Foundational certification of data-flow analyses

Maria João Frade
Departamento de Informática
Universidade do Minho
Campus de Gualtar, P-4710-057 Braga, Portugal
Email: mjf@di.uminho.pt

Ando Saabas and Tarmo Uustalu
Institute of Cybernetics
Tallinn University of Technology
Akadeemia tee 21, EE-12618 Tallinn, Estonia
Email: {ando|tarmo}@cs.ioc.ee

Abstract

Data-flow analyses, such as live variables analysis, available expressions analysis etc., are usefully specifiable as type systems. These are sound and, in the case of distributive analysis frameworks, complete wrt. appropriate natural semantics on abstract properties. Applications include certification of analyses and "optimization" of functional correctness proofs alongside programs.

On the example of live variables analysis, we show that analysis type systems are applied versions of more foundational Hoare logics describing either the same abstract property semantics as the type system (liveness states) or a more concrete natural semantics on transition traces of a suitable kind (future defs and uses). The rules of the type system are derivable in the Hoare logic for the abstract property semantics and those in turn in the Hoare logic for the transition trace semantics. This reduction of the burden of trusting the certification vehicle can be compared to foundational proof-carrying code, where general-purpose program logics are preferred to special-purpose type systems and universal logic to program logics.

We also look at conditional liveness analysis to see that the same foundational development is also possible for conditional data-flow analyses proceeding from type systems for combined "standard state and abstract property" semantics.

Keywords: natural semantics, Hoare logics, type systems, data-flow analyses, program optimizations, certification of analyses and optimizations, applied vs. foundational

1 Introduction

In proof-carrying code [13], it is the responsibility of the code producer to produce evidence that the code shipped is safe and/or functionally correct. When code generation involves optimizations, an important useful intermediate mechanism is certification of the underlying program analyses, preferably based on a formalism rather than an informal mathematical theory.

It has been recognized, see, e.g., [15, 12], that classical data-flow analyses, such as live variables analysis, available expressions analysis etc., are usefully specifiable in a declarative way as *type systems* that may operate on source programs in a compositional (syntax-directed) manner, rather than on intermediate representations (such as flat control-flow graphs). These type systems make good formal vehicles for certification of analyses and can thus turn it very similar to customary certification of safety and functional correctness properties in program-logic like formalisms. A certificate is a typing derivation (or a typing judgement with sufficient additional information in the form of annotations to recover one), certificate checking is type(-derivation) checking and certificate generation amounts to principal type inference. The declarative character of type systems endows their use with additional value. For instance, there is no good reason for certificates to depend on an algorithmic definition of an analysis when only the certifier needs to produce analyses: the certificate checker should be able to check purported analyses based on a declarative definition (which, moreover, is probably more basic and thus easier to trust than any algorithmic one).

The analysis type systems are sound and, in the case of distributive analysis frameworks, complete wrt. appropriate *natural semantics* on abstract properties—a reformulation of the usual semantical justification of analyses.

In this paper, we shed further light on the type-systematic method by showing that analysis type systems are in fact *applied* versions of more *foundational* Hoare logics. These describe either the same property semantics as the type system (but without recourse to any ideas about approximations) or a more basic semantics on transition traces of a suitable kind and are therefore easier to trust. The applied formalisms are justifiable as sound wrt. the more foundational formalisms (and also their underlying semantics). This is analogous to foundational proof-carrying code [3], motivated by exactly the same idea of reducing the burden of trusting an applied formalism of certification by switching to a more foundational one. Moreover, we learn that not only can a textbook definition of an analysis be cast as a program-logic like formalism, but the same is possible for the more basic considerations that justify this definition.

These contributions are all based on the classical theory of monotone analysis frameworks and abstract interpretation [7, 8], but they demonstrate that the applied vs. foundational spectrum in proof-carrying code for safety or functional correctness carries over to certified optimization analyses. And they also emphasize that program analyses for optimizations are just as amenable to certification in program-logic like declarative formalisms as are functional correctness and safety.

Further, in addition to standard data-flow analyses, we also look at conditional analyses, based on a combined "standard state and abstract property" semantics, as a possible variation. We also comment briefly on type-systematic definition and justification of program optimizations and type-systematic "optimization" of functional correctness proofs alongside programs.

For the sake of brevity and intuitiveness of exposition, we limit our discussion to live variables analysis and dead code elimination for the WHILE language, but the approach is general and applies to a variety data-flow analyses and optimizations.

The paper is organized as follows. In Section 2, we introduce the method of defining data-flow analyses as type systems on the example of live variables analysis. We justify the type system by proving it sound and complete wrt. an appropriate natural semantics on liveness states and show that analyzing a program amounts to principal type inference. In Section 3, we show that the type system is an applied version of a Hoare logic for the same natural semantics. In Section 4, we define liveness of a variable as a predicate on def/use transition traces and justify the natural semantics and Hoare logic on liveness states in terms of a natural semantics and Hoare logic on def/use transition traces. In Section 6, we treat conditional liveness, to then proceed to a discussion of type-systematic program optimization on the example of dead code elimination in Section 6. Section 7 reviews some related work and Section 8 concludes.

2 A natural semantics and type system for live variables

We begin by an overview of the type-systematic approach to data-flow analyses [15, 12]. We do this on the example of live variables analysis (in Sec. 5, we also consider a variant, conditional liveness).

Informally, a variable is said to be live on a computation path, if it has a future useful use not preceded by a definition. A useful use is a use in an expression assigned to a live variable or a use in a guard expression. (This is the strong version of liveness, in contrast to the weaker one where any use triggers liveness.)

The textbook definition of live variables analysis and its justification, however, do not proceed directly from this definition but from derived considerations. The analysis (for source programs, not for the corresponding control-flow graphs) is justified by the following non-standard semantics, which we state as a natural (i.e., big-step) semantics.

States δ are assignments of values $\{\mathrm{dd}, \mathrm{ll}\}$, $\mathrm{dd} \sqsubseteq \mathrm{ll}$, to variables, understood as "liveness states". We define $\delta \sqsubseteq \delta'$ to mean that $\delta(y) \sqsubseteq \delta'(y)$ for any $y \in \mathbf{Var}$.

The evaluations of a statement are pairs of states (a prestate and a poststate) given by the rules in Figure 1, the notation $\delta \succ s \rightarrow \delta'$ meaning that δ and δ' are a possible pre- and poststate for s. The notation $\delta[x \mapsto v]$ stands for updating a state δ at a variable x with a value v and we also use a similar notation for simultaneous updates.

Intuitively, this semantics runs programs backwards. For any final liveness state, we get the initial liveness states corresponding to the computation paths the program can take. For example, the assignment rule, $:=_{\mathrm{lvns}}$, expresses that, if the lhs variable x is live in a poststate, then it becomes dead in the midstate (where the rhs has been evaluated but not assigned yet), because the assignment defines it, while all variables y of the rhs (including perhaps x as well) become live in the prestate, because they are usefully used. If x is dead in a poststate, the prestate is the same. The semantics is non-deterministic, as if- and while-statements can take multiple computation paths: liveness states fix no values for the variables.

A version that is deterministic (still in the backwards direction: for any poststate, there is exactly one prestate), the collecting semantics, is defined by

$$[\![s]\!](\delta') =_{\mathrm{df}} \bigsqcup \{\delta \mid \delta \succ s \rightarrow \delta'\}$$

The collecting semantics calculates the MOP ("meet over all paths") upper bound on the liveness prestates for a given liveness poststate.

The analysis (working with approximations) can be formulated as a type system. Types d are assignments of values from $\{\mathrm{dd}, \mathrm{ll}\}$, $\mathrm{dd} \sqsubseteq \mathrm{ll}$ to variables, just as states, but their pragmatics is different: they function as non upper bounds (over-approximations) of liveness states. The negation here is a formality that results from the analysis being backward, but validity of subtyping and typing being forward (for conformity with the standard definitions of validity; this design decision will be useful for us especially in Sec. 5). Ignoring this negation, the values dd and ll in types can be understood to mean "certainly dead" resp. "possibly live": a state is of a type, if all variables dead in the type are dead in the

$$\frac{}{\delta[x \mapsto \mathrm{dd}][y \mapsto \delta(y) \sqcup \delta(x) \mid y \in \mathrm{FV}(a)] \succ x := a \rightarrow \delta} :=_{\mathrm{lvns}}$$

$$\frac{}{\delta \succ \mathsf{skip} \rightarrow \delta}\ \mathrm{skip}_{\mathrm{lvns}} \qquad \frac{\delta \succ s_0 \rightarrow \delta' \quad \delta' \succ s_1 \rightarrow \delta''}{\delta \succ s_0; s_1 \rightarrow \delta''}\ \mathrm{comp}_{\mathrm{lvns}}$$

$$\frac{\delta \succ s_t \rightarrow \delta'}{\delta[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \succ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \rightarrow \delta'}\ \mathrm{if}^{\mathrm{tt}}_{\mathrm{lvns}} \qquad \frac{\delta \succ s_f \rightarrow \delta'}{\delta[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \succ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \rightarrow \delta'}\ \mathrm{if}^{\mathrm{ff}}_{\mathrm{lvns}}$$

$$\frac{\delta \succ s_t \rightarrow \delta' \quad \delta' \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \rightarrow \delta''}{\delta[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \rightarrow \delta''}\ \mathrm{while}^{\mathrm{tt}}_{\mathrm{lvns}} \qquad \frac{}{\delta[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \rightarrow \delta}\ \mathrm{while}^{\mathrm{ff}}_{\mathrm{lvns}}$$

Figure 1. Natural semantics for live variables

state (a variable live in the type can be both dead and live in the state).

The type system has one subtyping rule, reading

$$\frac{d' \sqsubseteq d}{d \leq d'}$$

The types of a statement are pairs of types (a pretype and a posttype): we write to $s : d \rightarrow d'$ to denote that d and d' are a possible pre- and posttype of s. The typing rules are in Figure 2. Note that while the assignment rule of the type system is similar to that in the semantics, the rules for if- and while-statements are different: a typing of a statement pertains to all computation paths of a statement, not just one. The while rule is similar to the while rule from standard Hoare logic by involving an invariant type. Likewise, the subsumption rule is an analogue of the consequence rule. The type system accepts all valid analyses of a program, not only the strongest one, so for the statement $s =_{\mathrm{df}}$ $\mathsf{if}\ w = 3\ \mathsf{then}\ x := y\ \mathsf{else}\ x := z$ and posttype $[w \mapsto \mathrm{dd}, x \mapsto \mathrm{ll}, y, z \mapsto \mathrm{dd}]$, both $[w \mapsto \mathrm{ll}, x \mapsto \mathrm{dd}, y, z \mapsto \mathrm{ll}]$ and $[w, x, y, z \mapsto \mathrm{ll}]$ are derivable as pretypes, but the former pretype corresponds to the strongest analysis.

To state and prove the type system adequate wrt. the semantics, we define $\delta \models d$ to mean $\delta \not\sqsubseteq d$ in agreement with the explanations above. Adequacy of subtyping ($d \leq d'$ iff d' being an upper bound of a state implies that d is also an upper bound) is trivial.

Theorem 1 (Soundness and completeness of subtyping) *$d \leq d'$ iff for any δ, $\delta \models d$ implies $\delta \models d'$ (i.e., $\delta \sqsubseteq d'$ implies $\delta \sqsubseteq d$).*

Soundness and completeness of typing ($s : d \longrightarrow d'$ iff d' being an upper bound on a poststate implies that d is an upper bound on the prestates) are proven separately.

Theorem 2 (Soundness of typing) *If $s : d \longrightarrow d'$, then, for any δ, δ' such that $\delta \succ s \rightarrow \delta'$, $\delta \models d$ implies $\delta' \models d'$ (i.e., $\delta' \sqsubseteq d'$ implies $\delta \sqsubseteq d$).*

Proof. By induction on $s : d \longrightarrow d'$ and subordinate induction on $\delta \succ s \rightarrow \delta'$ in the case $s = \mathsf{while}\ b\ \mathsf{do}\ s_t$. □

To prove completeness, we define a syntactic weakest pretype operator wpt:

$$\begin{aligned}
&\mathrm{wpt}(x := a, d') \\
&\quad =_{\mathrm{df}} \quad d'[x \mapsto \mathrm{dd}][y \mapsto d'(y) \sqcup d'(x) \mid y \in \mathrm{FV}(a)] \\
&\mathrm{wpt}(\mathsf{skip}, d') =_{\mathrm{df}} d' \\
&\mathrm{wpt}(s_0; s_1, d') =_{\mathrm{df}} \mathrm{wpt}(s_0, \mathrm{wpt}(s_1, d')) \\
&\mathrm{wpt}(\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f, d') \\
&\quad =_{\mathrm{df}} \quad (\mathrm{wpt}(s_t, d') \sqcup \mathrm{wpt}(s_f, d'))[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \\
&\mathrm{wpt}(\mathsf{while}\ b\ \mathsf{do}\ s_t, d') \\
&\quad =_{\mathrm{df}} \quad \nu(F) \text{ where} \\
&\qquad\qquad F(d) =_{\mathrm{df}} (\mathrm{wpt}(s_t, d) \sqcup d')[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)]
\end{aligned}$$

Here ν is the greatest (wrt. our subtyping $\leq$) fixpoint operation on monotone type transformers.

The wpt operator is the type-systematic formulation of an algorithm for computing the strongest analysis, i.e., the MFP ("maximal fixpoint") upper bound on the liveness prestates for a liveness poststate.

The following lemmata show that the wpt of a given type d' is a pretype of d', in the sense of typing, and greater than any semantic pretype of d'.

Lemma 1 $s : \mathrm{wpt}(s, d') \longrightarrow d'$.

Proof. By induction on s. □

Lemma 2 *If, for any δ, δ' such that $\delta \succ s \rightarrow \delta'$, $\delta \models d$ implies $\delta' \models d'$ (i.e., $\delta' \sqsubseteq d'$ implies $\delta \sqsubseteq d$), then $d \leq \mathrm{wpt}(s, d')$ (i.e., $\mathrm{wpt}(s, d') \sqsubseteq d$).*

Proof. Also by induction on s. □

Theorem 3 (Completeness of typing) *If, for any δ, δ' such that $\delta \succ s \rightarrow \delta'$, $\delta \models d$ implies $\delta' \models d'$ (i.e., $\delta' \sqsubseteq d'$ implies $\delta \sqsubseteq d$), then $s : d \longrightarrow d'$.*

Proof. Immediate from the two lemmata by the $\mathrm{conseq}_{\mathrm{lvts}}$ rule. □

From soundness and completeness we get that the collecting semantics and the weakest pretype agree perfectly, i.e., MOP=MFP.

$$\frac{}{x := a : d[x \mapsto \mathrm{dd}][y \mapsto d(y) \sqcup d(x) \mid y \in \mathrm{FV}(a)] \longrightarrow d} :=_{\mathrm{lvts}}$$

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

$$\frac{s_t : d \longrightarrow d' \quad s_f : d \longrightarrow d'}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : d[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \longrightarrow d'}\ \mathrm{if}_{\mathrm{lvts}} \qquad \frac{s_t : d \longrightarrow d[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)]}{\mathsf{while}\ b\ \mathsf{do}\ s_t : d[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \longrightarrow d}\ \mathrm{while}_{\mathrm{lvts}}$$

$$\frac{d \le d_0 \quad s : d_0 \longrightarrow d_0' \quad d_0' \le d'}{s : d \longrightarrow d'}\ \mathrm{conseq}_{\mathrm{lvts}}$$

Figure 2. Type system for live variables

Corollary 1 $[\![s]\!](\delta') \sqsubseteq \mathrm{wpt}(s, \delta')$.

Proof. By Lemma 1 and Thm. 2, $\delta \succ\!s\!\rightarrow \delta'$ yields $\delta \sqsubseteq \mathrm{wpt}(s, \delta')$ for any liveness state δ. □

Corollary 2 $\mathrm{wpt}(s, \delta') \sqsubseteq [\![s]\!](\delta')$

Proof. By Lemma 2. □

Lemma 2 and its consequences, including completeness of typing (Thm. 3) and MFP $\sqsubseteq$ MOP (Cor. 2), depend on the fact that the transfer functions of live variables analysis are distributive (preserve meets). They do not, for instance, hold for constant propagation, which fails to be distributive.

The following weaker semantics-independent property of the type system alone does not rest on distributivity (and holds thus also for non-distributive backward analyses): the wpt of a program wrt. a posttype d' is greater than any typing-sense pretype of d'. We already know that it also is a typing-sense pretype of d' (Lemma 1). In summary, the wpt is the principal pretype of d', in type systems jargon. And computing the strongest analysis is principal type inference.

Lemma 3 *If* $d_0' \le d'$*, then* $\mathrm{wpt}(s, d_0') \le \mathrm{wpt}(s, d')$.

Theorem 4 *If* $s : d \longrightarrow d'$*, then* $d \le \mathrm{wpt}(s, d')$.

Proof. By induction on $s : d \longrightarrow d'$, using Lemma 3 in some cases. (For live variables, one can also go via the semantics and get the theorem as an immediate corollary of Thm. 2 and Lemma 2.) □

3 A Hoare logic for live variables

While the type system is a description of the semantics, it is not a very direct one: the type system really relies on both the semantics and properties of upper bounds. Nor is the type system completely expressive; the only expressible assertions are negations of upper bound conditions. A more foundational formalism, which is also expressively complete, can be obtained by recasting the liveness semantics as a Hoare logic, in analogy with the Hoare logic characterization of the standard semantics.

The assertions of the Hoare logic are (generally open) formulae of the (first-order) theory of $(\{\mathrm{dd}, \mathrm{ll}\}, \sqsubseteq)$ over the signature with an extralogical constant $ls(x)$ (for the liveness value of x in the understood liveness state) for any program variable $x \in \mathbf{Var}$. The proof rules are in Figure 3. The notation $P[x \Mapsto a]$ denotes substituting the occurrences of x in P by a. Again the design is for the standard notion of validity, i.e., a forward implication: the precondition holding for a prestate implies that the postcondition holds for all possible poststates. Reversing the implication is possible by contraposition, i.e., by negating the two conditions.

The Hoare logic is adequate (both sound and complete) with respect to the intended interpretation, which has $[\![ls(y)]\!](\delta) =_{\mathrm{df}} \delta(y)$ (so that $ls(y)$ means the current liveness value of y).

Theorem 5 (Soundness) *If* $\{P\}\, s\, \{Q\}$*, then, for any* δ*,* δ' *such that* $\delta \succ\!s\!\rightarrow \delta'$*,* $\delta \models_\alpha P$ *implies* $\delta' \models_\alpha Q$ *for any valuation* α*.*

Lemma 4 $\{P\}\, s\, \{\mathrm{slp}(P, s)\}$ *(for a correctly defined strongest postcondition operator).*

Theorem 6 (Completeness) *If, for any* δ*,* δ' *such that* $\delta \succ\!s\!\rightarrow \delta'$*,* $\delta \models_\alpha P$ *implies* $\delta' \models_\alpha Q$ *for any valuation* α*, then* $\{P\}\, s\, \{Q\}$*.*

Note that since the domain of the intended interpretation of the language of assertions is a doubleton, this language is, in fact, essentially propositional. Hence the strongest postcondition operation is trivially definable. And completeness would still hold absolutely (as opposed to relatively to the level of completeness of an axiomatization of arithmetic), if we replaced the two entailment side conditions in the consequence rule with side conditions of deducibility in an appropriate proof system.

Clearly the Hoare logic is more foundational than the type system, as it formalizes the semantics directly. But this has the price that, generally, Hoare triple derivations are

$$\dfrac{}{\{P\}\, x := a\, \{(ls(x) = \mathrm{ll} \supset P[ls(y) \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(a)][ls(x) \mapsto \mathrm{dd}]) \wedge (ls(x) = \mathrm{dd} \supset P)\}} :=_{\mathrm{lvhoa}}$$

$$\dfrac{}{\{P\}\,\mathsf{skip}\,\{P\}}\ \mathrm{skip}_{\mathrm{lvhoa}} \qquad \dfrac{\{P\}\, s_0\, \{Q\} \quad \{Q\}\, s_1\, \{R\}}{\{P\}\, s_0; s_1\, \{R\}}\ \mathrm{comp}_{\mathrm{lvhoa}}$$

$$\dfrac{\{P[ls(y) \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)]\}\, s_t\, \{Q\} \quad \{P[ls(y) \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)]\}\, s_f\, \{Q\}}{\{P\}\, \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f\, \{Q\}}\ \mathrm{if}_{\mathrm{lvhoa}}$$

$$\dfrac{\{P[ls(y) \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)]\}\, s_t\, \{P\}}{\{P\}\, \mathsf{while}\ b\ \mathsf{do}\ s_t\, \{P[ls(y) \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)]\}}\ \mathrm{while}_{\mathrm{lvhoa}}$$

$$\dfrac{P \models P_0 \quad \{P_0\}\, s\, \{Q_0\} \quad Q_0 \models Q}{\{P\}\, s\, \{Q\}}\ \mathrm{conseq}_{\mathrm{lvhoa}}$$

Figure 3. Hoare logic for live variables

harder to construct than type derivations. For certification of analyses, however, this is unproblematic. Constructing a Hoare-logic derivation for a typing judgement (a purported analysis result) is no harder than constructing a type-system derivation: types admit a translation into Hoare-logic assertions and a Hoare-logic derivation of a translated typing judgement is mechanically obtainable from its type-system derivation, i.e., the translation of types extends to type derivations.

In agreement with the semantic meaning of types, a type d can be translated into the Hoare-logic assertion $[d] =_{\mathrm{df}} ls \not\sqsubseteq d$ (i.e., $\neg \bigwedge\{ls(y) \sqsubseteq d(y) \mid y \in \mathbf{Var}\}$). This translation preserves subtyping and typing.

Theorem 7 (Preservation of subtyping) *If $d \leq d'$ in the type system, then $[d] \models [d']$.*

Theorem 8 (Preservation of typing) *If $s : d \longrightarrow d'$ in the type system, then $\{[d]\}\, s\, \{[d']\}$ in the Hoare logic.*

Proof. A non-constructive indirect proof is immediate from soundness of the type system and completeness of the Hoare logic. An alternative constructive direct proof (by extending translation to type derivations) is by induction on $s : d \longrightarrow d'$. □

Of course the weakest preconditions in the Hoare logic are more precise than those in the type system. For the statement $s =_{\mathrm{df}}$ $\mathsf{if}\ w = 3\ \mathsf{then}\ x := y\ \mathsf{else}\ x := z$, for instance, we have that $\mathrm{wpt}(s, [w \mapsto \mathrm{dd}, x \mapsto \mathrm{ll}, y, z \mapsto \mathrm{dd}]) = [w \mapsto \mathrm{ll}, x \mapsto \mathrm{dd}, y, z \mapsto \mathrm{ll}]$ while

$$\begin{array}{rl} & \mathrm{wlp}(s, \neg(ls(w) = \mathrm{dd} \wedge ls(y) = \mathrm{dd} \wedge ls(z) = \mathrm{dd})) \\ = & \neg(ls(w) = \mathrm{ll} \wedge ls(x) = \mathrm{dd} \\ & \quad \wedge ((ls(y) = \mathrm{ll} \wedge ls(z) = \mathrm{dd}) \vee (ls(z) = \mathrm{ll} \wedge ls(y) = \mathrm{dd}))) \end{array}$$

(for a poststate where w, y, z are dead, the type system can only detect that x is dead in the prestate (w, y, z can be either dead or live), while the Hoare logic knows also that w is live and that exactly one of y and z is live).

4 A natural semantics and Hoare logic for future defs, uses

Our discussion of live variables analysis thus far has been quite detached from the (informal) definition of liveness we recalled in Sec. 2. Instead we built on a semantics on liveness states, which looks very different. In fact, the definition of liveness is part of a foundation for this semantics, but we did not show this. Now we will close the gap.

The only observation needed is that liveness states are an abstraction over another, more concrete (and thus more basic) non-standard notion of states. There is nothing like liveness states or a semantics for them "in the nature". Instead, the liveness definition speaks about "computation paths", more specifically, about future definitions and uses of variables on such paths. As no more information about paths is relevant for liveness, we can confine our interest to future transition traces, where the transitions are defs and uses.

Thus we introduce a natural semantics where states are lists of tokens D_x with $x \in \mathbf{Var}$ and U^x_V with $V \subseteq \mathbf{Var}$ and $x \in \mathbf{Var} + \{pc\}$. A token D_x means a definition of x. A token U^x_V means a use of the variables V for defining x. The pseudovariable pc (for "program counter") is for the case where the variables V are used to evaluate a guard. Lists of such tokens should be understood as defs and uses to take place in the future.

The evaluation rules of the semantics are in Figure 4. ("·" stands for both "cons" and "append".) Again the causality is backward, but compared to the rules of the liveness semantics, they are entirely basic. The assignment rule, for instance, tells us that the unique preagenda for a postagenda τ is $\mathrm{U}^x_{\mathrm{FV}(a)} \cdot \mathrm{D}_x \cdot \tau$, i.e., to use all variables of a, to define x and then to do τ.

Using traces as states, it is straightforward to formalize the intuitive definition of liveness. For a trace τ, the corresponding liveness state $\mathsf{LS}(\tau)$ is defined as follows.

$$\begin{array}{l} \mathsf{LS}(\tau)(z) = \mathrm{ll} \\ \quad \text{iff} \quad \exists \upsilon, \tau', x, V.\, \tau = \upsilon \cdot \mathrm{U}^x_V \cdot \tau' \wedge z \in V \end{array}$$

$$\frac{}{\mathrm{U}^{x}_{\mathrm{FV}(a)} \cdot \mathrm{D}_x \cdot \tau \succ x := a \to \tau}\ :=_{\mathrm{trns}} \qquad \frac{}{\tau \succ \mathsf{skip} \to \tau}\ \mathrm{skip}_{\mathrm{trns}} \qquad \frac{\tau \succ s_0 \to \tau' \quad \tau' \succ s_1 \to \tau''}{\tau \succ s_0; s_1 \to \tau''}\ \mathrm{comp}_{\mathrm{trns}}$$

$$\frac{\tau \succ s_t \to \tau'}{\mathrm{U}^{pc}_{\mathrm{FV}(b)} \cdot \tau \succ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \to \tau'}\ \mathrm{if}^{\mathrm{tt}}_{\mathrm{trns}} \qquad \frac{\tau \succ s_f \to \tau'}{\mathrm{U}^{pc}_{\mathrm{FV}(b)} \cdot \tau \succ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \to \tau'}\ \mathrm{if}^{\mathrm{ff}}_{\mathrm{trns}}$$

$$\frac{\tau \succ s_t \to \tau' \quad \tau' \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \to \tau''}{\mathrm{U}^{pc}_{\mathrm{FV}(b)} \cdot \tau \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \to \tau''}\ \mathrm{while}^{\mathrm{tt}}_{\mathrm{trns}} \qquad \frac{}{\mathrm{U}^{pc}_{\mathrm{FV}(b)} \cdot \tau \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \to \tau}\ \mathrm{while}^{\mathrm{ff}}_{\mathrm{trns}}$$

Figure 4. Natural semantics for future def/use traces

$$\wedge((\exists \tau''.\, \tau' = \mathrm{D}_x \cdot \tau'' \wedge \mathsf{LS}(\tau'')(x) = \mathrm{ll}) \vee x = pc)$$

This definition is wellformed, because it can be reorganized into the following structurally recursive one.

$$\begin{aligned}
\mathsf{LS}(\varepsilon)(z) &=_{\mathrm{df}} \mathrm{dd} \\
\mathsf{LS}(\mathrm{D}_x \cdot \tau)(z) &=_{\mathrm{df}} \begin{cases} \mathrm{dd} & \text{if } z = x \\ \mathsf{LS}(\tau)(z) & \text{otherwise} \end{cases} \\
\mathsf{LS}(\mathrm{U}^x_V \cdot \mathrm{D}_x \cdot \tau)(z) &=_{\mathrm{df}} \begin{cases} \mathsf{LS}(\tau)(z) \sqcup \mathsf{LS}(\tau)(x) & \text{if } z \in V \\ \mathsf{LS}(\mathrm{D}_x \cdot \tau)(z) & \text{otherwise} \end{cases} \\
\mathsf{LS}(\mathrm{U}^{pc}_V \cdot \tau)(z) &=_{\mathrm{df}} \begin{cases} \mathrm{ll} & \text{if } z \in V \\ \mathsf{LS}(\tau)(z) & \text{otherwise} \end{cases}
\end{aligned}$$

The abstraction function puts the semantics of liveness states into perfect agreement with the trace semantics. Hence it is right to say that the trace semantics and the abstraction function are a foundation for the liveness state semantics.

Theorem 9 *If $\tau \succ s \to \tau'$, then $\mathsf{LS}(\tau) \succ s \to \mathsf{LS}(\tau')$.*

Theorem 10 *If $\delta \succ s \to \mathsf{LS}(\tau')$, then there is a trace τ such that $\tau \succ s \to \tau'$ and $\mathsf{LS}(\tau) = \delta$.*

Similarly to the liveness state semantics, the trace semantics is also characterizable by a Hoare logic, by a completely analogous design. The assertions are (open) formulae of the (first-order) theory of lists of def, use tokens over the signature with an extralogical constant tr for the current def-use future. The inference rules are in Figure 5.

Again the logic is sound and complete for the obvious intended interpretation of the assertions where $[\![tr]\!](\tau) =_{\mathrm{df}} \tau$ (i.e., tr denotes the current trace). But completeness is only relative to the completeness level of an axiomatization of the theory of lists, if the entailments in the side conditions of the consequence rule are replaced by deducibilities: the incompleteness of axiomatizations of arithmetic applies.

Theorem 11 (Soundness) *If $\{P\}\, s\, \{Q\}$, then, for any τ, τ' such that $\tau \succ s \to \tau'$, $\tau \models_\alpha P$ implies $\tau' \models_\alpha Q$ for any valuation α.*

Theorem 12 (Completeness) *If, for any τ, τ' such that $\tau \succ s \to \tau'$, $\tau \models_\alpha P$ implies $\tau' \models_\alpha Q$ for any valuation α, then $\{P\}\, s\, \{Q\}$.*

Just as constructing proofs in the Hoare logic for liveness in general was harder than constructing type derivations in the type system for liveness, constructing proofs in the Hoare logic for traces is even harder, generally. But again, if we have a proof for a triple in the Hoare logic for liveness, the more foundational proof in the Hoare logic for traces is obtainable mechanically. The assertions of the Hoare logic for liveness can be translated into ones of the Hoare logic for traces and the translation extends to derivations. Define LS to be a syntactic version of LS (so that $[\![LS(t)]\!](\tau) = \mathsf{LS}([\![t]\!](\tau))$). An assertion P about the current liveness state is naturally translated as the assertion $[P] =_{\mathrm{df}} P[ls \Mapsto LS(tr)]$ about the current trace. This translation preserves derivable triples.

Theorem 13 (Preservation of derivable Hoare triples) *If $\{P\}\, s\, \{Q\}$ in the Hoare logic for live variables, then $\{[P]\}\, s\, \{[Q]\}$ in the Hoare logic for traces.*

Proof. A non-constructive indirect proof is immediate from soundness of the Hoare logic for live variables, Thm. 9 and completeness of the Hoare logic for traces. An alternative constructive direct proof is by induction on $\{P\}\, s\, \{Q\}$. □

For analyses other than live variables analysis, the notion of a trace considered need not be suitable. For available expressions, for instance, it suffices to keep track of past evaluations and modifications of non-trivial expressions on a computation path. A more universal notion would record all past and future "atomic actions" (which are assignments and evaluations of guards).

5 Conditional liveness

Having outlined the foundational spectrum of certification for live variables analysis, we now sketch a variant, much to illustrate the flexibility of our setup. Namely, we look at conditional liveness à la Strom and Yellin [19]. This dwells on the same definition of liveness on a computation path as before, but the states in the underlying concrete semantics are pairs of a store (standard state) and a computation path, so only these transitions between computation

$$\frac{}{\{P\}\, x := a\, \{P[tr \Mapsto \mathrm{U}^{x}_{\mathrm{FV}(a)} \cdot \mathrm{D}_x \cdot tr]\}}\ :=_{\mathrm{trhoa}} \qquad \frac{}{\{P\}\, \mathsf{skip}\, \{P\}}\ \mathrm{skip}_{\mathrm{trhoa}} \qquad \frac{\{P\}\, s_0\, \{Q\} \quad \{Q\}\, s_1\, \{R\}}{\{P\}\, s_0; s_1\, \{R\}}\ \mathrm{comp}_{\mathrm{trhoa}}$$

$$\frac{\{P[tr \Mapsto \mathrm{U}^{pc}_{\mathrm{FV}(b)} \cdot tr]\}\, s_t\, \{Q\} \quad \{P[tr \Mapsto \mathrm{U}^{pc}_{\mathrm{FV}(b)} \cdot tr]\}\, s_f\, \{Q\}}{\{P\}\, \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f\, \{Q\}}\ \mathrm{if}_{\mathrm{trhoa}} \qquad \frac{\{P[tr \Mapsto \mathrm{U}^{pc}_{\mathrm{FV}(b)} \cdot tr]\}\, s_t\, \{P\}}{\{P\}\, \mathsf{while}\ b\ \mathsf{do}\ s_t\, \{P[tr \Mapsto \mathrm{U}^{pc}_{\mathrm{FV}(b)} \cdot tr]\}}\ \mathrm{while}_{\mathrm{trhoa}}$$

$$\frac{P \models P_0 \quad \{P_0\}\, s\, \{Q_0\} \quad Q_0 \models Q}{\{P\}\, s\, \{Q\}}\ \mathrm{conseq}_{\mathrm{trhoa}}$$

Figure 5. Hoare logic for future def/use traces

paths are considered that are physically possible. Accordingly, the analysis is finer.

The analogue to the semantics in Sec. 2 has as states pairs (σ, δ) of stores (assignments of integers to variables) and liveness states. The evaluation rules are in Figure 6.

The type system defining the analysis has as types d assignments to variables of assertions of the standard Hoare logic (i.e., arithmetic formulae over a signature with an extralogical constant x for any variable x). There is one subtyping rule

$$\frac{d' \models d}{d \leq d'}$$

The typing rules are in Figure 7. In these rules, we have written $d' \models d$ to mean that $d'(y) \models d(y)$ for all $y \in \mathbf{Var}$, $d[x \Mapsto a]$ to mean $[y \mapsto d(y)[x \Mapsto a] \mid y \in \mathbf{Var}]$, $b \supset d$ to mean $[y \mapsto b \supset d(y) \mid y \in \mathbf{Var}]$ and $d_t \wedge d_f$ to mean $[y \mapsto d_t(y) \wedge d_f(y) \mid y \in \mathbf{Var}]$.

Adequacy (soundness and completeness) of the type system wrt. the semantics holds wrt. the intended interpretation of types, which is: $\sigma, \delta \models d$ iff it is <u>not</u> the case that, for all variables $y \in \mathbf{Var}$, $\delta(y) = \mathrm{ll}$ implies $\sigma \models d(y)$ (i.e., $d(y)$ is a necessary condition for y being live). (Again the negation is formal: the analysis is backward, but the validity notion definition is forward, hence the need for contraposition.) The type $\top$ (verum) thus corresponds to the type ll ("possibly live") of the unconditional type system of Sec. 2 while $\bot$ (falsum) is dd ("certainly dead").

The strongest analysis algorithm is described by the weakest pretype (wpt) operator defined as follows:

$$\begin{array}{l}
\mathrm{wpt}(x := a, d') \\
\quad =_{\mathrm{df}}\ (d'[x \mapsto \bot][y \mapsto d'(y) \vee d'(x) \mid y \in \mathrm{FV}(a)])[x \Mapsto a] \\
\mathrm{wpt}(\mathsf{skip}, d')\ =_{\mathrm{df}}\ d' \\
\mathrm{wpt}(s_0; s_1, d')\ =_{\mathrm{df}}\ \mathrm{wpt}(s_0, \mathrm{wpt}(s_1, d')) \\
\mathrm{wpt}(\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f, d') \\
\quad =_{\mathrm{df}}\ ((b \wedge \mathrm{wpt}(s_t, d')) \vee (\neg b \wedge \mathrm{wpt}(s_f, d'))) \\
\hfill [y \mapsto \top \mid y \in \mathrm{FV}(b)] \\
\mathrm{wpt}(\mathsf{while}\ b\ \mathsf{do}\ s_t, d')\ =_{\mathrm{df}}\ \nu(F)\ \text{where} \\
\qquad F(d) =_{\mathrm{df}}\ ((b \wedge \mathrm{wpt}(s_t, d)) \vee (\neg b \wedge d')) \\
\hfill [y \mapsto \top \mid y \in \mathrm{FV}(b)]
\end{array}$$

Type derivations in this type system can be translated into proofs in a Hoare logic. The assertions P are formulae of the two-sorted theory of both the integers and the liveness domain $(\{\mathrm{dd}, \mathrm{ll}\}, \sqsubseteq)$ over the signature with an integer constant y and liveness constant $ls(y)$ for any variable y. The proof rules are in Figure 8. A type d is translated as dictated by the interpretation of types, namely by the assertion $\neg \bigwedge \{ls(y) = \mathrm{ll} \supset d(y) \mid y \in \mathbf{Var}\}$.

We refrain from spelling out the semantics and Hoare logic for store and transition trace pairs.

6 Dead code elimination

As an example of a data-flow analysis based optimization, let us look at dead code elimination. This optimization removes assignments to dead variables. Type-systematically, it is straightforwardly defined by a transformational add-on to the analysis type system of Fig. 2. The rules of the extended type system are in Fig. 9. For any type assigned to a program, this also assigns a transformed program.

This type-systematic definition of the optimization allows for a simple proof of relational soundness of the analysis. Defining $\sigma \sim_d \sigma'$ to denote that $\sigma(x) = \sigma'(x)$ for any $x \in \mathbf{Var}$ such that $d(x) = \mathrm{ll}$ (i.e., that two states agree on all variables live in a type), the optimization is sound in the following sense: If $s : d \longrightarrow d' \hookrightarrow s_*$ and $\sigma \sim_d \sigma_*$, then (i) $\sigma \succ s \rightarrow \sigma'$ implies that there exists σ'_* such that $\sigma' \sim_{d'} \sigma'_*$ and $\sigma_* \succ s_* \rightarrow \sigma'_*$, (ii) $\sigma_* \succ s_* \rightarrow \sigma'_*$ implies that there exists σ' such that $\sigma' \sim_{d'} \sigma'_*$ and $\sigma \succ s \rightarrow \sigma'$. The proof is by induction on $s : d \longrightarrow d' \hookrightarrow s_*$. Moreover, the type-derivation based program transformation can be extended to a transformation of functional correctness proofs [4, 16].

Conditional liveness analysis facilitates more refined dead code elimination optimization. For assignment, one could, for instance, choose to give these rules:

$$\frac{d(x) \not\models \bot}{\begin{array}{l} x := a : \\ (d[x \mapsto \bot][y \mapsto d(y) \vee d(x) \mid y \in \mathrm{FV}(a)])[x \Mapsto a] \longrightarrow d \\ \hfill \hookrightarrow x := a \end{array}}$$

$$\frac{d(x) \models \bot}{x := a : d[x \Mapsto a] \longrightarrow d \hookrightarrow \mathsf{skip}}$$

$$\frac{}{\sigma, \delta[x \mapsto \mathrm{dd}][y \mapsto \delta(y) \sqcup \delta(x) \mid y \in \mathrm{FV}(a)] \succ x := a \rightarrow \sigma[x \mapsto [\![a]\!]\sigma], \delta}\ :=_{\mathrm{lvns}}$$

$$\frac{}{\sigma, \delta \succ \mathsf{skip} \rightarrow \sigma, \delta}\ \mathrm{skip}_{\mathrm{lvns}} \qquad \frac{\sigma, \delta \succ s_0 \rightarrow \sigma', \delta' \quad \sigma', \delta' \succ s_1 \rightarrow \sigma', \delta''}{\sigma, \delta \succ s_0; s_1 \rightarrow \sigma'', \delta''}\ \mathrm{comp}_{\mathrm{lvns}}$$

$$\frac{\sigma \models b \quad \sigma, \delta \succ s_t \rightarrow \sigma', \delta'}{\sigma, \delta[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \succ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \rightarrow \sigma', \delta'}\ \mathrm{if}^{\mathrm{tt}}_{\mathrm{lvns}} \qquad \frac{\sigma \not\models b \quad \sigma, \delta \succ s_f \rightarrow \sigma', \delta'}{\sigma, \delta[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \succ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f \rightarrow \sigma', \delta'}\ \mathrm{if}^{\mathrm{ff}}_{\mathrm{lvns}}$$

$$\frac{\sigma \models b \quad \sigma, \delta \succ s_t \rightarrow \sigma', \delta' \quad \sigma', \delta' \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \rightarrow \sigma', \delta''}{\sigma, \delta[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \rightarrow \sigma'', \delta''}\ \mathrm{while}^{\mathrm{tt}}_{\mathrm{lvns}} \qquad \frac{\sigma \not\models b}{\sigma, \delta[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \succ \mathsf{while}\ b\ \mathsf{do}\ s_t \rightarrow \sigma, \delta}\ \mathrm{while}^{\mathrm{ff}}_{\mathrm{lvns}}$$

Figure 6. Natural semantics for conditional liveness

$$\frac{}{x := a : (d[x \mapsto \bot][y \mapsto d(y) \vee d(x) \mid y \in \mathrm{FV}(a)])[x \Mapsto a] \longrightarrow d}\ :=_{\mathrm{lvts}}$$

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

$$\frac{s_t : b \supset d \longrightarrow d' \quad s_f : \neg b \supset d \longrightarrow d'}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : d[y \mapsto \top \mid y \in \mathrm{FV}(b)] \longrightarrow d'}\ \mathrm{if}_{\mathrm{lvts}} \qquad \frac{s_t : b \supset d \longrightarrow d[y \mapsto \top \mid y \in \mathrm{FV}(b)]}{\mathsf{while}\ b\ \mathsf{do}\ s_t : d[y \mapsto \top \mid y \in \mathrm{FV}(b)] \longrightarrow \neg b \supset d}\ \mathrm{while}_{\mathrm{lvts}}$$

$$\frac{d \leq d_0 \quad s : d_0 \longrightarrow d_0' \quad d_0' \leq d'}{s : d \longrightarrow d'}\ \mathrm{conseq}_{\mathrm{lvts}}$$

Figure 7. Type system for conditional liveness

$$\frac{}{\{P[x \mapsto a]\}\ x := a\ \{(\mathit{ls}(x) = \mathrm{ll} \supset P[\mathit{ls}(y) \Mapsto \mathrm{ll} \mid y \in \mathrm{FV}(a)][\mathit{ls}(x) \Mapsto \mathrm{dd}]) \wedge (\mathit{ls}(x) = \mathrm{dd} \supset P)\}}\ :=_{\mathrm{lvhoa}}$$

$$\frac{}{\{P\}\ \mathsf{skip}\ \{P\}}\ \mathrm{skip}_{\mathrm{lvhoa}} \qquad \frac{\{P\}\ s_0\ \{Q\} \quad \{Q\}\ s_1\ \{R\}}{\{P\}\ s_0; s_1\ \{R\}}\ \mathrm{comp}_{\mathrm{lvhoa}}$$

$$\frac{\{b \wedge P[\mathit{ls}(y) \Mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)]\}\ s_t\ \{Q\} \quad \{\neg b \wedge P[\mathit{ls}(y) \Mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)]\}\ s_f\ \{Q\}}{\{P\}\ \mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f\ \{Q\}}\ \mathrm{if}_{\mathrm{lvhoa}}$$

$$\frac{\{b \wedge P[\mathit{ls}(y) \Mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)]\}\ s_t\ \{P\}}{\{P\}\ \mathsf{while}\ b\ \mathsf{do}\ s_t\ \{\neg b \wedge P[\mathit{ls}(y) \Mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)]\}}\ \mathrm{while}_{\mathrm{lvhoa}}$$

$$\frac{P \models P_0 \quad \{P_0\}\ s\ \{Q_0\} \quad Q_0 \models Q}{\{P\}\ s\ \{Q\}}\ \mathrm{conseq}_{\mathrm{lvhoa}}$$

Figure 8. Hoare logic for conditional liveness

$$\frac{d(x) = \mathrm{ll}}{x := a : d[x \mapsto \mathrm{dd}][y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(a)] \longrightarrow d \hookrightarrow x := a}\ :=^1{}_{\mathrm{lvopt}} \qquad \frac{d(x) = \mathrm{dd}}{x := a : d \longrightarrow d \hookrightarrow \mathsf{skip}}\ :=^2{}_{\mathrm{lvopt}}$$

$$\frac{}{\mathsf{skip} : d \longrightarrow d \hookrightarrow \mathsf{skip}}\ \mathrm{skip}_{\mathrm{lvopt}} \qquad \frac{s_0 : d \longrightarrow d' \hookrightarrow s_0' \quad s_1 : d' \longrightarrow d'' \hookrightarrow s_1'}{s_0; s_1 : d \longrightarrow d'' \hookrightarrow s_0'; s_1'}\ \mathrm{comp}_{\mathrm{lvopt}}$$

$$\frac{s_t : d \longrightarrow d' \hookrightarrow s_t' \quad s_f : d \longrightarrow d' \hookrightarrow s_f'}{\mathsf{if}\ b\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f : d[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \longrightarrow d' \hookrightarrow \mathsf{if}\ b\ \mathsf{then}\ s_t'\ \mathsf{else}\ s_f'}\ \mathrm{if}_{\mathrm{lvopt}}$$

$$\frac{s_t : d \longrightarrow d[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \hookrightarrow s_t'}{\mathsf{while}\ b\ \mathsf{do}\ s_t : d[y \mapsto \mathrm{ll} \mid y \in \mathrm{FV}(b)] \longrightarrow d \hookrightarrow \mathsf{while}\ b\ \mathsf{do}\ s_t'}\ \mathrm{while}_{\mathrm{lvopt}}$$

$$\frac{d \leq d_0 \quad s : d_0 \longrightarrow d_0' \hookrightarrow s' \quad d_0' \leq d'}{s : d \longrightarrow d' \hookrightarrow s'}\ \mathrm{conseq}_{\mathrm{lvopt}}$$

Figure 9. Type system for dead code elimination

A more aggressive optimization could be built on conditional liveness information together with conditional constant propagation information.

Consider this example (b does not contain z, y and x; the pretypes for all constituent statements have been computed from the given global posttype on line 8, where we have chosen that z may be live and y, x must be dead).

```
1 $[z \mapsto b, y \mapsto \neg b, x \mapsto \bot]$  $x := 5;$
2 $[z \mapsto b, y \mapsto \neg b, x \mapsto \bot]$  if $b$ then
3 $[z \mapsto b, y \mapsto \bot, x \mapsto \neg b]$      $y := x$
                                                   else
4 $[z \mapsto b, y \mapsto \neg b, x \mapsto \bot]$      skip;
5 $[z \mapsto b, y \mapsto \neg b, x \mapsto \bot]$  if $b$ then
6 $[z \mapsto \top, y \mapsto \bot, x \mapsto \bot]$      $z := z + 1$
                                                   else
7 $[z \mapsto \bot, y \mapsto \top, x \mapsto \bot]$      $z := y$
8 $[z \mapsto \top, y \mapsto \bot, x \mapsto \bot]$
```

Because of path sensitivity, the assignment to x on line 1 is eliminated by the rules above, since x is necessarily dead after it. The assignment to y on line 3 is conditionally live and could be removed on the basis of additional forward analysis information that its liveness postcondition $\neg b$ cannot actually obtain.

Alternative designs would optimize partially dead assignments by code motion, e.g., move an assignment preceding an if into one branch of the if-statement.

7 Related work

We can only mention some items of related work. Use of type systems to define program analyses is an old idea, especially in the form of enrichments of standard type systems ("annotated types", especially for functional languages). The type systems philosophy is also central in the "flow logic" work of Nielsen and Nielsen [15]. Type-systematic accounts of classical data-flow analyses for optimizations (incl. soundness of optimizations and optimization of functional correctness proofs, cf. translation validation) for imperative languages have been given by Benton and the present authors [4, 12, 16]. Volpano et al.'s well-known type system [20] for secure information flow is of the same kind, but relatively weak (based on invariant state types instead of pre- and poststate types).

The idea of characterizing non-standard semantics with program logics is old as well. For imperative languages, it appears already, e.g., in Andrews and Reitman's Hoare logic for secure information flow [2] and H. R. Nielson's Hoare logic for computation time [14]. Denney and Fischer [9] have characterized safety policies with Hoare logics and applied these to certification of safety.

The conditional data-flow analyses à la Strom and Yellin [19] are an extension of Strom and Yemini's typestate checking paradigm [18], originally meant to address basic safety.

Proof-carrying code was invented by Necula and Lee [13], who certified safety of programs against a verification condition generator. Foundationalism in PCC was pioneered by Appel [3], who suggested using a universal logic formalization of the underlying semantics instead. Hamid, Zhao et al. [10] proposed that a foundational certificate can consist in a type derivation and a formal soundness proof of the type system.

Formal certification of data-flow analyses, especially for Java bytecode, is the subject of a number of recent works. Albert et al.'s analysis-carrying code [1] highlights that analysis results can serve as analysis certificates allowing for lightweight checking (without fixpoint recomputation, only fixpoint checking). Besson, Jensen et al. [6] have taken a more foundational approach, certifying also soundness of analysis algorithms and addressing the issue of minimizing certificate size. Also foundational is the work by Beringer, Hofmann et al. [5] on certified heap consumption analysis based on a type system specializing a program logic (considerably more similarly to this paper).

8 Conclusions and future work

We have shown that classical data-flow analyses, such as live variables analysis, can be certified on a variety of levels, completely analogous to certification of program safety or functional correctness. To accept a typing derivation in an analysis type system as a certificate of a computed analysis, one must believe in the textbook definition of the analysis (this is what the type system formalizes) or in a justification of this definition. To accept a derivation in the corresponding Hoare logic for abstract properties, it suffices to trust the definition of the abstract property semantics (which is also the justification for the type system, but only together with additional ideas about approximations). Finally, to accept a derivation in the Hoare logic for future def/use traces, even this is not necessary: one must only believe in the basic definition of the trace semantics and in the definition of the abstraction of traces into liveness states. How much prerequisite trust is required depends on the application, but a formalism is available for each level.

In the case of more foundational formalisms, one can ask what makes better certificates: direct derivations in the foundational formalism (obtainable from derivations in an applied formalism by translation) or derivations in an applied formalism accompanied by a proof of soundness of the applied formalism wrt. the foundational one. It is also an option to avoid trusting any program logics by relying instead on the descriptions of their underlying program semantics and universal (meta)logic. The program logics and program semantics we have presented here support the full spectrum of foundationalism.

As future work, we plan to take a closer look into type system definitions of bidirectional analyses, such as partial redundancy elimination [11]. We are also specifically interested in the foundational spectrum of certification of analyses for stack-based low-level languages such as Java bytecode, in continuation of the initial work we did on type systems for optimizing stack-based code [17].

Acknowledgements We are grateful to Bernd Fischer for a very useful discussion. This work was supported by the Estonian Science Foundation under grants No. 5567 and 6940 and by the EU FP6 IST project No. 15905 MOBIUS.

References

[1] E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-carrying code. In *Proc. of 11th Int. Conf. on Logics for Programming, Artif. Intell. and Reasoning, LPAR 2004*, v. 3452 of *Lect. Notes in Artif. Intell.*, pp. 380–397. Springer, 2005.

[2] G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM Trans. on Program. Lang. and Syst.*, 2(1):56–76, 1980.

[3] A. Appel. Foundational proof-carrying code. In *Proc. of 16th Ann. IEEE Symp. on Logic in Computer Science, LICS 2001*, pp. 247–256. IEEE CS Press, 2001.

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

[5] L. Beringer, 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, Artif. Intell., and Reasoning, LPAR 2004*, v. 3452 of *Lect. Notes in Artif. Intell.*, pp. 347–362. Springer, 2005.

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

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs. In *Conf. Record of 4th ACM Symp. on Principles of Program. Lang., POPL '77*, pp. 238–252. ACM Press, 1977.

[8] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conf. Record of 6th ACM Symp. on Principles of Program. Lang., POPL '79*, pp. 269–282. ACM Press, 1979.

[9] E. Denney and B. Fischer. Correctness of source-level safety policies. In K. Araki, S. Gnesi, and D. Mandrioli, eds., *Proc. of 2003 Int. Symp. of Formal Methods Europe, FME 2003*, v. 2805 of *Lect. Notes in Comput. Sci.*, pp. 894–913. Springer, 2003.

[10] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. *J. of Autom. Reasoning*, 31(3–4): 191–229, 2004.

[11] U. P. Khedker and D. M. Dhamdhere. A generalized theory of bit vector data flow analysis. *ACM Trans. on Program. Lang. and Syst.*, 16(5):1472–1511, 1994.

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

[13] G. C. Necula. Proof-carrying code. In *Proc. of 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 1997*, pp. 106–119. ACM Press, 1997.

[14] H. R. Nielson. A Hoare-like proof system for analysing the computation time of programs. *Sci. of Comput. Program.*, 9:107–136, 1987.

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

[16] A. Saabas and T. Uustalu. Program and proof optimizations with type systems. Submitted, 2006.

[17] A. Saabas and T. Uustalu. Type systems for optimizing stack-based code. To appear in *Proc. of 2nd Int. Wksh. on Bytecode Semantics, Verification, Analysis and Transformation, Bytecode 2007*, *Electr. Notes in Theor. Comput. Sci.*.

[18] R. E. Strom and S. Yemini. Typestate: a programming language concept for enhancing software reliability. *IEEE Trans. on Softw. Engin.*, 12(1):157-171, 1986.

[19] R. E. Strom and D. M. Yellin. Extended typestate checking using conditional liveness analysis. *IEEE Trans. on Softw. Engin.*, 19(5):487-485, 1993.

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