Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 176 (2007) 79–95
www.elsevier.com/locate/entcs

Structuring Optimizing Transformations and Proving Them Sound

Aditya Kanade[1] Amitabha Sanyal[2] Uday Khedker[3]

Dept. of Computer Science and Engineering, IIT Bombay.

Abstract

A compiler optimization is sound if the optimized program that it produces is semantically equivalent to the input program. The proofs of semantic equivalence are usually tedious. To reduce the efforts required, we identify a set of common *transformation primitives* that can be composed sequentially to obtain specifications of optimizing transformations. We also identify the conditions under which the transformation primitives preserve semantics and prove their sufficiency. Consequently, proving the soundness of an optimization reduces to showing that the soundness conditions of the underlying transformation primitives are satisfied.

The program analysis required for optimization is defined over the input program whereas the soundness conditions of a transformation primitive need to be shown on the version of the program on which it is applied. We express both in a temporal logic. We also develop a logic called *temporal transformation logic* to correlate temporal properties over a program (seen as a Kripke structure) and its transformation.

An interesting possibility created by this approach is a novel scheme for validating optimizer implementations. An optimizer can be instrumented to generate a trace of its transformations in terms of the transformation primitives. Conformance of the trace with the optimizer can be checked through simulation. If soundness conditions of the underlying primitives are satisfied by the trace then it preserves semantics.

Keywords: Optimization specification, Formal verification, Translation validation

1 Introduction

Modern compilers are equipped with sophisticated optimizations. An optimization is sound if the optimized program that it produces is semantically equivalent to the input program. The issue of soundness of optimizers has been addressed as verification of specifications [9,10] and translation validation [11,18]. The former approach seeks to guarantee soundness of specifications but does not address soundness of their implementations, while the latter approach checks the soundness of an optimizer on a run-by-run basis. The latter approach requires some heuristics [11] or hints from the compiler [18].

[1] Email: aditya@cse.iitb.ac.in
[2] Email: as@cse.iitb.ac.in
[3] Email: uday@cse.iitb.ac.in

doi:10.1016/j.entcs.2006.06.018

The proofs of semantic equivalence are usually tedious. To reduce the efforts required, we identify a set of common *transformation primitives* that can be composed sequentially to obtain specifications of optimizing transformations. We also identify the conditions under which the transformation primitives preserve semantics. For example, common subexpression elimination, partial redundancy elimination, and loop invariant code motion replace some occurrences of an expression by a variable. Although they may select different application points, the same soundness condition has to be satisfied at each of the application points: The variable must have the same value as that of the expression being replaced. Such a soundness condition guarantees semantics preservation under the respective transformation is a one time proof and is independent of any optimization. The primitives are small-step transformations as compared to the optimizations and hence the semantics preservation proofs are easier. This approach reduces proving the soundness of an optimization to merely showing that the soundness conditions of the underlying transformation primitives are satisfied. This is much simpler than directly proving semantics preservation for each optimization.

We specify the program analysis and the soundness conditions in first-order logic. They are interpreted over control flow representation of programs. The properties which relate information along control flow paths are expressed in a temporal logic. The program analysis is defined over the input program. Depending on its position in the transformation sequence, the soundness conditions of a transformation primitive need to be shown either on the input program or its appropriate transformation. We develop a logic called Temporal Transformation Logic (TTL) to correlate temporal properties over a program seen as a Kripke structure and its transformation.

Based on our approach of identifying transformation primitives and their soundness conditions, we suggest a novel validation scheme: An optimizer can be instrumented to generate a trace of its execution as a sequence of appropriately instantiated primitives. An execution preserves semantics if (1) the optimized program matches the output obtained after simulating the trace on the input program and (2) the soundness conditions of the transformation primitives used in the trace are satisfied.

The rest of the paper is organized as follows: Section 2 introduces the specification mechanism. Section 3 describes the verification technique and introduces TTL. Section 4 proposes a validation scheme. Section 5 describes how proof obligations can be automatically generated from the specifications of optimizations in PVS. Section 6 reviews related work. Section 7 concludes the paper and proposes future directions.

2 Specifying Optimizations

The optimizations are specified over an abstraction based on control flow graph representation of three-address code. We use PVS [14] as the specification and verification framework. PVS language is based on typed higher-order logic [13]. We

explain PVS language features wherever required.

2.1 Abstraction of Programs

A *program* is a directed graph with a single entry and a single exit. Each node denotes a control location called a program *point* and holds a *statement*. Each program point has at least one successor.[4] The entry point has no predecessors whereas the exit point has only a self-loop. *Variable*s and *constant*s form data part of a program. At present, we do not consider arrays and pointers. The *expression*s are formed from *operator*s and variable or constant type *operand*s. The operators are uninterpreted functions. We consider four kinds of statements: (1) SKIP is a "no-operation", (2) ASSIGN(Lhs: *variable*, Rhs: *expression*) is an "assignment", (3) ITE(Condition: *operand*, Tb, Fb: *point*) is an "if–then–else" statement where Tb and Fb are respectively targets of "if" and "else" branches, and (4) HALT halts the execution of a program. The "goto"s are modeled as directed edges.

The type *program* is defined as a record with four fields:

$$\mathit{program}: \text{TYPE} = \big[\#\ \mathsf{cfg}: \mathit{Graph}[\mathit{point}],\ \mathsf{entry},\ \mathsf{exit}: (\mathit{cfg}\text{`}S),\ \mathsf{L}: [(\mathit{cfg}\text{`}S) \to \mathit{statement}]\ \#\big]$$

cfg is the control flow graph whose nodes belong to a set of program points S and edges are given by a relation τ: S × S. They are respectively referred to as cfg`S and cfg`τ. entry and exit are respectively entry and exit points. The function L maps a program point in cfg`S to a statement.

The abstraction is well-defined if its control flow and contents are consistent with each other. For example, if the statement at program point p is $\mathsf{ITE}(\mathsf{c}, \mathsf{p}_1, \mathsf{p}_2)$ then p_1 and p_2 should be the only successors of p. These constraints are satisfied by a predicate subtype (*Program*) of *program* where Program: *program* → *bool* (definition omitted). In the running text, a program which satisfies this predicate is simply referred to as "program".

An aside on PVS typing. Given a type T and a predicate $\varphi : T \to \mathit{bool}$, (φ) denotes a predicate subtype of T. It is the set of entities from T that satisfy the predicate φ. PVS allows dependent types where the types are defined in terms of the components declared earlier. $(\mathit{cfg}\text{`}S)$ is a dependent predicate subtype indicating the set of program points S of cfg. Though cfg and S are field names, they are italicized when used in a type declaration. As a convention, we always italicize the types.

2.2 Computational Tree Logic with Branching Past

We use computational tree logic with branching past (CTL_{bp}) [8] for specifying global program properties. *Kripke structures* are used as models for CTL_{bp}. A Kripke structure is a directed graph whose nodes are labeled with atomic propositions. Formally, a Kripke structure $\mathsf{M} = (S, R, P, L)$, where S is a finite non empty set of states. $R : S \times S$ is a transition relation which is total in its first element.

[4] This is required for modeling programs as Kripke structures and is explained in section 2.2.

$$\begin{aligned}
&\mathsf{Transp}(\mathsf{prog},\mathsf{e})(\mathsf{p}):bool &&= \mathsf{Assign?}(\mathsf{prog}`\mathsf{L}(\mathsf{p})) \Rightarrow \mathsf{Lhs}(\mathsf{prog}`\mathsf{L}(\mathsf{p})) \notin \mathsf{VOperands}(\mathsf{e})\\
&\mathsf{Antloc}(\mathsf{prog},\mathsf{e})(\mathsf{p}):bool &&= \mathsf{Assign?}(\mathsf{prog}`\mathsf{L}(\mathsf{p})) \wedge \mathsf{Rhs}(\mathsf{prog}`\mathsf{L}(\mathsf{p})) = \mathsf{e}\\
&\mathsf{Dom}(\mathsf{prog},\mathsf{q})(\mathsf{p}):bool &&= \mathsf{AP}(\mathsf{prog},(\mathsf{p}))(\mathsf{q})\\
&\mathsf{DomS}(\mathsf{prog},\mathsf{xs})(\mathsf{p}):bool &&= \forall(\mathsf{q}:(xs)):\mathsf{Dom}(\mathsf{prog},\mathsf{q})(\mathsf{p})\\
&\mathsf{Scc}(\mathsf{prog},\mathsf{xs}):bool &&= \forall(\mathsf{p},\mathsf{q}:(xs)):\mathsf{EU}(\mathsf{prog},\mathsf{xs},(\mathsf{q}))(\mathsf{p})\\
&\mathsf{Loop}(\mathsf{prog})(\mathsf{xs}):bool &&= \mathsf{Scc}(\mathsf{prog},\mathsf{xs}) \wedge \exists(\mathsf{p}:(xs)):\mathsf{DomS}(\mathsf{prog},\mathsf{xs})(\mathsf{p}) \wedge\\
& && \quad \forall(\mathsf{p}:(xs)):\neg\mathsf{DomS}(\mathsf{prog},\mathsf{xs})(\mathsf{p}) \Rightarrow \mathsf{AY}(\mathsf{prog},\mathsf{xs})(\mathsf{p})\\
&\%\mathsf{Header}(\mathsf{prog}:(Program),\mathsf{xs}:(Loop(prog)))(\mathsf{p}:(prog`cfg`S)):bool\\
&\mathsf{Header}(\mathsf{prog},\mathsf{xs})(\mathsf{p}):bool &&= \mathsf{xs}(\mathsf{p}) \wedge \mathsf{DomS}(\mathsf{prog},\mathsf{xs})(\mathsf{p})\\
&\%\mathsf{LInv}(\mathsf{prog}:(Program),\mathsf{xs}:(Loop(prog)))(\mathsf{e}:(expressions(prog))):bool\\
&\mathsf{LInv}(\mathsf{prog},\mathsf{xs})(\mathsf{e}):bool &&= \forall(\mathsf{p}:(xs)):\mathsf{Transp}(\mathsf{prog},\mathsf{e})(\mathsf{p}) \wedge \big(\mathsf{Header}(\mathsf{prog},\mathsf{xs})(\mathsf{p})\\
& && \quad \Rightarrow \mathsf{AU}(\mathsf{prog},\mathsf{Transp}(\mathsf{prog},\mathsf{e}),\mathsf{Antloc}(\mathsf{prog},\mathsf{e}))(\mathsf{p})\big)\\
&\%\mathsf{Invs}(\mathsf{prog}:(Program),\mathsf{xs}:(Loop(prog)),\mathsf{e}:(LInv(prog,xs))):set[(xs)]\\
&\mathsf{Invs}(\mathsf{prog},\mathsf{xs},\mathsf{e}):set\big[(xs)\big] &&= \{\mathsf{p}:(xs) \mid \mathsf{Antloc}(\mathsf{prog},\mathsf{e})(\mathsf{p})\}
\end{aligned}$$

Fig. 1. Specification of program analyses pertaining to Loop Invariant Code Motion

To guarantee totality, in the program abstraction, every program point is required to have a successor. P is a set of atomic propositions. $L : S \to 2^P$ is a labeling function which associates states to propositions.

We view programs as Kripke structures. The program points $\mathsf{cfg}`\mathsf{S}$ form the set of states $S$. The set of edges $\mathsf{cfg}`\tau$ form the transition relation R. The propositions are generalized to predicates over program points.

CTL$_{bp}$ has propositional connectives and temporal operators. The future operators describe properties of descendants of a state. The past operators describe properties of ancestors of a state. Consider formulae φ and ψ. The future operators are $\mathsf{X}(\varphi)$ ("neXt time φ holds"), $\mathsf{U}(\varphi,\psi)$ ("φ holds Until ψ"), and $\mathsf{F}(\varphi)$ ("φ holds sometime in Future"). The past operators are $\mathsf{Y}(\varphi)$ ("φ holds Yesterday"), $\mathsf{S}(\varphi,\psi)$ ("φ holds Since ψ"), and $\mathsf{P}(\varphi)$ ("φ holds sometime in Past"). They are prefixed with path quantifiers E ("for some path") or A ("for all paths"). For example, $\mathsf{AU}(\varphi,\psi)$ states that "along all (forward) paths φ holds until ψ" or $\mathsf{EP}(\varphi)$ states that "along some (backward) path φ holds sometime". Since we transform a program step-by-step, we have different versions of it. To distinguish between the interpretations of CTL$_{bp}$ formulae over different programs, we parameterize them with respective programs.

2.3 Specifying Analyses

We use loop invariant code motion to explain the specification mechanism. We first specify analyses for identifying loops and determining whether an expression is invariant within a loop. The specifications are given in Fig. 1. For readability, we do not explicate the types. However, they are explained in the running text. Some interesting type signatures are inserted in the specification as comments, starting with %.

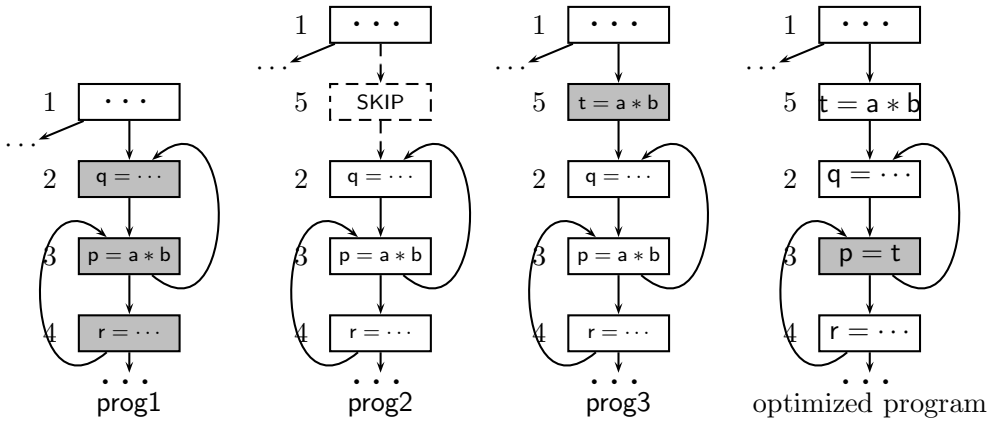

Fig. 2. An example of Loop Invariant Code Motion

Consider a program prog and an expression e in it. The expression e is transparent at a program point p if none of its operands is assigned to at p. This is defined as the predicate Transp and is in curried form. Assign? is the recognizer for assignment statements. Lhs is the accessor for left-hand side variable of an assignment. VOperands returns the set of variable type operands of an expression. The expression e is locally anticipatable at a program point p if the statement at p is an assignment whose right-hand side (accessible by Rhs) is e. This is defined as the predicate Antloc.

We now specify control flow analyses for identifying loops. A program point p dominates a program point q if along all backward paths from q, p is reachable. This property is defined as the predicate Dom. Let xs be a set of program points of prog. A program point p dominates xs if it dominates each member of xs. This is defined as the predicate DomS. The set xs is a strongly connected component if for every pair of program points in it there is a (non-empty) directed path whose intermediate program points also belong to xs. This is defined as the predicate Scc. A set of program points xs is a Loop if it is a strongly connected component, there is a program point in xs which dominates it, and the predecessors of other program points of xs are also in xs. The program point in a loop which dominates it is called its Header.

Let xs be a loop in program prog. An expression e is invariant within xs if it is transparent at all program points of xs and is anticipatable at the Header of xs, that is along all forward paths from the header, e is transparent until it is locally anticipatable. This is given as the predicate LInv. The function Invs gives program points of xs that contain occurrences a loop invariant expression e. These occurrences evaluate to the same value in every iteration and hence can be moved out of the loop.

2.4 Specifying Transformations

Consider a program prog1 shown in Fig. 2. The set $xs = \{2, 3, 4\}$ is a Loop with program point 2 as its Header. According to the analyses defined in Fig. 1, the

expression a ∗ b is invariant within xs and is locally anticipatable at program point 3. We want to hoist a ∗ b to the incoming edges of the loop header except along the looping edge. This is done by transforming the program step-by-step as follows:

(i) Split the edge ⟨1, 2⟩ by adding a *new program point*, say 5, containing a SKIP statement. A new program point is distinct from program points of the subject program.

(ii) Let t be a *new variable* with respect to prog2. A new variable does not appear anywhere in the subject program. Insert an assignment statement t = a ∗ b (simplified notation for ASSIGN(t, a ∗ b)) at program point 5.

(iii) Replace the occurrence of a ∗ b at program point 3 by the variable t.

The transformation is formally specified in Fig. 3. Let prog1 be a program, xs be a loop in it, and e be an invariant expression within xs. The predicate hdr denotes the headers of xs. By definition, a loop has only one header. The set predsNotInLoop denotes predecessors of the header which are not in xs. The function SE is a transformation primitive. Given a program and two sets of program points, it splits the edges going from the program points in the first set to those in the second set. It inserts new program points containing SKIP statements. Here, it takes the program prog1 and splits the edges from predsNotInLoop to the loop header. The transformed program is prog2.

Let invoccurs be the program points in xs where the expression e occurs. They are identified over prog1. Let t be a new variable w.r.t. prog2. newpoints are the new program points inserted by the first transformation. In prog2, they are predecessors of the loop header but are not in the loop. The function IA is a transformation primitive which takes a variable a and an expression b and inserts an assignment statement ASSIGN(a, b) at the given program points in the subject program. Here, it inserts ASSIGN(t, e) at newpoints in prog2. The resulting program is prog3.

The function RE is a transformation primitive which takes a program and replaces expressions at the given program points by a given variable. Here, it transforms program prog3 by replacing invariant occurrences of e at invoccurs by the variable t. This accomplishes loop invariant code motion.

$$
\begin{array}{l}
\%\mathsf{LICM}(\mathsf{prog1}:(\mathit{Program}),\mathsf{xs}:(\mathit{Loop}(\mathit{prog1})),\mathsf{e}:(\mathit{LInv}(\mathit{prog1},\mathit{xs}))):(\mathit{Program}) \\
\quad \mathsf{LICM}(\mathsf{prog1},\mathsf{xs},\mathsf{e}):(\mathit{Program}) \;= \\
\qquad\qquad \text{LET } \mathsf{hdr} \;= \mathsf{Header}(\mathsf{prog1},\mathsf{xs}), \\
\qquad\qquad\quad \mathsf{preds} \;= \mathsf{EX}(\mathsf{prog1},\mathsf{hdr}), \\
\qquad\qquad\quad \mathsf{predsNotInLoop} \;= (\mathsf{preds} \setminus \mathsf{xs}), \\
\qquad\qquad\quad \underline{\mathsf{prog2}} \;= \underline{\mathsf{SE}(\mathsf{prog1},\mathsf{predsNotInLoop},\mathsf{hdr})}, \\
\qquad\qquad\quad \overline{\mathsf{invoccurs}} \;= \overline{\mathsf{Invs}(\mathsf{prog1},\mathsf{xs},\mathsf{e})}, \\
\qquad\qquad\quad \mathsf{t} \;= \mathsf{NEWVAR}(\mathsf{prog2}), \\
\qquad\qquad\quad \mathsf{newpoints} \;= (\mathsf{prog2`cfg`S} \setminus \mathsf{prog1`cfg`S}), \\
\qquad\qquad\quad \underline{\mathsf{prog3}} \;= \underline{\mathsf{IA}(\mathsf{prog2},\mathsf{newpoints},\mathsf{t},\mathsf{e})} \\
\qquad\qquad \text{IN} \qquad\qquad\quad \underline{\overline{\mathsf{RE}(\mathsf{prog3},\mathsf{invoccurs},\mathsf{t})}}
\end{array}
$$

Fig. 3. Specification of Loop Invariant Code Motion transformation

```
RE(prog,points,v): (Program) =
  (# cfg  := prog‘cfg, entry := prog‘entry, exit := prog‘exit,
     L    := λ(p:(prog‘cfg‘S)):
               IF (p ∈ points) THEN ASSIGN(Lhs(prog‘L(p)),BASE(V(v)))
                 ELSE prog‘L(p) ENDIF #)

SoundRE(prog,points,v): bool =
 ∀(p:(points)):Assign?(prog‘L(p)) ∧
  LET e = Rhs(prog‘L(p)) IN v ∉ VOperands(e) ∧
   AY(prog,AS(prog,TranspNDef(prog,e,v),AssignStmt(prog,v,e)))(p)
```

Fig. 4. Definition of transformation primitive RE and its soundness conditions

2.5 Defining Transformation Primitives and their Soundness Conditions

The transformation primitives are usually easy to define and their soundness conditions simple to characterize. The transformation primitive RE is defined in Fig. 4. It replaces expressions at program points points in a program prog by a base expression constructed from a variable v. The constructor BASE gives an expression which merely consists of an operand. The constructor V constructs a variable type operand. The transformation primitive RE modifies only labeling function L of the subject program. The last transformation depicted in Fig. 2 is an application of RE transformation.

The soundness conditions of RE are defined as the predicate SoundRE in Fig. 4. Let p ∈ points. The transformations represented by RE are sound if: (1) p contains an assignment statement. (2) If e is the expression computed at p then the given variable v is not its operand. (3) Along all backward paths starting with the predecessors of p, the expression e is transparent and either the variable v is not defined or the expression assigned to it is e (denoted by TranspNDef) until a statement assigning e to v (denoted by AssignStmt) is encountered. This ensures that along all paths reaching p, the variable v has the same value as the expression e. In Appendix A, we prove that given the soundness conditions soundRE, RE preserves semantics of the input program.

3 Verifying Soundness of the Specifications

In the previous section, we identified some primitive transformations and expressed the optimizing transformations by composing them sequentially. We discussed the conditions under which the primitives preserve semantics. In this section, we discuss the verification scheme, TTL, and argue about soundness of LICM specification given in Fig. 3.

3.1 Verification Scheme

Consider an optimizing transformation T defined in terms of the transformation primitives $T_1, \cdots, T_k$:

$$T(M_1) \stackrel{\triangle}{=} \text{LET } M_2 = T_1(M_1, \pi_1), \cdots, M_k = T_{k-1}(M_{k-1}, \pi_{k-1}) \text{ IN } T_k(M_k, \pi_k)$$

where M_1 is the abstraction of the input program. A transformation T_i is applied to an abstraction M_i at program points π_i. Other parameters of the transformation primitives are implicit.

Let $\varphi_1, \ldots, \varphi_k$ be the soundness conditions of the primitives $T_1, \ldots, T_k$. If a transformation primitive T_i is applied to a subset of $\varphi_i(M_i)$, then T_i preserves semantics of M_i. Therefore, if we show that $\pi_i \subseteq \varphi_i(M_i)$, $1 \leq i \leq k$, then each of the constituent transformations preserves semantics. This implies that the overall transformation T also preserves semantics.

The program points π_i are identified over some program M_j, $j \leq i$, whereas the safe application points $\varphi_i(M_i)$ are identified over M_i only. Thus, we have to correlate program properties which define these points across different versions of the input program. To correlate temporal properties in such a manner, we develop a logic called temporal transformation logic (TTL). It relates temporal formulae whose outermost operators are the same. To prove the non-temporal properties, we use properties of the preceding transformations. In section 3.4, we discuss two non-temporal proof obligations viz. (A) and (B) and argue as to how they can be discharged.

3.2 Temporal Transformation Logic

A *K-transformation* $f : \mathcal{M} \times 2^S \to \mathcal{M}$ where $\mathcal{M}$ is the set of Kripke structures. Let $\mathsf{M}' = (S', R', P', L') = f(\mathsf{M}, \pi)$ where $\pi \subseteq S$. The K-transformations are classified depending on how they change structure of the input Kripke structure. Below we discuss a transformation and an inference rule associated with it that is relevant to this paper.

Consider two states i and j of M such that i is a predecessor of j. We want to add a *new* state k as a predecessor of j and a successor of i. The new state k is distinct from the states of M. We add an edge from i to k and an edge from k to j. The edge from i to j is deleted whereas other edges of M are preserved. We call this transformation *node addition* or *edge splitting*.

As stated in section 2.2, we parameterize CTL formulae with Kripke structures to distinguish between their interpretations on different Kripke structures. Let N be an atomic proposition that denotes the new states. Let Δ denote EX, AX, EY, and AY. Let ∇ denote EU, AU, ES, and AS. The transformed Kripke structure M' is obtained by adding some states to M. The rule NA gives inference rules for node addition transformation.

$$\text{(NA)}\quad \frac{\begin{array}{rrcl} \vdash & \varphi \vee N & \Longrightarrow & \varphi' \\ \vdash & \psi & \Longrightarrow & \psi' \end{array}}{\begin{array}{rrcl} \vdash & \Delta(\mathsf{M}, \varphi) & \Longrightarrow & \Delta(\mathsf{M}', \varphi') \\ \vdash & \nabla(\mathsf{M}, \varphi, \psi) & \Longrightarrow & \nabla(\mathsf{M}', \varphi', \psi') \end{array}}$$

Lemma 3.1 NA *is sound.*

Proof. Available in [7] □

If there is no structural change involved in a transformation then also above rule can be used. The set of new states N is empty in that case and the implication $N \implies \varphi'$ is vacuously true.

TTL has inference rules for other classes of transformations like node splitting, node merging, node deletion, edge addition, and edge deletion. These can be composed to express various kinds of transformations. More on TTL is available in [7].

3.3 Why does LICM *Preserve Semantics?*

The soundness of LICM specified in Fig. 3 can be informally justified as follows:

(i) The first transformation SE does not add or delete any control flow paths and inserts just SKIP statements. Hence the transformed program prog2 is semantically equivalent to prog1.

(ii) Since t is a new variable w.r.t. prog2, it does not modify any reaching definitions in prog2. The expression e is anticipatable at the loop header. The first transformation inserts newpoints as predecessors of the loop header. Hence the second transformation which inserts an assignment ASSIGN(t, e) at newpoints does not give rise to computation of any new value along any path and thus preserves semantics of prog2.

(iii) In the second transformation, an assignment ASSIGN(t, e) is inserted along all incoming edges of the loop header except the looping edge. Within the loop, neither the variable t nor any of the variable operands of the expression e are assigned. Hence t has the same value as e along all incoming paths to a program point in invoccurs. Therefore, the occurrence of e at such a program point can be replaced by t while preserving semantics of prog3.

3.4 An Example Proof of Soundness

We now prove soundness of the last transformation RE in LICM which is applied on prog3. It replaces the occurrences of the loop invariant expression e at the program points invoccurs within the loop xs with the variable t. We have to show that SoundRE(prog3, invoccurs, t) holds.

Let $\mathsf{p} \in \mathsf{invoccurs}$ and $\mathsf{e}' = \mathsf{Rhs}(\mathsf{prog3`L(p)})$. We have to show that p which is a program point in prog1 is also in prog3. The program points invoccurs are defined over prog1. The same program points are identified in the transformed programs prog2 and prog3 by invoccurs since the transformations do not delete any program points. Similar is the case for xs, hdr, and newpoints. PVS generates type correctness conditions (TCCs) for them. They are discharged by rewriting the definitions of the transformation primitives. From the definition of SoundRE in Fig. 4, we have the following proof obligations:

(A) Assign?(prog3`L(p)))

(B) $\mathsf{t} \notin \mathsf{VOperands}(\mathsf{e}')$

(C) AY(prog3,AS(prog3,TranspNDef(prog3, e′, t),AssignStmt(prog3, t, e′)))(p)

Proofs of (A) and (B):

(A) is proved by rewriting the definitions of the first two transformations SE and IA. SE inserts skip statements at the new points newpoints and IA replaces these skips by assignments ASSIGN(t, e). From the definitions of Invs, LInv, and Antloc in Fig. 1, we deduce that in prog1, the statement at p is an assignment statement. Since the first two transformations do not change contents of p, in prog3 it holds the same assignment statement as in prog1.

Clearly, the expression e′ is same as the expression e = Rhs(prog1‘L(p)). Since t is a new variable w.r.t. prog2, it cannot be an operand of any expression in prog2. prog2 is a transformation of prog1 and e is an expression in prog1, therefore, t ∉ VOperands(e). (B) follows because e = e′.

Proof of (C):

Since e = e′, we replace e′ by e in (C). We have to show that in prog3, for all predecessors of p along all backward paths, the expression e is transparent and either the variable t is not defined or the expression assigned to it is e until a statement assigning e to t is encountered.

Auxiliary Results:

(i) The first transformation SE inserts newpoints as predecessors to the loop header by splitting its incoming edges except the looping edge. The second transformation IA does not change control flow of prog2. Using NA,

$$\mathsf{hdr} \implies \mathsf{AY}(\mathsf{prog3}, \mathsf{newpoints} \vee \mathsf{xs}) \tag{1}$$

(ii) From the definition of LInv in Fig. 1, the expression e is transparent in the loop xs in prog1. The first transformation SE inserts only SKIP statements. The second transformation IA inserts statements assigning the expression e to the new variable t at newpoints only. Hence

$$\mathsf{xs} \vee \mathsf{newpoints} \implies \mathsf{TranspNDef}(\mathsf{prog3}, \mathsf{e}, \mathsf{t}) \tag{2}$$

$$\mathsf{newpoints} \iff \mathsf{AssignStmt}(\mathsf{prog3}, \mathsf{t}, \mathsf{e}) \tag{3}$$

(iii) The loop header belongs to the loop: hdr ⟹ xs. From (2),

$$\mathsf{hdr} \implies \mathsf{TranspNDef}(\mathsf{prog3}, \mathsf{e}, \mathsf{t}) \tag{4}$$

From (1) and (3),

$$\mathsf{hdr} \implies \mathsf{AY}(\mathsf{prog3}, \mathsf{AssignStmt}(\mathsf{prog3}, \mathsf{t}, \mathsf{e}) \vee \mathsf{xs}) \tag{5}$$

(iv) Our program abstraction has a unique entry. It has no incoming edges and is reachable along all backward paths. Since in CTL_{bp} the past is finite, we have the following (derived) proof rule:

$$(\beta \implies \mathsf{AS}(\mathsf{M}, \varphi, \psi)) \wedge (\psi \implies \varphi \wedge \mathsf{AY}(\mathsf{M}, \alpha \vee \beta)) \vdash (\beta \implies \mathsf{AS}(\mathsf{M}, \varphi, \alpha)) \tag{6}$$

Main Derivation:

{Program points in the loop xs are dominated by the loop header hdr in prog1.}

$$\mathsf{xs} \implies \mathsf{AS}(\mathsf{prog1}, \mathsf{xs}, \mathsf{hdr})$$

{The first transformation SE inserts newpoints. Using NA,}

$$\mathsf{xs} \implies \mathsf{AS}(\mathsf{prog2}, \mathsf{xs} \vee \mathsf{newpoints}, \mathsf{hdr})$$

{The second transformation IA does not change control flow of prog2.}

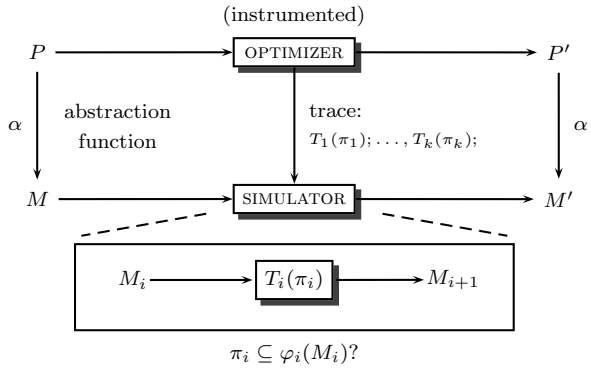

Fig. 5. Validate an optimization against a generated trace

$$\mathsf{xs} \implies \mathsf{AS}(\mathsf{prog3}, \mathsf{xs} \vee \mathsf{newpoints}, \mathsf{hdr})$$

{From (2), (3), (4), and (5), using (6)}

$$\mathsf{xs} \implies \mathsf{AS}(\mathsf{prog3}, \mathsf{TranspNDef}(\mathsf{prog3}, \mathsf{e}, \mathsf{t}), \mathsf{AssignStmt}(\mathsf{prog3}, \mathsf{t}, \mathsf{e}))$$

{All predecessors of a program point in xs belong to xs or it is the loop header.}

$$\mathsf{xs} \implies \mathsf{AY}(\mathsf{prog3}, \mathsf{AS}(\mathsf{prog3}, \mathsf{TranspNDef}(\mathsf{prog3}, \mathsf{e}, \mathsf{t}), \mathsf{AssignStmt}(\mathsf{prog3}, \mathsf{t}, \mathsf{e})))$$

By definition, $\mathsf{invoccurs} \implies \mathsf{xs}$. Since $\mathsf{p} \in \mathsf{invoccurs}$,

$$\mathsf{AY}(\mathsf{prog3}, \mathsf{AS}(\mathsf{prog3}, \mathsf{TranspNDef}(\mathsf{prog3}, \mathsf{e}, \mathsf{t}), \mathsf{AssignStmt}(\mathsf{prog3}, \mathsf{t}, \mathsf{e})))(\mathsf{p}) \tag{7}$$

Similarly, it can be shown that the soundness conditions of other transformations are satisfied by LICM specification.

4 A Possible Approach for Validating Optimizers

While implementations can be validated against their provenly sound specifications, our approach of identifying transformation primitives and their soundness conditions suggests a novel validation scheme shown in Fig. 5: Although an optimizer may not have been implemented using the transformation primitives, it can be instrumented to generate a trace of its execution as a sequence of appropriately instantiated primitives. For example, the trace of the transformations in Fig. 2 is $\mathsf{SE}(\{1\},\{2\});\ \mathsf{IA}(\{5\},\mathsf{t},\mathsf{a}*\mathsf{b});\ \mathsf{RE}(\{3\},\mathsf{t})$.

The validation scheme consists of two parts: (1) The input program is abstracted and its transformed version is derived by simulating the trace on it. If the abstraction of the optimized program matches this abstraction then the trace is faithful to the optimization performed. Two abstractions match each other iff their control flow graphs are isomorphic and contents at corresponding program points are same, modulo a globally consistent renaming of variables. (2) It is checked whether the soundness conditions of the transformation primitives used in the trace are met on the respective abstractions. This establishes whether the trace preserves semantics. If both these checks succeed then the optimizer preserves semantics of the input program.

This approach does not require any knowledge of the analysis employed in the optimizer because it directly uses the application points provided by the optimizer. Since the program abstractions are finite, it is reasonable to assume existence of an automatic checker for the soundness conditions. Such a checker along with a simulator, an implementation of the abstraction function, and a procedure for matching abstractions constitute the trusted computing base.

5 Automatically Generating Proof Obligations

We use PVS for specifying optimizations and verifying them. Emacs provides a front-end for PVS. We have built an Emacs based utility for automatically generating soundness proof obligations from the specifications.

PVS parses and typechecks the specifications. It annotates the parse tree with typing information and keeps it in Common Lisp Object System (CLOS) format in PVS ILisp. PVS ILisp process runs as a subprocess of Emacs Lisp interpreter [2]. We probe the CLOS objects through Emacs interpreter using pvs-send-and-wait command. We identify the transformation primitives and the context in which they are used by walking the annotated parse tree of the specification. We then generate a PVS theory containing the soundness proof obligations for each of the transformation primitives used with the appropriate context. These proof obligations need to be discharged in order to prove that the optimization specification is sound. We are trying to develop high-level proof strategies so that these proof obligations can be discharged easily.

6 Related Work

Lacey et al. [9] specify optimizations as conditional rewrites whose enabling conditions are expressed in a temporal logic. They manually show the semantic equivalence of input and optimized programs. The rewrites are composed simultaneously. Due to this, as the number of rewrites increase, the proofs of semantic equivalence would get more complicated. Although program analyses are specified as temporal formulae, they prove semantic equivalence and hence cannot use temporal logic in the proofs. We also specify program analyses using a temporal logic. However, our transformation primitives are not general rewrites. Hence, it is possible to define their soundness conditions. In our case, the proofs are much simpler owing to the fact that common patterns of semantic equivalence proofs are discharged separately and only once for each primitive. The primitives are composed sequentially and hence increase in the number of transformations does not affect the provability adversely.

Lerner et al. [10] follow an approach similar to Lacey et al. [9]. They use a restricted temporal logic to express analyses and require a property called witness for correlating analysis with semantics of the program. They can then automatically derive and discharge the required proof obligations. We also generate proof obligations automatically. Though our proofs are not automated, they are mechanizable

to a large extent.

The translation validation approaches check semantic equivalence of input and optimized programs. They either use heuristics to guess the optimizations performed [11] or expect program annotations from compilers [15,18,19,1]. We do not address validation in as broad a sense as them and hence alleviate the checking of semantic equivalences of input and optimized programs.

Goldberg et al. [5] present a proof rule for reasoning about loop optimizations. They develop heuristics to determine which optimizations occurred and synthesize intermediate versions of the input program which may not have been generated by the compiler. This is similar to the approach in Fig. 5. However, we require an optimizer to generate a trace and check soundness conditions of the primitives used in it instead of semantics preservation.

The Verifix project [4,6] addresses the issue of construction of correct compilers. They distinguish between correctness of specifications and their implementations. They also consider the correctness when the machine resources are finite. In [3], the concept of program checking with certificates is introduced and applied to optimizing compiler back-ends. The compiler generates a trace of its search for optimal target code as a certificate of correctness.

Certifying compilers [12] generate type specifications and code annotations in addition to assembly code. This additional information is used to prove type and memory safety of target code. The proof obligations are generated, discharged, and checked outside of the compiler. [17,16] propose how compilers themselves can generate correctness proof for each run. These approaches require extensive instrumentation of compiler.

7 Conclusions and Future Work

We address issues regarding soundness of optimizations in two steps. We first identify transformation primitives common to several optimizations and define sufficient conditions for ensuring soundness of these primitives. We then specify an optimizing transformation as sequential compositions of appropriately chosen transformation primitives. Consequently, proving the soundness of an optimization reduces to showing that soundness conditions of the underlying transformation primitives are satisfied. This reduces the overall verification efforts: The proofs of semantic equivalence need to be done only once for each primitive and are independent of any particular optimization. The primitives are much simpler than the optimizing transformations and hence the semantics preservation proofs are easier.

Based on our approach of identifying transformation primitives and their soundness conditions, we suggest a novel validation scheme: An optimizer can be instrumented to generate a trace of its transformations in terms of the transformation primitives. Conformance of the trace with the optimizer can be checked through simulation. If soundness conditions of the underlying primitives are satisfied by the trace then it preserves semantics.

At present, our method can handle optimizations based on bit vector analyses.

We are developing TTL inference rules for several kinds of transformations. As part of future work, we would like to apply them for proving soundness of control flow optimizations that may change program structures significantly like loop unrolling, loop fusion, etc. Our framework needs to be extended for handling optimizations like constant propagation which are based on non-bit vector analyses.

Acknowledgement

Authors wish to thank Supratik Chakraborty for many insightful discussions. Authors are thankful to César Muñoz for clarifying PVS related doubts.

References

[1] C. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. Zuck. TVOC: A translation validator for optimizing compilers. In *Proceedings of CAV'05*, volume 3576 of *LNCS*, pages 291–295. Springer-Verlag, July 2005.

[2] J. Crow, S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert. Evaluating, testing, and animating PVS specifications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, March 2001.

[3] S. Glesner. Using program checking to ensure the correctness of compiler implementations. *Journal of Universal Computer Science*, 9(3):191–222, 2003.

[4] W. Goerigk, A. Dold, T. Gaul, G. Goos, A. Heberle, F. von Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Ruess, and W. Zimmermann. Compiler correctness and implementation verification: The Verifix approach. In poster session of CC'96. Technical Report LiTH-IDA-R-96-12, Linkping, Sweden, 1996.

[5] B. Goldberg, L. Zuck, and C. Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. In *Proceedings of COCV'04*, volume 132(1) of *ENTCS*, pages 53–71. Elsevier, May 2005.

[6] G. Goos and W. Zimmermann. Verification of compilers. In Ernst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design*, volume 1710 of *LNCS*, pages 201–230. Springer, 1999.

[7] A. Kanade, A. Sanyal, and U. Khedker. Temporal transformation logic. Technical Report TR-CSE-001-06, CSE, IIT Bombay, January 2006. Available at: http://www.cse.iitb.ac.in/~aditya/reports/TR-TTL.ps.

[8] O. Kupferman and A. Pnueli. Once and for all. In *Proceedings of LICS'95*, pages 25–35. IEEE Computer Society, 1995.

[9] D. Lacey, N. D. Jones, E. Wyk, and C. Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Proceeding of POPL'02*, pages 283–294. ACM Press, 2002.

[10] S. Lerner, T. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *Proceedings of PLDI'03*, pages 220–231. ACM Press, 2003.

[11] G. Necula. Translation validation for an optimizing compiler. In *Proceedings of PLDI'00*, pages 83–94. ACM Press, 2000.

[12] G. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of PLDI'98*, pages 333–344. ACM Press, June 1998.

[13] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. CSL, SRI International, Menlo Park, CA, September 1999.

[14] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. CSL, SRI International, Menlo Park, CA, September 1999.

[15] A. Pnueli, M. Siegel, and O. Shtrichman. Translation validation: From SIGNAL to C. In E.-R. Olderog and B. Steffen, editors, *Correct System Design*, volume 1710 of *LNCS State-of-art Survey*, pages 231–255. Springer Verliag, 1999.

[16] A. Poetzsch-Heffter and M. Gawkowski. Towards proof generating compilers. In *Proceedings of COCV'04*, volume 132(1) of *ENTCS*, pages 37–51, 2005.

[17] M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verification*, July 1999.

[18] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A translation validator for optimizing compilers. In *Proceedings of COCV'02*, volume 65(2) of *ENTCS*, 2002.

[19] L. Zuck, A. Pnueli, B. Goldberg, C. Barrett, Y. Fang, and Y. Hu. Translation and run-time validation of loop transformations. *Formal Methods in System Design (FMSD)*, November 2005.

A Proof of Semantics Preservation for RE

Let $Value$ be the domain of values that variables in a program can take. It contains a special element *true*. A function $[\![\mathsf{op}]\!] : Value^n \to Value$ denotes an n-ary operator op. A constant is a 0-ary operator. Consider a domain $Store = Variables(prog) \to Value$ which denotes valuations of variables in a program prog. A function $[\![\cdot]\!] : Expressions(prog) \times Store \to Value$ evaluates expressions in a program prog. Let $\sigma \in Store$.

$$[\![\mathsf{v}]\!]\sigma = \sigma(\mathsf{v}) \text{ where } \mathsf{v} \text{ is a variable}$$
$$[\![\mathsf{c}]\!]\sigma = [\![\mathsf{c}]\!] \text{ where } \mathsf{c} \text{ is a constant}$$
$$[\![\mathsf{op}(\mathsf{o}_1, \ldots, \mathsf{o}_2)]\!]\sigma = [\![\mathsf{op}]\!]([\![\mathsf{o}_1]\!]\sigma, \ldots, [\![\mathsf{o}_\mathsf{n}]\!]\sigma) \text{ where } \mathsf{o}_1, \ldots, \mathsf{o}_\mathsf{n} \text{ are operands}$$

Definition A.1 (Statement Semantics.) Consider a program prog and a domain $State = (\mathsf{prog}\text{`}\mathsf{S} \cup \{\odot\}) \times Store$ representing program states where $\odot$ is a special program point indicating termination, i.e., it does not hold any statement. The state transition relation $\rightsquigarrow$: $State \times State$ defines how statements affect the program state. Let $\mathsf{p} \in \mathsf{prog}\text{`}\mathsf{S}$ and $\sigma \in Store$.

(i) If $\mathsf{prog}\text{`}\mathsf{L}(\mathsf{p}) = \mathsf{SKIP}$ then $(\mathsf{p}, \sigma) \rightsquigarrow (\mathsf{p}', \sigma)$ where $(\mathsf{p}, \mathsf{p}') \in \mathsf{prog}\text{`}\mathsf{cfg}\text{`}\tau$.

(ii) If $\mathsf{prog}\text{`}\mathsf{L}(\mathsf{p}) = \mathsf{ASSIGN}(\mathsf{v}, \mathsf{e})$ then $(\mathsf{p}, \sigma) \rightsquigarrow (\mathsf{p}', \sigma[\mathsf{v} \mapsto [\![e]\!]\sigma])$ where $(\mathsf{p}, \mathsf{p}') \in \mathsf{prog}\text{`}\mathsf{cfg}\text{`}\tau$ and $\sigma[v \mapsto z]$ updates σ by mapping v to z and keeping rest of the mappings the same.

(iii) If $\mathsf{prog}\text{`}\mathsf{L}(\mathsf{p}) = \mathsf{ITE}(\mathsf{C}, \mathsf{p}_1, \mathsf{p}_2)$ then for $[\![\mathsf{C}]\!]\sigma = true$, $(\mathsf{p}, \sigma) \rightsquigarrow (\mathsf{p}_1, \sigma)$; otherwise $(\mathsf{p}, \sigma) \rightsquigarrow (\mathsf{p}_2, \sigma)$.

(iv) If $\mathsf{prog}\text{`}\mathsf{L}(\mathsf{p}) = \mathsf{HALT}$ then $(\mathsf{p}, \sigma) \rightsquigarrow (\odot, \sigma)$.

Definition A.2 (Program Trace.) Consider a program prog. A program trace ρ is a possibly infinite sequence of states $s_1 \rightsquigarrow \cdots \rightsquigarrow s_n \rightsquigarrow \cdots$ where $s_1 = (\mathsf{prog}\text{`}\mathsf{entry}, \sigma_1)$ is the initial state and σ_1 is the initial store.

Definition A.3 (Semantic Equivalence.) Consider two programs $\mathsf{prog1}$ and $\mathsf{prog2}$ whose state transition relations are $\rightsquigarrow$ and $\rightsquigarrow'$ respectively. They are semantically equivalent if for every finite trace $\rho = s_1 \rightsquigarrow \cdots \rightsquigarrow (\odot, \sigma_{|\rho|})$ of $\mathsf{prog1}$ there exists a finite trace $\rho' = s'_1 \rightsquigarrow' \cdots \rightsquigarrow' (\odot, \sigma'_{|\rho'|})$ of $\mathsf{prog2}$ such that the initial and final stores are the same: $\sigma_1 = \sigma'_1$ and $\sigma_{|\rho|} = \sigma'_{|\rho'|}$.

Theorem A.4 *If* $\mathsf{SoundRE}(\mathsf{prog}, \mathsf{points}, \mathsf{v})$ *defined in Fig. 4 holds then the programs* prog *and* $\mathsf{RE}(\mathsf{prog}, \mathsf{points}, \mathsf{v})$ *are semantically equivalent.*

Proof. Let $\mathsf{prog1} = \mathsf{RE}(\mathsf{prog}, \mathsf{points}, \mathsf{v})$ and $\rightsquigarrow'$ be its state transition relation. Consider a finite trace $\rho = s_1 \rightsquigarrow \cdots \rightsquigarrow (\odot, \sigma_{|\rho|})$ of prog. We show by induction that for every prefix ρ_i of length $i \leq |\rho|$ of ρ there exists a prefix ρ'_i of a finite trace $\rho' = s'_1 \rightsquigarrow' \cdots \rightsquigarrow' (\odot, \sigma'_{|\rho'|})$ of $\mathsf{prog1}$ such that $s_i = s'_i$.

Base Case. Consider the initial state $s_1 = (\mathsf{prog}\text{`}\mathsf{entry}, \sigma_1)$ of $\rho$. From the definition of RE in Fig. 4, $\mathsf{prog}\text{`}\mathsf{entry} = \mathsf{prog1}\text{`}\mathsf{entry}$. Since both programs have the same variables, we can define $s'_1 = (\mathsf{prog1}\text{`}\mathsf{entry}, \sigma_1)$.

Induction Hypothesis. Suppose for some k, $1 \leq k < |\rho|$, and ρ_k there exists ρ'_k such that $s_k = s'_k$ where $s_k = (\mathsf{p}_k, \sigma_k)$ and $s'_k = (\mathsf{p}'_k, \sigma'_k)$.

Induction Step. Since the language is deterministic, every state has exactly one successor state. Suppose $s_k \rightsquigarrow s_{k+1}$ and $s'_k \rightsquigarrow' s'_{k+1}$. We have to show that $s_{k+1} = (\mathsf{p}_{k+1}, \sigma_{k+1}) = s'_{k+1} = (\mathsf{p}'_{k+1}, \sigma'_{k+1})$. We have two cases:

(A) Suppose $\mathsf{p}_k \notin \mathsf{points}$. From the hypothesis, $(\mathsf{p}_k, \sigma_k) = (\mathsf{p}'_k, \sigma'_k)$. From the definition of RE in Fig. 4, $\mathsf{prog}\text{`}\mathsf{L}(\mathsf{p}_k) = \mathsf{prog1}\text{`}\mathsf{L}(\mathsf{p}'_k)$. Since RE does not change the flow of control, for all p_j, $(\mathsf{p}_k, \mathsf{p}_j) \in \mathsf{prog}\text{`}\mathsf{cfg}\text{`}\tau$ implies that $(\mathsf{p}'_k, \mathsf{p}_j) \in \mathsf{prog1}\text{`}\mathsf{cfg}\text{`}\tau$. Thus, $(\mathsf{p}_{k+1}, \sigma_{k+1}) = (\mathsf{p}'_{k+1}, \sigma'_{k+1})$.

(B) Suppose $\mathsf{p}_k \in \mathsf{points}$. p_k satisfies the conditions stated in SoundRE. Let $\mathsf{prog}\text{`}\mathsf{L}(\mathsf{p_k}) = \mathsf{ASSIGN}(\mathsf{x}, \mathsf{e})$. From the hypothesis, $(\mathsf{p}_k, \sigma_k) = (\mathsf{p}'_k, \sigma'_k)$. From the definition of RE in Fig. 4, $\mathsf{prog1}\text{`}\mathsf{L}(\mathsf{p}'_\mathsf{k}) = \mathsf{ASSIGN}(\mathsf{x}, \mathsf{BASE}(\mathsf{V}(\mathsf{v})))$. From the definition A.1, $\sigma_{k+1} = \sigma_k[\mathsf{x} \mapsto [\![\mathsf{e}]\!]\sigma_k]$ and $\sigma'_{k+1} = \sigma'_k[\mathsf{x} \mapsto [\![\mathsf{v}]\!]\sigma'_k]$. Therefore, we have to show that $[\![\mathsf{e}]\!]\sigma_k = [\![\mathsf{v}]\!]\sigma'_k$. Since $\sigma_k = \sigma'_k$, this reduces to $[\![\mathsf{e}]\!]\sigma_k = [\![\mathsf{v}]\!]\sigma_k$.

From the definition of SoundRE in Fig. 4,

$$\mathsf{AY}(\mathsf{prog}, \mathsf{AS}(\mathsf{prog}, \mathsf{TranspNDef}(\mathsf{prog}, \mathsf{e}, \mathsf{v}), \mathsf{AssignStmt}(\mathsf{prog}, \mathsf{v}, \mathsf{e})))(\mathsf{p_k}) \tag{A.1}$$

From the definition of statement semantics A.1, $(\mathsf{p}_1, \ldots, \mathsf{p}_k)$ is a maximal backward control flow path in prog. From (A.1),

$$\exists j : 1 \leq j < k : \mathsf{AssignStmt}(\mathsf{prog}, \mathsf{v}, \mathsf{e})(\mathsf{p}_j) \wedge (\forall l : j < l < k : \mathsf{TranspNDef}(\mathsf{prog}, \mathsf{e}, \mathsf{v})(\mathsf{p}_l))$$

Let us instantiate the existential quantifier by j and skolemize the universal quantifier by l. Following are the definitions of AssignStmt and TranspNDef:

$$\mathsf{AssignStmt}(\mathsf{prog}, \mathsf{v}, \mathsf{e})(\mathsf{p}_j) \stackrel{\triangle}{=} \mathsf{prog}\text{`}\mathsf{L}(\mathsf{p}_j) = \mathsf{ASSIGN}(\mathsf{v}, \mathsf{e})$$

$$\mathsf{TranspNDef}(\mathsf{prog}, \mathsf{e}, \mathsf{v})(\mathsf{p}_l) \stackrel{\triangle}{=} \mathsf{Assign?}(\mathsf{prog}\text{`}\mathsf{L}(\mathsf{p}_l)) \implies (\mathsf{Lhs}(\mathsf{prog}\text{`}\mathsf{L}(\mathsf{p}_l)) \notin \mathsf{VOperands}(\mathsf{e}) \wedge (\mathsf{Lhs}(\mathsf{prog}\text{`}\mathsf{L}(\mathsf{p}_l)) \neq \mathsf{v} \vee \mathsf{Rhs}(\mathsf{prog}\text{`}\mathsf{L}(\mathsf{p}_l)) = \mathsf{e}))$$

From the statement semantics A.1,

$$\sigma_{j+1} = \sigma_j[\mathsf{v} \mapsto [\![\mathsf{e}]\!]\sigma_j] \text{ which implies that } [\![\mathsf{v}]\!]\sigma_{j+1} = [\![\mathsf{e}]\!]\sigma_j \tag{A.2}$$

$$[\![\mathsf{e}]\!]\sigma_{l+1} = [\![\mathsf{e}]\!]\sigma_l \text{ and } ([\![\mathsf{v}]\!]\sigma_{l+1} = [\![\mathsf{v}]\!]\sigma_l \text{ or } [\![\mathsf{v}]\!]\sigma_{l+1} = [\![\mathsf{e}]\!]\sigma_l) \tag{A.3}$$

From the definition of SoundRE, $\mathsf{v} \notin \mathsf{VOperands}(\mathsf{e})$. Therefore, $[\![\mathsf{e}]\!]\sigma_{j+1} = [\![\mathsf{e}]\!]\sigma_j$. From (A.2), $[\![\mathsf{v}]\!]\sigma_{j+1} = [\![\mathsf{e}]\!]\sigma_{j+1}$. A program point p_l is preceded either by a program point p_j or another program point p_l. Therefore, $[\![\mathsf{v}]\!]\sigma_l = [\![\mathsf{e}]\!]\sigma_l$. From (A.3),

$[\![\mathsf{v}]\!]\sigma_{l+1} = [\![\mathsf{e}]\!]\sigma_{l+1}$. Since $1 \leq j < k$ and $j < l < k$, we have $[\![\mathsf{v}]\!]\sigma_k = [\![\mathsf{e}]\!]\sigma_k$. Therefore, $\sigma_{k+1} = \sigma'_{k+1}$.

Since RE does not change the flow of control, for all p_j, $(\mathsf{p}_k, \mathsf{p}_j) \in \mathsf{prog}\text{`}\mathsf{cfg}\text{`}\tau$ implies that $(\mathsf{p}'_k, \mathsf{p}_j) \in \mathsf{prog1}\text{`}\mathsf{cfg}\text{`}\tau$. Thus, $(\mathsf{p}_{k+1}, \sigma_{k+1}) = (\mathsf{p}'_{k+1}, \sigma'_{k+1})$. □