Program and Proof Optimizations with Type Systems [1]

Ando Saabas and Tarmo Uustalu *

Institute of Cybernetics at Tallinn University of Technology, Akadeemia tee 21, EE-12618 Tallinn, Estonia

Received 17 March 2007; accepted 9 January 2008

Abstract

We demonstrate a method for describing data-flow analyses based program optimizations as compositional type systems with a transformation component. Analysis results are presented in terms of types ascribed to expressions and statements, certifiable by type derivations, and the transformation component carries out the optimizations that the type derivations license. We describe dead code elimination and common subexpression elimination. In the case of common subexpression elimination we circumvent non-compositionality with a combined type system for a combination of two analyses. The motivation of this work lies in certified code applications, where an optimization of a program must be supported by a checkable justification. As an example application we highlight "proof optimization", i.e., mechanical transformation of a program's functional correctness proof together with the program, based on the analysis type derivation.

Key words: data-flow analyses, optimizations, type systems, certification, proof optimization

1 Introduction

Imperative program optimizations based on data-flow analyses are usually presented in an algorithmic manner, whereby the algorithms typically do not work directly on the phrase structure of the given program, but rather on

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

[1] This paper expands on the talk given at NWPT 2006 in Reykjavík.

an intermediate form such as its control-flow graph. This is a good way to go about optimizing programs, but it is not an ideal presentation of what is done, if the optimizations are required to have justifications that can be communicated. In applications such as certified code, it is desirable to work with an account of optimizations fit to deliver machine-checkable justifications. Such an account must moreover be relatively simple, since the checker will be part of the trusted computing base. Hence it should be declarative and justify an optimization of a program by an argument based on its original form.

One strong candidate for a framework for certified optimization is given by *type systems*. Type systems attest and justify properties and transformations of programs directed by their phrase structure. Typing rules are relatively easy to understand and believe and type derivations are a human-friendly format of justification. Type systems may seem a bit detached from more algorithmic frameworks, but in fact they are not so far away: extracting at least a crude type inference algorithm from a type system definition is often quite easy.

Laud et al. [13] have demonstrated that type systems are indeed an adequate framework for describing data-flow analyses, reporting a general method for producing such descriptions. The same idea is present in a different terminology in the flow logic work of Nielson and Nielson [17]. Here we show that this technique extends also to optimizations. On the example of two classical optimizations, namely *dead code elimination* and *common subexpression elimination*, stated for WHILE-programs, we demonstrate a method for describing data-flow analyses based optimizations as type systems with a transformation component. Analysis results are presented in terms of types ascribed to expressions and statements, certified by type derivations, and the transformation component carries out the optimizations licensed by these type derivations.

The case of common subexpression elimination is technically interesting, as the standard presentation requires linking of expression evaluation points to potential value reuse points and coordinated modifications of the program near both ends of such links, which seems to go against compositionality. We solve the problem with a combined type system for a combination of two analyses, with the second analysis relying on the results of the first.

To demonstrate the usefulness of the type-systematic analysis certification mechanism, we show that it supports "*proof optimization*", i.e., mechanical transformation of a program's functional correctness certificate together with the program. This is important in the context of the proof-carrying code (PCC) paradigm. A code producer that optimizes code prior to transmitting it to the consumer can construct a functional correctness proof for an original program conventionally, e.g., with the help of interactive verification tools, but obtain a certificate for the optimized form with "proof optimization". The code consumer can check the functional correctness proof of the program she

receives from the producer relying on the accompanying certificate alone and without having to learn that this program was obtained by applying a sound optimization to some functionally correct prior program. While constructing "proof compilers" for non-optimizing program compilers is relatively straightforward, "proof optimization" is non-trivial, as the transformed proof must reflect the analysis results that led to the program optimization.

The present paper builds on the work by Laud et al. [13], related to the flow logic work of Nielson and Nielson [17]. But very close by its spirit is also Benton's work [4] where constant folding and dead code elimination are described as a type system and a relational variant of Hoare logic is formulated that facilitates derivation of consequences from the correctness of these optimizations within a program logic. [2]

The organization of the paper is as follows. In Section 2, we outline the method of using type systems and transformation add-ons to describe data-flow analyses and optimizations on the example of dead code elimination, commenting in particular on how the soundness of analyses and optimizations can be proved in the type-systematic setting. We also highlight the implications for "proof compilation". The more complicated optimization of common subexpression elimination is addressed in Section 3. In Sections 4 and 5, we comment more thoroughly on the related work and present our conclusions.

We explain our application of type systems, but must assume that our reader is familiar with data-flow analyses based optimizations, as explained in standard textbooks, e.g., [16], and with Hoare logic [10]. The programming language we consider is WHILE. Its statements $s \in \mathbf{Stm}$, arithmetic expressions $a \in \mathbf{AExp}$ and boolean expressions $b \in \mathbf{BExp}$ are defined over a set of program variables $x \in \mathbf{Var}$ in the following way:

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

The states $\sigma \in \mathbf{State}$ of the natural (i.e., big-step) semantics are stores, i.e., associations of integer values to variables, $\mathbf{State} =_{\mathrm{df}} \mathbf{Var} \to \mathbb{Z}$. We write $[\![a]\!]\sigma$ (resp. $[\![b]\!]\sigma$) for the integer value of an arithmetic expression a (resp. truth value of a boolean expression b) in a state σ. The circumstance that σ'

[2] The oft-cited work by Volpano et al. [23] on a type system for secure information flow is not strongly related. Their type system describes a flow-insensitive analysis (so programs types are invariant properties of states). We are interested in stronger, flow-sensitive analyses and type programs with state pre- and postproperty pairs. For secure information flow, a type system like this has been put forward by Hunt and Sands [11].

is a final state for a statement s and initial state σ is denoted by $\sigma \succ s \rightarrow \sigma'$. That Q is a derivable postcondition for s and a pre-condition P is written $\{P\}\, s\, \{Q\}$. For reference, the rules of the semantics and the Hoare logic are given in Appendix A.

2 Dead Code Elimination

2.1 Type System for Live Variables Analysis

We begin with type systems for dead code elimination and its underlying analysis, the live variables analysis. Discussing this analysis, we also comment on the general method for describing data-flow analyses as type systems [13].

We call a variable *live* at a program point, if there exists a path from that program point which (a) contains a useful use of the variable (by which we mean a use in an assignment to a variable that is live at the end of the assignment, or a use in an if- or while-guard) and (b) does not contain an assignment to the variable before this use [3]. The corresponding live variables analysis determines, for each program point, which variables *may* be live at the program point. It is a *backward* analysis, starting from a set of variables that one wishes to consider live at the end of the program (at the top level, this would typically be **Var**: the final values of all variables are of interest).

The types and the subtyping relation of the type system corresponding to a data-flow analysis are the same as the underlying poset of the analysis, in this case the poset $(D, \leq) =_{\mathrm{df}} (\mathcal{P}(\mathbf{Var}), \supseteq)$. A state on a computation path has type $\mathit{live} \in D$, if some variable live in that state is not in *live*. (The partial order is the opposite to the usual one for live variable analysis in order to get a natural subsumption rule, covariant in the posttype, contravariant in the pretype: from the point of subsumption, the natural analyses are forward may and backward must analyses; a backward may analysis is turned into a backward must analysis by reversing the partial order. The property specified by a type is negated, because the analysis is backward.) A typing judgement for an arithmetic expression is of the form $a : \mathit{live} \longrightarrow \mathit{live}'$, where *live*, live', the pretype and the posttype, are in each case elements of D (for boolean expressions and statements they are similar). Generally, the intended meaning is that, if the property specified by *live* holds before evaluating an expression a, then the property specified by live' after the evaluation. In our case, this says that, if some variable live before the evaluation is not in *live*, then some

[3] This is the strong version of liveness. In the alternative weaker version, any use of a variable makes it live.

$$\frac{}{x : live \cup \{x\} \longrightarrow live}\ \mathrm{var}_{\mathrm{lv}} \qquad \frac{}{n : live \longrightarrow live}\ \mathrm{num}_{\mathrm{lv}} \qquad \frac{a_0 : live \longrightarrow live'' \quad a_1 : live'' \longrightarrow live'}{a_0 + a_1 : live \longrightarrow live'}\ +_{\mathrm{lv}}$$

$$\frac{a_0 : live \longrightarrow live'' \quad a_1 : live'' \longrightarrow live'}{a_0 = a_1 : live \longrightarrow live'}\ =_{\mathrm{lv}}$$

$$\frac{x \in live' \quad a : live \longrightarrow live' \setminus \{x\}}{x := a : live \longrightarrow live'}\ :=_{1\mathrm{lv}} \qquad \frac{x \notin live}{x := a : live \longrightarrow live}\ :=_{2\mathrm{lv}}$$

$$\frac{}{\mathsf{skip} : live \longrightarrow live}\ \mathrm{skip}_{\mathrm{lv}} \qquad \frac{s_0 : live \longrightarrow live'' \quad s_1 : live'' \longrightarrow live'}{s_0; s_1 : live \longrightarrow live'}\ \mathrm{comp}_{\mathrm{lv}}$$

$$\frac{b : live \longrightarrow live'' \quad s_t : live'' \longrightarrow live' \quad s_f : live'' \longrightarrow live'}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : live \longrightarrow live'}\ \mathrm{if}_{\mathrm{lv}} \qquad \frac{b : live \longrightarrow live' \quad s_t : live' \longrightarrow live}{\mathsf{while}\ b\ \mathsf{do}\ s_t : live \longrightarrow live'}\ \mathrm{while}_{\mathrm{lv}}$$

$$\frac{live \leq live_0 \quad s : live_0 \longrightarrow live'_0 \quad live'_0 \leq live'}{s : live \longrightarrow live'}\ \mathrm{conseq}_{\mathrm{lv}}$$

Fig. 1. Type system for live variables analysis

variable live after is not in $live'$, or, contrapositively (in the direction of the analysis), if all variables live after the evaluation are in $live'$, then all variables live before are in $live$. The typing rules state the constraints of the analysis. For live variables, they appear in Figure 1.

The rule for variables reflects the fact that a use of a variable makes it live (again in the direction of the analysis, i.e., backwards; this is also the direction for the comments about all other rules below). As a result the weakest pretype of an expression is obtained by adding to the posttype its free variables.

There are two rules for assignment: for the cases where the assigned variable x is in the posttype and where it is not. In the first case, since x is possibly live at the end, the variables in the expression a assigned to it should be included in the pretype. However, x itself should first be removed as the assignment kills it (it will reappear in the pretype, if it is among the variables in the expression a). In the second case, since x is necessarily dead at the end, there is no point in making the variables in a possibly live at the beginning.

The rule $\mathrm{comp}_{\mathrm{lv}}$ for composition should be self-explanatory. To type an if-statement, both of the branches have to have the same type (with the $\mathrm{conseq}_{\mathrm{lv}}$ rule, the pretypes may be strengthened to agree). Additionally, variables used in the guard add to the pretype of the if-statement.

The rule $\mathrm{while}_{\mathrm{lv}}$ requires an invariant-type for the beginning of the loop body/end of the guard to type a loop. The analysis computes it from a given posttype as the greatest fixpoint of a function monotone with respect to $\leq$. The type system accepts any fixpoint. The $\mathrm{conseq}_{\mathrm{lv}}$ rule can be used to strengthen the given posttype to any suitable such type. There is also the invariant-type for the end of the loop body/beginning of the guard, obtained by adding the variables in the guard. There is an obvious similarity between the loop invariants here to those in Hoare logic.

The reason why the while_{lv} rule has the presented form can be made more clear on this example: $\mathsf{while}\ u < v\ \mathsf{do}\ (x := y; u := u + 1; y := z)$. If as a posttype of the loop we have the variable x (i.e., x is the only variable whose value we are interested in at the end), then in the naive approach (without strengthening it to the invariant for the beginning of the loop body) it would appear that the assignment to y is not necessary (while it clearly is so, since the second time the loop is entered, its value has changed, and the changed value is assigned to x). Also, the fact that the assignment to u is not considered necessarily dead is because the invariant for the end of the loop body has the free variables of the guard already included.

The $\text{conseq}_{\text{lv}}$ rule is a subsumption rule, but its role is completely analogous to that of the consequence rule in Hoare logic (except that checking subtyping is trivial whereas checking logical consequence needs a logic theorem prover, if no hints about the proof are supplied).

A big difference of the type system from the analysis algorithm is that while the algorithm computes the weakest preproperty for a given postproperty, the type system approves any valid pretype-posttype pair. Again, stronger pretypes are easy to get from the weakest one with $\text{conseq}_{\text{lv}}$. The analysis algorithm can in fact be seen as an algorithm for principal type inference: given a statement s and a posttype live', one attempts to construct a type derivation. Constructing the tightest one takes calculation of greatest fixpoints with respect to $\leq$ to obtain the invariant-types of the loops and as a result one learns the weakest pretype live. This type declares only these variables to be possibly live initially that really have some chance of being live. In type systems jargon, it makes sense to call live the principal type of s with respect to live'.

Some remarks are in order concerning important commonalities and differences between type systems such as the one we have just introduced and Hoare logics (such as the standard Hoare logic). By their general design and purpose, type systems and Hoare logics are very similar. A big difference however is that derivability checking in a Hoare logic is generally undecidable, since it reduces to checking of logical consequence. In a type system, we expect that that subtyping is trivially or easily checkable, hence the same holds of type-derivation checking. Also, constructing derivations in Hoare logic adds the task of construction of (useful formulae for) loop invariants, which is also incomputable (with greatest/least fixpoint operators, loop invariants can be constructed trivially). The loop invariants required in principal type derivations however are computable (for analysis domains with the ascending chain property), so that mechanical type inference becomes easy.

Thus, mechanical construction of Hoare logic proofs requires that the program comes annotated with loop invariants and hints for checking the required log-

ical consequences (verification conditions). Mechanical type inference, in contrast, requires no hints and is completely straightforward.

Soundness of the live variable analysis with respect to the natural semantics is conveniently formulated "relationally". Let $\sigma \sim_{live} \sigma_*$ denote that two states σ and σ_* agree on all variables in a set $live \subseteq \mathbf{Var}$, i.e., $\bigwedge_{x \in live} \sigma(x) = \sigma_*(x)$. Soundness states that any program is simulated by itself with respect to $\sim$.

Theorem 1 (Soundness of live variables analysis)
(o) If $live \leq live'$ *and* $\sigma \sim_{live} \sigma_*$, *then* $\sigma \sim_{live'} \sigma_*$.
(i) If $a : live \longrightarrow live'$ *and* $\sigma \sim_{live} \sigma_*$, *then* $[\![a]\!]\sigma = [\![a]\!]\sigma_*$ *and* $\sigma \sim_{live'} \sigma_*$.
(ii) If $b : live \longrightarrow live'$ *and* $\sigma \sim_{live} \sigma_*$, *then* $[\![b]\!]\sigma = [\![b]\!]\sigma_*$ *and* $\sigma \sim_{live'} \sigma_*$.
(iii) If $s : live \longrightarrow live'$ *and* $\sigma \sim_{live} \sigma_*$, *then*
— $\sigma \succ s \rightarrow \sigma'$ *implies the existence of* σ'_* *such that* $\sigma' \sim_{live'} \sigma'_*$ *and* $\sigma_* \succ s \rightarrow \sigma'_*$,
— $\sigma_* \succ s \rightarrow \sigma'_*$ *implies the existence of* σ' *such that* $\sigma' \sim_{live'} \sigma'_*$ *and* $\sigma \succ s \rightarrow \sigma'$.

[Note that the second half of (iii) is incidentally equivalent to the first as we compare two evaluations of the same statement and $\sim$ is symmetric. We shall soon encounter situations where two different statements are evaluated or/and the similarity relation is not symmetric.]

Intuitively, the theorem says that the initial values of the variables dead before an evaluation of the statement cannot affect the final values of those live after the evaluation (so if the evaluation is started with "wrong" initial values of the initially dead variables, the final values of the finally live variables will still come out "right", but nothing is guaranteed about the final values of the finally dead variables). This is exactly the semantic idea of liveness.

A neat byproduct of the type-systematic approach is that proving soundness of the analysis in the above sense becomes straightforward.

PROOF. (o) is proved relying on the definition of subtyping. (i)-(iii) are proved by induction on the structure of the type derivation. □

Soundness of the analysis with respect to the natural semantics has a formal counterpart in terms of derivations in the Hoare logic (which, after all, is a formal description of the semantics). Let $P|_{live}$ abbreviate the formula $\exists [v(x) | x \notin live](P[v(x)/x | x \notin live])$, where v is some assignment of unique logic variable names to program variables (so that, informally, $P|_{live}$ is obtained from P by quantifying out all program variables not in *live*). For example for assertion $P \equiv x = 2 \wedge y = 7$ and type $live = \{x\}$, $P|_{live}$ is $\exists y' \, (x = 2 \wedge y' = 7)$.

Theorem 2
(o) If $live \leq live'$, *then* $P|_{live} \models P|_{live'}$.

(i) If $a : live \longrightarrow live'$*, then* $(P[a/w])|_{live} \models (P|_{live'})[a/w]$ *(where* w *is a logic variable).*
(ii) If $b : live \longrightarrow live'$*, then* $(P[b/w])|_{live} \models (P|_{live'})[b/w]$ *(where* w *is a logic variable). As a consequence,* $P|_{live} \models b \supset ((b \wedge P)|_{live'})$ *and* $P|_{live} \models \neg b \supset ((\neg b \wedge P)|_{live'})$.
(iii) If $s : live \longrightarrow live'$ *and* $\{P\}\, s\, \{Q\}$*, then also* $\{P|_{live}\}\, s\, \{Q|_{live'}\}$.

The theorem can be concluded from Theorem 1 using the soundness and completeness of the Hoare logic.

PROOF. (Non-constructive proof) The proofs for (o)-(iii) via the semantics are as follows.

(o) Assume that $live \leq live'$. We are required to show that $P|_{live} \models P|_{live'}$, i.e., that for any state σ_* and valuation of logic variables α, $\sigma_* \models P|_{live}$ implies $\sigma_* \models P|_{live}$. Consider any σ_* and α such that $\sigma_* \models_\alpha P|_{live}$. There must exist a state σ such that $\sigma \sim_{live} \sigma_*$ and $\sigma \models P$. By Theorem 1 then $\sigma \sim_{live'} \sigma_*$, but that further implies $\sigma_* \models P|_{live'}$, as necessary.

(i) Assume that $a : live \longrightarrow live'$. We are required to show that $(P[a/w])|_{live} \models (P|_{live'})[a/w]$, i.e., that for any state σ_* and valuation of logic variables α, $\sigma_* \models_\alpha (P[a/w])|_{live}$ implies $\sigma_* \models_\alpha (P|_{live'})[a/w]$.

Consider any σ_* and α such that $\sigma_* \models_\alpha (P[a/w])|_{live}$. There must exist σ such that $\sigma \sim_{live} \sigma_*$ and $\sigma \models_\alpha P[a/w]$, i.e., $\sigma \models_{\alpha[w \mapsto [\![a]\!]\sigma]} P$. By Theorem 1, it must therefore be that $[\![a]\!]\sigma = [\![a]\!]\sigma_*$ and $\sigma \sim_{live'} \sigma_*$. The first gives us that $\sigma \models_{\alpha[w \mapsto [\![a]\!]\sigma_*]} P$ and the second further that $\sigma_* \models_{\alpha[w \mapsto [\![a]\!]\sigma_*]} P|_{live'}$, i.e. $\sigma_* \models_\alpha (P|_{live'})[a/w]$, as required.

(ii) The main statement is proved just as (i). Once we already know that $(P[b/w])|_{live} \models (P|_{live'})[b/w]$ for any P and w, we can conclude $P|_{live} \models (b \supset (b \wedge P))|_{live} \equiv ((w \supset (b \wedge P))[b/w])|_{live} \models ((w \supset (b \wedge P))|_{live'})[b/w] \models ((w \supset (b \wedge P)|_{live'})[b/w] \equiv b \supset (b \wedge P)|_{live'}$. Similarly for $\neg b$.

(iii) Assume that $s : live \longrightarrow live'$ and $\{P\}\, s\, \{Q\}$. We must show $\{P|_{live}\}\, s\, \{Q|_{live'}\}$. By completeness of the Hoare logic, it suffices to show that, for any states σ_*, σ'_* and valuation α of logic variables, $\sigma_* \models_\alpha P|_{live}$ and $\sigma_* \succ s \rightarrow \sigma'_*$ imply $\sigma'_* \models_\alpha Q|_{live'}$.

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

By Theorem 1 the facts $s : live \longrightarrow live'$, $\sigma \sim_{live} \sigma_*$ and $\sigma_* \succ s \rightarrow \sigma'_*$ yield that there must exist σ' such that $\sigma' \sim_{live'} \sigma'_*$ and $\sigma \succ s \rightarrow \sigma'$.

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

The theorem is also provable constructively, without any indirection via semantics, by induction on the structure of the type derivation. The proof of (iii) gives a transformation of a given Hoare triple derivation into a derivation for the modified triple.

PROOF. (Constructive proof)
(o) The assumption $live \leq live'$ means $live \supseteq live'$. Therefore, $P|_{live} \models P|_{live'}$ follows by existential introduction: for any variable $x \in live \setminus live'$, a suitable constructed witness for $v(x)$ is x.

(i) We construct a derivation of $(P[a/w])|_{live} \models (P|_{live'})[a/w]$ by induction on the derivation of $a : live \longrightarrow live'$, which gives the following cases.

- The type derivation is

$$\frac{}{x : live \cup \{x\} \longrightarrow live}\ \mathrm{var_{lv}}$$

  The required entailment $(P[x/w])|_{live\cup\{x\}} \models (P|_{live})[x/w]$ follows from $(P[x/w])|_{live\cup\{x\}} = P|_{live\cup\{x\}}[x/w]$, which holds trivially, and $P|_{live\cup\{x\}}[x/w] \models P|_{live}[x/w]$, which results from $P|_{live\cup\{x\}} \models P|_{live}$ from existential introduction (taking x as the constructed witness for $v(x)$), if $x \notin live$.
- The type derivation is

$$\frac{}{n : live \longrightarrow live}\ \mathrm{num_{lv}}$$

  The required entailment holds trivially: we have $(P[n/w])|_{live} = (P|_{live})[n/w]$.
- The type derivation is of the form

$$\frac{\begin{array}{cc}\vdots & \vdots \\ a_0 : live \longrightarrow live'' & a_1 : live'' \longrightarrow live'\end{array}}{a_0 + a_1 : live \longrightarrow live'}\ +_{\mathrm{lv}}$$

  The required entailment $(P[a_0 + a_1/w])|_{live} \models (P|_{live'})[a_0 + a_1/w]$ follows

from the following chain of entailments:

$$\begin{aligned}
(P[a_0 + a_1/w])|_{live} &\equiv (P[w_0 + w_1/w][a_1/w_1][a_0/w_0])|_{live} && w_0,\ w_1 \text{ are not free in } P \\
&\models ((P[w_0 + w_1/w][a_1/w_1])|_{live''})[a_0/w_0] && \text{by IH} \\
&\models ((P[w_0 + w_1/w])|_{live'})[a_1/w_1][a_0/w_0] && \text{by IH} \\
&\equiv (P|_{live'})[w_0 + w_1/w][a_1/w_1][a_0/w_0] && \text{trivially} \\
&\equiv (P|_{live'})[a_0 + a_1/w] && w_0,\ w_1 \text{ are not free in } P
\end{aligned}$$

where w_0 and w_1 are logic variables distinct from the free logic variables of P.

(ii) is proved similarly to (i).

(iii) We construct a derivation of $\{P|_{live}\}\, s\, \{Q|_{live'}\}$ by induction on the derivation of $s : live \longrightarrow live'$ and inspection of the derivation of $\{P\}\, s\, \{Q\}$. We have the following cases.

- The type derivation is of the form

$$\frac{\begin{array}{c}\vdots \\ x \in live' \quad a : live \longrightarrow live' \setminus \{x\}\end{array}}{x := a : live \longrightarrow live'}\ :=_{1\mathrm{lv}}$$

and the given Hoare derivation is

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

We get this modified Hoare triple derivation:

$$\frac{(P[a/x])|_{live} \models (P|_{live'})[a/x] \quad \dfrac{}{\{(P|_{live'})[a/x]\}\, x := a\, \{P|_{live'}\}}}{\{(P[a/x])|_{live}\}\, x := a\, \{P|_{live'}\}}$$

The entailment $(P[a/x])|_{live} \models (P|_{live'})[a/x]$ is the consequence of the following chain of entailments:

$$\begin{aligned}
(P[a/x])|_{live} &\equiv (P[w/x][a/w)|_{live} && w \text{ is not free in } P \\
&\models ((P[w/x])|_{live' \setminus \{x\}})[a/w] && \text{by (i)} \\
&\Leftrightarrow ((P[w/x])|_{live'})[a/w] && x \text{ does not occur in } P[w/x] \\
&\equiv (P|_{live'})[w/x][a/w] && x \in live' \\
&\equiv (P|_{live'})[a/x] && w \text{ is not free in } P
\end{aligned}$$

where w is a logic variable distinct from the free logic variables of P.

- The type derivation is of the form

$$\frac{x \notin \mathit{live}}{x := a : \mathit{live} \longrightarrow \mathit{live}} :=_{2\mathrm{lv}}$$

  and the given Hoare triple derivation is

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

  We have the Hoare triple derivation

$$\frac{(P[a/x])|_{\mathit{live}} \models (P|_{\mathit{live}})[a/x] \quad \overline{\{(P|_{\mathit{live}})[a/x]\}\, x := a\, \{P|_{\mathit{live}}\}}}{\{(P[a/x])|_{\mathit{live}}\}\, x := a\, \{P|_{\mathit{live}}\}}$$

  The entailment $(P[a/x])|_{\mathit{live}} \models (P|_{\mathit{live}})[a/x]$ is a consequence of $(P[a/x])|_{\mathit{live}} \models P|_{\mathit{live}}$, which holds by $x \notin \mathit{live}$ and existential elimination and introduction (for the constructed witness of $v(x)$ on the right one must take $a[w/x]$ where w is the assumed witness of $v(x)$ on the left), and $P|_{\mathit{live}} = (P|_{\mathit{live}})[a/x]$, which also holds since $x \notin \mathit{live}$, as P_{live} has therefore no occurrences of x.
- The type derivation is of the form

$$\frac{\begin{array}{c}\vdots\\ s_0 : \mathit{live} \longrightarrow \mathit{live}''\end{array} \quad \begin{array}{c}\vdots\\ s_1 : \mathit{live}'' \longrightarrow \mathit{live}'\end{array}}{s_0; s_1 : \mathit{live} \longrightarrow \mathit{live}'}\ \mathrm{comp}_{\mathrm{lv}}$$

  and the given Hoare derivation is of the form

$$\frac{\begin{array}{c}\vdots\\ \{P\}\, s_0\, \{R\}\end{array} \quad \begin{array}{c}\vdots\\ \{R\}\, s_1\, \{Q\}\end{array}}{\{P\}\, s_0; s_1\, \{Q\}}\ .$$

  We get the Hoare derivation

$$\frac{\begin{array}{c}\vdots\ \mathrm{IH}\\ \{P|_{\mathit{live}}\}\, s_0\, \{R|_{\mathit{live}''}\}\end{array} \quad \begin{array}{c}\vdots\ \mathrm{IH}\\ \{R|_{\mathit{live}''}\}\, s_0\, \{Q|_{\mathit{live}'}\}\end{array}}{\{P|_{\mathit{live}}\}\, s_0; s_1\, \{Q|_{\mathit{live}'}\}}$$

- The type derivation is of the form

$$\frac{\begin{array}{c}\vdots\\ b : \mathit{live} \longrightarrow \mathit{live}''\end{array} \quad \begin{array}{c}\vdots\\ s_t : \mathit{live}'' \longrightarrow \mathit{live}' \hookrightarrow s_t'\end{array} \quad \begin{array}{c}\vdots\\ s_f : \mathit{live}'' \longrightarrow \mathit{live}' \hookrightarrow s_f'\end{array}}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : \mathit{live} \longrightarrow \mathit{live}'}\ \mathrm{if}_{\mathrm{lv}}$$

  and the given Hoare triple derivation is of the form

$$\frac{\begin{array}{c}\vdots\\ \{b \wedge P\}\, s_t\, \{Q\}\end{array} \quad \begin{array}{c}\vdots\\ \{\neg b \wedge P\}\, s_f\, \{Q\}\end{array}}{\{P\}\, \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f\, \{Q\}}\ .$$

We have the Hoare triple derivation

$$\dfrac{\dfrac{P|_{live} \models \quad \vdots\ \text{IH} \quad \{(b \wedge P)|_{live''}\}\, s_t\, \{Q|_{live'}\}}{b \supset (b \wedge P)|_{live''} \qquad \{b \wedge (P|_{live})\}\, s_t\, \{Q|_{live'}\}} \qquad \dfrac{P|_{live} \models \quad \vdots\ \text{IH} \quad \{(\neg b \wedge P)|_{live''}\}\, s_f\, \{Q|_{live'}\}}{\neg b \supset (\neg b \wedge P)|_{live''} \qquad \{\neg b \wedge (P|_{live})\}\, s_f\, \{Q|_{live'}\}}}{\{P|_{live}\}\ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f\ \{Q|_{live'}\}}$$

The two entailments hold by (ii).

- The type derivation is of the form

$$\dfrac{\vdots \qquad \vdots}{b : live \longrightarrow live' \quad s_t : live' \longrightarrow live}\ \text{while}_{\text{lv}}$$
$$\mathsf{while}\ b\ \mathsf{do}\ s_t : live \longrightarrow live'$$

and the given Hoare triple derivation of the form

$$\dfrac{\vdots}{\{b \wedge P\}\, s_t\, \{P\}}$$
$$\{P\}\, \mathsf{while}\ b\ \mathsf{do}\ s_t\, \{\neg b \wedge P\}.$$

We have the Hoare triple derivation

$$\dfrac{\dfrac{\dfrac{(P|_{live}) \models b \supset (b \wedge P)|_{live'} \quad \vdots\ \text{IH} \quad \{(b \wedge P)|_{live'}\}\, s_t\, \{P|_{live}\}}{\{b \wedge (P|_{live})\}\, s_t\, \{P|_{live}\}}}{\{P|_{live}\}\, \mathsf{while}\ b\ \mathsf{do}\ s_t\, \{\neg b \wedge (P|_{live})\}} \qquad (P|_{live}) \models \neg b \supset (\neg b \wedge P)|_{live'}}{\{P|_{live}\}\, \mathsf{while}\ b\ \mathsf{do}\ s_t\, \{(\neg b \wedge P)|_{live'}\}}$$

The two entailments hold by (ii).

- The type derivation is of the form

$$\dfrac{\vdots}{live \leq live_0 \quad s : live_0 \longrightarrow live_0' \quad live_0' \leq live'}\ \text{conseq}_{\text{lv}}$$
$$s : live \longrightarrow live'$$

and the given Hoare triple derivation is of the form

$$\dfrac{\vdots}{P \models P' \quad \{P'\}\, s\, \{Q'\} \quad Q' \models Q}$$
$$\{P\}\, s\, \{Q\}.$$

We have the following Hoare triple derivation:

$$\dfrac{P|_{live} \models P'|_{live_0} \quad \vdots\ \text{IH} \quad \{P'|_{live_0}\}\, s\, \{Q'|_{live_0'}\} \quad Q'|_{live_0'} \models Q|_{live'}}{\{P|_{live}\}\, s\, \{Q|_{live'}\}}$$

$$\frac{x \in \mathit{live}' \quad a : \mathit{live} \longrightarrow \mathit{live}' \setminus \{x\}}{x := a : \mathit{live} \longrightarrow \mathit{live}' \hookrightarrow x := a} :=_{1_{\mathrm{lv}}}^{\mathrm{opt}} \qquad \frac{x \notin \mathit{live}}{x := a : \mathit{live} \longrightarrow \mathit{live} \hookrightarrow \mathsf{skip}} :=_{2_{\mathrm{lv}}}^{\mathrm{opt}}$$

$$\frac{}{\mathsf{skip} : \mathit{live} \longrightarrow \mathit{live} \hookrightarrow \mathsf{skip}}\ \mathrm{skip}_{\mathrm{lv}}^{\mathrm{opt}} \qquad \frac{s_0 : \mathit{live} \longrightarrow \mathit{live}'' \hookrightarrow s_0' \quad s_1 : \mathit{live}'' \longrightarrow \mathit{live}' \hookrightarrow s_1'}{s_0; s_1 : \mathit{live} \longrightarrow \mathit{live}' \hookrightarrow s_0'; s_1'}\ \mathrm{comp}_{\mathrm{lv}}^{\mathrm{opt}}$$

$$\frac{b : \mathit{live} \longrightarrow \mathit{live}'' \quad s_t : \mathit{live}'' \longrightarrow \mathit{live}' \hookrightarrow s_t' \quad s_f : \mathit{live}'' \longrightarrow \mathit{live}' \hookrightarrow s_f'}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : \mathit{live} \longrightarrow \mathit{live}' \hookrightarrow \mathsf{if}\ b\ \mathsf{then}\ s_t'\ \mathsf{else}\ s_f'}\ \mathrm{if}_{\mathrm{lv}}^{\mathrm{opt}}$$

$$\frac{b : \mathit{live} \longrightarrow \mathit{live}' \quad s_t : \mathit{live}' \longrightarrow \mathit{live} \hookrightarrow s_t'}{\mathsf{while}\ b\ \mathsf{do}\ s_t : \mathit{live} \longrightarrow \mathit{live}' \hookrightarrow \mathsf{while}\ b\ \mathsf{do}\ s_t'}\ \mathrm{while}_{\mathrm{lv}}^{\mathrm{opt}}$$

$$\frac{\mathit{live} \leq \mathit{live}_0 \quad s : \mathit{live}_0 \longrightarrow \mathit{live}_0' \hookrightarrow s' \quad \mathit{live}_0' \leq \mathit{live}}{s : \mathit{live} \longrightarrow \mathit{live}' \hookrightarrow s'}\ \mathrm{conseq}_{\mathrm{lv}}^{\mathrm{opt}}$$

Fig. 2. Type system for dead code elimination

The entailment $P|_{\mathit{live}} \models P'|_{\mathit{live}_0}$ follows from $P|_{\mathit{live}} \models P'|_{\mathit{live}}$, which holds by $P \models P'$, and $P'|_{\mathit{live}} \models P'|_{\mathit{live}_0}$, which holds by (o). Similarly for $Q'|_{\mathit{live}_0'} \models Q|_{\mathit{live}'}$. □

2.2 Type System for Dead Code Elimination

Dead code elimination removes from a statement the assignments that cannot affect the final values of the variables that are live at the end.

This optimization can be explained in an extended version of the live variables type system. Apart for assigning types to statements, it also defines their corresponding optimized forms. A typing judgement has the form $s : \mathit{live} \longrightarrow \mathit{live}' \hookrightarrow s'$, where s' is a statement; it says that s' is the optimized form of s. The rules of this extended type system are given in Figure 2. (Arithmetic and boolean expressions are not optimized, so we do not repeat their rules.)

The only rule where an actual optimization takes place is $:=_{2_{\mathrm{lv}}}^{\mathrm{opt}}$: if we know from the typing of an assignment that the assigned variable is necessarily dead after it, then its value cannot affect any live variables. Thus, we can replace the assignment with skip. We could add even stronger optimizations, for example removing a skip from a sequence or replacing an if-statement with skip, if both branches optimize to skip, but this can be seen as a separate optimization and we do not integrate it here.

An example of a derivation of a dead code elimination can be seen in Figure 3. In this example, we are interested in the program slice concerned with variable x. Thus the only variable in the posttype is x, and the code not affecting its final value is considered dead and thus removed.

The statement and proof of soundness of dead code elimination are similar to those for the underlying analysis. Soundness says that the original and optimized form of a program simulate each other with respect to $\sim$.

$$\dfrac{\dfrac{\dfrac{}{x*2 : \{x,y\} \longrightarrow \{y\}}}{x := x*2 : \{x,y\} \longrightarrow \{x,y\} \hookrightarrow x := x*2} \quad \dfrac{}{z := z+1 : \{x,y\} \longrightarrow \{x,y\} \hookrightarrow \mathsf{skip}}}{x := x*2; z := z+1 : \{x,y\} \longrightarrow \{x,y\} \hookrightarrow x := x*2; \mathsf{skip}}$$
$$\dfrac{}{\mathsf{while}\ x < y\ \mathsf{do}\ (x := x*2; z := z+1) : \{x,y\} \longrightarrow \{x,y\} \hookrightarrow \mathsf{while}\ x < y\ \mathsf{do}\ (x := x*2; \mathsf{skip})}$$
$$\mathsf{while}\ x < y\ \mathsf{do}\ (x := x*2; z := z+1) : \{x,y\} \longrightarrow \{x\} \hookrightarrow \mathsf{while}\ x < y\ \mathsf{do}\ (x := x*2; \mathsf{skip})$$

Fig. 3. An analysis and transformation of an example program

Theorem 3 (Soundness of dead code elimination) *If* $s : live \longrightarrow live' \hookrightarrow s'$ *and* $\sigma \sim_{live} \sigma_*$, *then*
— $\sigma \succ s \rightarrow \sigma'$ *implies the existence of* σ'_* *such that* $\sigma' \sim_{live'} \sigma'_*$ *and* $\sigma_* \succ s' \rightarrow \sigma'_*$,
— $\sigma_* \succ s' \rightarrow \sigma'_*$ *implies the existence of* σ' *such that* $\sigma' \sim_{live'} \sigma'_*$ *and* $\sigma \succ s \rightarrow \sigma'$.

[Now the two halves are no longer equivalent. They say that s is simulated by s' and vice versa, s and s' being different statements.]

PROOF. By induction on the type derivation. □

Similarly to Theorem 1, also Theorem 3 has a counterpart for the Hoare logic. Essentially, it says that optimization preserves and reflects Hoare triple derivability (in fact even actual derivations).

Theorem 4 *If* $s : live \longrightarrow live' \hookrightarrow s'$, *then*
— $\{P\}\, s\, \{Q\}$ *implies* $\{P|_{live}\}\, s'\, \{Q|_{live'}\}$,
— $\{P\}\, s'\, \{Q\}$ *implies* $\{P|_{live}\}\, s\, \{Q|_{live'}\}$.

Analogously to Theorem 2 (which is the counterpart of Theorem 1), this theorem can be proved non-constructively from Theorem 3, relying on the soundness and completeness of the Hoare logic, or constructively.

PROOF. (Non-constructive proof) We only prove the first half, relying on the second half of Theorem 3 (to prove preservation of Hoare derivability along the optimization one needs reflection of semantic derivability, and to prove reflection of Hoare derivability preservation of semantic derivability is needed).

Assume that $s : live \longrightarrow live' \hookrightarrow s'$ and $\{P\}\, s\, \{Q\}$. We must show $\{P|_{live}\}\, s'\, \{Q|_{live'}\}$. By completeness of the Hoare logic, it suffices to show that, for any states σ_*, σ'_* and valuation α of logic variables, $\sigma_* \models_\alpha P|_{live}$ and $\sigma_* \succ s' \rightarrow \sigma'_*$ imply $\sigma'_* \models_\alpha Q|_{live'}$.

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

By Theorem 3 (second half) the facts $s : \mathit{live} \longrightarrow \mathit{live}' \hookrightarrow s'$, $\sigma \sim_{\mathit{live}} \sigma_*$ and $\sigma_* \succ s' \rightarrow \sigma'_*$ yield that there must exist σ' such that $\sigma' \sim_{\mathit{live}'} \sigma'_*$ and $\sigma \succ s \rightarrow \sigma'$.

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

PROOF. (Constructive proof) We only look at the case of the rule $:=_{2\mathrm{lv}}^{\mathrm{opt}}$ in the proof of the first half.

The type derivation is

$$\frac{x \notin \mathit{live}}{x := a : \mathit{live} \longrightarrow \mathit{live} \hookrightarrow \mathsf{skip}} :=_{2\mathrm{lv}}^{\mathrm{opt}}$$

and the given Hoare triple derivation is

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

For the optimized program skip, we have the Hoare triple derivation

$$\frac{P[a/x]|_{\mathit{live}} \models P|_{\mathit{live}} \quad \overline{\{P|_{\mathit{live}}\}\, \mathsf{skip}\, \{P|_{\mathit{live}}\}}}{\{P[a/x]|_{\mathit{live}}\}\, \mathsf{skip}\, \{P|_{\mathit{live}}\}}$$

The entailment $P[a/x]_{\mathit{live}} \models P_{\mathit{live}}$ holds by $x \notin \mathit{live}$ and existential elimination and introduction (for the constructed witness of $v(x)$ the right we take $a[w/x]$ where w is the assumed witness for $v(x)$ on the left). □

The first half of Theorem 4 gives us "proof optimization". Given a Hoare triple derivation for an original program, we get a modified Hoare triple and its derivation for its optimized form. In the example of program analysis and optimization in Figure 3 we saw that the program

$$\mathsf{while}\ x < y\ \mathsf{do}\ (x := x * 2; z := z + 1)$$

admits the type $\{x, y\} \longrightarrow \{x\}$ and that the corresponding optimized form is $\mathsf{while}\ x < y\ \mathsf{do}\ (x := x{*}2; \mathsf{skip})$ (further simplifiable to $\mathsf{while}\ x < y\ \mathsf{do}\ x := x{*}2$ by a trivial post-processing pass based on the equivalence $s; \mathsf{skip} = s$).

A Hoare logic derivation for the triple

$$\begin{gathered}\{x = 2 \wedge z = 1 \wedge y > 1\} \\ \mathsf{while}\ x < y\ \mathsf{do}\ (x := x * 2; z := z + 1) \\ \{x = 2^z \wedge z = \mathit{ceil}(\log y)\}\end{gathered}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{x = 2^{z+1} \wedge z + 1 \leq ceil(\log y)\}\, z := z + 1\, \{x = 2^z \wedge z \leq ceil(\log y)\}}}{\{x * 2 = 2^{z+1} \wedge z + 1 \leq ceil(\log y)\}\, x := x * 2\, \{x = 2^{z+1} \wedge z + 1 \leq ceil(\log y)\}} \;/}{\{x * 2 = 2^{z+1} \wedge z + 1 \leq ceil(\log y)\}\, x := x * 2; z := z + 1\, \{x = 2^z \wedge z \leq ceil(\log y)\}}}{\{x < y \wedge x = 2^z \wedge z \leq ceil(\log y)\}\, x := x * 2; z := z + 1\, \{x = 2^z \wedge z \leq ceil(\log y)\}}}{\{x = 2^z \wedge z \leq ceil(\log y)\}\, \mathsf{while}\ x < y\ \mathsf{do}\ (x := x * 2; z := z + 1)\, \{x \not< y \wedge x = 2^z \wedge z \leq ceil(\log y)\}}}{\{x = 2 \wedge z = 1 \wedge y > 1\}\, \mathsf{while}\ x < y\ \mathsf{do}\ (x := x * 2; z := z + 1)\, \{x = 2^z \wedge z = ceil(\log y)\}}$$

Fig. 4. A proof of the example program

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\exists z'\,(x = 2^{z'} \wedge z' \leq ceil(\log y))\}\, \mathsf{skip}\, \{\exists z'\,(x = 2^{z'} \wedge z' \leq ceil(\log y))\}}}{\{\exists z'\,(x = 2^{z'+1} \wedge z' + 1 \leq ceil(\log y))\}\, \mathsf{skip}\, \{\exists z'\,(x = 2^{z'} \wedge z' \leq ceil(\log y))\}} \;/}{\{\exists z'\,(x * 2 = 2^{z'+1} \wedge z' + 1 \leq ceil(\log y))\}\, x := x * 2\, \{\exists z'\,(x = 2^{z'+1} \wedge z' + 1 \leq ceil(\log y))\}}}{\{\exists z'\,(x * 2 = 2^{z'+1} \wedge z' + 1 \leq ceil(\log y))\}\, x := x * 2; \mathsf{skip}\, \{\exists z'\,(x = 2^{z'} \wedge z' \leq ceil(\log y))\}}}{\{x < y \wedge \exists z'\,(x = 2^{z'} \wedge z' \leq ceil(\log y))\}\, x := x * 2; \mathsf{skip}\, \{\exists z'\,(x = 2^{z'} \wedge z' \leq ceil(\log y))\}}}{\{\exists z'\,(x = 2^{z'} \wedge z' \leq ceil(\log y))\}\, \mathsf{while}\ x < y\ \mathsf{do}\ (x := x * 2; \mathsf{skip})\, \{x \not< y \wedge \exists z'\,(x = 2^{z'} \wedge z' \leq ceil(\log y))\}}}{\{\exists z'\,(x = 2 \wedge z' = 1 \wedge y > 1)\}\, \mathsf{while}\ x < y\ \mathsf{do}\ (x := x * 2; \mathsf{skip})\, \{\exists z'\,(x = 2^{z'} \wedge z = ceil(\log y))\}}}{\{\exists z'\,(x = 2 \wedge z' = 1 \wedge y > 1)\}\, \mathsf{while}\ x < y\ \mathsf{do}\ (x := x * 2; \mathsf{skip})\, \{\exists y', z'\,(x = 2^{z'} \wedge z' = ceil(\log y'))\}}$$

Fig. 5. Transformed proof

is given in Figure 4. (In order to save space, we have not spelled out the side conditions of inferences by the consequence rule.)

This Hoare triple derivation is mechanically transformable, following the given type derivation, into the derivation of the modified Hoare triple

$$\begin{array}{c}\{\exists z'\,(x = 2 \wedge z' = 1 \wedge y > 1)\} \\ \mathsf{while}\ x < y\ \mathsf{do}\ x := x * 2 \\ \{\exists y', z'\,(x = 2^{z'} \wedge z' = ceil(\log y')\}\end{array}$$

given in Figure 5. (Note the inference by the consequence rule after the rule for skip: this involves a change of witness for z' as described in the proof of Theorem 4, a form of a shadow of the assignment $z := z + 1$ from the original program.) The modified Hoare triple is equivalent to

$$\{x = 2 \wedge y > 1\}\, \mathsf{while}\ x < y\ \mathsf{do}\ x := x * 2\, \{\exists z' > 0\,(x = 2^{z'})\}$$

The second half of Theorem 4 makes it possible to derive a functional correctness property for an original program based on a proof for the optimized form. This is potentially useful in a debugging situation: if an optimized form happens to satisfy an unexpected Hoare triple, the original program must satisfy a corresponding modified Hoare triple. This can hint what was wrong or

$$\frac{}{x : av \longrightarrow av}\ \mathrm{var}^{\mathrm{ts}}_{\mathrm{ae}} \qquad \frac{}{n : av \longrightarrow av}\ \mathrm{num}^{\mathrm{ts}}_{\mathrm{ae}} \qquad \frac{a_0 : av \longrightarrow av'' \quad a_1 : av'' \longrightarrow av'}{a_0 + a_1 : av \longrightarrow av' \cup \{a_0 + a_1\}}\ +^{\mathrm{ts}}_{\mathrm{ae}}$$

$$\frac{a_0 : av \longrightarrow av'' \quad a_1 : av'' \longrightarrow av'}{a_0 = a_1 : av \longrightarrow av'}\ =^{\mathrm{ts}}_{\mathrm{ae}}$$

$$\frac{a : av \longrightarrow av'}{x := a : av \longrightarrow av' \backslash mod(x)}\ :=^{\mathrm{ts}}_{\mathrm{ae}}$$

$$\frac{}{\mathsf{skip} : av \longrightarrow av}\ \mathrm{skip}^{\mathrm{ts}}_{\mathrm{ae}} \qquad \frac{s_0 : av \longrightarrow av'' \quad s_1 : av'' \longrightarrow av'}{s_0; s_1 : av \longrightarrow av'}\ \mathrm{comp}^{\mathrm{ts}}_{\mathrm{ae}}$$

$$\frac{b : av \longrightarrow av'' \quad s_t : av'' \longrightarrow av' \quad s_f : av'' \longrightarrow av'}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : av \longrightarrow av'}\ \mathrm{if}^{\mathrm{ts}}_{\mathrm{ae}} \qquad \frac{b : av \longrightarrow av' \quad s_t : av' \longrightarrow av}{\mathsf{while}\ b\ \mathsf{do}\ s_t : av \longrightarrow av'}\ \mathrm{while}^{\mathrm{ts}}_{\mathrm{ae}}$$

$$\frac{av \leq av_0 \quad s : av_0 \longrightarrow av_0' \quad av_0' \leq av'}{s : av \longrightarrow av'}\ \mathrm{conseq}^{\mathrm{ts}}_{\mathrm{ae}}$$

Fig. 6. Type system for available expressions analysis

overlooked in the original program.

3 Common Subexpression Elimination

3.1 Type System for Available Expressions Analysis

The idea in common subexpression elimination is to avoid re-evaluation of non-trivial expressions. This is considerably more complicated and subtle than dead code elimination. The first phase in this optimization is the analysis of available expressions.

An (non-trivial arithmetic) expression is *available* at a program point, if every path to it (a) contains an evaluation of this expression (as a subexpression of an assigned expression or the guard of an if- or while-statement) and (b) does not contain a later modification (an assignment to a variable of the expression). The available expressions analysis finds, for each program point, which expressions *must* be available at that point. It is a *forward* analysis and starts from the set of expressions that one wishes to regard as available at the beginning of the program (typically, this would be $\emptyset$).

The types and subtyping of the type system for available expressions are $(D, \leq) =_{\mathrm{df}} (\mathcal{P}(\mathbf{AExp}^+), \supseteq)$. A state on a computation path has type $av \in D$, if all expressions in av are available in that state. A typing judgement for an arithmetic expression has the form $a : av \longrightarrow av'$ and means that, if all expressions in av are available before an evaluation of a, then all expressions in av' are available after the evaluation (for boolean expressions and statements, they are similar). The typing rules appear in Figure 6. We use $mod(x)$ to denote the set of nontrivial expressions containing x, i.e., $mod(x) =_{\mathrm{df}} \{a \in \mathbf{AExp}^+ \mid x \in FV(a)\}$. Variables and numerals do not change the availability

of expressions. The rule $+^{\text{ts}}_{\text{ae}}$ expresses that a compound expression makes itself and the subexpressions of its operands available. The rules for boolean expressions are similar. The rule $:=^{\text{ts}}_{\text{ae}}$ says that, after an assignment $x := a$, the arithmetic subexpressions of expression a have been computed and are thus available. However, since x was assigned to, any precomputed value of an expression containing x is effectively killed. The skip and composition rules should be self-explanatory. The rule $\text{if}^{\text{ts}}_{\text{ae}}$ says that if both branches of an if-statement have the same typing, we can give their posttype to the whole statement. But since the guard is always evaluated before either of the branch, the pretype of both branches is the posttype of the guard. The rule $\text{while}^{\text{ts}}_{\text{ae}}$ requires an invariant-type for the beginning of the guard/end of the body of the loop, which will become the pretype of the loop itself. A given pretype for the loop can be weakened to this type using the $\text{conseq}^{\text{ts}}_{\text{ae}}$ rule.

A type derivation of a program gives us two kinds of information. Firstly, based on the typing, we know where an expression first becomes available: where the expression is not available in its pretype. Secondly, it tells us where a pre-computed value can be used: where it is available in the pretype.

Similarly to the soundness of the live variables analysis, it is possible to state and prove that the available expressions analysis is sound. We refrain from doing it here, as this requires an instrumentation of the standard natural semantics (the concept of state must be adjusted to record the last computed value of every non-trivial arithmetic expression, and the evaluation relation of the semantics must be adjusted accordingly). But we will state and prove the soundness of common subexpression elimination.

3.2 Type System for Conditional Partial Anticipability Analysis

The technique behind common subexpression elimination is to save computed values of expressions in new variables and to use these saved values instead of re-evaluating the expressions.

Although from the available expression analysis, we know where a particular expression becomes available and where a pre-computed value can be used, this information is not enough for the purpose of common subexpression elimination. The reason is of course that a new variable to save the computation of the subexpression should only be introduced when that subexpression is possibly used later on at a point where it is available. However, at the program point where an expression becomes available, we do not have that information from available expressions analysis. We would need to know what we call *conditionally partially anticipable* (cpant) expressions. We say that an expression is cpant at a program point, if there is a path from that program point that (a)

contains an evaluation of it where the expression is available at the beginning of the evaluation, and (b) does not contain an earlier evaluation of it.

The need for the expression to be available before becoming anticipable in the reverse control-flow graph can be explained through the following example:

$$\underbrace{\mathsf{if}\ b\ \mathsf{then}\ \overbrace{x := 3}^{s_2}\ \mathsf{else}\ \overbrace{x := z + u * v}^{s_3}}_{s_1}; \underbrace{y := u * v}_{s_4}.$$

The expression $u*v$ is not available after statement s_1, since it is not evaluated in both of its branches. So the expression can not be used for optimization at statement s_4 and therefore a new variable should not be introduced at statement s_3. The type system for cpant can give us this information; since the expression $u * v$ is not available before statement s_4, the statement, although using $u * v$, does not make it anticipable.

This analysis decides which expressions *may* be cpant at each program point. It relies on the results of the available expressions analysis and is a *backward* analysis. This analysis removes the need for establishing explicit "use-def" chains for expressions, i.e., associations of program points where an expression is evaluated ("defined") to program points where the computed value could be reused ("used"). Instead, it finds, for each program point, the expressions for which there can be a value reuse point (a future point where, for some reason, one could use the present value of an expression, or even a previously stored value of it, if it is presently available).

Since the cpant type system must use the typings from the available expressions type system, it is an extension. The types and the subtyping relation are $(D, \leq) =_{\mathrm{df}} (\{(av, cpant) \in \mathcal{P}(\mathbf{AExp}^+) \times \mathcal{P}(\mathbf{AExp}^+) \mid cpant \subseteq av\}, \supseteq \times \supseteq)$. A state (on a computation path) is in a type $(av, cpant) \in D$ if in that state all expressions in av are available and some cpant expression is not in $cpant$. A typing judgement for an arithmetic expression (or a boolean expression, or a statement) is therefore of the form $a : av, cpant \longrightarrow av', cpant'$ and says that, if all expressions in av are available before an evaluation of a, then all expressions of av' are available after the evaluation, and, moreover, if all expressions av are available before an evaluation and all expressions cpant after the evaluation are in $cpant'$, then all expressions cpant before the evaluation are in $cpant$.

The typing rules are given in Figure 7. The key rule is $+^{\mathrm{ts}}_{\mathrm{cpa}}$. As was explained, an expression is made cpant at the beginning (i.e., included in the cpant pretype) only if it is already necessarily available there (thus in the intersection with the availability pretype). The rest of the rules mimic the rules of the available expressions and live variables analyses.

$$\frac{}{x : av, cpant \longrightarrow av, cpant}\ \mathrm{var}^{\mathrm{ts}}_{\mathrm{cpa}} \qquad \frac{}{n : av, cpant \longrightarrow av, cpant}\ \mathrm{num}^{\mathrm{ts}}_{\mathrm{cpa}}$$

$$\frac{a_0 : av, cpant \longrightarrow av'', cpant'' \quad a_1 : av'', cpant'' \longrightarrow av', cpant'}{a_0 + a_1 : av, cpant \cup (\{a_0 + a_1\} \cap av) \longrightarrow av' \cup \{a_0 + a_1\}, cpant'}\ +^{\mathrm{ts}}_{\mathrm{cpa}}$$

$$\frac{a_0 : av, cpant \longrightarrow av'', cpant'' \quad a_1 : av'', cpant'' \longrightarrow av', cpant'}{a_0 = a_1 : av, cpant \longrightarrow av', cpant'}\ =^{\mathrm{ts}}_{\mathrm{cpa}}$$

$$\frac{a : av, cpant \longrightarrow av', cpant'}{x := a : av, cpant \longrightarrow av' \setminus mod(x), cpant'}\ :=^{\mathrm{ts}}_{\mathrm{cpa}}$$

$$\frac{}{\mathsf{skip} : av, cpant \longrightarrow av, cpant}\ \mathrm{skip}^{\mathrm{ts}}_{\mathrm{cpa}} \qquad \frac{s_0 : av, cpant \longrightarrow av'', cpant'' \quad s_1 : av'', cpant'' \longrightarrow av', cpant'}{s_0; s_1 : av, cpant \longrightarrow av', cpant'}\ \mathrm{comp}^{\mathrm{ts}}_{\mathrm{cpa}}$$

$$\frac{b : av, cpant \longrightarrow av'', cpant'' \quad s_t : av'', cpant'' \longrightarrow av', cpant' \quad s_f : av'', cpant'' \longrightarrow av', cpant'}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : av, cpant \longrightarrow av', cpant'}\ \mathrm{if}^{\mathrm{ts}}_{\mathrm{cpa}}$$

$$\frac{b : av, cpant \longrightarrow av', cpant' \quad s_t : av', cpant' \longrightarrow av, cpant}{\mathsf{while}\ b\ \mathsf{do}\ s_t : av, cpant \longrightarrow av', cpant'}\ \mathrm{while}^{\mathrm{ts}}_{\mathrm{cpa}}$$

$$\frac{av, cpant \leq av_0, cpant_0 \quad s : av_0, cpant_0 \longrightarrow av'_0, cpant'_0 \quad av'_0, cpant'_0 \leq av', cpant'}{s : av, cpant \longrightarrow av', cpant'}\ \mathrm{conseq}^{\mathrm{ts}}_{\mathrm{cpa}}$$

Fig. 7. Type system for cpant analysis

3.3 Type System for Common Subexpression Elimination

The cpant expressions type system can now be used to perform common subexpression elimination. The rules of the optimization type system extend the cpant expressions type system; the rules are given in Figure 8.

Since additional variables need to be introduced into the program, we use an assignment $nv : \mathbf{AExp}^+ \to \mathbf{Var}_{\mathrm{aux}}$ of a unique auxiliary program variable to every non-trivial arithmetical expression ($\mathbf{Var}_{\mathrm{aux}}$ being an additional supply of program variables not available for normal programming). This is a way for two program points which will evaluate resp. reuse an expression value to agree on a variable that can safely (without the danger of a redefinition on the way) carry this value.

The main optimization is done in the rules for arithmetic expressions. The judgements for arithmetic expressions have the form $a : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd, a')$, where nvd is a sequence of assignments (auxiliary variable definitions emanating from a; the empty sequence is denoted by ϵ) and a' is an expression (the optimized version of a) (for boolean expressions, the judgements are similar). Note that optimizations need can also be made "inside" expressions, since arithmetic subexpressions can be evaluated and later used within the same expression (for example, in the expression $x*y+x*y$, the subexpression $x*y$ does not have to be evaluated twice).

There are three rules for $+$: for the case where the compound expression is already available and can be replaced with the corresponding auxiliary variable (rule $+_{1\,\mathrm{cpa}}^{\mathrm{opt}}$), the case where the expression only becomes available, and is also cpant, so an auxiliary variable definition is introduced (rule $+_{2\,\mathrm{cpa}}^{\mathrm{opt}}$) and the

$$\frac{}{n : av, cpant \longrightarrow av, cpant \hookrightarrow (\epsilon, n)}\ \mathrm{num}^{\mathrm{opt}}_{\mathrm{cpa}} \qquad \frac{}{x : av, cpant \longrightarrow av, cpant \hookrightarrow (\epsilon, x)}\ \mathrm{var}^{\mathrm{opt}}_{\mathrm{cpa}}$$

$$\frac{a_0 + a_1 \in av \quad a_0 : av, cpant \longrightarrow av'', cpant'' \hookrightarrow (nvd_0, a'_0) \quad a_1 : av'', cpant'' \longrightarrow av', cpant' \hookrightarrow (nvd_1, a'_1)}{a_0 + a_1 : av, cpant \cup \{a_0 + a_1\} \longrightarrow av' \cup \{a_0 + a_1\}, cpant' \hookrightarrow (nvd_0; nvd_1, nv(a_0 + a_1))}\ +1^{\mathrm{opt}}_{\mathrm{cpa}}$$

$$\frac{a_0 + a_1 \in cpant' \quad a_0 + a_1 \notin av \quad a_0 : av, cpant \longrightarrow av'', cpant'' \hookrightarrow (nvd_0, a'_0) \quad a_1 : av'', cpant'' \longrightarrow av', cpant' \hookrightarrow (nvd_1, a'_1)}{a_0 + a_1 : av, cpant \longrightarrow av' \cup \{a_0 + a_1\}, cpant' \hookrightarrow (nvd_0; nvd_1; nv(a_0 + a_1) := a'_0 + a'_1, nv(a_0 + a_1))}\ +2^{\mathrm{opt}}_{\mathrm{cpa}}$$

$$\frac{a_0 + a_1 \notin cpant' \quad a_0 + a_1 \notin av \quad a_0 : av, cpant \longrightarrow av'', cpant'' \hookrightarrow (nvd_0, a'_0) \quad a_1 : av'', cpant'' \longrightarrow av', cpant' \hookrightarrow (nvd_1, a'_1)}{a_0 + a_1 : av, cpant \longrightarrow av' \cup \{a_0 + a_1\}, cpant' \hookrightarrow (nvd_0; nvd_1; a'_0 + a'_1)}\ +3^{\mathrm{opt}}_{\mathrm{cpa}}$$

$$\frac{a_0 : av, cpant \longrightarrow av'', cpant'' \hookrightarrow (nvd_0, a'_0) \quad a_1 : av'', cpant'' \longrightarrow av', cpant' \hookrightarrow (nvd_1, a'_1)}{a_0 = a_1 : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd_0; nvd_1; a'_0 = a'_1)}\ =^{\mathrm{opt}}_{\mathrm{cpa}}$$

$$\frac{a : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd; a')}{x := a : av, cpant \longrightarrow av' \setminus mod(x), cpant' \hookrightarrow nvd; x := a'}\ :=^{\mathrm{opt}}_{\mathrm{cpa}}$$

$$\frac{}{\mathsf{skip} : av, cpant \longrightarrow av, cpant \hookrightarrow \mathsf{skip}}\ \mathrm{skip}^{\mathrm{opt}}_{\mathrm{cpa}}$$

$$\frac{s_0 : av, cpant \longrightarrow av'', cpant'' \hookrightarrow s'_0 \quad s_1 : av'', cpant'' \longrightarrow av', cpant' \hookrightarrow s'_1}{s_0; s_1 : av, cpant \longrightarrow av', cpant' \hookrightarrow s'_0; s'_1}\ \mathrm{comp}^{\mathrm{opt}}_{\mathrm{cpa}}$$

$$\frac{b : av, cpant \longrightarrow av'', cpant'' \hookrightarrow (nvd, b') \quad s_t : av'', cpant'' \longrightarrow av', cpant' \hookrightarrow s'_t \quad s_f : av'', cpant'' \longrightarrow av', cpant' \hookrightarrow s'_f}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : av, cpant \longrightarrow av', cpant' \hookrightarrow nvd; \mathsf{if}\ b'\ \mathsf{then}\ s'_t\ \mathsf{else}\ s'_f}\ \mathrm{if}^{\mathrm{opt}}_{\mathrm{cpa}}$$

$$\frac{b : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd; b') \quad s_t : av', cpant' \longrightarrow av, cpant \hookrightarrow s'_t}{\mathsf{while}\ b\ \mathsf{do}\ s_t : av, cpant \longrightarrow av', cpant' \hookrightarrow nvd; \mathsf{while}\ b'\ \mathsf{do}\ (s'_t; nvd)}\ \mathrm{while}^{\mathrm{opt}}_{\mathrm{cpa}}$$

$$\frac{av, cpant \leq av_0, cpant_0 \quad s : av_0, cpant_0 \longrightarrow av'_0, cpant'_0 \hookrightarrow s' \quad av'_0, cpant'_0 \leq av', cpant'}{s : av, cpant \longrightarrow av', cpant' \hookrightarrow s'}\ \mathrm{conseq}^{\mathrm{opt}}_{\mathrm{cpa}}$$

Fig. 8. Type system for common subexpression elimination

case where an expression only becomes available, but is not cpant, so it is left as it is.

The judgements for statements are of the form $s : av, cpant \longrightarrow av', cpant' \hookrightarrow s'$, where s' is a statement (the optimized form of s). The rules for assignment, skip, composition and if-statements should be straightforward. The rule $\mathrm{while}^{\mathrm{opt}}_{\mathrm{cpa}}$ for while-loops allows for reuse of expressions that are evaluated in the guard. Since a guard may be entered from two program points (the beginning of the loop and end of the loop body), the auxiliary variable definitions have to appear at both places.

The derivation in Figure 9 is an example of common subexpression elimination. At the beginning of the program, $p * q$ is available (this is just an assumption made). The expression $u * v$ becomes available after the first statement; the expression is used at three places. The information that it is used later on

in the program reaches the first assignment (via the cpant expressions type). Therefore, the value of $u * v$ is recorded in the auxiliary variable. In the if-guard and then-branch, this expression is replaced with the new variable. In the else-branch, the expression $p * q$ is replaced with a variable holding the value of this expression.

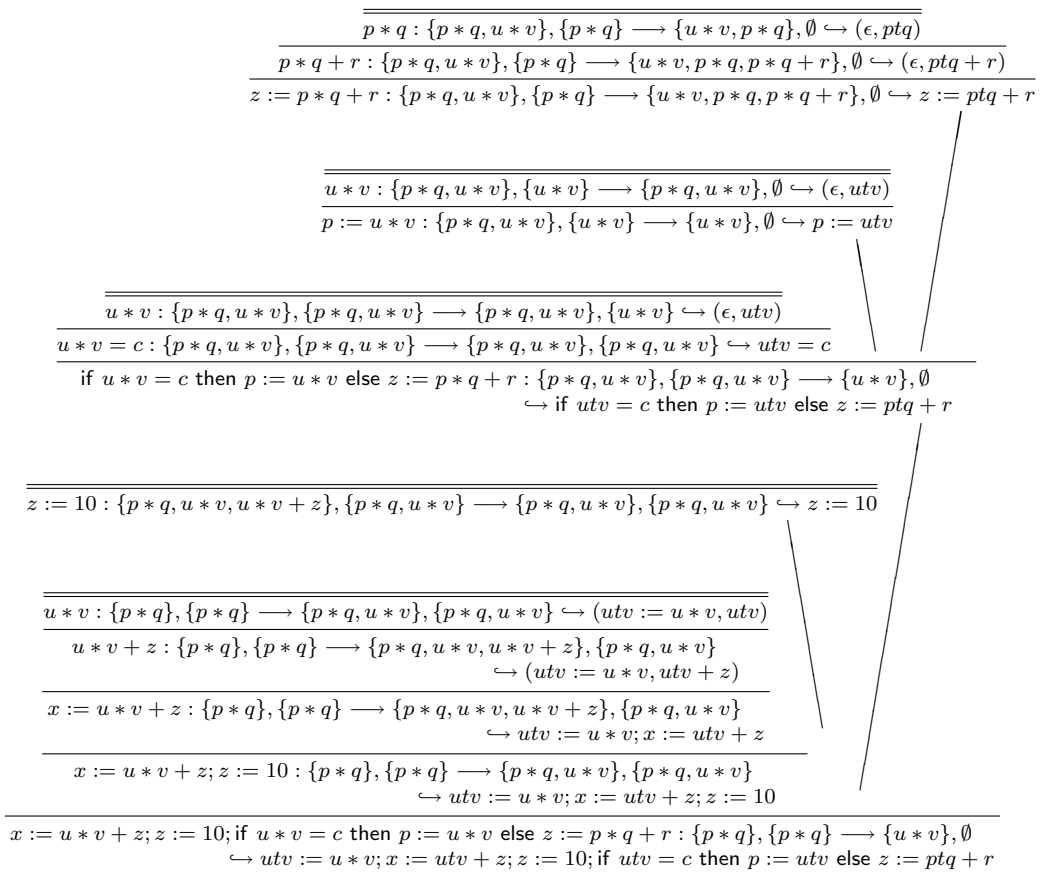

Fig. 9. An analysis and transformation of an example program

Common subexpression elimination is sound in the following sense: The optimized program and the original one simulate each other: at corresponding program points the states of the two programs agree on the normal program variables and, moreover, if some expression is both in the availability and cpant type, then its value in the state of the original program and the value of the corresponding auxiliary variable in the state of the optimized program coincide. Let $\sigma \sim_{cpant} \sigma_*$ denote that two states σ (on the normal program variables $\mathbf{Var}$) and σ_* (on the normal and auxiliary program variables $\mathbf{Var} + \mathbf{Var}_{\mathrm{aux}}$) agree modulo $cpant \subseteq \mathbf{AExp}^+$ in the following sense: $\bigwedge_{x \in \mathbf{Var}} \sigma(x) = \sigma_*(x) \wedge \bigwedge_{a \in cpant} [\![a]\!]\sigma = \sigma_*(nv(a))$. The soundness theorem is:

Theorem 5 (Soundness of common subexpression elimination)
(o) If $av, cpant \leq av', cpant'$ and $\sigma \sim_{cpant} \sigma_*$, then $\sigma \sim_{cpant'} \sigma_*$.

(i) If $a : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd, a')$, $\sigma_* \succ nvd \rightarrow \sigma'_*$, *and* $\sigma \sim_{cpant} \sigma_*$, *then* $[\![a]\!]\sigma = [\![a']\!]\sigma'_*$ *and* $\sigma \sim_{cpant'} \sigma'_*$.
(ii) If $b : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd, b')$, $\sigma_* \succ nvd \rightarrow \sigma'_*$, *and* $\sigma \sim_{cpant} \sigma_*$, *then* $[\![b]\!]\sigma = [\![b']\!]\sigma'_*$ *and* $\sigma \sim_{cpant'} \sigma'_*$.
(iii) If $s : av, cpant \longrightarrow av', cpant' \hookrightarrow s'$ *and* $\sigma \sim_{cpant} \sigma_*$ *, then*
— $\sigma \succ s \rightarrow \sigma'$ *implies the existence of* σ'_* *such that* $\sigma' \sim_{cpant'} \sigma'_*$ *and* $\sigma_* \succ s' \rightarrow \sigma'_*$,
— $\sigma_* \succ s' \rightarrow \sigma'_*$ *implies the existence of* σ' *such that* $\sigma' \sim_{cpant'} \sigma'_*$ *and* $\sigma \succ s \rightarrow \sigma'$.

As the proof of this theorem is more involved than those for Theorems 1, 3, we also show (parts of) the proof.

PROOF. (o) As $av, cpant \leq av', cpant'$ implies $cpant \supseteq cpant'$, it is immediate that $\sigma \sim_{cpant} \sigma_*$ is a stronger statement than $\sigma \sim_{cpant'} \sigma_*$.

All of (i)-(iii) are proved by induction on the structure of the type derivation. We only prove the first halves.

For (i), we use induction on the derivation of $a : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd, a')$. We assume that $\sigma_* \succ nvd \rightarrow \sigma'_*$ and $\sigma \sim_{cpant} \sigma_*$.

We keep in mind that nvd redefines no normal variables and no auxiliary variables for expressions in av and that all auxiliary variables of a' must be for expressions in av'.

We consider the following non-trivial cases.

- The type derivation is of the form

$$\frac{a_0 + a_1 \in av \quad \begin{array}{c}\vdots\\ a_0 : av, cpant \longrightarrow av'', cpant'' \\ \hookrightarrow (nvd_0, a'_0)\end{array} \quad \begin{array}{c}\vdots\\ a_1 : av'', cpant'' \longrightarrow av', cpant' \\ \hookrightarrow (nvd_1, a'_1)\end{array}}{a_0 + a_1 : av, cpant \cup \{a_0 + a_1\} \longrightarrow av' \cup \{a_0 + a_1\}, cpant' \hookrightarrow (nvd_0; nvd_1, nv(a_0 + a_1))} +1^{\text{opt}}_{\text{cpa}}$$

  There must exist a state σ''_* such that $\sigma_* \succ nvd_0 \rightarrow \sigma''_*$ and $\sigma''_* \succ nvd_1 \rightarrow \sigma'_*$. The assumption $\sigma \sim_{cpant \cup \{a_0 + a_1\}} \sigma_*$ implies $[\![a_0 + a_1]\!]\sigma = [\![nv(a_0 + a_1)]\!]\sigma_*$. As $a_0 + a_1 \in av$, we know that $nv(a_0 + a_1)$ is not modified by $nvd_0; nvd_1$, hence $[\![a_0 + a_1]\!]\sigma = [\![nv(a_0 + a_1)]\!]\sigma_* = [\![nv(a_0 + a_1)]\!]\sigma'_*$. The assumption $\sigma \sim_{cpant \cup \{a_0 + a_1\}} \sigma_*$ also tells us that $\sigma \sim_{cpant} \sigma_*$, from where by the first induction hypothesis it follows that $\sigma \sim_{cpant''} \sigma''_*$ and further by the second induction hypothesis that $\sigma \sim_{cpant'} \sigma'_*$.
- The type derivation is of the form

$$\frac{a_0 + a_1 \in cpant' \quad a_0 + a_1 \notin av \quad \begin{array}{c}\vdots\\ a_0 : av, cpant \longrightarrow av'', cpant'' \\ \hookrightarrow (nvd_0, a'_0)\end{array} \quad \begin{array}{c}\vdots\\ a_1 : av'', cpant'' \longrightarrow av', cpant' \\ \hookrightarrow (nvd_1, a'_1)\end{array}}{a_0 + a_1 : av, cpant \longrightarrow av' \cup \{a_0 + a_1\}, cpant' \hookrightarrow (nvd_0; nvd_1; nv(a_0 + a_1) := a'_0 + a'_1, nv(a_0 + a_1))} +2^{\text{opt}}_{\text{cpa}}$$

Again, we know that there must exist states σ_*^1 and σ_*^2 such that $\sigma_* \succ nvd_0 \rightarrow \sigma_*^1$, $\sigma_*^1 \succ nvd_1 \rightarrow \sigma_*^2$ and $\sigma_*^2 \succ nv(a_0 + a_1) := a_0' + a_1' \rightarrow \sigma_*'$, so that $\sigma_*' = \sigma_*^2[nv(a_0 + a_1) \mapsto [\![a_0' + a_1']\!]\sigma_*^2]$. From $\sigma \sim_{cpant} \sigma_*$ by the two induction hypotheses it follows that $[\![a_0]\!]\sigma = [\![a_0']\!]\sigma_*^1$ and $\sigma \sim_{cpant''} \sigma_*^1$ and further that $[\![a_1]\!]\sigma = [\![a_1']\!]\sigma_*^2$ and $\sigma \sim_{cpant'} \sigma_*^2$. Since nvd_1 redefines no normal variables and no auxiliary variables for expressions in av'' and since all auxiliary variables of a_0' are for expressions in av'', $[\![a_0']\!]\sigma_*^1 = [\![a_0']\!]\sigma_*^2$. Consequently, we see that $[\![a_0 + a_1]\!]\sigma = [\![a_0]\!]\sigma + [\![a_1]\!]\sigma = [\![a_0']\!]\sigma_*^2 + [\![a_1']\!]\sigma_*^2 = [\![a_0' + a_1']\!]\sigma_*^2 = [\![nv(a_0 + a_1)]\!]\sigma_*'$. This in combination with $\sigma \sim_{cpant'} \sigma_*^2$ and the fact that the only change from σ_*^2 to σ_*' concerns $nv(a_0 + a_1)$ also gives us that $\sigma \sim_{cpant'} \sigma_*'$.

- The type derivation is of the form

$$\frac{a_0 + a_1 \notin cpant' \quad a_0 + a_1 \notin av \quad \begin{array}{c}\vdots\\ a_0 : av, cpant \longrightarrow av'', cpant'' \hookrightarrow (nvd_0, a_0')\end{array} \quad \begin{array}{c}\vdots\\ a_1 : av'', cpant'' \longrightarrow av', cpant' \hookrightarrow (nvd_1, a_1')\end{array}}{a_0 + a_1 : av, cpant \longrightarrow av' \cup \{a_0 + a_1\}, cpant' \hookrightarrow (nvd_0; nvd_1; a_0' + a_1')} +_{3\,\mathrm{cpa}}^{\mathrm{opt}}$$

  We know that there must exist a state σ_*'' such that $\sigma_* \succ nvd_0 \rightarrow \sigma_*''$ and $\sigma_*'' \succ nvd_1 \rightarrow \sigma_*'$. From $\sigma \sim_{cpant} \sigma_*$ by the two induction hypotheses it follows that $[\![a_0]\!]\sigma = [\![a_0']\!]\sigma_*''$ and $\sigma \sim_{cpant''} \sigma_*''$ and further and $[\![a_1]\!]\sigma = [\![a_1']\!]\sigma_*'$ and $\sigma \sim_{cpant'} \sigma_*'$. Since nvd_1 redefines no normal variables and no auxiliary variables for expressions in av'' and since all auxiliary variables of a_0' are for expressions in av'', $[\![a_0']\!]\sigma_*'' = [\![a_0']\!]\sigma_*'$. Consequently, we see that $[\![a_0 + a_1]\!]\sigma = [\![a_0]\!]\sigma + [\![a_1]\!]\sigma = [\![a_0']\!]\sigma_*' + [\![a_1']\!]\sigma_*' = [\![a_0' + a_1']\!]\sigma_*'$.

The proof of (ii) is similar to that of (i). In particular, the case of $=_{\mathrm{cpa}}^{\mathrm{opt}}$ is analogous to the case of $+_{3\,\mathrm{cpa}}^{\mathrm{opt}}$.

(iii) We use induction on the structure of $s : av, cpant \longrightarrow av', cpant' \hookrightarrow s'$. We assume $\sigma \sim_{cpant} \sigma_*$ and $\sigma \succ s \rightarrow \sigma'$.

We consider the following cases:

- The type derivation is of the form

$$\frac{\begin{array}{c}\vdots\\ a : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd; a')\end{array}}{x := a : av, cpant \longrightarrow av' \setminus mod(x), cpant' \hookrightarrow nvd; x := a'} :=_{\mathrm{cpa}}^{\mathrm{opt}}$$

  The given semantic derivation must be of the form

$$\frac{}{\sigma \succ x := a \rightarrow \sigma[x \mapsto [\![a]\!]]}$$

  so $\sigma' = \sigma[x \mapsto [\![a]\!]\sigma]$. Let σ_*^1 be the unique state such that $\sigma_* \succ nvd \rightarrow \sigma_*^1$ and let $\sigma_*' = \sigma_*^1[x \mapsto [\![a']\!]\sigma_*^1]$, so we have a derivation of $\sigma_*^1 \succ x := a' \rightarrow \sigma_*'$. From the assumption $\sigma \sim_{cpant} \sigma_*$ by (i) we know that $[\![a]\!]\sigma = [\![a']\!]\sigma_*^1$ and $\sigma \sim_{cpant'} \sigma_*^1$. Consequently, $\sigma'(x) = [\![a]\!]\sigma = [\![a']\!]\sigma_*^1 = \sigma_*'(x)$. Further, since

$cpant' \subseteq av' \setminus mod(x)$, no expression in $cpant'$ contains x. Accordingly, no expression in $cpant'$ can change its value during the assignment $x := a'$ taking from σ_*^1 to σ_*'. Therefore, from the knowledge that $\sigma \sim_{cpant'} \sigma_*^1$ we may conclude that $\sigma' \sim_{cpant'} \sigma_*'$.

- The type derivation is of the form

$$\frac{\begin{array}{ccc} \vdots & \vdots & \vdots \\ b : av, cpant \longrightarrow av'', cpant'' & s_t : av'', cpant'' \longrightarrow av', cpant' & s_f : av'', cpant'' \longrightarrow av', cpant' \\ \hookrightarrow (nvd, b') & \hookrightarrow s_t' & \hookrightarrow s_f' \end{array}}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : av, cpant \longrightarrow av', cpant' \hookrightarrow nvd; \mathsf{if}\ b'\ \mathsf{then}\ s_t'\ \mathsf{else}\ s_f'}\ \mathrm{if}_{\mathrm{cpa}}^{\mathrm{opt}}$$

We have that either $\sigma \models b$ or $\sigma \not\models b$. In the first case, the given semantic derivation is of the form

$$\frac{\sigma \models b \quad \sigma \succ s_t \rightarrow \sigma'}{\sigma \succ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \rightarrow \sigma'}$$

Let σ_*'' be the unique state such that $\sigma_* \succ nvd \rightarrow \sigma_*''$. From $\sigma \sim_{cpant} \sigma_*$ by (ii) we learn that $[\![b]\!]\sigma = [\![b']\!]\sigma_*''$ and $\sigma \sim_{cpant''} \sigma_*''$. Thus we have the derivation

$$\frac{\sigma_*'' \models b' \quad \sigma_*'' \succ s_t' \rightarrow \sigma_*'}{\sigma_*'' \succ \mathsf{if}\ b'\ \mathsf{then}\ s_t'\ \mathsf{else}\ s_f' \rightarrow \sigma_*'}$$

where the premise is justified by the induction hypothesis (IH).

By induction hypothesis we get that $\sigma' \sim_{cpant'} \sigma_*'$.

Similar reasoning holds for $\sigma \not\models b$.

- The type derivation is of the form

$$\frac{b : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd; b') \quad s_t : av', cpant' \longrightarrow av, cpant \hookrightarrow s_t'}{\mathsf{while}\ b\ \mathsf{do}\ s_t : av, cpant \longrightarrow av', cpant' \hookrightarrow nvd; \mathsf{while}\ b'\ \mathsf{do}\ (s_t'; nvd)}\ \mathrm{while}_{\mathrm{cpa}}^{\mathrm{opt}}$$

We also invoke structural induction on the given semantic derivation of $\sigma \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \rightarrow \sigma'$. We have that either $\sigma \models b$ or $\sigma \not\models b$. In the first case, the given semantic derivation is of the form

$$\frac{\sigma \models b \quad \sigma \succ s_t \rightarrow \sigma'' \quad \sigma'' \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \rightarrow \sigma'}{\sigma \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \rightarrow \sigma'}$$

Let σ_*''' be the unique state such that $\sigma_* \succ nvd \rightarrow \sigma_*'''$. From $\sigma \sim_{cpant} \sigma_*$ by (ii) we infer that $[\![b]\!]\sigma = [\![b']\!]\sigma_*'''$ and $\sigma \sim_{cpant'} \sigma_*'''$. Thus we have the

derivation

$$\dfrac{\sigma'''_* \models b' \qquad \dfrac{\overset{\vdots}{\sigma'''_* \succ s'_t \to \sigma''_*} \qquad \overset{\vdots}{\sigma''_* \succ nvd \to \sigma''''_*}}{\sigma'''_* \succ s'_t; nvd \to \sigma''''_*} \qquad \overset{\vdots}{\sigma''''_* \succ \mathsf{while}\ b'\ \mathsf{do}\ (s'_t; nvd) \to \sigma'_*}}{\sigma'''_* \succ \mathsf{while}\ b'\ \mathsf{do}\ (s'_t; nvd) \to \sigma'_*}$$

(outer IH) (inner IH) (inner IH)

Here the outer hypothesis applies thanks to $\sigma \sim_{cpant'} \sigma'''_*$ and ensures the existence of σ''_* that also satisfies $\sigma'' \sim_{cpant} \sigma''_*$. Further, the inner hypothesis applies, taking care of the composition $nvd; \mathsf{while}\ b'\ \mathsf{do}\ (s'_t; nvd)$, and ensures the existence of σ'_* such that $\sigma' \sim_{cpant'} \sigma'_*$.

If $\sigma \not\models b$, then the semantic derivation must be

$$\dfrac{\sigma \not\models b}{\sigma \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \to \sigma}$$

i.e., $\sigma' = \sigma$. Let state σ'_* be the unique state such that $\sigma_* \succ nvd \to \sigma'_*$. From $\sigma \sim_{cpant} \sigma_*$ by (ii) we know that $[\![b]\!]\sigma = [\![b']\!]\sigma'_*$ and $\sigma \sim_{cpant'} \sigma'_*$. Thus we have the derivation

$$\dfrac{\sigma'_* \not\models b'}{\sigma'_* \succ \mathsf{while}\ b'\ \mathsf{do}\ (s'_t; nvd) \to \sigma'_*}$$

and we also know that $\sigma \sim_{cpant'} \sigma'_*$.

- The type derivation is of the form

$$\dfrac{av, cpant \le av_0, cpant_0 \quad s : av_0, cpant_0 \longrightarrow av'_0, cpant'_0 \hookrightarrow s' \quad av'_0, cpant'_0 \le av', cpant' \qquad \overset{\vdots}{}}{s : av, cpant \longrightarrow av', cpant' \hookrightarrow s'}\ \mathrm{conseq}^{\mathrm{opt}}_{\mathrm{cpa}}$$

We also have the given semantic judgement $\sigma \succ s \to \sigma'$. Since $cpant \supseteq cpant_0$, from $\sigma \sim_{cpant} \sigma_*$ we obtain that $\sigma \sim_{cpant_0} \sigma_*$. By the induction hypothesis, there must be a state σ'_* such that $\sigma_* \succ s' \to \sigma'_*$ and $\sigma' \sim_{cpant'_0} \sigma'_*$. Since $cpant'_0 \supseteq cpant'$, we have that $\sigma' \sim_{cpant'} \sigma'_*$. □

To state the corresponding theorem about the Hoare logic, we define $P|_{cpant}$ to abbreviate $P \wedge \bigwedge_{a \in cpant} a = nv(a)$ and $P|^{cpant}$ to mean $\exists[v(a) \mid a \notin cpant](P[a/nv(a) \mid a \in cpant][v(a)/nv(a) \mid a \notin cpant])$.

The theorem is the following:

Theorem 6
(o) If $av, cpant \le av', cpant'$, then $P|_{cpant} \models P|_{cpant'}$.
(i) If $a : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd, a')$, then
— $\{(P[a/w])|_{cpant}\}\ nvd\ \{(P|_{cpant'})[a'/w]\}$,
— $\{P\}\ nvd\ \{Q[a'/w]\}$ implies $P|^{cpant} \models (Q|^{cpant'})[a/w]$,

(ii) If $b : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd, b')$*, then*
— $\{(P[b/w])|_{cpant}\}\, nvd\, \{(P|_{cpant'})[b'/w]\}$*; it follows that* $\{P|_{cpant}\}\, nvd\, \{(b' \supset (b \wedge P)|_{cpant'}) \wedge (\neg b' \supset (\neg b \wedge P)|_{cpant'})\}$,
— $\{P\}\, nvd\, \{Q[b'/w]\}$ *implies* $P|^{cpant} \models (Q|^{cpant'})[b/w]$,
(iii) If $s : av, cpant \longrightarrow av', cpant' \hookrightarrow s'$*, then*
— $\{P\}\, s\, \{Q\}$ *implies* $\{P|_{cpant}\}\, s'\, \{Q|_{cpant'}\}$,
— $\{P\}\, s'\, \{Q\}$ *implies* $\{P|^{cpant}\}\, s\, \{Q|^{cpant'}\}$.

This theorem can again be proved non-constructively or constructively. We look at the constructive proof only.

PROOF. (o) Since $av, cpant \leq av', cpant'$ implies that $cpant \supseteq cpant'$, one gets $P|_{cpant} \models P|_{cpant'}$ by conjunction elimination.

Concerning (i)-(iii) we only prove the first halves. To save space, we do not show the entailment side conditions of the consequence rule.

(i) We use induction on the derivation of $a : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd, a')$. We restrict our attention to the following cases.

- The type derivation is of the form

$$\frac{a_0 + a_1 \in av \qquad \begin{array}{c}\vdots\\ a_0 : av, cpant \longrightarrow av'', cpant'' \\ \hookrightarrow (nvd_0, a_0')\end{array} \qquad \begin{array}{c}\vdots\\ a_1 : av'', cpant'' \longrightarrow av', cpant' \\ \hookrightarrow (nvd_1, a_1')\end{array}}{a_0 + a_1 : av, cpant \cup \{a_0 + a_1\} \longrightarrow av' \cup \{a_0 + a_1\}, cpant' \hookrightarrow (nvd_0; nvd_1, nv(a_0 + a_1))}\ +_{1\,\mathrm{cpa}}^{\mathrm{opt}}$$

  We have the following derivation.

$$\dfrac{\dfrac{\{(P[a_0 + a_1/w])|_{cpant \cup \{a_0+a_1\}}\}\, nvd_0; nvd_1\, \{(P|_{cpant'})[nv(a_0 + a_1)/w]\}}{\dfrac{\{(P[nv(a_0 + a_1)/w])|_{cpant}\}\, nvd_0\, \{(P|[nv(a_0 + a_1)/w])_{cpant''})\}}{\vdots\ \text{IH}} \qquad \vdots\ \text{IH}\ \{(P[nv(a_0 + a_1)/w])|_{cpant''}\}\, nvd_1\, \{(P[nv(a_0 + a_1)/w])|_{cpant'}\}}{\{(P[nv(a_0 + a_1)/w])|_{cpant}\}\, nvd_0; nvd_1\, \{(P[nv(a_0 + a_1)/w])|_{cpant'}\}}$$

  The entailment $(P[a_0 + a_1/w])|_{cpant \cup \{a_0+a_1\}} \models (P[nv(a_0 + a_1)/w])|_{cpant}$ holds as $(P[a_0 + a_1/w])|_{cpant \cup \{a_0+a_1\}} \models (P[a_0 + a_1/w])|_{cpant} \wedge a_0 + a_1 = nv(a_0 + a_1) \models (P[nv(a_0 + a_1)/w])|_{cpant}$ by substitution of equals for equals.
- The type derivation is of the form

$$\frac{a_0 + a_1 \in cpant' \quad a_0 + a_1 \notin av \qquad \begin{array}{c}\vdots\\ a_0 : av, cpant \longrightarrow av'', cpant'' \\ \hookrightarrow (nvd_0, a_0')\end{array} \qquad \begin{array}{c}\vdots\\ a_1 : av'', cpant'' \longrightarrow av', cpant' \\ \hookrightarrow (nvd_1, a_1')\end{array}}{a_0 + a_1 : av, cpant \longrightarrow av' \cup \{a_0 + a_1\}, cpant' \hookrightarrow (nvd_0; nvd_1; nv(a_0 + a_1) := a_0' + a_1', nv(a_0 + a_1))}\ +_{2\,\mathrm{cpa}}^{\mathrm{opt}}$$

We have the following derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{(P[a'_0+a'_1/w]|_{cpant'})[a'_0+a'_1/nv(a_0+a_1)]\}\, nv(a_0+a_1) := a'_0+a'_1\, \{(P[nv(a_0+a_1)/w]|_{cpant'})\}}}{\{P[a'_0+a'_1/w])|_{cpant'} \wedge a_0+a_1 = a'_0+a'_1\}\, nv(a_0+a_1) := a'_0+a'_1\, \{(P|_{cpant'})[nv(a_0+a_1)/w]\}} \quad \vdots\ \text{IH}}{\{((P[a'_0+w_1/w] \wedge a_0 = a'_0 \wedge a_1 = w_1)[a_1/w_1])|_{cpant''}\}\, nvd_1\, \{((P[a'_0+w_1/w] \wedge a_0 = a'_0 \wedge a_1 = w_1)|_{cpant'}[a'_1/w_1]\}} \quad \vdots\ \text{IH}}{\{((P[w_0+a_1/w] \wedge a_0 = w_0)[a_0/w_0])|_{cpant}\}\, nvd_0\, \{((P[w_0+a_1/w] \wedge a_0 = w_0)|_{cpant''})[a'_0/w_0]\}}}{\{(P[a_0+a_1/w])|_{cpant}\}\, nvd_0\, \{(P[a'_0+a_1/w] \wedge a_0 = a'_0)|_{cpant''}\}}}{\{(P[a_0+a_1/w])|_{cpant}\}\, nvd_0; nvd_1\, \{(P[a'_0+a'_1/w])|_{cpant'} \wedge a_0+a_1 = a'_0+a'_1\}}}{\{(P[a_0+a_1/w])|_{cpant}\}\, nvd_0; nvd_1; nv(a_0+a_1) := a'_0+a'_1\, \{(P|_{cpant'})[nv(a_0+a_1)/w]\}}$$

In this derivation, w_0, w_1 are picked fresh. The entailment $(P[a'_0+a'_1/w])|_{cpant'} \wedge a_0+a_1 = a'_0+a'_1 \models ((P[nv(a_0+a_1)/w])|_{cpant'})[a'_0+a'_1/nv(a_0+a_1)]$ holds because $(P[nv(a_0+a_1)/w])|_{cpant'} \Leftrightarrow (P[nv(a_0+a_1)/w])|_{cpant' \setminus \{a_0+a_1\}} \wedge a_0+a_1 = nv(a_0+a_1)$, so $(P[nv(a_0+a_1)/w]|_{cpant'})[a'_0+a'_1/nv(a_0+a_1)] \Leftrightarrow (P[a'_0+a'_1/w])|_{cpant' \setminus \{a_0+a_1\}} \wedge a_0+a_1 = a'_0+a'_1$.

- The type derivation is of the form

$$\dfrac{a_0+a_1 \notin cpant' \quad a_0+a_1 \notin av \quad \begin{array}{c}\vdots\\ a_0 : av, cpant \longrightarrow av'', cpant'' \hookrightarrow (nvd_0, a'_0)\end{array} \quad \begin{array}{c}\vdots\\ a_1 : av'', cpant'' \longrightarrow av', cpant' \hookrightarrow (nvd_1, a'_1)\end{array}}{a_0+a_1 : av, cpant \longrightarrow av' \cup \{a_0+a_1\}, cpant' \hookrightarrow (nvd_0; nvd_1; a'_0+a'_1)}\ +3^{\text{opt}}_{\text{cpa}}$$

We have this Hoare derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{(P[a'_0+a_1/w])|_{cpant''}\}\, nvd_1\, \{(P[a'_0+a'_1/w])|_{cpant'}\}} \quad \vdots\ \text{IH}}{\{((P[a'_0+w_1/w])[a_1/w_1])|_{cpant''}\}\, nvd_1\, \{((P[a'_0+w_1/w])|_{cpant'}[a'_1/w_1]\}} \quad \vdots\ \text{IH}}{\{((P[w_0+a_1/w])[a_0/w_0])|_{cpant}\}\, nvd_0\, \{((P[w_0+a_1/w])|_{cpant''})[a'_0/w_0]\}}}{\{(P[a_0+a_1/w])|_{cpant}\}\, nvd_0\, \{(P[a'_0+a_1/w])|_{cpant''}\}}}{\{(P[a_0+a_1/w])|_{cpant}\}\, nvd_0; nvd_1\, \{(P|_{cpant'})[a'_0+a'_1/w]\}}$$

The entailments are justified as in the previous case.

The proof of (ii) is similar to that of (i). In particular, the case of $=^{\text{opt}}_{\text{cpa}}$ is analogous to the case of $+3^{\text{opt}}_{\text{cpa}}$. Once the main statement has been established, the consequence $\{P|_{cpant}\}\, nvd\, \{(b' \supset (b \wedge P)|_{cpant'}) \wedge (\neg b' \supset (\neg b \wedge P)|_{cpant'})\}$ follows from the instance

$$\{(((w \supset (b \wedge P)) \wedge (\neg w \supset (\neg b \wedge P)))[b/w])|_{cpant}\}$$
$$nvd$$
$$\{(((w \supset (b \wedge P)) \wedge (\neg w \supset (\neg b \wedge P)))|_{cpant'})[b'/w]\}$$

by the entailments $P|_{cpant} \models ((b \supset (b \wedge P)) \wedge (\neg b \supset (\neg b \wedge P))|_{cpant} \equiv (((w \supset (b \wedge P)) \wedge (\neg w \supset (\neg b \wedge P)))[b/w])|_{cpant}$ and $(((w \supset (b \wedge P)) \wedge (\neg w \supset (\neg b \wedge$

$P)))|_{cpant'})[b'/w] \models ((w \supset (b \wedge P)|_{cpant'}) \wedge (\neg w \supset (\neg b \wedge P)|_{cpant'}))[b/w] \equiv (b' \supset (b \wedge P)|_{cpant'}) \wedge (\neg b' \supset (\neg b \wedge P)|_{cpant'})$.

(iii) We use induction on the derivation of $s : av, cpant \longrightarrow av', cpant' \hookrightarrow s'$. We consider the following cases.

- The type derivation is

$$\frac{a : av, cpant \longrightarrow av', cpant' \hookrightarrow (nvd; a')}{x := a : av, cpant \longrightarrow av' \setminus mod(x), cpant' \hookrightarrow nvd; x := a'} :=^{\text{opt}}_{\text{cpa}}$$

  The given Hoare derivation must be of the form

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

  This translates into the following Hoare derivation

$$\frac{\dfrac{\dfrac{\vdots\ \text{(ii)}}{\{(P[w/x][a/w])|_{cpant}\}\, nvd\, \{(P[w/x])|_{cpant'}[a'/w]\}} \quad \dfrac{}{\{P|_{cpant'}[a'/x]\}\, x := a'\, \{P|_{cpant'}\}}}{\{(P[a/x])|_{cpant}\}\, nvd\, \{P|_{cpant'}[w/x][a'/w]\}} \quad \dfrac{}{\{P|_{cpant'}[w/x][a'/w]\}\, x := a'\, \{P|_{cpant'}\}}}{\{P[a/x]|_{cpant}\}\, nvd; x := a'\, \{P|_{cpant'}\}}$$

  We have $(P[w/x])|_{cpant'}[a'/w] \models P|_{cpant'}[w/x][a'/w]$, since $cpant' \subseteq av' \setminus mod(x)$, so there are no expressions with x as a free variable in $cpant'$.
- The type derivation is of the form

$$\frac{b : av, cpant \longrightarrow av'', cpant'' \hookrightarrow (nvd, b') \qquad s_t : av'', cpant'' \longrightarrow av', cpant' \hookrightarrow s'_t \qquad s_f : av'', cpant'' \longrightarrow av', cpant' \hookrightarrow s'_f}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : av, cpant \longrightarrow av', cpant' \hookrightarrow nvd; \mathsf{if}\ b'\ \mathsf{then}\ s'_t\ \mathsf{else}\ s'_f}\ \text{if}^{\text{opt}}_{\text{cpa}}$$

  and the given Hoare derivation is of the form

$$\frac{\{b \wedge P\}\ \vdots\ s_t\, \{Q\} \qquad \{\neg b \wedge P\}\ \vdots\ s_f\, \{Q\}}{\{P\}\, \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f\, \{Q\}}\ .$$

  Let $P' = b' \supset (b \wedge P)|_{cpant''} \wedge \neg b' \supset (\neg b \wedge P)|_{cpant''}$. We can make the following derivation:

$$\frac{\dfrac{\vdots\ \text{(ii)}}{\{P|_{cpant}\}\, nvd\, \{P'\}} \qquad \dfrac{\dfrac{\vdots\ \text{IH}}{\{(b \wedge P)|_{cpant''}\}\, s_t\, \{Q|_{cpant'}\}} \quad \dfrac{\vdots\ \text{IH}}{\{(\neg b \wedge P)|_{cpant''}\}\, s_f\, \{Q|_{cpant'}\}}}{\{b' \wedge P'\}\, s_t\, \{Q|_{cpant'}\} \qquad \{\neg b' \wedge P'\}\, s_f\, \{Q|_{cpant'}\}}}{\{P|_{cpant}\}\, nvd; \mathsf{if}\ b'\ \mathsf{then}\ s'_t\ \mathsf{else}\ s'_f\, \{Q|_{cpant'}\}}\ .$$

- The type derivation is of the form

$$\frac{b : av, cpant \longrightarrow \vdots\ av', cpant' \hookrightarrow (nvd; b') \qquad s_t : av', cpant' \longrightarrow \vdots\ av, cpant \hookrightarrow s'_t}{\mathsf{while}\ b\ \mathsf{do}\ s_t : av, cpant \longrightarrow av', cpant' \hookrightarrow nvd; \mathsf{while}\ b'\ \mathsf{do}\ (s'_t; nvd)}\ \text{while}^{\text{opt}}_{\text{cpa}}$$

The given Hoare derivation must be of the form

$$\frac{\begin{array}{c}\vdots\\ \{b \wedge P\}\, s_t\, \{P\}\end{array}}{\{P\}\,\mathsf{while}\ b\ \mathsf{do}\ s_t\,\{\neg b \wedge P\}}$$

Let $P' \equiv b' \supset (b \wedge P)|_{cpant'} \wedge \neg b' \supset (\neg b \wedge P)|_{cpant'}$. The transformed Hoare derivation is

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\ \text{(ii)}\\ \{P|_{cpant}\}\, nvd\, \{P'\}\end{array}}{\{P|_{cpant}\}\, nvd; \mathsf{while}\ b'\ \mathsf{do}\ (s'_t; nvd)\, \{(\neg b \wedge P)|_{cpant'}\}} \quad \dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\ \text{IH}\\ \{(b \wedge P)|_{cpant'}\}\, s'_t\, \{P|_{cpant}\}\end{array}}{\{b' \wedge P'\}\, s'_t\, \{P|_{cpant}\}} \quad \begin{array}{c}\vdots\ \text{(ii)}\\ \{P|_{cpant}\}\, nvd\, \{P'\}\end{array}}{\{b' \wedge P'\}\, s'_t; nvd\, \{P'\}}}{\{P'\}\,\mathsf{while}\ b'\ \mathsf{do}\ (s'_t; nvd)\, \{\neg b' \wedge P'\}}}{\{P|_{cpant}\}\, nvd; \mathsf{while}\ b'\ \mathsf{do}\ (s'_t; nvd)\, \{(\neg b \wedge P)|_{cpant'}\}}$$

□

A Hoare proof for the program analyzed in Figure 9 is given in Figure 10.

$$\dfrac{\dfrac{\dfrac{}{\{u*v = u*v\}\, p := u*v\, \{p = u*v\}}}{\{u*v = c \wedge \top\}\, p := u*v\, \{p = u*v \vee z = p*q+r\}} \quad \dfrac{}{\{p*q+r = p*q+r\}\, z := p*q+r\, \{z = p*q+r\}}}{\{u*v \neq c \wedge \top\}\, z := p*q+r\, \{p = u*v \vee z = p*q+r\}}}{\{\top\}\ \mathsf{if}\ u*v = c\ \mathsf{then}\ p := u*v\ \mathsf{else}\ z := p*q+r\ \{p = u*v \vee z = p*q+r\}}$$

$$\dfrac{\dfrac{}{\{\top\}\ x := u*v+z\ \{\top\}} \quad \dfrac{}{\{\top\}\, z := 10\, \{\top\}}}{\{\top\}\ x := u*v+z; z := 10\ \{\top\}} \quad /$$

$$\{\top\}\ \begin{array}{l} x := u*v+z; z := 10; \\ \mathsf{if}\ u*v = c\ \mathsf{then}\ p := u*v\ \mathsf{else}\ z := p*q+r \end{array}\ \{p = u*v \vee z = p*q+r\}$$

Fig. 10. A proof of the example program

A Hoare proof for the optimized program is given in Figure 11. As can be seen from the example, the precondition of the program needs to be strengthened according to the given pretype. Similarly, assertions for the intermediate program points are also strengthened, according to the types.

4 Related Work

Concerning the use of type systems to describe data-flow analyses employed in optimizations, this paper builds directly on the work by Laud et al. [13], which gives a general method for casting monotone forward and backward data-flow analyses as type systems. This is similar to the flow logic work of Nielson and Nielson [17] continuing the annotated type systems thread. Closest is the work by Benton [4], which describes constant folding and dead code elimination as a

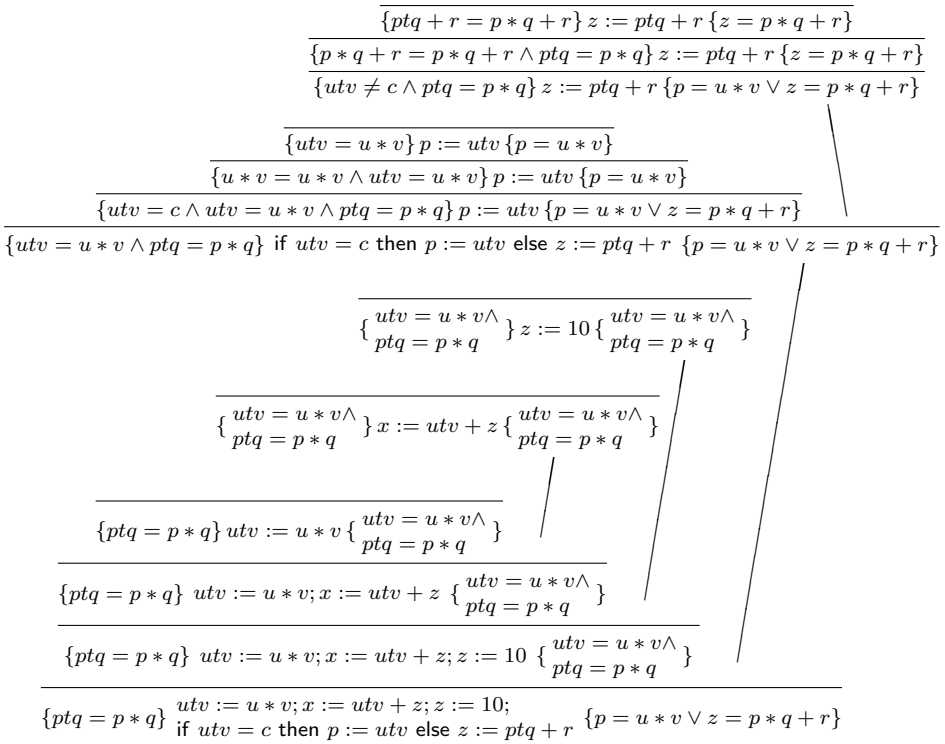

Fig. 11. Transformed proof

type system, but does not consider optimizations like common subexpression elimination where achieving compositionality is non-trivial. Benton also develops a relational Hoare logic to reason about pairs of programs. Differently from ours, this approach makes it possible to prove instances of the soundness of a program optimization (the semantic equivalence of a particular program and its the optimized form) within a program logic. But metatheoretic statements pertaining to an optimization as a transformation defined for all programs must still be proved as statements about the program logic, similarly to our work, where we have confined ourselves to the standard Hoare logic.

Data-flow analyses for safety policies admit similar descriptions by type systems. Applications in the context of certification have been explored for various basic safety policies for Java bytecode like languages, for (flow-insensitive and flow-sensitive) secure information flow, resource usage etc., see, e.g., [22,7,11,5,8]. The idea of certifying analysis results is central also in the work on abstraction-carrying code by Albert et al. [1], but rather than introducing type systems, they reason about the adequate abstract semantics directly.

There is a huge body of a work dedicated to proving optimization soundness, specifically also for imperative languages and data-flow analyses based optimizations, see, e.g., [12,14,6]. We are specifically interested in being able to extract from a general optimization soundness proof a soundness proof of the optimization of any given individual program (an instantiation of the gen-

eral soundness proof to this program), as is done in the translation validation work for optimizing compilers [15,24]. Transformation of functional correctness proofs as in the present paper has been studied by Barthe et al. [3]. Differently from us, they work with a wp-calculus rather than a Hoare logic and only allow similarity relations that strengthen equality of states on all program variables by further equations. As a consequence, existential relations (as the similarity relation in the case of live variables analysis) must be handled in ad hoc ways.

Aspinall et al. [2] require that an optimization not only preserves the semantics of a given program in appropriate sense, but also produces an improvement wrt. some quality measure, provably. This is possible, if one works with a resource-aware instrumented semantics or a corresponding Hoare logic to reason about the quality of programs, and their work explores exactly this avenue.

5 Conclusions and Future Work

We have described two different data-flow analyses based program optimizations as type systems, and also described the corresponding "proof optimizations" that follow from their soundness proofs. We believe these examples to be a convincing demonstration that type systems are a compact and, moreover, useful means for presenting results of data-flow analyses and program optimizations, specifically in certification applications. Specifically, we see several advantages in employing type systems.

Firstly and foremostly, type systems explain analyses and optimizations well. A set of declarative rules is easier to understand and believe than an algorithm. A program analysis and optimization algorithm can be explained as a type inference algorithm, but just in order to understand and believe a result it can deliver, it is actually unnecessary to know about the algorithm. A type derivation of a program can be compressed into an annotation of the program from where reconstruction of the full derivation is trivial. In the case of data-flow analyses, the important information is provided by the fixpoints that serve as loop invariants. If these are provided in an annotation, they do not have to be recomputed, one only needs to check that they are fixpoints indeed. Such annotations can thus serve as certificates in a proof-carrying-code like scenario where it is important to communicate checkable analysis results.

Further, a type-system presentation of an optimization also makes it easy to show that the optimization is generally sound, via a simple structural induction on type derivations. The constructive content of this proof is a transformation of evaluation derivations and that can be recast as a transformation of any given functional correctness proof of an original program into one of its

optimized form.

Also, as we have emphasized on the example of common subexpression elimination, links across the phrase structure of a program can be overcome with combined analyses that type systems can handle.

We plan to continue or are already continuing this work along several avenues [4]. First, we have developed a foundational approach to classical data-flow analyses, a program logic for reasoning about transition traces, into which the type systems for particular analyses can be embedded [9].

Second, we have demonstrated that our approach to type system descriptions also extends to data-flow analyses and program optimizations for low-level languages, in particular for stack-based languages (such as Java bytecode). Here, contrarily to what may seem, switching to a low-level language is not really challenging technically. One can build on the compositional approach to logics and type systems for low-level languages [18,19] (a piece of code is a single labelled instruction or basic block or a union of pieces of code) or on the popular non-compositional approach of Stata and Abadi [22] (a piece of code is a flat set of single labelled instructions or basic blocks). But the technical interest in the case of stack-based code is that even the simplest useful optimizations (load-pop pairs, store-reload pairs) require bidirectional analyses, unless special restrictive assumptions are made about the form of the code to be optimized. Accordingly, we have adapted the type system format to accommodate bidirectional analyses [20].

In relation to compilation from a high-level source language to a low-level language, we will also look at compilation of analysis-type derivations for program analyses that make sense both at the high and low level (e.g., live variables, copy propagation).

Third, we have started to look at type-systematic definability, soundness proofs and proof optimization for optimizations that restructure control-flow by moving code out of conditionals and loops, by a case study on partial redundancy elimination [21].

Fourth, it seems only plausible that type systems can also serve as a formalism for describing program obfuscations, for proving them sound and hence for obfuscating functional correctness proofs.

The final items on our agenda concern modularity of analyses and optimizations and improvement properties of optimizations. In this paper we proved common subexpression elimination sound in one monolithic step. To properly justify the use of our framework, this should ideally be done in three

[4] We worked on a number of the items listed below after this paper was submitted.

independent (and reusable) steps: by first proving our available expressions analysis sound, by then defining and proving conditional partial anticipability sound with respect to any sound available expressions analysis, and by finally defining and proving common subexpression elimination sound with respect to any sound conditional partial anticipability analysis. This approach is certainly feasible, but we would like to develop a systematic approach to arguments about cascaded analyses and optimizations. Likewise, we did not show that dead code elimination or common subexpression elimination improve programs. Program quality can be quantified in terms of measures computed "behind the scenes" by instrumented semantics and this paves a way toward reasoning about improvement properties.

Acknowledgements

We are grateful to Peter Thiemann, Bernd Fischer and David Sands for discussions and to the two anonymous referees of this paper for their very helpful remarks and suggestions. This work was supported by the Estonian Science Foundation grants no. 5567 and 6940 and the EU FP6 IST integrated project MOBIUS.

References

[1] E. Albert, G. Puebla and M. V. Hermenegildo, Abstraction-carrying code, in F. Baader and A. Voronkov, A. (Eds.), Proc. of 11th Int. Conf. on Logic for Programming, Artif. Intell. and Reasoning, LPAR 2004, Vol. 3452 of Lect. Notes in Artif. Intell., Springer-Verlag (2005), 380–397.

[2] D. Aspinall, L. Beringer and A. Momigliano, Optimisation validation, in J. Knoop, G. C. Necula, W. Zimmermann (Eds.), Proc. of 5th Int. Wksh. on Compiler Optimization Meets Compiler Verification, COCV 2006, v. 176(3) of Electron. Notes in Theor. Comput. Sci., Elsevier (2007), 37–59.

[3] G. Barthe, B. Grégoire, C. Kunz and T. Rezk, Certificate translation for optimizing compilers (extended abstract, in K. Yi (Ed.), Proc. of 13th Int. Symp. on Static Analysis, SAS 2006, Vol. 4134 of Lect. Notes in Comput. Sci., Springer-Verlag (2006), 301–317.

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

[5] L. Beringer and M. Hofmann, A. Momigliano and O. Shkaravska, Automatic certification of heap consumption, in F. Baader and A. Voronkov (Eds.),

Proc. of 11th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2004, Vol. 3452 of Lect. Notes in Artif. Intell., Springer-Verlag (2005), 347–362.

[6] Y. Bertot, B. Grégoire and X. Leroy, A structured approach to proving compiler optimizations based on dataflow analysis, in J.-C. Filliâtre, C. Paulin-Mohring and B. Werner (Eds.): Revised Selected Papers from 1st Int. Wksh. on Types for Proofs and Programs, TYPES 2004, Vol. 3839 of Lect. Notes in Comput. Sci., Springer-Verlag (2006), 66–81.

[7] F. Besson, T. Jensen and D. Pichardie, Proof-carrying code from certified abstract interpretation and fixpoint compression, Theor. Comput. Sci. **364**(3) (2006), pp. 273–291.

[8] D. Cachera, T. Jensen, D. Pichardie and G. Schneider, Certified memory usage analysis, in J. S. Fitzgerald, I. J. Hayes and A. Tarlecki (Eds.), Proc. of 2005 Symp. of Formal Methods Europe, FM 2005, Vol. 3582 of Lect. Notes in Comput. Sci., Springer-Verlag (2005), 91–106.

[9] 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, IEEE CS Press (2007), 107–116.

[10] C. A. R. Hoare, An axiomatic basis for computer programming, Commun. of ACM **12** (1969) 576–583.

[11] S. Hunt, D. Sands, On flow-sensitive security types, in Proc. of 33rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 2006, ACM Press (2006), 79–90.

[12] D. Lacey, N. D. Jones, E. Van Wyk and C. C. Frederiksen, Proving correctness of compiler optimizations by temporal logic, Higher-Order and Symb. Comput. **17**(3) (2004), 173–206.

[13] P. Laud, T. Uustalu and V. Vene, Type systems equivalent to data-flow analyses for imperative languages, Theor. Comput. Sci. **364**(3) (2006), 292–310.

[14] S. Lerner, T. Millstein and C. Chambers, Automatically proving the correctness of compiler optimizations, in Proc. of ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation, PLDI 2003, ACM Press (2003), 220–231.

[15] G. C. Necula, Translation validation for an optimizing compiler, in Proc. of 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2000 [= SIGPLAN Notices **35**(5)], ACM Press (2000), pp. 83–94.

[16] F. Nielson, H. R. Nielson, C. Hankin, Principles of Program Analysis, Springer-Verlag (1999).

[17] H. R. Nielson, F. Nielson, Flow logic: a multi-paradigmatic approach to static analysis, in T. Æ. Mogensen, D. A. Schmidt and I. H. Sudborough (Eds.), The

Essence of Computation, Complexity, Analysis, Transformation, Vol. 2566 of Lect. Notes in Comput. Sci., Springer-Verlag (2002), 223–244.

[18] A. Saabas and T. Uustalu, A compositional natural semantics and Hoare logic for low-level languages, Theor. Comput. Sci. **373**(3) (2007), 273–302.

[19] A. Saabas and T. Uustalu, Compositional type systems for stack-based low-level languages, in B. Jay and J. Gudmundssson (Eds.), Proc. of 12th Computing: Australasian Theory Symp., CATS 2006, Vol. 51 of Confs. in Research and Practice in Inform. Techn., Australian Comput. Soc. (2006), 27–39.

[20] A. Saabas and T. Uustalu, Type systems for optimizing stack-based code, in Proc. of 2nd Wksh. on Bytecode Semantics, Verification, Analysis and Transformation, BYTECODE 2007, Vol. 190(1) of Electron. Notes in Theor. Comput. Sci. (2007), 103–119.

[21] A. Saabas, T. Uustalu, Proof optimization for partial redundancy elimination, in Proc. of 2008 ACM SIGPLAN Wksh. on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 2008, ACM Press (2008), 91–101.

[22] R. Stata and M. Abadi, A type system for Java bytecode subroutines, ACM Trans. on Program. Lang. and Syst. **21**(1) (1999), 90–137.

[23] D. Volpano, G. Smith and C. Irvine, A sound type system for secure flow analysis, J. of Comput. Security **4**(2–3) (1996) 167–188.

[24] L. Zuck, A. Pnueli, Y. Fang and B. Goldberg, VOC: a methodology for the translation validation of optimizing compilers. J. of Univ. Comput. Sci. **9**(3) (2003), 223–247.

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 [10].

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

$$a ::= x \mid n \mid a_0 + a_1 \mid \ldots$$
$$b ::= a_0 = a_1 \mid \ldots \mid \mathrm{tt} \mid \mathrm{ff} \mid \neg b \mid \ldots$$
$$s ::= x := a \mid \mathsf{skip} \mid s_0; s_1 \mid \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \mid \mathsf{while}\ b\ \mathsf{do}\ s_t$$

$$\frac{}{\sigma \succ x := a \to \sigma[x \mapsto [\![a]\!]\sigma]}\ :=_{\mathrm{ns}}$$

$$\frac{}{\sigma \succ \mathsf{skip} \to \sigma}\ \mathrm{skip}_{\mathrm{ns}} \qquad \frac{\sigma \succ s_0 \to \sigma'' \quad \sigma'' \succ s_1 \to \sigma'}{\sigma \succ s_0; s_1 \to \sigma'}\ \mathrm{comp}_{\mathrm{ns}}$$

$$\frac{\sigma \models b \quad \sigma \succ s_t \to \sigma'}{\sigma \succ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \to \sigma'}\ \mathrm{if}^{\mathrm{tt}}_{\mathrm{ns}} \qquad \frac{\sigma \not\models b \quad \sigma \succ s_f \to \sigma'}{\sigma \succ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \to \sigma'}\ \mathrm{if}^{\mathrm{ff}}_{\mathrm{ns}}$$

$$\frac{\sigma \models b \quad \sigma \succ s_t \to \sigma'' \quad \sigma'' \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \to \sigma'}{\sigma \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \to \sigma'}\ \mathrm{while}^{\mathrm{tt}}_{\mathrm{ns}} \qquad \frac{\sigma \not\models b}{\sigma \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \to \sigma}\ \mathrm{while}^{\mathrm{ff}}_{\mathrm{ns}}$$

Fig. A.1. 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: $\mathbf{State} =_{\mathrm{df}} \mathbf{Store} =_{\mathrm{df}} \mathbf{Var} \to \mathbb{Z}$. The arithmetical and boolean expressions are interpreted relative to stores as integers and truth values by the semantic function $[\![-]\!] \in \mathbf{AExp} + \mathbf{BExp} \to \mathbf{Store} \to \mathbb{Z}$, defined in the denotational style by the usual equations. We write $\sigma \models b$ to say that $[\![b]\!]\sigma = \mathrm{tt}$.

Statements are interpreted via the evaluation relation $\succ - \to \subseteq \mathbf{State} \times \mathbf{Stm} \times \mathbf{State}$ defined inductively by the ruleset given in Figure A.1.

Lemma 7 (Determinacy) *If $\sigma \succ s \to \sigma'$ and $\sigma \succ s \to \sigma''$, 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 A.2. In the

$$\frac{}{\{Q[x \mapsto a]\}\, x := a\, \{Q\}}\ :=_{\text{hoa}}$$

$$\frac{}{\{P\}\, \mathsf{skip}\, \{P\}}\ \text{skip}_{\text{hoa}} \qquad \frac{\{P\}\, s_0\, \{R\} \quad \{R\}\, s_1\, \{Q\}}{\{P\}\, s_0; s_1\, \{Q\}}\ \text{comp}_{\text{hoa}}$$

$$\frac{\{b \wedge P\}\, s_t\, \{Q\} \quad \{\neg b \wedge P\}\, s_f\, \{Q\}}{\{P\}\, \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f\, \{Q\}}\ \text{if}_{\text{hoa}} \qquad \frac{\{b \wedge P\}\, s_t\, \{P\}}{\{P\}\, \mathsf{while}\ b\ \mathsf{do}\ s_t\, \{\neg b \wedge P\}}\ \text{while}_{\text{hoa}}$$

$$\frac{P \models P' \quad \{P'\}\, s\, \{Q'\} \quad Q' \models Q}{\{P\}\, s\, \{Q\}}\ \text{conseq}_{\text{hoa}}$$

Fig. A.2. Hoare rules of While

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

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