A Framework for Formal Verification of Compiler Optimizations

William Mansky and Elsa Gunter

Department of Computer Science, University of Illinois at Urbana-Champaign,
Thomas M. Siebel Center, 201 N. Goodwin, Urbana, IL 61801-2302
{mansky1,egunter}@cs.illinois.edu

Abstract. In this article, we describe a framework for formally verifying the correctness of compiler optimizations. We begin by giving formal semantics to a variation of the TRANS language [6], which is designed to express optimizations as transformations on control-flow graphs using temporal logic side conditions. We then formalize the idea of correctness of a TRANS optimization, and prove general lemmas about correctness that can form the basis of a proof of correctness for a particular optimization. We present an implementation of the framework in Isabelle, and as a proof of concept, demonstrate a proof of correctness of an algorithm for converting programs into static single assignment form.

Key words: optimizing compilers, theorem proving, program transformations, temporal logic

1 Introduction

Optimizations for time and memory efficiency are now an essential feature of almost all modern compilers. These optimizations are often complex program transformations, and establishing their correctness is a difficult process. In this paper, we propose a general framework for expressing and verifying compiler optimizations using established theorem-proving tools, with the goal of reducing the burden of proving correct any particular optimization.

The problem of verifying a compiler optimization can be divided into two main parts. The first is *specification* of the optimization by giving its semantics in some mathematical formalism. For this purpose, we use the TRANS language, proposed by Kalvala et al. [6], in which optimizations are defined as transformations on control flow graphs conditioned on the satisfaction of temporal logic formulae. This approach is particularly amenable to proofs of correctness for several reasons. First, it allows a more modular formulation of many optimizations than the traditional algorithms, reducing the amount of context that needs to be drawn in to the proof of each step of optimization. Second, it provides a uniform framework for expressing various types of optimizations; by proving certain facts about the fundamental operations of the language, we can take advantage of the redundancy among different optimizations. Finally, the use of temporal logic side conditions makes the assumptions of the transformation explicit in a formal

sense, and can help narrow the gap between the theoretical semantics of the optimization and its actual implementation (e.g., by using model checking to verify the condition before the transformation is performed). As part of the framework, we provide an Isabelle implementation of TRANS, and an instantiation capable of expressing the static single assignment (SSA) transformation.

The second part of the problem, and often the more intractable one, is *verification* of the optimization. Given a definition of correctness, one must construct a formal proof that the correctness property holds for the optimization (see for instance Leroy [8]). The complexity of this process can vary significantly depending on the choice of formalisms in previous steps. Optimizations are most commonly specified as algorithms operating on program code, an approach that is easy to implement but relatively difficult to verify. TRANS itself does not provide a verification procedure, but it does break optimizations down into combinations of simple graph transformations, offering a cleaner, more modular approach. By identifying the basic operations involved in common optimizations, and proving useful lemmas about their correctness, we hope to reduce the amount of effort involved in the proof of correctness of any particular optimization.

To demonstrate the versatility of the framework, we have defined and verified an algorithm for transforming programs into SSA form, a common precursor to compiler optimizations [3, 1]. The SSA transformation is particularly interesting because it extends the underlying language of a program, adding ϕ-functions that are used to determine which instance of a variable should be used based on the program's execution trace. While the transformation cannot be expressed in TRANS as originally presented, we offer a parametric view of TRANS which allows us to easily add transformation-specific constructs such as indexed variables and ϕ-functions. We then give a formal proof of correctness for the SSA conversion using the Isabelle/HOL theorem prover, using the lemmas provided by the framework. The result is a specification of the SSA transformation that is guaranteed to preserve the semantics of the original program.

2 Language Framework

TRANS is a language for expressing program transformations in terms of conditional rewrites on control-flow graphs. As such, in order to give its formal semantics, we must first define the language of programs and formalize the notion of control-flow graphs for those programs. We begin with the simple language L_0, which captures some of the basic functionality of intermediate representations. An L_0 program is a list of instructions, and the number provided to a `goto` command gives the target instruction as a position in the list. More formally:

$$\begin{aligned} \mathit{expr} ::= & \mathit{expr}\ \mathit{op}\ \mathit{expr} \mid \mathit{num} \mid \mathit{var} \\ \mathit{instr} ::= & \mathit{var} := \mathit{expr} \mid \texttt{ret}(\mathit{expr}) \mid \texttt{if}\ \mathit{expr}\ \texttt{goto}\ \mathit{num} \mid \texttt{skip} \mid \texttt{goto}\ \mathit{num} \\ \mathit{graph\_instr} ::= & \mathit{var} := \mathit{expr} \mid \texttt{ret}(\mathit{expr}) \mid \texttt{if}\ \mathit{expr} \mid \texttt{goto} \end{aligned}$$

The semantics of L_0 have been formalized by Lacey et al. [7].

Since TRANS operates on control-flow graphs (CFGs), rather than directly on programs, we must also formally define CFGs. Our definition is adapted from

that given by Kalvala [6], in which the nodes are labeled with single instructions rather than larger basic blocks. A CFG is a record $\mathcal{G} = (N, E \subseteq N \times N \times \{seq, branch\}, I : N \rightharpoonup graph\_instr^1, S : N \rightharpoonup N)$, where $A \rightharpoonup B$ indicates a partial function from A to B, satisfying the following properties:

1. the node set $N_\mathcal{G}$ contains the special nodes *Entry* and *Exit*
2. the instruction labeling $I_\mathcal{G}$ is defined on all nodes except *Entry* and *Exit*
3. *Entry* has no incoming edges, and *Exit* has no outgoing edges
4. the outgoing edges of each other node are consistent with the instruction labeling of that node
5. the reflexive transitive closure of the successor relation $S_\mathcal{G}$ is a linear order
6. if two nodes m and n are connected by an edge with label *seq*, then n is the successor in $S_\mathcal{G}$ of m

Of these, property 4 captures the core idea of a CFG, and its formal definition depends on the underlying language of the graph. In the case of L_0, the appropriate edges for each instruction type are as follows:

- The *Entry* node has one outgoing edge, with label *seq*
- a node labeled with := has one outgoing edge, with label *seq*
- a node labeled with `if` has two outgoing edges, one with label *seq* and one with label *branch*
- a node labeled with `goto` has one outgoing edge, with label *branch*
- a node labeled with `ret` has one outgoing edge, with label *seq*, connecting to the *Exit* node

The inclusion of a successor function aids us in transforming CFGs back into programs, so that TRANS transformations can be said to operate on programs as well as graphs.

A useful property in establishing program correctness is *recoverability*. A CFG is recoverable if it has a unique last node: that is, there is only one node that is labeled with `ret`, and that node is the only predecessor of the *Exit* node[2] [6]. Reasoning about the correctness of a transformation can sometimes be simplified by adding the assumption that the graph in question is recoverable.

Since the framework is to be used to prove that certain transformations preserve the semantics of CFGs, it must also include a notion of evaluation of a CFG. We can give L_0 CFGs a small-step execution semantics, based on the semantics of L_0, in a manner similar to Leroy [8]. The configurations of a CFG under execution are either *intermediate configurations* of the form (m, l, t), where m is a memory, l is a node in the graph (a program point), and t is an execution trace[3], or *values* v, indicating that the execution of the graph has terminated.

[1] Rather than retain the program's instruction numbering in the graph, we use edges to indicate the targets of `goto` and `if` statements, which also erases the distinction between `goto` and `skip` nodes; we use `goto` for either sort of instruction.

[2] Note that any program with at least one `ret` instruction can be restructured to satisfy this condition, by replacing returns with jumps to a single return instruction.

[3] While the execution trace t does not affect the outcome of any instruction in L_0, we include it for generality; in particular, we will make use of it in adjusting the framework to handle the SSA transformation.

Then we can define the small-step relation $\rightarrow_{\mathcal{G}}$ for a graph $\mathcal{G}$ as follows, where we use the notation *out_edges* $\mathcal{G}$ l to indicate the set of outgoing edges of l in $\mathcal{G}$, and assume standard evaluation semantics for arithmetic expressions:

$$\frac{\mathit{out\_edges}\ \mathcal{G}\ \mathit{Entry} = \{(\mathit{Entry}, l', \mathit{seq})\}}{(m, \mathit{Entry}, t) \rightarrow_{\mathcal{G}} (m, l', \mathit{Entry}; t)}$$

$$\frac{I_{\mathcal{G}}(l) = x := e \qquad (e, m) \Downarrow v \qquad \mathit{out\_edges}\ \mathcal{G}\ l = \{(l, l', \mathit{seq})\}}{(m, l, t) \rightarrow_{\mathcal{G}} (m[x \leftarrow v], l', l; t)}$$

$$\frac{I_{\mathcal{G}}(l) = \texttt{if}\ e \qquad (e, m) \Downarrow 0 \qquad \mathit{out\_edges}\ \mathcal{G}\ l = \{(l, l'_1, \mathit{seq}), (l, l'_2, \mathit{branch})\}}{(m, l, t) \rightarrow_{\mathcal{G}} (m, l'_1, l; t)}$$

$$\frac{I_{\mathcal{G}}(l) = \texttt{if}\ e \quad (e, m) \Downarrow v \quad v \neq 0 \quad \mathit{out\_edges}\ \mathcal{G}\ l = \{(l, l'_1, \mathit{seq}), (l, l'_2, \mathit{branch})\}}{(m, l, t) \rightarrow_{\mathcal{G}} (m, l'_2, l; t)}$$

$$\frac{I_{\mathcal{G}}(l) = \texttt{goto} \qquad \mathit{out\_edges}\ \mathcal{G}\ l = \{(l, l', \mathit{branch})\}}{(m, l, t) \rightarrow_{\mathcal{G}} (m, l', l; t)}$$

$$\frac{I_{\mathcal{G}}(l) = \texttt{ret}(e) \qquad (e, m) \Downarrow v}{(m, l, t) \rightarrow_{\mathcal{G}} v}$$

We can now define precisely what we mean when we say that two CFGs are semantically equivalent. We define the set of results of a CFG $\mathcal{G}$ starting from a configuration (m, l, t) as the set of values in the transitive closure of the small-step relation:

$$\frac{(m, l, t) \rightarrow_{\mathcal{G}} v}{(m, l, t, v) \in \mathit{result}\ \mathcal{G}}$$

$$\frac{(m, l, t) \rightarrow (m', l', l; t) \qquad (m', l', l; t, v) \in \mathit{result}\ \mathcal{G}}{(m, l, t, v) \in \mathit{result}\ \mathcal{G}}$$

Then two graphs $\mathcal{G}$ and $\mathcal{G}'$ are semantically equivalent if and only if

$$\forall v.(\mathit{empty}, \mathit{Entry}, [], v) \in \mathit{result}\ \mathcal{G} \Leftrightarrow (\mathit{empty}, \mathit{Entry}, [], v) \in \mathit{result}\ \mathcal{G}'$$

That is, starting from the entry point, the empty environment, and the empty trace, the result set of $\mathcal{G}$ is the same as the result set of $\mathcal{G}'$. It is worth noting that this is a *partial correctness* property, which ignores the possibility of non-termination of the optimization.

3 The TRANS Language

Now we have enough background to define the TRANS language itself. We will present here an overview of the syntax and semantics of the language, focusing on the differences between our formulation and its original presentation; for further details, see Kalvala et al. [6].

3.1 Overview

The basic units of TRANS are conditional graph rewrites of the form $A_1, A_2, ..., A_n$ if ϕ, where the A_i's are *actions* to be performed on a graph, and ϕ is a CTL-based *side condition*. Both the action and the condition may contain *metavariables*, which are instantiated with program objects when the transformation is applied. Actions are defined by

$$A ::= \mathsf{add\_edge}(n, m, t) \mid \mathsf{remove\_edge}(n, m, t) \mid \mathsf{replace}\ n\ \mathsf{with}\ p_1, ..., p_m \mid \mathsf{split\_edge}(n, m, t, p)$$

The action $\mathsf{add\_edge}(n, m, t)$ adds an edge labeled t between n and m; $\mathsf{remove\_edge}(n, m, t)$ removes an edge with label t between n and m; $\mathsf{replace}\ n\ \mathsf{with}\ p_1, ..., p_m$, replaces the instruction at n with the instructions $p_1, ..., p_m$; and $\mathsf{split\_edge}(n, m, t, p)$, inserts a node with instruction p in the middle of the edge from n to m labeled t. Many common program transformations can be expressed as a combination of these basic actions; in fact, many, including conversion to SSA form, can be expressed using only the replace action. Rather than using actual (graph) instructions, these actions take as arguments *instruction patterns*, in which metavariables can be used in place of program objects.

The side condition of a rewrite is an expression in CTL over paths in the graph, with the atomic predicates $node(n)$, which asserts that the metavariable n matches the current node, and $stmt(p)$, which asserts that the instruction pattern p matches the instruction label at the current node. These atomic predicates can be combined with the usual propositional connectives, as well as several CTL-specific operators [4], which fall into one of two categories. The first contains the *next-time* operators EX and AX, which assert that a statement holds on some successor and all successors of the current node respectively. The second group includes the *until* operators $E\phi_1 U \phi_2$ and $A\phi_1 U \phi_2$, which assert that ϕ_1 holds on all nodes until a node is reached on which ϕ_2 holds, along some path and all paths forward from the current node respectively. TRANS also uses reversed versions of the temporal operators, e.g. $\overleftarrow{EX}$, which make analogous assertions on nodes/paths backward from the current node. The formula $\phi@n$ asserts that ϕ holds starting from the node n. In addition to the CTL formulae on paths, TRANS side conditions include various basic predicates, such as $freevar(x, e)$, which asserts that x is a free variable of the expression e, or $fresh(x)$, which asserts that x does not appear in any instruction in the graph under consideration.

In order to determine the concrete transformation represented by a rewrite, a *valuation*, or map from metavariables to program objects, must be provided. This valuation, usually denoted by σ, is applied to both the action, to determine the actual nodes and instructions to be rewritten, and the side condition, to ensure that it produces a valid instance of the rewrite. A concrete instruction is obtained from a pattern p by applying the function $subst(\sigma, p)$, which replaces all the metavariables in p with the values given to them by σ.

A TRANS expression, then, is a group of conditional rewrites combined with any number of *strategies*, defined by

$$T ::= A_1, ..., A_m \text{if} \phi \mid \text{MATCH } \phi \text{ IN } T \mid T_1 \text{ THEN } T_2 \mid T_1 \square T_2 \mid \text{APPLY\_ALL } T$$

The TRANS expression MATCH ϕ IN T executes T under the restriction that its valuation must satisfy the condition ϕ; T_1 THEN T_2 applies T_1 and T_2 in sequence to a graph; $T_1 \square T_2$ applies either T_1 or T_2; and APPLY_ALL T recursively applies T under any possible valuation until it is no longer applicable.

Two aspects of the TRANS language are provided as parameters. The first parameter is the underlying language used to label the nodes of the CFGs; for instance, TRANS on L_0 CFGs has instruction patterns such as $x := e$ and `if` e. The second parameter is the set of basic predicates and atomic propositions used in the side conditions. Any property of a valuation and/or a CFG can serve as a predicate; any property of a valuation, a CFG, and a node can be used as an atomic proposition. We will modify both of these parameters when applying TRANS to the SSA transformation. Note that even TRANS on L_0 can express a variety of common optimizations; see Kalvala [6] for several examples, including dead code elimination and strength reduction.

3.2 Transformation Semantics

For the most part, the semantics of our definition of TRANS are identical to those given by Kalvala [6], with the addition of a few new basic predicates and local (defined) predicates. However, we take a simpler approach to the semantics of top-level transformations, and offer new definitions for the sequencing strategies THEN and APPLY_ALL which we believe are more consistent with the intended use of the strategies.

In the original presentation of TRANS, the semantics of a transformation were given by a semantic function $\lceil . \rceil : Transformation \to (PartValuation \times FlowGraph \to \mathbb{P}(FlowGraph \to FlowGraph))$ taking a transformation, a partial valuation (a partial function from metavariables to objects), and a graph, and giving a set of functions on graphs. That is, given a partial valuation and a graph, a transformation defined a set of functions to be applied to the graph. However, these functions are all intended to be applied to the graph provided, and are not guaranteed to be safe on any other graph (since the conditions of the transformation are checked on the original graph). Thus, the semantic function for a transformation can equally return the set of graphs resulting from the application of the transformation to the graph provided. We will take this approach in the following definitions, and use a semantic function $\lceil . \rceil : Transformation \to (PartValuation \times FlowGraph \to \mathbb{P}(FlowGraph))$.

Our second modification to the semantics deals with the compositional strategies THEN and APPLY_ALL. Originally, the semantics of THEN were given by

$$\lceil T_1 \text{ THEN } T_2 \rceil(\tau, \mathcal{G}) = \{f \circ g \mid f \in \lceil T_1 \rceil(\tau, \mathcal{G}) \wedge g \in \lceil T_2 \rceil(\tau, \mathcal{G})\}$$

Intuitively, both T_1 and T_2 are evaluated on the graph $\mathcal{G}$, and then the resulting functions are composed. This has the disadvantage that it violates one of the desired properties of TRANS, namely, that a transformation is only applied when its side condition is satisfied. Since T_2 is evaluated on $\mathcal{G}$, and then applied to $f(\mathcal{G})$ for some $f \in \lceil T_1 \rceil(\tau, \mathcal{G})$, the transformation it defines may be applied to

a graph on which its side condition does not hold. Using our modified semantic function, we propose the alternate definition

$$\lceil T_1 \text{ THEN } T_2 \rceil(\tau, \mathcal{G}) = \{\mathcal{G}'' \mid \exists \mathcal{G}'.\mathcal{G}' \in \lceil T_1 \rceil(\tau, \mathcal{G}) \wedge \mathcal{G}'' \in \lceil T_2 \rceil(\tau, \mathcal{G}')\}$$

Under this definition, T_2 is evaluated not on the original graph $\mathcal{G}$, but rather on the graph $\mathcal{G}'$ to which it will be applied, restoring the link between the condition and the transformation.

Our treatment of the APPLY_ALL strategy is similar, but more complex. The intention of the strategy is to apply a transformation everywhere in the graph, i.e., until the transformation can no longer be applied. The original semantics given for APPLY_ALL were

$$\lceil \text{APPLY\_ALL } T \rceil(\tau, \mathcal{G}) = \{f_1 \circ f_2 \circ ... \circ f_n \mid f_i \in \lceil T \rceil(\tau, \mathcal{G}) \setminus \{f_1, ..., f_{i-1}\}\}$$

This definition again allows the application of a transformation to a graph on which the side condition has not been checked, and also suffers from the problem that $\lceil T \rceil(\tau, \mathcal{G})$ may not be finite (or even countable). As a matter of fact, since the graph under consideration changes after each application of T, and the condition must be re-evaluated, the APPLY_ALL construct is essentially recursive. Thus, in order to give it the desired semantics, we must give it an inductive definition. We begin by defining an inductive set *apply_some* containing all graphs produced from applying T some number of times (possibly zero) to $\mathcal{G}$:

$$\mathcal{G} \in \mathit{apply\_some}(T, \tau, \mathcal{G})$$

$$\frac{\mathcal{G}' \in \lceil T \rceil(\tau, \mathcal{G}) \qquad \mathcal{G}'' \in \mathit{apply\_some}(T, \tau, \mathcal{G}')}{\mathcal{G}'' \in \mathit{apply\_some}(T, \tau, \mathcal{G})}$$

Then, by removing all of the intermediate graphs, we can define APPLY_ALL as yielding exactly the set of graphs in which T has been performed until it can no longer be applied:

$$\lceil \text{APPLY\_ALL } T \rceil(\tau, \mathcal{G}) = \mathit{apply\_some}(T, \tau, \mathcal{G}) \setminus \{\mathcal{G}' \mid \exists \mathcal{G}''.\mathcal{G}' \neq \mathcal{G}'' \wedge \mathcal{G}'' \in \lceil T \rceil(\tau, \mathcal{G}')\}$$

Under this definition, we once again have the property that a transformation is never applied to a graph on which its condition has not been evaluated, and we also know that T will no longer be applicable to the resulting graphs[4]. As part of the implementation of the framework, we have formalized the syntax and modified semantics of TRANS in Isabelle; the algebraic definitions can be translated directly into Isabelle datatypes and functions.

4 Theoretical Properties of TRANS

Although the TRANS approach has not previously been used to verify optimizations, the modularity provided by strategies and the use of CTL side conditions

[4] Note that if the recursive application of T is non-terminating, the set defined by APPLY_ALL T is empty.

make it well suited for formal verification. In this section, we state and sketch the proof of several simple but powerful properties of TRANS expressions, dealing with the effects of strategies and some common transformations. These lemmas also suggest a general methodology for verifying optimizations using TRANS: the first lemma can be used to break an optimization down into component transformations, and the following lemmas provide useful facts about common basic transformations.

When verifying a transformation, we frequently want to prove that some property of the program under consideration is *preserved*; that is, if it holds for the original program, then it also holds for the transformed program. Formally, we say that a transformation T preserves a property P of graphs if for all partial valuations τ and graphs $\mathcal{G}$, if P holds on $\mathcal{G}$, then for all result graphs $\mathcal{G}' \in \lceil T \rceil(\tau, \mathcal{G})$, P holds on $\mathcal{G}'$. We expect that if a group of transformations preserves some property, then any combination of those transformations also preserves the property. In fact, we can prove that the various strategies preserve any property that their component transformations preserve:

Lemma 1. *Suppose that T and T' preserve some property P. Then* MATCH ϕ IN T, T THEN T', $T \square T'$, *and* APPLY_ALL T *preserve P.*

Proof. For every strategy other than APPLY_ALL, the result follows directly from the semantics of the strategy. We can show that APPLY_ALL T preserves P by induction on the number of applications of T. □

This result allows us to break down the problem of showing that a complex transformation preserves a property into one of showing that each of its individual components (sub-expressions of the form $A_1, ..., A_m$ if ϕ) preserves the property. In the case study below we will use this lemma to show that a transformation preserves recoverability, but it could be used for any invariant on a graph.

The simplest transformation is one of the form replace n with p, where p is a single instruction pattern. This is the TRANS method of modifying a single instruction in the graph. Transformations of this form have many useful properties: for instance, they do not change the nodes, edges, or successor relation of a graph. In addition, we can give a simple condition under which such transformations preserve recoverability.

Lemma 2. *Consider a recoverable graph $\mathcal{G} = (N, E, I, S)$, and a valuation σ. If $I_{\mathcal{G}}\ \sigma(n)$ is the same type of instruction*[5] *as p, then $\mathcal{G}' = \lceil$replace n with $p\rceil(\sigma, \mathcal{G})$ is recoverable.*

Proof. The only effect of the single-instruction replace is to replace the instruction at $\sigma(n)$ with $subst(\sigma, p)$. Since the two instructions have the same type, this replacement does not affect the recoverability of the graph. □

This is a powerful lemma because it can be proved at the level of actions, and thus applies to any transformation using an action of this form. In other words, any

[5] I.e., they are both `if`-statements, or both `goto`-statements, etc.

transformation which performs only replacements of instructions by instructions of the same type is guaranteed to preserve recoverability. As we will see in our case study, some surprisingly complex transformations fall into this category.

A slightly more complicated transformation is one of the form replace n with $p_1, ..., p_m$, which replaces an instruction with a list of instructions. In this case, the nodes, edges, and successor function are all affected, but in a strictly local way. Once again, we can identify a common case in which such transformations preserve recoverability: that of inserting a list of assignments before some point in the program.

Lemma 3. *Consider a recoverable graph $\mathcal{G} = (N, E, I, S)$, and a valuation σ. If $\sigma(i) = I_{\mathcal{G}}\ \sigma(n)$ and $p_1, ..., p_m$ are all assignment patterns, then $\mathcal{G}' = \lceil$replace n with $p_1, ..., p_m, i\rceil(\sigma, \mathcal{G})$ is recoverable.*

Proof. The effect of this replace action is to insert a sequence of nodes before $\sigma(n)$. Since each inserted node is connected to its successor with a *seq*-edge, which is consistent with any assignment instruction, $\mathcal{G}'$ is still a CFG, and since no `ret` instructions are added or removed, $\mathcal{G}'$ is still recoverable. □

Note that the p_i's must be assignments because the edges inserted between the new nodes are labeled with *seq*. In fact, this lemma can be generalized to any instruction type for which the appropriate outgoing-edge set is a single *seq*-edge.

Lemma 4. *Consider a recoverable graph $\mathcal{G} = (N, E, I, S)$, and a valuation σ. If $\sigma(i) = I\ \mathcal{G}\ \sigma(n)$ and $p_1, ..., p_m$ are patterns consistent with an outgoing-edge set of a single seq-edge, then $\mathcal{G}' = \lceil$replace n with $p_1, ..., p_m, i\rceil(\sigma, \mathcal{G})$ is recoverable.*

In the case of L_0, these two statements are equivalent, but in more comprehensive languages, such as the L_1 language presented below, the more general statement of the lemma will be more useful.

5 The SSA Transformation

5.1 Static Single Assignment Form

One common program transformation in optimizing compilers is conversion to SSA form. While not in itself an optimization, this conversion allows for the application of various other optimizations and analyses [3, 1]. A program in SSA form, as its name suggests, has no more than one assignment statement for each variable in the program. This is accomplished by labeling each instance of each variable with a subscript, or *index*; a unique index is given to each definition of a variable, and each use is given the index of the definition that reaches it. At join-points in the program, where two or more branches converge, there may be more than one reaching definition of a given variable; in this case, a *ϕ-function* is inserted at that point. The ϕ-function takes as arguments all the indexed instances of the variable that could reach the current point, chooses the appropriate instance based on the trace of the current execution, and assigns it

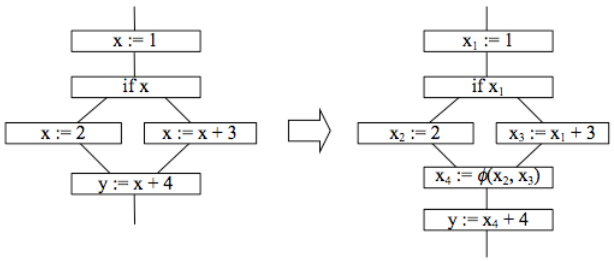

Fig. 1. Converting a CFG to SSA form

to a new instance of the variable. In this way, for every use of a variable in any instruction other than a ϕ-function, there is exactly one reaching definition, and so every variable can be labeled unambiguously with an index.

From a verification perspective, the SSA transformation is particularly interesting because 1) it introduces new variables into the program, and 2) it extends the language of the program by adding ϕ-functions. As such, we decided to test the power of the framework by using it to state and prove the correctness of SSA.

5.2 Defining SSA in TRANS

TRANS on L_0 is not sufficient to express the SSA transformation. However, we can extend L_0 to a new language L_1 that can support SSA by adding two new constructs: an indexed variable var_{num}, which can be used anywhere a variable can be used, and a ϕ-instruction $var := \phi(var, ..., var)$. The appropriate outgoing edge set for a ϕ-instruction is a single node with label *seq*, as in the case of an ordinary L_0 assignment instruction. The small-step evaluation rules for the ϕ-instruction are as follows, where *find_in_trace* $v_1, ..., v_n$ t $\mathcal{G}$ gives the most recent definition of any of the x_i's in the trace t:

$$\frac{I_{\mathcal{G}}(l) = x := \phi(v_1, ..., v_n) \qquad \mathit{find\_in\_trace}\ v_1, ..., v_n\ t\ \mathcal{G} = v' \qquad \mathit{out\_edges}\ \mathcal{G}\ l = \{(l, l', \mathit{seq})\}}{(m, l, t) \rightarrow_{\mathcal{G}} (m[x \leftarrow m(v')], l', l; t)}$$

$$\frac{I_{\mathcal{G}}(l) = x := \phi(v_1, ..., v_n) \qquad \forall v'. \mathit{find\_in\_trace}\ v_1, ..., v_n\ t\ \mathcal{G} \neq v' \qquad \mathit{out\_edges}\ \mathcal{G}\ l = \{(l, l', \mathit{seq})\}}{(m, l, t) \rightarrow_{\mathcal{G}} (m, l', l; t)}$$

Note that we now make use of the execution trace included in the configuration. We include a rule to handle the corner case in which none of the variables in the ϕ-function are defined; in this case, the ϕ-function has no effect on the memory. We chose this outcome, rather than allowing the execution to become stuck, to most closely mimic the behavior of the original graph: if a variable was undefined, the program would not become stuck until it reached a use of the variable, and

inserting a ϕ-function should not force a crash in a program that would not originally have crashed.

We also add several SSA-specific basic predicates to TRANS, such as *fresh_new*(x, j). The predicate *fresh_new*(x, j) holds when x is a variable, j is a number, and the indexed variable x_j does not appear anywhere in the program. We can use CTL on CFGs to define useful propositions on nodes such as *reaches*(x, j), which holds when x_j reaches the current node, and *multi_defs*(x), which holds when two different instances of x reach the current node.

We can now express the SSA conversion in TRANS on L_1. We break the conversion down into four stages, as follows: *add_index* adds indices to the left-hand side of each assignment in the program. *add_phi* inserts ϕ-functions (with empty bodies) at every point reached by multiple definitions[6]. *update* updates each use of a variable with the index of the reaching instance, and fills in the bodies of ϕ-functions. Finally, *refactor* transforms each indexed variable (which is semantically equivalent to the variable obtained by dropping the index) to an entirely new variable, giving it a separate location in memory. The conversion can be written in TRANS as follows:

$$\begin{aligned} \mathit{add\_index} ::= \ & \mathsf{replace}\ n\ \mathsf{with}\ (x_k := e) \\ & \mathsf{if}\ (\mathit{varlit}(x)\ \wedge\ \mathit{stmt}(x := e)\ @\ n\ \wedge\ \mathit{fresh\_new}(x, k)) \end{aligned}$$

The first step of the conversion is the simplest: each definition of a variable x is assigned a unique index k, without making any other changes to the graph.

$$\begin{aligned} & \mathit{add\_phi} ::= \\ & \ \mathsf{replace}\ n\ \mathsf{with}\ (x_k := \phi(), i) \\ & \ \mathsf{if}\ (\mathit{stmt}(i)\ @\ n\ \wedge\ \mathit{multi\_defs}\ x\ j_1\ j_2\ @\ n\ \wedge\ \mathit{fresh\_new}\ x\ k\ \wedge \\ & \quad \exists n_1\ (\exists n_2\ ((\neg n_1\ \mathit{is}\ n_2)\ \wedge\ (\overleftarrow{EX}\ \mathit{node}(n_1)\ \wedge\ \overleftarrow{EX}\ \mathit{node}(n_2))\ @\ n)) \wedge \\ & \quad A(\exists y, s\ (\mathit{stmt}(y := \phi(s))\ \wedge\ \neg x\ \mathit{is}\ y))U(\neg\exists y, s\ (\mathit{stmt}(y := \phi(s))))\ @\ n) \end{aligned}$$

In the second step, we insert the ϕ-functions at the proper locations in the program. Whereas in traditional algorithmic descriptions of the transformation, an analysis must be performed to determine where to place ϕ-functions, the TRANS approach allows us to explicitly state the condition under which a function should be inserted. The second half of the condition ensures that the ϕ-functions are only inserted at join points, and that no more than one ϕ-function per variable is inserted at each join point.

$$\begin{aligned} \mathit{update} ::= \ & \mathrm{MATCH}\ (\mathit{reaches}\ x\ k\ @\ n)\ \mathrm{IN} \\ & (\mathsf{replace}\ n\ \mathsf{with}\ (y := e[x_k])\ \mathsf{if}\ \mathit{stmt}(y := e[x])\ @\ n\ \square \\ & \ \mathsf{replace}\ n\ \mathsf{with}\ (\mathtt{if}\ e[x_k])\ \mathsf{if}\ \mathit{stmt}(\mathtt{if}\ e[x])\ @\ n\ \square \\ & \ \mathsf{replace}\ n\ \mathsf{with}\ (\mathtt{ret}(e[x_k]))\ \mathsf{if}\ \mathit{stmt}(\mathtt{ret}(e[x]))\ @\ n\ \square \\ & \ \mathsf{replace}\ n\ \mathsf{with}\ (x_{k'} := \phi(x_k, s))\ \mathsf{if}\ (\mathit{stmt}(x_{k'} := \phi(s))\ @\ n \wedge x_k \notin s)) \end{aligned}$$

In the third step, each use of a variable is annotated with the index of the reaching definition. Each ϕ-function is also populated with the reaching instances

[6] Note that this formulation of the conversion may not compute a minimal number of ϕ-functions; this could be achieved by adding an additional condition to *add_phi*.

of the variable to which it is assigned.

refactor ::=
MATCH ($fresh(z) \wedge (stmt(x_k := e) \vee stmt(x_k := \phi(s))) \ @ \ n$) IN
(($\mathsf{replace}\ n\ \mathsf{with}\ (z := e)\ \mathsf{if}\ stmt(x_k := e) \ @ \ n \ \square$
$\mathsf{replace}\ n\ \mathsf{with}\ (z := \phi(s))\ \mathsf{if}\ stmt(x_k := \phi(s)) \ @ \ n$) THEN
APPLY_ALL ($\mathsf{replace}\ m\ \mathsf{with}(y := f[z])\ \mathsf{if}\ stmt(y := f[x_k]) \ @ \ m \ \square$
$\mathsf{replace}\ m\ \mathsf{with}\ (\mathtt{if}\ f[z])\ \mathsf{if} stmt(\mathtt{if}\ f[x_k]) \ @ \ m \ \square$
$\mathsf{replace}\ m\ \mathsf{with}\ (\mathtt{ret}(f[z]))\ \mathsf{if}\ stmt(\mathtt{ret}(f[x_k])) \ @ \ m \ \square$
$\mathsf{replace}\ m\ \mathsf{with}\ (y := \phi(z, t))\ \mathsf{if}\ stmt(y := \phi(x_k, t)) \ @ \ m$))

The final step is to replace each indexed variable with an entirely new variable. While this is not explicitly a part of most SSA algorithms, it is necessary because of the memory model used in our implementation, in which every instance of a variable points to the same memory location. The advantage of this approach is that each individual step can be shown to preserve the program's semantics, allowing for a more modular proof of correctness, as will be shown below.

Each step performs the desired transformation for one variable, so we use the APPLY_ALL strategy to extend it to the entire program, giving the complete SSA transformation:

conversion ::= (APPLY_ALL *add_index*) THEN (APPLY_ALL *add_phi*) THEN
(APPLY_ALL *update*) THEN (APPLY_ALL *refactor*)

5.3 Proving SSA Correct

Given the definition of SSA in TRANS, and the lemmas shown above, we are now ready to construct a proof of correctness for SSA. In fact, using the Isabelle implementation of the framework, we have done exactly that, proving that any graph produced by the conversion is semantically equivalent to the input graph. We will outline the proof below. Since we have taken the trouble to express the conversion in as modular a fashion as possible, we can verify it by proving appropriate theorems for each of the individual steps, and then combining them with the simple facts about the strategies shown above.

Thanks to the power of the framework, we can easily show that each step of the transformation preserves recoverability.

Lemma 5. *add_index, add_phi, update, and refactor all preserve recoverability.*

Proof. All of the actions in *add_index*, *update*, and *refactor* are of the form specified in Lemma 2, so they preserve recoverability. The action in *add_phi* is of the form specified in Lemma 4, so it also preserves recoverability. Thus, we can conclude that the conversion preserves recoverability. □

Corollary 1. *The complete SSA conversion preserves recoverability.*

Proof. Since recoverability is a property of a graph, by Lemma 1 the conversion preserves recoverability if each of its steps preserves recoverability. □

Now we can demonstrate the correctness of the transformation. Once again, we will proceed modularly, stating an appropriate lemma for each step of the conversion. Of course, the steps are not independent of each other; they are semantics-preserving only in combination. However, we can separate them by stating appropriate conditions that must hold for each step to be correct, and then showing that these conditions hold after the previous step. Ultimately, we will have shown not only that the results of the transformation are semantically equivalent to its input, but also that the resulting graphs are in SSA form. We can identify three properties that should hold at various points in the transformation:

1. The graph has no more than one definition for each variable instance it contains.
2. There is no more than one instance of each variable in the graph that reaches each node labeled with a non-ϕ instruction, and each instruction uses a variable instance x_j only if x_j reaches the node labeled with the instruction.
3. If an instance of a variable x_j reaches a node in the graph labeled with a ϕ-function, and another instance x_k is in the body of the ϕ-function, then x_j is also in the body of the ϕ-function.

Note that if a graph has no indexed variables, then property 1 is sufficient to imply that the graph is in SSA form, since there is at most one instance of each variable.

Lemma 6. *Consider a CFG $\mathcal{G}$ with no ϕ-functions. Then every graph in $\lceil add\_index \rceil(empty, \mathcal{G})$ is semantically equivalent to $\mathcal{G}$ and has no ϕ-functions. Furthermore, every graph in $\lceil$*APPLY_ALL *$add\_index \rceil(empty, \mathcal{G})$ is semantically equivalent to $\mathcal{G}$, has no ϕ-functions, and has property 1.*

Lemma 7. *Consider a recoverable CFG $\mathcal{G}$ with no non-empty ϕ-functions such that property 1 holds for $\mathcal{G}$. Then any graph in $\lceil add\_phi \rceil(empty, \mathcal{G})$ is semantically equivalent to $\mathcal{G}$, has no non-empty ϕ-functions, and has property 1. Any graph in $\lceil$*APPLY_ALL *$add\_phi \rceil(empty, \mathcal{G})$ also is semantically equivalent to $\mathcal{G}$, has no non-empty ϕ-functions, and has property 1.*

Lemma 8. *Consider a CFG $\mathcal{G}$ with property 1 such that every ϕ-function in $\mathcal{G}$ is of the form $x_j = \phi(x_{k1}, ..., x_{kn})$ – i.e., every variable in the body of the ϕ-function has the same base as the variable on the left-hand side. Then every graph in $\lceil update \rceil(empty, \mathcal{G})$ is semantically equivalent to $\mathcal{G}$, has property 1, and has only ϕ-functions of the form $x_j = \phi(x_{k1}, ..., x_{kn})$. Furthermore, every graph in $\lceil$*APPLY_ALL *$update \rceil(empty, \mathcal{G})$ is semantically equivalent to $\mathcal{G}$, has only ϕ-functions of the form $x_j = \phi(x_{k1}, ..., x_{kn})$, and has properties 1, 2, and 3.*

Lemma 9. *Consider a CFG $\mathcal{G}$ with properties 1, 2, and 3. Then every graph in $\lceil refactor \rceil(empty, \mathcal{G})$ is semantically equivalent to $\mathcal{G}$ and has properties 1, 2, and 3. Furthermore, every graph in $\lceil$*APPLY_ALL *$refactor \rceil(empty, \mathcal{G})$ is semantically equivalent to $\mathcal{G}$, has properties 1, 2, and 3, and has no indexed variables.*

Theorem 1. *Consider a recoverable L_0 CFG $\mathcal{G}$. Every graph in $\lceil conversion \rceil(empty, \mathcal{G})$ is semantically equivalent to $\mathcal{G}$, and is in SSA form.*

Proof. We will proceed by breaking the conversion into steps, and then make use of the above lemmas. Since L_0 does not contain ϕ-functions, we know that $\mathcal{G}$ has no ϕ-functions. Thus, by Lemmas 5 and 6, every graph in $\lceil$APPLY_ALL *add_index*$\rceil(empty, \mathcal{G})$ is recoverable, is semantically equivalent to $\mathcal{G}$, has no ϕ-functions, and has property 1. If a graph has no ϕ-functions, then it certainly has no non-empty ϕ-functions. Thus, by Lemma 7, every graph in $\lceil$(APPLY_ALL *add_index* THEN (APPLY_ALL *add_phi*)$\rceil(empty, \mathcal{G})$ is semantically equivalent to $\mathcal{G}$, has no non-empty ϕ-functions, and has property 1. If every ϕ-function in a graph is empty, then certainly every ϕ-function is of the form $x_j = \phi(x_{k1}, ..., x_{kn})$. Thus, by Lemma 8, every graph in

$$\lceil \text{(APPLY\_ALL } add\_index\text{) THEN}$$
$$\text{(APPLY\_ALL } add\_phi\text{) THEN (APPLY\_ALL } update\text{)} \rceil (empty, \mathcal{G})$$

is semantically equivalent to $\mathcal{G}$, and has properties 1, 2, and 3. Finally, by Lemma 9, we can conclude that every graph in $\lceil conversion \rceil(empty, \mathcal{G})$ is semantically equivalent to $\mathcal{G}$, has properties 1, 2, and 3, and has no indexed variables, implying that it is in SSA form. □

6 Conclusions and Further Research

We have outlined a new approach to proving the correctness of compiler optimizations, as well as an Isabelle-implemented framework to support this approach. We have clarified and formalized the semantics of the TRANS language, making it into a tool for use in constructing fully formal proofs of correctness for program transformations; the Isabelle code can be found online [10]. By using TRANS to state a transformation in terms of conditional rewrites on CFGs, a verifier can take advantage of the general lemmas we have established about the semantics of TRANS. We have demonstrated the generality of this approach by parameterizing TRANS to express the SSA transformation, and then presented a new and relatively concise proof of correctness for the transformation within our framework. This is, to the best of our knowledge, the first machine-assisted verification of an optimization expressed in TRANS, and the first verified SSA conversion of any sort. We hope that the approach demonstrated here will significantly reduce the difficulty of the problem of verifying optimizations.

The L_0 language on which our implementation of TRANS is based is relatively simple, and does not include some common features of intermediate languages, such as arrays and function calls. Our experience with L_1 suggests that parameterizing TRANS with more expressive languages can be easily accomplished; the more difficult aspect of adding new features is providing a semantics for programs with these features and proving related lemmas about TRANS.

This would bring the framework a step closer to dealing with real-world intermediate languages, and enable it to handle a wider range of optimizations.

The tendency to leave compiler optimizations unverified might be defended with the argument that most optimizations have been in use for decades without major problems. However, bugs have been discovered even in C compilers [11], and the situation is even more complicated when dealing with concurrency. Most optimizations for parallel programs are experimental and have not been widely field-tested, and it is much more difficult to put forward a convincing informal justification for a parallel optimization. For this reason, we believe that compilers for parallel languages particularly stand to benefit from formal proofs of correctness. While formalisms such as control-flow graphs and CTL are sufficient for expressing and verifying optimizations on sequential programs, they are not as obviously applicable to the case of parallel programs. However, we believe that concurrent analogues of these formalisms, such as parallel program graphs [14] and alternating-time temporal logic [2], will allow us to extend our approach to parallel optimizations.

7 Related Work

This research builds on the work of Lacey et al. [7] and Kalvala et al. [6], who have defined the core concepts of the TRANS language and used it to express and (on paper) verify several optimizations.

One of the first computer-assisted verifications of a compiler was due to Moore [12]. The source language of this compiler was very low-level, and the compiler did not perform any optimizations.

Leroy [9] has developed a Coq-based framework for the verification of optimizations that depend on dataflow analysis. Code transformations operate on instructions, and are verified by comparing the semantics of the transformed instructions to that of the original instructions under conditions provided by the dataflow analysis. However, this approach can handle only limited modifications to program structure. Visser et al. [16] use a rewrite system to define optimizations on programs in a functional language. Their system includes a variety of rewriting strategies, but does not deal with conditional rewriting or verification.

Translation validation [13, 15] is another method of verifying compiler optimizations. In this approach, rather than proving the optimization correct for all programs, an automatic verifier is used on the results of each application of the optimization to ensure that the resulting program has the same semantics as the original program. Since the process is fully automated, the verification process is considerably more lightweight, but the range of optimizations that can be handled is more limited.

Various work has been done on the formal properties of code already in SSA form; see for instance Blech and Glesner [5], who have used Isabelle to verify an algorithm for code generation from SSA.

Acknowledgments: We would like to thank Sara Kalvala and Richard Warburton for introducing us to TRANS and clarifying various points.

References

1. Alpern, B., Wegman, M.N., Zadeck, F.K.: Detecting equality of variables in programs. In: POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages. pp. 1–11. ACM, New York, NY, USA (1988)
2. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. J. ACM 49(5), 672–713 (2002)
3. Appel, A.W.: Modern Compiler Implementation in ML. Cambridge University Press, New York, NY, USA (2004)
4. Ben-Ari, M., Manna, Z., Pnueli, A.: The temporal logic of branching time. In: POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages. pp. 164–176. ACM, New York, NY, USA (1981)
5. Blech, J.O., Glesner, S.: A formal correctness proof for code generation from ssa form in isabelle/hol. In: Proceedings der 3. Arbeitstagung Programmiersprachen (ATPS) auf der 34. Jahrestagung der Gesellschaft für Informatik. Lecture Notes in Informatics (September 2004), `http://www.info.uni-karlsruhe.de/papers/Blech-Glesner-ATPS-2004.pdf`
6. Kalvala, S., Warburton, R., Lacey, D.: Program transformations using temporal logic side conditions. ACM Trans. Program. Lang. Syst. 31(4), 1–48 (2009)
7. Lacey, D., Jones, N.D., Van Wyk, E., Frederiksen, C.C.: Proving correctness of compiler optimizations by temporal logic. In: POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages. pp. 283–294. ACM, New York, NY, USA (2002)
8. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages. pp. 42–54. ACM, New York, NY, USA (2006)
9. Leroy, X.: A formally verified compiler back-end. J. Autom. Reason. 43(4), 363–446 (2009)
10. Mansky, W.: TRANS in Isabelle. `http://www.cs.illinois.edu/homes/mansky1`
11. McKeeman, W.M.: A formally verified compiler back-end. Digital Technical Journal 10(1), 100–107 (1998)
12. Moore, J.S.: A mechanically verified language implementation. J. Autom. Reason. 5(4), 461–492 (1989)
13. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems. pp. 151–166. Springer-Verlag, London, UK (1998)
14. Sarkar, V.: Analysis and optimization of explicitly parallel programs using the parallel program graph representation. In: LCPC '97: Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing. pp. 94–113. Springer-Verlag, London, UK (1998)
15. Tristan, J.B., Leroy, X.: Formal verification of translation validators: a case study on instruction scheduling optimizations. In: POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages. pp. 17–27. ACM, New York, NY, USA (2008)
16. Visser, E., Benaissa, Z.e.A., Tolmach, A.: Building program optimizers with rewriting strategies. SIGPLAN Not. 34(1), 13–26 (1999)