URL: http://www.elsevier.nl/locate/entcs/volume65.html 0 pages

Correctness of Classical Compiler Optimizations using CTL

Carl Christian Frederiksen [1,2]

DIKU, University of Copenhagen
Universitetsparken 1
2100 Copenhagen East
Denmark

Abstract

In this paper, global compiler optimizations are captured by conditional rewrite rules of the form ($\mathcal{I} \Rightarrow \mathcal{I}'$ if ϕ), where $\mathcal{I}$ and $\mathcal{I}'$ are program instructions and ϕ is a condition expressed in a variant of CTL, a formalism well suited to describe properties involving the control flow of a given program. The goal: to formally prove that if the condition ϕ is satisfied, then the rewrite rule $\mathcal{I} \Rightarrow \mathcal{I}'$ can be applied to the program without changing the semantics of the program. Once a rewrite rule has been proven correct, it can be directly and automatically utilized in an optimizing compiler.

The framework is based on joint work with David Lacey, Neil Jones and Eric Van Wyk [6]. The present paper presents a slightly simplified version of the framework, with emphasis on the CTL variants relation to CTL, along with a correctness proof of a transformation eliminating recomputations of available expressions.

1 Introduction

Reasoning about the correctness of transformations used in optimizing compilers for imperative programs is often complicated by the gap between the semantics of the language and the properties needed to be proven; semantic frameworks such as denotational semantics do not lend themselves easily to reasoning about data dependencies over computational futures and pasts.

To this end, the use of temporal logics [2,3] has proven to be useful for reasoning about such data dependencies. In particular by using temporal logics to support proofs of universal correctness of program transformations used in optimizing compilers, the proofs have become more tractable and have

[1] Based on joint work with David Lacey, Neil D. Jones and Eric Van Wyk [6].
[2] Email: xeno@diku.dk, Fax: (+45) 35 32 14 01.

lead to stronger optimization algorithms [4]. Other works have investigated temporal logic as a means to express program analyses and to derive *analysis* algorithms [12,13].

However *declarative* methods for specifying compiler optimizations have received relatively little attention. One approach to reason about program transformations for imperative programs by Kozen and Patron [5] demonstrates how an extension of Kleene algebra, Kleene algebra with tests (KAT), can be utilized to prove correctness of an extensive collection of instances of program transformations. The paper sets out with a different perspective on program transformation; the main focus is to show how algebraic laws can be applied to program manipulations specified as KAT equalities in order to reason about program transformations. Unfortunately, no *automatic* method is given to apply the transformations.

The present paper aims to formalize a framework [6] for specifying and proving classical compiler optimizations correct. Many program transformations can be expressed as a conditional rewrite rule of the form ($\mathcal{I} \Rightarrow \mathcal{I}'$ if ϕ) where $\mathcal{I}$ and $\mathcal{I}'$ are program instructions and ϕ is temporal logic formula. The interpretation: If the side condition ϕ is satisfied for a particular model of the subject program then the statement $\mathcal{I}$ can be replaced by $\mathcal{I}'$. The use of temporal logic is central to the universal correctness of the transformation – essentially the temporal property corresponds to a program analysis.

The goal: to demonstrate how the conditional rewrite rules can be proven correct, i.e. that the transformation preserves the semantics of the subject program. To this end a relation on programs is defined: if $Apply(\pi, \pi', p, \mathcal{I} \Rightarrow \mathcal{I}'$ if $\phi)$ holds then application of the transformation ($\mathcal{I} \Rightarrow \mathcal{I}'$ if ϕ) to the subject program π at program point p results in π' where $[\![\pi]\!] = [\![\pi']\!]$, i.e. π and π' have the same semantics.

Furthermore, the application of program transformations specified in the framework to programs, is computable – i.e. the specifications can, once proven correct, be automatically and safely utilized by an optimizing compiler. This brings *specification* and *implementation* closer together, increasing the confidence in the compiler implementation.

1.1 Overview of the Remainder of the Paper

Section 2 introduces a small imperative language with unstructured control flow and the notion of a *computation prefix* is defined – a computation prefix is a prefix of a computation trace. The computation prefixes will be a key factor in proving semantic program equivalence for the transformations. Section 3 introduces an extension of CTL (CTL-FV) that will be used for specifying the side conditions in the transformation rules. In Section 4 the transformation rules are introduced and as an example of how a transformation can be proven correct, Section 5 gives a proof of the *elimination of recomputation of available expressions* transformation. Finally, a summary is given in Section 6.

2 Language

The language used in this paper is a small imperative language without procedures. A program has the following form.

$$\pi \in pgm \ ::= \texttt{read x};\ 1 : stmt_1;\ \ldots\ m : stmt_m;\ m+1 : \texttt{write y};$$

$$\mathcal{I} \in stmt ::= x := exp \mid \texttt{if } x \texttt{ then } p_1 \texttt{ else } p_2 \mid \texttt{skip}$$

$$e \in exp \ ::= c \mid x \mid \texttt{op}(e_1, \ldots, e_n)$$

$$x \in \mathit{Variable};\ p_1, p_2 \in \{1, \ldots, m+1\}.$$

Conditionals are not allowed to perform tests directly on expressions, but only on variables. Furthermore, the `read` statement always uses the variable `x` and the `write` statement always uses the variable `y`. Each statement is labeled by a label $p \in \{1, \ldots, \mathit{exit}(\pi)\}$, where $\mathit{exit}(\pi) = m+1$. The statement occurring at label p is referred to as $\mathcal{I}_p$. The set of variables occurring in the program is denoted $\mathit{vars}(\pi)$ and the set of expressions is denoted $\mathit{expr}(\pi)$.

2.1 Semantics

Programs are regarded as being specifications of partial functions; a program reads the input, performs some (possibly nonterminating) computation and outputs the result. Two programs are defined to be equivalent if and only if they compute the same partial function. We assume given a domain of values *Value*, a set of operations on it *Op* and an interpretation of the operators $[\![\cdot]\!]_{op} : \mathit{Value}^* \to \mathit{Value}$. For *Value* it is assumed that there exists a distinguished element *true*, but is otherwise unspecified.

Definition 2.1 *The program store is a finite map that maps variables to their values. The set of stores is denoted* $\mathit{Sto} = \mathit{Variable} \rightharpoonup \mathit{Value}$. *Suppose* $\sigma : X \to \mathit{Value}$, *where* $\mathit{dom}(\sigma) = X \subseteq \mathit{Variable}$:

(i) $\sigma[x \mapsto v]$ *is the extension of* σ *to* $X \cup \{x\}$ *such that* $\sigma[x \mapsto v](x) = v$ *and* $\forall y \in X \setminus \{x\} : \sigma[x \mapsto v](y) = \sigma(y)$.

(ii) $\sigma|_{X'}$ *denotes the restriction of* σ *to the domain* $X' \subseteq X$, $\sigma|_{X'} : X' \to \mathit{Value}$ *where* $\forall x \in X' : \sigma(x) = \sigma|_{X'}(x)$. *As a short-hand we will write* $\sigma \setminus x = \sigma|_{X \setminus \{x\}}$.

Definition 2.2 *Expression evaluation is defined by the function* $[\![\cdot]\!]\cdot : \mathit{exp} \to \mathit{Sto} \to \mathit{Value}$.

$$[\![\texttt{x}]\!]\sigma = \sigma(\texttt{x})$$

$$[\![\texttt{op}(e_1, \ldots, e_n)]\!]\sigma = [\![\texttt{op}]\!]_{op}([\![e_1]\!]\sigma, \ldots, [\![e_n]\!]\sigma)$$

Definition 2.3 *A* program state *for a given program* π, *is a pair* (p, σ) *where* p *is a label in* π *and* σ *is a store such that* $\mathit{vars}(\pi) \subseteq \mathit{dom}(\sigma)$. *The set of all program states is denoted PgmState, in order to avoid confusion when* model states *are introduced in Section 3.*

Definition 2.4 *Let the semantic transition relation* $\rightarrow \subseteq PgmState \times PgmState$ *for a program* $\pi \in pgm$ *be defined by*[3] *:*

(i) *If* $\mathcal{I}_p = \texttt{skip}$ *then* $(p, \sigma) \rightarrow (p+1, \sigma)$.

(ii) *If* $\mathcal{I}_p = (x{:=}e)$ *then* $(p, \sigma) \rightarrow (p+1, \sigma[x \mapsto [\![e]\!]\sigma])$.

(iii) *If* $\mathcal{I}_p = (\texttt{if } x \texttt{ then } p_1 \texttt{ else } p_2)$ *and* $(\sigma(x) = \texttt{true})$ *then* $(p, \sigma) \rightarrow (p_1, \sigma)$

(iv) *If* $\mathcal{I}_p = (\texttt{if } x \texttt{ then } p_1 \texttt{ else } p_2)$ *and* $(\sigma(x) \neq \texttt{true})$ *then* $(p, \sigma) \rightarrow (p_2, \sigma)$

(v) $(exit(\pi), \sigma) \rightarrow (exit(\pi), \sigma)$.

Definition 2.5 *The initial state* $in_\pi(v) \in PgmState$ *for a program* $\pi \in pgm$ *and input* $v \in Value$ *is defined by*

$$in_\pi(v) = (1, [\texttt{x} \mapsto v, y_1 \mapsto true, \ldots, y_k \mapsto true])$$

where $vars(\pi) \setminus \{\texttt{x}\} = \{y_1, \ldots, y_k\}$. *In short the initial state is the pair consisting of the entry label and a store that maps the variable* `x` *to the input of the program, and all other variables to the value true.*

Definition 2.6 *A* computation prefix *for a program* $\pi \in pgm$ *and a value* $v \in Value$ *is a finite or infinite sequence* $C \in PgmState^{*\omega}$

$$C = \pi, v \vdash (p_0, \sigma_0) \rightarrow (p_1, \sigma_1) \rightarrow \ldots$$

such that $(p_0, \sigma_0) = in_\pi(v)$ *and* $(p_i, \sigma_i) \rightarrow (p_{i+1}, \sigma_{i+1})$ *for all* $i \geq 0$ [4] .

The set of computation prefixes, for a given program π *and an input value* v*, is denoted* $\mathcal{T}_{pfx}(\pi, v)$.

Note that when a computation begins, the initial `read` instruction has already been executed. The program then proceeds to perform the computation, but keeps looping in the last state once done. The last point simplifies the correctness proofs slightly, since we can find computation prefixes of arbitrary length even when the program terminates.

Definition 2.7 *Suppose* $\pi \in pgm$ *is a program. The meaning function* $[\![\cdot]\!]\cdot : pgm \rightarrow Value \rightharpoonup Value$ *is defined as* $[\![\pi]\!]v = \sigma_k(\texttt{y})$ *if there exists a computation prefix* $\pi, v \vdash (p_0, \sigma_0) \rightarrow \ldots \rightarrow (p_k, \sigma_k)$ *such that* $p_k = exit(\pi)$.

3 Temporal Logic

A program transformation rule is expressed as a rewrite rule with a side condition specified in a variant of CTL. The use of temporal logic allows expressing properties of a model of the subject program which translates into properties of computation prefixes of the subject program. The side condition corresponds to some analysis that a compiler might perform in order to ensure that the semantics of the subject program is preserved.

[3] Note that the target labels in conditionals can be *any* two labels in the program.

[4] The notation S^* denotes the set of *finite words* over S, S^ω denotes *infinite words* and $S^{*\omega}$ denotes *finite or infinite words.*

3.1 CTL-FV

We present an extension to CTL called CTL-FV, in order to express side conditions for the transformation rules. The extension is 4-fold.

First, the ordinary definition of CTL only allows for reasoning about the "future" in the model, i.e. we can only reason about states that can be reached by the models transition relation ($\rightarrow$). Since we in essential ways will use the side condition to express some program analysis that must hold prior to the application of the transformation (e.g. expression `x+y` is available at label p), it is vital that we allow reasoning about the "past" in the model as well [11]. This is done by introducing two new versions of the path operators exists (E) and forall (A): $\overleftarrow{E}$ and $\overleftarrow{A}$. For the interpretation of the "past" operators the sub formulae must hold over backwards paths, i.e. following the $\leftarrow$ relation. Since the definition of a CTL model only requires the $\rightarrow$ relation to be total, a given well-defined CTL model might not be a well-defined CTL-FV model. We handle this by *dropping* the totality requirement on the $\rightarrow$ relation. This allows us to express the control flow in a more natural way, but at the same time we are forced to specify how to handle finite paths.

Definition 3.1 *For a given set of atomic propositions (AP), a* CTL-FV model *is a triple* $\mathcal{M} = (S, \rightarrow, V)$ *where* S *is a set of states and* $\rightarrow \subseteq S \times S$ *is a relation (not necessarily total) between elements of* S*. The function* $V : S \rightarrow 2^{AP}$ *is called the* valuation*, mapping states to atomic propositions that hold in the given state.*

Second, the definition of CTL-FV is extended to handle models with "dead states", states with no successor following the direction of the path. Since the definition is also extended to handle backwards paths, this applies in both directions. The key for defining a path in a such a model is the notion of a *maximal path*. We will not allow reasoning about a finite path that does not end in a dead state.

Definition 3.2 *A path* $(n_i)_{i\geq 0}$ *is a sequence of states in* $S^{*\omega}$ *such that either* $n_i \rightarrow n_{i+1}$ *for all* $i \geq 0$ *(a forwards path) or* $n_{i+1} \leftarrow n_i$ *for all* $i \geq 0$ *(a backwards path). A path* $(n_i)_{i\geq 0}$ *in* $S^{*\omega}$ *is* maximal *if* $(n_i)_{i\geq 0}$ *is infinite, or of the form* $(n_i)_{0\leq i\leq k}$ *where* n_k *is* dead *meaning:*

$$\neg\exists n \in S : n_k \leftarrow n \text{ if } (n_i)_{0\leq i\leq k} \text{ is a backwards path, and}$$

$$\neg\exists n \in S : n_k \rightarrow n \text{ if } (n_i)_{0\leq i\leq k} \text{ is a forwards path.}$$

The set of all maximal paths over S *will be denoted* S^{MAX}*.*

Third, since a path now can be finite, the definitions of the path operators must be extended to such paths. For the next operator (X), we will additionally require that the next state exists. The strong until operator (U) already expresses a property of a finite (but arbitrary length) segment of a path, so the ordinary CTL definition is sufficient. The definition of weak until (W) will require that the first clause must hold for *infinitely* many states, if

the second clause never holds. In particular this rules out finite paths, if the second clause is never satisfied. This modification allows us to reason across loops in the control flow model of the subject program.

Fourth, and most importantly, in order to be able to express general transformation specification (as opposed to concrete transformation rules), the logic is equipped with free variables. This allows a finer structure on the atomic propositions; in particular instead of considering the set of atomic propositions AP as "tokens", it can be regarded as a set of *predicates over terms extracted from the subject program.*

3.1.1 The Control Flow Model

The model in which the satisfaction of the side condition will be decided, is a model of the control flow of the subject program that includes syntactic information about the subject program. This information is easily computed by traversing the subject program. The model is sufficiently detailed to express many of the classical control flow based compiler optimizations.

Definition 3.3 *The* control flow model $\mathcal{M}_{cf}(\pi) = (S, \rightarrow_{cf}, V)$ *defines a model based on the program* $\pi \in pgm$*:*

(i) *The set of states* S *is defined by* $S = \{1, \ldots, exit(\pi)\}$.

(ii) *The state transition relation,* $\rightarrow_{cf}$: $S \times S$, *is defined as* $p \rightarrow_{cf} p'$ *if and only if*

$$\begin{aligned} \mathcal{I}_p = (\texttt{if } x \texttt{ then } p_1 \texttt{ else } p_2) \wedge p' \in \{p_1, p_2\} \quad \vee \\ (\mathcal{I}_p = (\texttt{skip}) \vee \mathcal{I}_p = (x{:=}e)) \wedge p' = p + 1 \quad \vee \\ \mathcal{I}_p = (\texttt{write y}) \wedge p' = p. \end{aligned}$$

(iii) *The* valuation $V : S \rightarrow 2^{AP}$ *is defined as* [5] *:*

$$\begin{aligned} node(q) \in V(p) &\Leftrightarrow p = q \\ stmt(\mathcal{I}) \in V(p) &\Leftrightarrow \mathcal{I}_p = \mathcal{I} \\ def(x) \in V(p) &\Leftrightarrow \exists e \in expr(\pi) : \mathcal{I}_p = (x{:=}e) \\ use(y) \in V(p) &\Leftrightarrow \exists x \in vars(\pi), \exists e \in expr(\pi), \exists p_1, p_2 \in \{1, \ldots, exit(\pi)\} : \\ &\qquad \mathcal{I}_p = (x{:=}e) \wedge y \in vars(e) \quad \vee \\ &\qquad \mathcal{I}_p = (\texttt{if } y \texttt{ then } p_1 \texttt{ else } p_2) \quad \vee \\ &\qquad \mathcal{I}_p = (\texttt{write } y) \\ trans(e) \in V(p) &\Leftrightarrow e \in expr(\pi) \wedge \forall x \in vars(e) : def(x) \notin V(p) \end{aligned}$$

By definition, each label, statement, variable and expression occurring in an atomic proposition in the valuation must occur in the subject program.

[5] An expression is said to be *transparent* if none of its variables are assigned to.

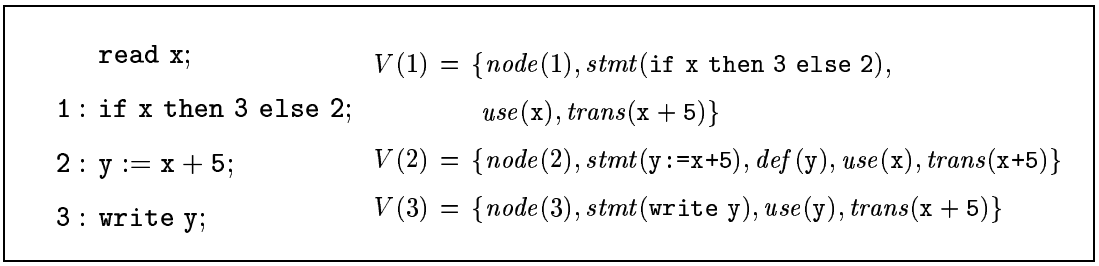

Figure 3.1: Example of a valuation.

When the control flow model is constructed, the valuation is generated by traversing the program syntax and inserting those atomic propositions that hold at a particular label p in the valuation $V(p)$. An example of an extracted valuation is given in Figure 3.1.

In order to later support correctness proofs, we make a few observations on the valuation for the control flow model.

Lemma 3.4 *Suppose* $(p_1, \sigma_1) \rightarrow (p_2, \sigma_2)$, $(p_1, \sigma_1') \rightarrow (p_2, \sigma_2')$, $\sigma_1 \setminus x = \sigma_1' \setminus x$ *and* $use(x) \notin V(p_1)$ *then* $\sigma_2 \setminus x = \sigma_2' \setminus x$.

Lemma 3.5 *Suppose* $(p_0, \sigma_0) \rightarrow \ldots \rightarrow (p_k, \sigma_k)$ *and* $\forall 0 \leq i < k : \forall x \in X :$ $def(x) \notin V(p_i)$ *then* $\sigma_0|_X = \sigma_k|_X$.

Next we establish a lemma stating that the control flow model is an abstraction of the semantics of a program.

Lemma 3.6 *Suppose* $\pi \in pgm$ *and* $C \in \mathcal{T}_{pfx}(\pi, v)$ *is a finite computation prefix for some value* $v \in Value$, $C = \pi, v \vdash (p_0, \sigma_0) \rightarrow \ldots \rightarrow (p_t, \sigma_t)$. *For all* $0 \leq i \leq t$:

(i) $p_i \rightarrow_{cf} \ldots \rightarrow_{cf} p_t$ *is a forwards path in* $\mathcal{M}_{cf}$.

(ii) $p_0 \leftarrow_{cf} \ldots \leftarrow_{cf} p_i$ *is a maximal backwards path in* $\mathcal{M}_{cf}$.

3.2 Semantics of CTL-FV

Definition 3.7 *The syntax of a CTL-FV formula is given below.*

$$\phi ::= true \mid ap(x_1, \ldots, x_n) \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid A\psi \mid E\psi \mid \overleftarrow{A}\psi \mid \overleftarrow{E}\psi$$

$$\psi ::= X\phi \mid \phi_1 U \phi_2 \mid \phi_1 W \phi_2$$

Definition 3.8 *A CTL-FV clause is, similarly to an ordinary CTL clause, written* $\mathcal{M}, n \models_\theta \phi$ *where* $\mathcal{M}$ *is the CTL-FV model,* n *is a state in the model,* ϕ *is a CTL-FV formula and* θ *is a substitution. The clause is* satisfied *if and only if the clause evaluates to true under the definition of CTL-FV satisfaction given in Figure 3.2. If the model is clear from the context then* $\mathcal{M}$ *can be omitted in the clause.*

Similarly to the usual definition of CTL, a number of derived operators can be defined: the path operators F, G, and the boolean operators $\vee$, $\Rightarrow$ etc.

The job of a CTL-FV model checker is not to find a set of states that satisfy the clause, but to find a set of *instantiation substitutions* such that the

$$
\begin{array}{ll}
\mathcal{M}, n \models_\theta true & \text{iff } true \\
\mathcal{M}, n \models_\theta ap(x_1, \ldots, x_n) & \text{iff } ap(\theta x_1, \ldots, \theta x_n) \in V(n) \\
\mathcal{M}, n \models_\theta \neg\phi & \text{iff not } \mathcal{M}, n \models_\theta \phi \\
\mathcal{M}, n \models_\theta \phi_1 \wedge \phi_2 & \text{iff } \mathcal{M}, n \models_\theta \phi_1 \text{ and } \mathcal{M}, n \models_\theta \phi_2 \\
\mathcal{M}, n \models_\theta A\psi & \text{iff } \forall(n = n_0 \to n_1 \to \ldots) \in S^{\text{MAX}} : \mathcal{M}, (n_i)_{i\geq 0} \models_\theta \psi \\
\mathcal{M}, n \models_\theta \overleftarrow{A}\psi & \text{iff } \forall(\ldots \to n_1 \to n_0 = n) \in S^{\text{MAX}} : \mathcal{M}, (n_i)_{i\geq 0} \models_\theta \psi \\
\mathcal{M}, n \models_\theta E\psi & \text{iff } \exists(n = n_0 \to n_1 \to \ldots) \in S^{\text{MAX}} : \mathcal{M}, (n_i)_{i\geq 0} \models_\theta \psi \\
\mathcal{M}, n \models_\theta \overleftarrow{E}\psi & \text{iff } \exists(\ldots \to n_1 \to n_0 = n) \in S^{\text{MAX}} : \mathcal{M}, (n_i)_{i\geq 0} \models_\theta \psi \\
 & \\
\mathcal{M}, (n_i)_{i\geq 0} \models_\theta X\phi & \text{iff } n_1 \text{ exists} \wedge n_1 \models_\theta \phi \\
\mathcal{M}, (n_i)_{i\geq 0} \models_\theta \phi_1 U \phi_2 & \text{iff } \exists i \geq 0 : \mathcal{M}, n_i \models_\theta \phi_2 \wedge \forall 0 \leq j < i : \mathcal{M}, n_j \models_\theta \phi_1 \\
\mathcal{M}, (n_i)_{i\geq 0} \models_\theta \phi_1 W \phi_2 & \text{iff } \exists i \geq 0 : \mathcal{M}, n_i \models_\theta \phi_2 \wedge \forall 0 \leq j < i : \mathcal{M}, n_j \models_\theta \phi_1 \\
 & \quad \vee\ (\forall k \geq 0 : n_k \models_\theta \phi_1 \wedge n_{k+1} \text{ exists})
\end{array}
$$

Figure 3.2: Semantics of CTL-FV .

CTL-FV clause is satisfied. In particular, $\mathcal{M}, n \models_\theta \phi$ is satisfied if and only if $\mathcal{M}, n \models \theta\phi$ is satisfied in CTL extended with finite paths and backwards path operators, as defined above. For example, when model checking $n \models_\theta use(v)$ over the control flow model for the example program in 3.1, the model checker would return: $\Theta = \{[n \mapsto 1, v \mapsto \mathtt{x}], [n \mapsto 2, v \mapsto \mathtt{x}], [n \mapsto 3, v \mapsto \mathtt{y}]\}$.

Returning to the control flow model, we observe that any maximal forward path is infinite, and that any maximal backwards path that can occur in an actual computation must be finite. The strong until operator asserts a severe restriction on the paths; if $\overleftarrow{A}(\phi_1 U \phi_2)$ is satisfied then it is not possible for ϕ_1 to be satisfied inside a loop and for ϕ_2 only to be satisfied outside the loop, since the (computationally unrealizable) path looping infinitely in the loop would violate the clause. For this reason the weak until operator is very useful for specifying properties of the control flow, precisely because it allows infinite loops. Also note that since any (computationally legal) backwards path is finite, then satisfaction of $\phi_1 W \phi_2$ for a backwards path implies that ϕ_2 is eventually satisfied.

4 Transformation

This section defines a framework for describing program transformation specifications. An important point is that the applicability of a transformation rule to a program at a given label is computable. It is thus possible to write a

compiler that admits transformation specifications that the compiler can use to transform subject programs.

4.1 Transformation Rule Syntax

A transformation rule is written: $(\mathcal{I} \Rightarrow \mathcal{I}' \text{ if } \phi)$, where $\mathcal{I}$ and $\mathcal{I}'$ are program statements with free variables and the side condition ϕ is a CTL-FV formula to be checked. This allows for specification of transformation templates that intuitively can be instantiated (by binding the free variables) to give a concrete transformation rule.

4.2 Application of Transformations

We now define what it means to apply a rewrite rule to a given program at a specified program point.

Definition 4.1 *Suppose $\pi, \pi' \in pgm$ are programs, p is a label in π, $(\mathcal{I} \Rightarrow \mathcal{I}' \text{ if } \phi)$ is a transformation rule and π has the form:*

$$\pi = \texttt{read x};\ 1 : \mathcal{I}_1;\ \ldots\ p : \mathcal{I}_p;\ \ldots\ m : \mathcal{I}_m;\ m+1 : \texttt{write y}.$$

The relation $Apply(\pi, \pi', p, \mathcal{I} \Rightarrow \mathcal{I}' \text{ if } \phi)$ holds if and only if for some substitution θ, the following holds:

(i) *(Verification): The CTL-FV clause $\mathcal{M}_{cf}(\pi), p \models_\theta stmt(\mathcal{I}) \wedge \phi$ is satisfied.*

(ii) *(Construction): The transformed program $\pi' \in pgm$ has the form*

$$\pi = \texttt{read x};\ 1 : \mathcal{I}_1;\ \ldots\ p : \theta(\mathcal{I}');\ \ldots\ m : \mathcal{I}_m;\ m+1 : \texttt{write y}.$$

If $Apply(\pi, \pi', p, \mathcal{I} \Rightarrow \mathcal{I}' \text{ if } \phi)$ holds then *application* of the transformation $(\mathcal{I} \Rightarrow \mathcal{I}' \text{ if } \phi)$ to program π at label p, would result in the program π'.

In other words, if the verification succeeds for some substitution θ then the instruction $\mathcal{I}_p$ is replaced by the left-hand side of the instantiated rewrite rule. The reason why the $stmt(\mathcal{I})$ proposition is included in the formula to be model checked, is to force the model checker to unify the left-hand side of the rewrite rule with the instruction at label p, e.g. satisfaction of $p \models_\theta stmt(\mathcal{I})$ implies $\mathcal{I}_p = \theta(\mathcal{I})$. Thus if the verification succeeds, then the side condition is satisfied and the substitution θ unifies the statement at label p with the left-hand side of the rewrite rule.

In an actual implementation, the set of substitutions Θ such that the side condition is satisfied can be found by model checking. Goal: All substitutions in the resulting set satisfy the entire side condition and each substitution gives rise to a sound transformation, provided that the transformation rule was sound to begin with. The framework can be easily extended to transformation rules with an arbitrary (but fixed) number of rewrite rules, each with a corresponding side condition.

5 Correctness Proofs

The correctness proofs of the program transformations presented in this section are based on showing that some invariant holds between computation prefixes of the subject program and the transformed program. The invariant is required to imply program equivalence and is proven to hold by induction on the length of the prefixes.

5.1 A Method for Showing Program Equivalence

Lemma 5.1 *Suppose $\pi \in pgm$ and $Apply(\pi, \pi', p, T)$ holds for some label p and some transformation T. The two programs are equivalent, $[\![\pi]\!] = [\![\pi']\!]$, if there exists a relation $\mathcal{R}$ between computation prefixes of π and π' such that the following holds for any input v and any two computation prefixes $C \in \mathcal{T}_{pfx}(\pi, v)$ and $C' \in \mathcal{T}_{pfx}(\pi', v)$,*

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

(i) *(Base)* $((\pi, v \vdash (p_0, \sigma_0)), (\pi', v \vdash (p'_0, \sigma'_0))) \in \mathcal{R}$.

(ii) *(Step) If* $(p_t, \sigma_t) \rightarrow (p_{t+1}, \sigma_{t+1})$ *and* $(p'_t, \sigma'_t) \rightarrow (p'_{t+1}, \sigma'_{t+1})$ *then* $C\mathcal{R}C' \Rightarrow C_2\mathcal{R}C'_2$ *where*

$$C_2 = \pi, v \vdash (p_0, \sigma_0) \rightarrow \ldots \rightarrow (p_{t+1}, \sigma_{t+1})$$
$$C'_2 = \pi', v \vdash (p'_0, \sigma'_0) \rightarrow \ldots \rightarrow (p'_{t+1}, \sigma'_{t+1}).$$

(iii) *(Equivalence) If* $C\mathcal{R}C'$ *then* $p_t = exit(\pi) \Leftrightarrow p'_t = exit(\pi')$ *and* $p_t = exit(\pi) \wedge p'_t = exit(\pi') \Rightarrow \sigma_t(\mathtt{y}) = \sigma'_t(\mathtt{y})$.

Proof. By induction over the two sets of computation prefixes, $C\mathcal{R}C'$ holds for any two computation prefixes of the same length. Suppose π does *not* terminate on input v, then $p_i \neq exit(\pi)$ for all $i > 0$, so $p'_i \neq exit(\pi')$ by assumption, implying that π' does not terminate on input v. Conversely nontermination of π' implies nontermination of π. Otherwise π terminates on input v. Let t be given such that $p_t = exit(\pi)$, which implies that $p'_t = exit(\pi')$ and $\sigma_t(\mathtt{y}) = \sigma'_t(\mathtt{y})$, thus $[\![\pi]\!](v) = [\![\pi']\!](v)$. Since v was an arbitrary value $[\![\pi]\!] = [\![\pi']\!]$. □

5.2 Elimination of Recomputation of Available Expressions

The elimination of recomputation of available expressions transformation detects assignments of expressions that are already available in some variable, and replaces occurrences of the expression by the variable.

Definition 5.2 *The following conditional rewrite rule defines the "elimination of recomputation of available expressions" transformation (ER),*

$$\mathbf{x}{:=}\mathbf{e} \Rightarrow \mathbf{x}{:=}\mathbf{z} \text{ if } \neg use(\mathbf{z}) \wedge \overleftarrow{A} X \overleftarrow{A} (\neg def(\mathbf{z}) \wedge trans(\mathbf{e}) \ W \ stmt(\mathbf{z}{:=}\mathbf{e})).$$

Informally the side condition states:

(i) The variable **z** may not occur in the expression **e**, since $\neg use(\mathbf{z})$ should be satisfied. If it was case, the expression **e** would (potentially) evaluate to a different value in the store after the assignment to variable **z**.

(ii) The second part of the side condition states that the value of **z** is unchanged for all backwards paths from the current label back to the statement **z**:=**e** (which must eventually be reached in any flow-legal computation). Furthermore any program variable occurring in the expression **e** does not change its value on the path. This implies that **e** evaluates to the same value at label where the transformation takes place and at the **z**:=**e** statement.

(iii) The side condition implicitly states that the CTL-FV variables **x** and **z** are bound to program variables, since they appear on the left hand side of assignments. The CTL-FV variable **e** must be bound to some expression, since it appears on the right hand side of assignments.

Theorem 5.3 *Suppose* $\pi, \pi' \in pgm$ *and* $Apply(\pi, \pi', p, ER)$ *then* $[\![\pi]\!] = [\![\pi']\!]$.

Proof. Let $\pi, \pi' \in pgm$, such that $Apply(\pi, \pi', p, ER)$ holds for some substitution θ and let $v \in Value$ be the input to the programs π and π'. In order to prove semantic equivalence, we will to establish that the *finest equivalence relation* on computation prefixes satisfies condition (i)-(iii) of Lemma 5.1. For the remainder of the proof we will assume that the transformation rule has been instantiated using the substitution θ.

Let $C \in \mathcal{T}_{pfx}(\pi, v)$, $C' \in \mathcal{T}_{pfx}(\pi', v)$ be program prefixes of the same length:

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

Base case

By inspecting the rewrite rule, it is observed that the transformation preserves all left hand sides of assignments, implying $vars(\pi) = vars(\pi')$. It then follows by the definition of the initial state that $(p_0, \sigma_0) = (p'_0, \sigma'_0)$.

Induction step

Since the semantics is deterministic there exists exactly one state (p_{t+1}, σ_{t+1}) and one state $(p'_{t+1}, \sigma'_{t+1})$, such that $(p_t, \sigma_t) \rightarrow (p_{t+1}, \sigma_{t+1})$ and $(p'_t, \sigma'_t) \rightarrow$

$(p'_{t+1}, \sigma'_{t+1})$. Assuming that $(p_i, \sigma_i) = (p'_i, \sigma'_i)$ for all $i \leq t$, we need to show that $(p_{t+1}, \sigma_{t+1}) = (p'_{t+1}, \sigma'_{t+1})$.

If $p_t \neq p$ then $\mathcal{I}_{p_t} = \mathcal{I}'_{p_t}$, so the claim follows by the semantics and the induction assumption.

Otherwise $p_t = p$, so $\mathcal{I}_{p_t} = (x{:=}e)$ and $\mathcal{I}'_{p_t} = (x{:=}z)$. By Lemma 3.4: $\sigma_{t+1} \setminus x = \sigma'_{t+1} \setminus x$, so the claim $\sigma_{t+1} = \sigma'_{t+1}$ holds if we can prove $\sigma_{t+1}(x) = \sigma'_{t+1}(x)$. The side condition must hold at p:

$$\mathcal{M}_{cf}(\pi), p \models_\theta \overleftarrow{A} X \overleftarrow{A} (\neg def(z) \wedge trans(e)\ W\ stmt(z{:=}e)).$$

Since any backwards path in the control flow model $\mathcal{M}_{cf}(\pi)$ is finite, it follows from the definition of CTL-FV that for any *finite* and computationally legal backwards path $(p = n_0 \leftarrow n_1 \leftarrow \ldots) \in S^{\text{MAX}}$ there exists an $1 \leq i < t$ such that

$$n_i \models_\theta stmt(z{:=}e)\ \wedge \forall 1 \leq j < i : n_j \models_\theta \neg def(z) \wedge trans(e).$$

The control flow model safely abstracts any possible computation – Lemma 3.6 – thus the above applies for computation prefix C, i.e. there exists an $1 \leq i < t$ such that:

$$\mathcal{I}_{p_i} = (z{:=}e)\ \wedge \forall i < j \leq t : def(z) \notin V(p_j)\ \wedge \forall y \in vars(e) : def(y) \notin V(p_j).$$

The rewrite rule does not change *trans* atomic propositions, so $trans(e) \in V(p_j)$ must also hold for C' for $i < j \leq t$. Similarly $p_i \models def(z)$ holds for C'. The side condition states that $use(z) \notin V(p_t)$, implying $z \notin vars(e)$, so the property $trans(e) \in V(p_j)$ must hold for $i = j$ as well, since $\mathcal{I}_i = (z{:=}e)$. By Lemma 3.5 we conclude $\sigma'_t|_{vars(e)} = \sigma'_i|_{vars(e)}$, that is: the expression e evaluates to the same value in the stores σ'_t and σ'_i. A similar argument shows that $\sigma'_{i+1}(z) = \sigma'_t(z)$. Putting the pieces together:

$$\begin{aligned}
\sigma_{t+1}(x) &= [\![e]\!]\sigma_t && \text{(semantics of } \mathcal{I}_{p_t} = (x{:=}e)) \\
&= [\![e]\!]\sigma_t|_{vars(e)} && \text{(follows from Definition 2.2)} \\
&= [\![e]\!]\sigma'_t|_{vars(e)} && \text{(induction assumption } \sigma_t = \sigma'_t) \\
&= [\![e]\!]\sigma'_i|_{vars(e)} && (1^{st} \text{ argument above}) \\
&= \sigma'_{i+1}(z) && \text{(semantics of } \mathcal{I}'_{p_t} = (z{:=}e)) \\
&= \sigma'_t(z) && (2^{nd} \text{ argument above}) \\
&= \sigma'_{t+1}(x) && \text{(semantics of } \mathcal{I}'_{p_t} = (x{:=}z))
\end{aligned}$$

Thus $\sigma_{t+1}(x) = \sigma'_{t+1}(x)$ so $\sigma_{t+1} = \sigma'_{t+1}$ which proves the induction step.

Equivalence

Clearly equivalence of computation prefixes implies program equivalence, so by Lemma 5.1 the transformation is semantics preserving: $[\![\pi]\!] = [\![\pi']\!]$. □

5.3 Loop Invariant Hoisting

As an example of what can be proven correct in this framework, the loop invariant hoisting transformation is presented without proof. The complete proof can be found in [6].

Definition 5.4 *A restricted version of a "code motion" transformation (CM) that covers the "Loop invariant hoisting" transformation is defined as*

$$
\begin{array}{ll}
 & \mathbf{p} : \texttt{skip} \Rightarrow \mathbf{x}{:=}\mathbf{e} \\
 & \mathbf{q} : \mathbf{x}{:=}\mathbf{e} \Rightarrow \texttt{skip} \\
\textit{if} & \mathbf{p} \models A(\neg \mathit{use}(\mathbf{x})\ W\ \mathit{node}(\mathbf{q})) \\
 & \mathbf{q} \models \neg \mathit{use}(\mathbf{x}) \wedge \overleftarrow{A} X \overleftarrow{A}((\neg \mathit{def}(\mathbf{x}) \vee \mathit{node}(\mathbf{q})) \wedge \mathit{trans}(\mathbf{e})\ W\ \mathit{node}(\mathbf{p})).
\end{array}
$$

This transformation involves two (different) statements in the subject program, so explicit labeling is required in the specification. The transformation moves an assignment at label **q** to label **p** provided that the following holds:

(i) The assigned variable **x** is dead after **p** until **q** (potentially) is reached (side condition for **p**). If this requirement holds, then introducing the assignment **x**:=**e** at label **p** will not change the semantics of the program.

(ii) The variable **x** is not used in the expression **e** (implied by the satisfaction of $\mathbf{q} \models_\theta \neg \mathit{use}(\mathbf{x})$).

(iii) The expression **e** evaluates to the same value at **p** and **q**, and that the definition of **x** reaches **q** after the transformation (implied by the side condition on **q**).

In order to prove this transformation correct, a more complex relation between the computation prefixes is needed. In particular, we need to capture state information in the relation. The definition of the prefix relation closely follows the CTL-FV clauses in the side condition. The relation keeps track of whether p or q (or neither) has been seen last in the two prefixes, which must agree on the labels.

Definition 5.5 *Suppose* $C \in \mathcal{T}_{pfx}(\pi, v)$ *and* $C' \in \mathcal{T}_{pfx}(\pi', v)$ *for some* π, π' *and* v *then define the CM relation on computation prefixes as:* $C \mathcal{R} C'$ *iff* $t = t'$, $p_i = p'_i$ *for all* $0 \leq i \leq t$ *and one of the following cases holds:*

(i) $\sigma_t = \sigma'_t \ \wedge\ \forall 0 \leq i \leq t : p_i \notin \{\mathbf{p}, \mathbf{q}\}$

(ii) $\sigma_t = \sigma'_t \ \wedge\ \exists 0 \leq i < t : p_i = \mathbf{q} \ \wedge\ \sigma_i = \sigma'_i \wedge \forall i < j < t : p_j \notin \{\mathbf{p}, \mathbf{q}\}$

(iii) $\sigma_t \setminus x = \sigma'_t \setminus x \wedge \exists 0 \leq i \leq t : p_i = \mathbf{p} \ \wedge$
$\sigma_i \setminus x = \sigma'_i \setminus x \ \wedge\ \forall i \leq j \leq t : p_j \notin \{\mathbf{p}, \mathbf{q}\}$

Theorem 5.6 *Suppose* $\pi, \pi' \in \mathit{pgm}$, $\mathit{Apply}(\pi, \pi', (\mathbf{p}, \mathbf{q}), \mathit{CM})$ *then* $[\![\pi]\!] = [\![\pi']\!]$.

Proof. Omitted, see [6].

6 Summary

In this paper, we have presented a framework for describing program transformations as rewrite rules with temporal logic formulae as side conditions. The use of temporal logic plays a central role in the correctness proofs – the side condition corresponds in an essential way to a program analysis. Using temporal logic to reason about program transformation is not new, but coupling rewrite rules with temporal logic conditions is [6,7].

So far the only creative part of the correctness proofs has been to find a suitable invariant between computation prefixes of the subject program and the transformed program. The remainder of the proofs are fairly straight forward and seem well suited for theorem proving. An interesting direction for future work would be to investigate whether soundness of transformation specifications is decidable and whether it is possible to find a *weakest* side condition such that a given transformation is sound.

The transformation specification language presented is fairly simplistic, and would have to be extended in order to express more involved program optimizations. An obvious extension would be the apply abstract interpretation to capture a more precise description (model) of the subject program. Other interesting extensions include languages that allow rearranging blocks of code or transformations with a variable number of rewrites. It is anticipated that the framework can be used to validate many of the traditional transformations used in optimizing compilers [1,8].

A shortcoming of the framework is the inability to insert and delete statements, e.g.. in the code motion transformation it would be preferable to simply *move* the statement instead of relying on the existence of `skip` instruction at "the right" places. The framework can be relatively easily be extended to handle this, but it complicates the presentation without providing any significant insights.

The framework is a step towards obtaining declarative methods for describing program transformations for imperative programs. Besides systematizing correctness proofs, the transformation specifications are computable, i.e. they can (once proven correct) be automatically and safely utilized by an optimizing compiler. This brings the *specification* of program optimizations closer to the *implementation*, increasing the confidence in the correctness of a given compiler implementation.

Acknowledgement

The author wishes to thank Eric Van Wyk, David Lacey and Neil D. Jones for their hard work during the joint effort to develop the framework in [6], on which the present paper is based. The author also extends his gratitude to the anonymous reviewers for their numerous and helpful suggestions for improvements.

References

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

[2] Huth, M.R.A, and M.D. Ryan, "Logic in Computer Science: Modelling and reasoning about systems", Cambridge University Press, 1999.

[3] Katoen, J., *Concepts, Algorithms and Tools for Model Checking* Lecture Notes of the Course "Mechanised Validation of Parallel Systems", Friedrich-Alexander Universität Erlangen-Nürnberg, 1998/1999.

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

[5] Kozen, D. 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., *Proceeding of the 1st International Conf. on Computational Logic* (CL2000), Lecture Notes in Artificial Intelligence, volume 1861, Springer-Verlag, London, July 2000, pp 568-582.

[6] Lacey, D., N.D. Jones, E. Van Wyk and C.C. Frederiksen, *Proving Correctness of Compiler Optimizations by Temporal Logic*, To appear in POPL2002.

[7] Lacey, D., 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*, pp 52-68. Springer-Verlag, 2001.

[8] Muchnick, S.S., "Advanced Compiler Design and Implementation", Morgan Kaufmann, 1997.

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

[10] de Moor, O., David Lacey and E. Van Wyk, *Universal Regular Path Queries*, (Submitted to HOSC), 2001.

[11] Pinter, S.S. and P. Wolper *A Temporal Logic for Reasoning About Partially Ordered Computations* In *Proc. of the 3rd ACM Symposium on Principles of Distributed Computing*, pp 28-37, 1984.

[12] 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.

[13] Schmidt, D.A. and 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.