Correctness of Classical Compiler Optimizations using CTL

Carl Christian Frederiksen

July 2, 2001

Contents

1 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 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.

This paper is based on the as yet unpublished ideas of David Lacey, Neil Jones and Eric Van Wyk[4]. The aim is to prove correct some conditional rewrite rules corresponding to classical compiler optimizations, beyond those considered by Lacey, et al.

2 Introduction

This paper presents a framework for specifying program transformations.

Programs are regarded as being effective 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 have the same termination properties and if one of the programs terminate for some input v, then both programs terminate on input v, outputting the same value – i.e. the two programs compute the same partial function.

The main focus is on the *correctness* of program transformations, in the sense that the subject program and the transformed program are semantically equivalent. The framework in which the transformations are specified is computable, i.e. given a subject program and a transformation rule it is possible to automatically compute the transformed program resulting from the application of the transformation.

The transformation rules (conditional rewrite rules) described in the paper are transformations that a compiler can utilize in order to obtain a more efficient target program. Other issues such as *optimality*, *efficiency* and *generality* are not considered in detail. Thus the paper will not describe the transformation strategy (i.e. the sequence in which the program transformations are applied).

The approach to applying a program transformation to a subject program:

1. Instantiate an applicable rule schema to a concrete transformation rule. For example, from the constant propagation transformation, we might obtain the following rewrite rule.

   $$\texttt{x} := \texttt{y} \Rightarrow \texttt{x} := \texttt{0}$$

   assuming a side condition: `x` is not assigned until the assignment `y:=0` is reached, following all possible backwards paths in the control flow graph of the program.

2. Verify the side condition. The side condition is specified using CTL-R – an extension of CTL – so verification amounts to model checking.

3. Construct the transformed program.

In an actual implementation, the set of substitutions Θ such that the side condition holds can be found by model checking. Each CTL-R formula is model checked and a set of answer substitutions is found. By unification, a set of substitutions that satisfies all CTL-R formulae in the side condition can be obtained. Goal: All substitutions in the resulting set satisfy the entire side condition, and each one gives rise to a sound transformation, provided that the transformation rule was sound to begin with.

2.1 Overview of the remainder of the paper

Section 3 introduces a small imperative language without procedure calls or exceptions and the notion of a *computation prefix* is defined – a computation prefix is the sequence of program states that is computed when executing the program for a given input value. The computation prefixes will be a key factor in proving semantic program equivalence for the transformations. Section 4 introduces an extension of CTL (CTL-R) that will be used for specifying the side conditions in the transformation rules. In Section 5 the transformation rules are introduced. In order to keep the presentation simple, the framework handles only transformations involving one statement being transformed. The application of a transformation rule to a program is formally described as well. Section 6 investigates two transformation rules: *copy propagation* and *elimination of recomputation of available expressions*. Both transformations are proven correct. Next in Section 7 some auxiliary transformations are proven correct outside the framework. In particular the transformations `skip` *insertion*, `skip` *removal* and *dead assignment insertion* are proven correct. From these and the transformations from Section 6, the *elimination of common subexpressions* transformation is easily obtained and the correctness follows from the correctness of the other transformations. Section 8 extends the framework to the case with several statements being affected by the transformation. The development is analogous to the single statement case. As an example, the *loop invariant hoisting* transformation is specified and proven correct. Finally Section 9 gives a summary of the work presented.

3 Language

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

$$\pi = \texttt{read x};\ l_1 : \mathcal{I}_{l_1};\ \ldots;\ l_m : \mathcal{I}_{l_m};\ l_{m+1} : \texttt{write y};$$

Each statement, except the `read` statement, is labeled by a unique *label* ranging over the set of labels: *Label* which is not further specified. The set of labels occurring in the program π is denoted $labels(\pi)$. The label l_1 at the first statement in the program after the `read` statement is denoted $entry(\pi)$, and the label l_{m+1} at the `write` statement is denoted $exit(\pi)$.

Syntax. The syntax of a program is given below.

$$
\begin{array}{lcl}
\pi \in \mathit{pgm} & ::= & \texttt{read x}; \mathit{stmts}; p : \texttt{write y}; \\
\mathit{stmts} & ::= & p : \mathit{stmt}; \mathit{stmts} \mid \epsilon \\
\mathcal{I} \in \mathit{stmt} & ::= & x := \mathit{exp} \mid \texttt{if } x \texttt{ then } p_1 \texttt{ else } p_2 \mid \texttt{skip} \\
e \in \mathit{exp} & ::= & c \mid x \mid \texttt{op}(\mathit{exp}_1, \ldots, \mathit{exp}_n) \\
x & \in & \mathit{Variable} \\
p & \in & \mathit{Label} \\
c & \in & \mathit{Value}
\end{array}
$$

There are three possible statements that can occur between the `read` statement and the `write` statement: assignments, conditionals and `skip` statements. Note that conditionals are not allowed to perform tests directly on expressions, but only on variables. This is not a serious restriction since we can compute the expression, assign it to a variable that does not occur in the program and use that variable in the test instead. All labels in the program are assumed to be unique, and any label referred in a conditional in the program must occur in the program. Note that the `read` statement always uses the variable `x` and the `write` statement always uses the variable `y`. The statement occurring at label p is referred to as $\mathcal{I}_p$. The set of variables occurring in the program is denoted $\mathit{vars}(\pi)$. Binary operators are written in infix notation in the example programs, but should be interpreted as having the form given in the syntax definition.

3.1 Semantics

In this section the semantics is defined. The domain of values *Value*, the operations on it *Op* and the interpretation of the operators $[\![\cdot]\!]_{op} : \mathit{Value}^* \to \mathit{Value}$ are left unspecified. However it is assumed that there exists a distinguished element *true* of *Value*.

Next label function. Suppose $\pi \in \mathit{pgm}$ is a program and suppose $\prec$ is a total ordering on *Label* such that $p \prec p'$ if and only if p appears syntactically before p' in the program. The *next-label function* $Nx_\pi : \mathit{Label} \to \mathit{Label}$ is defined by:

$$Nx_\pi(p) = min_\prec\{p' \in \mathit{Label} \mid p \prec p'\}.$$

The next label function is a device for getting from a statement to the next label in the program. The next label function Nx_π takes a label occurring in the program π and returns the label occurring immediately syntactically after, i.e. the "smallest" label that is "greater than" the given label with respect to the ordering $\prec$ on the labels. Since each statement in labeled by precisely one statement, the next label function is well-defined in the domain $\mathit{Label} \setminus \{\mathit{exit}(\pi)\}$.

Store. 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 $\mathit{dom}(\sigma) = X \subseteq \mathit{Variable}$:

1. $\sigma[x \mapsto v]$ is the extension of σ to $X \cup \{x\}$ such that $\sigma[x \mapsto v](x) = v$ and $\forall y \in X \setminus \{x\} : \sigma[x \mapsto v](y) = \sigma(y)$.
2. $\sigma|_{X'}$ denotes the restriction of σ to the domain $X' \subseteq X$, $\sigma|_{X'} : X' \to \mathit{Value}$ where $\forall x \in X' : \sigma(x) = \sigma|_{X'}(x)$.
3. $\sigma \setminus x = \sigma|_{X \setminus \{x\}}$ denotes σ restricted to its domain excluding x.

Expression evaluation. Expression evaluation is defined by the function

$$[\![\cdot]\!]\cdot : \mathit{Expr} \to \mathit{Sto} \to \mathit{Value}.$$

$$\begin{array}{rcl} [\![c]\!]\sigma & = & c \\ [\![\mathtt{x}]\!]\sigma & = & \sigma(\mathtt{x}) \\ [\![\mathtt{op}(exp_1, \ldots, exp_n)]\!]\sigma & = & [\![\mathtt{op}]\!]_{op}([\![exp_1]\!]\sigma, \ldots, [\![exp_n]\!]\sigma) \end{array}$$

The following lemma states that when evaluating an expression, the result depends only upon the variables occurring the expression, and their bindings.

Lemma 1. *Suppose* $\mathtt{E} \in \mathit{Exp}$ *and* $\sigma \in \mathit{Sto}$ *then* $[\![\mathtt{E}]\!]\sigma = [\![\mathtt{E}]\!]\sigma|_{vars(\mathtt{E})}$

Program state. A *program state* for a given program π, is a pair (p, σ) where p is a label in $\mathit{labels}(\pi)$ and σ is a store such that $\mathit{vars}(\pi) \subseteq \mathit{dom}(\sigma)$. The set of all program states is denoted *PgmState*.

Semantic transition relation. Let the semantic transition relation $\to \subseteq \mathit{PgmState} \times \mathit{PgmState}$ for a program $\pi \in \mathit{pgm}$ be defined by:

1. If $\mathcal{I}_p = \mathtt{skip}$ then $(p, \sigma) \to (Nx_\pi(p), \sigma)$.
2. If $\mathcal{I}_p = (\mathtt{x} := exp)$ then $(p, \sigma) \to (Nx_\pi(p), \sigma[\mathtt{x} \mapsto [\![exp]\!]\sigma])$.
3. If $\mathcal{I}_p = (\mathtt{if\ x\ then}\ p_1\ \mathtt{else}\ p_2)$ and $([\![\mathtt{x}]\!]\sigma = \mathtt{true})$ then $(p, \sigma) \to (p_1, \sigma)$
4. If $\mathcal{I}_p = (\mathtt{if\ x\ then}\ p_1\ \mathtt{else}\ p_2)$ and $([\![\mathtt{x}]\!]\sigma \neq \mathtt{true})$ then $(p, \sigma) \to (p_2, \sigma)$
5. $(\mathit{exit}(\pi), \sigma) \to (\mathit{exit}(\pi), \sigma)$.

Corollary 1. *The semantic transition relation is deterministic.*

Proof. Immediate from the definition since only one of the clauses in the definition can hold for a state for a given program. □

Initial state. The initial state $in_\pi(v) \in \mathit{PgmState}$ for a program $\pi \in \mathit{pgm}$ and input $v \in \mathit{Value}$ is defined by

$$in_\pi(v) = (\mathit{entry}(\pi), [\mathtt{x} \mapsto v, y_1 \mapsto \mathit{true}, \ldots, y_k \mapsto \mathit{true}])$$

where $\mathit{vars}(\pi) \setminus \{\mathtt{x}\} = \{y_1, \ldots, y_k\}$. 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*.

Sequence notation. Let S be some arbitrary set. The set of sequences of finite length over S is denoted S^*. The set of infinite sequences over S is denoted S^ω and the set of sequences of finite or infinite length is denoted $S^{*\omega}$.

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

$$(p_0, \sigma_0) \to (p_1, \sigma_1) \to \ldots$$

such that $(p_0, \sigma_0) = in_\pi(v)$ and $(p_i, \sigma_i) \to (p_{i+1}, \sigma_{i+1})$ for all i.

The above notation is used when the program and program input is clear from the context. To explicitly specify the program and input, computation prefixes are written:

$$\pi, v \vdash (p_0, \sigma_0) \to (p_1, \sigma_1) \to \ldots$$

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

Next prefix relation. Suppose $\pi \in pgm$ is a program, $v \in Value$ is the input and $C \in \mathcal{T}_{pfx}(\pi, v)$ is a finite computation prefix such that

$$C = (p_0, \sigma_0) \to \ldots \to (p_n, \sigma_n).$$

The *next prefix relation* is a relation $\to \subseteq \mathcal{T}_{pfx}(\pi, v) \times \mathcal{T}_{pfx}(\pi, v)$ where $C \to C'$ for a finite computation prefix $C' \in \mathcal{T}_{pfx}(\pi, v)$ if and only if C' has the form

$$C' = (p_0, \sigma_0) \to \ldots \to (p_n, \sigma_n) \to (p_{n+1}, \sigma_{n+1}).$$

Program semantics. Suppose $\pi \in pgm$ is a program. The partial meaning function $[\![\cdot]\!]\cdot : pgm \to Value \rightharpoonup Value$ is defined as $[\![\pi]\!]v = \sigma_k(\mathrm{y})$ if there exists a computation prefix $C = \pi, v \vdash (p_1, \sigma_1) \to \ldots \to (p_k, \sigma_k)$ such that $p_k = exit(\pi)$.

4 Temporal logic

The basic idea is to express a program transformation rule as a number of rewrite rules each with a side condition specified in a variant of CTL. The use of temporal logic allows expressing 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.

$$
\begin{array}{lll}
\mathcal{M}, n \models true & \text{iff} & true \\
\mathcal{M}, n \models false & \text{iff} & false \\
\mathcal{M}, n \models ap & \text{iff} & ap \in V(n) \\
\mathcal{M}, n \models \neg\phi & \text{iff} & \neg\mathcal{M}, n \models \phi \\
\mathcal{M}, n \models \phi_1 \wedge \phi_2 & \text{iff} & \mathcal{M}, n \models \phi_1 \wedge \mathcal{M}, n \models \phi_2 \\
\mathcal{M}, n \models A\psi & \text{iff} & \forall (n = n_0 \rightarrow n_1 \rightarrow \ldots) \in S^\omega : \mathcal{M}, (n_i)_{i \geq 0} \models \psi \\
\mathcal{M}, n \models E\psi & \text{iff} & \exists (n = n_0 \rightarrow n_1 \rightarrow \ldots) \in S^\omega : \mathcal{M}, (n_i)_{i \geq 0} \models \psi \\
 & & \\
\mathcal{M}, (n_i)_{i \geq 0} \models X(\phi) & \text{iff} & n_1 \models \phi \\
\mathcal{M}, (n_i)_{i \geq 0} \models \phi_1 U \phi_2 & \text{iff} & \exists i \geq 0 : \mathcal{M}, n_i \models \phi_2 \wedge \forall 0 \leq j < i : \mathcal{M}, n_j \models \phi_1
\end{array}
$$

Figure 4.1: Semantics of CTL.

4.1 CTL

First the ordinary definition of CTL is briefly presented.

CTL Model. A *CTL model* is a triple $\mathcal{M} = (S, \rightarrow, V)$ where S is a set of states, $\rightarrow \subseteq S \times S$ is a *total* relation 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. The atomic propositions are "facts" that can hold for any given state in the model.

A *path* in a CTL model is an *infinite* sequence of states $(n_i)_{i \geq 0} \in S^\omega$, such that $n_i \rightarrow n_{i+1}$ for all $i \geq 0$.

CTL syntax. The syntax of CTL is given below. A ϕ formula is called a *state formula* since ϕ formulae express properties over a single state in the model. Similarly a ψ formula is called a *path formula*.

$$
\begin{array}{lll}
\phi & ::= & true \mid false \mid ap \mid \neg\phi \mid \phi_1 \wedge \phi_2 \\
 & & \mid A\psi \mid E\psi \\
\psi & ::= & X\phi \mid \phi_1 U \phi_2 \\
ap & \in & AP
\end{array}
$$

CTL satisfaction. A CTL *clause* is written $\mathcal{M}, n \models \phi$ where $\mathcal{M}$ is a CTL model, n is a state in the model and ϕ is a CTL formula. If the model is clear from the context, the $\mathcal{M}$ can be omitted from the clause. A CTL clause is *satisfied* if and only if the clause evaluates to true by the semantics defined in Figure 4.1.

It is crucial to note that a given atomic proposition does not hold invariantly true or false for the entire model. The "truth" of a atomic proposition depends on the state for which it is evaluated. In the definition of CTL there are two path quantifiers forall (A) that asserts some property over all paths in the CTL model starting with the current state, and exists (E). The path operator next

(X) simply asserts a state property for the next state in the path. The until (U) path operator, asserts some property ϕ_1 that must hold for all states in the path until eventually a state is reached for which the ϕ_2 property holds. The until operator does not assert anything for the remainder of the path.

It is possible to define a number of derived operators that can be used to write the CTL formulae more succinctly. Since the paper uses an extension of CTL, the derived operators will be introduced after the extension, as it subsumes ordinary CTL.

4.2 CTL-R

We present an extension to CTL called CTL-R, in order to express side conditions for the transformation rules. The extension is three 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 $\rightarrow$ relation. 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. This is done by introducing two new versions of exists (E) and forall (A): $\overleftarrow{E}$ and $\overleftarrow{A}$. For the interpretation of the "past" quantifiers the subformulae must hold over backwards paths, i.e. following the $\leftarrow$ relation. Since the definition of a CTL model only requires the $\rightarrow$ relation to total, any well-defined CTL model might not be a well-defined CTL-R model. We handle this by dropping the totality requirement on the $\rightarrow$ relation, but at the same time we must specify how finite paths are handled.

CTL-R Model. A *CTL-R model* is a triple $\mathcal{M} = (S, \rightarrow, V)$ where S is a set of states, $\rightarrow \subseteq S \times S$ 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 will be extended to handle models with "dead states", states with no successor following the direction of the path. Since the definition is 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.

Maximal 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 $(n_i)_{0\leq i\leq k}$ and n_k is *dead* meaning:

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

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

$$
\begin{array}{lll}
\mathcal{M}, n \models true & \text{iff} & true \\
\mathcal{M}, n \models false & \text{iff} & false \\
\mathcal{M}, n \models ap(x_1, \ldots, x_n) & \text{iff} & ap(x_1, \ldots, x_n) \in V(n) \\
\mathcal{M}, n \models \neg\phi & \text{iff} & \text{not } \mathcal{M}, n \models \phi \\
\mathcal{M}, n \models \phi_1 \wedge \phi_2 & \text{iff} & \mathcal{M}, n \models \phi_1 \text{ and } \mathcal{M}, n \models \phi_2 \\
\mathcal{M}, n \models A\psi & \text{iff} & \forall (n = n_0 \rightarrow n_1 \rightarrow \ldots) \in S^{\text{MAX}} : \mathcal{M}, (n_i)_{i \geq 0} \models \psi \\
\mathcal{M}, n \models \overleftarrow{A}\psi & \text{iff} & \forall (\ldots \rightarrow n_1 \rightarrow n_0 = n) \in S^{\text{MAX}} : \mathcal{M}, (n_i)_{i \geq 0} \models \psi \\
\mathcal{M}, n \models E\psi & \text{iff} & \exists (n = n_0 \rightarrow n_1 \rightarrow \ldots) \in S^{\text{MAX}} : \mathcal{M}, (n_i)_{i \geq 0} \models \psi \\
\mathcal{M}, n \models \overleftarrow{E}\psi & \text{iff} & \exists (\ldots \rightarrow n_1 \rightarrow n_0 = n) \in S^{\text{MAX}} : \mathcal{M}, (n_i)_{i \geq 0} \models \psi \\
 & & \\
\mathcal{M}, (n_i)_{i \geq 0} \models X(\phi) & \text{iff} & n_1 \models \phi \\
\mathcal{M}, (n_i)_{i \geq 0} \models \phi_1 U \phi_2 & \text{iff} & \exists i \geq 0 : \mathcal{M}, n_i \models \phi_2 \wedge \forall 0 \leq j < i : \mathcal{M}, n_j \models \phi_1
\end{array}
$$

Figure 4.2: Semantics of CTL-R .

Third, in order to be able to express general transformation descriptions, a finer structure is introduced on the atomic propositions. In particular instead of considering the elements of AP as "tokens", we will regard the set of atomic propositions as a set of terms.

CTL-R syntax. The syntax of CTL-R is given below.

$$
\begin{array}{lll}
\phi & ::= & true \mid false \mid ap(x_1, \ldots, x_n) \mid \neg\phi \mid \phi_1 \wedge \phi_2 \\
 & & \mid A\psi \mid E\psi \mid \overleftarrow{A}\psi \mid \overleftarrow{E}\psi \\
\psi & ::= & X\phi \mid \phi_1 U \phi_2
\end{array}
$$

CTL-R operators. Similarly to an ordinary CTL clause, a CTL-R clause is written $\mathcal{M}, n \models \phi$ where $\mathcal{M}$ is the CTL-R model, n is a state in the model and ϕ is a CTL-R formula. The clause is *satisfied* if and only if the clause evaluates to *true* under the definition of CTL-R satisfaction given in Figure 4.2. As before, if the model is clear from the context then $\mathcal{M}$ can be omitted in the clause.

Derived CTL-R operators. Let $\gamma \in \{A, E, \overleftarrow{A}, \overleftarrow{E}\}$. Then the following operators can be defined.

$$
\begin{array}{lll}
\mathcal{M}, n \models \gamma F(\phi) & \text{iff} & \mathcal{M}, n \models \gamma(true\ U\ \phi) \\
\mathcal{M}, n \models \gamma G(\phi) & \text{iff} & \mathcal{M}, n \models \neg(\gamma^{-1}) F(\neg\phi) \\
\mathcal{M}, n \models \gamma(\phi_1\ W\ \phi_2) & \text{iff} & \mathcal{M}, n \models \gamma(\phi_1\ U\ \phi_2) \vee \gamma G(\phi_1)
\end{array}
$$

$$
\begin{array}{l}
A^{-1} = E \\
E^{-1} = A \\
\overleftarrow{A}^{-1} = \overleftarrow{E} \\
\overleftarrow{E}^{-1} = \overleftarrow{A}
\end{array}
$$

Similarly we can derive the ordinary boolean operators $\vee$, $\Rightarrow$ etc.

4.3 CTL-R Models

In this section the models needed for the correctness proofs are presented.

4.3.1 Fine structure on AP

Since the CTL-R side condition is obtained by instantiating the free variables in the transformation, we define a finer structure on the atomic propositions. This admits reasoning about the instantiated parts of the program.

Structure on AP. The set of atomic propositions used in the models in this paper collects syntactic information about the subject program π. The set is easily computed in linear time by traversing the subject program.

$$\begin{array}{rcl} AP & ::= & node(Label) \\ & | & entry \\ & | & stmt(stmt) \\ & | & def(Variable) \\ & | & use(Variable) \\ & | & inexp(Variable, exp) \\ & | & isvar(Variable) \end{array}$$

Each label, statement, variable and expression occurring in an atomic proposition must occur in the subject program. 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 $V(p)$. As an example consider the following program.

```
     read x;
1 :  y := 5;
2 :  x := x + y;
3 :  write y;
```

The valuation for this program would be the following.

$$\begin{array}{lllll} V(1) \supseteq \{node(1), & entry, & stmt(\texttt{y} := 5), & def(\texttt{y}) & \} \\ V(2) \supseteq \{node(2), & & stmt(\texttt{x} := \texttt{x} + \texttt{y}), & def(\texttt{x}), \quad use(\texttt{x}), use(\texttt{y}) & \} \\ V(3) \supseteq \{node(3), & & stmt(\texttt{write y}), & use(\texttt{y}) & \} \end{array}$$

The remaining atomic propositions are independent of the label p:

$$V(p) \supseteq \{isvar(\texttt{x}), isvar(\texttt{y}), inexp(\texttt{x}, \texttt{x} + \texttt{y}), inexp(\texttt{y}, \texttt{x} + \texttt{y})\}.$$

4.3.2 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 program. The model is sufficiently detailed to express many of the classical compiler optimizations.

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

1. The set of states S is defined by $S = labels(\pi)$.
2. The state transition relation, $\rightarrow_{cf}$: $S \times S$, is defined as $p \rightarrow_{cf} p'$ if and only if

$$\begin{array}{rcll} \mathcal{I}_p = (\texttt{if } x \texttt{ then } p_1 \texttt{ else } p_2) & \wedge & p' \in \{p_1, p_2\} & \vee \\ (\mathcal{I}_p = (\texttt{skip}) \vee \mathcal{I}_p = (x{:=}e)) & \wedge & p' = Nx_\pi(p) & \vee \\ \mathcal{I}_p = (\texttt{write y}) & \wedge & p' = p. & \end{array}$$

3. The *valuation* $V : S \rightarrow 2^{AP}$ is defined as:

$$\begin{array}{rcl} node(q) \in V(p) & \Leftrightarrow & p = q \\ entry \in V(p) & \Leftrightarrow & p = entry(\pi) \\ stmt(\mathcal{I}) \in V(p) & \Leftrightarrow & \mathcal{I}_p = \mathcal{I} \\ def(x) \in V(p) & \Leftrightarrow & \exists e \in expr(\pi) : \mathcal{I}_p = (x{:=}e) \\ use(y) \in V(p) & \Leftrightarrow & \exists x \in vars(\pi), e \in expr(\pi), \exists p_1, p_2 \in labels(\pi) : \\ & & \mathcal{I}_p = (x{:=}e) \wedge y \in vars(e) \vee \\ & & \mathcal{I}_p = (\texttt{if } y \texttt{ then } p_1 \texttt{ else } p_2) \vee \\ & & \mathcal{I}_p = (\texttt{write } y) \\ inexp(x, e) \in V(p) & \Leftrightarrow & x \in vars(\pi) \wedge e \in expr(\pi) \wedge x \in vars(e) \\ isvar(x) \in V(p) & \Leftrightarrow & x \in vars(\pi) \end{array}$$

The control flow model can be extracted from the subject program by a simple pass over the program. The last two types of atomic propositions express some global program properties that are independent of the label. The global properties do not belong in a CTL-R formula, but in order to avoid formalizing the machinery to deal with them, they have been included in the set of atomic propositions.

Lemma 2. *Suppose* $(p_0, \sigma_0) \rightarrow \ldots \rightarrow (p_k, \sigma_k)$ *and* $\forall 0 \leq i < k : \forall x \in X : \mathcal{M}_{cf}(\pi), p_i \models \neg def(x)$ *then* $\sigma_0|_X = \sigma_k|_X$.

Lemma 3. *Suppose* $\pi \in pgm$, $\sigma_1 \in Sto$, $(p_1, \sigma_1) \rightarrow (p_2, \sigma_2)$ *and* $\mathcal{M}_{cf}(\pi), p_1 \models def(x)$ *then* $\sigma_1 \setminus \texttt{x} = \sigma_2 \setminus \texttt{x}$.

Proof. The statement at p_1 must have the form $x{:=}e$ for some $e \in exp$. By the semantics $\sigma_2 = \sigma_1[x \mapsto v]$ where $v = [\![e]\!]\sigma_1$. Clearly this implies $\sigma_1 \setminus x = \sigma_2 \setminus x$. □

Lemma 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* $p_1 \models \neg use(x)$ *then* $\sigma_2 \setminus x = \sigma_2' \setminus x$.

4.3.3 Computation prefix model

In order to prove correctness we define a model that directly corresponds to a given computation prefix of the subject program. The model relation for the computation prefix model is defined as the semantic transition relation. Thus no computation prefix model contains any branching. Intuitively a computation prefix model is simply a computation prefix interpreted as a CTL-R model.

Computation prefix model. Suppose $\pi \in pgm$ and $C \in \mathcal{T}_{pfx}(\pi, v)$ such that $C = \pi, v \vdash (p_0, \sigma_1) \rightarrow \ldots \rightarrow (p_t, \sigma_t)$ is a finite computation prefix. The computation prefix model for the prefix C is defined as

$$\mathcal{M}_{pfx}(C) = (\{(p_0, \sigma_1), \ldots, (p_t, \sigma_t)\}, \rightarrow, V)$$

where the valuation V is identical to the valuation for the control flow model.

Next we establish a lemma stating that any atomic proposition that holds for the control flow model of a given program will also hold for any computation trace.

Lemma 5. *Suppose $\pi \in pgm$ and $C \in \mathcal{T}_{pfx}(\pi, v)$ is a finite computation prefix for some value $v \in Value$,*

$$C = \pi, v \vdash (p_0, \sigma_0) \rightarrow \ldots \rightarrow (p_t, \sigma_t) \text{ and } p_t = exit(\pi).$$

For all $0 \leq i \leq t$, the CTL-R clause $\mathcal{M}_{cf}(\pi), p_i \models ap(x_1, \ldots, x_n)$ is satisfied if and only if $\mathcal{M}_{pfx}(C), (p_i, \sigma_i) \models ap(x_1, \ldots, x_n)$ is satisfied.

4.3.4 Backwards until

In the control flow model, the until operator asserts a rather strong statement; The CTL-R formula $\overleftarrow{A}(\phi_1 \; U \; \phi_2)$ states that, for all backwards paths ϕ_1 should holds until ϕ_2 holds. If ϕ_1 occurs in a loop then there exists a backwards path that loops infinitely. Thus the state where ϕ_2 holds must also occur within the loop. For the sake of handling more powerful transformations it would be desirable to be able to express that ϕ_1 holds until ϕ_2 holds, while allowing the path to "go out of loops". Thus a more relaxed version is presented: $\overleftarrow{A}(\phi_1 \wedge \neg entry \; W \; \phi_2)$. Infinite backward paths are admitted since they are unrealizable for any actual computation prefix. At the same time, any finite backwards path must eventually satisfy ϕ_2 since any backwards path in the computation prefix model ends in a state where *entry* holds.

Lemma 6. *Suppose $\pi \in pgm$ is a program and $C \in \mathcal{T}_{pfx}(\pi)$ is a computation prefix of π then $\mathcal{M}_{cf}, p_t \models \overleftarrow{A}(\phi_1 \wedge \neg entry \; W \; \phi_2)$ implies*

$$\forall finite\ paths : (\rightarrow n_1 \rightarrow n_0 = p_t) \in Label^*$$
$$\exists i \geq 0 : n_i \models \phi_2 \wedge \forall 0 \leq j < i : n_j \models \phi_1.$$

That is: $\overleftarrow{A}(\phi_1 \; U \; \phi_2)$ is satisfied for all finite paths in the control flow model.

5 Transformation

This section defines framework for describing a schema of concrete program transformations. 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 rules that the compiler can use to transform (optimize) compiled programs.

5.1 Transformation rule syntax

Free variables are allowed in the transformation rules. This allows for specification of transformation templates that intuitively can be instantiated (by binding the free variables) to give concrete a transformation rule.

Transformation rules. A *transformation rule* has the syntax:

$$
\begin{array}{lcl}
rw & ::= & vstmt_1 \Rightarrow vstmt_2 \text{ if } \phi \\
vstmt & ::= & v := vexp \mid \texttt{if } v \texttt{ then } \mathbf{v}_1 \texttt{ else } \mathbf{v}_2 \mid \texttt{skip} \\
vexp & ::= & c \mid x \mid \mathbf{v} \mid \texttt{op}(vexp_1, \ldots, vexp_n) \\
\phi & ::= & true \mid false \mid ap(\mathbf{v}_1, \ldots, \mathbf{v}_n) \mid \neg\phi \mid \phi_1 \wedge \phi_2 \\
 & & \mid \forall y \in Y(\mathbf{v}_1, \ldots, \mathbf{v}_n) : \phi \mid \exists y \in Y(\mathbf{v}_1, \ldots, \mathbf{v}_n) : \phi \\
 & & \mid A\psi \mid E\psi \mid \overleftarrow{A}\psi \mid \overleftarrow{E}\psi \\
\psi & ::= & X\phi \mid \phi_1 \; U \; \phi_2 \\
Y & ::= & vars \\
\mathbf{v} & \in & FormulaVariable \\
c & \in & Value \\
x & \in & Variable \\
y & \in & QuantifierVariable
\end{array}
$$

The rewrite rule "rw" may contain free variables ranging over substructures of statements (to be bound in the instantiation phase). The side conditions are expressed in CTL-R, extended with two constructs forall ($\forall$) and exists ($\exists$). The purpose of the two constructs is two allow reasoning about substructures of statements. The constructs will be removed by "unrolling" before any model checking is performed, thus the formula to be model checked is an ordinary CTL-R formula.

The functions Y are names for functions that, given an expression in the program generates a set of expressions that can be quantified over. An example could be: $\forall y \in vars(E) : use(y)$. Once E is instantiated to an expression, say $\texttt{a} + \texttt{b}$, the set $vars(E) = \{\texttt{a}, \texttt{b}\}$ can be computed and the construct can be "unrolled" to $use(\texttt{a}) \wedge use(\texttt{b})$. The functionality admits expressing transformations such as the elimination of recomputation of available expressions transformation, as will be demonstrated subsequently.

Applying a transformation rule to a given program is done in three steps.

1. All free variables in the transformation rule are *instantiated*, by matching a statement in the program with the left-hand side of the rewrite rule in the transformation rule. All remaining free variable are instantiated. Next the program dependent subformulae in the side condition are unrolled to give an ordinary CTL-R formula.
2. The side condition is *verified* by model checking the side condition at the label where the transformation rule was instantiated. If the clause holds, the instantiated transformation can be applied to the program, provided that the transformation rule is sound.

3. The new program is *constructed*, simply by replacing the statement in the program with the (instantiated) right-hand side of the rewrite rule.

5.2 Concrete transformation rules

Given a program and a transformation rule, we can instantiate the transformation rule at a given program point to obtain a *concrete* transformation rule. Intuitively a concrete transformation rule is obtained by instantiating the free variables in the rule and unrolling the program dependent subformulae in the side condition.

Closed transformation rule. Suppose $\pi \in pgm$ is a program, $p \in labels(\pi)$ is a label in π, T is a transformation rule $lhs \Rightarrow rhs$ if ϕ and θ is a substitution binding all variables not occurring in lhs. The purpose of θ is to define unreferenced variables, that are referenced in the side condition, but not in left hand side of the rewrite rule. Define θ' as the substitution obtained by unifying the left hand side of the transformation rule with the statement at label p the π, i.e. $\theta' = Unify(lhs, \mathcal{I}_p)$. Provided that there exists a θ' unifying lhs and $\mathcal{I}_p$, the transformation rule T' is obtained from T by closure with respect to $\theta'' = \theta \circ \theta'$ if $T' = \theta''(T)$ and T' contains no free variables. The transformation rule T' i called a *closed transformation rule*, since it contains no free variables. The type of θ'' is given by $\theta'' : FormulaVariables \rightarrow Label \cup stmt \cup exp \cup Variable$.

From a closed transformation rule, the *concrete* transformation rules can be obtained by "unrolling" the program quantifiers in the side condition.

Concrete transformation rule. If $lhs \Rightarrow rhs$ *if* ϕ is a closed transformation rule, then the *concrete transformation rule* is defined as $lhs \Rightarrow rhs$ *if* $unroll(\phi)$ where *unroll* is a function that recursively traverses a CTL-R formula and expands the forall and exists constructs. When a forall or exists formula is reached, the set that the formula quantifies over is computed. The resulting is set required to be finite for any input. The function $[\![\cdot]\!]$ maps the names of set extraction functions to their definitions, the syntactic elements Y are mapped via. function the $[\![\cdot]\!] : Y \rightarrow exp \rightarrow \mathcal{P}(exp)$ to set extraction functions.

As an example consider $\forall y \in vars(a + b) : use(y)$. Syntactic element *vars* is associated via. the $[\![\cdot]\!]$ with a function that extracts the variables in an expression, resulting in the set $\{a, b\}$. The function $[\![vars]\!] : exp \rightarrow \mathcal{P}(exp)$ recursively traverses an expression and extracts all variables occurring the the expression. The formula would then be unrolled to the CTL-R formula $use(a) \wedge use(b)$.

$$
\begin{array}{lcl}
unroll(true) & = & true \\
unroll(false) & = & false \\
unroll(ap(x_1,\ldots,x_n)) & = & ap(x_1,\ldots,x_n) \\
unroll(\neg\phi) & = & \neg unroll(\phi) \\
unroll(\phi_1 \wedge \phi_2) & = & unroll(\phi_1) \wedge unroll(\phi_2) \\
unroll(\forall v \in Y(x_1,\ldots,x_n) : \phi) & = & \bigwedge_{y\in[\![Y]\!]} unroll([v \mapsto y]\phi) \\
unroll(\exists v \in Y(x_1,\ldots,x_n) : \phi) & = & \bigvee_{y\in[\![Y]\!]} unroll([v \mapsto y]\phi) \\
unroll(\gamma X(\phi)) & = & \gamma X(unroll(\phi)) \\
unroll(\gamma(\phi_1\ U\ \phi_2)) & = & \gamma(unroll(\phi_1)\ U\ unroll(\phi_2))
\end{array}
$$

Suppose given a transformation rule T, program $\pi \in pgm$ and a label p. The concrete transformation rule T' is obtained by *instantiating* the transformation rule T by the substitution θ at label p in program π if T' is obtained by first computing the closed transformation rule T'' from T with respect to θ and then computing T' by unrolling.

5.3 Transformation

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

Apply. Suppose $\pi, \pi' \in pgm$ are programs, p is a label in π, T is a transformation rule and θ is a substitution. The relation $Apply(\pi, p, T, \theta, \pi')$ holds if and only of all of the following hold.

1. (Instantiation): Given θ there exists an instantiation T' by the substitution θ' of T at label p on the form
$$\mathcal{I} \Rightarrow \mathcal{I}' \quad \text{if} \quad \phi.$$
2. (Verification): The CTL-R clause $\mathcal{M}_{cf}(\pi), p \models \phi$ is satisfied.
3. (Construction): The transformed program $\pi' \in pgm$ has the form
$$\pi' = \texttt{read x};\ l_1 : \mathcal{I}'_{l_1};\ \ldots;\ l_m : \mathcal{I}'_{l_m};\ l_{m+1} : \texttt{write y},$$
where the statements of π' are defined by
$$\mathcal{I}'_{l_i} = \begin{cases} \theta'(\mathcal{I}') & \text{if } l_i = p \\ \mathcal{I}_{l_i} & \text{otherwise} \end{cases}$$

The program π' is obtained by *applying* the transformation T to to program π at label p given the substitution θ if $Apply(\pi, p, T, \theta, \pi')$ holds.

By the definition of *Apply*, the labels in a transformed program appear in the same order as the labels in the subject program.

Corollary 2. *Suppose $\pi, \pi' \in pgm$ and $Apply(\pi, p, T, \theta, \pi')$ then $Nx_\pi = Nx_{\pi'}$.*

6 Correctness proofs

Lemma 7. *Suppose $\pi \in pgm$ and $Apply(\pi, p, RW, \theta, \pi')$ for some $p \in label(\pi)$ and some transformation rule RW. If $in_\pi(v) = (p_0, \sigma_0)$ and $in_{\pi'}(v) = (p'_0, \sigma'_0)$ for some $v \in Value$ then $p_0 = p'_0$ and $\sigma_0|_{vars(\pi)\cap vars(\pi')} = \sigma'_0|_{vars(\pi)\cap vars(\pi')}$.*

6.1 A method for showing program equivalence

Corollary 3. *Suppose $\pi \in pgm$ and $Apply(\pi, p, T, \theta, \pi')$ holds for some label p and some transformation T. Let $C \in \mathcal{T}_{pfx}(\pi)$ and $C' \in \mathcal{T}_{pfx}(\pi')$ be computation prefixes of the same length of π and π':*

$$C = \pi, v \vdash (p_0, \sigma_0) \to \ldots \to (p_n, \sigma_n)$$
$$C' = \pi', v \vdash (p'_0, \sigma'_0) \to \ldots \to (p'_n, \sigma'_n),$$

such that $p_i = p'_i$ for all $0 \le i \le n$.

If there exists a relation $\mathcal{R}$ between computation prefixes such that: $C\mathcal{R}C'$ implies $p_i = p'_i$ for all $0 \le i \le n$ and $p_n = exit(\pi)$ implies that $\sigma_n(\mathtt{y}) = \sigma'_n(\mathtt{y})$ then $[\![\pi]\!] = [\![\pi']\!]$.

Proof. Suppose π does *not* terminate on input v, then $p_i \neq exit(\pi)$ for all $i > 0$, so $p'_i \neq exit(\pi) = exit(\pi')$ by Lemma 2 implying that π' does not terminate on input v. Conversely nontermination of π' implies nontermination of π. Otherwise π terminates on input v. Let n be given such that $p_n = exit(\pi) = exit(\pi')$. By assumption $\sigma_n(\mathtt{y}) = \sigma'_n(\mathtt{y})$ so $[\![\pi]\!](v) = [\![\pi']\!](v)$. Since v was an arbitrary value $[\![\pi]\!] = [\![\pi']\!]$. □

6.2 Copy propagation

The copy propagation transformation propagates assignments of the form `x:=y` to other assignments of the same type. The purpose is to render intermediate variables in chains of copying assignments dead.

Example. As an example of copy propagation, consider the following program fragment.

```
1 :  y := z;
2 :  a := 5 + z;
3 :  b := a * 2;
4 :  x := y;
```

Performing copy propagation on the statement at label 4 results in the fragment:

```
1 :  y := z;
2 :  a := 5 + z;
3 :  b := a * 2;
4 :  x := z;
```

Since the program variables `y` and `z` are not defined (assigned) between 2 and 4, the two variables evaluate to the same value at label 4. Thus either variable

can be used in the assignment. If the variable y is not used in subsequent computations then it can be removed by a transformation eliminating definitions of dead variables.

Copy propagation. The following transformation rule defines the copy propagation transformation (CP).

$$\mathbf{x}{:=}\mathbf{y} \Rightarrow \mathbf{x}{:=}\mathbf{z} \quad \text{if} \quad \begin{array}{l} \mathit{isvar}(\mathbf{z}) \wedge \\ \overleftarrow{A}(\neg \mathit{def}(\mathbf{z}) \wedge \neg \mathit{def}(\mathbf{y}) \wedge \neg \mathit{entry}\ W\ \mathit{stmt}(\mathbf{y}{:=}\mathbf{z})) \end{array}$$

Informally the side condition states:

1. **z** must be bound to a program variable [1].
2. Along all backwards paths in the control flow graph starting from statement to be transformed, the program variables that **y** and **z** are bound to must not be assigned to until the assignment **y:=z** is reached (which *must* eventually happen). Note that **y** and **z** are implicitly forced to be bound to program variables, since they appear on the left hand side of an assignment (where only program variables can appear).

Theorem 1. *Suppose given a program $\pi, \pi' \in pgm$, a label $p \in labels(\pi)$ and suppose $Apply(\pi, p, CP, \theta, \pi')$ then $[\![\pi]\!] = [\![\pi']\!]$.*

Proof. Let $\pi, \pi' \in pgm$, $Apply(\pi, p, \mathrm{CP}, \theta, \pi')$. variables. Suppose the instantiated transformation rule is given by

$$x{:=}y \Rightarrow x{:=}z \quad \text{if} \quad \begin{array}{l} \mathit{isvar}(z) \wedge \\ \overleftarrow{A}(\neg \mathit{def}(z) \wedge \neg \mathit{def}(y) \wedge \neg \mathit{entry}\ W\ \mathit{stmt}(y{:=}z)). \end{array}$$

Note that the variables x, y, z are bound to actual program variables, since x and y appears on the left hand side of an assignment, and the satisfaction of the side condition implies that z is variable.

Let $v \in \mathit{Value}$ be the input to the programs and let $C \in \mathcal{T}_{pfx}(\pi, v)$ and $C' \in \mathcal{T}_{pfx}(\pi', v)$ be two finite prefixes of the same length of π and π' respectively:

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

The idea of the proof is to show that the transformation preserves the set of computation prefixes, i.e. that any state transition sequence occurring in the computation trace of the original program π on input v, will also occur in the computation trace of the transformed program π' on input v and vice versa. We can then use Lemma 3 to conclude that the program semantics is preserved. First we prove that $(p_i, \sigma_i) = (p'_i, \sigma'_i)$ for all $i \leq t$ by induction over the length of C and C'.

[1] Otherwise **z** could be bound to an expression

Base case. Since $vars(\pi) = vars(\pi')$ then by Lemma 7: $(p_0, \sigma_0) = (p'_0, \sigma'_0)$.

Induction step. Since the semantics is deterministic there exists exactly one $C_2 \in \mathcal{T}_{pfx}(\pi, v)$ and one $C'_2 \in \mathcal{T}_{pfx}(\pi', v)$ such that $C \rightarrow C_2$ and $C' \rightarrow C'_2$. Assuming that the two computation prefixes are equal up to index t:

$$\forall 0 \leq i \leq t : (p_i, \sigma_i) = (p'_i, \sigma'_i)$$

then we need to show equivalence at index $t+1$: $(p_{t+1}, \sigma_{t+1}) = (p'_{t+1}, \sigma'_{t+1})$.

If $p_t \neq p$ then $\mathcal{I}_{p_t} = \mathcal{I}'_{p_t}$. By the induction assumption $\sigma_t = \sigma'_t$. Lemma 2 states that the next label functions for π and π' are identical, so $(p_{t+1}, \sigma_{t+1}) = (p'_{t+1}, \sigma'_{t+1})$ since the transition relation is deterministic (Corollary 1).

Otherwise $p_t = p$, implying that $\mathcal{I}_{p_t} = (x{:=}y)$ and $\mathcal{I}'_{p_t} = (x{:=}z)$. Clearly the next label for either computation prefix is given by the next label function, since the statement at p_t is an assignment in both programs. Lemma 2 states that the next label function is identical for the two programs π and π', so $p_{t+1} = p'_{t+1}$.

Both statements at p_t defines (assigns) variable x, so $p_t \models def(x)$ is satisfied for both $\mathcal{M}_{cf}(\pi)$ and $\mathcal{M}_{cf}(\pi')$. By Lemma 3 $\sigma_t \backslash x = \sigma_{t+1} \backslash x$ and $\sigma'_t \backslash x = \sigma'_{t+1} \backslash x$. The induction assumption implies $\sigma_t = \sigma'_t$ so by transitivity $\sigma_{t+1} \setminus x = \sigma'_{t+1} \setminus x$. Thus the stores σ_{t+1} and σ'_{t+1} are equivalent if $\sigma_{t+1}(x) = \sigma'_{t+1}(x)$.

The side condition must by assumption hold at p_t in the model for the control flow of program π:

$$\mathcal{M}_{cf}(\pi), p_t \models \overleftarrow{A}(\neg def(z) \wedge \neg def(y) \wedge \neg entry \ W \ stmt(y{:=}z)).$$

Using Lemma 6 a strong until (U) version of the above statement is satisfied in the model $\mathcal{M}_{cf}(\pi)$ for all finite paths:

$$\begin{array}{l} \forall \text{finite paths} : (\rightarrow n_1 \rightarrow n_0 = p_t) \in Label^* \\ \exists i \geq 0 : n_i \models stmt(y{:=}z) \wedge \forall 0 \leq j < i : n_j \models (\neg def(z) \wedge \neg def(y)). \end{array}$$

Substituting the definition of the valuation for *stmt* valuations, we get:

$$\begin{array}{l} \forall \text{finite paths} : (\rightarrow n_1 \rightarrow n_0 = p :) \in Label^* \\ \exists i \geq 0 : \mathcal{I}_{n_i} = (y{:=}z) \wedge \forall 0 \leq j < i : n_j \models \neg def(z) \wedge n_j \models \neg def(y). \end{array}$$

Due to Lemma 5 any computation prefix must be described by the control flow model, thus the following holds for the computation prefix

$$\begin{array}{ll} \exists i \leq t : & \mathcal{I}_{p_i} = (y{:=}z) \wedge \\ & \forall t \geq j > i : \mathcal{M}_{pfx}(C), p_j \models \neg def(z) \wedge \mathcal{M}_{pfx}(C), p_j \models \neg def(y). \end{array}$$

Let i be given such that the above holds. The program variables z and y are not defined at statements $\mathcal{I}_{p_t}, \ldots, \mathcal{I}_{p_{i+1}}$, so Lemma 2 implies that y and z does not change their value from σ_{i+1} to σ_t: $\sigma_{i+1}(y) = \sigma_t(y)$ and $\sigma_{i+1}(z) = \sigma_t(z)$.

Furthermore $\mathcal{I}_{p_i} = (y{:=}z)$ so by the semantics: $\sigma_{i+1}(y) = \sigma_i(z)$. Clearly z is not defined at $\mathcal{I}_{p_i}$: $p_i \models \neg def(z)$, so by Lemma 2: $\sigma_i(z) = \sigma_{i+1}(z)$. Thus by transitivity $\sigma_t(y) = \sigma_t(z)$.

$$\begin{array}{rcll} \sigma_{t+1}(x) & = & \sigma_t(y) & \text{(semantics of } \mathcal{I}_{p_t} = (x{:=}y)) \\ & = & \sigma_t(z) & \text{(argument above)} \\ & = & \sigma'_t(z) & \text{(induction assumption)} \\ & = & \sigma'_{t+1}(x) & \text{(semantics of } \mathcal{I}_{p_{t+1}} = (x{:=}z)) \end{array}$$

This proves that $\sigma_{t+1} = \sigma'_{t+1}$, implying that $(p_{t+1}, \sigma_{t+1}) = (p'_{t+1}, \sigma'_{t+1})$.

Per induction for any finite computation prefixes $C \in \mathcal{T}_{pfx}(\pi, v)$ and $C' \in \mathcal{T}_{pfx}(\pi', v)$ of the same length t, the two computation prefixes are identical.

It then follows by Lemma 3 that $[\![\pi]\!] = [\![\pi']\!]$. □

6.3 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, provided that the changes in the store does not affect the variables which the expression depends on.

Example. As an example, consider the following program fragment.

```
1 : x := a + b
2 : if ... then 2 else 3
3 : z := 5
4 : v := a + b
```

Applying the transformation at label 4 results in the following fragment.

```
1 : x := a + b
2 : if ... then 2 else 3
3 : z := 5
4 : v := x
```

The transformation is semantics preserving since the expression `a+b` evaluates to the same value at label `1`, and `4` because neither variable changes it value from `1` to `4` regardless of the path taken. The expression is *available* in the variable `x` at label 4 since `x` is not modified either.

Elimination of recomputation of available expressions. The following rewrite rule defines the "elimination of recomputation of available expressions" transformation (ER).

$$\mathbf{x{:=}e} \Rightarrow \mathbf{x{:=}z} \quad \text{if} \quad \neg \mathit{inexp}(\mathbf{z}, \mathbf{e}) \wedge \overleftarrow{A}\, X\, \overleftarrow{A}(\quad \neg \mathit{def}(\mathbf{z}) \wedge \forall y \in \mathit{vars}(\mathbf{e}) : \neg \mathit{def}(y) \wedge \neg \mathit{entry}\ W \ \mathit{stmt}(\mathbf{z{:=}e}))$$

Informally the side condition states:

1. **z** may not occur in the expression **e**. If it did, the expression **e** would (potentially) evaluate to a different value in the store after the assignment.
2. The second part of the side condition states that for all backwards paths from the current label back to the statement **z:=e** (which must eventually be reached), the value of **z** is unchanged. Furthermore any program variable occurring in **e** does not change its value on the path. This implies that the expression **e** evaluates to the same expression at label where the transformation takes place and at the **z:=e** statement.
3. The side condition implicitly states that the CTL-R variables **x** and **z** are bound to program variables, since they appear on the left hand side of assignments. The CTL-R variable **e** must be bound to some program expression, since it appears on the right hand side of assignments.

Theorem 2. *Suppose* $\pi, \pi' \in pgm$, $Apply(\pi, p, ER, \theta, \pi')$ *then* $[\![\pi]\!] = [\![\pi']\!]$.

Proof. Let $\pi, \pi' \in pgm$, $Apply(\pi, p, ER, \theta, \pi')$. Let $v \in Value$ be the input to the programs π and π', and let $C \in \mathcal{T}_{pfx}(\pi, v)$ and $C' \in \mathcal{T}_{pfx}(\pi', v)$ be program prefixes of the same length:

$$\begin{array}{lcl} C & = & \pi, v \vdash (p_0, \sigma_0) \to \ldots \to (p_t, \sigma_t) \\ C' & = & \pi', v \vdash (p'_0, \sigma'_0) \to \ldots \to (p'_t, \sigma'_t). \end{array}$$

Suppose the instantiated transformation rule is given by:

$$\begin{array}{lll} x{:=}e \Rightarrow x{:=}z & \text{if} & \neg inexp(z, e) \wedge \\ & & \overleftarrow{A}\, X\, \overleftarrow{A}\,(\quad \neg def(z) \wedge \forall y \in vars(e) : \neg def(y) \wedge \neg entry\ W \\ & & \qquad stmt(z{:=}e)) \end{array}$$

First we prove by induction over the length of C, C' that the two computation prefixes are identical: $(p_i, \sigma_i) = (p'_i, \sigma'_i)$ for all $i \leq t$.

Base case. Since $vars(\pi) = vars(\pi')$ then by lemma 7: $(p_0, \sigma_0) = (p'_0, \sigma'_0)$.

Induction step. Since the semantics is deterministic there exists exactly one $C_2 \in \mathcal{T}_{pfx}(\pi, v), C'_2 \in \mathcal{T}_{pfx}(\pi', v)$ such that $C \to C_2$ and $C' \to C'_2$. 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 Lemma 2 and the induction assumption.

Otherwise $p_t = p$, so $\mathcal{I}_{p_t} = (x{:=}e)$, $\mathcal{I}'_{p_t} = (x{:=}z)$. By Lemma 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:

$$p \models \overleftarrow{A}\, X\, \overleftarrow{A}(\neg def(z) \wedge \forall y \in vars(e) : \neg def(y) \wedge \neg entry\ W\ stmt(z{:=}e)).$$

Using Lemma 6 this implies

$$\begin{array}{l} \forall \text{finite paths} : (\ldots \to n_1 \to n_0 = p) \in Label^* : \\ \quad \exists i \geq 1 : \quad n_i \models stmt(z{:=}e) \wedge \\ \qquad\qquad \forall 1 \leq j < i : (n_j \models \neg def(z)) \wedge \forall y \in vars(e) : n_j \models \neg def(y). \end{array}$$

Since the control flow model describes the control flow for any possible computation – Lemma 5 – the above applies for computation prefix C. Applying the definition of the valuation of the atomic propositions, the following must hold for the concrete computation prefix C:

$$\begin{array}{lll} \exists i < t: & \mathcal{I}_{p_i} = (z{:=}e) \wedge & \\ & \forall i < j \leq t: & p_j \models \neg def(z) \wedge \\ & & \forall y \in vars(e): p_j \models \neg def(y). \end{array}$$

The rewrite rule does not change *def* atomic propositions, so

$$\forall y \in vars(e): p_j \models \neg def(y)$$

must also hold for the control flow model for π': $\mathcal{M}_{cf}(\pi')$ for $i < j \leq t$. Similarly $p_i \models def(z)$ holds in $\mathcal{M}_{cf}(\pi')$. The side condition states that $z \notin vars(e)$, so

$$\forall y \in vars(e): p_j \models \neg def(y)$$

holds for the program point with the assignment $z{:=}e$, i.e. the property holds for $j = i$ as well. By Lemma 2 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.

Next we prove that $\sigma'_{i+1}(z) = [\![E]\!]\sigma'_i|_{vars(e)}$:

Suppose $p_i \neq p$ then $\mathcal{I}'_{p_i} = (z{:=}e)$ so by the semantics $[\![e]\!]\sigma'_i|_{vars(e)} = \sigma'_{i+1}(z)$.

Otherwise $p_i = p$. Since the transformation rule was instantiated $x = z$ and $\mathcal{I}'_{p_i} = (x{:=}x)$, so $\sigma'_i = \sigma'_{i+1}$ by the semantics. By the induction assumption $\sigma_i = \sigma'_i$ and $\sigma_{i+1} = \sigma'_{i+1}$, so by transitivity $\sigma_i = \sigma_{i+1}$. The statement at p_i in π is $\mathcal{I}_p = (x{:=}e)$, thus $\sigma_{i+1}(x) = [\![e]\!]\sigma_i|_{vars(e)}$. Due to equivalence $\sigma'_{i+1}(z) = [\![e]\!]\sigma'_i|_{vars(e)}$ holds.

Putting the pieces together:

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

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

Thus any two computation prefixes of the same length $C \in \mathcal{T}_{pfx}(\pi, v)$ and $C' \in \mathcal{T}_{pfx}(\pi', v)$ are equivalent. By Lemma 3 the transformation is semantics preserving: $[\![\pi]\!] = [\![\pi']\!]$.

□

7 Skip removal and insertion

As Lemma 2 states, the framework is not strong enough to handle transformations that insert or remove statements. For completeness three transformations are proven correct outside the framework.

1. *Statement insertion transformation*: inserting a `skip` statement between two statements in the subject program.
2. `skip` *removal transformation*: deleting a `skip` statement from the program while changing all references to the `skip` statement to the statement immediately syntactically after.
3. *dead assignment insertion*: rewriting a `skip` statement to an assignment to a variable that does not occur in the subject program.

These transformations and their proofs are not particularly interesting, they simply prove intuitive facts. Armed with these transformations, it is possible to perform the elimination of recomputation of available expressions transformation, for instance, without leaving the transformed program cluttered with useless `skip` statements.

As will be demonstrated subsequently, some transformations "insert" statements in the program and thus (when specified in this framework) will have to rely on the presence of `skip` statements in the "right" places. The `skip` insertion transformation is targeted to handle these issues.

Statement insertion. Suppose $\pi \in pgm$ has the form

$$\texttt{read x};l_1 : \mathcal{I}_{l_1};\ \ldots\ ;l_m : \mathcal{I}_{l_m};\ l_{m+1} : \texttt{write y};$$

and $p \in Label \setminus labels(\pi)$ is a unique label. For any label $p' \in labels(\pi)$ the *statement insertion transformation* is defined as $\pi' = SkipIns(\pi, p', p)$ where $p' = l_i$ for some i and $\pi' \in pgm$ is a program of the form

$$\texttt{read x};\ \ldots\ l_{i-1} : \mathcal{I}_{l_{i-1}};\ p : \texttt{skip};\ l_i : \mathcal{I}_{l_i};\ \ldots\ l_{m+1} : \texttt{write y};.$$

Theorem 3. *Suppose* $\pi \in pgm$, $l_i \in labels(\pi)$, $p \in Label \setminus labels(\pi)$ *and* $\pi' = SkipIns(\pi, l_i, p)$ *then* $[\![\pi]\!] = [\![\pi']\!]$.

Proof. Proof idea: show by induction for any two computation prefixes $C \in \mathcal{T}_{pfx}(\pi)$ and $C' \in \mathcal{T}_{pfx}(\pi')$:

$$C = \pi, v \vdash (p_0, \sigma_0) \rightarrow \ldots \rightarrow (p_m, \sigma_m)$$
$$C' = \pi', v \vdash (p'_0, \sigma'_0) \rightarrow \ldots \rightarrow (p'_n, \sigma'_n)$$

that for all $C_2 \in \mathcal{T}_{pfx}(\pi)$ and $C'_2 \in \mathcal{T}_{pfx}(\pi')$ such that $C \rightarrow C_2$, $C' \rightarrow C'_2$ and $C\mathcal{R}C'$ implies $C_2\mathcal{R}C'_2$ where $C\mathcal{R}C'$ if and only if $p \neq p_m = p'_n$ and $\sigma_m = \sigma'_n$.

Base case. Suppose $p'_0 \neq p$ then clearly the initial states are equivalent. Otherwise $\sigma_0 = \sigma'_0$ and $p_0 = Nx_{\pi'}(p'_0)$, so it follows by the semantics of `skip` that $(p'_1, \sigma'_1) = (p_0, \sigma_0)$.

Induction step. Suppose $p'_{n+1} \neq p$ implying $\mathcal{I}_{p_m} = \mathcal{I}'_{p'_n}$. The label p does not occur in the program π, so $p_m \neq p$. The induction assumption $(p_m, \sigma_m) = (p'_n, \sigma'_n)$ then implies that $(p_{m+1}, \sigma_{m+1}) = (p'_{n+1}, \sigma'_{n+1})$.

Otherwise $p'_{n+1} = p$. The semantics of `skip` and the definition of π' implies that $p'_{n+2} = p_{m+1}$. By the induction assumption it follows that $\sigma_{m+1} = \sigma'_{n+1} = \sigma'_{n+2}$. Thus the induction assumption holds for $m+1$ and $n+2$.

By induction for any computation prefix C of π there exists a computation prefix C' of π' such that $(p_m, \sigma_m) = (p'_n, \sigma'_n)$ and conversely for any computation prefix C' of π' such that $p'_n \neq p$ there exists a computation prefix C of π such that the final states are equivalent.

Clearly π terminates if and only if π' terminates and if they terminate, they compute the same store. Thus the equivalence $[\![\pi]\!] = [\![\pi']\!]$ follows by the definition of program semantics. □

Skip removal. Suppose $\pi \in pgm$ has the form

$$\texttt{read x}; l_1 : \mathcal{I}_{l_1}; \ldots; l_m : \mathcal{I}_{l_m}; l_{m+1} : \texttt{write y};.$$

The *skip removal transformation* is defined by: $\pi' = SkipRm(\pi, p)$ if $p = l_i$ for some i and

$$\pi' = [p \mapsto Nx_\pi(p)](\texttt{read x}; \ldots; l_{i-1} : \mathcal{I}_{l_{i-1}}; l_{i+1} : \mathcal{I}_{l_{i+1}}; \ldots; l_{m+1} : \texttt{write y};).$$

Theorem 4. *Suppose $\pi \in pgm$ is a program, $p \in labels(\pi)$ is a label in π, and $\pi' = SkipRm(\pi, p)$ then $[\![\pi]\!] = [\![\pi']\!]$.*

Proof. Proof idea: show by induction for any two computation prefixes $C \in \mathcal{T}_{pfx}(\pi)$ and $C' \in \mathcal{T}_{pfx}(\pi')$:

$$C = \pi, v \vdash (p_0, \sigma_0) \to \ldots \to (p_m, \sigma_m)$$
$$C' = \pi', v \vdash (p'_0, \sigma'_0) \to \ldots \to (p'_n, \sigma'_n)$$

that for all $C_2 \in \mathcal{T}_{pfx}(\pi)$ and $C'_2 \in \mathcal{T}_{pfx}(\pi')$ such that $C \to C_2$, $C' \to C'_2$ and $C \mathcal{R} C'$ implies $C_2 \mathcal{R} C'_2$ where $C \mathcal{R} C'$ if and only if $p \neq p_m = p'_n$ and $\sigma_m = \sigma'_n$.

Base case. Suppose $p'_0 \neq p$ then clearly the initial states are equivalent. Otherwise $\sigma_0 = \sigma'_0$ and $p'_0 = Nx_\pi(p_0)$, so it follows by the semantics of `skip` that $(p_1, \sigma_1) = (p'_0, \sigma'_0)$.

Induction step. Suppose $p_{m+1} \neq p$. The label p does not occur in the program π', so $p'_n \neq p$. The induction assumption $(p_m, \sigma_m) = (p'_n, \sigma'_n)$ then implies that $(p_{m+1}, \sigma_{m+1}) = (p'_{n+1}, \sigma'_{n+1})$.

Otherwise $p'_{n+1} = p$. The semantics of `skip` and the definition of π' implies that $p'_{n+2} = p_{m+1}$. By the induction assumption it follows that $\sigma_{m+1} = \sigma'_{n+1} = \sigma'_{n+2}$. Thus the induction assumption holds for $m+1$ and $n+2$.

By induction for any computation prefix C of π there exists a computation prefix C' of π' such that $(p_m, \sigma_m) = (p'_n, \sigma'_n)$ and conversely for any computation

prefix C' of π' such that $p'_n \neq p$ there exists a computation prefix C of π such that the final states are equivalent.

Clearly π terminates if and only if π' terminates and if they terminate, they compute the same store. Thus the equivalence $[\![\pi]\!] = [\![\pi']\!]$ follows by the definition of program semantics. □

Dead assignment insertion. Suppose $\pi \in \mathit{pgm}$ is a program, $x \in \mathit{Variable} \setminus \mathit{vars}(\pi)$ is a variable not occurring in π and $e \in \mathit{expr}$ is an expression such that $y \in \mathit{vars}(\pi)$ for all $y \in \mathit{vars}(e)$. Further suppose that for some label p: $\mathcal{I}_p = \texttt{skip}$. The *dead assignment insertion transformation* is defined as

$$\mathit{DeadIns}(\pi, p, x, e) = \texttt{read x}; l_1 : \mathcal{I}'_{l_1}; \ldots; l_m : \mathcal{I}'_{l_m}; l_{m+1} : \texttt{write y};$$

where the statements in the transformed program are given by

$$\mathcal{I}'_{l_i} = \begin{cases} x{:=}e & \text{if } l_i = p \\ \mathcal{I}_{l_i} & \text{otherwise} \end{cases} .$$

Theorem 5. *Suppose $\pi \in \mathit{pgm}$, $p \in \mathit{labels}(\pi)$, $x \in \mathit{Variable} \setminus \mathit{vars}(\pi)$ and $e \in \mathit{expr}$ such that $y \in \mathit{vars}(\pi)$ for all $y \in \mathit{vars}(e)$. If $\pi' = \mathit{DeadIns}(\pi, p, x, e)$ then $[\![\pi]\!] = [\![\pi']\!]$.*

Proof. Using Lemma 3 it can be shown by induction that the stores in two computation prefixes of the same length over π and π', given the same input $v \in \mathit{Value}$, are equivalent when restricted to $\mathit{vars}(\pi)$. This implies semantic program equivalence, since $\texttt{y} \in \mathit{vars}(\pi)$. □

7.1 Common subexpression elimination

At this point it is possible to prove correct the elimination of common subexpressions transformation. In fact the correctness follows directly from the correctness of the transformations already treated since they can be combined to obtain the common subexpression elimination transformation. The transformation detects expressions that are *very busy*, i.e. expressions that are computed eventually on all paths leading from a given program point, where the variables in the expression have not changed their value. An assignment of the expression to a fresh variable is inserted at the particular program point, rendering the subsequent recomputations redundant.

This can be achieved by using some of the transformations already proven formally correct.

1. Decide on a suitable label p, and fresh variable x and some expression e. Usually a compiler would select a label where some expression e is very busy.
2. Insert a `skip` statement labeled p' at label p.
3. Perform the dead assignment insertion transformation at label p' with variable x and expression e.

4. Eliminate as many redundant assignments by repeatedly applying the elimination of recomputation of available expressions transformation at various labels in the program.

5. Delete all `skip` statements from the transformed program.

Since the purpose of the paper is to prove correctness of some typically employed compiler optimizations, we will be satisfied with claiming that it is possible to perform elimination of common subexpressions by the above procedure.

8 Transformations with multiple program points

This section presents an extension of the framework that handles transformations in which more than one program statement is affected.

8.1 Transformation syntax

Transformation rules. A *multi-label transformation rule* has the syntax:

$$
\begin{array}{lcl}
\mathit{rule} & ::= & \mathit{rw}_1 \ldots \mathit{rw}_k \text{ if } \mathit{cond}_1 \ldots \mathit{cond}_k \\
\mathit{rw} & ::= & p : \mathit{vstmt}_1 \Rightarrow \mathit{vstmt}_2 \\
\mathit{cond} & ::= & p \models \phi \\
\mathit{vstmt} & ::= & v := \mathit{vexp} \mid \texttt{if } v \texttt{ then } \mathbf{v}_1 \texttt{ else } \mathbf{v}_2 \mid \texttt{skip} \\
\mathit{vexp} & ::= & c \mid x \mid \mathbf{v} \mid \mathtt{op}(\mathit{vexp}_1, \ldots, \mathit{vexp}_n) \\
\phi & ::= & \mathit{true} \mid \mathit{false} \mid \mathit{ap}(\mathbf{v}_1, \ldots, \mathbf{v}_n) \mid \neg\phi \mid \phi_1 \wedge \phi_2 \\
 & & \mid \forall y \in Y(\mathbf{v}_1, \ldots, \mathbf{v}_n) : \phi \mid \exists y \in Y(\mathbf{v}_1, \ldots, \mathbf{v}_n) : \phi \\
 & & \mid A\psi \mid E\psi \mid \overleftarrow{A}\psi \mid \overleftarrow{E}\psi \\
\psi & ::= & X\phi \mid \phi_1 \; U \; \phi_2 \\
Y & ::= & \mathit{vars} \\
\mathbf{v} & \in & \mathit{FormulaVariable} \\
c & \in & \mathit{Value} \\
x & \in & \mathit{Variable} \\
y & \in & \mathit{QuantifierVariable} \\
p & \in & \mathit{LabelVariable}
\end{array}
$$

The main difference from the single rewrite case is that since the transformation now matches against and changes statements at several labels, a device for referring to a specific label is needed. Thus each rewrite rule "rw" is now annotated with a label variable. For each rewrite rule labeled by some label, there must exist exactly one side condition annotated with the same label.

As previously noted, the label invariant atomic propositions do not really belong in the CTL-R clauses and in the situation where several clauses are present this is even more pronounced, since there is no natural clause to insert them in. However this approach does avoid introducing inessential concepts.

8.2 Concrete transformation

The concrete transformation is computed in a manner similar to the single rewrite rule case. When closing the transformation, a substitution θ' is computed such that θ' unifies all the left hand sides of the rewrite rules with their corresponding side conditions, for some label tuple $\vec{p}$.

Closed transformation rule. Suppose $\pi \in pgm$ is a program, $\vec{p} = (p_1 \ldots p_k) \in Label^*$ is a tuple of labels occurring in π, T is a multi-label transformation rule

$$\begin{array}{l} p_1 : lhs_1 \Rightarrow rhs_1 \ldots p_k : lhs_k \Rightarrow rhs_k \\ \text{if} \\ p_1 \models \phi_1 \ldots p_k \models \phi_k \end{array}$$

and θ is a substitution binding all variables not occurring in lhs_i for any i. Define θ' as the substitution that unifies the left hand side lhs_i with the statement at label p_i, for all $1 \leq i \leq k$. Provided that such a substitution exists, the multi-label transformation rule T' can be obtained from T by closure with respect to $\theta'' = \theta \circ \theta'$ if $T' = \theta''(T)$ and T' contains no free variables. The multi-label transformation rule T' i called a *closed transformation rule*, since it contains no free variables. The type of θ'' is given by $\theta'' : FormulaVariables \rightarrow Label \cup stmt \cup exp \cup Variable$.

From a closed multi-label transformation rule, the *concrete* transformation rules can be obtained by "unrolling" the program quantifiers in the all the CTL-R clauses in the the side condition.

Concrete transformation rule. Suppose T is the closed multi-label transformation rule as given above, then the *concrete multi-label transformation rule* is computed by unrolling all the side conditions

$$\begin{array}{l} p_1 : lhs_1 \Rightarrow rhs_1 \ldots p_k : lhs_k \Rightarrow rhs_k \\ \text{if} \\ p_1 \models unroll(\phi_1) \ldots p_k \models unroll(\phi_k). \end{array}$$

Suppose given a multi-label transformation rule T, program $\pi \in pgm$ and a label tuple $\vec{p}$. The concrete multi-label transformation rule T' is obtained by *instantiating* the multi-label transformation rule T by the substitution θ at label tuple $\vec{p}$ in program π if T' is obtained by first computing the closed transformation rule T'' from T with respect to θ and then computing T' by unrolling.

8.3 Transformation

The definition of application of a multi-label transformation at some specific labels in the subject program is analogous to the single rewrite rule case.

Apply. Suppose $\pi, \pi' \in pgm$ are programs, $\vec{p} = (p_1, \ldots, p_k) \in Label^*$ is a tuple of labels in π, T is a multi-label transformation rule and θ is a substitution. The program π' is obtained by *applying* the multi-label transformation T at the labels $\vec{p}$ with substitution θ if and only if $Apply^*(\pi, \vec{p}, T, \theta, \pi')$, where the relation $Apply^*$ holds if and only if all of the following cases hold.

1. (Instantiation): Given the substitution θ there exists an instantiation T' of T with the substitution θ', where T' has the form

$$\begin{array}{l} p_1 : \mathcal{I}_1 \Rightarrow \mathcal{I}'_1 \ldots p_k : \mathcal{I}_k \Rightarrow \mathcal{I}'_k \\ \text{if} \\ p_1 \models \phi_1 \ldots p_k \models \phi_k. \end{array}$$

2. (Verification): For all $i = 1 \ldots k$ the CTL-R clause $\mathcal{M}_{cf}(\pi), p_i \models \phi_i$ is satisfied.

3. (Construction): The transformed program π has the form

$$\pi' = \texttt{read x};\ l_1 : \mathcal{I}'_{l_1};\ \ldots;\ l_m : \mathcal{I}'_{l_m};\ l_{m+1} : \texttt{write y},$$

where the statements of π' are defined by

$$\mathcal{I}'_{l_i} = \begin{cases} \theta(\mathcal{I}'_i) & \text{if } \exists j : l_i = p_j \\ \mathcal{I}_{l_i} & \text{otherwise} \end{cases}$$

Corollary 4. *Suppose $\pi, \pi' \in pgm$ and $Apply^*(\pi, \vec{p}, T, \theta, \pi')$ then $Nx_\pi = Nx_{\pi'}$.*

8.4 Loop invariant hoisting

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

$$\begin{array}{l} p : \texttt{skip} \Rightarrow \mathbf{x}\texttt{:=}\mathbf{e} \\ q : \mathbf{x}\texttt{:=}\mathbf{e} \Rightarrow \texttt{skip} \\ \text{if} \\ p \models A(\neg use(\mathbf{x})\ U\ node(q)) \\ q \models \overleftarrow{A}\, X\, \overleftarrow{A}(\neg def(\mathbf{x}) \wedge \forall y \in vars(\mathbf{e}) : \neg def(y) \wedge \neg entry\ W\ node(p)). \end{array}$$

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

1. The assigned variable $\mathbf{x}$ is dead after p until q is reached. If this requirement holds, then introducing the assignment $\mathbf{x}\texttt{:=}\mathbf{e}$ 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 $\mathbf{e}$ should be available at q *after* the transformation.

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

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

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

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

CM relation. 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: $C \mathcal{R} C'$ if and only if $t = t' \wedge \forall 0 \leq i \leq t : p_i = p'_i$ and one of the following cases holds:

1. $\sigma_t = \sigma'_t \ \wedge \ \forall 0 \leq i \leq t : p_i \notin \{p, q\}$
2. $\sigma_t = \sigma'_t \ \wedge \ \exists 0 \leq i \leq t : p_i = q \ \wedge \ \forall i \leq j \leq t : p_j \neq p$
3. $\exists 0 \leq i \leq t : p_i = p \ \wedge$
   $(\sigma_i \setminus x = \sigma'_i \setminus x) \ \wedge \ (x \in vars(e) \Rightarrow \sigma_i(x) = \sigma'_i(x)) \ \wedge$
   $\forall i \leq j \leq t : p_j \notin \{p, q\}$

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

Theorem 6. *Suppose* $\pi, \pi' \in pgm$, $Apply^*(\pi, (p, q), CM, \theta, \pi')$ *then* $[\![\pi]\!] = [\![\pi']\!]$.

Proof. Suppose $\pi, \pi' \in pgm$ and $Apply^*(\pi, (p, q), CM, \theta, \pi')$. Let $v \in Value$ be the input to program π, π' and let $C \in \mathcal{T}_{pfx}(\pi, v)$, $C' \in \mathcal{T}_{pfx}(\pi', v)$ be two finite computation prefixes of the same length of π and π' respectively.

$$\begin{array}{rcl} C_1 & = & \pi, v \vdash (p_0, \sigma_1) \rightarrow \ldots \rightarrow (p_t, \sigma_t) \\ C'_1 & = & \pi', v \vdash (p'_0, \sigma'_1) \rightarrow \ldots \rightarrow (p'_t, \sigma'_t) \end{array}$$

Suppose the instantiated multi-label transformation rule is given by

p : `skip` $\Rightarrow x{:=}e$
q : $x{:=}e \Rightarrow$ `skip`
if
$p \models A(\neg use(x) \ U \ node(q))$
$q \models \overleftarrow{A} X \overleftarrow{A}(\neg def(x) \wedge \forall y \in vars(e) : \neg def(y) \wedge \neg entry \ W \ node(p))$.

First we show that $C\mathcal{R}C'$ by induction over the length of C, C'.

Base case. Since $vars(\pi) = vars(\pi')$ then by Lemma 7: $(p_0, \sigma_0) = (p'_0, \sigma'_0)$.

Induction step. The semantics for the language is deterministic, so there exists exactly one C_2 and one C'_2 such that: $C \to C_2$ and $C' \to C'_2$. Assuming that $C\mathcal{R}C'$ we need to show that $C_2\mathcal{R}C'_2$.

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

Suppose $p_t \notin \{p, q\}$, then $\mathcal{I}_{p_t} = \mathcal{I}'_{p_t}$.

1. Suppose $C\mathcal{R}_iC'$ where $i = 1, 2$. By assumption $\sigma_t = \sigma'_t$ so π and π' evaluate the same statement in the same store with the same next label function. It then follows from Lemma 2 $(p_{t+1}, \sigma_{t+1}) = (p'_{t+1}, \sigma'_{t+1})$, so $C_2\mathcal{R}_iC'_2$.

2. Otherwise $C\mathcal{R}_3C'$, so let $0 \leq i \leq t$ be given such that:
   $p_i = p \ \wedge \ (\sigma_i \setminus x = \sigma'_i \setminus x)$
   The side condition for p must be satisfied at p_i:

   $$p_i \models A(\neg use(x) \ U \ node(q)).$$

   By the definition of CTL-R this is equivalent to:

   $$\forall(p_i = n_0 \to n_1 \to n_2 \to \ldots) \in Label^*$$
   $$\exists \alpha \geq 0 : n_\alpha \models node(q) \wedge \forall 0 \leq \beta < \alpha : n_\beta \models \neg use(x)$$

   Since by Lemma 5 the control flow model describes all possible computation prefixes, the same must hold for the computation prefix C. By assumption we know that

   $$\forall i < j \leq t : p_j \neq q.$$

   Thus $p_j \models \neg use(x)$ is satisfied for all $i < j \leq t$. In particular $p_t \models \neg use(x)$ is satisfied and by assumption $(\sigma_i \setminus x = \sigma'_i \setminus x)$ so using Lemma 4 we conclude: $\sigma_{t+1} \setminus x = \sigma'_{t+1} \setminus x$.

   Suppose $x \in vars(e)$ then $\sigma_t = \sigma'_t$, so clearly $\sigma_{t+1} = \sigma'_{t+1}$, fulfilling the requirement on the stores in C and C'.

   It remains to be shown that the labels are equivalent at $t + 1$. Suppose p_{t+1} is given by the next label function. By the induction assumption $p_t = p'_t$, so clearly $p_{t+1} = p'_{t+1}$ holds. Otherwise the statement at p_t is a conditional and the next state depends on the variable being tested. By the above reasoning $p_t \models \neg use(x)$ implying that the conditional does not depend on the x. By induction assumption $\sigma_t \setminus x = \sigma'_t \setminus x$, so $p_{t+1} = p'_{t+1}$. Thus $C_2\mathcal{R}_3C'_2$ holds for i unchanged.

Suppose $p_t = p$, then $p_t \models A(\neg use(x)\ U\ node(q))$, $\mathcal{I}_{p_t} = (\texttt{skip})$ and $\mathcal{I}'_{p_t} = (x{:=}e)$. By the semantics p_{t+1} and p'_{t+1} is given by the next label functions Nx_π, $Nx_{\pi'}$, which by Lemma 2 are identical. Since $p_t = p'_t$ by the induction assumption, it follows that $p_{t+1} = p'_{t+1}$.

1. Suppose $C\mathcal{R}C'$ by case 1 or 2. The induction assumption implies that $\sigma_t = \sigma'_t$, so by the semantics: $\sigma_{t+1} \setminus x = \sigma'_{t+1} \setminus x$. Clearly $C_2\mathcal{R}_2C'_2$ holds for $i = t$.

2. Otherwise $C\mathcal{R}_3C'$: As previously shown this implies the satisfaction of $p_t \models \neg use(x)$, so by Lemma 4 the statement at p_t does not depend on the value on x, implying: $\sigma_{t+1} \setminus x = \sigma'_{t+1} \setminus x$.

   Suppose $x \in vars(e)$, which by the definition of the valuation implies the satisfaction of $p_t \models use(x)$, contradicting the above conclusion.

   Thus $C\mathcal{R}_3C'$ holds for i unchanged.

Otherwise $p_t = q$ so $\mathcal{I}_{p_t} = (x{:=}e)$ and $\mathcal{I}'_{p_t} = (\texttt{skip})$. By the semantics p_{t+1} and p'_{t+1} is given by the next label function, which by Lemma 2 is the identical for π and π'. Since $p_t = p'_t$ by the induction assumption then $p_{t+1} = p'_{t+1}$.

1. Suppose $C\mathcal{R}_1C'$: By Lemma 6 the side condition in the rewrite rule

$$p_t \models \overleftarrow{A}X\overleftarrow{A}(\neg def(x) \wedge \forall y \in vars(e) : \neg def(y) \wedge \neg entry\ W\ node(p)).$$

   implies that any backwards path eventually ends up in a state p_k where $p_k \models node(p)$ is satisfied, i.e. $p_k = p$. This contradicts the assumption $C_1\mathcal{R}_1C'_1$.

2. Suppose $C\mathcal{R}_2C'$: Per assumption there exists an i such that $p_i = q$. Looking at the side condition for p_t we get:

$$p_t \models \overleftarrow{A}X\overleftarrow{A}(\neg def(x) \wedge \ldots \wedge \neg entry\ W\ node(p)).$$

   This implies by Lemma 6

$$\begin{array}{l}\forall \text{finite paths} : (\ldots \to p_{t-2} \to p_{t-1}) \in Label^* : \\ \exists \alpha < t : p_\alpha \models node(p) \wedge \forall \alpha < \beta \leq t-1 : p_\beta \models \neg def(x)\end{array}$$

   Since by assumption $p_j \neq p$ for all $i \leq j \leq t$ it follows from the definition of the valuation that the clause $p_j \models \neg node(p)$ is satisfied, so $\alpha < i < t$. In particular $p_i \models \neg def(x)$ is satisfied. But statement $\mathcal{I}_q = (x{:=}e)$ clearly defines x, so $p_i = q \models def(x)$ contradicting the above conclusion.

3. Otherwise $C\mathcal{R}_3C'$: By assumption there exists an i such that $p_i = p$, $(\sigma_i \setminus x = \sigma'_i \setminus x)$, $(x \in vars(e) \Rightarrow \sigma_i(x) = \sigma'_i(x))$ and

$$\forall i \leq j \leq t : p_j \neq q \wedge \sigma_j \setminus x = \sigma'_j \setminus x.$$

   We wish to show that $C_2\mathcal{R}C'_2$ holds by case 2 for $i = t$, i.e. $\sigma_{t+1} = \sigma'_{t+1}$.

First we show that $\sigma_{t+1} \setminus x = \sigma'_{t+1} \setminus x$. We know from the assumption $p_t = p$ that the statement at p_t defines x: $\mathcal{M}_{cf}(\pi), p_t \models def(x)$, so by Lemma 3: $\sigma_t \setminus x = \sigma_{t+1} \setminus x$.

$$\begin{array}{rcll} \sigma_{t+1} \setminus x & = & \sigma_t \setminus x & \text{(Lemma 3)} \\ & = & \sigma'_t \setminus x & \text{(induction assumption)} \\ & = & \sigma'_{t+1} \setminus x & \text{(semantics of } \texttt{skip}\text{)} \end{array}$$

Next we need to show that $\sigma_{t+1}(x) = \sigma'_{t+1}(x)$. First we make an observation:

The assumption $C\mathcal{R}_3C'$ implies that $\forall i < k < t : p_k \neq p$, so the side condition (from p_t)

$$p_t \models \overleftarrow{A} X \overleftarrow{A}(\neg def(x) \wedge \forall y \in vars(e) : \neg def(y) \wedge \neg entry\ W\ node(p))$$

implies that $\mathcal{M}_{cf}(\pi), p_k \models \neg def(x)$ and $\mathcal{M}_{cf}(\pi), p_k \models \neg def(y)$ for all $y \in vars(e)$ and $i < k < t$. Clearly the same holds for $\mathcal{M}_{cf}(\pi')$, since the statements at p_k in π will be identical to the statement in π' at p_k, as $p_k \notin \{p, q\}$ for $i < k < t$. By Lemma 2 variable x and $vars(e)$ does not change from $i+1$ to t: $\sigma'_{i+1}(x) = \sigma'_k(x)$ and $\sigma_{i+1}|_{vars(e)} = \sigma_k|_{vars(e)}$ for all $i < k \leq t$.

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

$$\begin{array}{rcll} \sigma_{t+1}(x) & = & [\![e]\!]\sigma_t & \text{(semantics of } \mathcal{I}_{p_k} = (x{:=}e)\text{)} \\ & = & [\![e]\!]\sigma_t|_{vars(e)} & \text{(Lemma 1)} \\ & = & [\![e]\!]\sigma_{i+1}|_{vars(e)} & (1^{st} \text{ argument above}) \\ & = & [\![e]\!]\sigma_i|_{vars(e)} & \text{(semantics of } \mathcal{I}_{p_i} = (\texttt{skip})\text{)} \\ & = & [\![e]\!]\sigma'_i|_{vars(e)} & \text{(induction assumption)} \\ & = & [\![e]\!]\sigma'_i & \text{(Lemma 1)} \\ & = & \sigma'_{i+1}(x) & \text{(semantics of } \mathcal{I}'_{p_i} = (x{:=}e)\text{)} \\ & = & \sigma'_t(x) & (2^{nd} \text{ argument above}) \\ & = & \sigma'_{t+1}(x) & \text{(semantics of } \mathcal{I}'_{p_t} = (\texttt{skip})\text{)} \end{array}$$

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

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

Suppose that the program π does not terminate on input v. Any finite computation prefix of π will have a corresponding computation prefix in π' such that the labels in the computation prefixes are pairwise equivalent. Thus π' will not terminate on input v either. The reverse direction follows by symmetry. Otherwise π and π' both terminates on input v. Consider the store at the `write` statement: $\mathcal{I}_m = \mathcal{I}'_m = (\texttt{write y})$. Suppose $C\mathcal{R}C'$ for t=m by case 3. Then by the side condition there exists an $i > m$ such that $p_i \models node(q)$, i.e. $p_i = q$. But by the semantics $p_i = exit(\pi) \neq q$ for all $i > t$ leading to a contradiction. Thus either case 1 or 2 must hold. In either case

$\sigma_t = \sigma'_t$ implying $\sigma_t(\mathrm{y}) = \sigma'_t(\mathrm{y})$. It follows by Lemma 3 that the transformation is semantics preserving: $[\![\pi]\!] = [\![\pi']\!]$.

□

9 Summary

In this paper a framework for describing program transformations as conditional rewrite rules has been presented. Specifying program transformations as rewrite rules with temporal logic side conditions is a relatively new approach to proving correctness of program transformations [5, 4].

The main focus in this paper has been on the *correctness* of program transformations rather than orthogonal issues. In order to apply a transformation in the present framework, the overall transformation strategy must decide on a particular program point, before the side condition is verified. In practice this leads to inefficient transformation strategies[2], if implemented naively. A possible solution[4] is to extend the definition of CTL to admit free variables. The model checker would return a set of substitutions satisfying the side condition, rather than only validating the side condition for a particular label. Each substitution corresponds to an instantiation of the rule schema at particular program points for which the side condition is satisfied. The transformation strategy could then select one of the possible (legal) substitutions and immediately construct the transformed program. The overall development is only slightly more involved than the present paper, and the results from the paper should be easily adaptable. However the simpler framework is sufficient when the main focus is purely on the correctness of program transformations.

The language treated, was a simplistic imperative language without procedure calls or exceptions. In order to allow specification of programs with these language features, the framework would have to be extended to account for the different control flow. This would most likely complicate the proofs since more language constructs are introduced, which would have to be reasoned about in the proofs. However extending the framework to include procedures and exceptions, as well as extensions to object oriented languages, remain an interesting area of future investigations.

10 Acknowledgements

The author would like to thank the following people for their kind help during the writing of the present paper. Neil Jones for his guidance and for getting the author started on using temporal logic to prove correctness of program transformations. David Lacey and Eric Van Wyk for inspiration and ideas. Chin Soon Lee for kindly proofreading the paper, resulting in numerous constructive suggestions.

[2] The transformation strategy is the order in which a number of program transformations are applied to the given subject program.

References

[1] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. Compilers: Principles, Techniques and Tools. Addison-Wesley, 1996.

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

[3] Joost-Pieter Katoen. 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] David Lacey, Neil D. Jones, Eric Van Wyk and Carl C. Frederiksen. Proving Correctness of Compiler Optimizations by Temporal Logic. (Unpublished), 2001.

[5] David Lacey and Oege de Moor. Imperative Program Transformation by Rewriting. Compiler Construction (CC2001) thread of ETAPS 2001.

[6] Flemming Nielson, Hanne R. Nielson and Chris Hankin. Principles of Program Analysis. Springer-Verlag, 1999.

[7] Oege de Moor, David Lacey and Eric Van Wyk. Universal Regular Path Queries. (Submitted to HOSC), 2001.