables. The second step is to use a register allocation algorithm to transform the control flow graph to fit the form described above.

We next consider conditional branch nodes. If the condition is the constant true or false, the node corresponds to an unconditional branch instruction. Otherwise, the condition must compare a register variable with zero so that the instruction corresponds either to a branch if zero instruction or a branch if not zero instruction.

9.2 More Complex Instruction Sets

Many processors offer more complex instructions that, in effect, do multiple things in a single cycle. In the ARM instruction set, for example, the execution of each instruction may be predicated on several condition codes. ARM instructions can therefore be modeled as consisting of a conditional branch plus the other operations in the instruction. The x86 instruction set has instructions that assign values to several registers.

We believe the correct approach for these more complex instruction sets is to let the compiler writer extend the possible types of nodes in the control flow graph. The semantics of each new type of node would be given in terms of the base nodes in standard control flow graphs. We illustrate this approach with an example.

For instruction sets with condition codes, the programmer would define a new variable for each condition code and new assignment nodes that set the condition codes appropriately. The semantics of each new node would be given as a small control flow graph that performed the assignment, tested the appropriate conditions, and set the appropriate condition code variables. If the instruction set also has predicated execution, the control flow graph would use conditional branch nodes to check the appropriate condition codes before performing the instruction.

Each new type of node would come with proof rules automatically derived from its underlying control flow graph. The proof checker could therefore verify proofs on control flow graphs that include these types of nodes. The code generator would require the preceding phases of the compiler to produce a control flow graph that contained only those types of nodes that translate directly into a single instruction on the target architecture. With this approach, all complex code generation algorithms could operate on control flow graphs, with their results checked for correctness.

10 Related Work

Most existing research on compiler correctness has focused on techniques that deliver a compiler guaranteed to operate correctly on every input program [6, 5]; we call such a compiler a *totally correct* compiler. A credible compiler, on the other hand, is not necessarily guaranteed to operate correctly on all programs — it merely produces a proof that it has operated correctly on the current program.

In the absence of other differences, one would clearly prefer a totally correct compiler to a credible compiler. After all, the credible compiler may fail to compile some programs correctly, while the totally correct compiler will always work. But the totally correct compiler approach imposes a significant pragmatic drawback: it requires the source code of the compiler, rather than its output, to be proved correct. So programmers must express the compiler in a way that is amenable to these correctness proofs. In practice this invasive constraint has restricted the compiler to a limited set of source languages and compiler algorithms. Although the concept of a totally correct compiler has been around for many years, there are, to our knowledge, no totally correct compilers that produce close to production-quality code for realistic programming languages. Credible compilation offers the compiler developer much more freedom. The compiler can be developed in any language using any methodology and perform arbitrary transformations. The only constraint is that the compiler produce a proof that its result is correct.

The concept of credible compilers has also arisen in the context of compiling synchronous languages [3, 9]. Our approach, while philosophically similar, is technically much different. It is designed for standard imperative languages and therefore uses drastically different techniques for deriving and expressing the correctness proofs.

We often are asked the question "How is your approach different from proof-carrying code [8]?"[4] In our view, credible compilers and proof-carrying code are orthogonal concepts. Proof-carrying code is used to prove properties of *one* program, typically the compiled program. Credible compilers establish a correspondence between *two* programs: an original program and a compiled program. Given a safe programming language, a credible compiler will produce guarantees that are stronger than those provided by typical applications of proof-carrying code. So, for example, if the source language is type safe and a credible compiler produces a proof that the compiled program correctly implements the original program, then the compiled program is also type safe.

But proof-carrying code can, in principle, be used to prove properties that are not visible in the semantics of the language. For example, one might use proof-carrying code to prove that a program does not execute a sequence of instructions that may damage the hardware. Because most languages simply do not deal with the kinds of concepts required to prove such a property as a correspondence between two programs, credible compilation is not particularly relevant to these kinds of problems.

11 Conclusions

Most research on compiler correctness has focused on obtaining a compiler that is guaranteed to generate correct code for every input program. This paper presents a less ambitious, but hopefully much more practical approach: require the compiler to generate a proof that the generated code correctly implements the input program. Credible compilation, as we call this approach, gives the compiler developer maximum flexibility, helps developers find compiler bugs, and eliminates the need to trust the developers of compiler passes.

This paper presents logics that a compiler can use to prove that its transformations are correct, and provides examples that illustrate how the proofs would work for several standard transformations. The logics support the standard two-phase approach to optimization: there is a logic that the compiler can use to prove that its analysis results are correct, and a logic that the compiler can use to prove that the transformed program correctly implements the original program.

[4]Proof-carrying code is code augmented with a proof that the code satisfies safety properties such as type safety or the absence of array bounds violations.

12 Acknowledgment

The authors would like to thank Radu Rugina for many interesting discussions about pointer analysis, including the discussion that led to the first version of the proof rules for flow-insensitive analyses.

References

[1] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994.

[2] K. Apt and E. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, 1997.

[3] A. Cimatti, F. Giunchiglia, P. Pecchiari, B. Pietra, J. Profeta, D. Romano, P. Traverso, and B. Yu. A provably correct embedded verifier for the certification of safety critical software. In *Proceedings of the 9th International Conference on Computer Aided Verification*, pages 202–213, Haifa, Israel, June 1997.

[4] R. Floyd. Assigning meanings to programs. In J. Schwartz, editor, *Proceedings of the Symposium in Applied Mathematics*, number 19, pages 19–32, 1967.

[5] Wolfgang Goerigk. Towards Rigorous Compiler Implementation Verification. In Rudolf Berghammer and Friedemann Simon, editors, *Proc. of the 1997 Workshop on Programming Languages and Fundamentals of Programming*, pages 118 – 126, Avendorf, Germany, November 1997.

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

[7] J. M. Morris. A general axiom of assignment. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 25–34. D. Reidel Publishing, 1982.

[8] G. Necula. Proof-carrying code. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages*, pages 106–119, Paris, France, January 1997.

[9] A. Pnueli, M. Siegal, and E. Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lisbon, Portugal, March 1998.

[10] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, January 1996.