
Higher-Order and Symbolic Computation, 17, 173–206, 2004

Compiler Optimization Correctness by Temporal Logic*

DAVID LACEY david.lacey@dcs.warwick.ac.uk
University of Warwick

NEIL D. JONES neil@diku.dk
University of Copenhagen

ERIC VAN WYK evw@cs.umn.edu
University of Minnesota

CARL CHRISTIAN FREDERIKSEN carl@lyon.is.s.u-tokyo.ac.jp
University of Tokyo

Abstract. Rewrite rules with side conditions can elegantly express many classical compiler optimizations for imperative programming languages. In this paper, programs are written in an intermediate language and transformation-enabling side conditions are specified in a temporal logic suitable for describing program data flow.

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. A transformation is correct if whenever it can be applied to a program, the original and transformed programs are semantically equivalent, i.e., they compute the same input-output function. The proofs of semantic equivalence inductively show that a transformation-specific bisimulation relation holds between the original and transformed program computations.

Keywords: compiler verification, optimizing compilers, temporal logic, model checking

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 *formulae 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, an *induction* relation

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

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 v; and vice versa.

Compiler optimizing transformations. A great many program transformations are done by optimizing compilers; an exhaustive catalog may be found in Muchnick [28]. 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.

1.1. Semantics-based program manipulation

Much has happened in this field since the path-breaking 1977 Cousot and Cousot paper [7] and 1980 conference [15]. The field of "abstract interpretation" [1, 6, 7, 17, 31, 32] arose as a mainly European, theory-based counterpart to the well-developed more pragmatic North American approach to program analysis [2, 14, 29]. The goal of semantics-based program manipulation ([16] 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 is that 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.

1.2. 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., Winskel [46]) typically present many example proofs of equivalences between program fragments. However most of these are small (excepting the monumental and indigestible [27]), 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 a program's statement ordering, 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 [30] has an unpublished chapter proving correctness of "constant folding" perhaps omitted from the journal paper [31], because of the complexity of its development and proof. Havelund's thesis [13] 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 [27]).

Cousot provides a general framework for designing program transformations whose analyses are abstract interpretations [9], this framework is given at a higher level of generality than the proofs given in this paper, though in some ways the proofs could be seen as fitting into that framework. Some transformation correctness proofs have been made for functional languages, especially the sophisticated work by Wand and colleagues, for example [39], using "logical relations." These methods are mathematically rather more sophisticated than those of this paper, which seem more appropriate for traditional intermediate-code optimizations.

1.3. Model checking and program analysis

This situation has improved with the advent of model checking approaches to program analysis [5, 34–37, 42]. Work by Steffen and Schmidt [36, 37, 40, 41] 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 et al. [18] 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.

1.4. Kleene algebra with tests

Another approach to correctness of compiler optimizations is presented by Kozen and Patron [19]. 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 programs are equal under the algebraic laws of KAT. In many instances these terms are not ground, that is, they contain variables, and thus the reasoning could be applied to more general programs. One has to note that the paper sets out with a different perspective than ours on program transformation. It is geared towards establishing a framework where one can formally reason about program manipulations specified as KAT equalities that imply semantic equivalence. Unfortunately the results in the paper are

not directly applicable to compilers since no automatic method is given for applying the optimizations described.

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 compilers.

1.5. Model checking and program transformation

In this paper we give a formalism (essentially it is a subset of Lacey and de Moor [23]) 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 the paper [11], in the longer unpublished report [10], and in the thesis [22].

The approach is similar to other approaches to specifying transformations and analyses together, for example in [3, 44, 45]. This paper shows how our method of specification is particularly useful when considering correctness. A recent development: Technical report [26] was directly inspired by Lacey et al. [24]. In essence it allows a small subset of our temporal program properties to be specified, but it has been more fully automated than this work, building on a platform described in Lerner et al. [25]. One may regard their work as a "proof of concept and relevance" of our approach to compiler transformation correctness proofs.

Many optimizing transformations can be elegantly expressed using rewrite rules of form: $I \Rightarrow I'$ 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 [23]: If the current program π contains an instruction of the form I at some control point p, and if flow condition ϕ is satisfied at p, then replace I by I' creating the transformed program π'. The purpose of this paper is to show how such transformations may be proven correct. Correctness means that if a transformation changes any program π into a program π', then the two programs are semantically equivalent, i.e., they compute the same input-output function. Section 4 shows how semantic equivalence can be established between π and π'.

2. Programs and their semantics

In this section we provide fundamental definitions used in our representations of programs. In Section 2.1 we introduce a simple imperative programming language that we use to demonstrate program transformations and their proofs of correctness. Section 2.2 defines the semantics of the language, including the notion of semantic equivalence which is key to defining the correctness of transformations.

2.1. *A simple programming language*

Definition 1. A *program* π has the form:

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

where $I_1, \ldots, I_{m-1}$ are *instructions*. By convention every instruction has a unique *label* in $Nodes_\pi = \{0, 1, 2, \ldots, m\}$, with 0 labeling the initial `read x` and m labeling the final `write y`. 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:

$$\begin{aligned}
\mathit{Inst} \ni I &::= \texttt{skip} \mid \texttt{X := E} \mid \\
&\quad\ \texttt{if X goto}\ n\ \texttt{else}\ n' \\
\mathit{Expr} \ni \texttt{E} &::= \texttt{X} \mid \texttt{O E...E} \\
\mathit{Op} \ni \texttt{O} &::= \text{various unspecified operators } o, \\
&\quad\ \text{each with } \mathit{arity}(o) \geq 0 \\
\mathit{Var} \ni \texttt{X} &::= \texttt{x} \mid \texttt{y} \mid \texttt{z} \mid \ldots \\
\mathit{Label} \ni n, n' &::= 1 \mid 2 \ldots \mid m
\end{aligned}$$

Program semantics is as expected and are formally defined below in Section 2.2. Figure 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.

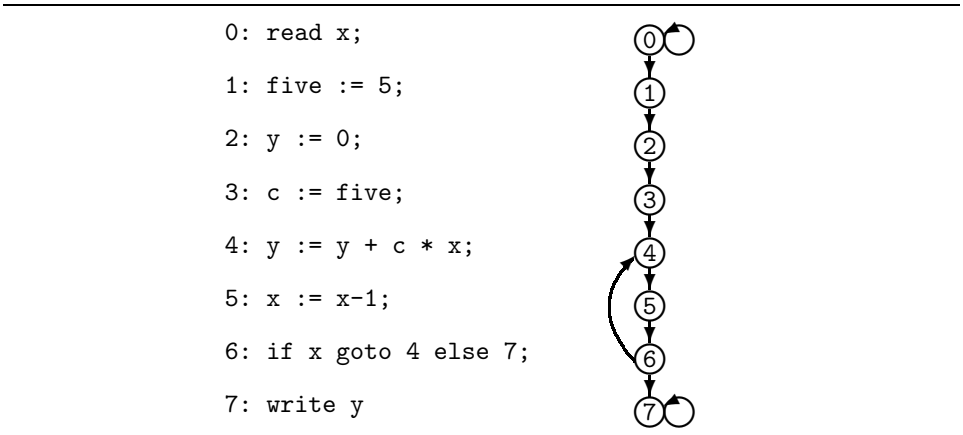

Figure 1. Example program and control flow graph.

2.2. *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 2 (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 $[\![o]\!]_{\text{op}} : \mathit{Value}^n \to \mathit{Value}$. Note that $[\![o]\!]_{\text{op}} \in \mathit{Value}$ if $n = 0$.

  We assume that these functions are total. In Section 6.3 we discuss the issues raised by partial functions and exceptions, such as division by zero, and describe how our proof method is general enough to accommodate them.

Definition 3 (Expression evaluation). A *store* is a function $\sigma \in \mathit{Store} = \mathit{Var} \rightharpoonup \mathit{Value}$. *Expression evaluation* $[\![\mathit{Expr}]\!]_{\text{exp}} : \mathit{Store} \to \mathit{Value}$ is defined by:

$$[\![\mathtt{X}]\!]_{\text{exp}}\sigma = \sigma(\mathtt{X})$$
$$[\![\mathtt{O}\ \mathtt{E}_1 \ldots \mathtt{E}_n]\!]_{\text{exp}}\sigma = [\![\mathtt{O}]\!]_{\text{op}}([\![\mathtt{E}_1]\!]_{\text{exp}}\sigma, \ldots, [\![\mathtt{E}_n]\!]_{\text{exp}}\sigma)$$

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 $v \in \mathit{Value}$.

Definition 4 (Semantics). At any point in its computation, program π will be in a *state* of the form $s = (p, \sigma) \in \mathit{State}_\pi = \mathit{Nodes}_\pi \times \mathit{Store}$. 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. A *final state* is one that has the form (m, σ), where m is the label of the last instruction in 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 $(p, \sigma) \to (p + 1, \sigma[\mathtt{X} \mapsto [\![\mathtt{E}]\!]_{\text{exp}}\sigma])$.
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 $(p, \sigma) \to (p'', \sigma)$.

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 $\mathcal{T}_{\text{run}}$.

Definition 5. 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*.

Definition 6. The *execution transition system* for program π and input $v \in \mathit{Value}$ is by definition

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

where $s_1 \rightarrow s_2$ is as in Definition 4.

Definition 7. The *semantic function* $[\![\pi]\!] : \mathit{Value} \rightharpoonup \mathit{Value}$ is the partial function defined by:

$$[\![\pi]\!](v) = \sigma(\mathrm{y})$$

iff there exists a finite sequence from the initial state to a final state

$$\mathit{In}(v) = s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_t = (m, \sigma)$$

In order to reason about the computational history of program executions we will also introduce the notion of computational prefix, and a corresponding transition system $\mathcal{T}_{\text{pfx}}$ (both defined in Section 4).

3. Analysis and transformation

In Section 3.1 we describe the control flow graph representation of programs that serves as the model over which the temporal logic formulae in the rewrite rules are checked. The temporal logic CTL with free variables is presented in Section 3.2. The rewriting rules are defined in Sections 3.3 and 3.5 provides the specifications for the dead code elimination and constant folding transformations.

3.1. 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 type of transition system and it also an example of a model (as used in model checking). Models are transition systems where each state is labeled with certain information:

Definition 8. A *model* (or *Kripke structure* [20]) is a triple $\mathcal{M} = (S, \rightarrow, L)$ (and a set of propositions P) 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 graph for program π is a system whose states are program points and whose transitions from one program point to another could occur consecutively in the execution.

Definition 9. $\mathcal{T}_{\text{cf}}(\pi) = (\mathit{Nodes}_\pi, \rightarrow_{\text{cf}})$ is the *control flow graph for* π, where the (total) relation $\rightarrow_{\text{cf}}$ is defined by $n_1 \rightarrow_{\text{cf}} n_2$ if and only if

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

Note that the self-loop transitions on the `read` and `write` nodes do not have corresponding transitions in the execution transition system $\mathcal{T}_{\text{run}}(\pi, v)$. They exist in $\mathcal{T}_{\text{cf}}(\pi)$ only to satisfy the totality requirement (in both arguments) of $\rightarrow_{\text{cf}}$ imposed by CTL-FV, see Section 3.2.

We will sometimes drop the *cf* subscript when it is clear that the control flow transition relation is being used.

We set up a control flow model by labeling the states of the control flow graph (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 shows the control flow model for the program in Figure 1 in which node 2, whose instruction `y := 0` is labeled by the propositions *node*(2), *stmt*(`y := 0`), *def*(`y`), and *conlit*(`0`).

$$
\begin{aligned}
\mathit{Nodes}_\pi &= \{0, 1, 2, 3, 4, 5, 6, 7\}\\
\rightarrow_{\text{cf}} &= \{0 \rightarrow_{\text{cf}} 0, 0 \rightarrow_{\text{cf}} 1, 1 \rightarrow 2, 2 \rightarrow_{\text{cf}} 3, 3 \rightarrow_{\text{cf}} 4,\\
&\quad\; 4 \rightarrow_{\text{cf}} 5, 5 \rightarrow_{\text{cf}} 6, 6 \rightarrow_{\text{cf}} 7, 6 \rightarrow_{\text{cf}} 4, 7 \rightarrow_{\text{cf}} 7\}
\end{aligned}
$$

$$
\begin{aligned}
L_\pi(0) &= \{\mathit{node}(0), \mathit{stmt}(\texttt{read x}), \mathit{def}(\texttt{x})\} &&\cup \mathit{trans}_0\\
L_\pi(1) &= \{\mathit{node}(1), \mathit{stmt}(\texttt{five := 5}), \mathit{def}(\texttt{five}), \mathit{conlit}(\texttt{5})\} &&\cup \mathit{trans}_1\\
L_\pi(2) &= \{\mathit{node}(2), \mathit{stmt}(\texttt{y := 0}), \mathit{def}(\texttt{y}), \mathit{conlit}(\texttt{0})\} &&\cup \mathit{trans}_2\\
L_\pi(3) &= \{\mathit{node}(3), \mathit{stmt}(\texttt{c := five}), \mathit{def}(\texttt{c}), \mathit{use}(\texttt{five})\} &&\cup \mathit{trans}_3\\
L_\pi(4) &= \{\mathit{node}(4), \mathit{stmt}(\texttt{y := y + c * x}), \mathit{def}(\texttt{y}),\\
&\qquad\qquad \mathit{use}(\texttt{y}), \mathit{use}(\texttt{c}), \mathit{use}(\texttt{x})\} &&\cup \mathit{trans}_4\\
L_\pi(5) &= \{\mathit{node}(5), \mathit{stmt}(\texttt{x := x - 1}), \mathit{def}(\texttt{x}), \mathit{use}(\texttt{x}),\\
&\qquad\qquad \mathit{conlit}(\texttt{1})\} &&\cup \mathit{trans}_5\\
L_\pi(6) &= \{\mathit{node}(6), \mathit{stmt}(\texttt{if x goto 4 else 7}), \mathit{use}(\texttt{x})\} &&\cup \mathit{trans}_6\\
L_\pi(7) &= \{\mathit{node}(7), \mathit{stmt}(\texttt{write y}), \mathit{use}(\texttt{y})\} &&\cup \mathit{trans}_7
\end{aligned}
$$

Figure 2. Control flow model for the example program.

Definition 10. The *control flow model for program* π is defined as $\mathcal{M}_{\text{cf}}(\pi) = (Nodes_\pi, \rightarrow_{\text{cf}}, L_\pi)$ where $(Nodes_\pi, \rightarrow_{\text{cf}})$ are as in Definition 9, and $L_\pi(n)$ is defined as follows for $n \in Nodes_\pi$:

$$
\begin{aligned}
L_\pi(n) = {} & \{stmt(I_n) \mid 0 \le n \le m\} \\
& \cup \{node(\texttt{n})\} \\
& \cup \{def(\texttt{X}) \mid I_n \text{ has form } \texttt{X:=E} \text{ or } \texttt{read X}\} \\
& \cup \{use(\texttt{X}) \mid I_n \text{ form: } \texttt{Y:=E} \text{ with } \texttt{X} \text{ in } \texttt{E}, \text{ or} \\
& \qquad\qquad I_n = \texttt{if X goto } p \texttt{ else } p'\} \\
& \cup \{use(\texttt{Y}) \mid n = m \text{ and } I_n = \texttt{write Y}\} \\
& \cup \{conlit(\texttt{O}) \mid \texttt{O} \text{ is a constant in } I_n, \text{ i.e., an} \\
& \qquad\qquad \text{operator with } arity(\texttt{O}) = 0\} \\
& \cup trans_n
\end{aligned}
$$

where

$$
trans_n = \{trans(\texttt{E}) \mid \texttt{E} \text{ is an expression in } \pi \text{ and } I_n \text{ is not of form: } \texttt{X:=E}' \text{ or } \texttt{read X} \text{ with } \texttt{X} \text{ in } vars(\texttt{E})\}
$$

The predicates $stmt(I)$, $def(\texttt{X})$, $use(\texttt{X})$, $conlit(\texttt{O})$, $trans(\texttt{E})$ are the building blocks for the conditions that specify when optimizing transformations can be safely applied. These conditions are specified as CTL-FV formulae. Note that $trans(\texttt{E})$ is the set of *transparent* expressions on a node: the expressions whose value is not changed by the node.

3.2. CTL with free variables

A *path over* $\rightarrow$ is an infinite sequence of nodes $n_0 \rightarrow n_1 \rightarrow \ldots$ such that $\forall i \ge 0 : n_i \rightarrow n_{i+1}$. A *backwards path* is an path over the inverse of $\rightarrow$ (written as $\rightarrow^\circ$) and is written as either $n_0 \rightarrow^\circ n_1 \rightarrow^\circ \ldots$ or $n_0 \leftarrow n_1 \leftarrow \ldots$.

The temporal logic CTL-FV used in specifying transformation conditions is in two respects a generalization of CTL [4]. First, as is common, the existential and universal 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 $\overleftarrow{A}$. We consider a branching notion of past which is infinite, as in *POTL* [33, 47] and not the finite branching past in CTL_{bp} [21]. A branching past is more appropriate here than the linear past in *PCTL*$*$ [12] which can also be used to augment 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 $stmt(x := e)$ where *stmt* is from the set Pr of predicate names, 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.

A CTL-FV formula is either a *state formula* ϕ or a *path formula* ψ, generated by the following grammar with non-terminals ϕ, ψ, terminals *true*, *false*, $pr \in Pr$ and free variables $x_1, \ldots, x_n$, start symbol ϕ and the productions:

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

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 $n \models_\theta \phi$ is defined in Figure 3 and will hold for any θ such that $n \models \theta(\phi)$. Here, $\theta(\phi)$ is a standard CTL formula with no free variables and $\models$ is as traditionally defined in Clarke et al. [4]. Thus, the standard abbreviations from CTL, e.g. $F\phi \equiv true\ U\ \phi$, $G\phi \equiv \neg F \neg \phi$ and $\phi_1\ W\ \phi_2 \equiv (\phi_1\ U\ \phi_2) \vee G\phi_1$, hold in CTL-FV as well.

State Formulae:

$n \models_\theta true$	iff	*true*
$n \models_\theta false$	iff	*false*
$n \models_\theta pr(x_1, \ldots, x_n)$	iff	$pr(\theta x_1, \ldots, \theta x_n) \in L_\pi(n)$
$n \models_\theta \neg\phi$	iff	not $n \models_\theta \phi$
$n \models_\theta \phi_1 \wedge \phi_2$	iff	$n \models_\theta \phi_1$ and $n \models_\theta \phi_2$
$n \models_\theta \phi_1 \vee \phi_2$	iff	$n \models_\theta \phi_1$ or $n \models_\theta \phi_2$

$n \models_\theta E\ \psi$ iff $\exists\, path\ (n = n_0 \to n_1 \to n_2 \ldots)$: $(n_i)_{i \geq 0} \models_\theta \psi$
$n \models_\theta A\ \psi$ iff $\forall\, path\ (n = n_0 \to n_1 \to n_2 \ldots)$: $(n_i)_{i \geq 0} \models_\theta \psi$
$n \models_\theta \overleftarrow{E}\ \psi$ iff $\exists\, path\ (\ldots n_2 \to n_1 \to n_0 = n)$: $(n_i)_{i \geq 0} \models_\theta \psi$
$n \models_\theta \overleftarrow{A}\ \psi$ iff $\forall\, path\ (\ldots n_2 \to n_1 \to n_0 = n)$: $(n_i)_{i \geq 0} \models_\theta \psi$

Path Formulae:

$(n_i)_{i \geq 0} \models_\theta X\ \phi$ iff $n_1 \models_\theta \phi$

$(n_i)_{i \geq 0} \models_\theta \phi_1\ U\ \phi_2$ iff
$\exists k \geq 0 : [n_k \models_\theta \phi_2 \wedge \forall i : [0 \leq i < k$ implies $n_i \models_\theta \phi_1]]$

$(n_i)_{i \geq 0} \models_\theta \phi_1\ W\ \phi_2$ iff
$(\exists k \geq 0 : [n_k \models_\theta \phi_2$ and $\forall i : 0 \leq i < k \Rightarrow n_i \models_\theta \phi_1])$
or $(\forall k \geq 0 : [n_k \models_\theta \phi_1])$

Figure 3. CTL-FV satisfaction relation.

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 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.)

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

Of particular interest when analyzing the control flow model is the universal weak until operator (AW). Its use ensures that loops in the control flow model do not invalidate optimization opportunities where they can be safely applied, where as AU would.

3.3. Rewriting

Definition 11. A *rewrite rule* has form: $I \Rightarrow I'$ if ϕ, where I, I' are instructions built from program and CTL variables, and ϕ is a CTL-FV formula. By definition *Rewrite*(π, π', n, $I \Rightarrow I'$ if ϕ) is true if and only if for some substitution θ, the following holds:

$n \models_\theta$ *stmt*(I) $\wedge \phi$
$\pi =$ `read x;` I_1`;` ... I_n`;` ... I_{m-1}`; write y,`
where $I_n = \theta(I)$, and
$\pi' =$ `read x;` I_1`;` ... $\theta(I')$`;` ... 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:

$$n : I_1 \Rightarrow I_1'$$
$$m : I_2 \Rightarrow I_2'$$

if

$$n \models \phi_1$$
$$m \models \phi_2$$

The operational interpretation of this is that we find a substitution θ that satisfies both $n \models_\theta$ *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 at places n and m.

3.4. Computational aspects

We discuss computational aspects only briefly; more can be found in Lacey and de Moor [23] and related papers.

Model checking with respect to $I \Rightarrow I'$ if ϕ 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 immediately dead assignments can be 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 formulae times could be higher, since $|\mathcal{T}|$ depends on the size of labelling function $L : Nodes_\pi \rightarrow 2^{AP}$ as in Definition 10. For each node n, $L_\pi(n)$ can be found in time proportional at most to the size of the instruction I_n, with one exception: Propositions *trans*(E) can require time and space proportional to the size of π at each node n. For greater efficiency these can be treated specially, maintaining a single global data structure for the transparency relation.

Experience from Lacey and de Moor [23] 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.

3.5. 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).

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. This is convenient since it means the original and transformed programs have labels in a 1-1 correspondence. (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, other transformations (especially automated ones) often enable these optimizations.

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:

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

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. We can thus express dead code elimination as a rewrite rule with a side condition:

$$x := e \;\Rightarrow\; \texttt{skip}$$

if

$$AX\ A(\ \neg use(x)\ W\ def(x) \wedge \neg use(x)).$$

Since we do not care whether x is used at the current node, we skip past it with the AX operator. After this point we stipulate that x is never used again or not used until it is redefined (when $def(x)$ holds).

Constant folding. A weak form of constant folding is a transformation to replace a variable reference with a constant value:

$$x := y \;\Rightarrow\; x := c.$$

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[1]:

$$x := y \;\Rightarrow\; x := c$$

if

$$\overleftarrow{A}\,(\neg def(y) \wedge \neg stmt(\texttt{read x})\ W\ stmt(y := c) \wedge conlit(c))$$

The clause $\neg stmt(\texttt{read x})$ ensures that all paths from the entry `read` instruction to the instruction $x := y$ contain an instruction $x := c$.

Code motion/loop invariant hoisting. First, an example program, where the statement `x:=a+b` may be lifted from label 3 to label 1:

```
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;
```

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

$$p : \texttt{skip} \Rightarrow x := e$$
$$q : x := e \Rightarrow \texttt{skip}$$

if

$$p \models A(\neg use(x)\ W\ node(q))$$
$$q \models \neg use(x) \wedge \overleftarrow{A}\,((\neg def(x) \vee node(q)) \wedge trans(e) \wedge \neg stmt(\texttt{read x})\ W\ node(p))$$

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 remains 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.

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 not all paths from p may eventually 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. (In general the transformation by itself could slow down the computation, as is the case in our illustrating example, since there is no need to compute the expression if the expression is not needed; but this is not our point.)

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.

4. A method for showing semantic equivalence

Our method proves that the transformed program's computations are *bisimilar* with those of the original.

Definition 12. A *bisimulation* between transition systems $\mathcal{T} = (\mathcal{C}, \rightarrow)$ and $\mathcal{T}' = (\mathcal{C}', \rightarrow')$ is a relation $\mathcal{R} \subseteq \mathcal{C} \times \mathcal{C}'$ such that if $s \in \mathcal{C}$, $s' \in \mathcal{C}'$ and $s\mathcal{R}s'$ then

1. $s \rightarrow s_1$ implies $s' \rightarrow s'_1$ for some s'_1 with $s_1 \mathcal{R} s'_1$
2. $s' \rightarrow s'_1$ implies $s \rightarrow s_1$ for some s_1 with $s_1 \mathcal{R} s'_1$

Transformation correctness. For each rewrite rule $I \Rightarrow I'$ if ϕ we need to show that if $Rewrite(\pi, \pi', p, I \Rightarrow I' \text{ if } \phi)$, i.e., if π is transformed into π' then $[\![\pi]\!] = [\![\pi']\!]$, meaning that for any input v, program π has a terminating computation $In(v) \rightarrow^* (m, \sigma)$ if and only if program π' has a terminating computation $In(v) \rightarrow^* (m', \sigma')$ with $\sigma(\mathrm{y}) = \sigma'(\mathrm{y})$. The problem now is how to link the *temporal* property ϕ, which concerns "futures" and "pasts", to the transformation $I \Rightarrow I'$.

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

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

Some informal remarks. Suppose we have model checked $p \models_\theta \phi$ on program π's control flow graph. If ϕ contains only "past" operators, then the resulting substitutions also describe places in the computation prefix C where ϕ is true. 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 define a *prefix transition system* $\mathcal{T}_{\text{pfx}}(\pi, v)$ so $C \to C_1 \in \mathcal{T}_{\text{pfx}}(\pi, v)$ if and only if C_1 is identical to C, but with one additional state:

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

Now reasoning that involves futures can be done by ordinary induction: assuming $C\mathcal{R}C'$, show $C \to C_1$ implies $C' \to C'_1$ for a C'_1 with $C_1\mathcal{R}C'_1$, and $C' \to C'_1$ implies $C \to C_1$ for a 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 a sequence (finite or infinite)

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

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

A *terminating* computation prefix is one that reaches the `read y` instruction.

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

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

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

$$C_1 = \pi, v \vdash s_0 \to s_1 \to \ldots \to s_t,$$
$$C_2 = \pi, v \vdash s_0 \to s_1 \to \ldots \to s_t \to s_{t+1}.$$

where $s_t \to s_{t+1}$ is the state transition relation from Definition 4. *Note that we use the same symbol, $\to$, to represent both the transition relation for the execution transition system $\mathcal{T}_{\text{run}}$ and the computation prefix transition system $\mathcal{T}_{\text{pfx}}$ but that the relations can be distinguished by their context.*

Goal. Consider two programs, π and π' such that:

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

and

$$\pi' = \texttt{read x}; I'_1; I'_2; \ldots I'_{m'-1}; \texttt{write y}.$$

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 terminating computation prefix

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

there exists a terminating computation prefix for the transformed program

$$\pi', v \vdash \mathit{In}(v) \to (p'_1, \sigma'_1) \to \ldots \to (m', \sigma')$$

such that $\sigma(\mathrm{y}) = \sigma'(\mathrm{y})$, and conversely.

It is natural to try to prove this by induction on the length of computation prefixes. In practice the art is to find a relation $\mathcal{R}$ that holds between finite computation prefixes of the original program and those of the transformed program. $\mathcal{R}$ must be provable, and imply output equivalence for any program input v.

More explicitly: If C, C' are computation prefixes of π, π' on the same input v, we show that $s_i \mathcal{R} s'_j$ for every corresponding pair of states in C, C' where $\mathcal{R}$ is a relation on states that expresses "correct simulation".

Remark. The transformations in this paper all satisfy $m = m'$. Further, $C\mathcal{R}C'$ holds only if C, C' have the same length, and $p_i = p'_i$ for any i. Thus $C\mathcal{R}C'$ implies that $C \to C_1$ for some C_1 iff $C' \to C'_1$ for some C'_1.

The following lemma details the work that needs to be done to show that semantic equivalence is preserved by any one step by the transition system.

Lemma 1 (*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 the following three conditions hold*:

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

$$[\pi, v \vdash In(v)] \; \mathcal{R} \; [\pi', v \vdash In(v)]$$

2. (*Step case*) *If $C_1 \mathcal{R} C'_1$, $C_1 \to C_2$ and $C'_1 \to C'_2$ then $C_2 \mathcal{R} C'_2$.*
3. (*Equivalence*) *If*

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

then

$$\text{(i)} \quad p_t = m \;\Leftrightarrow\; p'_{t'} = m \text{ and}$$
$$\text{(ii)} \quad p_t = p'_{t'} = m \Rightarrow \sigma_t(\mathrm{y}) = \sigma'_{t'}(\mathrm{y})$$

Proof: Straightforward by two inductions. □

Proofs of equivalence are split into these three steps. This schema of proof provides a "top-level" approach to proving the correctness of optimizations. The questions still remain however of how to determine the relation $\mathcal{R}$ and how to prove the conditions of Lemma 1. In particular, it is the "step case" that is hardest to prove.

The relation $\mathcal{R}$ will clearly be derived from the CTL-FV side conditions of the transformations. Imagine we are defining $\mathcal{R}$ as a relation that holds between two computation prefixes C and C' where:

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

for some programs π,π' and initial value v. Let $s_i = (p_i, \sigma_i)$ and $s'_i = (p'_i, \sigma'_i)$ for all $i \geq 0$.

Consider the case where we have a side condition $p \models A(\phi_1\ U\ \phi_2)$ or $p \models A(\phi_1\ W\ \phi_2)$. A complete (terminating) trace that contains $p = p_i$ will have the following form:

$$\overbrace{s_0 \to \ldots \to s_{i-1}}^{\text{prior to until}} \to \underbrace{s_i \to \ldots \to s_{t-1}}_{\text{until section: } \phi_1 \text{ holds}} \to \overset{\phi_2 \text{ holds}}{\overset{\downarrow}{s_t}} \to \overbrace{s_{t+1} \to \ldots \to s_w}^{\text{after until}}$$

The computation is split into an "until section" and sections that are outside the until. For non-terminating computational prefixes, the last state of the prefix may also be in the until section. In the case of $p \models A(\phi_1\ W\ \phi_2)$, the until section may continue to the end of a terminating computation prefix with ϕ_2 never holding. The relation $\mathcal{R}$ will depend on this; *either* the last state of a computation prefix is within an until section and one condition, say $\mathcal{A}$, holds *or* it is outside this section and a different condition, say $\mathcal{B}$, holds. The relation $\mathcal{R}$ for transformations with *AU* or *AW* side conditions will naturally have the form where one of the following two cases hold:

– $\mathcal{A}$
– $\exists i : i < t \wedge p_i = p \wedge (\forall j : i \leq j < t \Rightarrow p_j \models \neg\phi_2)$ and $\mathcal{B}$

Here $\mathcal{A}$ and $\mathcal{B}$ are conditions chosen depending on the transformation and the formulae ϕ_1 and ϕ_2 in the temporal side condition (see the examples in the next section).

What about backwards until and waits-for formulae? In this case, where $p \models \overleftarrow{A}(\phi_1\ U\ \phi_2)$ or $p \models \overleftarrow{A}(\phi_1\ W\ \phi_2)$ is a side condition, we need to show as part of the relation $\mathcal{R}$ that being inside an until section preserves some property that can be used when leaving this section. So (in addition to some other conditions) the following will hold:

– $(\exists i : i < t \wedge p_i \models \phi_2 \wedge (\forall j : i < j \leq t \Rightarrow p_j \models \phi_1)) \Rightarrow \mathcal{A}$

A complete trace that contains $p = p_t$ will have the following form:

$$\overbrace{s_0 \to \ldots \to s_{i-1}}^{\text{prior to until}} \to \overset{\phi_2 \text{ holds}}{\overset{\downarrow}{s_i}} \to \underbrace{\ldots \to s_{t-1} \to s_t}_{\text{until section: } \phi_1 \text{ holds}} \to \overbrace{s_{t+1} \to \ldots \to s_w}^{\text{after until}}$$

Here $\mathcal{A}$ will again depend on the transformation and the formula ϕ_1 and ϕ_2. As before, in the case of the waits-for side condition, ϕ_1 may hold all the way to the beginning of the computation prefix with ϕ_2 never holding.

If the relation $\mathcal{R}$ follows these schemata then in conjunction with the side conditions of the transformation it will provide information about the last state of the computation prefix during the step case of the proof. This information will allow us to prove that the relation holds of the prefixes extended by one state.

The following two lemmas show how until formulae together with other conditions on a computation will indicate that a particular condition holds of the current state in a computation. The first says that $p \models A(\phi_1\ W\ \phi_2)$ implies that if ϕ_2 has never been true since p, then ϕ_1 has always been true since p. The second is its reverse-time analog of this result.

Lemma 2. *Suppose* $\pi, v \vdash (p_0, \sigma_0) \rightarrow \ldots \rightarrow (p_t, \sigma_t)$ *is a computation prefix and we know that*

$p \models A(\phi_1\ W\ \phi_2)$
and
$\exists i : i < t \wedge p_i = p \wedge (\forall j : i \leq j < t \Rightarrow p_j \models \neg\phi_2)$
and
$p_t \models \neg\phi_2$.

Then $p_t \models \phi_1$.

Proof: Straightforward from the definition of CTL-FV. □

Lemma 3. *Suppose* $\pi, v \vdash (p_0, \sigma_0) \rightarrow \ldots \rightarrow (p_t, \sigma_t)$ *is a computation prefix and we know that*

$p \models \overleftarrow{A}(\phi_1\ W\ \phi_2)$
and
$(\exists i : i < t \wedge p_i \models \phi_2 \wedge (\forall j : i \leq j < t \Rightarrow p_j \models \phi_1)) \Rightarrow \mathcal{A}$
and
$\exists i < t :\ p_i \models \neg\phi_1$
and
$p_t = p$.

Then $\mathcal{A}$ *will hold.*

Proof: Straightforward from the definition of CTL-FV. □

These two lemmas along with the side conditions of the transformation and the relation $\mathcal{R}$ will provide information about the current state of the computation. Given this information the table in Figure 4 shows how this will affect the next state of the computation. This will enable us to complete the step case of the proof. The following lemma shows how Figure 4 can be used.

Lemma 4. *Suppose* (σ_t, p_t) *is a state of* π, (σ'_t, p'_t) *is a state of* π' *and* $I_{p_t} = I'_{p'_t}$. *If* $(\sigma_t, p_t) \rightarrow (\sigma_{t+1}, p_{t+1})$ *and* $(\sigma'_t, p'_t) \rightarrow (\sigma'_{t+1}, p'_{t+1})$ *then the contents of a table entry in Figure* 4 *will be true if* p_t *satisfies the condition in the row header of the table and the states satisfy the condition in the column header* (*note that a* "?" *signifies that nothing significant holds of the next states*).

	$\sigma_t = \sigma'_t$	$\sigma_t \setminus x = \sigma'_t \setminus x$	$\sigma_t(x) = \sigma'_t(x)$
$\neg def(x)$	$\sigma_{t+1} = \sigma'_{t+1}$ $p_{t+1} = p'_{t+1}$ $\sigma_{t+1}(x) = \sigma_t(x)$ $\sigma'_{t+1}(x) = \sigma'_t(x)$ $\sigma_{t+1}(x) = \sigma'_{t+1}(x)$	$\sigma_{t+1}(x) = \sigma_t(x)$ $\sigma'_{t+1}(x) = \sigma'_t(x)$	$\sigma_t(x) =$ $\sigma_{t+1}(x) =$ $\sigma'_{t+1}(x) =$ $\sigma'_t(x)$
$trans(e)$	$\sigma_{t+1} = \sigma'_{t+1}$ $p_{t+1} = p'_{t+1}$ $[\![e]\!]_{\exp}\sigma_t =$ $[\![e]\!]_{\exp}\sigma_{t+1} =$ $[\![e]\!]_{\exp}\sigma'_{t+1} =$ $[\![e]\!]_{\exp}\sigma'_t$	$[\![e]\!]_{\exp}\sigma_{t+1} = [\![e]\!]_{\exp}\sigma_t$ $[\![e]\!]_{\exp}\sigma'_{t+1} = [\![e]\!]_{\exp}\sigma'_t$	?
$def(x) \wedge$ $\neg use(x)$	$\sigma_{t+1} = \sigma'_{t+1}$ $p_{t+1} = p'_{t+1}$	$\sigma_{t+1} = \sigma'_{t+1}$ $p_{t+1} = p'_{t+1}$	?
$\neg use(x)$	$\sigma_{t+1} = \sigma'_{t+1}$ $[-.7pt]p_{t+1} = p'_{t+1}$	$\sigma_{t+1} \setminus x = \sigma'_{t+1} \setminus x$ $p_{t+1} = p'_{t+1}$	?
$True$	$\sigma_{t+1} = \sigma'_{t+1}$ $p_{t+1} = p'_{t+1}$	?	?

Figure 4. Local pre/post conditions.

Proof: Tedious but straightforward from CTL-FV and Definition 4. □

Now we have a uniform method for proving that a transformation is correct. The following seven steps are to be followed:

1. Choose a relation $\mathcal{R}$ based on the side conditions of the transformation.
2. Use Lemma 1 to reduce the proof into 3 steps: the base case, the step case and the final equivalence step.
3. The base case is usually trivial.
4. For the step case: split into different cases depending on whether we are at the point of transformation and whether we are entering or leaving an until section.
5. If appropriate use Lemmas 2 and 3 to determine the conditions true in the current state.
6. Use the pattern of the rewrite or Lemma 4 to complete the step case of induction.
7. Finish by proving the third part of Lemma 1.

The next section provides three example proofs using this method.

5. The three examples

5.1. *Dead code elimination*

The dead code elimination rewrite rule described earlier was:

$$x := e \;\Rightarrow\; \texttt{skip}$$

if

$$AX\ A(\neg use(x)\ W\ (def(x) \wedge \neg use(x))).$$

Following Definition 11 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 X and e to a particular expression E. In this case we need to prove that an original program π and transformed program π' are equivalent. Below, we assume that $Rewrite(\pi, \pi', p, x{:=}e \Rightarrow$ skip if $AX\ A(\neg use(x)\ W\ def(x) \wedge \neg use(x)))$.

To prove that these two programs are equivalent we will use the method described in Lemma 1. However, to do this we need to come up with a relation $\mathcal{R}$ which holds between the two programs and ensures that the return value of the programs will be the same. To arrive at this relation we can examine the side condition of the transformation[2]:

$$AX\ A(\neg use(\mathtt{X})\ W\ def(\mathtt{X}) \wedge \neg use(\mathtt{X}))$$

A key sub-formula in this condition is $def(\mathtt{X}) \wedge \neg use(\mathtt{X})$ which, for brevity, shall be abbreviated in this proof to ϕ_2 (following the schema in the previous section). We can see from this condition that any label immediately following p will satisfy the formula $A(\neg use(x)\ W\ \phi_2)$ and this gives us a schema for the relation $\mathcal{R}$ (as described in the previous section) where one of the following two conditions holds:

– $\mathcal{A}$
– $\exists i : i < t \wedge p_i = p \wedge (\forall j : i \leq j < t \Rightarrow p_j \models \neg\phi_2)$ and $\mathcal{B}$

The question remains of what the conditions $\mathcal{A}$ and $\mathcal{B}$ should be. Outside of the until section we will know nothing about the program, so to ensure the return values are the same we need to stipulate that the store of the original program and its transformed version are the same i.e. $\mathcal{A} \equiv \sigma_t = \sigma'_t$. Within the until section we can see that $\neg use(\mathtt{X})$ will hold, this suggests (see Figure 4) that $\mathcal{B} \equiv \sigma_t \setminus \mathtt{X} = \sigma'_t \setminus \mathtt{X}$.

Definition 15. Consider $C \in \mathcal{T}_{\text{pfx}}(\pi, v)$ and $C' \in \mathcal{T}_{\text{pfx}}(\pi', v)$ 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 $\forall i : [0 \leq i \leq t \Rightarrow s_i = (p_i, \sigma_i)]$ and $\forall i : [0 \leq i \leq r \Rightarrow s'_i = (p_i, \sigma'_i)]$.

Then $C \mathcal{R} C'$ if and only if $t = r$, $p_t = p'_t$ and one of the following two cases holds:

1. $\sigma_t = \sigma'_t$.
2. $\exists j : j < t \wedge p = p_j \wedge \forall k : j < k < t \Rightarrow p_k \models \neg\phi_2$ and $\sigma_t \setminus \mathtt{X} = \sigma'_t \setminus \mathtt{X}$.[3]

Having defined a suitable $\mathcal{R}$ we can then proceed to prove the correctness of dead code elimination using the method laid out in the previous section.

Proof: (Dead code elimination satisfies the conditions of Lemma 1.)

Base case. For any initial prefixes: $\sigma_t = \sigma'_t$, so $C\mathcal{R}C$.

Step case. Suppose $C_1\mathcal{R}C'_1$ and neither is terminated. 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:

$$C_1 = \pi, v \vdash (p_0, \sigma_0) \rightarrow \ldots \rightarrow (p_t, \sigma_t)$$
$$C'_1 = \pi', v \vdash (p_0, \sigma'_0) \rightarrow \ldots \rightarrow (p_t, \sigma'_t)$$

Let I_p be the instruction in program π at p and I'_p be the instruction in program π' at p. To prove that $C_2\mathcal{R}C'_2$ we need to split the situation into several different cases based on whether we are at the point of transformation ($p_t = p$) or if we are potentially leaving the until section of the computation ($p_t \models \phi_2$). There follows the proofs for every part of the case split:

– Case 1: $p_t = p$

$C_1\mathcal{R}C'_1 \wedge p_t = p$
$\Rightarrow$ {Definition of $\mathcal{R}$}
$\sigma_t \setminus \mathtt{X} = \sigma'_t \setminus \mathtt{X} \wedge p_t = p$
$\Rightarrow$ {$I_p = x := e$, $I'_p = \mathtt{skip}$}
$\sigma_t \setminus \mathtt{X} = \sigma'_t \setminus \mathtt{X} \wedge p_t = p \wedge \sigma_{t+1} = \sigma_t[\mathtt{X} \mapsto [\![\mathtt{E}]\!]_{\exp}\sigma_t]$
$\wedge\ \sigma'_{t+1} = \sigma'_t \wedge p_{t+1} = p'_{t+1} = p_t + 1$
$\Rightarrow$ {Predicate calculus, properties of stores}
$p_{t+1} = p'_{t+1}$
$\wedge\ \exists j : j < (t+1) \wedge p = p_j \wedge \forall k : j < k < (t+1) \Rightarrow p_k \models \neg\phi_2$
$\wedge\ \sigma_{t+1} \setminus \mathtt{X} = \sigma'_{t+1} \setminus \mathtt{X}$
$\Rightarrow$ {Definition or $\mathcal{R}$ (Case 2)}
$C_2\mathcal{R}C'_2$

– Case 2: $p_t \neq p \wedge p_t \models \phi_2$

$p_t \neq p \wedge C_1\mathcal{R}C'_1 \wedge p_t \models \phi_2$
$\Rightarrow$ {Definition of $\mathcal{R}$}
$p_t \neq p \wedge \sigma_t \setminus \mathtt{X} = \sigma'_t \setminus \mathtt{X} \wedge p_t \models \mathit{def}(\mathtt{X}) \wedge \neg \mathit{use}(\mathtt{X})$
$\Rightarrow$ {Lemma 4}
$p_{t+1} = p'_{t+1} \wedge \sigma_{t+1} = \sigma'_{t+1}$
$\Rightarrow$ {Definition of $\mathcal{R}$ (Case 1)}
$C_2\mathcal{R}C'_2$

– Case 3: $p_t \neq p \wedge p_t \models \neg\phi_2 \wedge C_1 \mathcal{R} C_1'$ (by Case 1 of definition of $\mathcal{R}$)

$$
\begin{aligned}
& p_t \neq p \wedge p_t \models \neg\phi_2 \wedge C_1 \mathcal{R} C_1' \\
\Rightarrow\ & \{\text{Definition of } \mathcal{R} \text{ (Case 1)}\} \\
& p_t \neq p \wedge p_t = p_t' \wedge \sigma_t = \sigma_t' \\
\Rightarrow\ & \{\text{Lemma 4}\} \\
& \sigma_{t+1} = \sigma_{t+1}' \wedge p_{t+1} = p_{t+1}' \\
\Rightarrow\ & \{\text{Definition of } \mathcal{R} \text{ (Case 1)}\} \\
& C_2 \mathcal{R} C_2'
\end{aligned}
$$

– Case 4: $p_t \neq p \wedge p_t \models \neg\phi_2 \wedge C_1 \mathcal{R} C_1'$ (by Case 2 of definition of $\mathcal{R}$)

$$
\begin{aligned}
& p_t \neq p \wedge p_t \models \neg\phi_2 \wedge C_1 \mathcal{R} C_1' \\
\Rightarrow\ & \{\text{Definition of } \mathcal{R} \text{ (Case 2)}\} \\
& \quad \exists j : j < t \wedge p = p_j \wedge \forall k : j < k < t \Rightarrow p_k \models \neg\phi_2 \\
& \wedge\ \sigma_t \setminus \mathtt{X} = \sigma_t' \setminus \mathtt{X} \\
& \wedge\ p_t \neq p \wedge p_t \models \neg\phi_2 \\
\equiv\ & \{\text{Linear algebra}\} \\
& \quad \exists j : j < (t+1) \wedge p = p_j \wedge \forall k : j < k < (t+1) \Rightarrow p_k \models \neg\phi_2 \\
& \wedge\ \sigma_t \setminus \mathtt{X} = \sigma_t' \setminus \mathtt{X} \\
& \wedge\ p_t \neq p \\
\Rightarrow\ & \{\text{Lemma 2}\} \\
& \quad \exists j : j < (t+1) \wedge p = p_j \wedge \forall k : j < k < (t+1) \Rightarrow p_k \models \neg\phi_2 \\
& \wedge\ \sigma_t \setminus \mathtt{X} = \sigma_t' \setminus \mathtt{X} \\
& \wedge\ p_t \neq p \wedge p_t \models \neg \mathit{use}(\mathtt{X}) \\
\Rightarrow\ & \{\text{Lemma 4}\} \\
& \quad \exists j : j < (t+1) \wedge p = p_j \wedge \forall k : j < k < (t+1) \Rightarrow p_k \models \neg\phi_2 \\
& \wedge\ \sigma_{t+1} \setminus \mathtt{X} = \sigma_{t+1}' \setminus \mathtt{X} \\
& \wedge\ p_{t+1} = p_{t+1}' \\
\Rightarrow\ & \{\text{Definition of } \mathcal{R} \text{ (Case 2)}\} \\
& C_2 \mathcal{R} C_2'
\end{aligned}
$$

So in every case $C_2 \mathcal{R} C_2'$ and we have proven the step case of Lemma 1.

Equivalence. From the definition of $\mathcal{R}$, either $\sigma_t = \sigma_t'$ (in which case clearly $\sigma_t(\mathrm{y}) = \sigma_t'(\mathrm{y})$ or the following will hold:

$$\exists j : j \leq t \wedge p = p_j \wedge \forall k : j < k < t \Rightarrow p_k \models \neg\phi_2$$

In this case either $p_t \models \phi_2$ and it will not use $\mathtt{X}$ or $p_t \models \neg\phi_2$ and by Lemma 2 it will not use $\mathtt{X}$, in either case the instruction at p_t does not use $\mathtt{X}$ i.e., $\mathrm{y} \neq \mathtt{X}$. In this case we also know that $\sigma_t(\mathrm{y}) = \sigma_t'(\mathrm{y})$ since $\sigma_t \setminus \mathtt{X} = \sigma_t' \setminus \mathtt{X}$.

Therefore, by Lemma 1, we can conclude that $[\![\pi]\!] = [\![\pi']\!]$. □

5.2. *Constant folding*

The constant folding rule is:

$$x := v \Rightarrow x := c$$

if

$$\overleftarrow{A}(\neg def(v) \wedge \neg stmt(\texttt{read x})\ W\ stmt(v := c) \wedge conlit(c))$$

Following Definition 11, 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 X, v to a particular program variable V and c to a particular constant C. In this case we need to prove that an original program π and transformed program π' are equivalent. We will assume in the following that $Rewrite(\pi, \pi', p, x := v \Rightarrow x := c$ if $\overleftarrow{A}(\neg def(v) \wedge \neg stmt(\texttt{read x})\ W\ stmt(v := c) \wedge conlit(c)))$.

Again we shall use Lemma 1 in our proof. In this case we have one condition which is a backwards (weak) until. Following the method in the previous section we can see that part of the induction relation $\mathcal{R}$ should be of the form:

$$(\exists i : i < t \wedge p_i \models stmt(\texttt{V} := \texttt{C}) \wedge \forall j : i < j < t \Rightarrow p_j \models \neg def(\texttt{V})) \Rightarrow \mathcal{A}$$

It remains to decide what else must hold and what the condition $\mathcal{A}$ is. Since there is no other information available apart from the backwards until condition we need the final states of the computation prefix to be the same to ensure the final written values of the program are the same. For the condition $\mathcal{A}$ to maintain this at the point of transformation we require $\mathcal{A} \equiv \sigma_t(\texttt{V}) = [\![\texttt{C}]\!]_{\text{op}}$.

Definition 16. Consider $C \in \mathcal{T}_{\text{pfx}}(\pi, v)$ and $C' \in \mathcal{T}_{\text{pfx}}(\pi', v)$ 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 $\forall i : [0 \leq i \leq t \Rightarrow s_i = (p_i, \sigma_i)]$ and $\forall i : [0 \leq i \leq r \Rightarrow s'_i = (p_i, \sigma'_i)]$.

Then $C \mathcal{R} C'$ if and only if $t = r$, $p_t = p'_t$, $\sigma_t = \sigma'_t$ and

$$(\exists i : i < t \wedge p_i \models stmt(\texttt{V} := \texttt{C}) \wedge \forall j : i < j < t \Rightarrow p_j \models \neg def(\texttt{V})) \Rightarrow \sigma_t(\texttt{V}) = [\![\texttt{C}]\!]_{\text{op}}$$

Again, once we have defined the relation $\mathcal{R}$, we can proceed according to the method laid out in Lemma 1:

Proof: (Constant folding satisfies the conditions of Lemma 1.)

Base case. This holds trivially since traces of length one have the same initial (and final) state and the implication part of the relation will hold since its antecedent is false.

Step case. Suppose $\mathcal{R}$ 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 $C_2 \mathcal{R} C_2'$. There are two parts to this proof corresponding to different parts of the definition of $\mathcal{R}$. The first part is split depending on whether $p = p_t$.

- Case $p_t = p$: At the point of transformation $I_{p_t} = (\mathtt{X} := \mathtt{V})$ and $I_{p_t'} = (\mathtt{X} := \mathtt{C})$. So $p_{t+1} = p'_{t+1} = p_t + 1$, $\sigma_{t+1} = \sigma_t[\mathtt{X} \mapsto \sigma_t(\mathtt{V})]$, and $\sigma'_{t+1} = \sigma'_t[\mathtt{X} \mapsto [\![\mathtt{C}]\!]_{\text{op}}]$. By Lemma 3, we also know that $\sigma_t(\mathtt{V}) = [\![\mathtt{C}]\!]_{\text{op}}$, so $\sigma_{t+1} = \sigma_{t+1}$.
- Case $p_t \neq p$: By $\mathcal{R}$ we know that $\sigma_t = \sigma_t'$. So by Lemma 4: $\sigma_{t+1} = \sigma'_{t+1}$ and $p_{t+1} = p'_{t+1}$.

It remains to prove the second part of $C_2 \mathcal{R} C_2'$ i.e.,

$$\begin{gathered}(\exists i : i < (t+1) \wedge p_i \models \mathit{stmt}(\mathtt{V} := \mathtt{C}) \\ \wedge \forall j : i < j < (t+1) \Rightarrow p_j \models \neg \mathit{def}(\mathtt{V})) \\ \text{implies} \\ \sigma_t(\mathtt{V}) = [\![\mathtt{C}]\!]_{\text{op}}\end{gathered}$$

This part of the proof is split into two cases depending on whether $p_t \models \mathit{stmt}(\mathtt{V} := \mathtt{C})$

- Case $p_t \models \mathit{stmt}(\mathtt{V} := \mathtt{C})$: In this case, we know that $\sigma_{t+1}(\mathtt{V}) = (\sigma_t[\mathtt{V} \mapsto [\![\mathtt{C}]\!]_{\text{op}}])(\mathtt{V}) = [\![\mathtt{C}]\!]_{\text{op}}$.
- Case $p_t \models \neg \mathit{stmt}(\mathtt{V} := \mathtt{C}) \wedge \neg \mathit{def}(V)$: We can reason:

$$\begin{gathered}(\exists i : i < (t+1) \wedge p_i \models \mathit{stmt}(\mathtt{V} := \mathtt{C}) \\ \wedge \forall j : i < j < (t+1) \Rightarrow p_j \models \neg \mathit{def}(\mathtt{V})) \\ \Rightarrow \{p_t \models \neg \mathit{stmt}(\mathtt{V} := \mathtt{C}) \text{ implies } i < t\} \\ (\exists i : i < t) \wedge p_i \models \mathit{stmt}(\mathtt{V} := \mathtt{C}) \\ \wedge \forall j : i < j < t \Rightarrow p_j \models \neg \mathit{def}(\mathtt{V})) \\ \Rightarrow \{\text{Since } C_1 \mathcal{R} C_1'\} \\ \sigma_t(\mathtt{V}) = [\![\mathtt{C}]\!]_{\text{op}} \\ \Rightarrow \{\text{Since } p_t \models \neg \mathit{def}(\mathtt{V}), \textit{by Lemma 4}\} \\ \sigma_{t+1}(\mathtt{V}) = [\![\mathtt{C}]\!]_{\text{op}}\end{gathered}$$

– Case $p_t \models \neg stmt(\texttt{V := C}) \wedge def(V)$: We can reason:

$$
\begin{aligned}
&(\exists i : i < (t+1) \wedge p_i \models stmt(\texttt{V := C}) \\
&\wedge \forall j : i < j < (t+1) \Rightarrow p_j \models \neg def(\texttt{V})) \\
\Rightarrow\ &\{p_t \models \neg stmt(\texttt{V := C}) \text{ implies } i < t\} \\
&p_t \models \neg def(\texttt{V})) \\
\Rightarrow\ &\{\text{But } p_t \models def(\texttt{V})\} \\
&\mathit{False} \\
\Rightarrow\ &\{\text{Propositional calculus}\} \\
&\sigma_{t+1}(\texttt{V}) = [\![\texttt{C}]\!]_{\text{op}}
\end{aligned}
$$

So both parts of $\mathcal{R}$ hold i.e., $C_2 \mathcal{R} C_2'$.

Equivalence. Since $\mathcal{R}$ implies that $p_t = p_t'$ and $\sigma_t = \sigma_t'$, 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 1. □

5.3. *Code motion/loop invariant hoisting*

The code motion/loop invariant hoisting rule is:

$$
\begin{aligned}
&p : \texttt{skip} \Rightarrow x := e \\
&q : x := e \Rightarrow \texttt{skip}
\end{aligned}
$$

if

$$
\begin{aligned}
&p \models A(\neg use(x)\ W\ node(q)) \\
&q \models \neg use(x) \wedge \\
&\qquad \overleftarrow{A}((\neg def(x) \vee node(q)) \wedge trans(e) \wedge \neg stmt(\texttt{read x })\ W\ node(p))
\end{aligned}
$$

Following Definition 11, 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 X and e to a particular program expression E. In this case we need to prove that an original program π and transformed program π' are equivalent.

For this transformation we have both a forward until and a backward until so the relation $\mathcal{R}$ (and associated proof) will be a combination of the two patterns we have already seen. Note that we shall abbreviate the formula $(\neg def(x) \vee node(q)) \wedge trans(e) \wedge \neg stmt(\texttt{read x})$ to ϕ_1.

Definition 17. Suppose $C \in \mathcal{T}_{\text{pfx}}(\pi, v)$ and $C' \in \mathcal{T}_{\text{pfx}}(\pi', v)$ for some v such that

$$
\begin{aligned}
C &= \pi, v \vdash (p_0, \sigma_1) \rightarrow \ldots \rightarrow (p_t, \sigma_t) \\
C' &= \pi', v \vdash (p_0', \sigma_1') \rightarrow \ldots \rightarrow (p_{t'}', \sigma_{t'}')
\end{aligned}
$$

We then define the $\mathcal{R}$ relation on computation prefixes as: $C\mathcal{R}C'$ if and only if $t = t'$ and both $C\mathcal{R}_1C'$ and $C\mathcal{R}_2C'$ where $\mathcal{R}_1$ and $\mathcal{R}_2$ are defined by:

- $C\mathcal{R}_1C'$ if $p_t = p'_t$ and one of the following holds:
  1. $\sigma_t = \sigma'_t$
  2. $\exists i : i < t \wedge p_i = p \wedge (\forall j : i \leq j < t \Rightarrow p_j \neq q) \wedge \sigma_t \setminus \mathtt{X} = \sigma'_t \setminus \mathtt{X}$
- $C\mathcal{R}_2C'$ if the following holds:

$$(\exists i : i < t \wedge p_i = p \wedge (\forall j : i \leq j < t \Rightarrow p_j \models \phi_1)) \Rightarrow \sigma'_t(\mathtt{X}) = [\![\mathtt{E}]\!]_{\exp}\sigma_t$$

The proof of correctness will again be a combination of the two proofs similar to the ones we have already seen.

Proof: (Invariant hoisting satisfies the conditions of Lemma 1.)

Base case. This holds trivially.

Step case. Suppose $\mathcal{R}$ 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 $C_2\mathcal{R}C'_2$. First, we will prove that $C_2\mathcal{R}_1C'_2$. The proof is split depending on whether p_t is p or q or neither.

- Case $p_t = p$: In this case we know that $I_{p_t} = \mathtt{skip}$ and $I'_{p'_t} = (\mathtt{X} := \mathtt{E})$. So:

  $C_1\mathcal{R}C_1 \wedge I_{p_t} = \mathtt{skip} \wedge I'_{p'_t} = (\mathtt{X} := \mathtt{E})$
  $\Rightarrow$ {Definition of $\mathcal{R}_1$}
  $\sigma_t \setminus \mathtt{X} = \sigma'_t \setminus \mathtt{X} \wedge I_{p_t} = \mathtt{skip} \wedge I'_{p'_t} = \mathtt{X} := \mathtt{E}$
  $\Rightarrow$ {Semantics}
  $\sigma_{t+1} \setminus \mathtt{X} = \sigma'_{t+1} \setminus \mathtt{X} \wedge p_{t+1} = p'_{t+1} = p_t + 1$
  $\Rightarrow$ {Linear algebra, $p_t = p$, $p \neq q$}
  $\exists i : i < (t+1) \wedge p_i = p \wedge (\forall j : i \leq j < (t+1) \Rightarrow p_j \neq q)$
  $\wedge \sigma_{t+1} \setminus \mathtt{X} = \sigma'_{t+1} \setminus \mathtt{X} \wedge p_{t+1} = p'_{t+1} = p_t + 1$
  $\Rightarrow$ {Definition of $\mathcal{R}_1$ (Case 2)}
  $C_2\mathcal{R}_1C'_2$

– Case $p_t = q$: Here, $I_{p_t} = (\mathtt{X} := \mathtt{E})$ and $I'_{p'_t} = \mathtt{skip}$.

$C_1 \mathcal{R} C'_1$
$\Rightarrow$ {Definition of $\mathcal{R}$, Lemma 3 since in C_1 every q is preceded by a p}
$\sigma_t \setminus \mathtt{X} = \sigma'_t \setminus \mathtt{X} \wedge \sigma'_t(\mathtt{X}) = [\![\mathtt{E}]\!]_{\exp}\sigma_t \wedge I_{p_t} = (\mathtt{X} := \mathtt{E}) \wedge I'_{p'_t} = \mathtt{skip}$
$\Rightarrow$ {Semantics}
$\sigma_t \setminus \mathtt{X} = \sigma'_t \setminus \mathtt{X} \wedge \sigma'_t(\mathtt{X}) = [\![\mathtt{E}]\!]_{\exp}\sigma_t$
$\wedge\ \sigma_{t+1} = \sigma_t[\mathtt{X} \mapsto [\![\mathtt{E}]\!]_{\exp}\sigma_t] \wedge \sigma'_t = \sigma'_{t+1} \wedge p_{t+1} = p'_{t+1} = p_t + 1$
$\Rightarrow$ {Properties of stores}
$\sigma_{t+1} = \sigma'_{t+1} \wedge p_{t+1} = p'_{t+1}$
$\Rightarrow$ {Definition of $\mathcal{R}_1$ (Case 1)}
$C_2 \mathcal{R}_1 C_2$

– Case $p_t \notin \{p, q\} \wedge \sigma_t = \sigma'_t$:

$p_t \notin \{p, q\} \wedge \sigma_t = \sigma'_t$
$\Rightarrow$ {Lemma 4}
$p_{t+1} = p'_{t+1} \wedge \sigma_{t+1} = \sigma'_{t+1}$
$\Rightarrow$ {Definition of $\mathcal{R}_1$ (Case 1)}
$C_2 \mathcal{R}_1 C_2$

– Case $p_t \notin \{p, q\} \wedge \sigma_t \neq \sigma'_t$:

$\sigma_t \neq \sigma'_t \wedge C_1 \mathcal{R} C_1$
$\Rightarrow$ {Definition of $\mathcal{R}$}
$\sigma_t \setminus \mathtt{X} = \sigma'_t \setminus \mathtt{X}$
$\wedge\ \exists i : i < t \wedge p_i = p \wedge (\forall j : i \leq j < t \Rightarrow p_j \neq q)$
$\Rightarrow$ {$p_t \neq q$, Lemma 2}
$\sigma_t \setminus \mathtt{X} = \sigma'_t \setminus \mathtt{X} \wedge p_t \models \neg use(\mathtt{X})$
$\wedge\ \exists i : i < t \wedge p_i = p \wedge (\forall j : i \leq j \leq t \Rightarrow p_j \neq q)$
$\Rightarrow$ {Lemma 4, Linear Algebra}
$\sigma_{t+1} \setminus \mathtt{X} = \sigma'_{t+1} \setminus \mathtt{X}$
$\wedge\ \exists i : i < (t+1) \wedge p_i = p \wedge (\forall j : i \leq j < (t+1) \Rightarrow p_j \neq q)$
$\Rightarrow$ {Definition of $\mathcal{R}_1$ (Case 2)}
$C_2 \mathcal{R}_1 C_2$

So $C_2 \mathcal{R}_1 C'_2$ in every case and it remains to prove that $C_2 \mathcal{R}_2 C'_2$ i.e.,

$$(\exists i : i < (t+1) \wedge p_i = p \wedge (\forall j : i \leq j < (t+1) \Rightarrow p_j \models \phi_1)) \Rightarrow \sigma_{t+1}(\mathtt{X}) = [\![\mathtt{E}]\!]_{\exp}\sigma_{t+1}$$

This proof is split into four cases depending on whether $p_t = p$, whether $p_t = q$ and whether $p_t \models \phi_1$. Note that $FV(\mathtt{E})$ is the set of program variables in $\mathtt{E}$.

– Case $p_t = p$: In this case, we know that $I'_p = (\mathtt{X} := \mathtt{E})$ and $I_p = \mathtt{skip}$, So:

$\sigma'_{t+1}(\mathtt{X})$
$\equiv$ {Semantics}
$(\sigma'_t[\mathtt{X} \mapsto [\![\mathtt{E}]\!]_{\exp}\sigma'_t)(\mathtt{X})$
$\equiv$ {Property of stores}
$[\![\mathtt{E}]\!]_{\exp}\sigma'_t$
$\equiv$ {$\mathtt{X} \notin FV(\mathtt{E})$, $\sigma_t \setminus \mathtt{X} = \sigma'_t \setminus \mathtt{X}$ (since $C_1 \mathcal{R} C_2$)}
$[\![\mathtt{E}]\!]_{\exp}\sigma_t$
$\equiv$ {$I_p = \mathtt{skip}$}
$[\![\mathtt{E}]\!]_{\exp}\sigma_{t+1}$

– Case $p_t \neq p \wedge p_t \models \neg\phi_1$: We can reason:

$(\exists i : i < (t+1) \wedge p_i = p \wedge (\forall j : i \leq j < (t+1) \Rightarrow p_j \models \phi_1))$
$\Rightarrow$ {$p_t \neq p$ implies $i < t$}
$p_t \models \phi_1$
$\Rightarrow$ {But $p_t \models \neg\phi_1$}
False
$\Rightarrow$ {Propositional calculus}
$\sigma_{t+1}(\mathtt{X}) = [\![\mathtt{E}]\!]_{\exp}\sigma_{t+1}$

– Case $p_t \neq p \wedge p_t \models \phi_1 \wedge p_t \neq q$: We can reason:

$(\exists i : i < (t+1) \wedge p_i = p \wedge (\forall j : i \leq j < (t+1) \Rightarrow p_j \models \phi_1))$
$\Rightarrow$ {$p_t \neq p$ implies $i < t$}
$(\exists i : i < t \wedge p_i = p \wedge (\forall j : i \leq j < t \Rightarrow p_j \models \phi_1))$
$\Rightarrow$ {Since $C_1 \mathcal{R} C'_1$}
$\sigma'_t(\mathtt{X}) = [\![\mathtt{E}]\!]_{\exp}\sigma_t$
$\Rightarrow$ {$p_t \models \phi_1 \Rightarrow p_t \models$ *trans*($\mathtt{E}$), Lemma 4}
$\sigma'_t(\mathtt{X}) = [\![\mathtt{E}]\!]_{\exp}\sigma_{t+1}$
$\Rightarrow$ {Since $p_t \neq q$: $p_t \models \phi_1 \Rightarrow p_t \models \neg$*def*($\mathtt{X}$), Lemma 4}
$\sigma'_{t+1}(\mathtt{X}) = [\![\mathtt{E}]\!]_{\exp}\sigma_{t+1}$

– Case $p_t \neq p \wedge p_t \models \phi_1 \wedge p_t = q$: We can reason:

$(\exists i : i < (t+1) \wedge p_i = p \wedge (\forall j : i \leq j < (t+1) \Rightarrow p_j \models \phi_1))$
$\Rightarrow$ {$p_t \neq p$ implies $i < t$}
$(\exists i : i < t \wedge p_i = p \wedge (\forall j : i \leq j < t \Rightarrow p_j \models \phi_1))$

$\Rightarrow$ {Since $C_1 \mathcal{R} C_1'$}
$\sigma_t'(\mathtt{X}) = [\![\mathtt{E}]\!]_{\exp}\sigma_t$
$\Rightarrow$ {$I_{p_t} = (\mathtt{X} := \mathtt{E}), \mathtt{X} \notin FV(\mathtt{E})$}
$\sigma_t'(\mathtt{X}) = [\![\mathtt{E}]\!]_{\exp}\sigma_{t+1}$
$\Rightarrow$ {$I'_{p'_t} = \mathtt{skip}$}
$\sigma'_{t+1}(\mathtt{X}) = [\![\mathtt{E}]\!]_{\exp}\sigma_{t+1}$

Therefore, in every case, both $C_2 \mathcal{R}_1 C_2'$ and $C_2 \mathcal{R}_2 C_2'$ and we can conclude that $C_2 \mathcal{R} C_2'$ holds.

Equivalence. From the definition of $\mathcal{R}$, either $\sigma_t = \sigma_t'$ (in which case clearly $\sigma_t(\mathrm{y}) = \sigma_t'(\mathrm{y})$ or the following will hold:

$$\exists j : j \leq t \wedge p = p_j \wedge \forall k : j < k < t \Rightarrow p_i \neq q$$

In this case (since p_t will not be q), we know by Lemma 2 that the instruction at p_t does not use $\mathtt{X}$ i.e., $\mathrm{y} \neq \mathtt{X}$. In this case we also know that $\sigma_t(\mathrm{y}) = \sigma_t'(\mathrm{y})$ since $\sigma_t \setminus \mathtt{X} = \sigma_t' \setminus \mathtt{X}$.

Therefore, by Lemma 1, we can conclude that $[\![\pi]\!] = [\![\pi']\!]$. □

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 formulae.

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 identical 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.

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 [23].

The following sections discuss some issues on how the approach can be used in a real-world compiler and also how the specification language could be extended in small ways to allow specification of quite complex transformations.

6.1. *Real world application*

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. However, the specification of the transformations does not change, only their interpretation changes. This (perhaps) surprising fact shows how the logical formulae capture the essence of the transformation under which the details can be automatically derived. For more information see [22]. A follow-up paper describing the required adjustments is in preparation.

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 is closely related with the temporal logic side conditions.

An interesting direction of further work would be to discover if the relation could be completely mechanically created from the side conditions. In an initial experiment we transfered the proof of the correctness of dead code elimination into the theorem prover PVS [38]. This was not, however, an easy process but with more investigation we believe that more highly automated proofs could be performed.

The language we have treated is rather like a traditional compiler's "intermediate language". The conference version of this paper [24] stated "We anticipate that our method could be used to validate a great many traditional optimizing compiler transformations, e.g., many found in Aho et al. [2] and Muchnick [28]." This anticipation has born fruit in the form of two reports: Frederiksen [10] and Lerner et al. [25]. The first contains manual proofs of a number of such optimizations. The second describes an implementation, using a mechanised theorem-prover, that is clearly inspired by Lacey et al. [24]. Its rule annotation language is less powerful than CTL, but it uses an execution model that is closer to existing machine codes than the simplified version of this paper, and is more fully automated.

6.2. *Extensions to the specification language*

The language for specifying transformations is quite simple and to aid exposition we have described some simple transformations. However, more complicated transformations such as *lazy code motion* [18] can be specified given some small extensions to the language. One main extension is the ability to combine and alter rewrites using certain operators (similar to strategies in Visser et al. [43]). This also allows us to rewrite at an arbitrary number of points in the control flow graph. This is achieved by the *APPLY_ALL* operator. The following:

$$\mathit{APPLY\_ALL}\ (n : s_1 \Rightarrow s_2 \text{ if } \phi),$$

will replace the statement s_1 with s_2 at *every* program point at which the side condition holds. How will this affect our proof method? The answer is quite simple, instead of rewriting one point it will rewrite a set of points and the case splits will be based on membership of this set (of course within the set we know that the condition ϕ will hold). The rest of the proof method is the same as when rewriting a fixed set of points.

Using extensions such as this, a large number of compiler optimizations can be expressed, for more examples see [22].

6.3. Programs with exceptions

In our correctness proofs above, we defined semantic equivalence, $[\![\pi]\!] = [\![\pi']\!]$, to mean that both programs terminate on the same input with the same output. In Definition 2 we stated that the base language functions are all total; this prevents programs from terminating exceptionally. Thus, in keeping with standard practice, rewriting `x := 1/0` to `skip` is allowed if the value of `x` is not used. In practice, however, the base language functions are not total and the transformed program in this case will terminate normally whereas the original one will terminate with an exception; so the programs are not semantically equivalent using our above definition.

This is a semantic problem that is most often overlooked. In the proof schema and optimization correctness proofs above we have also overlooked this problem for simplicity and to be consistent with what is done in practice.

However, our formalism is general enough that there are several ways in which we can address this problem. In one possible solution, we first alter our definition of semantic equivalence so that only normally terminating traces of the original program π and the transformed program π' are considered. These are the computation prefixes that reach program point m, where $I_m =$ `read y`, i.e. those where $[\![\pi]\!]$ is defined. That is, π and π' are semantically equivalent if

$$[\![\pi]\!] \setminus (Dom([\![\pi]\!]) \cap Dom([\![\pi']\!])) = [\![\pi']\!] \setminus (Dom([\![\pi]\!]) \cap Dom([\![\pi']\!]))$$

where $Dom([\![\pi]\!])$ is the set of inputs leading to exception-free termination of π, that is, the domain of the partial function $[\![\pi]\!]$. We can then remove clause (i) of the third condition of Lemma 1 as follows:

3. (Equivalence) If

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

then

$$p_t = p'_{t'} = m \Rightarrow \sigma_t(\mathrm{y}) = \sigma'_{t'}(\mathrm{y})$$

This states that if both π and π' terminate normally then they output the same value, i.e. $\sigma_t(\mathrm{y}) = \sigma'_t(\mathrm{y})$. The proofs are modified as expected.

Acknowledgments

The Warwick and Minnesota authors would like to thank their colleagues in 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.

Notes

1. The *conlit* is introduced so that the model checker will not match c with a non-constant expression.
2. Note here that we have "substituted in" the values of the meta-variables in the formula.
3. Note that this is stronger than the condition in the schema presented earlier in that we've replaced the first $\leq$ in the $\forall$ with a $<$.

References

1. Abramsky, S. and Hankin, C. *Abstract Interpretation of Declarative Languages*. Ellis-Horwood, 1987.
2. Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.
3. Assmann, U. How to uniformly specify program analysis and transformation. In *Proc. 6th International Conference on Compiler Construction (CC'96)*, vol. 1060 of *Lecture Notes in Computer Science*, Springer-Verlag, 1996, pp. 121–135.
4. Clarke, E.M., Emerson, E.A., and Sistla, A.P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **8**(2) (1986) 244–263.
5. Cleaveland, R. and Jackson, D. In *Proceedings of First ACM SIGPLAN Workshop on Automated Analysis of Software*. Paris, France, Jan. 1997.
6. Cousot, P. Semantic foundations of program analysis. In *Program Flow Analysis: Theory and Applications*, S.S. Muchnick and N.D. Jones (Eds.), Englewood Cliffs, NJ, Prentice Hall, 1981, chap. 10, pp. 303–342.
7. Cousot, P. and Cousot, R. 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, New York, ACM, 1977, pp. 238–252.
8. Cousot, P. and Cousot, R. Systematic design of program transformations by abstract interpretation. In *Proc. of 29th ACM Symposium on Principles of Programming Languages*, ACM, 2000, pp. 178–190.
9. Cousot, P. and Cousot, R. Systematic design of program transformation frameworks by abstract interpretation. In *Proc. of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland, Oregon, ACM Press, 2002, pp. 178–190.
10. Frederiksen, C.C. Correctness of classical compiler optimizations using CTL. Unpublished TOPPS report, University of Copenhagen, 2001. www.diku.dk/research-groups/topps/bibliography/2001.html#D-443.
11. Frederiksen, C.C. Correctness of classical compiler optimizations using CTL logic. In *Compiler Optimization meets Compiler Verification (COCV)*, Satellite workshop at ETAPS 2002.
12. Hafer, Th. and Thomas, W. Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree. In *Automata, Languages and Programming Proceedings, ICALP'87*, vol. 267 of *Lecture Notes in Computer Science*, Springer-Verlag, 1987, pp. 267–279.
13. Havelund, K. *Stepwise Development of a Denotational Stack Semantics.* M.Sc. thesis, University of Copenhagen, 1984.
14. Hecht, M. *Flow Analysis of Computer Programs*. North-Holland, 1977.

15. Jones, N.D. (Ed.), *Semantics-Directed Compiler Generation*, vol. 94 of *Lecture Notes in Computer Science*, Springer-Verlag, 1980.
16. Jones, N.D. Semantique: Semantic-based program manipulation techniques. In *Bulletin European Association for Theoretical Computer Science*, **39** (1989) 74–83.
17. Jones, N.D. and Nielson, F. Abstract interpretation: A semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*, S. Abramsky, D. Gabbay, and T. Maibaum (Eds.), Oxford University Press, 1994, pp. 527–629.
18. Knoop, J., Rüthing, O., and Steffen, B. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **16**(4) (1994) 1117–1155.
19. Kozen, D. and Patron, M. Certification of compiler optimizations using Kleene algebra with tests. In *Proceedings of the 1st International Conference on Computational Logic (CL2000)*, J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L.M. Pereira, Y. Sagiv, and P.J. Stuckey (Eds.), *Lecture Notes in Artificial Intelligence*, vol. 1861, Springer-Verlag, London, 2000, pp. 568–582.
20. Kripke, S. Semantical analysis of modal logic i: Normal modal propositional calculi. *Zeitschrift f. Math. Logik und Grundlagen d. Math.*, **9** (1963).
21. Kupferman, O. and Pnueli, A. Once and for all. In *Proc. 10th IEEE Symposium on Logic in Computer Science*, San Diego, 1995, pp. 25–35.
22. Lacey, D. Program transformation using temporal logic specification. DPhil Thesis (forthcoming). Oxford University Computing Laboratory, 2003.
23. Lacey, D. and de Moor, O. Imperative program transformation by rewriting. In *Proc. 10th International Conf. on Compiler Construction*, vol. 1113 of *Lecture Notes in Computer Science*, Springer-Verlag, 2001, pp. 52–68.
24. Lacey, D., Jones, N.D., Van Wyk, E. and Frederiksen, C.C. Proving correctness of compiler optimizations by temporal logic. In *29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002, pp. 283–294.
25. Lerner, S., Grove, D., and Chambers, C. Composing dataflow analyses and transformations. In *29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002, pp. 283–294.
26. Lerner, S., Millstein, T., and Chambers, C. Automatically proving correctness of compiler optimizations. Technical Report UW-CSE-02-11-02, University of Washington, 2002.
27. Milne, R. and Strachey, C. *A Theory of Programming Language Semantics*. Chapman and Hall, 1976.
28. Muchnick, S.S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
29. Muchnick, S.S. and Jones, N.D. (Eds.), *Program Flow Analysis: Theory and Applications*. Englewood Cliffs, NJ, Prentice Hall, 1981.
30. Nielson, F. *Semantic Foundations of Data Flow Analysis.* M.Sc. thesis, Aarhus University, DAIMI PB-131, 1981.
31. Nielson, F. A denotational framework for data flow analysis. *Acta Informatica*, **18** (1982) 265–287.
32. Nielson, F., Nielson, H.R., and Hankin, C. *Principles of Program Analysis*. Springer-Verlag, 1999.
33. Pinter, S.S. and Wolper, P. A temporal logic for reasoning about partially ordered computations. In *Proc. 3rd ACM Symposium on Principles of Distributed Computing*, 1984, pp. 28–37.
34. Podelski, A., Steffen, B., and Vardi, M. *Schloss Ringberg Seminar: Model Checking and Program Analysis.* Workshop, Feb. 2000, Bavaria.
35. Rus, T. and Van Wyk, E. Using model checking in a parallelizing compiler. *Parallel Processing Letters*, **8**(4) (1998) 459–471.
36. Schmidt, D.A. Data-flow analysis is model checking of abstract interpretations. In *Proc. of 25th ACM Symposium on Principles of Programming Languages*, ACM, 1998.
37. Schmidt, D.A. and Steffen, B. Program analysis *as* model checking of abstract interpretations. In *Proc. of 5th Static Analysis Symposium*, G. Levi (Ed.), Pisa, vol. 1503 of *Lecture Notes in Computer Science*, Springer-Verlag, 1998.
38. SRI International. The PVS specification and verification system http://pvs.csl.sri.com/
39. Steckler, P.A. and Wand, M. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems, ACM*, **19**(1) (1997) 48–86.
40. Steffen, B. Data flow analysis as model checking. In *Proc. of 1st International Conference on Theoretical Aspects of Computer Software (TACS'91)*, Heidelberg, vol. 526 of *Lecture Notes in Computer Science*, Springer-Verlag, 1991, pp. 346–364.

41. Steffen, B. Generating data flow analysis algorithms from modal specifications. In *Science of Computer Programming*, 1993, vol. 21, pp. 115–139.
42. Steffen, B., Claßen, A., Klein, M., Knoop, J., and Margaria, T. The fixpoint analysis machine. In *Proc. of the 6th International Conference on Concurrency Theory (CONCUR'95)*, J. Lee and S. Smolka (Eds.), Philadelphia, Pennsylvania (USA), vol. 962 of *Lecture Notes in Computer Science*, Springer-Verlag, 1995, pp. 72–87.
43. Visser, E., Benaissa, Z., and Tolmach, A. Building program optimizers with rewriting strategies. In *Proc. of ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, ACM, 1998, pp. 13–26.
44. Whitfield, D. and Soffa, M.L. Automatic generation of global optimisers. In *Proc. of ACM SIGPLAN on Program Language Design and Implementation (PLDI'91)*, ACM, 1991, pp. 120–129.
45. Whitfield, D. and Soffa, M.L. An approach for exploring code-improving transformations. In *ACM Transactions on Programming Languages and Systems*, ACM, 1997, vol. 19, no. 6, pp. 1053–1084.
46. Winskel, G. *The Formal Semantics of Programming Languages*. Boston, MA, the MIT Press, 1993.
47. Wolper, P. On the relation of programs and computations to models of temporal logic. In *Proc. Temporal Logic in Specification*, vol. 398 of *Lecture Notes in Computer Science*, Springer-Verlag, 1987, pp. 75–123.