

Proof Optimization for Partial Redundancy Elimination[☆]

Ando Saabas^a, Tarmo Uustalu^{a,1}

^a*Institute of Cybernetics, Tallinn University of Technology,
Akadeemia tee 21, EE-12618 Tallinn, Estonia*

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 automatic 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.

Key words: 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

[☆]This article is an expanded version of the conference paper in Proc. of 2008 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 2008 (San Francisco, CA, Jan. 2008), ACM Press, 2008, pp. 91–101.

Email addresses: `ando@cs.ioc.ee` (Ando Saabas), `tarmo@cs.ioc.ee` (Tarmo Uustalu)

¹Corresponding author

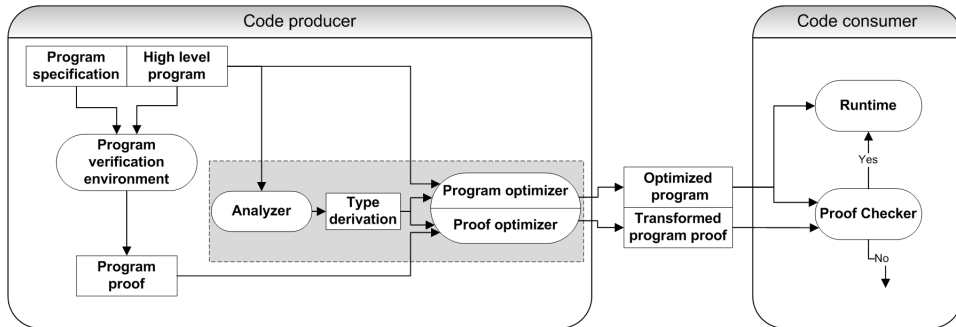


Figure 1: Proof optimization in PCC

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.

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 [4] 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 the strongest 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 [19]. 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 al. [20] and Xue and Knoop [26] 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 [13, 23, 24, 22], where we have already 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. In particular, in [23], we considered the simple examples of dead code elimination and common subexpression elimination to show that a type system can be used to prove semantic soundness of the optimization and that these arguments can be transferred to program proofs, yielding automatic program proof transformability. In [24], we gave type-systematic treatments of four stack usage optimizations for an operand-stack based low-level language, two of them based on bidirectional analyses.

Against this background, the present paper is first of all a demonstration of the scalability of our approach. PRE is made a particularly worthwhile case study by the fact that it is a very useful (and hence practical) optimization but at the same time subtle and complicated. But we also make two important novel general contributions. First, we show that our approach is smoothly adjustable to cover optimizations that change the structure of a program; specifically, we handle “edge splitting” by production of new code at subsumption inferences. Second, we demonstrate that the type-systematic approach facilitates not only soundness statements and arguments for optimizations but also statements and arguments of improvement. This is achieved with instrumented semantics and Hoare logics.

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 describe the same development for full PRE in the formulation of Paleri et al. [20]. Section 4 reviews some items of most related work and Section 5 concludes.

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 handle 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:

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

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, $\mathbf{State} =_{\text{df}} \mathbf{Var} \rightarrow \mathbb{Z}$. We write $\llbracket a \rrbracket \sigma$ (resp. $\llbracket b \rrbracket \sigma$) 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 $\sigma \succ_s \sigma'$. The notation $\sigma[x \mapsto z]$ refers to the state obtained by updating σ at x with value z . In the Hoare logic, the judgement that Q is a derivable postcondition for s and a precondition P is written $\{P\} s \{Q\}$. The notation $P[a/x]$ refers to the result of substituting a for x in P . For reference, the rules of the natural semantics and Hoare logic are given in Appendix A.

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

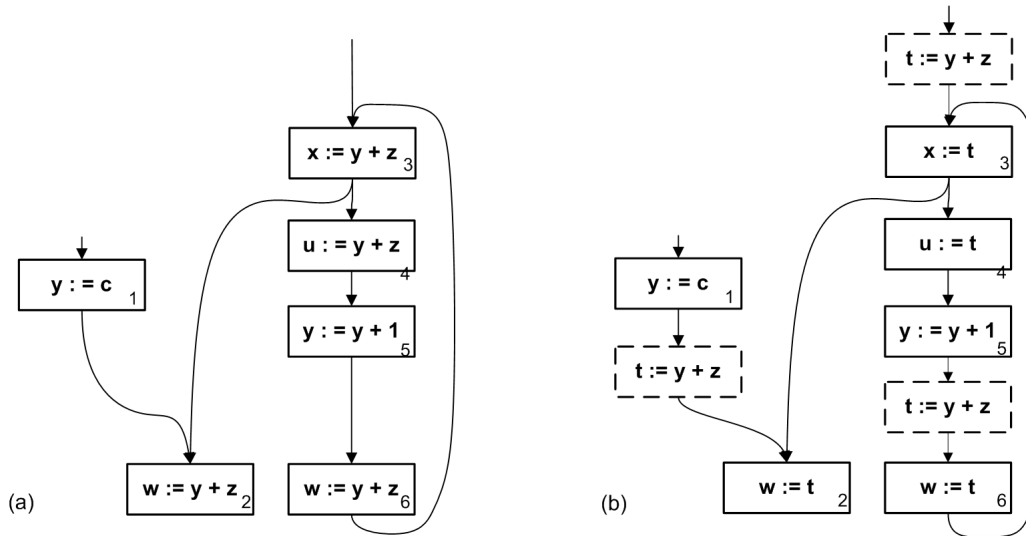


Figure 2: Example application of PRE

Figure 2(b) is the program after partial redundancy elimination optimization. Computations of $y + z$ in nodes 2, 3 and 4 are partially redundant in the original program and can be eliminated. The result of computation of $y + z$ at node 6 can be saved into an auxiliary variable t (thus a new node is added in front of node 6). 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 $y + z$ on the path leading from node 3 to 2 and two less in the loop. 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. More intuitively: if an expression is anticipable at some program point, it means that no matter which path we take from that program point, this expression will be evaluated at some point in the future, and moreover, none of the expression's operands are modified

until then. This means that expression could be precomputed at that program point, and the result saved in an auxiliary variable, since it can definitely be re-used later on. The *partial availability* analysis computes, for each program point, which expressions have already been evaluated and later not modified on some path through this program point. If an expression is partially available at a program point, then on some path the most recently computed value of this expression is still valid at this point, so if the computation result had been stored in an auxiliary variable, this variable could be used instead of the expression (a problem is however that this is not necessarily true about the other paths). Simple PRE uses a slightly modified version of this analysis (thus *conditional* partial availability), which depends on anticipability: an expression becomes partially available only when that expression is also anticipable. The crux of the design of simple PRE is this: the optimization guarantees that, if an expression is conditionally partially available at a point of the given program, then in the optimized program at the corresponding point it is totally available, i.e., evaluated and later not modified on all paths, with the value stored in the auxiliary variable agreeing with the expression value on all of them.

With both analyses computed, two changes may apply to assignments. If an expression is anticipable after an assignment, then it will definitely be evaluated later on in the program, so its corresponding auxiliary variable can be defined, to carry the result of the evaluation, and used instead of the expression in the assignment (provided that the assignment does itself not change the expression value). If the expression is also conditionally partially available before the assignment, we can assume it has already been computed, with the auxiliary variable readily holding its value, and simply replace the expression with the auxiliary variable. If neither case holds, we leave the assignment unchanged. Ensuring that conditionally partially available expressions are totally available in the optimized program necessitates additional expression evaluations: if an expression is not conditionally partially available at a point but is so at the successor point (this can happen, if the successor point is a control flow join point), an evaluation of this expression must be inserted between the two points, splitting the edge. That the optimization may remove some evaluations of an expression to re-insert some at different places produces the effect of expression evaluations being moved.

The standard-style inequational description of the algorithm for CFGs 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 inequations for the analyses are given below (*s* and *f* correspond to the start and finish nodes of the whole CFG).

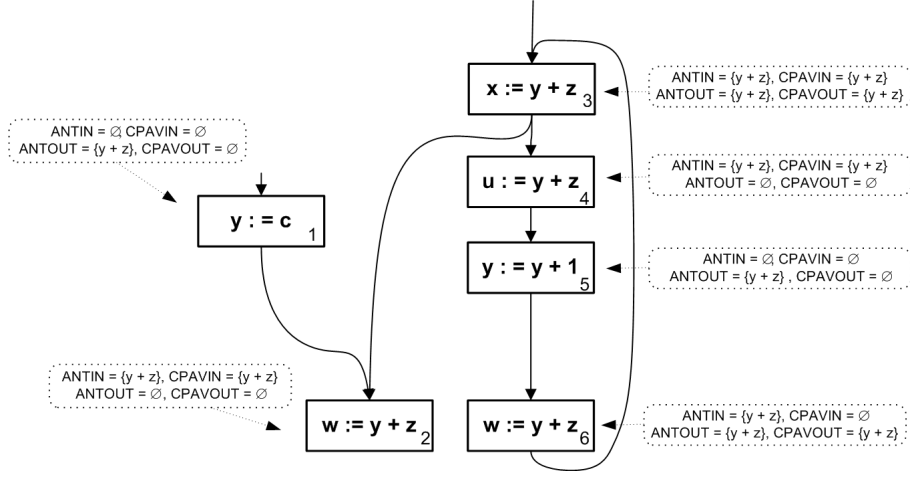


Figure 3: Property annotations on example program

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

(We use inequalities here instead of equalities to be more in line with our type system, since the type system will accept all valid analysis results, not only the strongest one. Also note that the last inequalities do not correspond to transfer functions. Instead they state a domain restriction on conditional partial availability sets.)

One solution of these inequations is given in Figure 3, leading to the optimization in Figure 2. For example, evaluations of $y+z$ at nodes 2, 3 and 4 can be replaced, since the expressions are conditionally partially available at the entry of the node ($CPAVIN$) and will be made totally available by the optimization. At node 6, the expression is anticipable at the exit, and not conditionally partially available at the entry, thus this is a place where the expression should be precomputed and stored in the temporary variable so in the given assignment the temporary variable can already replace it. Edge splitting happens between nodes 1 and 2, and at the entry edge into node 3. At the exit of node 1, the

expression is not conditionally partially available, but is so at the entry to node 2. Thus the edge needs to be split, and a new node computing the expression added, to ensure that the expression is totally available at the entry of node 2 in the optimized version. A similar thing happens to the entry edge of node 3.

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 the expressions in ant' are anticipable after running a statement s , then the expressions in ant are anticipable before the run. The typing rules are given in Figure 4. 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., $mod(x) =_{df} \{a \in \mathbf{AExp}^+ \mid x \in FV(a)\}$. 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 conditionally partially available before running s are in $cpav$, then the expressions conditionally partially available after the run are in $cpav'$.

$$\begin{array}{c}
\frac{}{x := a : ant' \setminus mod(x) \cup eval(a) \longrightarrow ant'} \quad \frac{}{skip : ant \longrightarrow ant} \quad \frac{s_0 : ant \longrightarrow ant'' \quad s_1 : ant'' \longrightarrow ant'}{s_0; s_1 : ant \longrightarrow ant'} \\
\frac{s_t : ant \longrightarrow ant' \quad s_f : ant \longrightarrow ant'}{if \ b \ then \ s_t \ else \ s_f : ant \longrightarrow ant'} \quad \frac{s_t : ant \longrightarrow ant}{while \ b \ do \ s_t : ant \longrightarrow ant} \\
\frac{ant \leq ant_0 \quad s : ant_0 \longrightarrow ant'_0 \quad ant'_0 \leq ant'}{s : ant \longrightarrow ant'}
\end{array}$$

Figure 4: Type system for anticipability

The typing rules of the combined type system are given in Figure 5. 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 conditionally partially available, if it is actually anticipable (certainly worth precomputing), thus the intersection with the anticipability type.

The optimization component of the type system is shown in Figure 6. 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 $:=_{3pre}$). 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 $:=_{2pre}$). The auxiliary function nv delivers a unique new auxiliary variable for every non-trivial arithmetic expression. Edge splitting is performed by the subsumption rule $conseq_{pre}$, which introduces auxiliary variable definitions when the conditional partial availability type grows (this typically happens at the beginning of loops and at the end of conditional branches and loop bodies). This is in fact the key rule of the type system. By the application of this rule, expressions that are partially available are made fully available. Subsumption is applied for example in conjunction with the if-statement, when an expression becomes available in one of the branches, but not in the other. In this case, to type the program, subsumption is applied to the posttype of the second branch, so that both branches get the same posttype. The optimization rule then says that new variable definitions need to be inserted where the subsumption took place. This is in fact what happened in Figure 3 on the edge between node 1 and 2 and also on the entry edge to node 3, although we did not speak about the type system there yet. Since the expression is not conditionally partially available at the exit of node 1, but is partially available at the entry of node 2, in the type system the types would have to be fitted with the help of subsumption, which triggers the edge splitting and introduction of the additional auxiliary variable definition.

$$\begin{array}{c}
\frac{}{x := a : \overline{ant' \setminus mod(x) \cup eval(a), cpav \longrightarrow ant', (cpav \cup eval(a) \setminus mod(x)) \cap ant'}} \\
\frac{}{skip : ant, cpav \longrightarrow ant, cpav} \quad \frac{s_0 : ant, cpav \longrightarrow ant'', cpav'' \quad s_1 : ant'', cpav'' \longrightarrow ant', cpav'}{s_0; s_1 : ant, cpav \longrightarrow ant', cpav'} \\
\frac{s_t : ant, cpav \longrightarrow ant', cpav' \quad s_f : ant, cpav \longrightarrow ant', cpav'}{if\ b\ then\ s_t\ else\ s_f : ant, cpav \longrightarrow ant', cpav'} \quad \frac{s_t : ant, cpav \longrightarrow ant, cpav}{while\ b\ do\ s_t : ant, cpav \longrightarrow ant, cpav} \\
\frac{ant, cpav \leq ant_0, cpav_0 \quad s : ant_0, cpav_0 \longrightarrow ant'_0, cpav'_0 \quad ant'_0, cpav'_0 \leq ant', cpav'}{s : ant, cpav \longrightarrow ant', cpav'}
\end{array}$$

Figure 5: Type system for the underlying analyses of simple PRE

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 [5]. We take soundness to mean that an original program and its optimized version simulate each other up to a similarity relation \sim on their states, indexed by conditional partial availability types of program points.

Let $\sigma \sim_{cpav} \sigma_*$ denote that two states σ and σ_* of an original and optimized program agree on the auxiliary variables wrt. $cpav \subseteq \mathbf{AExp}^+$ in the sense that $\forall x \in \mathbf{Var}. \sigma(x) = \sigma_*(x)$ and $\forall a \in cpav. \llbracket a \rrbracket \sigma = \sigma_*(nv(a))$. 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 $\sigma \succ s \rightarrow \sigma'$.

Although our language is deterministic, nontermination is possible, therefore both directions (preservation and reflection of evaluations) are necessary to establish equitermination. Reflection in particular establishes that a compiled program cannot terminate more often than the original form.

PROOF. We only prove the first half of the theorem. The proof of the other half is similar.

Given $s : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s_*$, $\sigma \sim_{cpav} \sigma_*$ and $\sigma \succ s \rightarrow \sigma'$, we must find 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. We look at the following nontrivial cases.

- Case $:=_{pre}$: The type derivation is of the form

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

where $ant =_{df} ant' \setminus mod(x) \cup eval(a)$, $cpav' =_{df} (cpav \cup eval(a) \setminus mod(x)) \cap ant'$. We note that in particular this means that $cpav \cup eval(a) \supseteq cpav'$

$$\begin{array}{c}
\frac{a \notin cpav \quad a \notin ant' \setminus mod(x)}{x := a : ant' \setminus mod(x) \cup eval(a), cpav \longrightarrow ant', (cpav \cup eval(a) \setminus mod(x)) \cap ant' \hookrightarrow x := a} \text{:=}_{1\text{pre}} \\
\frac{a \notin cpav \quad a \in ant' \setminus mod(x)}{x := a : ant' \setminus mod(x) \cup eval(a), cpav \longrightarrow ant', (cpav \cup eval(a) \setminus mod(x)) \cap ant' \hookrightarrow nv(a) := a; x := nv(a)} \text{:=}_{2\text{pre}} \\
\frac{a \in cpav}{x := a : ant' \setminus mod(x) \cup eval(a), cpav \longrightarrow ant', (cpav \cup eval(a) \setminus mod(x)) \cap ant' \hookrightarrow x := nv(a)} \text{:=}_{3\text{pre}} \\
\frac{}{\text{skip} : ant, cpav \longrightarrow ant, cpav \hookrightarrow \text{skip}} \text{skip}_{\text{pre}} \\
\frac{s_0 : ant, cpav \longrightarrow ant'', cpav'' \hookrightarrow s'_0 \quad s_1 : ant'', cpav'' \longrightarrow ant', cpav' \hookrightarrow s'_1}{s_0; s_1 : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s'_0; s'_1} \text{comp}_{\text{pre}} \\
\frac{s_t : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s'_t \quad s_f : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s'_f}{\text{if } b \text{ then } s_t \text{ else } s_f : ant, cpav \longrightarrow ant', cpav' \hookrightarrow \text{if } b \text{ then } s'_t \text{ else } s'_f} \text{if}_{\text{pre}} \\
\frac{s_t : ant, cpav \longrightarrow ant, cpav \hookrightarrow s'_t}{\text{while } b \text{ do } s_t : ant, cpav \longrightarrow ant, cpav \hookrightarrow \text{while } b \text{ do } s'_t} \text{while}_{\text{pre}} \\
\frac{ant, cpav \leq ant_0, cpav_0 \quad s : ant_0, cpav_0 \longrightarrow ant'_0, cpav'_0 \hookrightarrow s' \quad ant'_0, cpav'_0 \leq ant', cpav'}{s : ant, cpav \longrightarrow ant', cpav' \hookrightarrow [nv(a) := a \mid a \in cpav_0 \setminus cpav]; s'; [nv(a) := a \mid a \in cpav' \setminus cpav'_0]} \text{conseq}_{\text{pre}}
\end{array}$$

Figure 6: Type system for simple PRE, with the optimization component

and $cpav' \cap mod(x) = \emptyset$. The corresponding given semantic derivation must be of the form

$$\overline{\sigma \succ x := a \rightarrow \sigma[x \mapsto \llbracket a \rrbracket \sigma]}$$

hence $\sigma' = \sigma[x \mapsto \llbracket a \rrbracket \sigma]$.

- Subcase $\text{:=}_{1\text{pre}}$: We know that $a \notin cpav$. We also know that either $a \notin ant'$ or $a \in mod(x)$, so $cpav \supseteq cpav'$. Moreover, $s_* =_{\text{df}} x := a$.

We have the semantic derivation

$$\overline{\sigma_* \succ x := a \rightarrow \sigma'_*}$$

where $\sigma'_* =_{\text{df}} \sigma_*[x \mapsto \llbracket a \rrbracket \sigma_*]$. From $\sigma \sim_{cpav} \sigma_*$ it follows that $\llbracket a \rrbracket \sigma = \llbracket a \rrbracket \sigma_*$, so that, using $cpav \supseteq cpav'$ as well, we can conclude $\sigma' = \sigma[x \mapsto \llbracket a \rrbracket \sigma] \sim_{cpav'} \sigma_*[x \mapsto \llbracket a \rrbracket \sigma] = \sigma_*[x \mapsto \llbracket a \rrbracket \sigma_*] = \sigma'_*$.

- Subcase $\text{:=}_{2\text{pre}}$: We have that $a \notin cpav$. We also have that a is nontrivial and $cpav \cup \{a\} \supseteq cpav'$. Also, $s_* =_{\text{df}} nv(a) := a; x := nv(a)$. We have the semantic derivation

$$\frac{\overline{\sigma_* \succ nv(a) := a \rightarrow \sigma''_*} \quad \overline{\sigma''_* \succ x := nv(a) \rightarrow \sigma'_*}}{\sigma_* \succ nv(a) := a; x := nv(a) \rightarrow \sigma'_*}$$

where $\sigma''_* =_{\text{df}} \sigma_*[nv(a) \mapsto \llbracket a \rrbracket \sigma_*]$ and $\sigma'_* =_{\text{df}} \sigma''_*[x \mapsto \sigma''_*(nv(a))] = \sigma''_*[x \mapsto \llbracket a \rrbracket \sigma_*]$. From $\sigma \sim_{cpav} \sigma_*$ it is immediate that $\sigma \sim_{cpav \cup \{a\}} \sigma_*[nv(a) \mapsto \llbracket a \rrbracket \sigma] = \sigma_*[nv(a) \mapsto \llbracket a \rrbracket \sigma_*] = \sigma''_*$ and therefore by $cpav \cup \{a\} \supseteq cpav'$ we have $\sigma' = \sigma[x \mapsto \llbracket a \rrbracket \sigma] \sim_{cpav'} \sigma''_*[x \mapsto \llbracket a \rrbracket \sigma] = \sigma''_*[x \mapsto \llbracket a \rrbracket \sigma_*] = \sigma'_*$.

- Subcase $:=_{3\text{pre}}$: We have that $a \in cpav$, so it follows that $cpav \supseteq cpav'$. We have $s_* =_{\text{df}} x := nv(a)$. We have the semantic derivation

$$\overline{\sigma_* \succ x := nv(a) \rightarrow \sigma'_*}$$

where $\sigma'_* =_{\text{df}} \sigma_*[x \mapsto \sigma_*(nv(a))]$. We know that $a \in cpav$, so from $\sigma \sim_{cpav} \sigma_*$ we learn $\llbracket a \rrbracket \sigma = \sigma_*(nv(a))$. Further, using also that $cpav \supseteq cpav'$, we realize that $\sigma' = \sigma[x \mapsto \llbracket a \rrbracket \sigma] \sim_{cpav'} \sigma_*[x \mapsto \llbracket a \rrbracket \sigma] = \sigma_*[x \mapsto \sigma_*(nv(a))] = \sigma'_*$.

- Case $\text{conseq}_{\text{pre}}$: The type derivation is of the form

$$\frac{\begin{array}{c} \vdots \\ s : ant_0, cpav_0 \longrightarrow ant'_0, cpav'_0 \hookrightarrow s_* \end{array}}{s : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s'; s_*; s''}$$

where $(ant, cpav) \leq (ant_0, cpav_0)$, $(ant'_0, cpav'_0) \leq (ant', cpav')$, $s' =_{\text{df}} [nv(a) := a \mid a \in cpav_0 \setminus cpav]$ and $s'' =_{\text{df}} [nv(a) := a \mid a \in cpav' \setminus cpav'_0]$. First we find σ_0 such that $\sigma_* \succ s' \rightarrow \sigma_0$ and $\sigma \sim_{cpav_0} \sigma_0$. We have the semantic derivation

$$\overline{\sigma_* \succ s' \rightarrow \sigma_0}$$

where $\sigma_0 =_{\text{df}} \sigma_*[nv(a) \mapsto \llbracket a \rrbracket \sigma_* \mid a \in cpav_0 \setminus cpav]$. From $\sigma \sim_{cpav} \sigma_*$ using $cpav \subseteq cpav_0$ we get that $\sigma \sim_{cpav_0} \sigma_*[nv(a) \mapsto \llbracket a \rrbracket \sigma \mid a \in cpav_0 \setminus cpav] = \sigma_*[nv(a) \mapsto \llbracket a \rrbracket \sigma_* \mid a \in cpav_0 \setminus cpav] = \sigma_0$, since every expression in the difference of $cpav_0$ and $cpav$ is explicitly made equal to its corresponding auxiliary variable and no variables from **Var** are modified. From the induction hypothesis we obtain that there is a state σ_1 such that $\sigma_0 \succ s_* \rightarrow \sigma_1$ and $\sigma' \sim_{cpav'_0} \sigma_1$. It is now enough to show that there is a state σ'_* such that $\sigma_1 \succ s'' \rightarrow \sigma'_*$ and $\sigma' \sim_{cpav'} \sigma'_*$. Similarly for the case of s' , we have the derivation

$$\overline{\sigma_1 \succ s'' \rightarrow \sigma'_*}$$

where $\sigma'_* =_{\text{df}} \sigma_1[nv(a) \mapsto \llbracket a \rrbracket \sigma_1 \mid a \in cpav' \setminus cpav'_0]$. Again it is easy to realize that $\sigma' \sim_{cpav'_0} \sigma_1$ with $cpav'_0 \subseteq cpav'$ gives us $\sigma' \sim_{cpav'} \sigma_1[nv(a) \mapsto \llbracket a \rrbracket \sigma' \mid a \in cpav' \setminus cpav'_0] = \sigma_1[nv(a) \mapsto \llbracket a \rrbracket \sigma_1 \mid a \in cpav' \setminus cpav'_0] = \sigma'_*$. \square

It is possible to show more than just soundness of the optimization 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 expression motion might introduce unneeded evaluations.

To state this property, we must have a way to count the expression evaluations. This can be done in a simple instrumented semantics. In this 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 essentially as for the standard semantics, except that for assignments of nontrivial expressions we stipulate

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

The corresponding similarity relation between the states of an original and optimized program 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 $\forall a \in \mathbf{AExp}^+ \setminus cpav. r_*(a) \leq r(a)$.

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 conditional partial availability 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 $(\sigma, r) \succ_s \rightarrow (\sigma', r')$.

PROOF. Again we only show the proof of the first part. The reasoning is similar to that we showed in the proof of Theorem 1, so we only worry about the evaluation counting states and ignore the standard states.

Given $s : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s_*$, $(\sigma, r) \approx_{cpav} (\sigma_*, r_*)$ and $(\sigma, r) \succ_s \rightarrow (\sigma', r')$, we are after a state (σ'_*, r'_*) such that $(\sigma', r') \approx_{cpav'} (\sigma'_*, r'_*)$ and $(\sigma_*, r_*) \succ_{s_*} \rightarrow (\sigma'_*, r'_*)$. The proof is by induction on the type derivation and we look at the following nontrivial cases.

- Case $:=_{pre}$: The type derivation is of the form

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

where $ant =_{\text{df}} ant' \setminus mod(x) \cup eval(a)$, $cpav' =_{\text{df}} (cpav \cup eval(a) \setminus mod(x)) \cap ant'$.

We note that from the constraint $cpav \subseteq ant$ it follows that $cpav \subseteq cpav' \cup eval(a)$:

$$\begin{aligned}
cpav &= cpav \cap ant \\
&= cpav \cap ((ant' \setminus mod(x)) \cup eval(a)) \\
&\subseteq (cpav \setminus mod(x) \cap ant') \cup eval(a) \\
&= ((cpav \cup eval(a)) \setminus mod(x) \cap ant') \cup eval(a) \\
&= cpav' \cup eval(a)
\end{aligned}$$

At the same time also $cpav \cup eval(a) \supseteq (cpav \cup eval(a)) \setminus mod(x) \cap ant' = cpav'$. So for any nontrivial expression $a' \neq a$, $a' \in cpav$ and $a' \in cpav'$ are in fact equivalent.

The given semantic derivation must be of the form

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

so $\sigma' = \sigma[x \mapsto \llbracket a \rrbracket \sigma]$, $r' = r[a \mapsto r(a) + 1]$.

- Subcase $:=_{1\text{pre}}$: We have that $a \notin cpav$ and either $a \notin ant'$ or $a \in mod(x)$, so $a \notin cpav'$. Moreover, $s_* =_{\text{df}} x := a$.

We have the semantic derivation

$$\overline{(\sigma_*, r_*) \succ x := a \rightarrow (\sigma'_*, r'_*)} .$$

where $(\sigma'_*, r'_*) =_{\text{df}} (\sigma_*[x \mapsto \llbracket a \rrbracket \sigma_*], r_*[a \mapsto r_*(a) + 1])$. From the assumption we know that $r_*(a) \leq r(a)$, so $r'_*(a) = r_*(a) + 1 \leq r(a) + 1 = r'(a)$.

- Subcase $:=_{2\text{pre}}$: We have that $a \notin cpav$ and both $a \in ant'$ and $a \notin mod(x)$, hence $a \in cpav'$. Moreover, $s_* =_{\text{df}} nv(a) := a; x := nv(a)$.

We have the semantic derivation

$$\frac{\overline{(\sigma_*, r_*) \succ nv(a) := a \rightarrow \sigma''_*, r''_*} \quad \overline{(\sigma''_*, r''_*) \succ x := nv(a) \rightarrow (\sigma'_*, r'_*)}}{(\sigma_*, r_*) \succ nv(a) := a; x := nv(a) \rightarrow (\sigma'_*, r'_*)} .$$

where $(\sigma''_*, r''_*) =_{\text{df}} (\sigma_*[nv(a) \mapsto \llbracket a \rrbracket \sigma_*], r_*[a \mapsto r_*(a) + 1])$ and $(\sigma'_*, r'_*) =_{\text{df}} (\sigma''_*[x \mapsto \sigma''_*(nv(a))], r''_*)$. Similarly to the previous case, from $r_*(a) \leq r(a)$ we obtain $r'_*(a) = r''_*(a) = r_*(a) + 1 \leq r(a) + 1 = r'(a) < r'(a) + 1$.

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

$$\overline{(\sigma_*, r_*) \succ x := nv(a) \rightarrow (\sigma'_*, r'_*)} .$$

where $(\sigma'_*, r'_*) =_{\text{df}} (\sigma_*[x \mapsto \sigma_*(nv(a))], r_*)$. From $r_*(a) \leq r(a) + 1$ we get that $r'_*(a) = r_*(a) \leq r(a) + 1 = r'(a)$ (so all is well both if $a \notin cpav'$ and if $a \in cpav'$; both situations are possible).

In all three subcases, for any nontrivial $a' \neq a$, if $a' \in cpav$, then we have that $a' \in cpav'$ as well and therefore from $r_*(a') \leq r(a') + 1$ we get $r'_*(a') = r_*(a') \leq r(a') + 1 = r'(a') + 1$. Similarly, for $a' \neq a$ such that $a' \notin cpav$ we have $a' \notin cpav'$, so $r_*(a') \leq r(a')$ gives us $r'_*(a') = r_*(a') \leq r(a') = r'(a')$.

- Case $\text{conseq}_{\text{pre}}$: The type derivation is of the form

$$\frac{\begin{array}{c} \vdots \\ s : ant_0, cpav_0 \longrightarrow ant'_0, cpav'_0 \hookrightarrow s_* \end{array}}{s : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s'; s_*; s''}$$

where $(ant, cpav) \leq (ant_0, cpav_0)$, $(ant'_0, cpav'_0) \leq (ant', cpav')$ and $s' =_{\text{df}} [nv(a) := a \mid a \in cpav_0 \setminus cpav]$, $s'' =_{\text{df}} [nv(a) := a \mid a \in cpav' \setminus cpav'_0]$. First we find a state (σ_0, r_0) such that $(\sigma_*, r_*) \succ_{s'} (\sigma_0, r_0)$ and $(\sigma, r) \approx_{cpav_0} (\sigma_0, r_0)$.

We have the semantic derivation

$$\overline{(\sigma_*, r_*) \succ_{s'} (\sigma_0, r_0)} .$$

where $(\sigma_0, r_0) =_{\text{df}} (\sigma_*[nv(a) \mapsto \llbracket a \rrbracket \sigma_* \mid a \in cpav_0 \setminus cpav], r_*[a \mapsto r_*(a) + 1 \mid a \in cpav_0 \setminus cpav])$. For any expression $a \in cpav$, from $cpav \subseteq cpav_0$ we have $a \in cpav_0$, whereas from $r_*(a) \leq r(a) + 1$ we can conclude $r_0(a) = r_*(a) \leq r(a) + 1$. Similarly, for any nontrivial expression $a \notin cpav_0$, from $cpav \subseteq cpav_0$ we learn that $a \notin cpav$, and then $r_*(a) \leq r(a)$ tells us that $r_0(a) = r_*(a) \leq r(a)$. If an expression a is in $cpav_0 \setminus cpav$, then from $r_*(a) \leq r(a)$ we can conclude $r_0(a) = r_*(a) + 1 \leq r(a) + 1$. Hence, $(\sigma, r) \approx_{cpav_0} (\sigma_0, r_0)$ as desired.

By the induction hypothesis, there must exist a state (σ_1, r_1) such that $(\sigma_0, r_0) \succ_{s_*} (\sigma_1, r_1)$ and $(\sigma', r') \approx_{cpav'_0} (\sigma_1, r_1)$. It is now enough to exhibit a state (σ'_*, r'_*) such that $(\sigma_1, r_1) \succ_{s''} (\sigma'_*, r'_*)$ and $(\sigma', r') \approx_{cpav'} (\sigma'_*, r'_*)$. This can be done in the same way as for s' . \square

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 for simple PRE

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, which can be seen as the crux of the paper:

Theorem 3 (Preservation of Hoare logic provability/proofs).

If $s : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s_$, then
 $\neg\{P\} s \{Q\}$ implies $\{P|_{cpav}\} s_* \{Q|_{cpav'}\}$.*

The main significance of the theorem is not that Hoare logic provability is preserved per se (which is pretty obvious, since the optimization is sound—we show it in the nonconstructive proof). Rather, the major contribution here is the constructive proof, which gives us an algorithm for automatic transformation of proofs using the type system.

PROOF. Nonconstructively, this theorem is a corollary from soundness of the optimization (Theorem 1, second half) and soundness and completeness of the Hoare logic (Theorems 9, 10): Assume that $s : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s'$ and $\{P\} s \{Q\}$. We must show $\{P|_{cpav}\} s_* \{Q|_{cpav'}\}$. By completeness of the Hoare logic, it suffices to show that, for any states σ_* , σ'_* and valuation α of logic variables, $\sigma_* \models_\alpha P|_{cpav}$ and $\sigma_* \succ_{s_*} \sigma'_*$ imply $\sigma'_* \models_\alpha Q|_{cpav'}$.

Consider any σ_* , σ'_* and α such that $\sigma_* \models_\alpha P|_{cpav}$ and $\sigma_* \succ_{s_*} \sigma'_*$. From $\sigma_* \models_\alpha P|_{cpav}$, it follows that there must exist σ such that $\sigma \sim_{cpav} \sigma_*$ and $\sigma \models_\alpha P$.

By Theorem 1 (second half) the facts $s : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s'$, $\sigma \sim_{cpav} \sigma_*$ and $\sigma_* \succ_{s_*} \sigma'_*$ yield that there must exist σ' such that $\sigma' \sim_{cpav'} \sigma'_*$ and $\sigma \succ_{s'} \sigma'$.

From $\{P\} s \{Q\}$, $\sigma \models_\alpha P$ and $\sigma \succ_{s'} \sigma'$, using soundness of the Hoare logic, we conclude that $\sigma' \models_\alpha Q$. Combined with $\sigma' \sim_{cpav'} \sigma'_*$, this gives us $\sigma'_* \models_\alpha Q|_{cpav'}$ as required.

We now present the constructive proof that yields automatic Hoare proof transformation. Given a derivation of $s : ant, cpav \longrightarrow ant', cpav' \hookrightarrow s_*$ and an aligned Hoare proof of $\{P\} s \{Q\}$, we induct on the type derivation and transform the given Hoare proof into one of $\{P|_{cpav}\} s_* \{Q|_{cpav'}\}$.

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

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

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

where $ant =_{\text{df}} ant' \setminus mod(x) \cup eval(a)$, $cpav' =_{\text{df}} (cpav \cup eval(a) \setminus mod(x)) \cap ant'$. We notice that this implies $cpav \cup eval(a) \supseteq cpav'$ and $cpav' \cap mod(x) = \emptyset$. The latter observation gives that $P[a'/x]|_{cpav'} \Leftrightarrow P|_{cpav'}[a'/x]$ for any a' and P .

The given Hoare logic proof is

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

- Subcase $:=_{1\text{pre}}$: We have that $a \notin cpav$. We also have that either $a \notin ant'$ or $a \in mod(x)$, so $cpav \supseteq cpav'$. Moreover, $s_* =_{\text{df}} x := a$. From $cpav \supseteq cpav'$ it follows that $P[a/x]|_{cpav} \models P[a/x]|_{cpav'}$. The transformed Hoare logic proof is

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

- Subcase $:=_{2\text{pre}}$: We have that $a \notin cpav$. We also have that a is nontrivial, so that $cpav \cup \{a\} \supseteq cpav'$. And $s_* =_{\text{df}} nv(a) := a; x := nv(a)$. From $cpav \cup \{a\} \supseteq cpav'$ it follows that $P[nv(a)/x]|_{cpav \cup \{a\}} \models P[nv(a)/x]|_{cpav'}$. 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

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

where $B_0 \equiv$

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

and $B_1 \equiv$

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

- Subcase $:=_{3\text{pre}}$: We have that $a \in cpav$, so $cpav \supseteq cpav'$. Moreover, $s_* =_{\text{df}} x := nv(a)$. From $a \in cpav$ and $cpav \supseteq cpav'$ it follows that $P[a/x]|_{cpav} \models P[a/x]|_{cpav'} \wedge nv(a) = a$. 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

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

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

$$\frac{\begin{array}{c} \vdots \\ s : \text{ant}_0, \text{cpav}_0 \longrightarrow \text{ant}'_0, \text{cpav}'_0 \hookrightarrow s_* \end{array}}{s : \text{ant}, \text{cpav} \longrightarrow \text{ant}', \text{cpav}' \hookrightarrow s'; s_*; s''}$$

where $(\text{ant}, \text{cpav}) \leq (\text{ant}_0, \text{cpav}_0)$, $(\text{ant}'_0, \text{cpav}'_0) \leq (\text{ant}, \text{cpav})$ and $s' =_{\text{df}} [nv(a) := a \mid a \in \text{cpav}_0 \setminus \text{cpav}]$, $s'' =_{\text{df}} [nv(a) := a \mid a \in \text{cpav}' \setminus \text{cpav}'_0]$. The given Hoare logic proof is

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

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 $P|_{cpav} \models P_0|_{cpav}$.

By reflexivity of equality $P_0|_{cpav} \Leftrightarrow P_0|_{cpav} \wedge \bigwedge [a = a \mid a \in \text{cpav}_0 \setminus \text{cpav}] \models P_0|_{cpav_0}[a/nv(a) \mid a \in \text{cpav}_0 \setminus \text{cpav}]$. Hence from the axiom $\{P_0|_{cpav_0}[a/nv(a) \mid a \in \text{cpav}_0 \setminus \text{cpav}]\} s' \{P_0|_{cpav_0}\}$ 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. \square

An example application of the type system and transformation of Hoare logic proofs is shown in Figures 7, 8 and 9. We have a program $s =_{\text{df}} \text{while } i < k \text{ do } (n := n + (y + z); i := i + 1); x := y + z$ 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

$$\begin{array}{c}
\frac{n := n + (y + z) : \{y + z\}, \{y + z\} \longrightarrow \{y + z\}, \{y + z\} \quad \hookrightarrow n := n + t}{\quad} \quad \frac{x := y + z : \{y + z\}, \{y + z\} \longrightarrow \emptyset, \emptyset \quad \hookrightarrow x := t}{\quad} \\
\left| \quad \frac{\frac{i := i + 1 : \{y + z\}, \{y + z\} \longrightarrow \{y + z\}, \{y + z\} \quad \hookrightarrow i := i + 1}{\quad}}{n := n + (y + z); i := i + 1 : \{y + z\}, \{y + z\} \longrightarrow \{y + z\}, \{y + z\} \quad \hookrightarrow n := n + t; i := i + 1} \right. \\
\frac{\text{while } i < k \text{ do } (n := n + (y + z); i := i + 1) : \{y + z\}, \{y + z\} \longrightarrow \{y + z\}, \{y + z\} \quad \hookrightarrow \text{while } i < k \text{ do } (n := n + t; i := i + 1)}{\quad} \\
\frac{\text{while } i < k \text{ do } (n := n + (y + z); i := i + 1) : \{y + z\}, \emptyset \longrightarrow \{y + z\}, \{y + z\} \quad \hookrightarrow t := y + z; \text{while } i < k \text{ do } (n := n + t; i := i + 1)}{\quad} \\
\frac{\text{while } i < k \text{ do } (n := n + (y + z); i := i + 1); x := y + z : \{y + z\}, \emptyset \longrightarrow \emptyset, \emptyset \quad \hookrightarrow t := y + z; \text{while } i < k \text{ do } (n := n + t; i := i + 1); x := t}{\quad}
\end{array}$$

Figure 7: Type derivation for the example program

$$\begin{array}{c}
\frac{\quad}{\{n = k * (y + z)\} x := y + z \{n = k * (y + z)\}} \\
\frac{\quad}{\{i + 1 \leq k \wedge n = (i + 1) * (y + z)\} i := i + 1 \{i \leq k \wedge n = i * (y + z)\}} \\
\frac{\frac{i + 1 \leq k \wedge n + (y + z) = (i + 1) * (y + z) \{n := n + (y + z)\} \{i + 1 \leq k \wedge n = (i + 1) * (y + z)\}}{\{i < k \wedge n = i * (y + z)\} n := n + (y + z); i := i + 1 \{i \leq k \wedge n = i * (y + z)\}}}{\{i \leq k \wedge n = i * (y + z)\} \text{while } i < k \text{ do } (n := n + (y + z); i := i + 1) \{i \not\leq k \wedge i \leq k \wedge n = i * (y + z)\}} \\
\frac{\quad}{\{n = 0 \wedge i = 0 \wedge k \geq 0\} \text{while } i < k \text{ do } (n := n + (y + z); i := i + 1); x := y + z \{n = k * (y + z)\}}
\end{array}$$

Figure 8: The original proof for the example program

$$\begin{array}{c}
\frac{\quad}{\{n = k * (y + z)\} x := t \{n = k * (y + z)\}} \\
\frac{\quad}{\{ \wedge \quad \frac{n = k * (y + z)}{t = y + z} \} x := t \{n = k * (y + z)\}} \\
\frac{\quad}{\{ \wedge \quad \frac{i + 1 \leq k \wedge n = (i + 1) * (y + z)}{t = y + z} \} i := i + 1 \{ \wedge \quad \frac{i \leq k \wedge n = i * (y + z)}{t = y + z} \}} \\
\frac{\quad}{\{i \leq k \wedge n = i * (y + z)\} t := y + z \{ \wedge \quad \frac{i \leq k \wedge n = i * (y + z)}{t = y + z} \}} \\
\left| \quad \frac{\frac{\frac{\frac{\{ \wedge \quad \frac{i + 1 \leq k \wedge n + t = (i + 1) * (y + z)}{t = y + z} \} n := n + t \{ \wedge \quad \frac{i + 1 \leq k \wedge n = (i + 1) * (y + z)}{t = y + z} \}}{\{ \wedge \quad \frac{i < k \wedge n = i * (y + z)}{t = y + z} \} n := n + t; i := i + 1 \{ \wedge \quad \frac{i \leq k \wedge n = i * (y + z)}{t = y + z} \}}}{\{ \wedge \quad \frac{i \leq k \wedge n = i * (y + z)}{t = y + z} \} \text{while } i < k \text{ do } \quad \frac{n := n + (y + z); i := i + 1 \quad \{ \wedge \quad \frac{i \not\leq k \wedge i \leq k \wedge n = i * (y + z)}{t = y + z} \}}}{\{i \leq k \wedge n = i * (y + z)\} \quad \frac{t := y + z; \text{while } i < k \text{ do } (n := n + (y + z); i := i + 1) \quad \{ \wedge \quad \frac{i = k \wedge n = i * (y + z)}{t = y + z} \}}}{\quad} \\
\frac{\quad}{\{n = 0 \wedge i = 0 \wedge k \geq 0\} \quad \frac{t := y + z; \text{while } i < k \text{ do } (n := n + t; i := i + 1); x := t \quad \{n = k * (y + z)\}}{\quad}}
\end{array}$$

Figure 9: The transformed proof

$$\begin{array}{c}
\frac{}{\{i+1 \leq k \wedge c+1 = i+1\} n := n + (y+z) \{i+1 \leq k \wedge c = (i+1)\}} \\
\frac{}{\{c+1 = k+1\} x := y+z \{c = k+1\}} \\
\frac{}{\{i+1 \leq k \wedge c = i+1\} i := i+1 \{i \leq k \wedge c = i\}} \\
\frac{}{\{i < k \wedge c = i\} n := n + (y+z); i := i+1 \{i \leq k \wedge c = i\}} \\
\frac{}{\{i \leq k \wedge c = i\} \text{while } i < k \text{ do } (n := n + (y+z); i := i+1) \{i \not\leq k \wedge i \leq k \wedge c = i\}} \\
\frac{}{\{c = 0 \wedge i = 0 \wedge k \geq 0\} \text{while } i < k \text{ do } (n := n + (y+z); i := i+1); x := y+z \{c = k+1\}}
\end{array}$$

Figure 10: An original proof for resource usage

$$\begin{array}{c}
\frac{}{\{i \leq k \wedge c+1 \leq i+1\} t := y+z \{i \leq k \wedge c \leq i+1\}} \\
\frac{}{\{c \leq k+1\} x := t \{c \leq k+1\}} \\
\frac{}{\{i+1 \leq k \wedge c \leq (i+1)+1\} i := i+1 \{i \leq k \wedge c \leq i+1\}} \\
\frac{}{\{i+1 \leq k \wedge c \leq (i+1)+1\} n := n+t \{i+1 \leq k \wedge c \leq (i+1)+1\}} \\
\frac{}{\{i < k \wedge c \leq i+1\} n := n+t; i := i+1 \{i \leq k \wedge c \leq i+1\}} \\
\frac{}{\{i \leq k \wedge c \leq i+1\} \text{while } i < k \text{ do } (n := n + (y+z); i := i+1) \{i \not\leq k \wedge i \leq k \wedge c \leq i+1\}} \\
\frac{}{\{i \leq k \wedge c \leq i\} t := y+z; \text{while } i < k \text{ do } (n := n + (y+z); i := i+1) \{i = k \wedge c \leq i+1\}} \\
\frac{}{\{c \leq 0 \wedge i = 0 \wedge k \geq 0\} t := y+z; \text{while } i < k \text{ do } (n := n + t; i := i+1); x := t \{c \leq k+1\}}
\end{array}$$

Figure 11: The transformed proof for resource usage

not present in the modified loop. Figure 9 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 all expressions $a \in \mathbf{AExp}^+$. 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

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

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

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

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

Here $v(a)$ generates a new unique logic variable for every nontrivial arithmetic expression.

$P\|_{cpav}$ is a transformed version of an assertion P about a state of an original program to an assertion about a related state of the optimized program, guided by the type $cpav$. It is obtained from P by replacing the counters of all expressions by existentially quantified logic variables. These are constrained to have values not exceeding those of the counters by more than 1 or not exceeding them at all, depending on the type.

Assertion transformation defined, 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 7. Figure 10 demonstrates that in the instrumented Hoare logic we can prove that the program computes $y + z$ exactly $k + 1$ times (we have abbreviated $\lceil y + z \rceil$ to c). The invariant for the while-loop is $i \leq k \wedge c = i$. Figure 11 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; in particular, the loop invariant is now $i \leq k \wedge c = i + 1$. (In this proof, we have enhanced readability by replacing the existentially quantified assertions yielded by the automatic transformation with equivalent quantifier-free simplifications.)

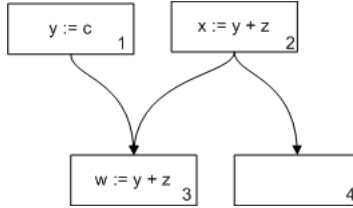
As expected, this formal counterpart of the semantic improvement argument is no smarter than the semantic improvement argument itself. In our semantic improvement statement we claimed that the optimized program performs not worse than possibly by one extra evaluation for precomputed expressions than 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 [23].

3. Full PRE

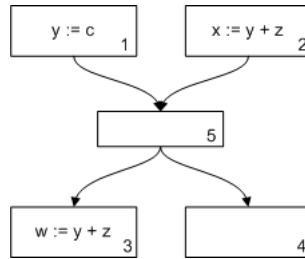
We now look at the formulation of full PRE by Paleri et al. [20]. As was explained in Section 2, simple PRE does not use all optimization opportunities.

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 unchanged by simple PRE, since $y + z$ is not anticipable at the exit of node 2. Full PRE would optimize the program by recording the value of $y + z$ in an auxiliary variable in node 2 and inserting an evaluation of $y + z$ into the edge leaving node 1. This would allow skipping the evaluation of $y + z$ in node 3.

This does not mean that it is possible to simply replace 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 inserted an evaluation of $y + z$ into the edge (1,5), we would potentially worsen the runtime behavior of the program, as 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 suffices instead of the total anticipability we used in simple PRE only if the path leading from the point where the expression becomes partially available (the exit of node 2 in the examples) to a point where the expression becomes partially anticipable (the entry of node 3 in the examples) contains no points 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 (i.e., 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 available or anticipable at that program point.

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

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

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

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., $\leq_{\text{df}} \supseteq$. The typing rules for availability are given in Figure 12.

$$\begin{array}{c}
\frac{}{x := a : av \longrightarrow av \cup eval(a) \setminus mod(x)} \quad \frac{}{skip : av \longrightarrow av} \quad \frac{s_0 : av \longrightarrow av'' \quad s_1 : av'' \longrightarrow av'}{s_0; s_1 : av \longrightarrow av'} \\
\frac{s_t : av \longrightarrow av' \quad s_f : av \longrightarrow av'}{\text{if } b \text{ then } s_t \text{ else } s_f : av \longrightarrow av'} \quad \frac{s_t : av \longrightarrow av}{\text{while } b \text{ do } s_t : av \longrightarrow av} \\
\frac{av \leq av_0 \quad s : av_0 \longrightarrow av'_0 \quad av'_0 \leq av'}{s : av \longrightarrow av'}
\end{array}$$

Figure 12: Type system for available expressions

$$\begin{array}{c}
\frac{}{x ::= a : (spant' \setminus mod(x) \cup eval(a)) \cap safe, spav \longrightarrow spant', (spav \cup eval(a) \setminus mod(x)) \cap safe'} \\
\frac{}{skip : spant, spav \longrightarrow spant, spav} \quad \frac{s_0 : spant, spav \longrightarrow spant'', spav'' \quad s_1 : spant'', spav'' \longrightarrow spant', spav'}{s_0; s_1 : spant, spav \longrightarrow spant', spav'} \\
\frac{s_t : spant, spav \longrightarrow spant', spav' \quad s_f : spant, spav \longrightarrow spant', spav'}{\text{if } b \text{ then } s_t \text{ else } s_f : spant, spav \longrightarrow spant', spav'} \\
\frac{s_t : spant, spav \longrightarrow spant, spav}{\text{while } b \text{ do } s_t : spant, spav \longrightarrow spant, spav} \\
\frac{spant, spav \leq spant_0, spav_0 \quad \underline{s} : spant_0, spav_0 \longrightarrow spant'_0, spav'_0 \quad spant'_0, spav'_0 \leq spant', spav'}{\underline{s} : spant, spav \longrightarrow spant', spav'}
\end{array}$$

Figure 13: Type system for the underlying analyses of full PRE

The type systems for safe partial availability and safe partial anticipability are given as a single type system in Figure 13. They do not depend on each other, but depend on the safety component. We use \underline{s} to denote a full type derivation of $s : ant, av \longrightarrow ant', av'$, thus safety *safe* in the pretype of \underline{s} is defined as $ant \cup av$, safety in the posttype *safe'* is $ant' \cup av'$.

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

Without the extra restrictions on types, the type system would still be sound, but it would lose its improvement property, i.e., it might allow optimizations which introduce unneeded evaluations. The restriction $ant \subseteq spant$ guarantees that the set of totally anticipable expressions cannot be bigger than the set of partially anticipable expressions. This is guaranteed by the principal type inference algorithm, but in the type system the subsumption rule could break this relation without the extra restriction. The same holds for full and partial availability. The restrictions $spant \subseteq ant \cup av$ and $spav \subseteq ant \cup av$ guarantee *safety* as in the original algorithm of Paleri et al.

The optimizing type system for full PRE is given in Figure 14. Insertion of auxiliary variable definitions at subsumption is now guided by the intersec-

$$\begin{array}{c}
\frac{a \notin spav \quad a \notin spant' \setminus mod(x)}{\underline{x := a : (spant' \setminus mod(x) \cup eval(a)) \cap safe, spav \longrightarrow spant', (spav \cup eval(a) \setminus mod(x)) \cap safe' \hookrightarrow x := a}} \\
\frac{a \notin spav \quad a \in spant' \setminus mod(x)}{\underline{x := a : (spant' \setminus mod(x) \cup eval(a)) \cap safe, spav \longrightarrow spant', (spav \cup eval(a) \setminus mod(x)) \cap safe' \hookrightarrow nv(a) := a; x := nv(a)}} \\
\frac{a \in spav}{\underline{x := a : (spant' \setminus mod(x) \cup eval(a)) \cap safe, spav \longrightarrow spant', (spav \cup eval(a) \setminus mod(x)) \cap safe' \hookrightarrow x := nv(a)}} \\
\frac{}{\underline{skip : spant, spav \longrightarrow spant, spav \hookrightarrow skip}} \\
\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{\text{if } b \text{ then } s_t \text{ else } s_f : spant, spav \longrightarrow spant', spav' \hookrightarrow \text{if } b \text{ then } s'_t \text{ else } s'_f}} \\
\frac{\underline{s_t : spant, spav \longrightarrow spant, spav \hookrightarrow s'_t}}{\underline{\text{while } b \text{ do } s_t : spant, spav \longrightarrow spant, spav \hookrightarrow \text{while } b \text{ 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 (spant_0 \cap spav_0) \setminus spav]; s'; [nv(a) := a \mid a \in (spant' \cap spav') \setminus spav'_0]}}
\end{array}$$

Figure 14: Type system for full PRE, with the optimization component

tion 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.

Theorem 5 (Soundness of full PRE). *If $\underline{s} : spant, spav \longrightarrow spant', spav' \hookrightarrow s_*$ and $\sigma \sim_{spant \cap spav} \sigma_*$, then*

- $\sigma \succ_s \rightarrow \sigma'$ implies the existence of σ'_* such that $\sigma' \sim_{spant' \cap spav'} \sigma'_*$ and $\sigma_* \succ_{s_*} \rightarrow \sigma'_*$,
- $\sigma_* \succ_{s_*} \rightarrow \sigma'_*$ implies the existence of σ' such that $\sigma' \sim_{spant' \cap spav'} \sigma'_*$ and $\sigma \succ_s \rightarrow \sigma'$.

The proof is quite similar to that for simple PRE. Full PRE is using partial anticipability instead of total anticipability, but this does not affect the soundness of the optimization. We only have to show that the use of *safety* can not affect soundness in a hazardous way.

PROOF. Again we only prove the first half of the theorem. The proof is by induction on the structure of the type derivation and we look at the same cases as for simple PRE.

- Case $:=_{pre}$: The type derivation is of the form

$$\underline{x := a : spant, spav \longrightarrow spant', spav' \hookrightarrow s_*}$$

where $spant =_{\text{df}} (spant' \setminus mod(x)) \cup eval(a) \cap safe$, $spav' =_{\text{df}} ((spav \cup eval(a)) \setminus mod(x)) \cap safe'$.

We notice that $safe \cup mod(x) \supseteq safe'$ since

$$\begin{aligned}
safe \cup mod(x) &= ant \cup av \cup mod(x) \\
&= (ant' \setminus mod(x)) \cup eval(a) \cup av \cup mod(x) \\
&\supseteq ant' \cup ((av \cup eval(a)) \setminus mod(x)) \\
&= ant' \cup av' \\
&= safe'
\end{aligned}$$

From this it follows that $(spant \cap spav) \cup eval(a) \supseteq spant' \cap spav'$:

$$\begin{aligned}
(spant \cap spav) \cup eval(a) &= (((spant' \setminus mod(x)) \cup eval(a)) \cap spav \cap safe) \cup eval(a) \\
&= (spant' \setminus mod(x) \cap spav \cap safe) \cup eval(a) \\
&\supseteq spant' \cap (spav \cup eval(a)) \setminus mod(x) \cap safe \\
&= spant' \cap (spav \cup eval(a)) \setminus mod(x) \cap (safe \cup mod(x)) \\
&\supseteq spant' \cap (spav \cup eval(a)) \setminus mod(x) \cap safe' \\
&= spant' \cap spav'
\end{aligned}$$

We also note it separately that $spant' \cap spav' \cap mod(x) = \emptyset$ (trivially, since $spav' \cap mod(x) = \emptyset$).

The given semantic judgement must be of the form

$$\overline{\sigma \succ x := a \rightarrow \sigma[x \mapsto \llbracket a \rrbracket \sigma]}$$

- Subcase $:=_{1\text{pre}}$: We know that $a \notin spav$. We also know that either $a \notin spant'$ or $a \in mod(x)$ (so $a \notin spav'$), so $a \notin spant' \cap spav'$, which implies $spant \cap spav \supseteq spant' \cap spav'$. Moreover, $s_* =_{\text{df}} x := a$.

We have the semantic derivation

$$\overline{\sigma_* \succ x := a \rightarrow \sigma'_*}$$

where $\sigma'_* =_{\text{df}} \sigma_*[x \mapsto \llbracket a \rrbracket \sigma_*]$. From $\sigma \sim_{spant \cap spav} \sigma_*$ it follows that $\llbracket a \rrbracket \sigma = \llbracket a \rrbracket \sigma_*$, so that, using $spant \cap spav \supseteq spant' \cap spav'$ as well, we can conclude $\sigma' = \sigma[x \mapsto \llbracket a \rrbracket \sigma] \sim_{spant' \cap spav'} \sigma_*[x \mapsto \llbracket a \rrbracket \sigma] = \sigma_*[x \mapsto \llbracket a \rrbracket \sigma_*] = \sigma'_*$.

- Subcase $:=_{2\text{pre}}$: We have that $a \notin spav$ and a is nontrivial. Also, $s_* =_{\text{df}} nv(a) := a; x := nv(a)$. We have the semantic derivation

$$\frac{\overline{\sigma_* \succ nv(a) := a \rightarrow \sigma''_*} \quad \overline{\sigma''_* \succ x := nv(a) \rightarrow \sigma'_*}}{\sigma_* \succ nv(a) := a; x := nv(a) \rightarrow \sigma'_*}$$

where $\sigma_*'' =_{\text{df}} \sigma_*[nv(a) \mapsto \llbracket a \rrbracket \sigma_*]$ and $\sigma_*' =_{\text{df}} \sigma_*''[x \mapsto \sigma_*''(nv(a))] = \sigma_*''[x \mapsto \llbracket a \rrbracket \sigma_*]$. From $\sigma \sim_{spant \cap spav} \sigma_*$ it is immediate that $\sigma \sim_{(spant \cap spav) \cup \{a\}} \sigma_*[nv(a) \mapsto \llbracket a \rrbracket \sigma] = \sigma_*[nv(a) \mapsto \llbracket a \rrbracket \sigma_*] = \sigma_*''$ and therefore by $(spant \cap spav) \cup \{a\} \supseteq spant' \cap spav'$ we have $\sigma' = \sigma[x \mapsto \llbracket a \rrbracket \sigma] \sim_{spant' \cap spav'} \sigma_*''[x \mapsto \llbracket a \rrbracket \sigma] = \sigma_*''[x \mapsto \llbracket a \rrbracket \sigma_*] = \sigma_*'$.

- Subcase $:=_{3\text{pre}}$: We have that $a \in spav$, but then a is nontrivial and $a \in safe$ (as $spav \subseteq safe$), so further $a \in ((spant' \setminus mod(x)) \cup eval(a)) \cap safe = spant$ as well, i.e., $a \in spant \cap spav$. As a consequence, $spant \cap spav \supseteq spant' \cap spav'$, too. We have $s_* =_{\text{df}} x := nv(a)$. We have the semantic derivation

$$\overline{\sigma_* \succ x := nv(a) \rightarrow \sigma_*'}$$

where $\sigma_*' =_{\text{df}} \sigma_*[x \mapsto \sigma_*(nv(a))]$. From $a \in spant \cap spav$ and $\sigma \sim_{spant \cap spav} \sigma_*$ we learn that $\llbracket a \rrbracket \sigma = \sigma_*(nv(a))$. Further, using also that $spant \cap spav \supseteq spant' \cap spav'$, we realize that $\sigma' = \sigma[x \mapsto \llbracket a \rrbracket \sigma] \sim_{spant' \cap spav'} \sigma_*[x \mapsto \llbracket a \rrbracket \sigma] = \sigma_*[x \mapsto \sigma_*(nv(a))] = \sigma_*'$.

- Case $\text{conseq}_{\text{pre}}$: The type derivation is of the form

$$\frac{\begin{array}{c} \vdots \\ \underline{s : spant_0, spav_0 \longrightarrow spant'_0, spav'_0 \hookrightarrow s_*} \end{array}}{s : spant, spav \longrightarrow spant', spav' \hookrightarrow s'; s_*; s''}$$

where $(spant, spav) \leq (spant_0, spav_0)$, $(spant'_0, spav'_0) \leq (spant', cpav')$, $s' =_{\text{df}} [nv(a) := a \mid a \in (spant_0 \cap spav_0) \setminus spav]$ and $s'' =_{\text{df}} [nv(a) := a \mid a \in (spant' \cap spav') \setminus spav'_0]$. First we find a state σ_0 such that $\sigma_* \succ s' \rightarrow \sigma_0$ and $\sigma \sim_{spant_0 \cap spav_0} \sigma_0$. We have the semantic derivation

$$\overline{\sigma_* \succ s' \rightarrow \sigma_0}$$

where $\sigma_0 =_{\text{df}} \sigma_*[nv(a) \mapsto \llbracket a \rrbracket \sigma_* \mid a \in (spant_0 \cap spav_0) \setminus spav]$.

As $\sigma \sim_{spant \cap spav} \sigma_*$, it is enough to show that expressions not in $spant \cap spav$ but in $spant_0 \cap spav_0$ are made equal to their corresponding auxiliary variables by s' . But this is exactly what s' does, as

$$\begin{aligned} & (spant_0 \cap spav_0) \setminus (spant \cap spav) \\ &= (spant_0 \cap spav_0 \setminus spant) \cup (spant_0 \cap spav_0 \setminus spav) \\ &= (spant_0 \cap spav_0) \setminus spav \end{aligned}$$

using $spant \supseteq spant_0$.

From the induction hypothesis we obtain that there is a state σ_1 such that $\sigma_0 \succ s_* \rightarrow \sigma_1$ and $\sigma' \sim_{spant'_0 \cap spav'_0} \sigma_1$. It is now enough to show that there is a state σ_*' such that $\sigma_1 \succ s'' \rightarrow \sigma_*'$ and $\sigma' \sim_{spav' \cap spav'} \sigma_*'$. This can be done similarly to the case of s' . \square

For transformation of Hoare logic proofs of functional correctness, let $P|_{spant \cap spav}$ abbreviate $\bigwedge[nv(a) = a \mid a \in spant \cap spav] \wedge P$. We have the following theorem.

Theorem 6 (Preservation of Hoare logic provability/proofs for full PRE).

If $\underline{s} : spant, spav \longrightarrow spant', spav' \hookrightarrow s_*$, then
 $\text{---}\{P\} s \{Q\}$ implies $\{P|_{spant \cap spav}\} s_* \{Q|_{spant' \cap spav'}\}$.

PROOF. The proof is by induction on the type derivation and we look at the same cases as before.

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

$$\overline{x := a : spant, spav \longrightarrow spant', spav' \hookrightarrow s_*}$$

where $spant =_{df} (spant' \setminus mod(x) \cup eval(a)) \cap safe$, $spav' =_{df} (spav \cup eval(a) \setminus mod(x)) \cap safe'$.

From the soundness proof (Theorem 5) we remember that $(spant \cap spav) \cup eval(a) \supseteq spant' \cap spav'$ and $spant' \cap spav' \cap mod(x) = \emptyset$. The latter fact tells us that for any a' and P , we have $P[a'/x]|_{spant' \cap spav'} \Leftrightarrow P|_{spant' \cap spav'}[a'/x]$.

The given Hoare logic proof must be of the form

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

- Subcase $:=_{1pre}$: We have that $a \notin spav$. We also have that either $a \notin spant'$ or $a \in mod(x)$ (so $a \notin spav'$), so altogether $a \notin spant' \cap spav'$, which implies $spant \cap spav \supseteq spant' \cap spav'$. Moreover, $s_* =_{df} x := a$. From $spant \cap spav \supseteq spant' \cap spav'$, it follows that $P[a/x]|_{spant \cap spav} \models P[a/x]|_{spant' \cap spav'}$. The transformed Hoare logic proof is

$$\frac{\frac{\overline{\{P|_{spant' \cap spav'}[a/x]\} x := a \{P|_{spant' \cap spav'}\}}{\overline{\{P[a/x]|_{spant' \cap spav'}\} x := a \{P|_{spant' \cap spav'}\}}}{\overline{\{P[a/x]|_{spant \cap spav}\} x := a \{P|_{spant' \cap spav'}\}}}$$

- Subcase $:=_{2pre}$: We have that $a \notin spav$. We also have that a is nontrivial. And $s_* =_{df} nv(a) := a; x := nv(a)$.

From $(spant \cap spav) \cup \{a\} \supseteq spant' \cap spav'$, it follows that $P[nv(a)/x]|_{(spant \cap spav) \cup \{a\}} \models P[nv(a)/x]|_{spant' \cap spav'}$. From reflexivity of equality, $P[a/x]|_{spant \cap spav} \Leftrightarrow P[a/x]|_{spant \cap spav} \wedge a = a \Leftrightarrow (P[nv(a)/x]|_{(spant \cap spav) \cup \{a\}})[a/nv(a)]$.

The transformed Hoare logic proof is

$$\frac{\overline{B_0 \quad B_1}}{\overline{\{P[a/x]|_{spant \cap spav}\} nv(a) = a; x := nv(a) \{P|_{spant' \cap spav'}\}}}$$

where $B_0 \equiv$

$$\frac{\overline{\{P[nv(a)/x]|_{(spant \cap spav) \cup \{a\}}[a/nv(a)]\} nv(a) = a \{P[nv(a)/x]|_{(spant \cap spav) \cup \{a\}}\}}}{\overline{\{P[a/x]|_{spant \cap spav}\} nv(a) = a \{P[nv(a)/x]|_{spant' \cap spav'}\}}}$$

and $B_1 \equiv$

$$\frac{\overline{\{P|_{spant' \cap spav'}[nv(a)/x]\} x := nv(a) \{P|_{spant' \cap spav'}\}}}{\{P[nv(a)/x]|_{spant' \cap spav'}\} x := nv(a) \{P|_{spant' \cap spav'}\}}$$

- Subcase $:=_{3\text{pre}}$: We have that $a \in spav$. We also have that $a \in spant$ (as $a \in spav \subseteq safe$ and a is nontrivial). This has $a \in spant \cap spav$ as a consequence and therefore we also get $spant \cap spav \supseteq spant' \cap spav'$. Further, $s_* =_{\text{df}} x := nv(a)$.

From $a \in spant \cap spav$ and $spant \cap spav \supseteq spant' \cap spav'$ we get $P[a/x]|_{spant \cap spav} \models P[a/x]|_{spant' \cap spav'} \wedge nv(a) = a$. Substitution of equals for equals gives $P|_{spant' \cap spav'}[a/x] \wedge nv(a) = a \models P|_{spant' \cap spav'}[nv(a)/x]$. The transformed Hoare logic proof is

$$\frac{\frac{\overline{\{P|_{spant' \cap spav'}[nv(a)/x]\} x := nv(a) \{P|_{spant' \cap spav'}\}}}{\{P|_{spant' \cap spav'}[a/x] \wedge nv(a) = a\} x := nv(a) \{P|_{spant' \cap spav'}\}}}{\{P[a/x]|_{spant' \cap spav'} \wedge nv(a) = a\} x := nv(a) \{P|_{spant' \cap spav'}\}} \frac{}{\{P[a/x]|_{spant \cap spav}\} x := nv(a) \{P|_{spant' \cap spav'}\}}$$

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

$$\frac{\begin{array}{c} \vdots \\ s : spant_0, spav_0 \longrightarrow spant'_0, spav'_0 \hookrightarrow s_* \end{array}}{s : spant, spav \longrightarrow spant', spav' \hookrightarrow s'; s_*; s''}$$

where $(spant, spav) \leq (spant_0, spav_0)$, $(spant'_0, spav'_0) \leq (spant, spav)$ and $s' =_{\text{df}} [nv(a) := a \mid a \in (spant_0 \cap spav_0) \setminus spav]$, $s'' =_{\text{df}} [nv(a) := a \mid a \in (spant' \cap spav') \setminus spav'_0]$. The given Hoare logic proof is

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

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

By the induction hypothesis, there is a Hoare logic proof of $\{P_0|_{spant_0 \cap spav_0}\} s_* \{Q_0|_{spant'_0 \cap spav'_0}\}$.

It is an assumption that $P \models P_0$, hence $P|_{spant \cap spav} \models P_0|_{spant \cap spav}$. Using reflexivity of equality we get $P_0|_{spant \cap spav} \Leftrightarrow P_0|_{spant \cap spav} \wedge \bigwedge [a = a \mid a \in (spant_0 \cap spav_0) \setminus (spant \cap spav)] \models P_0|_{spant_0 \cap spav_0}[a/nv(a) \mid a \in (spav_0 \cap spant_0) \setminus (spant \cap spav)]$.

From the soundness proof (Theorem5) we remember that $(spant_0 \cap spav_0) \setminus (spant \cap spav) = spant_0 \cap spav_0 \setminus spav$. Hence from the axiom $\{P_0|_{spant_0 \cap spav_0}[a/nv(a) \mid a \in (spant_0 \cap spav_0) \setminus spav]\} s' \{P_0|_{spant_0 \cap spav_0}\}$ by the consequence rule we have a proof of $\{P|_{spant \cap spav}\} s' \{P_0|_{spant_0 \cap spav_0}\}$.

Similarly we can make a proof of $\{Q_0|_{spant'_0 \cap spav'_0}\} s'' \{Q|_{spant' \cap spav'}\}$.

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

The similarity relation between the states for showing the improvement property is more involved than in the case of simple PRE. We define $(\sigma, r) \approx_{spant \cap spav, av} (\sigma_*, r_*)$ to mean that two states (σ, r) and (σ_*, r_*) are similar wrt. $spant \cap spav$ and av in the sense that $\sigma \sim_{spant \cap spav} \sigma_*$ and, moreover, $\forall a \in spant \cap spav \setminus av. r_*(a) \leq r(a) + 1$ and $\forall a \notin spant \cap spav \setminus av. r_*(a) \leq r(a)$.

For simple PRE, an expression being in a type $cpav$ meant a “promise” that there will be a use of the expression, because $cpav$ implied anticipability. For full PRE this is not the case, since $spant \cap spav$ does not imply that there is a future use of the expression. This is where the notion of *safety* comes into play. Since $spant \cap spav \subseteq safe$ where $safe = ant \cup av$, then if $a \in spant \cap spav$, either $a \in ant$ or $a \in av$. Notice that $a \in spant \cap spav \setminus av$ implies $a \in ant$ which means that we allow an increase in the number of expression evaluations only at places where the expression is anticipable. The reason why there cannot be an increase of evaluations of a where $a \in av$ is simply that availability implies that the expression is already computed on all paths to that program point, so no extra evaluation could have been added. This is the reason why we need to parameterize the similarity relation with av . A simple example explaining this is the program $x := a; \text{if } b \text{ then } y := a \text{ else skip}$, which can be optimized to $nv(a) = a; x := nv(a); \text{if } b \text{ then } y := nv(a) \text{ else skip}$. After the first statement, a is clearly both safely partially available and safely partially anticipable. But the count of evaluations of a should not be one larger for the optimized program than for the original one, hence we need to take into account that a is available. Consequently we can guarantee that the overall number of evaluations of a for the optimized program is no bigger than that for the original one.

Theorem 7 (Improvement property of full PRE). *If $\underline{s} : spant, spav \longrightarrow spant', spav' \hookrightarrow s_*$ and $(\sigma, r) \approx_{(spant \cap spav, av)} (\sigma_*, r_*)$, then*

- $(\sigma, r) \succ_{s \rightarrow} (\sigma', r')$ *implies the existence of (σ'_*, r'_*) such that $(\sigma', r') \approx_{(spant' \cap spav', av')} (\sigma'_*, r'_*)$ and $(\sigma_*, r_*) \succ_{s_* \rightarrow} (\sigma'_*, r'_*)$,*
- $(\sigma_*, r_*) \succ_{s_* \rightarrow} (\sigma'_*, r'_*)$ *implies the existence of (σ', r') such that $(\sigma', r') \approx_{(spant' \cap spav', av')} (\sigma'_*, r'_*)$ and $(\sigma, r) \succ_{s \rightarrow} (\sigma', r')$.*

PROOF. We only prove the first half of theorem. The proof is by induction on the structure of the type derivation. We look at the following nontrivial cases.

- Case $:=_{pre}$: The type derivation is of the form

$$\frac{}{x := a : spant, spav \longrightarrow spant', spav' \hookrightarrow s_*}$$

where $spant \stackrel{\text{df}}{=} ((spant' \setminus mod(x)) \cup eval(a)) \cap safe$, $spav' \stackrel{\text{df}}{=} (spav \cup eval(a)) \setminus mod(x) \cap safe'$.

We notice that $safe \subseteq safe' \cup mod(x)$ since

$$\begin{aligned}
safe &= ant \cup av \\
&= (ant' \setminus mod(x)) \cup eval(a) \cup av \\
&\subseteq ant' \cup ((av \cup eval(a)) \setminus mod(x)) \cup mod(x) \\
&= ant' \cup av' \cup mod(x) \\
&= safe' \cup mod(x)
\end{aligned}$$

This gives us that $spant \cap spav \setminus av \subseteq (spant' \cap spav' \setminus av') \cup eval(a)$:

$$\begin{aligned}
spant \cap spav \setminus av &= ((spant' \setminus mod(x)) \cup eval(a)) \cap safe \cap spav \setminus av \\
&\subseteq (spant' \cap spav \setminus mod(x) \cap safe \setminus av) \cup eval(a) \\
&\subseteq (spant' \cap spav \setminus mod(x) \cap (safe' \cup mod(x)) \setminus av) \cup eval(a) \\
&= (spant' \cap spav \setminus mod(x) \cap safe' \setminus av) \cup eval(a) \\
&= (spant' \cap (spav \cup eval(a)) \setminus mod(x) \cap safe' \setminus (av \cup eval(a))) \cup eval(a) \\
&= (spant' \cap (spav \cup eval(a)) \setminus mod(x) \cap safe' \setminus ((av \cup eval(a)) \setminus mod(x))) \cup eval(a) \\
&= (spant' \cap spav' \setminus av') \cup eval(a)
\end{aligned}$$

From the proof of soundness (Theorem 5) we remember that $(spant \cap spav) \cup eval(a) \supseteq spant' \cap spav'$. We also recall that $spant' \cap spav' \cap mod(x) = \emptyset$. This yields that $(spant \cap spav \setminus av) \cup eval(a) \supseteq ((spant \cap spav) \cup eval(a)) \setminus av \supseteq (spant' \cap spav') \setminus av \supseteq spant' \cap spav' \setminus ((av \cup eval(a)) \setminus mod(x)) = spant' \cap spav' \setminus av'$.

So for any nontrivial expression $a' \neq a$, $a' \in spant \cap spav \setminus av$ and $a' \in spant' \cap spav' \setminus av'$ are equivalent.

The given semantic derivation must be of the form

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

- Subcase $:=_{1\text{pre}}$: We have that $a \notin spav$, which implies that $a \notin spant \cap spav \setminus av$, and either $a \notin spant'$ or $a \in mod(x)$ (so $a \notin spav'$), which implies that $a \notin spant' \cap spav' \setminus av'$. Moreover, $s_* =_{\text{df}} x := a$. We have the semantic derivation

$$\overline{(\sigma_*, r_*) \succ x := a \rightarrow (\sigma'_*, r'_*)} .$$

where $(\sigma'_*, r'_*) =_{\text{df}} (\sigma_*[x \mapsto \llbracket a \rrbracket \sigma_*], r_*[a \mapsto r_*(a) + 1])$. From the assumption we know that $r_*(a) \leq r(a)$, so $r'_*(a) = r_*(a) + 1 \leq r(a) + 1 = r'(a)$.

- Subcase $:=_{2\text{pre}}$: We have that $a \notin spav$, implying $a \notin spant \cap spav \setminus av$, and both $a \in spant'$ and $a \notin mod(x)$, hence $a \in av'$, implying

$a \notin \text{spant}' \cap \text{spav}' \setminus \text{av}'$. Moreover, $s_* =_{\text{df}} \text{nv}(a) := a; x := \text{nv}(a)$. We have the semantic derivation

$$\frac{(\sigma_*, r_*) \succ \text{nv}(a) := a \rightarrow \sigma''_*, r''_* \quad \overline{(\sigma''_*, r''_*) \succ x := \text{nv}(a) \rightarrow (\sigma'_*, r'_*)}}{(\sigma_*, r_*) \succ \text{nv}(a) := a; x := \text{nv}(a) \rightarrow (\sigma'_*, r'_*)} .$$

where $(\sigma''_*, r''_*) =_{\text{df}} (\sigma_*[\text{nv}(a) \mapsto \llbracket a \rrbracket \sigma_*], r_*[a \mapsto r_*(a)+1])$ and $(\sigma'_*, r'_*) =_{\text{df}} (\sigma_*[x \mapsto \sigma''_*(\text{nv}(a))], r''_*)$. Given these circumstances, from $r_*(a) \leq r(a)$ we obtain $r'_*(a) = r''_*(a) = r_*(a) + 1 \leq r(a) + 1 = r'(a)$.

- Subcase $:=_{3\text{pre}}$: We have that $a \in \text{spav}$, which has as a consequence that $a \in \text{spant}$ (as a is nontrivial and $a \in \text{spav} \subseteq \text{safe}$), but we do not know whether $a \notin \text{av}$. Hence we do not know whether $a \in \text{spant} \cap \text{spav} \setminus \text{av}$. However, since $a \in \text{spav}'$ if and only if $a \in \text{av}'$ (both hold if $a \notin \text{mod}(x)$ and neither holds if $a \in \text{mod}(x)$), we know that $a \notin \text{spant}' \cap \text{spav}' \setminus \text{av}'$. Moreover, $s_* =_{\text{df}} x := \text{nv}(a)$. We have the semantic derivation

$$\overline{(\sigma_*, r_*) \succ x := \text{nv}(a) \rightarrow (\sigma'_*, r'_*)} .$$

where $(\sigma'_*, r'_*) =_{\text{df}} (\sigma_*[x \mapsto \sigma_*(\text{nv}(a))], r_*)$. From $r_*(a) \leq r(a) + 1$ (which holds both if $a \notin \text{spant} \cap \text{spav} \setminus \text{av}$ and if $a \in \text{spant} \cap \text{spav} \setminus \text{av}$) we get that $r'_*(a) = r_*(a) \leq r(a) + 1 = r'(a)$.

In all three subcases, for any nontrivial $a' \neq a$, if $a' \in \text{spant} \cap \text{spav} \setminus \text{av}$, then we have that $a' \in \text{spant}' \cap \text{spav}' \setminus \text{av}'$ as well and therefore from $r_*(a') \leq r(a') + 1$ we get $r'_*(a') = r_*(a') \leq r(a') + 1 = r'(a') + 1$. Similarly, for $a' \neq a$ such that $a' \notin \text{spant} \cap \text{spav} \setminus \text{av}$ we have $a' \notin \text{spant}' \cap \text{spav}' \setminus \text{av}'$, so $r_*(a') \leq r(a')$ gives us $r'_*(a') = r_*(a') \leq r(a') = r'(a')$.

- Case $\text{conseq}_{\text{pre}}$: The type derivation is of the form

$$\frac{\begin{array}{c} \vdots \\ \underline{s : \text{spant}_0, \text{spav}_0 \longrightarrow \text{spant}'_0, \text{spav}'_0 \hookrightarrow s_*} \end{array}}{\underline{s : \text{spant}, \text{spav} \longrightarrow \text{spant}', \text{spav}' \hookrightarrow s'; s_*; s''}}$$

where $(\text{ant}, \text{av}) \leq (\text{ant}_0, \text{av}_0)$, $(\text{spant}, \text{spav}) \leq (\text{spant}_0, \text{spav}_0)$, $(\text{ant}'_0, \text{av}'_0) \leq (\text{ant}', \text{av}')$, $(\text{spant}'_0, \text{spav}'_0) \leq (\text{spant}', \text{spav}')$ and $s' =_{\text{df}} [\text{nv}(a) := a \mid a \in (\text{spant}_0 \cap \text{spav}_0) \setminus \text{spav}]$, $s'' =_{\text{df}} [\text{nv}(a) := a \mid a \in (\text{spant}' \cap \text{spav}') \setminus \text{spav}'_0]$. First we find a state (σ_0, r_0) such that $(\sigma_*, r_*) \succ s' \rightarrow (\sigma_0, r_0)$ and $(\sigma, r) \approx_{\text{spant}_0 \cap \text{spav}_0, \text{av}_0} (\sigma_0, r_0)$. We have the semantic derivation

$$\overline{(\sigma_*, r_*) \succ s' \rightarrow (\sigma_0, r_0)} .$$

where $(\sigma_0, r_0) =_{\text{df}} (\sigma_*[\text{nv}(a) \mapsto \llbracket a \rrbracket \sigma_* \mid a \in (\text{spant}_0 \cap \text{spav}_0) \setminus \text{spav}], r_*[a \mapsto r_*(a)+1 \mid a \in (\text{spant}_0 \cap \text{spav}_0) \setminus \text{spav}])$ and we know that $(\sigma, r) \approx_{\text{spant} \cap \text{spav}, \text{av}} (\sigma_*, r_*)$.

For any expression $a \in spav \setminus av$, from $spav \setminus av \subseteq spav \setminus av_0 \subseteq safe \setminus av_0 \subseteq ant_0 \subseteq spant_0$, $spav \subseteq spav_0$ and $av \supseteq av_0$, we have $a \in spant_0 \cap spav_0 \setminus av_0$, whereas from $r_*(a) \leq r(a) + 1$ (which is necessarily guaranteed) we can conclude $r_0(a) = r_*(a) \leq r(a) + 1$. For any expression $a \in spav \cap av$, we have $a \notin spant \cap spav \setminus av$, so from $r_*(a) \leq r(a)$ we can conclude $r_0(a) = r_*(a) \leq r(a)$ (which is sufficient to ensure).

For any nontrivial expression $a \notin spant_0 \cap spav_0$, it is obvious that $a \notin spant_0 \cap spav_0 \setminus av_0$ and from $spav \setminus av \subseteq spav \setminus av_0 \subseteq safe \setminus av_0 \subseteq ant_0 \subseteq spant_0$ and $spav \subseteq spav_0$ we learn that $a \notin spant \cap spav \setminus av$. In this situation $r_*(a) \leq r(a)$ tells us that $r_0(a) = r_*(a) \leq r(a)$.

If an expression a is in $(spant_0 \cap spav_0) \setminus spav$, then $a \notin spant \cap spav \setminus av$ and $a \in spant_0 \cap spav_0 \setminus av_0$ (as $a \notin spav \supseteq av \supseteq av_0$), thus from $r_*(a) \leq r(a)$ we can conclude $r_0(a) = r_*(a) + 1 \leq r(a) + 1$.

By the induction hypothesis, there must exist a state (σ_1, r_1) such that $(\sigma_0, r_0) \succ_{s_*} (\sigma_1, r_1)$ and $(\sigma', r') \approx_{spant'_0 \cap spav'_0, av'_0} (\sigma_1, r_1)$. It is now enough to exhibit a state (σ'_*, r'_*) such that $(\sigma_1, r_1) \succ_{s''} (\sigma'_*, r'_*)$ and $(\sigma', r') \approx_{spant' \cap spav', av' } (\sigma'_*, r'_*)$. This can be done in the same way as for s' . \square

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 Riis Nielson and Nielson’s work on “flow logics” [21] (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 [5] it appears for structured languages together with the relational method of stating and proving dataflow analyses and optimizations sound. Lerner et al. [17, 18] 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. [4, 3]. Differently from this paper (and other works of ours), 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 [3] 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. [25] and Chaieb [7] 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. [2]. The same group of authors has also studied certification of resource consumption in a VDM-style program logic [1].

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 [12] (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 [23, 24, 22], we promoted the type-systematic method for describing analyses and optimizations, by stating and proving type systems sound for dead-code elimination and common subexpression elimination 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 Hoare logics. In another piece of work [13], 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.

To the best of our knowledge, we are the first authors to employ type systems to describe and reason about program optimizations and our method of obtaining automatic transformations of general Hoare proofs (as opposed to wp proofs) guided by certificates of general valid analyses (as opposed to strongest analyses) is entirely novel. In fact we are not aware of any other work dedicated to automatic transformation of Hoare logic proofs.

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 [19] used a bidirectional analysis and so did many subsequent versions [10, 8] addressing its shortcomings. The formulations in the innovative work of Knoop et al. [15, 16] and the subsequent new wave of papers [11, 9, 6] are based on cascades of unidirectional analyses. The best motivated formulations today are those of Paleri et al. [20] and Xue and Knoop [26]. 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 performs edge splitting to place moved expression evaluations; type-systematically, this corresponds to new assignments appearing at subsumption inferences. To our knowledge, this is the first type-systematic description of partial redundancy elimination.

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.

Also, minimized number of expression evaluations is not the only measure for the quality of a PRE-like transformation. There are other properties of PRE that we did not prove, e.g., that PRE does not introduce dead assignments to the auxiliary variables.

- Type-systematic optimizations vs. type-indexed similarity relations used in semantic statements of soundness and improvement: In this work, the similarity relations used in semantic statements of optimization 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 [17, 18] 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 and the EU FP6 IST integrated project no. 15905 MOBIUS. The first author received further support from the Estonian Doctoral School in ICT, the EITSA Tiger University Plus programme and the Estonian Association of Information Technology and Telecommunications (ITL).

References

- [1] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, A. Momigliano, A program logic for resources, *Theor. Comput. Sci.* 389(3) (2007) 411–445.
- [2] D. Aspinall, L. Beringer, A. Momigliano, Optimisation validation, in: J. Knoop, G. C. Necula, W. Zimmermann (Eds.), *Proc. 5th Int. Workshop on Compiler Optimization Meets Compiler Verification, COCV '06* (Vienna, Apr. 2006), *Electron. Notes in Theor. Comput. Sci.*, vol. 176(3), Elsevier, 2007, pp. 37–59,
- [3] 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), *Lect. Notes in Comput. Sci.*, vol. 4134, Springer, 2006, pp. 301–317.
- [4] 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. Workshop on Formal Aspects in Security and Trust, FAST 2005* (Newcastle upon Tyne, July 2005), *Lect. Notes in Comput. Sci.*, vol. 3866, Springer, 2006, pp. 112–126.
- [5] 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), ACM Press, 2004, pp. 14–25.
- [6] D. Bronnikov, A practical adoption of partial redundancy elimination, *ACM SIGPLAN Notices* 39(8) (2004) 49–53.

- [7] 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), Lect. Notes in Comput. Sci., vol. 4281, Springer, 2006, pp. 287–301.
- [8] D. M. Dhamdhere, Practical adaptation of the global optimization algorithm of Morel and Renvoise, ACM Trans. on Program. Lang. and Syst. 13(2) (1991) 291–294.
- [9] D. M. Dhamdhere, E-path_PRE—partial redundancy elimination made easy, ACM SIGPLAN Notices 37(8) (2002) 53–65.
- [10] 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. 10(4) (1988) 635–640.
- [11] K. H. Drechsler, M. P. Stadel, A variation of Knoop, R uthing and Steffen’s “Lazy code motion”, ACM SIGPLAN Notices 28(5) (1993) 29–38.
- [12] R. W. Floyd, Assigning meanings to programs, in: J. T. Schwartz (Ed.), Mathematical Aspects of Computer Science, Proc. of Symp. in Appl. Math., vol. 19, Amer. Math. Soc., 1967, pp. 19–33.
- [13] M. J. Frade, A. Saabas, T. Uustalu, Foundational certification of data-flow analyses, in: Proc. of 1st IEEE and IFIP Int. Symp. on Theor. Aspects of Software Engineering, TASE 2007 (Shanghai, June 2007), IEEE CS Press, 2007, pp. 107–116.
- [14] C. A. R. Hoare, An axiomatic basis for computer programming, Commun. of ACM 12(10) (1969) 576–583.
- [15] J. Knoop, O. R uthing, B. Steffen, Lazy code motion, in: Proc. of ACM SIGPLAN 2002 Conf. on Programming Language Design and Implementation, PLDI ’92 (San Francisco, CA, June 1992), ACM Press, 1992, pp. 224–234.
- [16] J. Knoop, O. R uthing, B. Steffen, Optimal code motion: theory and practice, ACM Trans. on Program. Lang. and Syst. 16(4) (1994) 1117–1155.
- [17] S. Lerner, T. Millstein, C. Chambers, Automatically proving the correctness of compiler optimizations, in: Proc. of ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation, PLDI ’03 (San Diego, CA, June 2003), ACM Press, 2003, pp. 220–231.
- [18] 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), ACM Press, 2005, pp. 364–377.
- [19] E. Morel, C. Renvoise, Global optimization by suppression of partial redundancies, Commun. of ACM 22(2) (1979) 96–103.

- [20] V. K. Paleri, Y. N. Srikant, P. Shankar, Partial redundancy elimination: a simple, pragmatic, and provably correct algorithm, *Sci. of Comput. Program.* 48(1) (2003) 1–20.
- [21] H. Riis 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*, *Lect. Notes in Comput. Sci.*, vol. 2566, Springer, 2002, pp. 223–244.
- [22] A. Saabas, *Logics for low-level code and proof-preserving program transformations* (Ph.D. thesis), Thesis on Informatics and System Engineering C43, Tallinn Univ. of Techn., 2008.
- [23] A. Saabas, T. Uustalu, Program and proof optimizations with type systems, *J. of Log. and Algebr. Program.* 77(1–2) (2008) 131–154.
- [24] A. Saabas, T. Uustalu. Type systems for optimizing stack-based code, in: M. Huisman, F. Spoto (Eds.), *Proc. of 2nd Workshop on Bytecode Semantics, Verification, Analysis and Transformation, Bytecode 2007* (Braga, March 2007), *Electron. Notes in Theor. Comput. Sci.*, vol. 190(1), Elsevier, 2007, pp. 103–119.
- [25] 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), *Lect. Notes in Comput. Sci.*, v. 2895, Springer, 2003, pp. 230–245.
- [26] 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.*, vol. 3923, Springer, 2006, pp. 139–154.

A. The high-level language While

This section is a summary of the syntax, natural semantics and the standard Hoare logic of the basic high-level language WHILE [14].

A.1. Syntax

The syntax proceeds from a countable supply of arithmetic variables $x \in \mathbf{Var}$. Over these, three syntactic categories of arithmetic expressions $a \in \mathbf{AExp}$, boolean expressions $b \in \mathbf{BExp}$ and statements $s \in \mathbf{Stm}$ are defined by means of the grammar

$$\begin{aligned}
 a & ::= x \mid n \mid a_0 + a_1 \mid \dots \\
 b & ::= a_0 = a_1 \mid \dots \mid \text{tt} \mid \text{ff} \mid \neg b \mid \dots \\
 s & ::= x := a \mid \text{skip} \mid s_0; s_1 \mid \text{if } b \text{ then } s_t \text{ else } s_f \mid \text{while } b \text{ do } s_t
 \end{aligned}$$

$$\begin{array}{c}
\frac{}{\sigma \succ x := a \rightarrow \sigma[x \mapsto \llbracket a \rrbracket \sigma]} :=_{\text{ns}} \\
\frac{}{\sigma \succ \text{skip} \rightarrow \sigma} \text{skip}_{\text{ns}} \quad \frac{\sigma \succ s_0 \rightarrow \sigma'' \quad \sigma'' \succ s_1 \rightarrow \sigma'}{\sigma \succ s_0; s_1 \rightarrow \sigma'} \text{comp}_{\text{ns}} \\
\frac{\sigma \models b \quad \sigma \succ s_t \rightarrow \sigma'}{\sigma \succ \text{if } b \text{ then } s_t \text{ else } s_f \rightarrow \sigma'} \text{if}_{\text{ns}}^{\text{tt}} \quad \frac{\sigma \not\models b \quad \sigma \succ s_f \rightarrow \sigma'}{\sigma \succ \text{if } b \text{ then } s_t \text{ else } s_f \rightarrow \sigma'} \text{if}_{\text{ns}}^{\text{ff}} \\
\frac{\sigma \models b \quad \sigma \succ s_t \rightarrow \sigma'' \quad \sigma'' \succ \text{while } b \text{ do } s_t \rightarrow \sigma'}{\sigma \succ \text{while } b \text{ do } s_t \rightarrow \sigma'} \text{while}_{\text{ns}}^{\text{tt}} \quad \frac{\sigma \not\models b}{\sigma \succ \text{while } b \text{ do } s_t \rightarrow \sigma} \text{while}_{\text{ns}}^{\text{ff}}
\end{array}$$

Figure 15: Natural semantics rules of WHILE

We denote the set of non-trivial (i.e., non-variable, non-numeral) arithmetic expressions by \mathbf{AExp}^+ .

A.2. Natural semantics

The semantics is given in terms of states. The states are defined as stores $\sigma \in \mathbf{Store}$, i.e., mappings of variables to integers: $\mathbf{State} =_{\text{df}} \mathbf{Store} =_{\text{df}} \mathbf{Var} \rightarrow \mathbb{Z}$. The arithmetical and boolean expressions are interpreted relative to stores as integers and truth values by the semantic function $\llbracket - \rrbracket \in \mathbf{AExp} + \mathbf{BExp} \rightarrow \mathbf{Store} \rightarrow \mathbb{Z}$, defined in the denotational style by the usual equations. We write $\sigma \models b$ to say that $\llbracket b \rrbracket \sigma = \text{tt}$.

Statements are interpreted via the evaluation relation $\succ - \rightarrow \subseteq \mathbf{State} \times \mathbf{Stm} \times \mathbf{State}$ defined inductively by the ruleset given in Figure 15.

Lemma 8 (Determinacy). *If $\sigma \succ s \rightarrow \sigma'$ and $\sigma \succ s \rightarrow \sigma''$, then $\sigma' = \sigma''$.*

A.3. Hoare logic

The assertions $P \in \mathbf{Assn}$ are defined as formulae of an unspecified underlying logic over a signature consisting of (a) constants for integers and function and predicate symbols for the standard integer-arithmetical operations and relations and (b) the program variables $x \in \mathbf{Var}$ as constants. For the completeness result, the language is assumed to be expressive enough to allow the expression of the weakest liberal precondition of any statement wrt. any given postcondition. We write $\sigma \models_{\alpha} P$ to express that P holds in the structure on \mathbb{Z} determined by (a) the standard meanings of the arithmetical constants, function and predicate symbols and (b) a state σ , under an assignment α of the logic variables. The writing $P \models Q$ means that $\sigma \models_{\alpha} P$ implies $\sigma \models_{\alpha} Q$ for any σ, α .

The derivable judgements of the logic are given by the relation $\{ \} - \{ \} \subseteq \mathbf{Assn} \times \mathbf{Stm} \times \mathbf{Assn}$ defined inductively by the ruleset in Figure 16. In the consequence rule, we rely on semantic entailment rather than derivability in some proof system for arithmetic, to circumvent the incompleteness of any such system.

Theorem 9 (Soundness). *If $\{P\} s \{Q\}$, then, for any σ, σ' and α , $\sigma \models_{\alpha} P$ and $\sigma \succ s \rightarrow \sigma'$ imply $\sigma' \models_{\alpha} Q$.*

$$\begin{array}{c}
\overline{\{Q[a/x]\} x := a \{Q\}} \quad ::=_{\text{hoa}} \\
\frac{\overline{\{P\} \text{skip } \{P\}}}{\text{skip}_{\text{hoa}}} \quad \frac{\{P\} s_0 \{R\} \quad \{R\} s_1 \{Q\}}{\{P\} s_0; s_1 \{Q\}} \quad \text{comp}_{\text{hoa}} \\
\frac{\{b \wedge P\} s_t \{Q\} \quad \{\neg b \wedge P\} s_f \{Q\}}{\{P\} \text{if } b \text{ then } s_t \text{ else } s_f \{Q\}} \quad \text{if}_{\text{hoa}} \quad \frac{\{b \wedge P\} s_t \{P\}}{\{P\} \text{while } b \text{ do } s_t \{ \neg b \wedge P \}} \quad \text{while}_{\text{hoa}} \\
\frac{P \models P' \quad \{P'\} s \{Q'\} \quad Q' \models Q}{\{P\} s \{Q\}} \quad \text{conseq}_{\text{hoa}}
\end{array}$$

Figure 16: Hoare rules of WHILE

Theorem 10 (Completeness). *If, for any σ, σ' and α , $\sigma \models_{\alpha} P$ and $\sigma \succ\text{-}s\text{-}\sigma'$ imply $\sigma' \models_{\alpha} Q$, then $\{P\} s \{Q\}$.*