# Credible Compilation

Martin C. Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

March 10, 1999

**Abstract**

This paper presents a new concept in compiler correctness: instead of proving that the compiler performs all of its transformations correctly, the compiler generates a proof that the transformed program correctly implements the input program. A simple proof checker can then verify that the program was compiled correctly. We call a compiler that produces such proofs a *credible compiler*, because it produces verifiable evidence that it is operating correctly.

Compiler optimizations usually consist of two steps — an analysis step determines if it is legal to apply the optimization, and a transformation step applies the optimization to generate a transformed program that computes the same result as the original program. Our approach supports this two-step structure. It provides a logic that the compiler can use to prove that its program analysis results are correct, and a logic that the compiler can use to prove that the transformed program correctly simulates the original program. These logics are defined for a standard program representation, control flow graphs. This report defines these logics and proves that they are sound with respect to a standard operational semantics. It also presents detailed examples that demonstrate how a compiler can use the logics to prove the correctness of several standard optimizations.

We believe that credible compilation has the potential to revolutionize the way compilers are built and used. Specifically, they will allow programmers to quickly determine if the compiler compiled their program correctly, help developers find and eliminate bugs in compiler passes, allow large groups of mutually untrusting people to collaborate productively on the same compiler, increase the speed with which compilers are developed and released, and make it possible to aggressively upgrade large, stable compiler systems without fear of inadvertantly introducing undetected errors.

## 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 he or she runs the program, the programmer 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 a fundamental shift in the relationship between the compiler and the programmer. Every time the compiler transforms the program, it generates a proof that the transformed program produces the same result as the original program. When the compiler finishes, the programmer can use a simple proof checker to verify that the program was compiled correctly. We call a compiler that generates these proofs a *credible* compiler, because it produces verifiable evidence that it is operating correctly.

We believe that credible compilation has the potential to revolutionize the way compilers are built and used. Instead of having to accept whatever the compiler generates on blind faith, programmers will be able to verify that the compiler compiled their program correctly. Credible compilers will also help developers find and eliminate bugs in compiler passes, allow large groups of mutually untrusting people to collaborate productively on the same compiler, make it possible to aggressively upgrade large, stable systems without fear of inadvertantly introducing undetected errors, promote the use of compilers that are customized for specific application domains, shrink the length of the compiler development cycle by making it practical to use buggy compilers, and make the use of compilers that do not produce correctness proofs a successful basis for product liability claims.

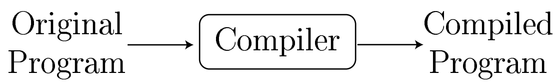


Figure 1: Traditional Compilation

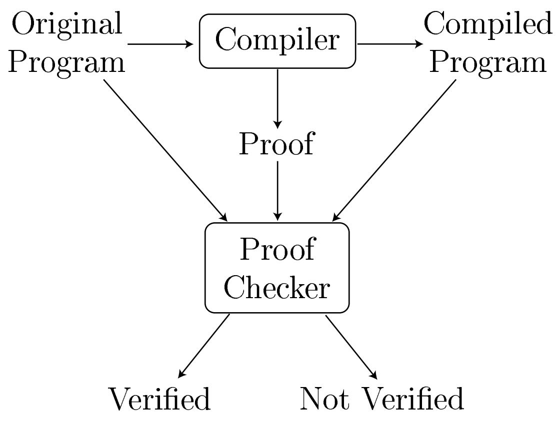


Figure 2: Credible Compilation

Figures 1 and 2 graphically illustrate the difference between traditional compilation and credible compilation. A traditional compiler generates a compiled program and nothing else. A credible compiler, on the other hand, also generates a proof that the compiled program correctly implements the original program. A proof checker can then take the original program, the proof, and the compiled program, and check if the proof is correct. If so, the compilation is verified and the compiled program is guaranteed to correctly implement the original program. If the proof does not check, the compilation is not verified and all bets are off.

This paper introduces the basic techniques required to build credible compilers for standard programming languages such as C and Java. The organization is as follows. Section 2 presents an example that illustrates the basic concepts of *standard invariants*, which are used to prove that program analysis results are correct, and *simulation invariants*, which are used to prove that a transformed program generates the same result as the original program. Section 3 presents the technical core of the paper: the logics used to prove standard invariants and simulation invariants, and the proofs that these logics are sound. Section 4 presents a running example that shows how to generate correctness proofs for several standard transformations. Section 5 discusses some anomalies associated with proving that loops terminate, Section 6 discusses issues related to code generation, and Section 7 discusses related work. Section 8 discusses the potential impact of credible compilation. We present our conclusions in Section 9.

# 2 Example

In this section we present an example that explains how a credible compiler can prove that it performed a translation correctly. Figure 3 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 $i$. There is also a conditional branch node $4:$ 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.

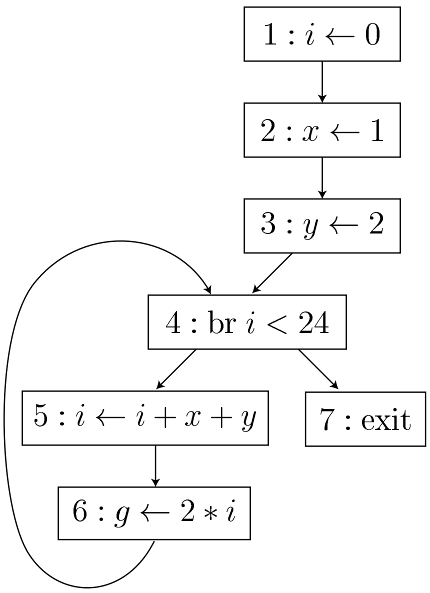


Figure 3: Original Program

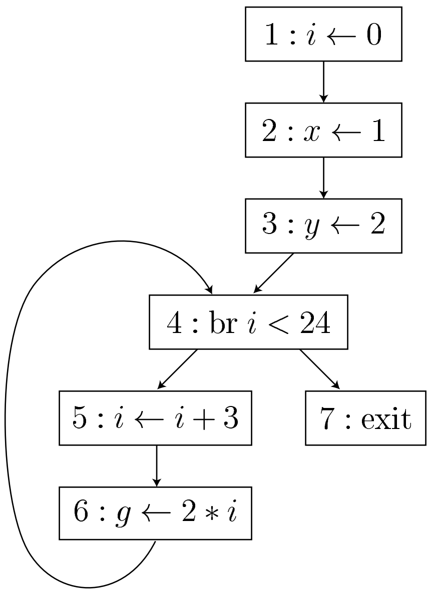


Figure 4: Program After Constant Propagation and Constant Folding

Figure 4 presents the program after constant propagation and constant folding. The compiler has replaced the node $5: i \leftarrow i+x+y$ at label 5 with the node $5: i \leftarrow i+3$. 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 $x$ is always 1 and $y$ is always 2 at the program point before node 5. So, $x + y$ is always 3 at this program point.

- **Transformation:** The compiler used the analysis information to transform the program so that generates the same result while (hopefully) executing in less time or space or consuming less power. In our example, the compiler simplifies the expression $x + y$ to 3.

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.

## 2.1 Proving Analysis Results Correct

Many years ago, Floyd came up with a technique for proving properties of programs [4]. This technique was generalized and extended, and eventually came to be understood as a logic whose proof rules are derived from the structure of the program [2]. The basic idea is to assert a set of properties about the relationships between variables at different points in the program, then use the logic to prove that the properties always hold. If so, each property is called an invariant, because it is always true when the flow of control reaches the corresponding point in the program.

In our example, the key invariant is that at the point just before the program executes node 5, it is always true that $x = 1$ and $y = 2$. We represent this invariant as $\langle x = 1 \wedge y = 2\rangle 5$. Section 3.3 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.

In our example, the simplest way for the compiler to generate a proof of $\langle x = 1 \wedge y = 2\rangle 5$ 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. Here is the set of invariants in our example:

- $\langle x = 1\rangle 3$

- $\langle x = 1 \wedge y = 2\rangle 4$

- $\langle x = 1 \wedge y = 2\rangle 5$

- $\langle x = 1 \wedge y = 2\rangle 6$

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 3.3 to produce machine-verifiable versions of these proofs.

- $\langle x = 1\rangle 3$ because the only preceding node, node 2, sets $x$ to 1.

- To prove $\langle x = 1 \wedge y = 2 \rangle 4$, first assume $\langle x = 1 \rangle 3$ and $\langle x = 1 \wedge y = 2 \rangle 6$. Then consider the two preceding nodes, nodes 3 and 6. Because $\langle x = 1 \rangle 3$ and 3 sets $y$ to 2, $\langle x = 1 \wedge y = 2 \rangle 4$. Because $\langle x = 1 \wedge y = 2 \rangle 6$ and node 6 does not affect the value of either $x$ or $y$, $\langle x = 1 \wedge y = 2 \rangle 4$.

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 3.3 supports both styles of proofs.

## 2.2 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 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 simulates the original program in the following sense: for every execution of the original program $P$ that reaches the final node $7_P$, there exists an execution of the transformed program $T$ that reaches the final node $7_T$ such that $g_P$ at $7_P = g_T$ at $7_T$. We call such a correspondence a *simulation invariant*, and write it as $\langle g_P \rangle 7_P \rhd \langle g_T \rangle 7_T$. In Section 3.4 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 $\langle g_P \rangle 7_P \rhd \langle g_T \rangle 7_T$. First, the compiler is given that $\langle g_P \rangle 1_P \rhd \langle g_T \rangle 1_T$ — 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:

- $\langle (g_P, i_P) \rangle 2_P \rhd \langle (g_T, i_T) \rangle 2_T$
- $\langle (g_P, i_P) \rangle 3_P \rhd \langle (g_T, i_T) \rangle 3_T$
- $\langle (g_P, i_P) \rangle 4_P \rhd \langle (g_T, i_T) \rangle 4_T$

- $\langle (g_P, i_P) \rangle 5_P \rhd \langle (g_T, i_T) \rangle 5_T$
- $\langle (g_P, i_P) \rangle 6_P \rhd \langle (g_T, i_T) \rangle 6_T$
- $\langle g_P \rangle 7_P \rhd \langle g_T \rangle 7_T$

The key simulation invariants are $\langle g_P \rangle 7_P \rhd \langle g_T \rangle 7_T$, $\langle (g_P, i_P) \rangle 6_P \rhd \langle (g_T, i_T) \rangle 6_T$ and $\langle (g_P, i_P) \rangle 4_P \rhd \langle (g_T, i_T) \rangle 4_T$. We next outline the proofs of these two invariants. The compiler can use the logic presented in Section 3.4 to produce machine-verifiable versions of these proofs.

- To prove $\langle g_P \rangle 7_P \rhd \langle g_T \rangle 7_T$, first assume that $\langle (g_P, i_P) \rangle 4_P \rhd \langle (g_T, i_T) \rangle 4_T$. For each path to $7_P$ in $P$, we must find a corresponding path in $T$ to $7_P$ such that the values of $g_P$ and $g_T$ are the same in both paths. The only path to $7_P$ goes from $4_P$ to $7_P$ when $i_P \geq 24$. The corresponding path in $T$ goes from $4_T$ to $7_T$ when $i_T \geq 24$. Because $\langle (g_P, i_P) \rangle 4_P \rhd \langle (g_T, i_T) \rangle 4_T$, control flows from $4_T$ to $7_T$ whenever control flows from $4_P$ to $7_P$. The simulation invariant $\langle (g_P, i_P) \rangle 4_P \rhd \langle (g_T, i_T) \rangle 4_T$ also implies that the values of $g_P$ and $g_T$ are the same in both cases.

- To prove $\langle (g_P, i_P) \rangle 6_P \rhd \langle (g_T, i_T) \rangle 6_T$, assume $\langle (g_P, i_P) \rangle 5_P \rhd \langle (g_T, i_T) \rangle 5_T$. The only path to $6_P$ goes from $5_P$ to $6_P$, with $i_P$ at $6_P = i_P$ at $5_P + x_P$ at $5_P + y_P$ at $5_P$. The analysis proofs showed that $x_P$ at $5_P + y_P$ at $5_P = 3$, so $i_P$ at $6_P = i_P$ at $5_P + 3$. The corresponding path in $T$ goes from $5_T$ to $6_T$, with $i_T$ at $6_T = i_T$ at $5_T + 3$.

  The assumed simulation invariant $\langle (g_P, i_P) \rangle 5_P \rhd \langle (g_T, i_T) \rangle 5_T$ allows us verify a correspondence between the values of $i_P$ at $6_P$ and $i_T$ at $6_P$; namely that they are equal. Because $5_P$ does not change $g_P$ and $5_T$ does not change $g_T$, $g_P$ at $6_P$ and $g_T$ at $6_P$ have the same value.

- To prove $\langle (g_P, i_P) \rangle 4_P \rhd \langle (g_T, i_T) \rangle 4_T$, first assume $\langle (g_P, i_P) \rangle 3_P \rhd \langle (g_T, i_T) \rangle 3_T$ and $\langle (g_P, i_P) \rangle 6_P \rhd \langle (g_T, i_T) \rangle 6_T$. There are two paths to $4_P$:
  - Control flows from $3_P$ to $4_P$. The corresponding path in $T$ is from $3_T$ to $4_T$, so we can apply the assumed simulation invariant $\langle (g_P, i_P) \rangle 3_P \rhd \langle (g_T, i_T) \rangle 3_T$ to derive $g_P$ at $4_P = g_T$ at $4_T$ and $i_P$ at $4_P = i_T$ at $4_T$.
  - Control flows from $6_P$ to $4_P$, with $g_P$ at $4_P = 2 * i_P$ at $6_P$. The corresponding path in $T$ is from $6_T$ to $4_T$, with $g_T$ at $4_T = 2 * i_T$ at $6_T$. We can apply the assumed simulation invariant $\langle (g_P, i_P) \rangle 6_P \rhd \langle (g_T, i_T) \rangle 6_T$ to derive $2 * i_P$ at $6_P$ $= 2 * i_T$ at $6_T$. Since $6_P$ does not change $i_P$ and $6_T$ does not change $i_T$, we can derive $g_P$ at $4_P = g_T$ at $4_T$ and $i_P$ at $4_P = i_T$ at $4_T$.

# 3 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 prove that this logic is sound. We also present the logic used to prove simulation invariants and prove that this logic is sound.

## 3.1 Program Representation

We propose that compiler passes use a common intermediate representation based on control flow graphs. It is possible, of course, to write translators between intermediate representations so that passes that use specialized or merely different intermediate representations can participate. In this section we define a simple intermediate representation that we use to present the major ideas and concepts in the remainder of the paper. We expect that a practical implementation would require a more elaborate intermediate representation.

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 disjoint sets of local variables $l \in L$ and externally observable variables $o \in O$; the set of variables $v \in V = L \cup O$ is the union of these two sets. Variables have integer values and expressions evaluate to integers. The following abstract syntax defines the set of expressions $e$.

$$\begin{array}{rcl} e & ::= & \mathcal{Z}|V|e+e|e-e|e*e|e/e|e\%e|-e| \\ & & \text{true}|\text{false}|e=e|e\neq e|e>e|e\geq e|e<e|e\leq e|\neg e|e\wedge e|e\vee e|e\Rightarrow e|e\Leftrightarrow e \end{array}$$

In some cases, we interpret an expression 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 $s : v \leftarrow e\ t$ has its label $s$, a variable $v$, an expression $e$ and a label $t$. When the node executes, it evaluates $e$ and assigns the value to $v$. Execution continues at the node whose label is $t$.

- **Conditional Branch:** A conditional branch node $s : \text{br}\ c\ t_1\ t_2$ 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 $s : \text{nop}\ t$ 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 $s_x : \text{exit}$ 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 $s : v \leftarrow e\ t$ is true if there exists an assignment node with label $s$, variable $v$, expression $e$ and label $t$ in the control flow graph, and false otherwise. Also, $s : \text{br}\ c\ t_1\ t_2$ 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 $t$:*

$$\text{pred}(t) = \{s | s : v \leftarrow e\ t\} \cup \{s | s : \text{nop}\ t\} \cup \{s | s : \text{br}\ c\ t\ t'\} \cup \{s | s : \text{br}\ c\ t'\ t\}$$

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 $s$, $s_x \notin \text{pred}(s)$.

## 3.2 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 $m : V \rightarrow \mathcal{Z}$ that maps each variable to its value. We start by extending the domain of the memory function $m$ to constants and expressions as shown in Figure 5.

$$\begin{array}{l}
m(z) = z \\
m(e_1 + e_2) = m(e_1) + m(e_2) \\
m(e_1 - e_2) = m(e_1) - m(e_2) \\
m(e_1 * e_2) = m(e_1) * m(e_2) \\
m(e_1 / e_2) = m(e_1) / m(e_2) \\
m(e_1 \% e_2) = m(e_1) \% m(e_2) \\
m(-e) = -m(e) \\
m(\text{true}) = \text{true} \\
m(\text{false}) = \text{false} \\
m(e_1 = e_2) = m(e_1) = m(e_2) \\
m(e_1 > e_2) = m(e_1) > m(e_2) \\
m(e_1 \geq e_2) = m(e_1) \geq m(e_2) \\
m(e_1 < e_2) = m(e_1) < m(e_2) \\
m(e_1 \leq e_2) = m(e_1) \leq m(e_2) \\
m(\neg e) = \neg m(e) \\
m(c_1 \wedge c_2) = m(c_1) \wedge m(c_2) \\
m(c_1 \vee c_2) = m(c_1) \vee m(c_2) \\
m(c_1 \Rightarrow c_2) = m(c_1) \Rightarrow m(c_2) \\
m(c_1 \Leftrightarrow c_2) = m(c_1) \Leftrightarrow m(c_2)
\end{array}$$

Figure 5: Extending $m$ to Constants and Expressions

The operational semantics is defined using a transition function $\rightarrow$ 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 6 presents the rules that define the transition function. In the initial memory $m_0$, local variables have value 0 and observable variables have arbitrary values.

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*

$$\frac{s : v \leftarrow e\ t}{\langle s, m\rangle \rightarrow \langle t, m[v \mapsto m(e)]\rangle} \qquad \textit{op-assign} \qquad (1)$$

$$\frac{s : \mathrm{nop}\ t}{\langle s, m\rangle \rightarrow \langle t, m\rangle} \qquad \textit{op-nop} \qquad (2)$$

$$\frac{s : \mathrm{br}\ c\ t_1\ t_2, m(c)}{\langle s, m\rangle \rightarrow \langle t_1, m\rangle} \qquad \textit{op-brtrue} \qquad (3)$$

$$\frac{s : \mathrm{br}\ c\ t_1\ t_2, \neg m(c)}{\langle s, m\rangle \rightarrow \langle t_2, m\rangle} \qquad \textit{op-brfalse} \qquad (4)$$

Figure 6: Operational Semantics

$\langle s_0, m_0\rangle \rightarrow \cdots \rightarrow \langle s_n, m_n\rangle$ *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.*

## 3.3 Standard Invariants

We next present the logic that the compiler can use to construct proofs that its analysis results are correct. The logic consists of a set of proof rules; these rules are a version of the standard 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 $I$. Figure 7 presents the rules. We assume the existence of a logic for proving standard relationships between integers such as $z < z + 1$ and $x < 4 \wedge y < 3 \Rightarrow x + y < 7$.

### 3.3.1 Proof Trees

Proofs consist of a tree whose nodes are rule uses. One rule use is a child of another rule use if the consequent of the first rule use is an antecedent of the second rule use. Contrary to computer science custom (but consistent with nature), proof trees are customarily drawn with each parent node below its children. There is a partial order defined on the rule uses — a first use precedes a second use if the second use appears on the path from the first use to the root. The last rule in the proof tree is therefore the root.

$$\frac{\mathrm{pred}(t) \neq \emptyset, \forall s \in \mathrm{pred}(t).\ I \vdash s\langle i\rangle t}{I \vdash \langle i\rangle t} \qquad \textit{std-pred} \qquad (5)$$

$$\frac{s : \mathrm{nop}\ t, I \vdash \langle i\rangle s \cdot t}{I \vdash s\langle i\rangle t} \qquad \textit{std-nop} \qquad (6)$$

$$\frac{s : v \leftarrow e\ t, I \vdash \langle i[e/v]\rangle s \cdot t}{I \vdash s\langle i\rangle t} \qquad \textit{std-assign} \qquad (7)$$

$$\frac{s : \mathrm{br}\ c\ t_1\ t_2, I \vdash \langle c \Rightarrow i\rangle s \cdot t_1}{I \vdash s\langle i\rangle t_1} \qquad \textit{std-brtrue} \qquad (8)$$

$$\frac{s : \mathrm{br}\ c\ t_1\ t_2, I \vdash \langle \neg c \Rightarrow i\rangle s \cdot t_2}{I \vdash s\langle i\rangle t_2} \qquad \textit{std-brfalse} \qquad (9)$$

$$\frac{I \vdash \langle i\rangle s}{I \vdash \langle i\rangle s \cdot t} \qquad \textit{std-seq} \qquad (10)$$

$$\frac{\langle i'\rangle s \in I, i' \Rightarrow i}{I \vdash \langle i\rangle s \cdot t} \qquad \textit{std-induction} \qquad (11)$$

$$\frac{i}{I \vdash \langle i\rangle s} \qquad \textit{std-base} \qquad (12)$$

Figure 7: Proof Rules for Standard Invariants

$$\dfrac{6 : g \leftarrow 2 * i\ 4 \qquad \dfrac{\langle x = 1 \wedge y = 2\rangle 6 \in I \quad x = 1 \wedge y = 2 \Rightarrow x = 1 \wedge y = 2}{I \vdash \langle x = 1 \wedge y = 2\rangle 6 \cdot 4}}{I \vdash 6\langle x = 1 \wedge y = 2\rangle 4}$$

Figure 8: Full Proof Tree for $I \vdash 6\langle x = 1 \wedge y = 2\rangle 4$

$$\dfrac{\dfrac{\langle x = 1 \wedge y = 2\rangle 6 \in I}{I \vdash \langle x = 1 \wedge y = 2\rangle 6 \cdot 4}}{I \vdash 6\langle x = 1 \wedge y = 2\rangle 4}$$

Figure 9: Abbreviated Proof Tree for $I \vdash 6\langle x = 1 \wedge y = 2\rangle 4$

$$\dfrac{\dfrac{\dfrac{\langle x = 1\rangle 3 \in I}{I \vdash \langle x = 1 \wedge 2 = 2\rangle 3 \cdot 4}}{I \vdash 3\langle x = 1 \wedge y = 2\rangle 4} \quad \dfrac{\dfrac{\langle x = 1 \wedge y = 2\rangle 6 \in I}{I \vdash \langle x = 1 \wedge y = 2\rangle 6 \cdot 4}}{I \vdash 6\langle x = 1 \wedge y = 2\rangle 4}}{I \vdash \langle x = 1 \wedge y = 2\rangle 4}$$

Figure 10: Abbreviated Proof Tree for $I \vdash \langle x = 1 \wedge y = 2\rangle 4$

Figure 8 presents the proof tree for $I \vdash 6\langle x = 1 \wedge y = 2\rangle 4$. To save space on the page, from now on we present proof trees in abbreviated form. This form omits details such as antecedents that are references to nodes in the control flow graph or trivial implications. Figure 9 presents the abbreviated proof tree for $I \vdash 6\langle x = 1 \wedge y = 2\rangle 4$. Figure 10 presents the abbreviated proof tree for $I \vdash \langle x = 1 \wedge y = 2\rangle 4$. In these proof trees,

$$I = \{\langle x = 1\rangle 3, \langle x = 1 \wedge y = 2\rangle 4, \langle x = 1 \wedge y = 2\rangle 5, \langle x = 1 \wedge y = 2\rangle 6\}$$

### 3.3.2 Soundess of Proof Rules for Standard Invariants

We next prove 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 first prove a lemma used in the theorem, then prove the theorem itself.

**Lemma 1** *Assume for all $\langle i\rangle s \in I$, $I \vdash \langle i\rangle s$. 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 $m(i')$ is* true*. Then $m(i)$ is* true*.*

**Proof:** We do a case analysis of the last rule in the proof of $I \vdash \langle i\rangle s \cdot t$. Rules 10 and 11 are the only rules of the correct form.

- The last rule in the proof is rule 10. In this case we have a proof of $I \vdash \langle i\rangle s$, and by assumption $m(i)$ is true.
- The last rule in the proof is rule 11 with $\langle i'\rangle s \in I$ and $i' \Rightarrow i$, which implies $m(i') \Rightarrow m(i)$. By assumption $\langle i'\rangle s \in I$ implies $I \vdash \langle i'\rangle s$, which implies $m(i')$ is true. We can therefore simplify $m(i') \Rightarrow m(i)$ to $m(i)$ is true.

**Theorem 1** *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* $\langle s_0, m_0 \rangle \rightarrow \cdots \rightarrow \langle t, m \rangle$ *implies* $m(i)$ *is* true.

**Proof:** Induction on the length of the partial execution $\langle s_0, m_0 \rangle \rightarrow \cdots \rightarrow \langle t, m \rangle$.
**Base:** In this case $t = s_0$, which implies $\mathrm{pred}(t) = \emptyset$. The proof is therefore a use of rule 12 with $i$, which implies $m(i)$ is true.
**Induction:** In this case the partial execution is at least one step long, so we can write it as $\langle s_0, m_0 \rangle \rightarrow \cdots \langle s, m' \rangle \rightarrow \langle t, m \rangle$ for some $s \in \mathrm{pred}(t)$. We do a case analysis of the last rule in the proof of $I \vdash \langle i \rangle t$. Rules 12 and 5 are the only rules of the correct form.

- The last rule is 12 with $i$, which implies $m(i)$ is true.
- The last rule is 5. Because $s \in \mathrm{pred}(t)$, there is a proof of $I \vdash s\langle i \rangle t$. We do a case analysis of the last rule in this proof. Rules 6, 7, 8 and 9 are the only rules of the correct form.
  - The last rule is 6, with $s : \mathrm{nop}\ t$. Then $m = m'$ and we have a proof of $I \vdash \langle i \rangle s \cdot t$. By Lemma 1, $m(i)$ is true.
  - The last rule is 7, with $s : v \leftarrow e\ t$. Then $m = m'[v \mapsto m'(e)]$ and we have a proof of $I \vdash \langle i[e/v] \rangle s \cdot t$. By Lemma 1, $m'(i[e/v])$ is true, which we can rewrite as $m'[v \mapsto m'(e)](i)$ is true, then simplify to $m(i)$ is true.
  - The last rule is 8, with $s : \mathrm{br}\ c\ t\ t'$, $m' = m$, and $m'(c)$ is true, and there is a proof of $I \vdash \langle c \Rightarrow i \rangle s \cdot t$. By Lemma 1, $m'(c \Rightarrow i)$ is true, which we can simplify to $m'(c) \Rightarrow m'(i)$, then to $m(i)$ is true.
  - The last rule is 9, with $s : \mathrm{br}\ c\ t'\ t$, $m' = m$, and $m'(c)$ is false, and there is a proof of $I \vdash \langle \neg c \Rightarrow i \rangle s \cdot t$. By Lemma 1, $m'(\neg c \Rightarrow i)$ is true, which we can simplify to $m'(\neg c) \Rightarrow m'(i)$, then to $m(i)$ is true.

## 3.4 Simulation Invariants

We next present the logic that the compiler uses to prove simulation invariants between two programs $P$ and $T$. 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 $\{o_1^P, \ldots, o_n^P\}$ and $\{o_1^T, \ldots, o_n^T\}$ of externally observable variables and that corresponding externally observable 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 $1 \le i \le n$.

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. Simulation invariants consist of two partial simulation invariants that together express a simulation relationship between the partial executions of the programs. For example, $\langle c_1, e_1 \rangle s_1 \rhd \langle c_2, e_2 \rangle s_2$ is true if for all partial executions of $P$ that reach $s_1$ with the condition $c_1$ true, there exists a partial execution of $T$ that reaches $s_2$ with $c_2$ true such that $e_1 = e_2$. Like the logic for standard invariants presented in Section 3.3, 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, $\langle \text{true}, e\rangle t$, $s\langle \text{true}, e\rangle t$, or $\langle \text{true}, e\rangle s\cdot t$.

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

Figures 11, 12, and 13 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 13 or rule 14. The rules in Figure 12 propagate the partial simulation invariant from the original program; the rules in Figure 13 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 $\langle c_1, e_1\rangle s_1 \rhd \langle c_2, e_2\rangle s_2$. 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 15 requires that the simulation invariant hold on all paths in the original program. Rule 22 requires only that the simulation invariant hold on one path in the transformed program. This difference reflects the asymmetry in the implicit quantifiers of the simulation invariant, which is true if for all paths in the original program, there exists a path in the transformed program that satisfies the appropriate conditions.

$$\frac{(o_1^P, \ldots, o_n^P) = (o_1^T, \ldots, o_n^T) \wedge c_1 \Rightarrow c_2 \wedge e_1 = e_2}{I \vdash \langle c_1, e_1\rangle s_0^P \rhd \langle c_2, e_2\rangle s_0^T} \quad \textit{base} \qquad (13)$$

$$\frac{\begin{array}{c} I \vdash \langle i_1\rangle s_1, I \vdash \langle i_2\rangle s_2, \langle c_1', e_1'\rangle s_1 \rhd \langle c_2', e_2'\rangle s_2 \in I, \\ i_1 \wedge c_1 \Rightarrow c_1', i_1 \wedge i_2 \wedge c_1 \wedge e_1' = e_2' \Rightarrow (c_2 \wedge e_1 = e_2) \end{array}}{I \vdash \langle c_1, e_1\rangle s_1 \cdot t \rhd \langle c_2, e_2\rangle s_2} \quad \textit{induction} \qquad (14)$$

Figure 11: Simulation Invariant Base and Induction Proof Rules

### 3.4.1 The Simulation Condition

To prove that the transformed program simulates 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 concepts as follows:

**Definition 5** *A transformed program $T$ simulates 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 $\langle c_1, e_1\rangle s_1 \rhd \langle c_2, e_2\rangle s_2 \in I$, $I \vdash \langle c_1, e_1\rangle s_1 \rhd \langle c_2, e_2\rangle s_2$, and*

$$\frac{\mathrm{pred}(t) \neq \emptyset, \forall s \in \mathrm{pred}(t).\ I \vdash s\langle c, e\rangle t \rhd p}{I \vdash \langle c, e\rangle t \rhd p} \quad \textit{orig-pred} \tag{15}$$

$$\frac{s : \mathrm{nop}\ t, I \vdash \langle c, e\rangle s \cdot t \rhd p}{I \vdash s\langle c, e\rangle t \rhd p} \quad \textit{orig-nop} \tag{16}$$

$$\frac{s : v \leftarrow e'\ t, I \vdash \langle c[e'/v], e[e'/v]\rangle s \cdot t \rhd p}{I \vdash s\langle c, e\rangle t \rhd p} \quad \textit{orig-assign} \tag{17}$$

$$\frac{s : \mathrm{br}\ c'\ t_1\ t_2, I \vdash \langle c \wedge c', e\rangle s \cdot t_1 \rhd p}{I \vdash s\langle c, e\rangle t_1 \rhd p} \quad \textit{orig-brtrue} \tag{18}$$

$$\frac{s : \mathrm{br}\ c'\ t_1\ t_2, I \vdash \langle c \wedge \neg c', e\rangle s \cdot t_2 \rhd p}{I \vdash s\langle c, e\rangle t_2 \rhd p} \quad \textit{orig-brfalse} \tag{19}$$

$$\frac{I \vdash \langle c_1, e\rangle s \cdot t \rhd p, I \vdash \langle c_2, e\rangle s \cdot t \rhd p, c \Rightarrow c_1 \vee c_2}{I \vdash \langle c, e\rangle s \cdot t \rhd p} \quad \textit{orig-case} \tag{20}$$

$$\frac{I \vdash \langle c, e\rangle s \rhd p}{I \vdash \langle c, e\rangle s \cdot t \rhd p} \quad \textit{orig-step} \tag{21}$$

Figure 12: Proof Rules for the Original Program $P$

$$\frac{\exists s \in \mathrm{pred}(t).\ I \vdash p \rhd s\langle c, e\rangle t}{I \vdash p \rhd \langle c, e\rangle t} \quad \textit{trans-pred} \tag{22}$$

$$\frac{s : \mathrm{nop}\ t, I \vdash p \rhd \langle c, e\rangle s}{I \vdash p \rhd s\langle c, e\rangle t} \quad \textit{trans-nop} \tag{23}$$

$$\frac{s : v \leftarrow e'\ t, I \vdash p \rhd \langle c[e'/v], e[e'/v]\rangle s}{I \vdash p \rhd s\langle c, e\rangle t} \quad \textit{trans-assign} \tag{24}$$

$$\frac{s : \mathrm{br}\ c'\ t_1\ t_2, I \vdash p \rhd \langle c \wedge c', e\rangle s}{I \vdash p \rhd s\langle c, e\rangle t_1} \quad \textit{trans-brtrue} \tag{25}$$

$$\frac{s : \mathrm{br}\ c'\ t_1\ t_2, I \vdash p \rhd \langle c \wedge \neg c', e\rangle s}{I \vdash p \rhd s\langle c, e\rangle t_2} \quad \textit{trans-brfalse} \tag{26}$$

Figure 13: Proof Rules for the Transformed Program $T$

- *the simulation invariant* $\langle(o_1^P,\ldots,o_n^P)\rangle s_x^P \rhd \langle(o_1^T,\ldots,o_n^T)\rangle s_x^T \in I$*, 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* $T$*.*

### 3.4.2 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. A standard form proof has the following structure. Each leaf in the proof tree is a use of rule 13 or 14. Along each path in the proof tree from the leaves towards the root, the proof first uses rules 22 through 26 to propagate the partial simulation invariant from the transformed program through the program. Note that in this phase of the proof tree, each rule use has exactly one child. Next, uses of rules 15 through 21 appear on the path. These uses propagate the partial simulation invariant from the original program $P$. Because the proof must verify the simulation invariant for all paths in the original program, uses of rule 5 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 22 through 26 precede all uses of rules 15 through 21.*

**Theorem 2** *If* $I \vdash p_1 \rhd p_2$*, then there exists a proof of* $I \vdash p_1 \rhd p_2$ *that is in standard form.*

**Proof:** Induction on the depth of the proof of $I \vdash p_1 \rhd p_2$.
**Base:** The proof is a use of rule 13 or 14. By definition of standard form, the proof is in standard form.
**Induction Step:** We assume that the proof is in standard form except for the last rule, then find an equivalent proof in standard form. We do a case analysis of the last rule.

- The last rule is one of 15 through 21. By definition of standard form, the proof is in standard form.

- The last rule is one of 22 through 26. The proof is in standard form unless the next-to-last rule is one of 15 through 21. We do a case analysis on the next-to-last rule.

  - The next-to-last rule is 21 or one of 16 through 19. Then the proof is of the form:

$$\dfrac{\dfrac{\dfrac{\pi}{I \vdash p_1' \rhd p_2'}}{I \vdash p_1 \rhd p_2'}}{I \vdash p_1 \rhd p_2}$$

    We can switch the last two rules of the proof to obtain the following equivalent proof:

$$\dfrac{\dfrac{\dfrac{\pi}{I \vdash p_1' \rhd p_2'}}{I \vdash p_1' \rhd p_2}}{I \vdash p_1 \rhd p_2}$$

By the induction hypothesis, we can obtain an equivalent standard form proof $\pi'$ for the proof

$$\dfrac{\dfrac{\pi}{I \vdash p_1' \rhd p_2'}}{I \vdash p_1' \rhd p_2}$$

The following proof is then in standard form:

$$\dfrac{\pi'}{I \vdash p_1 \rhd p_2}$$

– The next-to-last rule is 15. Then the proof is of the form:

$$\dfrac{\dfrac{\dfrac{\pi_1}{I \vdash p_1^1 \rhd p_2'} \quad \cdots \quad \dfrac{\pi_n}{I \vdash p_1^n \rhd p_2'}}{I \vdash p_1 \rhd p_2'}}{I \vdash p_1 \rhd p_2}$$

We can push the last rule of the proof through rule 15 to convert the proof to an equivalent proof of the form:

$$\dfrac{\dfrac{\dfrac{\pi_1}{I \vdash p_1^1 \rhd p_2'}}{I \vdash p_1^1 \rhd p_2} \quad \cdots \quad \dfrac{\dfrac{\pi_n}{I \vdash p_1^n \rhd p_2'}}{I \vdash p_1^n \rhd p_2}}{I \vdash p_1 \rhd p_2}$$

By the induction hypothesis, we can obtain a standard form proof $\pi_i'$ for each of the proofs

$$\dfrac{\dfrac{\pi_i}{I \vdash p_1^i \rhd p_2'}}{I \vdash p_1^i \rhd p_2}$$

then use these standard form proofs to construct the following standard form proof of $I \vdash p_1 \rhd p_2$:

$$\dfrac{\pi_1' \cdots \pi_n'}{I \vdash p_1 \rhd p_2}$$

– The next-to-last rule is 20. Then the proof is of the form:

$$\dfrac{\dfrac{\dfrac{\pi_1}{I \vdash \langle c_1, e\rangle s \cdot t \rhd p'} \quad \dfrac{\pi_2}{I \vdash \langle c_2, e\rangle s \cdot t \rhd p'} \quad c \Rightarrow c_1 \vee c_2}{I \vdash \langle c, e\rangle s \cdot t \rhd p'}}{I \vdash \langle c, e\rangle s \cdot t \rhd p}$$

We can push the last rule of the proof through rule 20 to convert the proof to an equivalent proof of the form:

$$\dfrac{\dfrac{\dfrac{\pi_1}{I \vdash \langle c_1, e\rangle s \cdot t \rhd p'}}{I \vdash \langle c_1, e\rangle s \cdot t \rhd p} \quad \dfrac{\dfrac{\pi_2}{I \vdash \langle c_2, e\rangle s \cdot t \rhd p'}}{I \vdash \langle c_2, e\rangle s \cdot t \rhd p} \quad c \Rightarrow c_1 \vee c_2}{I \vdash \langle c, e\rangle s \cdot t \rhd p}$$

By the induction hypothesis, we can obtain standard form proofs $\pi_1'$ and $\pi_2'$ for the two proofs

$$\frac{\dfrac{\pi_1}{I \vdash \langle c_1, e\rangle s \cdot t \rhd p'}}{I \vdash \langle c_1, e\rangle s \cdot t \rhd p} \quad \frac{\dfrac{\pi_2}{I \vdash \langle c_2, e\rangle s \cdot t \rhd p'}}{I \vdash \langle c_2, e\rangle s \cdot t \rhd p}$$

then use these standard form proofs to construct the following standard form proof of $I \vdash p_1 \rhd p_2$:

$$\frac{\pi_1' \quad \pi_2' \quad c \Rightarrow c_1 \vee c_2}{I \vdash p_1 \rhd p_2}$$

### 3.4.3 Soundness of Proof Rules for Simulation Invariants

We next show that the proof rules for simulation invariants are sound. We first prove 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 $\langle c_1, e_1\rangle s_1 \rhd \langle c_2, e_2\rangle s_2 \in I$, $I \vdash \langle c_1, e_1\rangle s_1 \rhd \langle c_2, e_2\rangle s_2$. Assume a standard form proof of $I \vdash p \rhd \langle c_2, e_2\rangle s_2$ whose last rule is one of 13, 14 or 22, where $p = \langle c_1, e_1\rangle s_1$ or $p = \langle c_1, e_1\rangle s_1 \cdot t$. Also assume a partial execution $\langle s_0^P, m_0^P\rangle \to \cdots \to \langle s_1, m_1\rangle$ such that $m_1(c_1)$ is* true*. If $p = \langle c_1, e_1\rangle s_1 \cdot t$, also assume that $I \vdash \langle c', e'\rangle s_1 \rhd \langle c, e\rangle s$ and $m_1(c')$ is* true *implies that there exists a partial execution $\langle s_0^T, m_0^T\rangle \to \cdots \to \langle s, m\rangle$ such that $m(c)$ is* true *and $m_1(e') = m(e)$. Then there exists 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 *and $m_1(e_1) = m_2(e_2)$.*

**Proof:** Induction on the length of the proof of $I \vdash p \rhd \langle c_2, e_2\rangle s_2$.
**Base:** The proof consists of a use of either rule 13 or rule 14. We do a case analysis of this rule.

- The proof is a use of rule 13 with $(o_1^P, \ldots, o_n^P) = (o_1^T, \ldots, o_n^T) \Rightarrow c_2 \wedge e_1 = e_2$. Then $m_1 = m_0^P$ and $m_2 = m_0^T$, which implies $m_1(e_1) = m_2(e_2)$ and $m_1(c_1) \Rightarrow m_2(c_2)$. Because $m_1(c_1)$ is true, $m_2(c_2)$ is true.

- The proof is a use of rule 14 with $p = \langle c_1, e_1\rangle s_1 \cdot t$, $I \vdash \langle i_1\rangle s_1$, $I \vdash \langle i_2\rangle s_2$, $\langle c_1', e_1'\rangle s_1 \rhd \langle c_2', e_2'\rangle s_2 \in I$, $i_1 \wedge c_1 \Rightarrow c_1'$ and $i_1 \wedge i_2 \wedge c_1 \wedge e_1' = e_2' \Rightarrow (c_2 \wedge e_1 = e_2)$. By assumption $m_1(c_1)$ is true, by Theorem 1 $m_1(i_1)$ is true, so $i_1 \wedge c_1 \Rightarrow c_1'$ implies $m_1(c_1')$ is true. By assumption, $\langle c_1', e_1'\rangle s_1 \rhd \langle c_2', e_2'\rangle s_2 \in I$ implies that $I \vdash \langle c_1', e_1'\rangle s_1 \rhd \langle c_2', e_2'\rangle s_2$, so there exists a partial execution $\langle s_0^T, m_0^T\rangle \to \cdots \to \langle s, m\rangle$ such that $m(c_2')$ is true and $m_1(e_1') = m(e_2')$. By Theorem 1 $m(i_1)$ is true. Let $m_2 = m$. Then $i_1 \wedge i_2 \wedge c_1 \wedge e_1' = e_2' \Rightarrow (c_2 \wedge e_1 = e_2)$ implies $m_1(i_1) \wedge m_2(i_2) \wedge m_1(c_1) \wedge m_1(e_1') = m_2(e_2') \Rightarrow (m_2(c_2) \wedge m_1(e_1) = m_2(e_2))$, which can be simplified to obtain $m_2(c_2)$ is true and $m_1(e_1) = m_2(e_2)$.

**Induction:** We do a case analysis of the last rule of the proof. Because the proof is at least two rules deep, the last rule cannot be rule 13 or 14. So the last rule must be 22. In this case there is standard form proof of $I \vdash p \rhd s\langle c_2, e_2\rangle s_2$. We do a case analysis of the last rule of this proof. Because the proof is in standard form, rules 23, 24, 25, and 26 are the only possibilities.

- The last rule is 23 with $s$ : nop $s_2$. There is a standard form proof $\pi$ of $I \vdash p \rhd \langle c_2, e_2 \rangle s$. Consider the last rule in $\pi$. Because this proof can be extended using rule 23, then rule 22 to a standard form proof of $I \vdash p \rhd \langle c_2, e_2 \rangle s_2$, the last rule in $\pi$ is not one of rules 15 through 21. The only other rules that are of the correct form are rules 13, 14, and 22. By the induction hypothesis, there exists a partial execution $\langle s_0^T, m_0^T \rangle \to \cdots \to \langle s, m \rangle$ such that $m(c_2)$ is true and $m_1(e_1) = m(e_2)$. We can extend this partial execution to a partial execution $\langle s_0^T, m_0^T \rangle \to \cdots \to \langle s, m \rangle \to \langle s_2, m_2 \rangle$, where $m_2 = m$. Then $m_2(c_2)$ is true and $m_1(e_1) = m_2(e_2)$.

- The last rule is 24 with $s : v \leftarrow e\ s_2$. There is a standard form proof $\pi$ of $I \vdash p \rhd \langle c_2[e/v], e_2[e/v] \rangle s$. Consider the last rule in $\pi$. Because this proof can be extended using rule 24, then rule 22 to a standard form proof of $I \vdash p \rhd \langle c_2, e_2 \rangle s_2$, the last rule in $\pi$ is not one of rules 15 through 21. The only other rules that are of the correct form are rules 13, 14, and 22. By the induction hypothesis, there exists a partial execution $\langle s_0^T, m_0^T \rangle \to \cdots \to \langle s, m \rangle$ such that $m(c_2[e/v])$ is true and $m_1(e_1) = m(e_2[e/v])$. We can extend this partial execution to a partial execution $\langle s_0^T, m_0^T \rangle \to \cdots \to \langle s, m \rangle \to \langle s_2, m_2 \rangle$, where $m_2 = m[v \mapsto m(e)]$. We can then simplify $m(c_2[e/v])$ is true to $m[v \mapsto m(e)](c_2)$ is true, then to $m_2(c_2)$ is true. Similarly, we can simplify $m_1(e_1) = m(e_2[e/v])$ to $m_1(e_1) = m_2(e_2)$.

- The last rule is rule 25 with $s$ : br $c\ s_2\ t$. There is a standard form proof of $I \vdash p \rhd \langle c_2 \wedge c, e_2 \rangle s$ whose last rule is 13, 14, or 22. By the induction hypothesis, there exists a partial execution $\langle s_0^T, m_0^T \rangle \to \cdots \to \langle s, m \rangle$ such that $m(c_2 \wedge c)$ is true and $m_1(e_1) = m(e_2)$. Because $m(c)$ is true, we can extend this partial execution to a partial execution $\langle s_0^T, m_0^T \rangle \to \cdots \to \langle s, m \rangle \to \langle s_2, m_2 \rangle$, where $m_2 = m$, and obtain $m_2(c_2)$ is true and $m_1(e_1) = m_2(e_2)$.

- The last rule is rule 26 with $s$ : br $c\ t\ s_2$. There is a standard form proof of $I \vdash p \rhd \langle c_2 \wedge \neg c, e_2 \rangle s$ whose last rule is 13, 14, or 22. By the induction hypothesis, there exists a partial execution $\langle s_0^T, m_0^T \rangle \to \cdots \to \langle s, m \rangle$ such that $m(c_2 \wedge \neg c)$ is true and $m_1(e_1) = m(e_2)$. Because $m(c)$ is false, we can extend this partial execution to a partial execution $\langle s_0^T, m_0^T \rangle \to \cdots \to \langle s, m \rangle \to \langle s_2, m_2 \rangle$, where $m_2 = m$, and obtain $m_2(c_2)$ is true and $m_1(e_1) = m_2(e_2)$.

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

**Proof:** Consider the proof tree of $I \vdash \langle c_1, e_1 \rangle s_1 \cdot t \rhd \langle c_2, e_2 \rangle s_2$. Given a path in this tree from the root to a leaf, we can start at the root and compute the number of consecutive uses of rule 20 until the first use of a different rule. We call this number the case analysis number of the path. If, for example, the last rule in the proof is not a use of rule 20, then the root is not a use of rule 20 and, for all paths, the case analysis number is zero. The

proof is by induction on the maximum over all paths from the root to a leaf of the case analysis number of the path.
**Base:** In this case the last rule of the proof is not 20. The only other rules that are of the correct form are rules 14, 22, and 21. We do a case analysis on the last rule of the proof:

- The last rule is 14 or 22. By Lemma 2, there exists 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 and $m_1(e_1) = m_2(e_2)$.

- The last rule is 21, with $I \vdash \langle c_1, e_1\rangle s_1 \rhd \langle c_2, e_2\rangle s_2$. Then by assumption, there exists 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 and $m_1(e_1) = m_2(e_2)$.

**Induction:** Assume that the proof has maximum case analysis number of $k$, where $k$ is at least one, and the lemma holds for all proofs with maximum case analysis number less than $k$. In this case the last rule of the proof is 20 with $c_1 \Rightarrow c_1^1 \vee c_1^2$, and proofs of $I \vdash \langle c_1^1, e_1\rangle s_1 \cdot t \rhd \langle c_2, e_2\rangle s_2$, $I \vdash \langle c_1^2, e_1\rangle s_1 \cdot t \rhd \langle c_2, e_2\rangle s_2$. Note that these proofs have a maximum case analysis number less than $k$. If we can show that either $m_1(c_1^1)$ is true or $m_2(c_1^2)$ is true, we can apply the induction hypothesis to one of the proofs.

Note that $c_1 \Rightarrow c_1^1 \vee c_1^2$ implies $m_1(c_1) \Rightarrow m_1(c_1^1) \vee m_1(c_1^2)$. Because $m_1(c_1)$ is true, either $m_1(c_1^1)$ is true or $m_1(c_1^2)$ is true. Then by the induction hypothesis, there exists 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 and $m_1(e_1) = m_2(e_2)$.

**Theorem 3** *Assume that for all $\langle i\rangle s \in I$, $I \vdash \langle i\rangle s$ and for all $\langle c_1, e_1\rangle s_1 \rhd \langle c_2, e_2\rangle s_2 \in I$, $I \vdash \langle c_1, e_1\rangle s_1 \rhd \langle c_2, e_2\rangle s_2$. Then for all standard form proofs of $I \vdash \langle c_1, e_1\rangle s_1 \rhd \langle c_2, e_2\rangle s_2$ and for all partial executions $\langle s_0^P, m_0^P\rangle \to \cdots \to \langle s_1, m_1\rangle$ such that $m_1(c_1)$ is* true*, there exists 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 *and $m_1(e_1) = m_2(e_2)$.*

**Proof:** Induction on the length of the partial execution $\langle s_0^P, m_0^P\rangle \to \cdots \to \langle s_1, m_1\rangle$.
**Base:** If the length is 0, then $s_1 = s_0^P$ and $\mathrm{pred}(s_1) = \emptyset$. We do a case analysis of the last rule of the proof of $I \vdash \langle c_1, e_1\rangle s_1 \rhd \langle c_2, e_2\rangle s_2$. The only rules that are of the correct form are rules 13, 15, and 22. Because $\mathrm{pred}(s_1) = \emptyset$, rule 15 cannot be the last rule.

- The last rule is 13 or 22. Then by Lemma 2, there exists 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 and $m_1(e_1) = m_2(e_2)$.

**Induction:** In this case the partial execution of $P$ is at least one step long, so we can write it as $\langle s_0^P, m_0^P\rangle \to \cdots \to \langle s, m\rangle \to \langle s_1, m_1\rangle$. We do a case analysis of the last rule of the proof of $I \vdash \langle c_1, e_1\rangle s_1 \rhd \langle c_2, e_2\rangle s_2$. The only rules that are of the correct form are rules 13, 15, and 22.

- The last rule is 13 or 22. By Lemma 2, there exists 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 and $m_1(e_1) = m_2(e_2)$.

- The last rule is 15. Because $s \in \mathrm{pred}(s_1)$, there is a standard form proof of $I \vdash s\langle c_1, e_1\rangle s_1 \rhd \langle c_2, e_2\rangle s_2$. We do a case analysis of the last rule in this proof. Because the proof is in standard form, 22 is not the last rule. The only other rules that are of the correct form are 16, 17, 18, and 19.

– The last rule is 16, with $s$ : nop $s_1$. Then $m_1 = m$. By assumption $m_1(c_1)$ is true, which implies that $m(c_1)$ is true. There is also a standard form proof of $I \vdash \langle c_1, e_1\rangle s \cdot s_1 \rhd \langle c_2, e_2\rangle s_2$. By Lemma 3, there exists 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 and $m(e_1) = m_2(e_2)$, which implies that $m_1(e_1) = m_2(e_2)$.

– The last rule is 17, with $s : v \leftarrow e\ s_1$. Then $m_1 = m[v \mapsto m(e)]$. By assumption $m_1(c_1)$ is true, which implies that $m(c_1[e/v])$ is true. There is also a standard form proof of $I \vdash \langle c_1[e/v], e_1[e/v]\rangle s \cdot s_1 \rhd \langle c_2, e_2\rangle s_2$. By Lemma 3, there exists 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 and $m(e_1[e/v]) = m_2(e_2)$, which we can simplify to $m_1(e_1) = m_2(e_2)$.

– The last rule is 18, with $s$ : br $c\ s_1\ t$. Then $m_1 = m$ and $m(c)$ is true. By assumption $m_1(c_1)$ is true, which implies that $m(c_1 \wedge c)$ is true. There is also a standard form proof of $I \vdash \langle c_1 \wedge c, e_1\rangle s \cdot s_1 \rhd \langle c_2, e_2\rangle s_2$. By Lemma 3, there exists 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 and $m_1(e_1) = m_2(e_2)$.

– The last rule is 19, with $s$ : br $c\ t\ s_1$. Then $m_1 = m$ and $m(c)$ is false. By assumption $m_1(c_1)$ is true, which implies that $m(c_1 \wedge \neg c)$ is true. There is also a standard form proof of $I \vdash \langle c_1 \wedge \neg c, e_1\rangle s \cdot s_1 \rhd \langle c_2, e_2\rangle s_2$. By Lemma 3, there exists 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 and $m_1(e_1) = m_2(e_2)$.

# 4 Optimization Schemas

We next present examples that illustrate how to prove the correctness of a variety of standard optimizations. 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.

## 4.1 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 14 and 15 present an example that we use to illustrate the schema. This example continues the example introduced in Section 2. Figure 16 presents the invariants that the compiler generates for this example.

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 17, which contains the proof tree for $\langle (g_P, i_P)\rangle 4_P \rhd \langle (g_T, i_T)\rangle 4_T$, illustrates this situation.

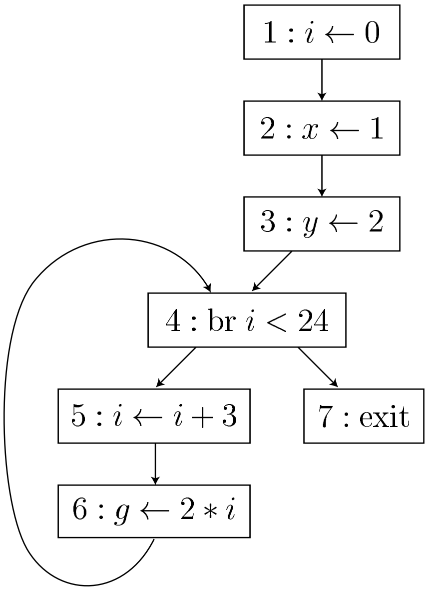

Figure 14: Program $P$ Before Dead Assignment Elimination

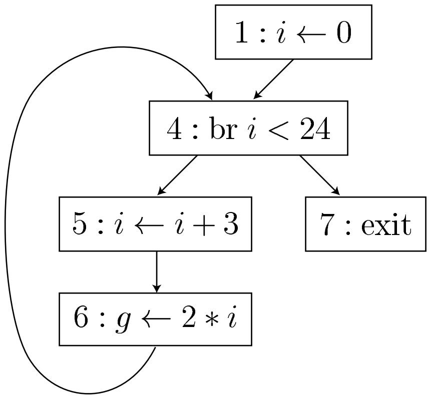

Figure 15: Program $T$ After Dead Assignment Elimination

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

Figure 16: Invariants for Dead Assignment Elimination

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(g_P) = (g_T) \Rightarrow (g_P, 0) = (g_T, 0)}{I \vdash \langle(g_P, 0)\rangle 1_P \rhd \langle(g_T, 0)\rangle 1_T}}{I \vdash \langle(g_P, 0)\rangle 1_P \rhd 1_T\langle(g_T, i_T)\rangle 4_T}}{I \vdash \langle(g_P, 0)\rangle 1_P \rhd \langle(g_T, i_T)\rangle 4_T}}{I \vdash \langle(g_P, 0)\rangle 1_P \cdot 2_P \rhd \langle(g_T, i_T)\rangle 4_T}}{I \vdash 1_P\langle(g_P, i_P)\rangle 2_P \rhd \langle(g_T, i_T)\rangle 4_T}}{I \vdash \langle(g_P, i_P)\rangle 2_P \rhd \langle(g_T, i_T)\rangle 4_T}}{I \vdash \langle(g_P, i_P)\rangle 2_P \cdot 3_P \rhd \langle(g_T, i_T)\rangle 4_T}}{I \vdash 2_P\langle(g_P, i_P)\rangle 3_P \rhd \langle(g_T, i_T)\rangle 4_T}}{I \vdash \langle(g_P, i_P)\rangle 3_P \rhd \langle(g_T, i_T)\rangle 4_T}}{I \vdash \langle(g_P, i_P)\rangle 3_P \cdot 4_P \rhd \langle(g_T, i_T)\rangle 4_T}}{I \vdash 3_P\langle(g_P, i_P)\rangle 4_P \rhd \langle(g_T, i_T)\rangle 4_T} \quad \dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\langle i_P\rangle 6_P \rhd \langle i_T\rangle 6_T \in I, \\ i_P = i_T \Rightarrow (2 * i_P, i_P) = (2 * i_T, i_T)\end{array}}{I \vdash \langle(2 * i_P, i_P)\rangle 6_P \cdot 4_P \rhd \langle(2 * i_T, i_T)\rangle 6_T}}{I \vdash \langle(2 * i_P, i_P)\rangle 6_P \cdot 4_P \rhd 6_T\langle(g_T, i_T)\rangle 4_T}}{I \vdash \langle(2 * i_P, i_P)\rangle 6_P \cdot 4_P \rhd \langle(g_T, i_T)\rangle 4_T}}{I \vdash 6_P\langle(g_P, i_P)\rangle 4_P \rhd \langle(g_T, i_T)\rangle 4_T}}{I \vdash \langle(g_P, i_P)\rangle 4_P \rhd \langle(g_T, i_T)\rangle 4_T}$$

Figure 17: Proof Tree for $I \vdash \langle(g_P, i_P)\rangle 4_P \rhd \langle(g_T, i_T)\rangle 4_T$

## 4.2 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 18 presents the program before branch movement; Figure 19 presents the program after branch movement. Figure 20 presents the set of invariants that the compiler generates for this example.

Figure 23 presents the proof tree for $I \vdash \langle g_P \rangle 7_P \rhd \langle g_T \rangle 7_T$. One of the paths that the proof must consider is the path in the original program $P$ from $1_P$ to $4_P$ to $7_P$. No execution of $P$, of course, will take this path — the loop always executes at least once, and this path corresponds to the loop executing zero times. The fact that this path will never execute shows up as a false condition in the partial simulation invariant for $P$ that is propagated from $7_P$ back to $1_P$. The corresponding path in $T$ that is used to prove $I \vdash \langle g_P \rangle 7_P \rhd \langle g_T \rangle 7_T$ is the path from $1_T$ through $5_T$, $6_T$, and $4_T$ to $7_T$. Although the values of $g_P$ and $g_T$ are not the same on the two paths, the fact that the condition in the partial simulation invariant from $P$ is false enables the use of rule 13 at the leaf of the proof tree. Figure 21 presents the branch of the proof tree for this path.

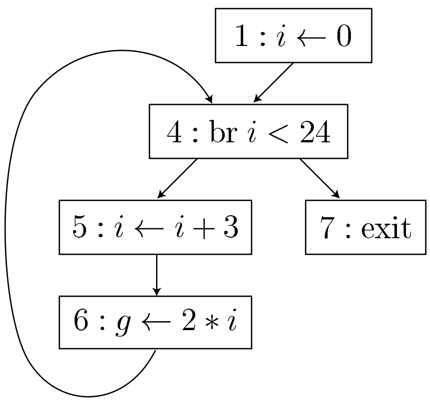


Figure 18: Program $P$ Before Branch Movement

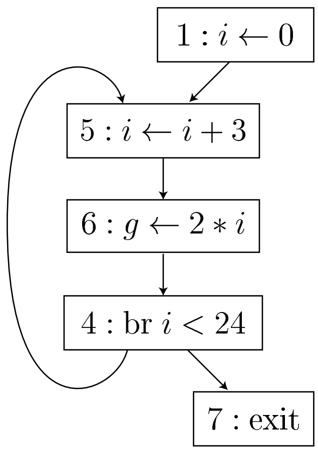


Figure 19: Program $T$ After Branch Movement

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

Figure 20: Invariants for Branch Movement

$$
\begin{array}{c}
0 \geq 24 \Rightarrow 3 \geq 24 \wedge g_P = 6 \\
\hline
I \vdash \langle 0 \geq 24, g_P \rangle 1_P \rhd \langle 3 \geq 24, 6 \rangle 1_T \\
\hline
I \vdash \langle 0 \geq 24, g_P \rangle 1_P \rhd 1_T \langle i_T + 3 \geq 24, 2 * (i_T + 3) \rangle 5_T \\
\hline
I \vdash \langle 0 \geq 24, g_P \rangle 1_P \rhd \langle i_T + 3 \geq 24, 2 * (i_T + 3) \rangle 5_T \\
\hline
I \vdash \langle 0 \geq 24, g_P \rangle 1_P \rhd 5_T \langle i_T \geq 24, 2 * i_T \rangle 6_T \\
\hline
I \vdash \langle 0 \geq 24, g_P \rangle 1_P \rhd \langle i_T \geq 24, 2 * i_T \rangle 6_T \\
\hline
I \vdash \langle 0 \geq 24, g_P \rangle 1_P \rhd 6_T \langle i_T \geq 24, g_T \rangle 4_T \\
\hline
I \vdash \langle 0 \geq 24, g_P \rangle 1_P \rhd \langle i_T \geq 24, g_T \rangle 4_T \\
\hline
I \vdash \langle 0 \geq 24, g_P \rangle 1_P \rhd 4_T \langle g_T \rangle 7_T \\
\hline
I \vdash \langle 0 \geq 24, g_P \rangle 1_P \rhd \langle g_T \rangle 7_T \\
\hline
I \vdash \langle 0 \geq 24, g_P \rangle 1_P \cdot 4_P \rhd \langle g_T \rangle 7_T \\
\hline
I \vdash 1_P \langle i_P \geq 24, g_P \rangle 4_P \rhd \langle g_T \rangle 7_T
\end{array}
$$

Figure 21: Proof Tree $\pi_1$ for $I \vdash 1_P \langle i_P \geq 24, g_P \rangle 4_P \rhd \langle g_T \rangle 7_T$

$$
\begin{array}{c}
\langle i_P \rangle 6_P \rhd \langle i_T \rangle 6_T \in I, i_P \geq 24 \wedge i_P = i_T \Rightarrow (i_T \geq 24 \wedge 2 * i_P = 2 * i_T) \\
\hline
I \vdash \langle i_P \geq 24, 2 * i_P \rangle 6_P \cdot 4_P \rhd \langle i_T \geq 24, 2 * i_T \rangle 6_T \\
\hline
I \vdash \langle i_P \geq 24, 2 * i_P \rangle 6_P \cdot 4_P \rhd 6_T \langle i_T \geq 24, g_T \rangle 4_T \\
\hline
I \vdash \langle i_P \geq 24, 2 * i_P \rangle 6_P \cdot 4_P \rhd \langle i_T \geq 24, g_T \rangle 4_T \\
\hline
I \vdash \langle i_P \geq 24, 2 * i_P \rangle 6_P \cdot 4_P \rhd 4_T \langle g_T \rangle 7_T \\
\hline
I \vdash \langle i_P \geq 24, 2 * i_P \rangle 6_P \cdot 4_P \rhd \langle g_T \rangle 7_T \\
\hline
I \vdash 6_P \langle i_P \geq 24, g_P \rangle 4_P \rhd \langle g_T \rangle 7_T
\end{array}
$$

Figure 22: Proof Tree $\pi_2$ for $I \vdash 6_P \langle i_P \geq 24, g_P \rangle 4_P \rhd \langle g_T \rangle 7_T$

$$
\begin{array}{c}
\pi_1 \quad \pi_2 \\
\hline
I \vdash \langle i_P \geq 24, g_P \rangle 4_P \rhd \langle g_T \rangle 7_T \\
\hline
I \vdash \langle i_P \geq 24, g_P \rangle 4_P \cdot 7_P \rhd \langle g_T \rangle 7_T \\
\hline
I \vdash 4_P \langle g_P \rangle 7_P \rhd \langle g_T \rangle 7_T \\
\hline
I \vdash \langle g_P \rangle 7_P \rhd \langle g_T \rangle 7_T
\end{array}
$$

Figure 23: Proof Tree for $I \vdash \langle g_P \rangle 7_P \rhd \langle g_T \rangle 7_T$

## 4.3 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 24 presents the program before induction variable elimination; Figure 25 presents the program after induction variable elimination. Figure 26 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 27 presents the proof tree for $I \vdash \langle 2 * i_P \rangle 4_P \rhd \langle g_T \rangle 4_T$; Figure 28 presents the proof tree for $I \vdash \langle g_P \rangle 7_P \rhd \langle g_T \rangle 7_T$.

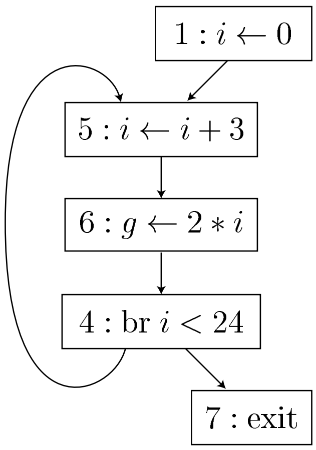


Figure 24: Program $P$ Before Induction Variable Elimination

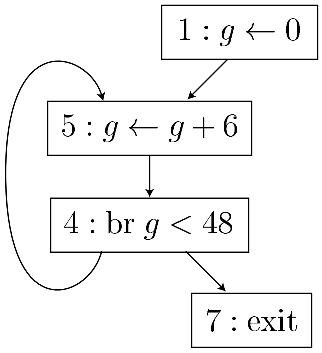


Figure 25: Program $T$ After Induction Variable Elimination

$$I = \{\langle g_P = 2 * i_P \rangle 4_P, \langle 2 * i_P \rangle 5_P \rhd \langle g_T \rangle 5_T, \langle 2 * i_P \rangle 4_P \rhd \langle g_T \rangle 4_T, \langle g_P \rangle 7_P \rhd \langle g_T \rangle 7_T\}$$

Figure 26: Invariants for Induction Variable Elimination

$$
\begin{array}{c}
\langle 2 * i_P \rangle 5_P \rhd \langle g_T \rangle 5_T \in I, 2 * i_P = g_T \Rightarrow 2 * (i_P + 3) = g_T + 6 \\ \hline
I \vdash \langle 2 * (i_P + 3) \rangle 5_P \cdot 6_P \rhd \langle g_T + 6 \rangle 5_T \\ \hline
I \vdash \langle 2 * (i_P + 3) \rangle 5_P \cdot 6_P \rhd 5_T \langle g_T \rangle 4_T \\ \hline
I \vdash \langle 2 * (i_P + 3) \rangle 5_P \cdot 6_P \rhd \langle g_T \rangle 4_T \\ \hline
I \vdash 5_P \langle 2 * i_P \rangle 6_P \rhd \langle g_T \rangle 4_T \\ \hline
I \vdash \langle 2 * i_P \rangle 6_P \rhd \langle g_T \rangle 4_T \\ \hline
I \vdash \langle 2 * i_P \rangle 6_P \cdot 4_P \rhd \langle g_T \rangle 4_T \\ \hline
I \vdash 6_P \langle 2 * i_P \rangle 4_P \rhd \langle g_T \rangle 4_T \\ \hline
I \vdash \langle 2 * i_P \rangle 4_P \rhd \langle g_T \rangle 4_T
\end{array}
$$

Figure 27: Proof Tree for $I \vdash \langle 2 * i_P \rangle 4_P \rhd \langle g_T \rangle 4_T$

$$
\begin{array}{c}
I \vdash \langle g_P = 2 * i_P \rangle 4_P, \langle 2 * i_P \rangle 4_P \rhd \langle g_T \rangle 4_T \in I, \\
g_P = 2 * i_P \land i_P \geq 24 \land 2 * i_P = g_T \Rightarrow (g_T \geq 48 \land g_P = g_T) \\ \hline
I \vdash \langle i_P \geq 24, g_P \rangle 4_P \cdot 7_P \rhd \langle g_T \geq 48, g_T \rangle 4_T \\ \hline
I \vdash \langle i_P \geq 24, g_P \rangle 4_P \cdot 7_P \rhd 4_T \langle g_T \rangle 7_T \\ \hline
I \vdash \langle i_P \geq 24, g_P \rangle 4_P \cdot 7_P \rhd \langle g_T \rangle 7_T \\ \hline
I \vdash 4_P \langle g_P \rangle 7_P \rhd \langle g_T \rangle 7_T \\ \hline
I \vdash \langle g_P \rangle 7_P \rhd \langle g_T \rangle 7_T
\end{array}
$$

Figure 28: Proof Tree for $I \vdash \langle g_P \rangle 7_P \rhd \langle g_T \rangle 7_T$

## 4.4 Loop Unrolling

The next optimization unrolls the loop once. Figure 29 presents the program before loop unrolling; Figure 30 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 dead code elimination optimization discussed in Section 4.5.

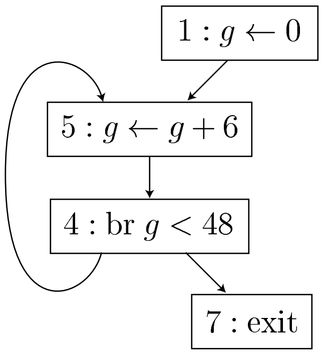


Figure 29: Program $P$ Before Loop Unrolling

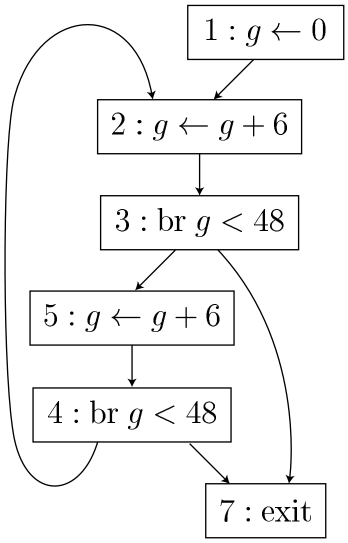


Figure 30: Program $T$ After Loop Unrolling

$$
\begin{aligned}
I \;=\; & \{\langle g_P\%12 = 0 \vee g_P\%12 = 6\rangle 4_P, \langle g_P\%12 = 0, g_P\rangle 5_P \rhd \langle g_T\rangle 2_T, \\
& \langle g_P\%12 = 6, g_P\rangle 4_P \rhd \langle g_T\rangle 3_T, \langle g_P\%12 = 6, g_P\rangle 5_P \rhd \langle g_T\rangle 5_T, \\
& \langle g_P\%12 = 0, g_P\rangle 4_P \rhd \langle g_T\rangle 4_T, \langle g_P\rangle 7_P \rhd \langle g_P\rangle 7_P\}
\end{aligned}
$$

Figure 31: Invariants for Loop Unrolling

Figure 31 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. 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 original 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 34 presents the proof tree for $I \vdash \langle g_P\rangle 7_P \rhd \langle g_T\rangle 7_T$. The key part of the proof is the use of the case analysis rule, rule 20. This rule is a key component of correctness proofs for transformations, like loop unrolling, that replicate code.

$$
\cfrac{\cfrac{\cfrac{\begin{array}{c}\langle g_P\%12 = 0, g_P\rangle 4_P \rhd \langle g_T\rangle 4_T \in I, g_P\%12 = 0 \wedge g_P \geq 48 \Rightarrow g_P\%12 = 0,\\ g_P\%12 = 0 \wedge g_P \geq 48 \wedge g_P = g_T \Rightarrow (g_T \geq 48 \wedge g_P = g_T)\end{array}}{I \vdash \langle g_P\%12 = 0 \wedge g_P \geq 48, g_P\rangle 4_P \cdot 7_P \rhd \langle g_T \geq 48, g_T\rangle 4_T}}{I \vdash \langle g_P\%12 = 0 \wedge g_P \geq 48, g_P\rangle 4_P \cdot 7_P \rhd 4_T\langle g_T\rangle 7_T}}{I \vdash \langle g_P\%12 = 0 \wedge g_P \geq 48, g_P\rangle 4_P \cdot 7_P \rhd \langle g_T\rangle 7_T}
$$

Figure 32: Proof Tree $\pi_1$ for $I \vdash \langle g_P\%12 = 0 \wedge g_P \geq 48, g_P\rangle 4_P \cdot 7_P \rhd \langle g_T\rangle 7_T$

$$
\cfrac{\cfrac{\cfrac{\begin{array}{c}I \vdash \langle g_P\%12 = 0 \vee g_P\%12 = 6\rangle 4_P, \langle g_P\%12 = 6, g_P\rangle 4_P \rhd \langle g_T\rangle 3_T \in I,\\ (g_P\%12 = 0 \vee g_P\%12 = 6) \wedge (g_P\%12 \neq 0 \wedge g_P \geq 48) \Rightarrow g_P\%12 = 6,\\ (g_P\%12 = 0 \vee g_P\%12 = 6) \wedge (g_P\%12 \neq 0 \wedge g_P \geq 48) \wedge g_P = g_T \Rightarrow (g_T \geq 48 \wedge g_P = g_T)\end{array}}{I \vdash \langle g_P\%12 \neq 0 \wedge g_P \geq 48, g_P\rangle 4_P \cdot 7_P \rhd \langle g_T \geq 48, g_T\rangle 3_T}}{I \vdash \langle g_P\%12 \neq 0 \wedge g_P \geq 48, g_P\rangle 4_P \cdot 7_P \rhd 3_T\langle g_T\rangle 7_T}}{I \vdash \langle g_P\%12 \neq 0 \wedge g_P \geq 48, g_P\rangle 4_P \cdot 7_P \rhd \langle g_T\rangle 7_T}
$$

Figure 33: Proof Tree $\pi_2$ for $I \vdash \langle g_P\%12 \neq 0 \wedge g_P \geq 48, g_P\rangle 4_P \cdot 7_P \rhd \langle g_T\rangle 7_T$

$$
\cfrac{\cfrac{\cfrac{\pi_1 \quad \pi_2 \quad g_P \geq 48 \Rightarrow (g_P\%12 = 0 \wedge g_P \geq 48) \vee (g_P\%12 \neq 0 \wedge g_P \geq 48)}{I \vdash \langle g_P \geq 48, g_P\rangle 4_P \cdot 7_P \rhd \langle g_T\rangle 7_T}}{I \vdash 4_P\langle g_P\rangle 7_P \rhd \langle g_T\rangle 7_T}}{I \vdash \langle g_P\rangle 7_P \rhd \langle g_T\rangle 7_T}
$$

Figure 34: Proof Tree for $I \vdash \langle g_P\rangle 7_P \rhd \langle g_T\rangle 7_T$

## 4.5 Dead Code Elimination

Our next optimization is dead code elimination. We continue with our example by eliminating the branch in the middle of the loop at node 3. Figure 35 presents the program before the branch is eliminated. The key property that allows the compiler to remove the branch is that $g\%12 = 6 \wedge g \leq 48$ at 3, which implies that $g < 48$ at 3. In other words, the condition in the branch is always true. Figure 36 presents the program after the branch is eliminated. Figure 37 presents the set of invariants that the compiler generates for this example.

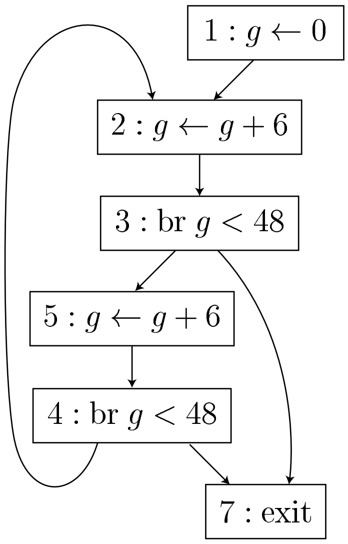


Figure 35: Program $P$ Before Dead Code Elimination

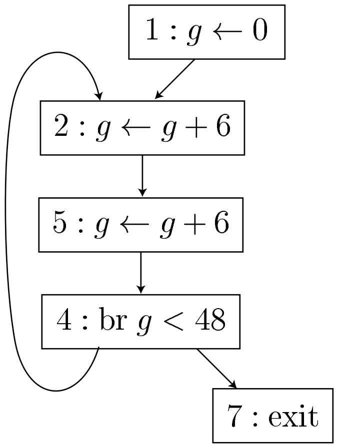


Figure 36: Program $T$ After Dead Code Elimination

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

Figure 37: Invariants for Dead Code Elimination

Figure 40 presents the proof tree for $I \vdash \langle i_P\rangle 7_P \rhd \langle i_T\rangle 7_T$. One of the paths that the proof must consider is the potential loop exit in the original program $P$ from $3_P$ to $7_P$; Figure 39 presents the branch of the proof tree that corresponds to this path. In fact, the loop always exits from $4_P$, not $3_P$. This fact shows up because the conjunction of the standard invariant $\langle g_P\%12 = 6 \wedge g_P \leq 48\rangle 3_P$ with the condition $g_P \geq 48$ from the partial simulation invariant for $P$ at $3_P$ is false. The corresponding path in $T$ that is used to prove $I \vdash \langle i_P\rangle 7_P \rhd \langle i_T\rangle 7_T$ is the path from $5_T$ to $4_T$ to $7_T$. Although the values of $g_P$ and $g_T$ are not the same on the two paths, the fact that the conjunction described above is false enables the use of rule 14 at the leaf of the proof tree.

$$\dfrac{\dfrac{\dfrac{\dfrac{\langle g_P\rangle 4_P \rhd \langle g_T\rangle 4_T \in I,\quad g_P = g_T \wedge g_P \geq 48 \Rightarrow (g_T \geq 48 \wedge g_P = g_T)}{I \vdash \langle g_P \geq 48, g_P\rangle 4_P \cdot 7_P \rhd \langle g_T \geq 48, g_T\rangle 4_T}}{I \vdash \langle g_P \geq 48, g_P\rangle 4_P \cdot 7_P \rhd 4_T\langle g_T\rangle 7_T}}{I \vdash \langle g_P \geq 48, g_P\rangle 4_P \cdot 7_P \rhd \langle g_T\rangle 7_T}}{I \vdash 4_P\langle g_P\rangle 7_P \rhd \langle g_T\rangle 7_T}$$

Figure 38: Proof Tree $\pi_1$ for $I \vdash 4_P\langle g_P\rangle 7_P \rhd \langle g_T\rangle 7_T$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{I \vdash \langle g_P\%12 = 6 \wedge g_P \leq 48\rangle 3_P, \langle g_P\rangle 3_P \rhd \langle g_T\rangle 5_T \in I,\quad g_P\%12 = 6 \wedge g_P \leq 48 \wedge g_P \geq 48 \wedge g_P = g_T \Rightarrow (g_T + 6 \geq 48 \wedge g_P = g_T + 6)}{I \vdash \langle g_P \geq 48, g_P\rangle 3_P \cdot 7_P \rhd \langle g_T + 6 \geq 48, g_T + 6\rangle 5_T}}{I \vdash \langle g_P \geq 48, g_P\rangle 3_P \cdot 7_P \rhd 5_P\langle g_T \geq 48, g_T\rangle 4_T}}{I \vdash \langle g_P \geq 48, g_P\rangle 3_P \cdot 7_P \rhd \langle g_T \geq 48, g_T\rangle 4_T}}{I \vdash \langle g_P \geq 48, g_P\rangle 3_P \cdot 7_P \rhd 4_P\langle g_T\rangle 7_T}}{I \vdash \langle g_P \geq 48, g_P\rangle 3_P \cdot 7_P \rhd \langle g_T\rangle 7_T}}{I \vdash 3_P\langle g_P\rangle 7_P \rhd \langle g_T\rangle 7_T}$$

Figure 39: Proof Tree $\pi_2$ for $I \vdash 3_P\langle g_P\rangle 7_P \rhd \langle g_T\rangle 7_T$

$$\dfrac{\pi_1 \quad \pi_2}{I \vdash \langle g_P\rangle 7_P \rhd \langle g_T\rangle 7_T}$$

Figure 40: Proof Tree for $I \vdash \langle g_P\rangle 7_P \rhd \langle g_T\rangle 7_T$

# 5 Termination Anomalies

Throughout the paper so far, we have required that the transformed program simulate the original program in the sense that for every execution in the original program that reaches the exit node, there exists an execution in the transformed 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 original program contains an infinite loop? Then *any* program simulates the original program. One can imagine that programmers might like to have stronger guarantees.

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 41 and 42 that compute $g = 48$. Using the logic presented in Section 3.4, it is not possible to prove that the iterative program in Figure 41 simulates the program in Figure 42. Roughly speaking, the problem is that the logic cannot prove that the loop in the iterative program terminates.

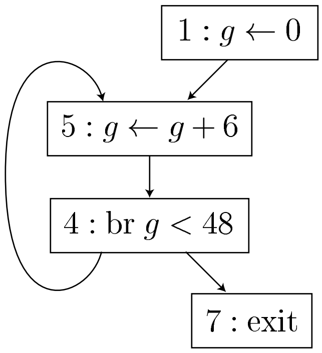


Figure 41: Iterative Program to Compute $g = 48$

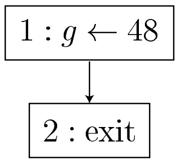


Figure 42: Closed Form Program to Compute $g = 48$

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.

# 6 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.

## 6.1 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.

## 6.2 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.

# 7 Related Work

Most existing research on compiler correctness has focused on techniques that deliver a compiler guaranteed to operate correctly on every input program [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, 7]. 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 [6]?"[1] 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.

# 8 Impact of Credible Compilation

We next discuss the potential impact of credible compilation. We consider five areas: debugging compilers, increasing the flexibility of compiler development, just-in-time compilers, concept of an open compiler, and the relationship of credible compilation to building custom compilers.

## 8.1 Debugging Compilers

Compilers are notoriously difficult to build and debug. In a large compiler, a surprising part of the difficulty is simply recognizing incorrectly generated code. The current state of the art is to generate code after a set of passes, then test that the generated code produces the same result as the original code. Once a piece of incorrect code is found, the developer must spend time tracing the bug back through layers of passes to the original source.

Requiring the compiler to generate a proof for each transformation will dramatically simplify this process. As soon as a pass operates incorrectly, the developer will immediately be directed to the incorrect code. Bugs can be found and eliminated as soon as they occur.

[1] 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.

## 8.2 Flexible Compiler Development

It is difficult, if not impossible, to eliminate all of the bugs in a large software system such as a compiler. Over time, the system tends to stabilize around a relatively reliable software base as it is incrementally debugged. The price of this stability is that people become extremely reluctant to change the software, either to add features or even to fix relatively minor bugs, for fear of inadvertantly introducing new bugs. At some point the system becomes obsolete because the developers are unable to upgrade it quickly enough for it to stay relevant.

Credible compilation, combined with the standard organization of the compiler as a sequence of passes, promises to make it possible to continually introduce new, unreliable code into a mature compiler without compromising functionality or reliability. Consider the following scenario. Working under deadline pressure, a compiler developer has come up a prototype implementation of a complex transformation. This transformation is of great interest because it dramatically improves the performance of several SPEC benchmarks. But because the developer cut corners to get the implementation out quickly, it is unreliable. With credible compilation, this unreliability is not a problem at all — the transformation is introduced into the production compiler as another pass, with the compiler driver checking the correctness proof and discarding the results if it didn't work. The compiler operates as reliably as it did before the introduction of the new pass, but when the pass works, it generates much better code.

It is well known that the effort required to make a compiler work on all conceivable inputs is much greater than the effort required to make the compiler work on all likely inputs. Credible compilation makes it possible to build the entire compiler as a sequence of passes that work only for common or important cases. Because developers would be under no pressure to make passes work on all cases, each pass could be hacked together quickly with little testing and no complicated code to handle exceptional cases. The result is that the compiler would be much easier and cheaper to build and much easier to target for good performance on specific programs.

A final extrapolation is to build speculative transformations. The idea is that the compiler simply omits the analysis required to determine if the transformation is legal. It does the transformation anyway and generates a proof that the transformation is correct. This proof is valid, of course, only if the transformation is correct. The proof checker filters out invalid transformations and keeps the rest.

This approach shifts work from the developer to the proof checker. The proof checker does the analysis required to determine if the transformation is legal, and the developer can focus on the transformation and the proof generation, not on writing the analysis code.

## 8.3 Just-In-Time Compilers

The increased network interconnectivity resulting from the deployment of the Internet has enabled and promoted a new way to distribute software. Instead of compiling to native machine code that will run only on one machine, the source program is compiled to a portable byte code. An interpreter executes the byte code.

The problem is that the interpreted byte code runs much slower than native code. The proposed solution is to use a just-in-time compiler to generate native code either when the byte code arrives or dynamically as it runs. Dynamic compilation also has the advantage that it can use dynamically collected profiling information to drive the compilation process.

Note, however, that the just-in-time compiler is another complex, potentially erroneous software component that can affect the correct execution of the program. If a compiler generates native code, the only subsystems that can change the semantics of the native code binary during normal operation are the loader, dynamic linker, operating system and hardware, all of which are relatively static systems. An organization that is shipping software can generate a binary and test it extensively on the kind of systems that its customers will use. If the customer finds an error, the organization can investigate the problem by running the program on a roughly equivalent system.

But with dynamic compilation, the compiled code constantly changes in a way that may be very difficult to reproduce. If the dynamic compiler incorrectly compiles the program, it may be extremely difficult to reproduce the conditions that caused it to fail. This additional complexity in the compilation approach makes it more difficult to build a reliable compiler. It also makes it difficult to assign blame for any failure. When an error shows up, it could be either the compiler or the application. The organizations that built each product tend to blame each other for the error, and neither one is motivated to work hard to find and fix the problem. The end result is that the total system stays broken.

Credible compilation can eliminate this problem. If the dynamic compiler emits a proof that it executed correctly, the run-time system can check the proof before accepting the generated code. All incorrect code would be filtered out before it caused a problem. This approach restores the reliability properties of distributing native code binaries while supporting the convenience and flexibility of dynamic compilation and the distribution of software in portable byte-code format.

## 8.4 An Open Compiler

We believe that credible compilers will change the social context in which compilers are built. Before a developer can safely integrate a pass into the compiler, there must be some evidence that pass will work. But there is currently no way to verify the correctness of the pass. Developers are therefore typically reduced to relying on the reputation of the person that produced the pass, 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 pass, integrate into their compiler, and use it. If a pass 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.

The keys to making this vision a reality are a standard intermediate representation, logics for expressing the proofs, and a verifier that checks the proofs. The representation must be expressive and support the range of program representations required for both high level and low level analyses and transformations. Ideally, the representation would be extensible, with developers able to augment the system with new constructs and new axioms that characterize these constructs. The verifier would be a standard piece of software. We expect several independent verifiers to emerge that would be used by most programmers; paranoid programmers can build their own verifier. It might even be possible to do a formal correctness proof of the verifier.

Once this standard infrastructure is in place, we can leverage the Internet to create a compiler development community. One could imagine, for example, a compiler development web portal with code transformation passes, front ends, and verifiers. Anyone can download a transformation; anyone can use any of the transformations without fear of obtaining an incorrect result. Each developer can construct his or her own custom compiler by stringing together a sequence of optimization passes from this web site. One could even imagine an intellectual property market emerging, as developers license passes or charge electronic cash for each use of a pass. In fact, future compilers may consist of a set of transformations distributed across multiple web sites, with the program (and its correctness proofs) flowing through the sites as it is optimized.

## 8.5 Custom Compilers

Compilers are traditionally thought of and built as general-purpose systems that should be able to compile any program given to them. As a consequence, they tend to contain analyses and transformations that are of general utility and almost always applicable. Any extra components would slow the compiler down and increase the complexity.

The problem with this situation is that general techniques tend to do relatively pedestrian things to the program. For specific classes of programs, more specialized analyses and transformations would make a huge difference [9, 8, 1]. But because they are not generally useful, they don't make it into widely used compilers.

We believe that credible compilation can make it possible to develop lots of different custom compilers that have been specialized for specific classes of applications. The idea is to make a set of credible passes available, then allow the compiler builder to combine them in arbitrary ways. Very specialized passes could be included without threatening the stability of the compiler. One could easily imagine a range of compilers quickly developed for each class of applications.

It would even be possible extrapolate this idea to include optimistic transformations. In some cases, it is difficult to do the analysis required to perform a specific transformation. In this case, the compiler could simply omit the analysis, do the transformation, and generate a proof that would be correct if the analysis would have succeeded. If the transformation is incorrect, it will be filtered out by the compiler driver. Otherwise, the transformation goes through.

This example of optimistic transformations illustrates a somewhat paradoxical property of credible compilation. Even though credible compilation will make it much easier to develop correct compilers, it also makes it practical to release much buggier compilers. In fact, as described below, it may change the reliability expectations for compilers.

Programmers currently expect that the compiler will work correctly for every program that they give it. And you can see that something very close to this level of reliability is required if the compiler fails silently when it fails — it is very difficult for programmers to build a system if there is a reasonable probability that a given error can be caused by the compiler and not the application.

But credible compilation completely changes the situation. If the programmer can determine whether or not the the compiler operated correctly before testing the program, the development process can tolerate a compiler that occasionally fails.

In this scenario, the task of the compiler developer changes completely. He or she is no longer responsible for delivering a program that works almost all of the time. It is enough to deliver a system whose failures do not significantly hamper the development of the system. There is little need to make very uncommon cases work correctly, especially if there are known work-arounds. The result is that compiler developers can be much more aggressive — the length of the develoment cycle will shrink and new techniques will be incorporated into production compilers much more quickly.

# 9 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 optmization: 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.

This paper marks the beginning of the research. Our future plans include integrating techniques for handling pointers, dynamic memory allocation, and dynamic method dispatch into the framework. We also intend to implement a credible compiler. This implementation will provide valuable insight into the level of performance achievable with a credible compiler and the size of the correctness proofs.

In a broader context, humans evolved in small groups characterized by deep, lifelong personal relationships based on mutual familiarity and trust. But the major changes in the organization of human society — agriculture, urbanization, the industrial revolution, and telecommunications — have all changed the human experience towards ever more ephemeral, anonymous interactions with larger groups of people. A global computer network and the concommitant rise of a society organized primarily around information will accelerate this trend with a vengeance. As people interact increasingly with and through networked computers instead of other people, we need a replacement for the trust that comes with personal relationships. One possible replacement, at least for relationships based primarily on information manipulation, is to augment information with evidence that it is in some sense correct. This approach decouples the trustworthiness of the information from its source, eliminating the need to trust the entities with whom one interacts. Credible compilers are one concrete example of this principle.

# References


[1] S. Amarasinghe, J. Anderson, M. Lam, and C. Tseng. The SUIF compiler for scalable parallel machines. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

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

[6] 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.

[7] 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.

[8] M. Rinard and P. Diniz. Commutativity analysis: A new framework for parallelizing compilers. In *Proceedings of the SIGPLAN '96 Conference on Program Language Design and Implementation*, pages 54–67, Philadelphia, PA, May 1996. ACM, New York.

[9] R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, May 1999.