Proving Correctness of Compiler Optimizations by Temporal Logic *

David Lacey
David.Lacey@comlab.ox.ac.uk
University of Oxford

Neil D. Jones
neil@diku.dk
University of Copenhagen

Eric Van Wyk
Eric.Van.Wyk@comlab.ox.ac.uk
University of Oxford

Carl Christian Frederiksen
xeno@diku.dk
University of Copenhagen

ABSTRACT

Many classical compiler optimizations can be elegantly expressed using rewrite rules of form: $I \Longrightarrow I'$ if ϕ, where I, I' are intermediate language instructions and ϕ is a property expressed in a temporal logic suitable for describing program data flow. Its reading: If the current program π contains an instruction of form I at some control point p, and if flow condition ϕ is satisfied at p, then replace I by I'.

The purpose of this paper is to show how such transformations may be proven correct. Our methodology is illustrated by three familiar optimizations, dead code elimination, constant folding and code motion. The meaning of correctness is that for any program π, if $Rewrite(\pi, \pi', p, I \Longrightarrow I' \text{ if } \phi)$ then $[\![\pi]\!] = [\![\pi']\!]$, i.e. π and π' have exactly the same semantics.

1. INTRODUCTION

This paper shows that temporal logic can be used to validate some classical compiler optimizations in a very strong sense.

First, typical optimizing transformations are shown to be simply and elegantly expressible as *conditional rewrite rules* on imperative programs, where the conditions are *formulas in a suitable temporal logic*. In this paper the temporal logic is an extension of CTL with free variables. The first transformation example expresses *dead code elimination*, the second expresses *constant folding* and the third expresses *loop invariant hoisting*. The first involves computational futures, the second, computational pasts and the third, involves both the computational future and past.

Second, the optimizing transformations are proven to be *fully semantics preserving* : in each case, if π is a program and π' is the result of transforming it, a *induction* relation is established between the computations of π and π'. A consequence is that if π has a terminating computation with "final answer" v, then π' also has a terminating computation with the same final answer; and vice versa.

*This research was partially supported by the Danish Natural Science Research Council (*PLT* project), the EEC (*Daedalus* project) and Microsoft Research.

POPL '02, Jan. 16-18, 2002 Portland, OR USA © 2002 ACM ISBN 1-58113-450-9/02/01. . .$5.00

1.1 Compiler optimizing transformations

A great many program transformations are done by optimizing compilers; an exhaustive catalog may be found in [20]. These have been a great success pragmatically, so it is important that there be no serious doubt of their correctness: that transformed programs are always semantically equivalent to those from which they were derived.

Proof of transformation correctness must, by its very nature, be a *semantics-based* endeavor.

1.2 Semantics-based program manipulation

Much has happened in this field since the path-breaking 1977 Cousot paper [5] and 1980 conference [11]. The field of "abstract interpretation" [1, 5, 6, 13, 23, 25] arose as a mainly European, theory-based counterpart to the well-developed more pragmatic North American approach to program analysis [2, 10, 21]. The goal of semantics-based program manipulation ([12] and the PEPM conference series) is to place program analysis and transformation on a solid foundation in the semantics of programming languages, making it possible to prove that analyses are sound and that transformations do not change program behaviors.

This approach succeeded well in placing on solid semantic foundations some program *analyses* used by optimizing compilers, notable examples being sign analysis, constant propagation, and strictness analysis. An embarrassing fact must be admitted, though: Rather less success was achieved by the semantics-based approach toward the goal of validating correctness of program *transformations*, in particular, data-flow analysis-based optimizations as used in actual compilers.

One root of this problem: Semantic frameworks such as denotational and operational semantics describe program execution in precise mathematical or operational terms; but representation of *data dependencies* along computational futures and pasts is rather awkward, even when continuation semantics is used. Worse, such dependency information lies

at the heart of the most widely used compiler optimizing transformations.

1.3 Semantics-based transformation correctness

Transformation correctness is somewhat complex to establish, as it involves proving a soundness relation among three "actors": the *condition* that enables applying the transformation and the semantics of the subject program both *before* and *after* transformation. Denotational and operational semantics (e.g., [33]) typically present many example proofs of equivalences between program fragments. However most of these are small (excepting the monumental and indigestible [19]), and their purpose is mainly to illustrate proof methodology and subtle questions involving Scott domains or program contexts, rather than to support applications.

A problem is that denotational and operational methods seem ill-suited to validating transformations that involve a program's computational future or computational past. Even more difficult are transformations that change program control flow, notable examples being "code motion" and "strength reduction."

Few formal proofs have been made of correctness of such transformations. Two works relating the semantics-based approaches to transformation correctness: Nielson's thesis [24] has an unpublished chapter proving correctness of "constant folding" perhaps omitted from the journal paper [23], because of the complexity of its development and proof. Havelund's thesis [9] carefully explores semantic aspects of transformations from a Pascal-like mini-language into typical stack-based imperative intermediate code, but correctness proofs were out of its scope (and would have been impractically complex in a denotational framework, witness [19]).

Another approach to verifying the correctness of compiler optimizations is presented by Kozen and Patron [16]. Using an extension of Kleene algebra, Kleene algebra with tests (KAT), an extensive collection of instances of program transformations are proven correct, i.e. a concrete optimization is proven correct given a concrete source program and the transformed program. Programs are represented as algebraic terms in KAT and it is shown that the original and transformed program are equal under the algebraic laws of KAT. In many instances these KAT terms are not ground, that is, they contain variables, and thus the reasoning could be applied to general program transformations. One has to note, however, that the paper sets out with a different perspective on program transformation. The paper is geared towards establishing a framework where one can formally reason about program manipulations specified as KAT equalities that imply semantic equivalence. Although the results in the paper are not directly applicable to compilers since no automatic method is given for applying the optimizations described, they are however applicable to proof carrying code (PCC) [22] and efficient code certification (ECC) [15], as Kozen and Patron state. In ECC, an optimized program contains a proof verifying that the transformations applied to it were sound, in which case proofs about specific instances are exactly what is needed.

In contrast, the present paper aims to formalize a framework for describing and formally proving classical compiler optimizations. We claim that these specifications (once proven correct) can be directly and automatically utilized in optimizing compliers. It is also worth noting that Kozen and Patron study only well-structured programs, i.e. those with *while* loops but no *goto* statements, whereas we do not make this restriction.

Some transformation correctness proofs have been made for functional languages, especially the work by Wand and colleagues, for example [31], using "logical relations." These methods are mathematically rather more sophisticated than those of this paper, which seem more appropriate for traditional intermediate-code optimizations.

1.4 Model checking and program analysis

This situation has improved with the advent of model checking approaches to program analysis [4, 27, 29, 30, 32, 28]. Work by Steffen and Schmidt [29, 30] showed that temporal logic is well-suited to describing data dependencies and other program properties exploited in classical compiler optimizations. In particular, work by Knoop, Steffen and Rüthing [14] showed that new insights could be gained from using temporal logic, enabling new and stronger code motion algorithms, now part of several commercial compilers.

More relevant to this paper: The code motion transformations could be proven correct.

1.5 Model checking and program transformation

In this paper we give a formalism (essentially it is a subset of [18]) for succinctly expressing program transformations, making use of temporal logic; and use this formalism to prove the universal correctness (semantics preservation for all programs) of the three optimizing transformations: dead code elimination, constant folding and loop invariant hoisting. The thrust of the work is not just to prove these three transformations correct, though, but rather to establish a framework within which a wide spectrum of classical compiler optimizations can be validated. More instances of this paper's approach may be found in [7].

Many optimizing transformations can be elegantly expressed using rewrite rules of form: $I \Longrightarrow I' \,\text{if}\, \phi$, where I, I' are intermediate language instructions and ϕ is a property expressed in a temporal logic suitable for describing program data flow. Its reading [18]: If the current program π contains an instruction of form I at some control point p, and if flow condition ϕ is satisfied at p, then replace I by I'.

The purpose of this paper is to show how such transformations may be proven correct. The meaning of correctness is that for any program π, if $Rewrite(\pi, \pi', p, I \Longrightarrow I' \,\text{if}\, \phi)$ then $[\![\pi]\!] = [\![\pi']\!]$, i.e. π and π' have exactly the same semantics.

2. PROGRAMS AND TRANSFORMATIONS

In this section we provide fundamental definitions used in our representations of programs, analyses, transformations,

```
0:  read N;
1:  Five := 5
2:  Sum := 0
3:  C := Five
4:  Sum := Sum + C * N
5:  N := N-1
6:  if N goto 4 else 7
7:  write Sum
```

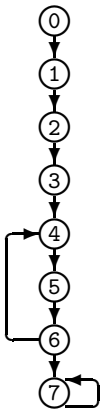

Figure 2.1: Example program and control flow.

and correctness proofs. In section 2.1 we introduce a simple imperative programming language that we use to demonstrate program transformations and their proofs of correctness. In section 2.2 we describe the control flow graph representation of programs that serves as the model over which the temporal logic formulas in the rewrite rules are checked. The temporal logic CTL with free variables is presented in section 2.3. The rewriting rules are defined in section 2.4 and section 2.5 provides the specifications for the dead code elimination and constant folding transformations.

2.1 A simple programming language

Definition 1 A *program* π has form:

$$\pi \;=\; \texttt{read X;}\ I_1\texttt{;}\ I_2\texttt{;}\ \ldots\ \ I_{m-1}\texttt{; write Y}$$

where $I_1, \ldots I_{m-1}$ are *instructions*, labeled by the *program labels* of π in $Nodes_\pi = \{0, 1, 2, \ldots, m\}$. Further, let instruction I_0 be the initial `read` X, and instruction I_m be the concluding `write` Y. The `read` and `write` instructions must, and can only, appear respectively at the beginning and end of a program. The syntax of all other instructions in π is given by the following grammar:

Inst	$\ni$	I	::=	`skip` | X `:=` E | `if` X `goto` n `else` n'
Expr	$\ni$	E	::=	X | $O\ E...E$
Op	$\ni$	O	::=	various unspecified operators o, each with $arity(o) \geq 0$
Var	$\ni$	X	::=	`X` | `Y` | `Z` | ...
Label	$\ni$	n, n'	::=	$1 \mid 2 \ldots \mid m$

Semantics are as expected and are formally defined below in Section 3. Figure 2.1 contains an example program. For readability it has explicit instruction labels, and operators are written in infix position.

In order to provide a simple framework for proving correctness this language has no exceptions or procedures. We expect the technique can be extended to include such features and maintain its fundamental nature, but this is future work.

2.2 Modeling program control flow

In order to reason about the program with a view to transform it, we look at the *control flow graph* of the program. This is a type of transition system. We introduce this concept here and will further use it to describe the semantics of a program:

Definition 2 A *transition system* is a pair $\mathcal{T} = (S, \rightarrow)$, where S is a set and $\rightarrow \subseteq S \times S$. The elements of S are referred to as *states* or *nodes*.

A *path* is a maximal sequence of nodes (finite[1] or infinite) $n_0 \rightarrow n_1 \rightarrow \ldots$ such that $\forall i \geq 0, n_i \rightarrow n_{i+1}$. A *backwards path* is a path over the inverse of $\rightarrow$ (written as $\rightarrow^\circ$) and written as $n_0 \rightarrow^\circ n_1 \rightarrow^\circ \ldots$ or $n_0 \leftarrow n_1 \leftarrow \ldots$.

Closely related to transition systems are models (as used in model checking). These are transition systems where each state is labeled with certain information:

Definition 3 A *model* is a triple $\mathcal{M} = (S, \rightarrow, L)$ where $(S, \rightarrow)$ is a transition system, and *labeling* function $L : S \rightarrow 2^P$ labels each state in S with a set of propositions in P.

The control flow transition system is a system where states are program points and transitions are between pairs of program points that could follow each other in the execution.

Definition 4 The *control flow transition system for* π is $\mathcal{T}_{cf}(\pi) = (Nodes_\pi, \rightarrow_{cf})$ where the (total) relation $\rightarrow_{cf}$ is defined by $n_1 \rightarrow_{cf} n_2$ if and only if

$$\begin{array}{ll} & (I_{n_1} \in \{X{:=}E, \texttt{skip}, \texttt{read}\ X\} \wedge n_2 = n_1 + 1) \\ \vee & (I_{n_1} = \texttt{if}\ X\ \texttt{goto}\ n\ \texttt{else}\ n' \wedge (n_2 = n \vee n_2 = n')\ ) \\ \vee & (I_{n_1} = \texttt{write}\ Y \wedge n_2 = n_1) \end{array}$$

We set up a control flow model by labeling the states of the system (program points in this case) with propositions of interest. These will include the instruction at that program point plus information on which variables are defined or used at that point. Figure 2.2 shows an example model in which node 2, whose instruction `Sum := 0` is labeled by the propositions $node(2)$, $stmt(\texttt{Sum} := \texttt{0})$, $def(\texttt{Sum})$, and $conlit(\texttt{0})$.

Definition 5 The *control flow model for program* π is defined as $\mathcal{M}_{cf}(\pi) = (Nodes_\pi, \rightarrow_{cf}, L)$ where $(Nodes_\pi, \rightarrow_{cf})$ are as in Definition 4, and $L(n)$ is defined as follows for $n \in Nodes_\pi$:

[1] A finite path $n_0 \rightarrow n_1 \rightarrow \ldots \rightarrow n_m$ is maximal if $\forall n, \neg(n_m \rightarrow n)$. That is, n_m has no successors.

$$
\begin{array}{lcl}
S & = & \{0,1,2,3,4,5,6,7\} \\
\rightarrow & = & \{0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow 4, 4 \rightarrow 5, 5 \rightarrow 6, \\
 & & \quad 6 \rightarrow 7, 6 \rightarrow 4, 7 \rightarrow 7\}
\end{array}
$$

$$
\begin{array}{lcl}
L(0) & \supseteq & \{node(0), stmt(\texttt{read N}), def(\texttt{N})\} \\
L(1) & \supseteq & \{node(1), stmt(\texttt{Five := 5}), def(\texttt{Five}), conlit(\texttt{5})\} \\
L(2) & \supseteq & \{node(2), stmt(\texttt{Sum := 0}), def(\texttt{Sum}), conlit(\texttt{0})\} \\
L(3) & \supseteq & \{node(3), stmt(\texttt{C := Five}), def(\texttt{C}), use(\texttt{Five})\} \\
L(4) & \supseteq & \{node(4), stmt(\texttt{Sum := Sum + C * I}), def(\texttt{Sum}), \\
 & & \quad use(\texttt{Sum}), use(\texttt{C}), use(\texttt{I})\} \\
L(5) & \supseteq & \{node(5), stmt(\texttt{N := N - 1}), def(\texttt{N}), use(\texttt{N}), \\
 & & \quad conlit(\texttt{1})\} \\
L(6) & \supseteq & \{node(6), stmt(\texttt{if N goto 4 else 7}), use(\texttt{N})\} \\
L(7) & \supseteq & \{node(7), stmt(\texttt{write Sum}), use(\texttt{Sum})\}
\end{array}
$$

Figure 2.2: Control flow model for the example program.

$$
\begin{array}{lcll}
L(n) & = & \{stmt(I_n) & | \quad 0 \leq n \leq m\} \\
 & \cup & \{node(\texttt{N}) & | \quad n = \texttt{N}\} \\
 & \cup & \{def(\texttt{X}) & | \quad I_n \text{ has form } \texttt{X:=E} \text{ or } \texttt{read X}\} \\
 & \cup & \{use(\texttt{X}) & | \quad I_n \text{ form: } \texttt{Y:=E} \text{ with } \texttt{X} \text{ in } \texttt{E}, \text{ or} \\
 & & & \quad I_n = \texttt{if X goto } p \texttt{ else } p'\} \\
 & \cup & \{use(\texttt{Y}) & | \quad n = m \text{ and } I_n = \texttt{write Y}\} \\
 & \cup & \{conlit(\texttt{0}) & | \quad \texttt{0} \text{ is a constant in } I_n \\
 & & & \quad \text{(operator with } arity(\texttt{0}) = 0)\} \\
 & \cup & \{trans(\texttt{E}) & | \quad \texttt{E} \text{ is an expression in } \pi \text{ and} \\
 & & & \quad I_n \text{ is not of form: } \texttt{X:=E}' \text{ or} \\
 & & & \quad \texttt{read X} \text{ with } \texttt{X} \text{ in } vars(\texttt{E}) \ \}
\end{array}
$$

The predicates $stmt(I)$, $def(X)$, $use(X)$, $conlit(O)$, $trans(E)$ are the building blocks for the conditions that specify when optimizing transformations can be safely applied. These conditions are specified as CTL-FV formulas.

Definition 6 An expression E is *transparent* at a program point if none of the variables in the expression are defined at that point (i.e. assigned by := or read).

2.3 CTL with free variables

The temporal logic CTL-FV used in specifying transformation conditions is in two respects a generalization of CTL [3]. First, as is common, the temporal path quantifiers E and A are extended to also quantify over *backwards paths* in the obvious way. Our notation for this: $\overleftarrow{E}$ and $\overleftarrow{A}$. These paths may be finite and our quantifications over paths are thus over infinite and maximal finite paths. That is, we consider a branching notion of past which may be *either* finite, as in CTL_{bp} [17], or infinte, as in $POTL$ [26, 34]. A branching past is more appropriate here than the linear past in $PCTL*$ [8] which can also be used to augement branching time logics with past time operators.

Second, propositions are generalized to *predicates over free variables*. (A traditional atomic proposition is simply a predicate with no arguments.) For example, the formula $stmt(\texttt{x := e})$ where $stmt \in Pr$, has free variables x and e ranging over program variables and expressions, respectively. These free variables will henceforth be called *CTL-variables* to avoid confusion with variables or program points appearing in the program being transformed or analyzed.

The effect of model checking will be to bind CTL-variables to program points or bits of program syntax, e.g., dead variables or available expressions.

CTL-FV formulas are either *state* or *path* formulas generated by the grammar with non-terminals $\{\phi, \psi\}$, terminals $true$, $false$, $pr \in Pr$ and free variables $x_1, \ldots, x_n$, start symbol ϕ and the productions:

$$
\begin{array}{lll}
\phi ::= true & \phi ::= E\ \psi & \psi ::= X\ \phi \\
\phi ::= false & \phi ::= A\ \psi & \psi ::= \phi\ U\ \phi \\
\phi ::= pr(x_1, \ldots, x_n) & \phi ::= \overleftarrow{E}\ \psi & \psi ::= \phi\ W\ \phi \\
\phi ::= \phi \wedge \phi & \phi ::= \overleftarrow{A}\ \psi & \\
\phi ::= \neg\phi & &
\end{array}
$$

Operational interpretation: A model checker will not simply find which nodes in a model satisfy a (state) formula, but will instead find the instantiation substitutions that satisfy the formula. Mathematically, we model this by extending the satisfaction relation $n \models \phi$ to include a substitution θ binding its free variables. The extended satisfaction relation $n \models_\theta \phi$ will hold for any θ such that $n \models \theta(\phi)$. This relation is defined in Figure 2.3. All is as usual, except for θ and the interpretation of W on maximal finite paths.

The job of the model checker is thus, given ϕ, to return the set of all n and θ such that $n \models_\theta \phi$. For the example program in Figure 2.1 and formula $def(x) \wedge use(x)$, the model checker returns the following set of instantiation substitutions. (For brevity, CTL-variable n is bound to the program point in the substitutions.)

$$\{\theta_1, \theta_2\} = \{[n \mapsto 4, x \mapsto \texttt{Sum}], [n \mapsto 5, x \mapsto \texttt{N}]\}$$

Of particular interest when analysing the control flow model is the universal weak until operator ($A\ W$). This is due to the following lemma:

Lemma 1 If $n_0 \models A(\phi_1\ W\ \phi_2)$ then for any *maximal finite* path $n_0 \rightarrow \ldots \rightarrow n_N$, there exists an j such that $n_j \models \phi_2$ and $\forall_{0 \leq i < j}\ n_i \models \phi_1$.

A similar result holds for backwards paths.

Proof Omitted.

We use this lemma in the correctness proofs since the maximal finite paths are exactly the set of terminating program traces of the execution transition system of Definition 10.

2.4 Rewriting

Definition 7 A *rewrite rule* has the form $I \Longrightarrow I'$ if ϕ, where I, I' are instructions built from the program and CTL variables, and ϕ is a CTL-FV temporal logic formula. By definition $Rewrite(\pi, \pi', n, I \Longrightarrow I' \text{ if } \phi)$ is true if and only if for some substitution θ, the following hold:

State Formulas:

$n \models_\theta true$ iff $true$
$n \models_\theta false$ iff $false$
$n \models_\theta pr(x_1, \ldots, x_n)$ iff $pr(\theta x_1, \ldots, \theta x_n) \in L(n)$
$n \models_\theta \neg\phi$ iff not $n \models_\theta \phi$
$n \models_\theta \phi_1 \wedge \phi_2$ iff $n \models_\theta \phi_1$ and $n \models_\theta \phi_2$

$n \models_\theta E\ \psi$ iff $\exists path\ \ p = n \rightarrow n_1 \rightarrow n_2 \ldots, [p \models_\theta \psi]$
$n \models_\theta A\ \psi$ iff $\forall paths\ p = n \rightarrow n_1 \rightarrow n_2 \ldots, [p \models_\theta \psi]$
$n \models_\theta \overleftarrow{E}\ \psi$ iff $\exists path\ \ p = n \rightarrow^\circ n_1 \rightarrow^\circ n_2 \ldots, [p \models_\theta \psi]$
$n \models_\theta \overleftarrow{A}\ \psi$ iff $\forall paths\ p = n \rightarrow^\circ n_1 \rightarrow^\circ n_2 \ldots, [p \models_\theta \psi]$

Path Formulas: (below $\rightarrow'$ is $\rightarrow$ or $\rightarrow^\circ$)

$p \models_\theta X\ \phi$ iff $p = n_0 \rightarrow' n_1 \rightarrow' \ldots$ and $n_1 \models_\theta \phi$

$p \models_\theta \phi_1\ U\ \phi_2$ iff $p = n_0 \rightarrow' n_1 \rightarrow' \ldots$ and
$\exists i \geq 0\ [n_i \models_\theta \phi_2 \wedge \forall j[0 \leq j < i$ implies $n_j \models_\theta \phi_1]]$

$p \models_\theta \phi_1\ W\ \phi_2$ iff $p = n_0 \rightarrow' n_1 \rightarrow' \ldots$ and
$(\exists k \geq 0\ [n_k \models_\theta \phi_2$ and $\forall i, 0 \leq i < k \implies n_i \models_\theta \phi_1)]$
or $(\forall k \geq 0\ [n_k \models_\theta \phi_1$ and n_{k+1} exists])

Figure 2.3: CTL-FV satisfaction relation

$n \models_\theta stmt(I) \wedge \phi$
π = `read X;` I_1; $\ldots I_n; \ldots I_{m-1}$; `write Y`,
where $I_n = \theta(I)$, and
π' = `read X;` I_1; $\ldots \theta(I')$; $\ldots I_{m-1}$; `write Y`

Sometimes we may want to alter the program at more than one point. In this case we specify several rewrites and side conditions at once. For example, to transform two nodes the form of the rewrite would be:

$$\begin{array}{l} n : I_1 \Longrightarrow I_1' \\ m : I_2 \Longrightarrow I_2' \\ \text{if} \\ n \models \phi_1 \\ m \models \phi_2 \end{array}$$

The operational interpretation of this is that we find a substitution θ that satisfies both of $n \models_\theta stmt(I_1) \wedge \phi_1$ and $m \models_\theta stmt(I_2) \wedge \phi_2$ and then use this substitution to alter the program in the two relevant places.

2.5 Sample transformations

Following are versions of three classical optimizations (simplified in comparison to compiler practice, to make it easier to follow the techniques used in the proofs).

For convenience we express code removal as replacement of an instruction by `skip`, and code motion as simultaneous replacement of an instruction I and `skip` by (respectively) `skip` and instruction I. We assume the compiler will remove useless occurrences of `skip`.

While most programmers do not write code that contains dead code or opportunities for constant folding it often results from other transformations, especially automated ones.

Dead Code Elimination: Dead code elimination removes assignment statements that assign a value that is never used. In our model, the rewrite replaces the assignment with the `skip` instruction:

$$x := e \implies \texttt{skip}$$

The side condition on the rewrite must specify that the value assigned is never referenced again. This is exactly the kind of condition that temporal logic can specify. The rewrite rule with its side condition is thus written

$$x := e \implies \texttt{skip} \text{ if } AX\ \neg E(\ true\ U\ use(x)\ ).$$

Since we do not care whether x is used at the current node, we skip past it with the AX operator.

Constant Folding: Constant folding is a transformation that replaces a variable reference with a constant value:

$$x := y \implies x := c.$$

One method of implementing constant folding for a variable `Y` is to check whether all possible assignments to `Y` assign it the same constant value. To check this condition we use the past temporal operators, specifying the complete transformation as follows:[2]

$$\begin{array}{l} x := y \implies x := c \\ \text{if} \\ \overleftarrow{A}(\neg def(y)\ W\ stmt(y := c) \wedge conlit(c)) \end{array}$$

Code motion/loop invariant hoisting: A restricted version of a "code motion" transformation (CM) that covers the "loop invariant hoisting" transformation is defined as

$$\begin{array}{l} p : \texttt{skip} \Longrightarrow x{:=}e \\ q : x{:=}e \Longrightarrow \texttt{skip} \\ \text{if} \\ p \models\ \ A(\neg use(x)\ W\ node(q)) \\ q \models\ \ \neg use(x) \wedge \\ \qquad \overleftarrow{A}((\neg def(x) \vee node(q)) \wedge trans(e)\ W\ node(p)) \end{array}$$

This transformation involves two (different) statements in the subject program. The transformation moves an assignment at label q to label p provided that two conditions are met:

1. The assigned variable x is dead after p and remain so until q is reached. If this requirement holds, then introducing the assignment $x{:=}e$ at label p will not change the semantics of the program.

2. The second requirement (in combination with the first rewrite rule) states that the expression e should be available at q *after* the transformation.

[2]The *conlit* is introduced so that the modelchecker will not match c with a non-constant expression.

This transformation could also be obtained by applying two transformations: One that inserts the statement $x{:=}e$ provided that x is dead between p and q, followed by the elimination of available expressions transformation. With the two transformations one would need some mechanism of controlling where to insert which assignments. By formulating the transformation as a single transformation, the two labels p and q are explicitly linked.

Since all paths from p may not evelually reach q, it is possible to move assignments to labels such that e is still available in q and x is dead in all paths not leading to q, which would still be a semantics preserving transformation.

```
1 :  skip;
2 :  if ... then 3 else 6;
3 :  x := a + b;
4 :  y := y − 1;
5 :  if y then 3 else 6;
6 :  x := 0;
```

In this program is is possible to lift the statement `x:=a+b` from label 3 to label 1. In general the transformation by itself could slow down the computation, since there is no need to compute the expression if the expression is not needed.

Note that we use the weak until (W), this is so that the transformation is not disabled by cycles in the control flow graph that do not affect the correctness of the transformation.

2.6 Computational aspects

We discuss computational aspects only briefly; more can be found in [18] and related papers.

Model checking with respect to $I \Longrightarrow I' \,\mathrm{if}\, \phi$ yields a set of pairs $\{(p_1, \theta_1), \ldots, (p_k, \theta_k)\}$ satisfying ϕ. Consequence: $\{p_1, \ldots, p_k\}$ is the set of *all* places where this rule can be applied. For instance, all places where dead code elimination can be done are found by a single model check.

The time to model check $n \models p$ for transition system $\mathcal{T}$ is a low-degree polynomial, near linear for many transition systems, and $|\mathcal{T}|^2 \cdot |\phi|$ in the worst case. Of course, in the case of model checking CTL-FV formulas times could be higher, since $|\mathcal{T}|$ depends on the size of labelling function $L : \mathit{Nodes}_\pi \to 2^{AP}$ as in Definition 5. For each node n, $L(n)$ can be found in time proportional at most to the size of the instruction I_n, with one exception: propositions $\mathit{trans}(\mathtt{E})$, which can take time proportional to the size of π. For greater efficiency these can be treated specially, maintaining a single global table for the transparency relation.

Experience from [18] and related work indicates that their algorithm for model checking CTL-FV is not too expensive in practice, i.e. that the free variables do not impose an unreasonable time cost.

3. PROGRAM SEMANTICS

In this section we define the semantics of the simple programming language introduced in Definition 1. In Section 5 we use this semantics to show for a program π and its transformed version π' that their semantics are the same. That is, $[\![\pi]\!] = [\![\pi']\!]$.

Definition 8 (Semantic framework) We assume the following have been fixed in advance, and apply to all programs:

- A set *Value* of values (not specified here), containing a designated element *true*.
- A fixed interpretation of every n-ary operator symbol o as a function $[\![o]\!] : \mathit{Value}^n \to \mathit{Value}$. Note that $[\![o]\!] \in \mathit{Value}$ if $n = 0$.

Definition 9 (Expression evaluation) A *store* is a function $\sigma \in \mathit{Store} = \mathit{Var} \rightharpoonup \mathit{Value}$. *Expression evaluation* $[\![\mathit{Expr}]\!] : \mathit{Store} \to \mathit{Value}$ is defined by:

$$\begin{array}{rcl} [\![\mathtt{X}]\!]\sigma & = & \sigma(\mathtt{X}) \\ [\![\mathtt{o}\ \mathtt{E}_1 \ldots \mathtt{E}_n]\!]\sigma & = & [\![\mathtt{o}]\!]([\![\mathtt{E}_1]\!]\sigma, \ldots, [\![\mathtt{E}_n]\!]\sigma) \end{array}$$

Define $\sigma\backslash\mathtt{X}$ to be the store function σ restricted to its original domain minus `X`. Further, $\sigma[\mathtt{X} \mapsto v]$ is the same as σ except that it maps `X` to v.

Definition 10 (Semantics) At any point in its computation, the program will be in a *state* of the form $s = (p, \sigma) \in \mathit{State} = \mathit{Nodes}_\pi \times \mathit{Store}$. The *Initial state* for input $v \in \mathit{Value}$ is $\mathit{In}(v) = (0, \sigma)$ where $\sigma(\mathtt{X}) = v$ and $\sigma(\mathtt{Z}) = \mathit{true}$ for all other variables appearing in program π.

The *state transition relation* $\to \subseteq \mathit{State} \times \mathit{State}$ is defined by:

1. If $I_p = $ `skip` or $I_p = $ (`read X`) then $(p, \sigma) \to (p+1, \sigma)$.
2. If $I_p = $ (`X := E`) then $(p, \sigma) \to (p+1, \sigma[\mathtt{X} \mapsto [\![\mathtt{E}]\!]\sigma])$.
3. If $I_p = $ (`if X goto` p' `else` p'') and $\sigma(\mathtt{X}) = \mathit{true}$ then $(p, \sigma) \to (p', \sigma)$.
4. If $I_p = $ (`if X goto` p' `else` p'') and $\sigma(\mathtt{X}) \neq \mathit{true}$ then $(p, \sigma) \to (p'', \sigma)$.
5. $(m, \sigma) \to (m, \sigma)$ for any store σ.

Note that the `read X` has no effect on the store σ since the initial value v of `X` is set in the initial state.

The operational semantics of a program is given the form of a transition system: the execution transition system $\mathcal{T}_{run}$.

Definition 11 The *execution transition system* for program π and input $v \in \mathit{Value}$ is by definition

$$\mathcal{T}_{run}(\pi, v) = (\mathit{Nodes}_\pi \times \mathit{Store}, \to)$$

where $s_1 \to s_2$ is as in Definition 10.

Definition 12 The *semantic function* is the partial function $[\![\_]\!] : \mathit{Value} \rightharpoonup \mathit{Value}$ defined by:

$$[\![\pi]\!](v) = \sigma(\mathtt{Y})$$

if there exists a finite sequence

$$s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_t = (m, \sigma)$$

In order to reason about the computational history of program executions we also introduce the notion of computational prefix, and a corresponding transition system $\mathcal{T}_{pfx}$ (both defined below).

The control flow model defined above in Definition 5 is an abstraction of each of these, see Lemma 3. These are used in the correctness proofs to relate a program's semantics to its control flow model.

4. A METHOD FOR SHOWING SEMANTIC EQUIVALENCE

For all rewrite rules $I \Longrightarrow I'$ if ϕ we need to show that $\mathit{Rewrite}(\pi, \pi', p, I \Longrightarrow I' \text{ if } \phi)$ implies $[\![\pi]\!] = [\![\pi']\!]$, i.e. for all input v, $\mathit{In}(v) \rightarrow^* (m, \sigma)$ is a terminating computation for program π if and only if $\mathit{In}(v) \rightarrow^* (m', \sigma')$ is a terminating computation for program π' with $\sigma(\mathtt{Y}) = \sigma'(\mathtt{Y})$. The problem now is how to link the *temporal* property ϕ, which concerns "futures" and "pasts", to the transformation $I \Longrightarrow I'$.

For this it is not sufficient to regard states one at a time due to operators, such as AU and $\overleftarrow{A}U$, giving access to information computed earlier or later. Our solution is to enrich the semantics and its transition system by considering *computation prefixes* of form:

$$C = \pi, v \vdash s_0 \rightarrow \ldots \rightarrow s_t.$$

Now suppose $p \models \phi$ has been model checked. The resulting substitutions (see Lemma 3 below) also describe the computation prefix C. Conclusion: The results of the model check, contain information about the state sequence in C, thus relating past and present states.

What about futures? Our choice is to build a *transition system* $\mathcal{T}_{pfx}(\pi, v)$ so $C \rightarrow C_1 \in \mathcal{T}_{pfx}(\pi, v)$ if and only if C_1 is identical to C, but with one additional state:

$$C_1 = \pi, v \vdash s_0 \rightarrow \ldots \rightarrow s_t \rightarrow s_{t+1}.$$

Now reasoning that involves futures can be done by ordinary induction: assuming $C \mathcal{R} C'$, show $C \rightarrow C_1$ implies $C' \rightarrow C_1'$ for a C_1' with $C_1 \mathcal{R} C_1'$, and $C' \rightarrow C_1'$ for C_1 with $C_1 \mathcal{R} C_1'$.

Definition 13 For a program π and initial value $v \in \mathit{Value}$, a *computation prefix* is an sequence (finite or infinite)

$$\pi, v \vdash s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots$$

such that $s_0 = \mathit{In}(v)$ and $s_i \rightarrow s_{i+1}$ for $i = 0, 1, 2, \ldots$

Definition 14 The *computation prefix transition system* for program π and input $v \in \mathit{Value}$ is by definition

$$\mathcal{T}_{pfx}(\pi, v) = (\mathcal{C}, \rightarrow)$$

where $\mathcal{C}$ is the set of all finite computation prefixes, and $C_1 \rightarrow C_2$ if and only if

$$\begin{aligned} C_1 &= \pi, v \vdash s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_t, \\ C_2 &= \pi, v \vdash s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_t \rightarrow s_{t+1}. \end{aligned}$$

where $s_t \rightarrow s_{t+1}$ is the state transition relation from Definition 10. Note that we use the same symbol, $\rightarrow$, to represent both the transition relation for the execution transition system $\mathcal{T}_{run}$ and the computation prefix transition system $\mathcal{T}_{pfx}$ but that the relations can be distinguished by their context.

Goal: Show that if C, C' are computation prefixes of π, π' on same input v then $s_i \mathcal{R} s_j'$ for every corresponding pair of states in C, C' where $\mathcal{R}$ is a relation on states that expresses "correct simulation".[3]

Consider two programs, π and π' such that:

$$\pi \;=\; \mathtt{read\ X}; I_1; I_2; \ldots I_{m-1}; \mathtt{write\ Y}$$

and

$$\pi' \;=\; \mathtt{read\ X}; I_1'; I_2'; \ldots I_{m'-1}'; \mathtt{write\ Y}.$$

The aim is to show that π and π' are semantically equivalent, $[\![\pi]\!] = [\![\pi']\!]$. That is, for any value v either both $[\![\pi]\!](v)$ and $[\![\pi']\!](v)$ are not defined or for any computation prefix

$$\pi, v \vdash \mathit{In}(v) \rightarrow (p_1, \sigma_1) \rightarrow \ldots \rightarrow (m, \sigma)$$

there exists a computation prefix for the transformed program

$$\pi', v \vdash \mathit{In}(v) \rightarrow (p_1', \sigma_1') \rightarrow \ldots \rightarrow (m', \sigma')$$

such that $\sigma(\mathtt{Y}) = \sigma'(\mathtt{Y})$, and conversely. We can naturally prove this result by induction on the length of the prefixes. In practice the induction hypothesis needs to be strengthened for the proof to work. The general form of the strengthened hypothesis is that a relation $\mathcal{R}$ holds between computation prefixes of the original program and computation prefixes of the transformed program.

Noting that the transitions between prefixes are deterministic (since the language is), we can see that proving the step case of induction is achieved by proving that the relation between two prefixes is preserved by any one step in the transition system. The following lemma details the work that needs to be done to show semantic equivalence.

Lemma 2 (Program Equivalence/Induction)

Programs π and π' are semantically equivalent: $[\![\pi]\!] = [\![\pi']\!]$ if there exists a relation $\mathcal{R}$, such that for all values v:

1. (Base Case) $\mathcal{R}$ holds between the initial computation prefixes i.e.

$$((\pi, v \vdash \mathit{In}(v)), (\pi', v \vdash \mathit{In}(v))) \in \mathcal{R}$$

2. (Step Case) If $C_1 \mathcal{R} C_1'$, $C_1 \rightarrow C_2$ and $C_1' \rightarrow C_2'$ then

$$C_2 \mathcal{R} C_2'.$$

[3]This is actually closer to bi-simulation.

3. (Equivalence) If

$$C\mathcal{R}C' \text{ and}$$
$$C = \pi, v \vdash s_0 \rightarrow s_1 \ldots \rightarrow (p_t, \sigma) \text{ and}$$
$$C' = \pi', v \vdash s'_0 \rightarrow s'_1 \ldots \rightarrow (p_{t'}, \sigma')$$

then

$$(i) \quad p_t = m \iff p'_{t'} = m \text{ and}$$
$$(ii) \quad p_t = p'_{t'} = m \implies \sigma_t(\mathtt{Y}) = \sigma'_{t'}(\mathtt{Y})$$

So proofs of equivalence are split into these three steps. Unsurprising, it is the step case of induction that is the hardest to prove.

If a flow condition in a rewrite rule holds then it states a fact about the control flow graph. This relation between this graph and the computational prefix system is captured in the following lemma that follows from the definitions of the two systems:

Lemma 3 Suppose $C = \pi, v \vdash s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_t$ is a computation prefix in $\mathcal{T}_{pfx}(\pi, v)$, where $s_i = (p_i, \sigma_i)$. For any $0 \leq j \leq t$:

1. $p_j \rightarrow_{cf} p_{j+1} \rightarrow_{cf} \ldots \rightarrow_{cf} p_t$ is a path in $\mathcal{T}_{cf}(\pi)$
2. $p_j \rightarrow_{cf} p_{j-1} \rightarrow_{cf} \ldots \rightarrow_{cf} p_0$ is a maximal backwards path in $\mathcal{T}_{cf}(\pi)$

Here we can see that the flow graph is an abstraction of the semantics of the program.

We know that if we are not at the program point specified in the rewrite then the instruction in the original program and the transformed program will coincide:

Lemma 4 Suppose π and π' are programs that are related by $Rewrite(\pi, \pi', p, R)$, then $p \neq q \Rightarrow I_q = I'_q$

We also know that if we are not at the transformed point in the program then the original program and the new program will behave identically:

Lemma 5 Suppose π and π' are programs that only differ by having a different instruction at program point p. Let $s_1 = (p_1, \sigma_1)$ with $p_1 \neq p$, then for any s_2:

$$\mathcal{T}_{run}(\pi, v) \vdash s_1 \rightarrow s_2 \Rightarrow \mathcal{T}_{run}(\pi', v) \vdash s_1 \rightarrow s_2$$

5. THE THREE EXAMPLES

The following lemma states that if an expression $\mathtt{E}$ does not contain the variable $\mathtt{X}$ and two stores σ and σ' differ at most in their value of $\mathtt{X}$ then the evaluation of $\mathtt{E}$ is the same under both stores.

Lemma 6 For any program variable $\mathtt{X}$, expression $\mathtt{E}$, and store σ, if $\mathtt{X} \notin vars(\mathtt{E})$ and $\sigma \setminus \mathtt{X} = \sigma' \setminus \mathtt{X}$ then $[\![\mathtt{E}]\!]\sigma = [\![\mathtt{E}]\!]\sigma'$.

Given this lemma, we can see that if a variable is not used in an instruction and two stores differ only by that variable then the program will behave in the same way:

Lemma 7 Suppose we have program points p, p_2, p'_2 and stores σ_1, σ'_1, σ_2, σ'_2 such that $(\sigma_1, p) \rightarrow (\sigma_2, p_2)$, $(\sigma'_1, p) \rightarrow (\sigma'_2, p'_2)$ and $\sigma_1 \setminus \mathtt{X} = \sigma'_1 \setminus \mathtt{X}$.

If $p \models \neg use(\mathtt{X})$ then $\sigma_2 \setminus \mathtt{X} = \sigma'_2 \setminus \mathtt{X}$ and $p_2 = p'_2$

Proof If $I_p = \mathtt{skip}$ then trivially $\sigma_2 \setminus \mathtt{X} = \sigma_1 \setminus \mathtt{X} = \sigma'_1 \setminus \mathtt{X} = \sigma'_2 \setminus \mathtt{X}$ and $p_2 = p + 1 = p'_2$.

If $I_p = (\mathtt{Z} := \mathtt{E})$ then $p \models \neg use(\mathtt{X})$ implies that $\mathtt{X} \notin vars(\mathtt{E})$. So by Lemma 6: $[\![\mathtt{E}]\!]\sigma_1 = [\![\mathtt{E}]\!]\sigma'_1$. So $\sigma_2 \setminus \mathtt{X} = \sigma'_2 \setminus \mathtt{X}$ as required. Trivially $p_2 = p + 1 = p'_2$.

If $I_p = (\mathtt{if\ Z\ goto}\ p'\ \mathtt{else}\ \mathrm{p}'')$ then $p \models \neg use(\mathtt{X})$ implies that $\mathtt{X} \neq \mathtt{Z}$. So $\sigma_1(\mathtt{Z}) = (\sigma_1 \setminus \mathtt{X})(\mathtt{Z}) = (\sigma'_1 \setminus \mathtt{X})(\mathtt{Z}) = \sigma'_1(\mathtt{Z})$, therefore $p'_2 = p_2$. The statement does not affect the stores so trivially $\sigma_2 \setminus \mathtt{X} = \sigma'_2 \setminus \mathtt{X}$. □

We also note the following lemma that states that if we have a series of instructions that do not define a variable then the value of the store with respect to that variable does not change.

Lemma 8 Consider a state sequence $(p_0, \sigma_0) \ldots (p_t, \sigma_t)$ in $\mathcal{T}_{run}(\pi, v)$ such that $(p_i, \sigma_i) \rightarrow (p_{i+1}, \sigma_{i+1})$ for $0 \leq i < t$. If the instruction at each p_i does not define a variable $\mathtt{X}$ then $\sigma_0(\mathtt{X}) = \sigma_t(\mathtt{X})$.

Lemma 9 Consider a state sequence $(p_0, \sigma_0) \ldots (p_t, \sigma_t)$ in $\mathcal{T}_{run}(\pi, v)$ such that $(p_i, \sigma_i) \rightarrow (p_{i+1}, \sigma_{i+1})$ for $0 \leq i < t$. If expression $\mathtt{E}$ is transparent at each p_i, then $[\![\mathtt{E}]\!]\sigma_i = [\![\mathtt{E}]\!]\sigma_t$.

5.1 Dead Code Elimination

The dead code elimination rewrite rule described earlier was:

$$x := e \implies \mathtt{skip} \text{ if } AX\ \neg E[\ true\ U\ use(x)\ ].$$

Following Definition 7 of rewriting, for this rewrite to apply the model checker must find a particular program point p and a substitution that maps x to a particular program variable $\mathtt{X}$ and e to a particular expression $\mathtt{E}$. In this case we need to prove that an original program π and transformed program π' are equivalent. Below, we assume that

$$Rewrite(\pi, \pi', p,\ x{:=}e \implies \mathtt{skip} \text{ if } AX\neg E[\ true\ U\ use(x)\ ]\ )$$

holds.

Definition 15 Consider $C \in \mathcal{T}_{pfx}(\pi, v)$ and $C' \in \mathcal{T}_{pfx}(\pi', v)$ such that:

$$C = \pi, v \vdash s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_t\ ,$$
$$C' = \pi', v \vdash s'_0 \rightarrow s'_1 \rightarrow \ldots \rightarrow s'_r$$

in which $\forall i[0 \leq i \leq t \implies s_i = (p_i, \sigma_i)]$ and $\forall i[0 \leq i \leq r \implies s'_i = (p_i, \sigma'_i)]$.

Then $C\mathcal{R}C'$ if and only if $t = r$ and for any $i, 0 \leq i \leq t$:

1. $p_i = p'_i$
2. $[\forall j, j < i \implies p \neq p_j] \implies \sigma_i = \sigma'_i$ and
3. $[\exists j, j < i \wedge p = p_j] \implies \sigma_i \backslash \mathtt{X} = \sigma'_i \backslash \mathtt{X}$

Base Case: Note that $C \mathcal{R} C$ for any $C \in \mathcal{C}$, i.e. $\mathcal{R}$ is reflexive. In particular it will hold for prefixes of length 1.

Step Case: Suppose $C_1 \mathcal{R} C'_1$. The language is deterministic so $C_1 \rightarrow C_2$ and $C'_1 \rightarrow C'_2$ for exactly one C_2 and C'_2. We need to show that $C_2 \mathcal{R} C'_2$. By Definition 15:

$$\begin{array}{rcl} C_1 & = & \pi, v \vdash (p_0, \sigma_0) \rightarrow \ldots \rightarrow (p_t, \sigma_t) \\ C'_1 & = & \pi', v \vdash (p_0, \sigma'_0) \rightarrow \ldots \rightarrow (p_t, \sigma'_t) \end{array}$$

Let I_p be the instruction in program π at p and I'_p be the instruction in program π' at p.

Suppose $p \neq p_i$ for $0 \leq i \leq t$. Then by Definition 15 each $\sigma_i = \sigma'_i$, so $C_1 = C'_1$, implying $C_2 = C'_2$ (by Lemma 5) and so $C_2 \mathcal{R} C'_2$.

Alternatively, suppose $p = p_t$. Then $I_p = (\mathtt{X} := \mathtt{E})$ and $I'_p = \mathtt{skip}$. So $\sigma_{t+1} = \sigma_t[\mathtt{X} \mapsto [\![\mathtt{E}]\!]\sigma_t]$ and $\sigma'_{t+1} \setminus \mathtt{X} = \sigma'_t \setminus \mathtt{X} = \sigma_t \setminus \mathtt{X}$. Therefore, $\sigma_{t+1} \setminus \mathtt{X} = \sigma'_{t+1} \setminus \mathtt{X}$ and $C_2 \mathcal{R} C'_2$.

Finally, consider $p \neq p_t$ and $p = p_k$ for some $k < t$. By Definition 15, $\sigma_t \setminus \mathtt{X} = \sigma'_t \setminus \mathtt{X}$. Thus, by Lemma 4, $I_{p_t} = I'_{p_t}$.

Now by Lemma 3, sequence $p_k \rightarrow_{cf} p_{k+1} \rightarrow_{cf} \ldots \rightarrow_{cf} p_t$ is a path in flow chart $\mathcal{T}_{cf}$ and by condition

$$p \models AX\neg E(\ true\ U\ use(\mathtt{X})\ )$$

we can conclude that $p_i \models \neg use(\mathtt{X})$ for all i with $k < i \leq t$. By Lemma 7 this implies $\sigma_{t+1} \backslash \mathtt{X} = \sigma'_{t+1} \backslash \mathtt{X}$ and $p_{t+1} = p'_{t+1}$. So again we have $C_2 \mathcal{R} C'_2$.

Equivalence: By $\mathcal{R}$, the program points of computation prefixes of π and π' are the same and thus π terminates if and only if π' terminates. We then need to show that the variable $\mathtt{Y}$ written by π is the same as $\mathtt{Y}'$ written by π'. By the side condition of the rewrite we know that $\mathtt{Y} \neq \mathtt{X}$. So given that both programs terminate at program point p_n with their prefixes related by $\mathcal{R}$, we know that at least $\sigma_n \setminus \mathtt{X} = \sigma'_n \setminus \mathtt{X}$. Therefore, $\sigma_n(\mathtt{Y}) = \sigma'_n(\mathtt{Y})$ and by Lemma 2 we can conclude that $[\![\pi]\!] = [\![\pi']\!]$.

5.2 Constant Folding

The constant folding rule is:

$$\begin{array}{l} x := v \Longrightarrow x := c \\ \text{if} \\ \quad \overleftarrow{A}(\neg def(v)\ W\ stmt(v := c) \wedge conlit(c)) \end{array}$$

Following Definition 7, for this rewrite to apply, the model checker must find a particular program point p and a substitution that maps x to a particular program variable $\mathtt{X}$, v to a particular program variable $\mathtt{V}$ and c to a particular constant $\mathtt{C}$. In this case we need to prove that an original program π and transformed program π' are equivalent.

In this case the relation $\mathcal{R}$ is the identity relation. That is, we wish to prove that for any length n, a computation prefix of π of length n is equal to a computation prefix of π' with the same length.

Base case: $(p_0, \sigma_0) = (p'_0, \sigma'_0)$ since $vars(\pi) = vars(\pi')$.

Step case: Suppose $\mathcal{R}$ (equality) holds between relations C_1 and C'_1 where:

$$C_1 = C'_1 = \pi, v \vdash (p_0, \sigma_0) \rightarrow \ldots \rightarrow (p_t, \sigma_t)$$

Also suppose that $C_1 \rightarrow C_2$ (by the semantics of π) and $C'_1 \rightarrow C'_2$ (by the semantics of π'). We wish to prove that $C_2 = C'_2$. The proof is split depending on whether $p = p_t$.

Suppose $p_t \neq p$. Now by Lemma 5 it follows that $C_2 = C'_2$.

Suppose $p_t = p$ then $I_{p_t} = (\mathtt{X} := \mathtt{V})$ and $I'_{p_t} = (\mathtt{X} := \mathtt{C})$. We know from the side condition and Lemma 3 that the path $p_t \leftarrow \ldots \leftarrow p_0$ is a maximal finite backwards path in the flow graph. The side condition states that

$$p_t \models_\theta \overleftarrow{A}(\neg def(v)\ W\ stmt(v := c) \wedge conlit(c))$$

for a substitution θ that maps v to $\mathtt{V}$ and c to $\mathtt{C}$. By Lemma 1 we know that there exists i such that $i \leq t$ and $I_{p_i} = (\mathtt{V} := \mathtt{C})$ and we know p_j does not define $\mathtt{V}$ for all $i < j < t$. Thus $\sigma_{i+1}(\mathtt{V}) = \mathtt{C}$ and by Lemma 8: $\sigma_t(\mathtt{V}) = \sigma_{i+1}(\mathtt{V}) = \mathtt{C}$. So the instruction `X := V` and `X := C` will set $\mathtt{X}$ to the same value. Therefore C_2 will be the same as C'_2.

Equivalence: Since $\mathcal{R}$ is the identity, the program points of computation prefixes of π and π' are the same, and thus π terminates if and only if π' terminates. Clearly if two terminating prefixes are equal they will have the same value in their final stores. So $[\![\pi]\!] = [\![\pi']\!]$ by Lemma 2.

5.3 Code motion/loop invariant hoisting

The code motion/loop invariant hoisting rule is:

$$\begin{array}{l} p : \mathtt{skip} \Longrightarrow x{:=}e \\ q : x{:=}e \Longrightarrow \mathtt{skip} \\ \text{if} \\ \quad p \models\ A(\neg use(x)\ W\ node(q)) \\ \quad q \models\ \neg use(x) \wedge \\ \qquad \overleftarrow{A}((\neg def(x) \vee node(q)) \wedge trans(e)\ W\ node(p)) \end{array}$$

Following Definition 7, for this rewrite to apply, the model checker must find particular program points p and q and a substitution that maps x to a particular program variable $\mathtt{X}$ and e to a particular program expression $\mathtt{E}$. In this case we need to prove that an original program π and transformed program π' are equivalent.

Definition 16 Suppose $C \in \mathcal{T}_{pfx}(\pi, v)$ and $C' \in \mathcal{T}_{pfx}(\pi', v)$ for some v such that

$$\begin{array}{lcl} C & = & \pi, v \vdash (p_0, \sigma_1) \rightarrow \ldots \rightarrow (p_t, \sigma_t) \\ C' & = & \pi', v \vdash (p'_0, \sigma'_1) \rightarrow \ldots \rightarrow (p'_{t'}, \sigma'_{t'}) \end{array}$$

We then define the $\mathcal{R}$ relation on computation prefixes as: $C\mathcal{R}C'$ if and only if $t = t'$, $p_i = p'_i$ for all $0 \leq i \leq t$ and one of the following cases holds:

1. $\sigma_t = \sigma'_t \ \wedge \ \forall i \ [0 \leq i < t \implies p_i \notin \{p, q\}]$
2. $\sigma_t = \sigma'_t \ \wedge \ \exists i \ [0 \leq i < t \wedge p_i = q \ \wedge \ \sigma_i = \sigma'_i \ \wedge \ \forall j \ (i < j < t \implies p_j \notin \{p, q\})]$
3. $\exists i \ [0 \leq i < t \wedge p_i = p \ \wedge \ (\sigma_t \setminus \mathtt{X} = \sigma'_t \setminus \mathtt{X}) \ \wedge \ (\sigma_i \setminus \mathtt{X} = \sigma'_i \setminus \mathtt{X}) \ \wedge \ \forall j \ (i < j < t \implies p_j \notin \{p, q\})]$

The notation $C\mathcal{R}_k C'$ will be used to indicate that $C\mathcal{R}C'$ holds by case k, as defined above.

Base case: $(p_0, \sigma_0) = (p'_0, \sigma'_0)$ since $vars(\pi) = vars(\pi')$. Also, case 1 of the relation holds trivially.

Step case: Suppose $C \rightarrow C_2$ and $C' \rightarrow C'_2$. Assuming that $C\mathcal{R}C'$ we need to show that $C_2\mathcal{R}C'_2$.

The proof is split up into cases depending on the label p_t and each of these is further split up, depending on the case for which $C\mathcal{R}C'$ holds.

(A) **Suppose** $p_t \notin \{p, q\}$, then $I_{p_t} = I'_{p_t}$.

(i) Suppose $C\mathcal{R}_k C'$ where $k = 1$ or 2. By assumption $\sigma_t = \sigma'_t$, implying $(p_{t+1}, \sigma_{t+1}) = (p'_{t+1}, \sigma'_{t+1})$ by Lemma 5. Thus $C_2\mathcal{R}_k C'_2$ holds.

(ii) Otherwise let $0 \leq i \leq t$ be given such that $C\mathcal{R}_3 C'$ holds. The side condition for p must be satisfied at p_i:

$$p_i \models A(\neg use(\mathtt{X}) \ W \ node(q)).$$

Since the control flow model describes all possible computation prefixes (Lemma 3), the same must hold for the computation prefix C. By assumption (A) and (ii) we know that $\forall j \ (i < j \leq t \implies p_j \neq q)$, so $p_j \models \neg use(\mathtt{X})$ must be satisfied for all $i < j \leq t$. In particular this holds for $j = t$. By assumption $\sigma_i \setminus \mathtt{X} = \sigma'_i \setminus \mathtt{X}$ so by Lemma 7 we conclude: $\sigma_{t+1} \setminus \mathtt{X} = \sigma'_{t+1} \setminus \mathtt{X}$ and $p_{t+1} = p'_{t+1}$. Thus $C_2\mathcal{R}_3 C'_2$ holds for i unchanged.

(B) **Suppose** $p_t = p$, then $I_{p_t} = (\mathtt{skip})$ and $I'_{p_t} = (\mathtt{X} := \mathtt{E})$. By the semantics and the induction assumption $p_{t+1} = p_t + 1 = p'_t + 1 = p'_{t+1}$. Also, $C_1\mathcal{R}C'_1$ implies that $\sigma_t \setminus \mathtt{X} = \sigma'_t \setminus \mathtt{X}$. Since the instructions only alter $\mathtt{X}$, we can conclude that $\sigma_{t+1} \setminus \mathtt{X} = \sigma'_{t+1} \setminus \mathtt{X}$. So $C_2\mathcal{R}_3 C'_2$ holds for $i = t$.

(C) **Otherwise** $p_t = q$, so $I_{p_t} = (\mathtt{X} := \mathtt{E})$ and $I'_{p_t} = \mathtt{skip}$. By the semantics and the induction assumption $p_{t+1} = p_t + 1 = p'_t + 1 = p'_{t+1}$.

(i) Suppose $C\mathcal{R}_1 C'$: By Lemma 1 the side condition $q \models \overleftarrow{A}((\neg def(x) \vee node(q)) \wedge trans(e) \ W \ node(p))$ in the rewrite rule implies that any maximal finite backwards path eventually ends up in a state p_k where $p_k = p$. This contradicts the assumption $C_1\mathcal{R}_1 C'_1$.

(ii) Suppose $C\mathcal{R}_2 C'$: By assumption there exists an i such that $p_i = q$. The side condition of q implies, by Lemma 1:

$$\begin{array}{l} \exists g \ [g \leq t \wedge p_g \models node(p) \ \wedge \\ \forall h \ (g < h \leq t \implies p_h \models (\neg def(\mathtt{X}) \vee node(q)) \wedge trans(\mathtt{E}))] \end{array}$$

Since by assumption $p_j \neq p$ for all $i \leq j \leq t$ it follows that $g < i$. Thus for all j with $i \leq j < t$, $p_j \models \neg def(\mathtt{X}) \ \vee \ node(q)$. Now the induction assumption says that $p_j \neq q$ for all $j, i < j < t$. This implies that $p_j \models \neg def(\mathtt{X})$ for all j with $i < j < t$. By Lemma 8 $\sigma'_t(\mathtt{X}) = \sigma'_{i+1}(\mathtt{X})$. Further, $p_j \models trans(\mathtt{E})$ for all j with $i \leq j < t$, and by Lemma 9: $[\![\mathtt{E}]\!]\sigma_t = [\![\mathtt{E}]\!]\sigma_i$. Therefore

$$\begin{array}{lcll} \sigma'_{t+1}(\mathtt{X}) & = & \sigma'_t(\mathtt{X}) & \text{(semantics of } I'_{p_t} = \mathtt{skip}) \\ & = & \sigma'_{i+1}(\mathtt{X}) & \text{(argument above)} \\ & = & \sigma_{i+1}(\mathtt{X}) & \text{(since } C\mathcal{R}_2 C' \implies \sigma_{i+1} = \sigma'_{i+1}) \\ & = & [\![\mathtt{E}]\!]\sigma_i & \text{(semantics of } I_{p_i} = (\mathtt{X} := \mathtt{E})) \\ & = & [\![\mathtt{E}]\!]\sigma_t & \text{(argument above)} \\ & = & \sigma_{t+1}(\mathtt{X}) & \text{(semantics of } I_{p_t} = (\mathtt{X} := \mathtt{E})) \end{array}$$

Since I_{p_t} only changes the variable $\mathtt{X}$ we can conclude that $\sigma_{t+1} = \sigma'_{t+1}$. Therefore $C_2\mathcal{R}_2 C'_2$.

(iii) Otherwise there exists an i such that $C\mathcal{R}_3 C'$ holds: We wish to show that $C_2\mathcal{R}C'_2$ holds by case 2 of $\mathcal{R}$ for $i = t$, i.e. $\sigma_{t+1} = \sigma'_{t+1}$.

First note that since the statements I_{p_t} and I'_{p_t} affect at most the variable $\mathtt{X}$, we can see that $\sigma_{t+1} \setminus \mathtt{X} = \sigma'_{t+1} \setminus \mathtt{X}$. Next we need to show that $\sigma_{t+1}(\mathtt{X}) = \sigma'_{t+1}(\mathtt{X})$.

By assumption there exists an i such that $p_i = p$. The side condition of q implies, by Lemma 1:

$$\begin{array}{l} \exists \alpha \leq t \ [p_\alpha \models node(p) \ \wedge \\ \forall \beta \ (\alpha < \beta \leq t \implies p_\beta \models (\neg def(\mathtt{X}) \vee node(q)) \wedge trans(\mathtt{E})] \end{array}$$

Since by assumption $p_j \neq p$ for all $i \leq j \leq t$ it follows that $\alpha \leq i$. Thus $p_j \models \neg def(\mathtt{X}) \ \vee \ node(q)$ for all $i < j < t$. Now the induction assumption says that $p_j \neq q$ for all $i < j < t$. This implies that $p_j \models \neg def(\mathtt{X})$ for all $i < j < t$. By Lemma 8 $\sigma'_t(\mathtt{X}) = \sigma'_{i+1}(\mathtt{X})$. Also, $p_j \models trans(\mathtt{E})$ for all $i < j < t$. By Lemma 9: $[\![\mathtt{E}]\!]\sigma_t = [\![\mathtt{E}]\!]\sigma_{i+1}$.

Using the above observations it can be shown that $\mathtt{X}$ maps to the same value in σ_{t+1} and σ'_{t+1}:

$$\begin{array}{lcll} \sigma_{t+1}(\mathtt{X}) & = & [\![\mathtt{E}]\!]\sigma_t & \text{(semantics of } I_{p_t} = (\mathtt{X} := \mathtt{E})) \\ & = & [\![\mathtt{E}]\!]\sigma_{i+1} & \text{(argument above)} \\ & = & [\![\mathtt{E}]\!]\sigma_i & \text{(semantics of } I_{p_i} = \mathtt{skip}) \\ & = & [\![\mathtt{E}]\!](\sigma_i \setminus \mathtt{X}) & \text{(since } \mathtt{X} \notin vars(\mathtt{E})) \\ & = & [\![\mathtt{E}]\!](\sigma'_i \setminus \mathtt{X}) & \text{(by } C\mathcal{R}_3 C', \sigma_i \setminus \mathtt{X} = \sigma'_i \setminus \mathtt{X}) \\ & = & [\![\mathtt{E}]\!](\sigma'_i) & \text{(since } \mathtt{X} \notin vars(\mathtt{E})) \\ & = & \sigma'_{i+1}(\mathtt{X}) & \text{(semantics of } I'_{p_i} = (\mathtt{X}{:=}\mathtt{E})) \\ & = & \sigma'_t(\mathtt{X}) & \text{(argument above)} \\ & = & \sigma'_{t+1}(\mathtt{X}) & \text{(semantics of } I'_{p_t} = \mathtt{skip}) \end{array}$$

Thus $\sigma_{t+1} = \sigma'_{t+1}$, so case 2 holds for $i = t$.

By induction we conclude that $C\mathcal{R}C'$ for any two computation prefixes $C \in \mathcal{T}_{pfx}(\pi)$ and $C' \in \mathcal{T}_{pfx}(\pi')$ of the same length on the same input v.

Equivalence: Again, by $\mathcal{R}$ the program points of computation prefixes of π and π' are the same and thus π terminates if and only if π' terminates. Now suppose π and π' both terminate on input v. Consider the store at the `write` statement: $I_m = I'_m = (\texttt{write Y})$. Suppose $C\mathcal{R}C'$ for $t = m$ by case 3, then by the side condition there exists an $i > m$ such that $p_i = q$ (since π terminates). But by the semantics $p_i = m \neq q$ for all $i > t$ leading to a contradiction. Thus either case 1 or 2 must hold. In either case $\sigma_t = \sigma'_t$, implying $\sigma_t(\texttt{Y}) = \sigma'_t(\texttt{Y})$. It follows by Lemma 2 that the transformation is semantics preserving: $[\![\pi]\!] = [\![\pi']\!]$.

6. DISCUSSION AND FUTURE WORK

In this paper we have described a framework in which temporal logic plays a crucial role in the proofs of correctness of classical optimizing transformations performed by a compiler. In this framework transformations are specified as rewrite rules with side conditions that are written as temporal logic formulas.

To prove the correctness of the transformations we had to show that if a transformation is applied it does not change the semantics of the program – that is, $[\![\pi]\!] = [\![\pi']\!]$. The only creative part of the proof is finding the relation $\mathcal{R}$, but this relation is often closely related to the temporal logic side conditions of the transformation. The remainder of the proof is straightforward. It is either routine, as when showing that the program states of computation prefixes of π and π' are the same before encountering the transformed program point. Otherwise it deals directly with the program transformation point. The proof of these cases is dramatically simplified since we can assume that the temporal logic side condition holds (otherwise the transformation would not have happened), and this assumption leads almost immediately to the proof of the case.

While the proofs presented here have been done by hand, the nature of the proofs seems well suited to (semi-) automated theorem proving. The creative step in the proof is to create the relation we wish to prove inductively. The rest of the proofs tend to involve mechanically performing case splits and applying a small set of lemmas. However, even the "creative" step of providing the relation seems to be closely related with the temporal logic side conditions. An interesting direction of further work would be to discover if the relation could be mechanically created from the side conditions.

The programming language on which these transformations have been applied is admittedly very simple. There are very few types of statements and it does not include necessary language features like exceptions and procedures. Limiting the number of types of statements reduces the number of cases in the proofs and this simplifies their presentation, but adding additional statements does not affect the applicability of our method. Exceptions and procedures would however, require changes to the control flow model and the transition systems used in the proof. The specification of the transformations however does not dramatically change. A follow-up paper describing the required adjustments is in preparation.

The language we have treated is rather like a traditional compiler's "intermediate language". We anticipate that our method could be used to validate a great many traditional optimizing compiler transformations, e.g., many found in [2] and [20].

This work is part of a larger project to study declarative methods of specifying optimizations and means of automatically generating optimizers from these specifications. Here, the specifications of optimizing transformations are rewrite rules with temporal logic side conditions that are atomically implemented by a graph rewriting system and model checker [18].

Acknowledgements

The Oxford authors would like to thank the Programming Tools Group in Oxford and in particular Oege de Moor for fruitful discussions about this work and Microsoft Research for its support of this research as part of the Intentional Programming project.

7. REFERENCES

[1] S. Abramsky and C. Hankin. *Abstract Interpretation of Declarative Languages*. Ellis-Horwood, 1987.

[2] A.V. Aho and R. Sethi and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[3] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[4] R. Cleaveland and D. Jackson. *Proceedings of First ACM SIGPLAN Workshop on Automated Analysis of Software*. Paris, France, January 1997.

[5] P. Cousot and R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fix-points. In *Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, January 1977*, pp. 238–252, New York: ACM, 1977.

[6] P. Cousot, Semantic foundations of program analysis, in S.S. Muchnick and N.D. Jones (eds.), *Program Flow Analysis: Theory and Applications*, chapter 10, pp. 303–342, Englewood Cliffs, NJ: Prentice Hall, 1981.

[7] C.C. Frederiksen. *Correctness of classical compiler optimizations using CTL.* Unpublished TOPPS report, University of Copenhagen, 2001. `http://www.diku.dk/research-groups/topps/bibliography/2001.html#D-443`

[8] Th. Hafer and W. Thomas. Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree. In *Automata, Languages and Programming Proceedings, ICALP'87*, volume 267 of *Lecture Notes in Computer Science*, pages 267–279. Springer-Verlag, 1987.

[9] K. Havelund. *Stepwise Development of a Denotational Stack Semantics.* M.Sc. thesis, University of Copenhagen, 1984.

[10] M. Hecht. *Flow analysis of computer programs.* North-Holland, 1977.

[11] N.D. Jones (ed.), *Semantics-Directed Compiler Generation.* volume 94 of *Lecture Notes in Computer Science*, Springer-Verlag, 1980.

[12] N.D. Jones. Semantique: Semantic-Based Program Manipulation Techniques. In *Bulletin European Association for Theoretical Computer Science* 39:74-83, 1989.

[13] N.D. Jones and F. Nielson. Abstract Interpretation: a Semantics-Based Tool for Program Analysis.. In *Handbook of Logic in Computer Science,* edited by S. Abramsky, D, Gabbay, T. Maibaum, pages 527–629, Oxford University Press, 1994.

[14] J. Knoop and O. Rüthing and B. Steffen. Optimal Code Motion: Theory and Practice. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1117–1155, 1994.

[15] D. Kozen, *Efficient code certification.* Technical Report 98-1661, Computer Science Department, Cornell University, January, 1998.

[16] D. Kozen and M. Patron. Certification of compiler optimizations using Kleene algebra with tests. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, eds, *Proceedings of the 1st International Conference on Computational Logic (CL2000)*, Lecture Notes in Artificial Intelligence volume 1861, Springer-Verlag, London, July 2000, pp. 568–582.

[17] O. Kupferman and A. Pnueli. Once and for all. In *Proc. 10th IEEE Symposium on Logic in Computer Science*, pages 25–35, San Diego, June 1995.

[18] D. Lacey and O. de Moor. Imperative program transformation by rewriting. In *Proc. 10th International Conf. on Compiler Construction*, volume 1113 of *Lecture Notes in Computer Science*, pages 52–68. Springer-Verlag, 2001.

[19] R. Milne and C. Strachey *A Theory of Programming Language Semantics.* Chapman and Hall, 1976.

[20] S.S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[21] S.S. Muchnick and N.D. Jones (eds.) *Program Flow Analysis: Theory and Applications.* Englewood Cliffs, NJ: Prentice Hall, 1981.

[22] G. Necula, Proof-carrying code. In *24th ACM Symposium on Principles of Programming Languages, Paris, France, January 1997*, pp. 106–119, New York: ACM, 1997.

[23] F. Nielson. A Denotational Framework for Data Flow Analysis. *Acta Informatica*, 18:265–287, 1982.

[24] F. Nielson. *Semantic Foundations of Data Flow Analysis.* M.Sc. thesis, Aarhus University, DAIMI PB-131, 1981.

[25] F. Nielson, H.R. Nielson and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[26] S. S. Pinter and P. Wolper. A temporal logic for reasoning about partially ordered computations. In *Proc. 3rd ACM Symposium on Principles of Distributed Computing*, pages 28–37, 1984.

[27] A. Podelski, B. Steffen, M. Vardi. *Schloss Ringberg Seminar: Model Checking and Program Analysis.* Workshop, February 2000, Bavaria.

[28] T. Rus and E. Van Wyk. Using model checking in a parallelizing compiler. *Parallel Processing Letters*, 8(4):459-471, 1998.

[29] D.A. Schmidt. Data-flow analysis is model checking of abstract interpretations. In *Proc. of 25th ACM Symposium on Principles of Programming Languages*, ACM, 1998.

[30] D.A. Schmidt, B. Steffen. Program analysis *as* model checking of abstract interpretations. In *Proc. of 5th Static Analysis Symposium*, G. Levi. ed., Pisa, volume 1503 of *Lecture Notes in Computer Science*, Springer-Verlag, 1998.

[31] Paul A. Steckler and Mitchell Wand. Lightweight Closure Conversion. *ACM Transactions on Programming Languages and Systems*, ACM, 19(1):48–86, January 1997.

[32] B. Steffen, A. Claßen, M. Klein, J. Knoop, T. Margaria. The Fixpoint Analysis Machine. In *Proc. of the 6th International Conference on Concurrency Theory (CONCUR'95)*, J. Lee, S. Smolka eds., Philadelphia, Pennsylvania (USA), volume 962 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 72–87, 1995.

[33] G. Winskel. *The Formal Semantics of Programming Languages.* Boston, MA: the MIT Press, 1993.

[34] P. Wolper. On the relation of programs and computations to models of temporal logic. In *Proc. Temporal Logic in Specification*, volume 398 of *Lecture Notes in Computer Science*, pages 75–123. Springer-Verlag, 1987.