Proof Optimization for Partial Redundancy Elimination

Ando Saabas Tarmo Uustalu

Institute of Cybernetics, Tallinn University of Technology
Akadeemia tee 21, EE-12618 Tallinn, Estonia
{ando,tarmo}@cs.ioc.ee

Abstract

Partial redundancy elimination is a subtle optimization which performs common subexpression elimination and expression motion at the same time. In this paper, we use it as an example to promote and demonstrate the scalability of the technology of *proof optimization*. By this we mean automatic transformation of a given program's Hoare logic proof of functional correctness or resource usage into one of the optimized program, guided by a type-derivation representation of the result of the underlying dataflow analyses. A proof optimizer is a useful tool for the producer's side in a natural proof-carrying code scenario where programs are proved correct prior to optimizing compilation before transmission to the consumer.

We present a type-systematic description of the underlying analyses and of the optimization for the WHILE language, demonstrate that the optimization is semantically sound and improving in a formulation using type-indexed relations, and then show that these arguments can be transferred to mechanical transformations of functional correctness/resource usage proofs in Hoare logics. For the improvement part, we instrument the standard semantics and Hoare logic so that evaluations of expressions become a resource.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, optimization; F.3.1 [*Logics and Meanings of Programs*]: Specifying, Verifying and Reasoning about Programs—Logics of programs; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Operational semantics, Program analysis

General Terms Languages, Theory, Verification

Keywords partial redundancy elimination, soundness and improvement of dataflow analyses and optimizations, type systems, proof-carrying code, program proof transformation

1. Introduction

Proof-carrying code (PCC) is based on the idea that in a security-critical code transmission setting, the code producer should provide some evidence that the program she distributes is safe and/or functionally correct. The code consumer would thus receive the program together with a certificate (proof) that attests that the program has the desired properties.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PEPM'08, January 7–8, 2008, San Francisco, California, USA.
Copyright © 2008 ACM 978-1-59593-977-7/08/0001…$5.00

The code producer would typically use some (interactive) verification environment to prove her source program. The question is how to communicate the verification result to the code consumer who will not have access to the source code of the program. It is clear that there should be a mechanism to allow compilation of the program proof together with the program.

It has been shown [3] that proof compilation (automatic transformation of a program's proof alongside compiling the program) is simple in the case of a nonoptimizing compiler. However the same does not hold, if optimizations take place—a valid proof of a program may not be valid for the optimized version of the same program.

In this paper we tackle exactly this more challenging situation for dataflow-analysis based optimizations. We demonstrate a mechanism for *proof optimization*, a process where a program's proof is automatically transformed into one for the optimized program alongside the transformation of the program and based on the same dataflow analysis result. (Note that we do not mean that a proof of a fixed property of a fixed program is optimized somehow; rather we use the phrase 'proof optimization' for transforming a given program's proof to match the optimized program.) We describe dataflow analyses declaratively as type systems, so the result of a particular program's analysis is a type derivation. It turns out that we can use the same type derivation as self-sufficient guidance for automatically transforming not only the program, but also its proofs.

A simplified view of such a PCC scenario with program optimization happening on the producer's side is given in Figure 1. We are concerned with the stages shown in the gray box—simultaneous transformation of both a program and its proof guided by a type derivation representing the result of analyzing the program.

We also show that type systems are a compact and useful way of describing dataflow analyses and optimizations in general: they can explain them well in a declarative fashion (separating the issues of determining what counts as a valid analysis or optimization result and how to find one) and make soundness and improvement simple to prove by structural induction on type derivations. In fact, proof optimization works namely because of this: automatic proof transformations are a formal version of the constructive content of these semantic arguments.

As the example optimization we use partial redundancy elimination (PRE). PRE is a widely used compiler optimization that eliminates computations that are redundant on some but not necessarily all computation paths of a program. As a consequence, it performs both common subexpression elimination and expression motion. This optimization is notoriously tricky and has been extensively studied since it was invented by Morel and Renvoise [16]. There is no single canonical definition of the optimization. Instead, there is a plethora of subtly different ones and they do not all achieve exactly the same. The most modern and clear versions by Paleri et

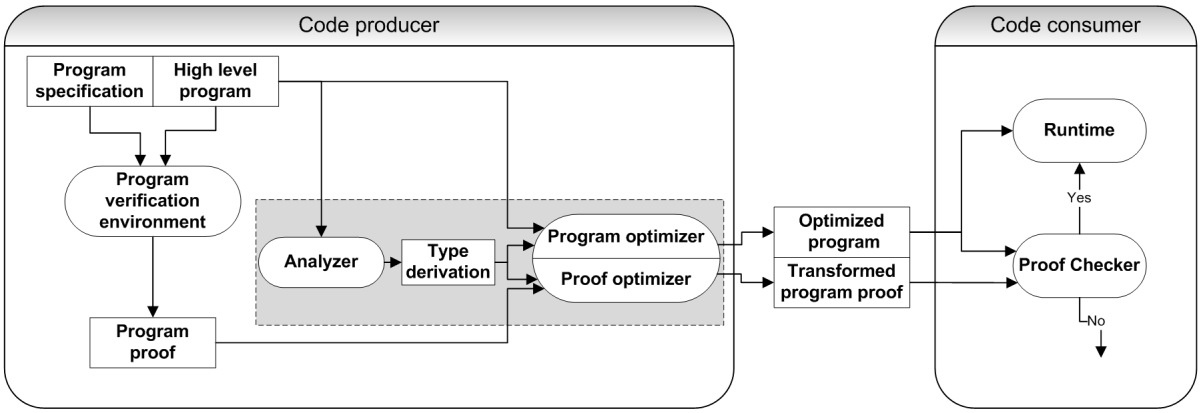


Figure 1. Proof optimization in PCC

al. [18] and Xue and Knoop [22] are based on four unidirectional dataflow analyses.

As a case study, the optimization is interesting in several aspects. As already said it is a highly nontrivial optimization. It is also interesting in the sense that it modifies program structure by inserting new nodes into the edges of the control flow graph. This makes automatic proof transformation potentially more difficult.

The present paper continues earlier work of ours [11, 19, 20] where we have advocated the use of type systems to describe dataflow analyses and optimizations in settings where it is important to be able to document and communicate analysis and optimization results in a checkable fashion. PRE is a worthwhile case study for one of the promising applications, automatic transformation of program proofs.

The organization of the paper is as follows. In Section 2, which is the central section, we consider a simple version of PRE that does not deal with all partial redundancies, but is very powerful already and rich enough for explaining all issues of our interest. After an informal explanation of the optimization we define it as a type system, argue that the optimization is semantically sound and improving in a similarity-relational sense and spell out the automatic proof transformations corresponding to these arguments. In Section 3, we sketch the same development for full PRE in the formulation of Paleri et al. [18] Section 4 reviews some items of most related work.

The language used Instead of using control-flow graph (CFG) based representations as is common in program optimization literature, we work directly with WHILE programs. This should make the paper more readable for the reader whose background is in program verification, but the techniques we present here are equally applicable to CFGs and we give some basic intuition also on CFGs. Following standard practice, we allow expressions to contain at most one operator. This is an inessential restriction: with the help of just a little more infrastructure we could also instead treat optimizations handling deep expressions directly.

The literals $l \in \mathbf{Lit}$, arithmetic expressions $a \in \mathbf{AExp}$, boolean expressions $b \in \mathbf{BExp}$ and statements $s \in \mathbf{Stm}$ are defined over a supply of program variables $x \in \mathbf{Var}$ and the numerals $n \in \mathbb{Z}$ in the following ways:

$$\begin{array}{lcl} l & ::= & x \mid n \\ a & ::= & l \mid l_0 + l_1 \mid l_0 * l_1 \mid \ldots \\ b & ::= & l_0 = l_1 \mid l_0 \leq l_1 \mid \ldots \\ s & ::= & x := a \mid \mathsf{skip} \mid s_0; s_1 \mid \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \mid \mathsf{while}\ b\ \mathsf{do}\ s_t. \end{array}$$

We write $\mathbf{AExp}^+$ for the set $\mathbf{AExp} \setminus \mathbf{Lit}$ of nontrivial arithmetic expressions. The states $\sigma \in \mathbf{State}$ of the natural semantics are stores, i.e., associations of integer values to variables, $\mathbf{State} =_{\mathrm{df}} \mathbf{Var} \to \mathbb{Z}$, and the definition of evaluation is standard. We write $[\![a]\!]\sigma$ (resp. $[\![b]\!]\sigma$) for the integer value of an arithmetic expression a (resp. truth value of a boolean expression b) in a state σ. The circumstance that σ' is a final state for a statement s and initial state σ is denoted by $\sigma \succ s \rightarrow \sigma'$. The definition of a sound and relatively completely Hoare logic for this language is standard as well. That Q is a derivable postcondition for s and a precondition P is written $\{P\}\, s\, \{Q\}$.

2. Simple PRE

What is PRE? As we have already stated it is an optimization to avoid computations of expressions that are redundant on some but not necessarily all computation paths of the program. Elimination of these computations happens at the expense of precomputing expressions and using auxiliary variables to remember their values.

An example application of partial redundancy elimination is shown in Figure 2 where the graph in Figure 2(a) is an original program and the graph in Figure 2(b) is the program after partial redundancy elimination optimization. Computations of $y + z$ in nodes 2 and 3 are partially redundant in the original program and can be eliminated. The result of computation of $y + z$ at node 5 can be saved into an auxiliary variable t (thus a new node is added in front of node 5). Additional computations of $y + z$ are inserted into the edge leading from node 1 to node 2 and the edge entering node 3. This is the optimal arrangement of computations, since there is one less evaluation of $y + z$ in the loop and on the path leading from node 3 to 2. Furthermore, the number of evaluations of the expression has not increased on any path through the program.

In this section we scrutinize a simplified version of PRE that is more conservative than full PRE. Although powerful already, it does not eliminate all partial redundancies, but is more easily presentable, relying on two dataflow analyses. The example program in Figure 2(a) can be fully optimized by simple PRE. The deficiencies of simple PRE as compared to full PRE will be discussed in Section 3.

The two dataflow analyses are backward anticipability analysis and a forward nonstandard, *conditional* partial availability analysis that uses the results of the anticipability analysis. The anticipability analysis computes for each program point which nontrivial arithmetic expressions will be evaluated on all paths before any of their operands are modified. The partial availability analysis computes which expressions have already been evaluated and later not

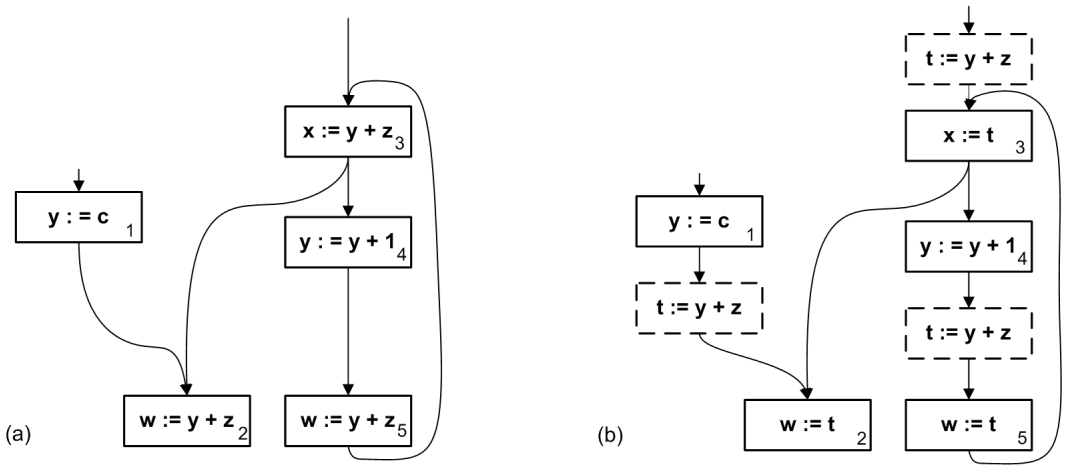


Figure 2. Example application of PRE

modified on some paths to a program point where that expression is anticipable. Note the dependency on the anticipability analysis. There are two possible optimizations for assignments. If we know that an expression is anticipable after an assignment, it means that the expression will definitely be evaluated later on in the program, so an auxiliary variable can be introduced, to carry the result of the evaluation. If we know that an expression is conditionally partially available at an assignment, we can assume it has already been computed and replace the expression with the auxiliary variable holding its value. If neither case holds, we leave the assignment unchanged. To make a conditionally partially available expression fully available, we must perform code motion, i.e., move evaluations of expressions to program points at which they are not partially available, but are partially available at the successor points.

The standard-style description of the algorithm relies on the properties $ANTIN$, $ANTOUT$, $CPAVIN$, $CPAVOUT$, MOD, $EVAL$. The global properties $ANTIN_i$ ($ANTOUT_i$) denote anticipability of nontrivial arithmetic expressions at the entry (exit) of node i. Similarly $CPAVIN_i$ and $CPAVOUT_i$ denote conditional partial availability. The local property MOD_i denotes the set of expressions whose value might be modified at node i whereas $EVAL_i$ denotes the set of nontrivial expressions which are evaluated at node i. The standard inequations for the analyses for CFGs are given below (s and f correspond to the start and finish nodes of the whole CFG).

$$
\begin{array}{rcl}
ANTOUT_i & \subseteq & \begin{cases} \emptyset & \text{if } i = f \\ \bigcap_{j \in succ(i)} ANTIN_j & \text{otherwise} \end{cases} \\
ANTIN_i & \subseteq & (ANTOUT_i \setminus MOD_i) \cup EVAL_i \\
CPAVIN_i & \supseteq & \begin{cases} \emptyset & \text{if } i = s \\ \bigcup_{j \in pred(i)} CPAVOUT_i & \text{otherwise} \end{cases} \\
CPAVOUT_i & \supseteq & ((CPAVIN_i \cup EVAL_i) \setminus MOD_i) \\
 & \cap & ANTOUT_i \\
CPAVIN_i & \subseteq & ANTIN_i \\
CPAVOUT_i & \subseteq & ANTOUT_i
\end{array}
$$

(The last inequalities do not correspond to transfer functions. Instead they state a domain restriction on conditional partial availability sets.)

Based on this analysis, one can now compute the local properties needed for code transformation. For example if $a \in CPAVIN_i$ holds for an expression a at node i, a use of expression a at that node can be replaced with a use of the corresponding auxiliary variable.

2.1 Type system for simple PRE

We now present the two analyses as type systems. Types and subtyping for anticipability correspond to the poset underlying its dataflow analysis, sets of nontrivial arithmetic expressions, with set inclusion, i.e, $(\mathcal{P}(\mathbf{AExp}^+), \subseteq)$. A program point has type $ant \in \mathcal{P}(\mathbf{AExp}^+)$ if all the expressions in ant are anticipable, i.e., on all paths from that program point, there will be a use of the expression before any of its operands are modified. Subtyping is set inclusion, i.e., $ant' \leq ant$ iff $ant' \subseteq ant$. Typing judgements $s : ant' \longrightarrow ant$ associate a statement with a pre- and posttype pair, stating that, if at the end of a statement s the expressions in ant are anticipable, then at the beginning the expressions in ant' must be anticipable. The typing rules are given in figure 3. We use $eval(a)$ to denote the set $\{a\}$, if a is a nontrivial expression, and $\emptyset$ otherwise, and $mod(x)$ to denote the set of nontrivial expressions containing x, i.e., $mod(x) =_{\mathrm{df}} \{a \mid x \in FV(a)\}$. The assignment rule states that the assignment to x kills all expressions containing x and at the same time the expression assigned becomes anticipable, if nontrivial. To type an if-statement the pre- and posttypes for both branches have to match. For a while loop, some type must be invariant for the loop body. The subsumption rule (analogous to the consequence rule in the standard Hoare logic) is unsurprising. We note that the type system accepts all valid anticipability analysis results (all solutions to the inequations), not only the strongest one. Finding the strongest one corresponds to principal type inference (finding the greatest pretype for a given posttype). This separation of the algorithmic from the declarative is typical to the type-systematic description of dataflow analyses and should be appreciated as a good thing.

The anticipability analysis gives us the information about where it is definitely profitable to precompute expressions. Intuitively, they should be precomputed where they first become available and are anticipable, and reused where they are already available. The second analysis, the conditional partial availability analysis, propagates this information forward in the control flow graph. As it depends on the anticipability analysis, we need to combine the two in the type system. For the combined type system, a type is a pair $(ant, cpav) \in \mathcal{P}(\mathbf{AExp}^+) \times \mathcal{P}(\mathbf{AExp}^+)$ satisfying the constraint $cpav \subseteq ant$, where ant is an anticipability type and $cpav$ is a conditional partial availability type. Subtyping $\leq$

is pointwise set inclusion, i.e. $(ant', cpav) \leq (ant, cpav')$ iff $ant' \subseteq ant$ and $cpav \subseteq cpav'$. Typing judgements take the form $s : ant', cpav \longrightarrow ant, cpav'$. The intended meaning of the added conditional partial availability component here is that, if the expressions in $cpav$ are conditionally partially available before running s, then expressions in $cpav'$ may be conditionally partially available after running the program.

The typing rules of the combined type system are given in Figure 4. The rules for assignment now have the conditional partial availability component. An expression is conditionally partially available in the posttype of an assignment if it is so already in the pretype or is evaluated by the assignment and the assignment does not modify any of its operands. Additionally, the expression is only declared conditonally partially available if it is actually anticipable (worth to be precomputed), thus the intersection with the anticipability type.

The optimization component of the type system is shown in Figure 5. Definitions of auxiliary variables can be introduced in two places, before assignments (if the necessary conditions are met) and at subsumptions. An already computed value is used if an expression is conditionally partially available (rule $:=_{3_{\mathrm{pre}}}$). If it is not, but is anticipable (will definitely be used), and the assignment does not change the value of the expression, then the result of evaluating it is recorded in the auxiliary variable for that expression (rule $:=_{2_{\mathrm{pre}}}$). Code motion is performed by the subsumption rule, which introduces auxiliary variable definitions when there is shrinking or growing of types (this typically happens at the beginning of loops and at the end of conditional branches and loop bodies). The auxiliary function nv delivers a unique new auxiliary variable for every nontrivial arithmetic expression.

2.2 Semantic soundness and improvement

Soundness in the sense of preservation of semantics (to an appropriate precision) of the type system for simple PRE can be stated and shown using the relational method [4]. We take soundness to mean that an original program and its optimized version simulate each other up to a similarity relation $\sim$ on states, indexed by conditional partial availability types of program points.

Let $\sigma \sim_{cpav} \sigma'$ denote that two states σ and σ' agree on the auxiliary variables wrt. $cpav \subseteq \mathbf{AExp}^+$ in the sense that $\forall x \in \mathbf{Var}.\, \sigma(x) = \sigma'(x)$ and $\forall a \in cpav.\, [\![a]\!]\sigma = \sigma'(nv(a))$. We can then obtain the following soundness theorem.

THEOREM 1 (Soundness of simple PRE).
If $s : ant', cpav \longrightarrow ant, cpav' \hookrightarrow s_*$ and $\sigma \sim_{cpav} \sigma_*$, then
— $\sigma \succ s \rightarrow \sigma'$ implies the existence of σ'_* such that $\sigma' \sim_{cpav'} \sigma'_*$ and $\sigma_* \succ s_* \rightarrow \sigma'_*$,
— $\sigma_* \succ s_* \rightarrow \sigma'_*$ implies the existence of σ' such that $\sigma' \sim_{cpav'} \sigma'_*$ and $\sigma \succ s \rightarrow \sigma'$.

The proof is by induction on the typing derivation with a subsidiary induction on the derivation of the semantic judgement.

It is possible to show more than just preservation of semantics using the relational method. One can also show that the optimization is actually an improvement in the sense that the number of evaluations of an expression on any given program path cannot increase. This means that no new computations can be introduced which are not used later on in the program. This is not obvious, since code motion might introduce unneeded evaluations.

To show this property, there must be a way to count the expression uses. This can be done via a simple instrumented semantics, which counts the number of evaluations of every expression. In the instrumented semantics a state is a pair (σ, r) of a standard state $\sigma \in \mathbf{Var} \rightarrow \mathbb{Z}$ (an assignment of integer values to variables) and a "resource" state $r \in \mathbf{AExp}^+ \rightarrow \mathbb{N}$ associating to every nontrivial arithmetic expression a natural number for the number of times it has been evaluated. The rules of the semantics are as those for the standard semantics, except that for assignments of nontrivial expressions we stipulate

$$\frac{}{(\sigma, r) \succ x := a \rightarrow (\sigma[x \mapsto [\![a]\!]\sigma], r[a \mapsto r(a) + 1])}$$

The corresponding similarity relation between the states is the following. We define $(\sigma, r) \approx_{cpav} (\sigma', r')$ to mean that two states (σ, r) and (σ', r') are similar wrt. $cpav \subseteq \mathbf{AExp}^+$ in the sense that $\sigma \sim_{cpav} \sigma'$ and, moreover, $\forall a \in cpav.\, r'(a) \leq r(a) + 1$ and $\forall a \notin cpav.\, r'(a) \leq r(a)$.

The cute point here is that conditional partial availability types serve us as an "amortization" mechanism. The intuitive meaning of an expression being in the type of a program point is that there will be a use of this expression somewhere in the future, where this expression will be replaced with a variable already holding its value. Thus it is possible that a computation path of an optimized program has one more evaluation of the expression before this point than the corresponding computation path of the original program due to an application of subsumption. This does not break the improvement argument, since the type increase at the subsumption point contains a promise that this evaluation will be taken advantage of ("amortized") in the future.

THEOREM 2 (Improvement property of simple PRE).
If $s : ant', cpav \longrightarrow ant, cpav' \hookrightarrow s_*$ and $(\sigma, r) \approx_{cpav} (\sigma_*, r_*)$, then
— $(\sigma, r) \succ s \rightarrow (\sigma', r')$ implies the existence of (σ'_*, r'_*) such that $(\sigma', r') \approx_{cpav'} (\sigma'_*, r'_*)$ and $(\sigma_*, r_*) \succ s_* \rightarrow (\sigma'_*, r'_*)$,
— $(\sigma_*, r_*) \succ s_* \rightarrow (\sigma'_*, r'_*)$ implies the existence of (σ', r') such that $(\sigma', r') \approx_{cpav'} (\sigma'_*, r'_*)$ and $(\sigma, r) \succ s \rightarrow (\sigma', r')$.

Again, the proof is by induction on the structure of the type derivation.

To prove that an optimization is really optimal in the sense of achieving the best possible improvement (which simple PRE really is not), we would have to fix what kind of modifications of a given program we consider as possible transformation candidates (they should not modify the control flow graph other than by splitting edges, they should not take advantage of the real domains and interpretation of expressions etc.). The argument would have to compare the optimization to other sound transformation candidates. This is outside the scope of the present paper.

2.3 Automatic transformation of Hoare logic proofs

Simple PRE can change the structure of a program, so a given Hoare proof for the program may be incompatible with the optimized program already solely by its structure. Moreover, even the Hoare triple proved for the original program may not be provable for the optimized program.

For example, given a proof of the Hoare triple $\{y + z = 5\}\, x := y + z\, \{x = 5\}$ and a derivation of the typing judgement $x := y + z : \{y + z\}, \{y + z\} \longrightarrow \emptyset, \emptyset \hookrightarrow x := nv(y + z)$, it is clear that $\{y + z = 5\}\, x := nv(y + z)\, \{x = 5\}$ is not a provable Hoare triple anymore. But the original Hoare proof, including the triple it proves, can be transformed, guided by the type derivation, which carries all the information on how and where code is transformed. The key observation is that the expressions which are conditionally partially available must have been computed and their values not modified, thus their values are equal to the values of the corresponding auxiliary variables that have been defined.

Let $P|_{cpav}$ abbreviate $\bigwedge[\, nv(a) = a \mid a \in cpav\,] \wedge P$.

We have the following theorem.

$$\frac{}{x := a : ant \setminus mod(x) \cup eval(a) \longrightarrow ant} \qquad \frac{}{\mathsf{skip} : ant \longrightarrow ant} \qquad \frac{s_0 : ant' \longrightarrow ant'' \quad s_1 : ant'' \longrightarrow ant}{s_0; s_1 : ant' \longrightarrow ant}$$

$$\frac{s_t : ant' \longrightarrow ant \quad s_f : ant' \longrightarrow ant}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : ant' \longrightarrow ant} \qquad \frac{s_t : ant \longrightarrow ant}{\mathsf{while}\ b\ \mathsf{do}\ s_t : ant \longrightarrow ant} \qquad \frac{ant \leq ant_0 \quad s : ant_0 \longrightarrow ant_0' \quad ant_0' \leq ant'}{s : ant \longrightarrow ant'}$$

Figure 3. Type system for anticipability

$$\frac{}{x := a : ant \setminus mod(x) \cup eval(a), cpav \longrightarrow ant, (cpav \cup eval(a) \setminus mod(x)) \cap ant}$$

$$\frac{}{\mathsf{skip} : ant, cpav \longrightarrow ant, cpav} \qquad \frac{s_0 : ant, cpav \longrightarrow ant'', cpav'' \quad s_1 : ant'', cpav'' \longrightarrow ant', cpav'}{s_0; s_1 : ant, cpav \longrightarrow ant', cpav'}$$

$$\frac{s_t : ant', cpav \longrightarrow ant, cpav' \quad s_f : ant', cpav \longrightarrow ant, cpav'}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : ant', cpav \longrightarrow ant, cpav'} \qquad \frac{s_t : ant, cpav \longrightarrow ant, cpav}{\mathsf{while}\ b\ \mathsf{do}\ s_t : ant, cpav \longrightarrow ant, cpav}$$

$$\frac{ant, cpav \leq ant_0, cpav_0 \quad s : ant_0, cpav_0 \longrightarrow ant_0', cpav_0' \quad ant_0', cpav_0' \leq ant', cpav'}{s : ant, cpav \longrightarrow ant', cpav'}$$

Figure 4. Type system for the underlying analyses of simple PRE

$$\frac{a \notin cpav \quad a \notin ant \vee x \in FV(a)}{x := a : ant \setminus mod(x) \cup eval(a), cpav \longrightarrow ant, (cpav \cup eval(a) \setminus mod(x)) \cap ant \hookrightarrow x := a}\ :=_{1\,\mathrm{pre}}$$

$$\frac{a \notin cpav \quad a \in ant \quad x \notin FV(a)}{x := a : ant \setminus mod(x) \cup eval(a), cpav \longrightarrow ant, (cpav \cup eval(a) \setminus mod(x)) \cap ant \hookrightarrow nv(a) := a; x := nv(a)}\ :=_{2\,\mathrm{pre}}$$

$$\frac{a \in cpav}{x := a : ant \setminus mod(x) \cup eval(a), cpav \longrightarrow ant, (cpav \cup eval(a) \setminus mod(x)) \cap ant \hookrightarrow x := nv(a)}\ :=_{3\,\mathrm{pre}}$$

$$\frac{}{\mathsf{skip} : ant, cpav \longrightarrow ant, cpav \hookrightarrow \mathsf{skip}}\ \mathrm{skip}_{\mathrm{pre}} \qquad \frac{s_0 : ant, cpav \longrightarrow ant'', cpav'' \hookrightarrow s_0' \quad s_1 : ant'', cpav'' \longrightarrow ant', cpav' \hookrightarrow s_1'}{s_0; s_1 : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s_0'; s_1'}\ \mathrm{comp}_{\mathrm{pre}}$$

$$\frac{s_t : ant', cpav \longrightarrow ant, cpav' \hookrightarrow s_t' \quad s_f : ant', cpav \longrightarrow ant, cpav' \hookrightarrow s_f'}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : ant', cpav \longrightarrow ant, cpav' \hookrightarrow \mathsf{if}\ b\ \mathsf{then}\ s_t'\ \mathsf{else}\ s_f'}\ \mathrm{if}_{\mathrm{pre}}$$

$$\frac{s_t : ant, cpav \longrightarrow ant, cpav \hookrightarrow s_t'}{\mathsf{while}\ b\ \mathsf{do}\ s_t : ant, cpav \longrightarrow ant, cpav \hookrightarrow \mathsf{while}\ b\ \mathsf{do}\ s_t'}\ \mathrm{while}_{\mathrm{pre}}$$

$$\frac{ant, cpav \leq ant_0, cpav_0 \quad s : ant_0, cpav_0 \longrightarrow ant_0', cpav_0' \hookrightarrow s' \quad ant_0', cpav_0' \leq ant', cpav'}{s : ant, cpav \longrightarrow ant', cpav' \hookrightarrow [\, nv(a) := a \mid a \in cpav_0 \setminus cpav \,]; s'; [\, nv(a) := a \mid a \in cpav' \setminus cpav_0' \,]}\ \mathrm{conseq}_{\mathrm{pre}}$$

Figure 5. Type system for simple PRE, with the optimization component

THEOREM 3 (Preservation of Hoare logic provability/proofs). *If* $s : ant', cpav \longrightarrow ant, cpav' \hookrightarrow s_*$, *then* —$\{P\}\ s\ \{Q\}$ *implies* $\{P|_{cpav}\}\ s_*\ \{Q|_{cpav'}\}$.

Nonconstructively, this theorem is a corollary from the soundness of the type system (second half) and soundness and relative completeness of Hoare logic. The constructive proof is by induction on the structure of type derivation (and inspection of the aligned Hoare triple proof). The constructive proof gives automatic Hoare proof transformation, i.e., an algorithm which takes a proof of $\{P\}\ s\ \{Q\}$ and returns the proof of $\{P|_{cpav}\}\ s'\ \{Q|_{cpav'}\}$.

We can look at some interesting cases where actual modifications happen (the cases for sequence, if, and while constructs are trivial).

- Case $:=_{\mathrm{pre}}$: The type derivation is

$$\frac{}{x := a : ant', cpav \longrightarrow ant, cpav' \hookrightarrow s_*}$$

where $ant' =_{\mathrm{df}} ant \setminus mod(x) \cup eval(a)$, $cpav' =_{\mathrm{df}} (cpav \cup eval(a) \setminus mod(x)) \cap ant$. The given Hoare logic proof is

$$\frac{}{\{P[a/x]\}\ x := a\ \{P\}}$$

We notice that $cpav'$ can include no expressions containing x, hence $P[a/x]|_{cpav'} \Leftrightarrow P|_{cpav'}[a/x]$ and $P[nv(a)/x]|_{cpav'} \Leftrightarrow P|_{cpav'}[nv(a)/x]$.

  - Subcase $:=_{1\,\mathrm{pre}}$: We have that $a \notin cpav$. We also have that either $a \notin ant$ or $x \in FV(a)$ (i.e., $a \notin mod(x)$). Moreover, $s_* =_{\mathrm{df}} x := a$.

From the assumptions it follows that $cpav' \subseteq cpav$ and hence $P[a/x]|_{cpav} \models P[a/x]|_{cpav'}$. The transformed Hoare logic proof is

$$\dfrac{\dfrac{}{\{P|_{cpav'}[a/x]\}\, x := a\, \{P|_{cpav'}\}}}{\{P[a/x]|_{cpav'}\}\, x := a\, \{P|_{cpav'}\}}}{\{P[a/x]|_{cpav}\}\, x := a\, \{P|_{cpav'}\}}$$

- Subcase $:=_{2\mathrm{pre}}$: We have that $a \notin cpav$. We also have that $a \in ant$ and $x \notin FV(a)$. And $s_* =_{\mathrm{df}} nv(a) := a; x := nv(a)$.

  From the assumptions it follows that $cpav' \subseteq cpav \cup \{a\}$, so $P[nv(a)/x]|_{cpav\cup\{a\}} \models P[nv(a)/x]|_{cpav'}$. From reflexivity of equality, $P[a/x]|_{cpav} \Leftrightarrow P[a/x]|_{cpav} \wedge a = a \Leftrightarrow (P[nv(a)/x]|_{cpav\cup\{a\}})[a/nv(a)]$. The transformed Hoare logic proof is

$$\dfrac{B_0 \quad B_1}{\{P[a/x]|_{cpav}\}\, nv(a) = a; x := nv(a)\, \{P|_{cpav'}\}}$$

  where $B_0 \equiv$

$$\dfrac{\{P[nv(a)/x]|_{cpav\cup\{a\}}[a/nv(a)]\}\, nv(a) = a\, \{P[nv(a)/x]|_{cpav\cup\{a\}}\}}{\{P[a/x]|_{cpav}\}\, nv(a) = a\, \{P[nv(a)/x]|_{cpav'}\}}$$

  and $B_1 \equiv$

$$\dfrac{\dfrac{}{\{P|_{cpav'}[nv(a)/x]\}\, x := nv(a)\, \{P|_{cpav'}\}}}{\{P[nv(a)/x]|_{cpav'}\}\, x := nv(a)\, \{P|_{cpav'}\}}$$

- Subcase $:=_{3\mathrm{pre}}$: We have that $a \in cpav$, $s_* =_{\mathrm{df}} x := nv(a)$.

  It is given that $a \in cpav$ and it follows that $cpav' \subseteq cpav$. Hence $P[a/x]|_{cpav} \models P[a/x]|_{cpav'} \wedge nv(a) = a$. Substitution of equals for equals gives $P|_{cpav'}[a/x] \wedge nv(a) = a \models P|_{cpav'}[nv(a)/x]$. The transformed Hoare logic proof is

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{P|_{cpav'}[nv(a)/x]\}\, x := nv(a)\, \{P|_{cpav'}\}}}{\{P|_{cpav'}[a/x] \wedge nv(a) = a\}\, x := nv(a)\, \{P|_{cpav'}\}}}{\{P[a/x]|_{cpav'} \wedge nv(a) = a\}\, x := nv(a)\, \{P|_{cpav'}\}}}{\{P[a/x]|_{cpav}\}\, x := nv(a)\, \{P|_{cpav'}\}}$$

- Case $\mathrm{conseq}_{\mathrm{pre}}$: The type derivation is

$$\dfrac{\begin{array}{c}\vdots\\ s : ant_0, cpav_0 \longrightarrow ant_0', cpav_0' \hookrightarrow s_*\end{array}}{s : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s'; s_*; s''}$$

  where $(ant, cpav) \leq (ant_0, cpav_0), (ant_0', cpav_0') \leq (ant, cpav)$ and $s' =_{\mathrm{df}} [nv(a) := a \mid a \in cpav_0 \setminus cpav]$, $s'' =_{\mathrm{df}} [nv(a) := a \mid a \in cpav' \setminus cpav_0']$. The given Hoare logic proof is

$$\dfrac{\begin{array}{c}\vdots\\ \{P_0\}\, s\, \{Q_0\}\end{array}}{\{P\}\, s\, \{Q\}}$$

  where $P \models P_0$ and $Q_0 \models Q$.

  By the induction hypothesis, there is a Hoare logic proof of $\{P_0|_{cpav_0}\}\, s_*\, \{Q_0|_{cpav_0'}\}$. It is an assumption that $P \models P_0$, hence $P|_{cpav} \models P_0|_{cpav}$.

  The assumptions also say $cpav \subseteq cpav_0$, so using reflexivity of equality we get $P_0|_{cpav} \Leftrightarrow P_0|_{cpav} \wedge \bigwedge[\, a = a \mid \mid a \in cpav_0 \setminus cpav\,] \Leftrightarrow P_0|_{cpav_0}[\, a/nv(a) \mid a \in cpav_0 \setminus cpav\,]$.

  Hence from the axiom $\{P_0|_{cpav_0}[\, a/nv(a) \mid a \in cpav_0 \setminus cpav\,]\}\, s'\, \{P_0|_{cpav_0}\}$ by the consequence rule we have a proof of $\{P|_{cpav}\}\, s'\, \{P_0|_{cpav_0}\}$.

  Similarly we can make a proof of $\{Q_0|_{cpav_0'}\}\, s''\, \{Q|_{cpav'}\}$.

  Putting everything together with the sequence rule, we obtain a proof of $\{P|_{cpav}\}\, s'; s_*; s''\, \{Q|_{cpav'}\}$, which is the required transformed Hoare logic proof.

An example application of the type system and transformation of Hoare logic proofs is shown in Figures 6, 7 and 8. We have a program $s =_{\mathrm{df}} \mathsf{while}\ i < k\ \mathsf{do}\ (n := n+(y+z); i := i+1); x := y + z$ and a Hoare derivation tree for $\{n = 0 \wedge i = 0 \wedge k \geq 0\}\, s\, \{n = k * (y + z)\}$. (Note that to make the derivation trees smaller, we use $n := n + (y + z)$, i.e. an expression with more than one operator. This can be considered as syntactic sugar, since the assignment could be rewritten as $n' := y + z; n := n + n'$.) The optimization lifts the computation of $y + z$ out of the while-loop. This renders the original proof of the program impossible to associate to the transformed program. For example, the old loop invariant is not valid any more, since it talks about $y + z$, but the expression is not present in the modified loop. Figure 8 shows the proof tree where this has been remedied using the information present in the types.

We can also achieve an automatic proof transformation corresponding to the improvement property. This allows us to invoke a performance bound of a given program to obtain one for its optimized version.

Similarly to semantic improvement, where we needed an instrumented semantics, now we need an instrumented Hoare logic. We extend the signature of the standard Hoare logic with an extralogical constant $\ulcorner a \urcorner$ for any expression $a \in \mathbf{AExp}^+$. The inference rules of the instrumented Hoare logic are the analogous to those for the standard Hoare logic except that the axiom for nontrivial assignment becomes

$$\dfrac{}{\{P[a/x][\ulcorner a \urcorner + 1/\ulcorner a \urcorner]\}\, x := a\, \{P\}}$$

It should not come as a surprise that the instrumented Hoare logic is sound and relatively complete wrt. the instrumented semantics.

Now, we define $P\|_{cpav}$ to abbreviate

$$\begin{array}{ll}[\, \exists v(a) \mid a \in \mathbf{AExp}^+\,]. & \\ \qquad \bigwedge[\, nv(a) = a \wedge \ulcorner a \urcorner \leq v(a) + 1 \mid a \in cpav\,] & \\ \wedge \quad \bigwedge[\, \ulcorner a \urcorner \leq v(a) \mid a \notin cpav\,] & \\ \wedge \quad P[v(a)/\ulcorner a \urcorner] & \end{array}$$

Here $v(a)$ generates a new unique logic variable for every nontrivial arithmetic expression. With this notation we can state a refined theorem, yielding transformation of proofs of the instrumented Hoare logic.

THEOREM 4. (Preservation of instrumented Hoare logic provability/proofs)
If $s : ant', cpav \longrightarrow ant, cpav' \hookrightarrow s_*$, *then*
— $\{P\}\, s\, \{Q\}$ *implies* $\{P\|_{cpav}\}\, s_*\, \{Q\|_{cpav'}\}$.

The proofs (nonconstructive and constructive) are similar to those of the previous theorem.

To witness the theorem in action we revisit the program analyzed in Figure 6. Figure 9 demonstrates that in the instrumented Hoare logic we can prove that the program computes $y + z$ exactly $k + 1$ times (we have abbreviated $\ulcorner y + z \urcorner$ to c). The invariant for the while-loop is $i \leq k \wedge c = i$. Figure 10 contains the transformed proof for the optimized program. We can prove that $y + z$ is computed at most $k + 1$ times, but the proof is quite different;

$$\dfrac{\dfrac{}{x := y + z : \{y+z\}, \{y+z\} \longrightarrow \emptyset, \emptyset \hookrightarrow x := t} \quad \dfrac{\dfrac{}{n := n + (y+z) : \{y+z\}, \{y+z\} \longrightarrow \{y+z\}, \{y+z\} \hookrightarrow n := n + t} \quad \dfrac{}{i := i + 1 : \{y+z\}, \{y+z\} \longrightarrow \{y+z\}, \{y+z\} \hookrightarrow i := i + 1}}{n := n + (y+z); i := i + 1 : \{y+z\}, \{y+z\} \longrightarrow \{y+z\}, \{y+z\} \hookrightarrow n := n + t; i := i + 1}}{\dfrac{\text{while } i < k \text{ do } (n := n + (y+z); i := i + 1) : \{y+z\}, \{y+z\} \longrightarrow \{y+z\}, \{y+z\} \hookrightarrow \text{while } i < k \text{ do } (n := n + t; i := i + 1)}{\dfrac{\text{while } i < k \text{ do } (n := n + (y+z); i := i + 1) : \{y+z\}, \emptyset \longrightarrow \{y+z\}, \{y+z\} \hookrightarrow t := y + z; \text{while } i < k \text{ do } (n := n + t; i := i + 1)}{\text{while } i < k \text{ do } (n := n + (y+z); i := i + 1); x := y + z : \{y+z\}, \emptyset \longrightarrow \emptyset, \emptyset \hookrightarrow t := y + z; \text{while } i < k \text{ do } (n := n + t; i := i + 1); x := t}}}$$

Figure 6. Type derivation for an example program

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{n = k*(y+z)\}\, x := y + z\, \{n = k*(y+z)\}}}{\{i+1 \le k \wedge n = (i+1)*(y+z)\}\, i := i + 1\, \{i \le k \wedge n = i*(y+z)\}}}{\{i+1 \le k \wedge n + (y+z) = (i+1)*(y+z)\}\, n := n + (y+z)\, \{i+1 \le k \wedge n = (i+1)*(y+z)\}}}{\dfrac{\{i < k \wedge n = i*(y+z)\}\, n := n + (y+z); i := i + 1\, \{i \le k \wedge n = i*(y+z)\}}{\{i \le k \wedge n = i*(y+z)\}\, \text{while } i < k \text{ do } (n := n + (y+z); i := i + 1)\, \{i \not< k \wedge i \le k \wedge n = i*(y+z)\}}}{\{n = 0 \wedge i = 0 \wedge k \ge 0\}\, \text{while } i < k \text{ do } (n := n + (y+z); i := i + 1); x := y + z\, \{n = k*(y+z)\}}$$

Figure 7. An original proof for the example program

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{n = k*(y+z)\}\, x := t\, \{n = k*(y+z)\}}}{\{\ n = k*(y+z) \ \wedge\ t = y+z\ \}\, x := t\, \{n = k*(y+z)\}}}{\{\ i+1 \le k \wedge n = (i+1)*(y+z) \ \wedge\ t = y+z\ \}\, i := i + 1\, \{\ i \le k \wedge n = i*(y+z) \ \wedge\ t = y+z\ \}}}{\dfrac{\{i \le k \wedge n = i*(y+z)\}\, t := y + z\, \{\ i \le k \wedge n = i*(y+z) \ \wedge\ t = y+z\ \}}{\dfrac{\{\ i+1 \le k \wedge n + t = (i+1)*(y+z) \ \wedge\ t = y+z\ \}\, n := n + t\, \{\ i+1 \le k \wedge n = (i+1)*(y+z) \ \wedge\ t = y+z\ \}}{\dfrac{\{\ i < k \wedge n = i*(y+z) \ \wedge\ t = y+z\ \}\, n := n + t; i := i + 1\, \{\ i \le k \wedge n = i*(y+z) \ \wedge\ t = y+z\}}{\{\ i \le k \wedge n = i*(y+z) \ \wedge\ t = y+z\ \}\, \text{while } i < k \text{ do } (n := n + (y+z); i := i + 1)\, \{\ i \not< k \wedge i \le k \wedge n = i*(y+z) \ \wedge\ t = y+z\ \}}}}{\{i \le k \wedge n = i*(y+z)\}\, t := y + z; \text{while } i < k \text{ do } (n := n + (y+z); i := i + 1)\, \{\ i = k \wedge n = i*(y+z) \ \wedge\ t = y+z\ \}}}{\{n = 0 \wedge i = 0 \wedge k \ge 0\}\, t := y + z; \text{while } i < k \text{ do } (n := n + t; i := i + 1); x := t\, \{n = k*(y+z)\}}$$

Figure 8. The transformed proof

$$\dfrac{\dfrac{\dfrac{}{\{c+1 = k+1\}\, x := y + z\, \{c = k+1\}}}{\{i+1 \le k \wedge c+1 = i+1\}\, n := n + (y+z)\, \{i+1 \le k \wedge c = (i+1)\}} \quad \dfrac{}{\{i+1 \le k \wedge c = (i+1)\}\, i := i + 1\, \{i \le k \wedge c = i\}}}{\dfrac{\{i < k \wedge c = i\}\, n := n + (y+z); i := i + 1\, \{i \le k \wedge c = i)\}}{\dfrac{\{i \le k \wedge c = i\}\, \text{while } i < k \text{ do } (n := n + (y+z); i := i + 1)\, \{i \not< k \wedge i \le k \wedge c = i)\}}{\{c = 0 \wedge i = 0 \wedge k \ge 0\}\, \text{while } i < k \text{ do } (n := n + (y+z); i := i + 1); x := y + z\, \{c = k+1)\}}}$$

Figure 9. An original proof for resource usage

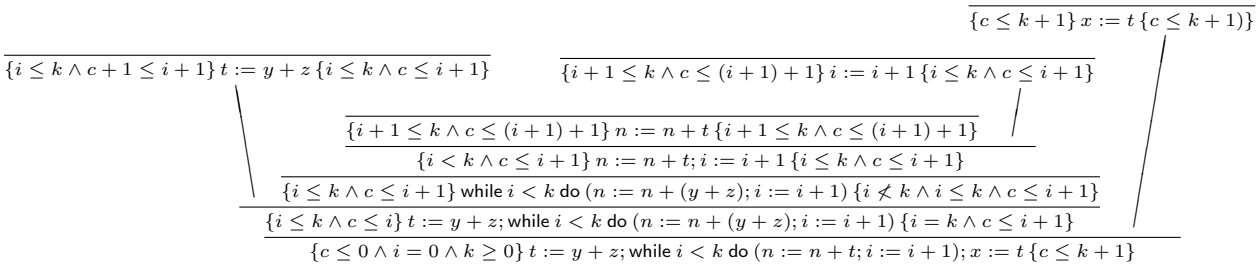


Figure 10. The transformed proof for resource usage

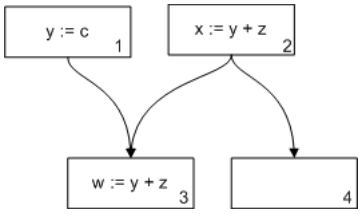
in particular, the loop invariant is now $i \le k \wedge c = i+1$. (In this proof, we have enhanced readability by replacing the existentially quantified assertions yielded by the automatic transformation with equivalent quantifier-free simplifications.)

As expected, this formal counterpart of the semantic improvement argument is no smarter than the semantic improvement argument. In our semantic improvement statement we claimed that the optimized program performs not worse than possibly by one extra evaluation for precomputed expressions than the original one. Had we claimed something more specific and stronger about, e.g., those loops from where at least one assignment can be moved out, our corresponding automatic proof transformation could have been stronger as well. It is not our goal to delve deeper into this interesting point here. Rather, we are content here with the observation that constructive and structured semantic arguments have can be given formal counterparts in the form of automatic proof transformations.

We finish this section by remarking that the first halves of the semantic soundness and improvement theorems yield a transformation of a proof of an optimized program into one of the original program, which can also be made constructive. We do not discuss this here; the idea has been demonstrated elsewhere [19].

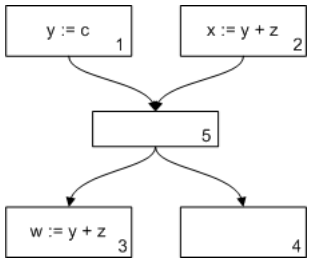
3. Full PRE

We now look at the formulation of full PRE by Paleri et al. [18]. As was explained in Section 2, simple PRE does not use all optimization opportunities. This stems from the fact that it only takes into account total anticipability. An example of a program which simple PRE does not optimize is the following one.



The program is left unoptimized by simple PRE since $y+z$ is not anticipable at the exit of node 2. Full PRE would optimize the program by introducing a new auxiliary variable to hold the computation of $y+z$ in node 2 and copying the computation of $y+z$ into the edge leaving node 1. This would allow to skip the computation of $y+z$ in node 3.

This does not mean that it is possible to simply substitute the total anticipability analysis with partial anticipability. The following example illustrates this.



While it is seemingly similar to the previous example, it cannot be optimized the same way, since if we moved a computation of $y+z$ into the edge (1,5), we would potentially worsen the runtime behavior of the program, since going through the program through nodes (1, 5, 4), there would be an extra evaluation of $y+z$ that was not present in the original program. In fact no further optimization of this program is possible.

The fundamental observation which allows us to perform PRE fully and correctly is that partial anticipability is enough only if the path leading from the node where the expression becomes available (node 2 in the examples) to a node where the expression becomes anticipable (node 3 in the examples) contains no nodes at which the expression is neither anticipable nor available.

The last condition can be detected by two additional dataflow analyses, thus the full PRE algorithm requires four analyses in total. These are standard (total) availability and anticipability, and safe partial availability and safe partial anticipability analyses. The two latter depend on availability and anticipability. Their descriptions rely on the notion of safety. A program point is said to be safe wrt. an expression if that expression is either available or anticipable at that program point.

The dataflow inequations for the whole program in the CFG representation are the following.

$$
\begin{array}{rcl}
ANTOUT_i & \subseteq & \begin{cases} \emptyset & \text{if } i = f \\ \bigcap_{j \in succ(i)} ANTIN_j & \text{otherwise} \end{cases} \\
ANTIN_i = & \subseteq & ANTOUT_i \setminus MOD_i \cup EVAL_i \\
AVIN_i & \subseteq & \begin{cases} \emptyset & \text{if } i = s \\ \bigcap_{j \in pred(i)} AVOUT_i & \text{otherwise} \end{cases} \\
AVOUT_i & \subseteq & (AVIN_i \cup EVAL_i) \setminus MOD_i \\
SPANTOUT_i & \supseteq & \begin{cases} \emptyset & \text{if } i = f \\ \bigcup_{j \in succ(i)} SPANTIN_j & \text{otherwise} \end{cases} \\
SPANTIN_i & \supseteq & (SPANTOUT_i \setminus MOD_i \cup EVAL_i) \\
 & \cap & SAFEIN_i \\
SPAVIN_i & \supseteq & \begin{cases} \emptyset & \text{if } i = s \\ \bigcup_{j \in pred(i)} SPAVOUT_i & \text{otherwise} \end{cases} \\
SPAVOUT_i & \supseteq & ((SPAVIN_i \cup EVAL_i) \setminus MOD_i) \\
 & \cap & SAFEOUT_i
\end{array}
$$

$$
\begin{aligned}
SPANTOUT_i &\subseteq SAFEOUT_i \\
SPANTIN_i &\subseteq SAFEIN_i \\
SPAVIN_i &\subseteq SAFEIN_i \\
SPAVOUT_i &\subseteq SAFEOUT_i \\
SAFEIN_i &= ANTIN_i \cup AVIN_i \\
SAFEOUT_i &= ANTOUT_i \cup AVOUT_i
\end{aligned}
$$

Using the results of the analysis, it is possible to optimize the program in the following way. A computation of an expression should be added on edge (i, j), if the expression is safely partially available at the entry of node j, but not at the exit of node i. Furthermore, the expression should be safely partially anticipable at the entry of node j. This transformation makes partially redundant expressions fully redundant exactly in places where it is necessary, thus the checking of safe partial anticipability. Note that the latter was not necessary in simple PRE, since conditional partial availability already implied anticipability. In a node where an expression is evaluated if the expression is already safely partially available, its evaluation can be replaced with a use of the auxiliary variable. If the expression is not available, but is safely partially anticipable, the result of the evaluation can be saved in the auxiliary variable.

We now present these analyses and the optimization as type systems. The type system for anticipability was already described in Section 2. The type system for availability is very similar. Types $av \in \mathcal{P}(\mathbf{AExp}^+)$ are sets of nontrivial arithmetic expressions. Since availability is a forward must analysis, subtyping for availability is reversed, i.e., $\leq_{=\mathrm{df}\supseteq}$. The typing rules for availability are given in Figure 11.

The type systems for safe partial availability and safe partial anticipability are given as a single type system in Figure 12. They do not depend on each other, but depend on the safety component. We use $\underline{s}$ to denote a full type derivation of $s : av, ant \longrightarrow av', ant'$, thus safety $safe$ in the pretype of $\underline{s}$ is defined as $av \cup ant$, safety in the posttype $safe'$ is $av' \cup ant'$.

The complete type of a program point is thus $(av, ant, spav, spant) \in \mathcal{P}(\mathbf{AExp}^+) \times \mathcal{P}(\mathbf{AExp}^+) \times \mathcal{P}(\mathbf{AExp}^+) \times \mathcal{P}(\mathbf{AExp}^+)$, satisfying the conditions $spav \subseteq av \cup ant$ and $spant \subseteq av \cup ant$. Subtyping for safe partial availability is set inclusion, i.e., $\leq_{=\mathrm{df}\subseteq}$. For safe partial anticipability this is reversed, $\leq_{=\mathrm{df}\supseteq}$.

The optimizing type system for standard PRE is given in Figure 13. Code motion at subsumption is now guided by the intersection of safe partial availability and safe partial anticipability. In the definition of soundness, the similarity relation on states also has to be invoked at this intersection. The same holds for proof transformation.

4. Related Work

Proving compilers and optimizers correct is a vast subject. In this paper we have been interested in systematic descriptions of program optimizations with soundness and improvement arguments from which it is possible to isolate a soundness and improvement argument of the optimization for any given program. Such arguments give us automatic transformations of program proofs.

The type-systematic approach to dataflow analyses appears, e.g., in Nielson and Nielson's work on "flow logics" [17] (the "compositional" flow logics are for structured or high-level languages and the "abstract" ones for control-flow-graph like or low-level languages). In Benton's work [4] it appears for structured languages together with the relational method of stating and proving dataflow analyses and optimizations sound. Lerner et al. [14, 15] have looked at ways to make soundness arguments more systematic.

Automatic transformation of program proofs for nonoptimizing compilation and for optimizations has been considered by Barthe et al. [3, 2]. As a minor detail, differently from this paper, these works consider weakest precondition calculi instead of Hoare logics; the compilation work studies compilation from a high-level language to low-level language; the optimization work concerns a low-level language. More importantly, however, the approach to proof transformation for optimizers [2] does not deal properly with optimizations sound for similarity relations weaker than equality on the original program variables: to treat dead code elimination, dead assignments are not removed, but replaced by assignments to "shadow" variables, so the proofs produced in proof transformation do not pertain to the optimized program but a variant of it.

Seo et al. [21] and Chaieb [6] have noted that for program properties expressible in the standard Hoare logics ("extensional" properties), dataflow analysis results can be written down as Hoare logic proofs.

The question of formally proving the optimized versions of given programs improved has been studied by Aspinall et al. [1]. The same group of authors has also studied certification of resource consumption in general.

The linguistic differences between high-level and low-level languages that may seem of importance in works relating program analyses and program logics are in fact not deep and are overcome easily. Although analyses are typically stated for CFG like, low-level languages, and Hoare logics and wp-calculi are better known for structured, high-level languages, program logic has been done for low-level languages since Floyd (who considered control-flow graphs), with a renewed interest recently due to the advent of PCC, and dataflow analyses admit unproblematic direct structured descriptions for high-level languages as explicit, e.g., in the work on compositional flow logics.

In our own related earlier work [19, 20], we promoted the type-systematic method for describing analyses and optimizations and stating and proving them sound for dead-code elimination and common subexpression for a high-level language with deep expressions as well as for some stack-specific optimizations for a stack-based low-level language. We also explained the associated technology of automatic transformation of program proofs. In another piece of work [11], we spelled out the deeper relation between (special-purpose) type systems and Hoare logics usable to specify analyses of various degrees of precision on various levels on foundationality, subsuming the results by Seo et al. and Chaieb.

The optimization of partial redundancy elimination has a complicated history of nearly 30 years. Because of its power and sophistication it has been remarkably difficult to get right. The first definition of Morel and Renvoise [16] used a bidirectional analysis and so did many subsequent versions [9, 7] addressing its shortcomings. The formulations in the innovative work of Knoop et al. [12, 13] and the subsequent new wave of papers [10, 8, 5] are based on cascades of unidirectional analyses. The best motivated formulations today are those of Paleri et al. [18] and Xue and Knoop [22]. The one by Paleri et al. stands out by its relative simplicity thanks to certain symmetries and the ambition to provide an understandable soundness proof.

5. Conclusion

The thrust of this paper has been to show that the type-systematic approach to description of dataflow analyses and optimizations scales up viably to complicated optimizations maintaining its applicability to automatic transformation of program proofs. To this end, we have studied partial redundancy elimination, which is a highly nontrivial program transformation. In particular, it does edge splitting to place moved expression evaluations; type-systematically,

$$\frac{}{x := a : av \longrightarrow av \cup eval(a) \setminus mod(x)} \qquad \frac{}{\mathsf{skip} : av \longrightarrow av} \qquad \frac{s_0 : av \longrightarrow av'' \quad s_1 : av'' \longrightarrow av'}{s_0; s_1 : av \longrightarrow av'}$$

$$\frac{s_t : av \longrightarrow av' \quad s_f : av \longrightarrow av'}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : av \longrightarrow av'} \qquad \frac{s_t : av \longrightarrow av}{\mathsf{while}\ b\ \mathsf{do}\ s_t : av \longrightarrow av} \qquad \frac{av \leq av_0 \quad s : av_0 \longrightarrow av_0' \quad av_0' \leq av'}{s : av \longrightarrow av'}$$

Figure 11. Type system for available expressions

$$\frac{}{\underline{x := a} : (spant \backslash mod(x) \cup eval(a)) \cap safe, spav \longrightarrow spant, (spav \cup eval(a) \setminus mod(x)) \cap safe'}$$

$$\frac{}{\underline{\mathsf{skip}} : spant, spav \longrightarrow spant, spav} \qquad \frac{\underline{s_0} : spant, spav \longrightarrow spant'', spav'' \quad \underline{s_1} : spant'', spav'' \longrightarrow spant', spav'}{\underline{s_0; s_1} : spant, spav \longrightarrow spant', spav'}$$

$$\frac{\underline{s_t} : spant', spav \longrightarrow spant, spav' \quad \underline{s_f} : spant', spav \longrightarrow spant, spav'}{\underline{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f} : spant', spav \longrightarrow spant, spav'} \qquad \frac{\underline{s_t} : spant, spav \longrightarrow spant, spav}{\underline{\mathsf{while}\ b\ \mathsf{do}\ s_t} : spant, spav \longrightarrow spant, spav}$$

$$\frac{spant, spav \leq spant_0, spav_0 \quad \underline{s} : spant_0, spav_0 \longrightarrow spant_0', spav_0' \quad spant_0', spav_0' \leq spant', spav'}{\underline{s} : spant, spav \longrightarrow spant', spav'}$$

Figure 12. Type system for the underlying analyses of full PRE

$$\frac{a \in spav}{\underline{x := a} : (spant \backslash mod(x) \cup eval(a)) \cap safe, spav \longrightarrow spant, (spav \cup eval(a) \setminus mod(x)) \cap safe' \hookrightarrow x := nv(a)}$$

$$\frac{a \notin spav \quad a \in spant \quad x \notin FV(a)}{\underline{x := a} : (spant \backslash mod(x) \cup eval(a)) \cap safe, spav \longrightarrow spant, (spav \cup eval(a) \setminus mod(x)) \cap safe' \hookrightarrow x := nv(a) \hookrightarrow nv(a) := a; x := nv(a)}$$

$$\frac{a \notin spav \quad a \notin spant \vee x \in FV(a)}{\underline{x := a} : (spant \backslash mod(x) \cup eval(a)) \cap safe, spav \longrightarrow spant, (spav \cup eval(a) \setminus mod(x)) \cap safe' \hookrightarrow x := a}$$

$$\frac{}{\underline{\mathsf{skip}} : spant, spav \longrightarrow spant, spav \hookrightarrow \mathsf{skip}} \qquad \frac{\underline{s_0} : spant, spav \longrightarrow spant'', spav'' \hookrightarrow s_0' \quad \underline{s_1} : spant'', spav'' \longrightarrow spant', spav' \hookrightarrow s_1'}{\underline{s_0; s_1} : spant, spav \longrightarrow spant', spav' \hookrightarrow s_0'; s_1'}$$

$$\frac{\underline{s_t} : spant', spav \longrightarrow spant, spav' \hookrightarrow s_t' \quad \underline{s_f} : spant', spav \longrightarrow spant, spav' \hookrightarrow s_f'}{\underline{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f} : spant', spav \longrightarrow spant, spav' \hookrightarrow \mathsf{if}\ b\ \mathsf{then}\ s_t'\ \mathsf{else}\ s_f'} \qquad \frac{\underline{s_t} : spant, spav \longrightarrow spant, spav \hookrightarrow s_t'}{\underline{\mathsf{while}\ b\ \mathsf{do}\ s_t} : spant, spav \longrightarrow spant, spav \hookrightarrow \mathsf{while}\ b\ \mathsf{do}\ s_t'}$$

$$\frac{spant, spav \leq spant_0, spav_0 \quad \underline{s} : spant_0, spav_0 \longrightarrow spant_0', spav_0' \hookrightarrow s' \quad spant_0', spav_0' \leq spant', spav'}{\underline{s} : spant, spav \longrightarrow spant', spav' \hookrightarrow [\, nv(a) := a \mid a \in (spav_0 \cap spant_0) \setminus spav \,]; s'; [\, nv(a) := a \mid a \in (spav' \cap spant') \setminus spav_0' \,]}$$

Figure 13. Type system for full PRE, with the optimization component

this corresponds to new assignments appearing at subsumption inferences.

We have demonstrated that soundness and improvement stated in terms of type-indexed similarity relations and established with semantic arguments yield automatic transformations of functional correctness and resource usage proofs, a useful facility for the code producer in a scenario of proof-carrying code where proved code is optimized prior to shipping to the consumer.

Some issues for future work are the following.

- Modular soundness and improvement: In this paper, we stated and proved soundness and improvement of PRE in one monolithic step. But often, when an optimization is put together of a cascade of analyses followed by a transformation (as is the case also with PRE), it is possible to arrange the proofs accordingly, going through a series of instrumented semantics (or a series of interim "optimizations" which "implement" these semantics via the standard semantics). This does not necessarily give the shortest or simplest proof, but explains more, making explicit the contribution of each individual analysis. We would like to design a systematic framework for cascaded semantic arguments and proof transformations about such cascaded optimizations.
- Semantic arguments of improvement and optimality, their formalized versions: We have shown that PRE improves a program in a nonstrict sense, i.e., does not make it worse. In general this is the best improvement one can achieve, as an already optimal given program cannot be strictly improved. But strict improvement results must be possible for special situations, e.g., for loops from where expression evaluations are moved out. We plan to study this issue. Also, we have not shown that PRE yields an optimal program, i.e., one that cannot be improved further. One could dream of a systematic framework for optimality arguments. In such a framework one must be able to define a space of acceptable systematic modifications of programs; an optimal modification is then a sound acceptable modification improving more than any other.

- Type-systematic optimizations vs. type-indexed similarity relations used in semantic statements of soundness and improvement: right now, the similarity relations used in semantic statements of soundness and improvement appear crafted on an ad hoc basis. We intend to investigate systematic ways of relating type systems and the semantic similarity relations.

All of the above have an impact on automatic transformability of Hoare logic proofs. We expect some of the Cobalt and Rhodium work [14, 15] on automated soundness arguments to be of significance in addressing these issues.

Acknowledgement This work was supported by the Estonian Science Foundation grant no. 6940, the Estonian Doctoral School in ICT, the EU FP6 IST integrated project no. 15905 MOBIUS.

References

[1] D. Aspinall, L. Beringer, A. Momigliano. Optimisation validation. In *Proc. 5th Int. Wksh. on Compiler Optimization Meets Compiler Verification, COCV '06 (Vienna, Apr. 2006)*, v. 176, n. 3 of *Electron. Notes in Theor. Comput. Sci.*, pp. 37–59, 2007.

[2] G. Barthe, B. Grégoire, C. Kunz, T. Rezk. Certificate translation for optimizing compilers. In K. Yi, ed., *Proc. of 13th Int. Static Analysis Symp., SAS 2006 (Seoul, Aug. 2006)*, v. 4134 of *Lect. Notes in Comput. Sci.*, pp. 301–317. Springer, 2006.

[3] G. Barthe, T. Rezk, A. Saabas. Proof obligations preserving compilation. In T. Dimitrakos, F. Martinelli, P. Y. A. Ryan, S. Schneider, eds., *Revised Selected Papers from 3rd Int. Wksh. on Formal Aspects in Security and Trust, FAST 2005 (Newcastle upon Tyne, July 2005)*, v. 3866 of *Lect. Notes in Comput. Sci.*, pp. 112–126. Springer, 2006.

[4] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proc. of 31st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 2004 (Venice, Jan. 2004)*, pp. 14–25, ACM Press, 2004.

[5] D. Bronnikov. A practical adoption of partial redundancy elimination. *ACM SIGPLAN Notices*, v. 39, n. 8, pp. 49–53, 2004.

[6] A. Chaieb. Proof-producing program analysis. In K. Barkaoui, A. Cavalcanti, A. Cerone, eds., *Proc. of 3rd Int. Coll. on Theor. Aspects of Computing, ICTAC 2006 (Tunis, Nov. 2006)*, v. 4281 of *Lect. Notes in Comput. Sci.*, pp. 287–301. Springer, 2006.

[7] D. M. Dhamdhere. Practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Trans. on Program. Lang. and Syst.*, v. 13, n. 2, pp. 291–294, 1991.

[8] D. M. Dhamdhere. E-path_PRE—partial redundancy elimination made easy. *ACM SIGPLAN Notices*, v. 37, n. 8, pp. 53–65, 2002.

[9] K. H. Drechsler, M. P. Stadel. A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies". *ACM Trans. on Program. Lang. and Syst.*, v. 10, n. 4, pp. 635–640, 1988.

[10] K. H. Drechsler, M. P. Stadel. A variation of Knoop, Rüthing and Steffen's "Lazy code motion". *ACM SIGPLAN Notices*, v. 28, n. 5, pp. 29–38, 1993.

[11] M. J. Frade, A. Saabas, T. Uustalu. Foundational certification of dataflow analyses. In *Proc. of 1st IEEE and IFIP Int. Symp on Theor. Aspects of Software Engineering, TASE 2007 (Shanghai, June 2007)*, pp. 107–116. IEEE CS Press, 2007.

[12] J. Knoop, O. Rüthing, B. Steffen. Lazy code motion. In *Proc. of PLDI '92 (San Francisco, CA, June 1992)*, pp. 224–234. ACM Press, 1992.

[13] J. Knoop, O. Rüthing, B. Steffen. Optimal code motion: theory and practice. *ACM Trans. on Program. Lang. and Syst.*, v. 16, n. 4, pp. 1117-1155, 1994.

[14] S. Lerner, T. Millstein, C. Chambers. Automatically proving the correctness of compiler optimizations. In *Proc. of PLDI '03 (San Diego, CA, June 2003)*, pp. 220–231. ACM Press, 2003.

[15] S. Lerner, T. Millstein, E. Rice, C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proc. of 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL '05 (Long Beach, CA, Jan. 2005)*, pp. 364–377.

[16] E. Morel, C. Renvoise. Global optimization by suppression of partial redundancies. *Commun. of ACM*, v. 22, n. 2, pp. 96–103, 1979.

[17] H. R. Nielson, F. Nielson. Flow logic: a multi-paradigmatic approach to static analysis. In T. Æ. Mogensen, D. Smith, I. H. Sudborough, eds., *The Essence of Computation, Complexity, Analysis, Transformation: Essays Dedicated to Neil D. Jones*, v. 2566 of *Lect. Notes in Comput. Sci.*, pp. 223–244. Springer, 2002.

[18] V. K. Paleri, Y. N. Srikant, P. Shankar. Partial redundancy elimination: a simple, pragmatic, and provably correct algorithm. *Sci. of Comput. Program.*, v. 48, n. 1, pp. 1–20, 2003.

[19] A. Saabas, T. Uustalu. Program and proof optimizations with type systems. Submitted to *J. of Logic and Algebraic Program.*

[20] A. Saabas, T. Uustalu. Type systems for optimizing stack-based code. In M. Huisman, F. Spoto, eds., *Proc. of 2nd Wksh. on Bytecode Semantics, Verification, Analysis and Transformation, Bytecode 2007 (Braga, March 2007)*, v. 190, n. 1 of *Electron. Notes in Theor. Comput. Sci.*, pp. 103–119. Elsevier, 2007.

[21] S. Seo, H. Yang, K. Yi. Automatic construction of Hoare proofs from abstract interpretation results. In A. Ohori, ed., *Proc. of 1st Asian Symp. on Programming Languages and Systems, APLAS 2003 (Beijing, Nov. 2003)*, v. 2895 of *Lect. Notes in Comput. Sci.*, pp. 230–245. Springer, 2003.

[22] J. Xue, J. Knoop. A fresh look at PRE as a maximum flow problem. In A. Mycroft, A. Zeller, eds., *Proc. of 15th Int. Conf. on Compiler Construction, CC 2006 (Vienna, March 2006)*, *Lect. Notes in Comput. Sci.*, v. 3923, pp. 139–154. Springer, 2006.