Type Systems Equivalent to Data-Flow Analyses for Imperative Languages ⋆

Peeter Laud [a,*] Tarmo Uustalu [b] Varmo Vene [a,b]

[a] *Dept. of Computer Science, University of Tartu, J. Liivi 2, EE-50409 Tartu, Estonia*

[b] *Institute of Cybernetics at Tallinn University of Technology, Akadeemia tee 21, EE-12618 Tallinn, Estonia*

Received 13 January 2006; accepted 17 March 2006

Abstract

We show that a large class of data-flow analyses for imperative languages are describable as type systems in the following technical sense: possible results of an analysis can be described in a language of types so that a program checks with a type if and only if this type is a supertype of the result of applying the analysis. Type-checking is easy with the help of a certificate that records the "eureka"-bits of a typing derivation. Certificate-assisted type-checking amounts to a form of lightweight analysis à la Rose. For secure information flow, we obtain a type system that is considerably more precise than that of Volpano et al., but not more sophisticated. Importantly, our type systems are compositional.

Key words: Data-Flow Analyses, Secure Information Flow, Type Systems

1 Introduction

Both program analyses and type systems can be used to infer and state properties of programs. Clearly, program analyses are more algorithmic by their nature while type systems are more declarative. In particular, although program analyses are convenient tools for inferring properties of programs, type

⋆ An extended abstract of this paper appeared in the informal proceedings of the APPSEM '05 workshop.

∗ Corresponding author

Email addresses: `peeter_l@ut.ee` (Peeter Laud), `tarmo@cs.ioc.ee` (Tarmo Uustalu), `varmo@cs.ut.ee` (Varmo Vene).

systems have advantages over them, when it comes to communicating the properties established.

Given these considerations, it is most natural to wish to relate program analyses and type systems. This idea has been around for quite some time. In particular, Cousot has shown that both are instances of abstract interpretation [1] and there are several works describing how to derive type systems from program analysis definitions [2,3]. But the project has not been carried out really satisfactorily for imperative languages. The main obstacle seems to have been a confusion over what a type system for an imperative language should look like.

In this paper, we tackle exactly the problem of deriving type systems from analyses for imperative languages (more specifically, data-flow analyses) and solve it based on a simple philosophical idea of what types are all about. Our position is that a type must always be an abstraction of the denotation of a program, be it then an abstraction of its denotation with respect to the standard semantics of language or some non-standard semantics. And a type system must be a compositional way of ascribing types to programs such that type checking is "easy" (and type inference "not too hard", preferably). Since imperative programs denote transformations of (standard or non-standard) states — a state before a run of the program is sent to the state after it —, types for imperative programs must be *implications* between abstractions of (standard or non-standard) states, i.e., pairs of abstract states. (Compare this to Hoare logic, which is based on one instance of this idea: "types" of programs are implications between predicates on states in the sense that, for any legal prestate-poststate pair, if the the prestate satisfies the antecedent predicate, then the poststate satisfies the succedent predicate). We show that this quite basic idea leads to very simple type systems with smooth meta-theory which, we hope, will find applications in the area of certified software.

Our approach is in strong contrast to works where authors have chosen types for imperative programs to be single abstract states, e.g., the classic work by Volpano et al. on secure information flow [4], where the type of a program is an assignment of invariant security classes to its variables. They may have drawn their intuitions from the fact that, in many languages, the data-types of the variables of a program are required to stay invariant during its runs. But data-types and languages where variables cannot migrate from one data-type to another are more of an exception than a rule in the big picture of imperative languages and notions of abstract state. Most kinds of abstract states or types (such as which variables are live etc.) do change during runs. Because of the better notion of type, our type systems allow us to infer and state sharper properties of programs than type systems based on the idea that a type is a single abstract state.

More technically, our achievement and contribution in this paper is the following. We introduce a framework for defining data-flow analyses that caters, besides the most classical data-flow analyses such as available expressions or very busy expressions, also abstract interpretation inspired analyses such as constant propagation, and analyses based on non-standard semantics of conditional constructs such as secure information flow. We then introduce a method for deriving a type system from an analysis defined in the framework and show that the type system obtained is always equivalent to the analysis. What the method does is very simple: in fact we only introduce two schematic type systems, one for all forward may-analyses and one for all backward must-analyses definable in the framework. This does not restrict generality, since any forward must-analysis (resp. backward may-analysis) is trivially turned into a forward may-analysis (resp. backward must-analysis) by reversing the partial order of the abstract state space of the analysis. A type system for a concrete analysis is obtained by mechanical instantiation of the appropriate schematic type system with the definition of the analysis. We look at a number of example analyses, most prominently secure information flow. We also discuss extraction of certificates (essential information) from typing derivations and lightweight, certificate-assisted type-checking that should be useful in the venture of certified code.

The paper is structured as follows. In Section 2, we introduce our framework for defining data-flow analyses. In Section 3, we present our method of deriving a type system from the definition of an analysis. In Section 4, we substantiate our development with examples. In Section 5, we discuss certificate generation from type derivations and certificate-assisted type-checking. Section 6 overviews the related work and Section 7 presents our conclusions.

2 A Framework for Data-Flow Analyses

We consider a simple imperative programming language (WHILE-language). Its *statements* (or, *programs*) are given by the following grammar:

$$P ::= p \mid P_1; P_2 \mid \mathit{if}\ b\ \mathit{then}\ P_1\ \mathit{else}\ P_2 \mid \mathit{while}\ b\ \mathit{do}\ P_1$$

Here p is a primitive statement and b a Boolean expression (whose evaluation does not change the state of the program), similar to [3]. Primitive statements may be assignments or *skip*-statements; their details do not concern us here. Let **Stm** be the set of all programs.

Data-flow analyses work by calculating an assignment of abstract states to the edges of a control-flow graph. Depending on whether the analysis is forward or backward, either the direct or inverse control-flow graph of a given program is used and the calculation happens by propagating abstract states across the nodes of the graph in the appropriate direction.

We consider two types of data-flow analyses: forward may-analyses (based on least fixedpoint computation on the direct control-flow graph of a program) and backward must-analyses (based on greatest fixedpoint computation on the inverse control-flow diagram). The reasons to choose these two kinds as primitive will be explained in the next section. Forward must-analyses and backward may-analyses are variants obtained by reversing the partial order on the abstract state space.

Both types of analyses proceed from an *abstract state space* $(D, \sqsubseteq)$ that is a partial order. For forward may-analyses, we require furthermore that $(D, \sqsubseteq)$ is an ω-complete upper semilattice, i.e.

- any two elements d, d' have a least upper bound $d \sqcup d'$;
- any chain $\{d_n\}_{n\in\mathbb{N}}$, where $d_n \in D$ and $d_n \sqsubseteq d_{n+1}$ for all $n \in \mathbb{N}$, has a least upper bound $\bigsqcup_{n\in\mathbb{N}} d_n$.

We assume that each primitive statement p is given a denotation $[\![p]\!] : D \to D$ (sending an abstract state before running p to the abstract state after). The function $[\![p]\!]$ must be monotone and upward ω-continuous. Also, two denotations $[\![b]\!]_{\mathsf{t}}, [\![b]\!]_{\mathsf{f}} : D \to D$ are associated with each boolean expression b. If d is an abstract state before evaluating b, then $[\![b]\!]_{\mathsf{t}}(d)$ [resp. $[\![b]\!]_{\mathsf{f}}(d)$] correspond to the abstract states after evaluating to true, resp. false. After evaluating b the result of the evaluation is known and these denotations can add that knowledge to their arguments. The functions $[\![b]\!]_{\mathsf{t}}$ and $[\![b]\!]_{\mathsf{f}}$ must be monotone and upward ω-continuous as well. Often they are also reductive (i.e., $[\![b]\!]_{\mathsf{t}}(d), [\![b]\!]_{\mathsf{f}}(d) \sqsubseteq d$).

For backward must-analyses, the abstract state space $(D, \sqsubseteq)$ must be an ω-complete lower semilattice. Again we must have functions $[\![p]\!]$, $[\![b]\!]_{\mathsf{t}}$, $[\![b]\!]_{\mathsf{f}} : D \to D$, this time sending an abstract state after running p or evaluating b to true or false to the abstract state before. All these must be monotone and downward ω-continuous.

A *control-flow graph* G is always a tuple $(V, ar, \ell, E, e^{\mathrm{i}}, e^{\mathrm{f}}, \sigma, \tau)$ where V is a set of *nodes*, $ar : V \to \mathbb{N}$ gives the *indegree* of a node (the number of incoming edges), $\ell : (v : V) \to (D^{ar(v)} \to D)$ maps a node to a corresponding *transfer function*, E is a set of *edges*, $e^{\mathrm{i}}, e^{\mathrm{f}} \in E$ are two distinguished edges called the *initial and final edge*, respectively, $\sigma : E\backslash\{e^{\mathrm{i}}\} \to V$ gives the *source* node of an edge and $\tau : E\backslash\{e^{\mathrm{f}}\} \to V \times \mathbb{N}$ gives the *target* node of an edge together with its *sequence number* among the incoming edges of that node. Note that the initial edge does not have a source and the final edge does not have a target. The function τ is injective—merging of abstract values may only be done by the transfer functions. Hence there exists an inverse partial function $\tau^{-1} : V \times \mathbb{N} \rightharpoonup E\backslash\{e^{\mathrm{f}}\}$.

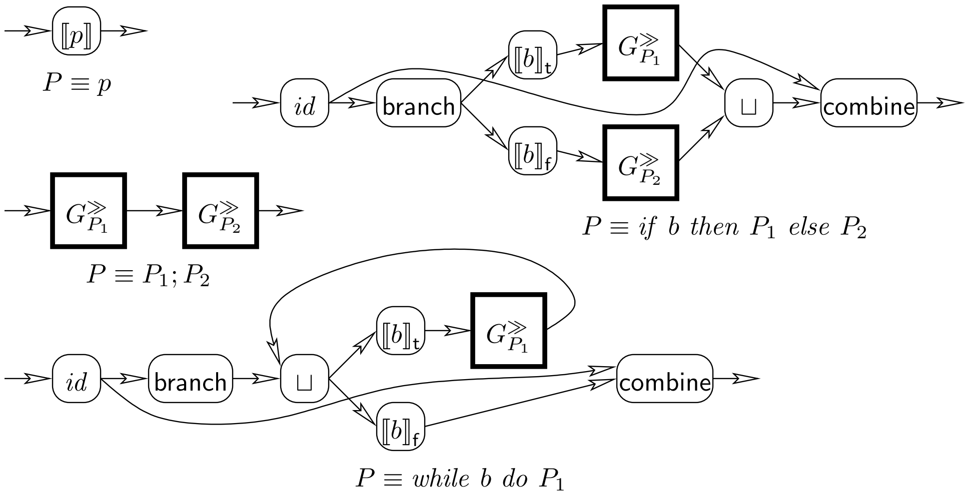

Fig. 1. Schematic control-flow graph $G^{\gg}_P$ of a program P for forward may-analyses

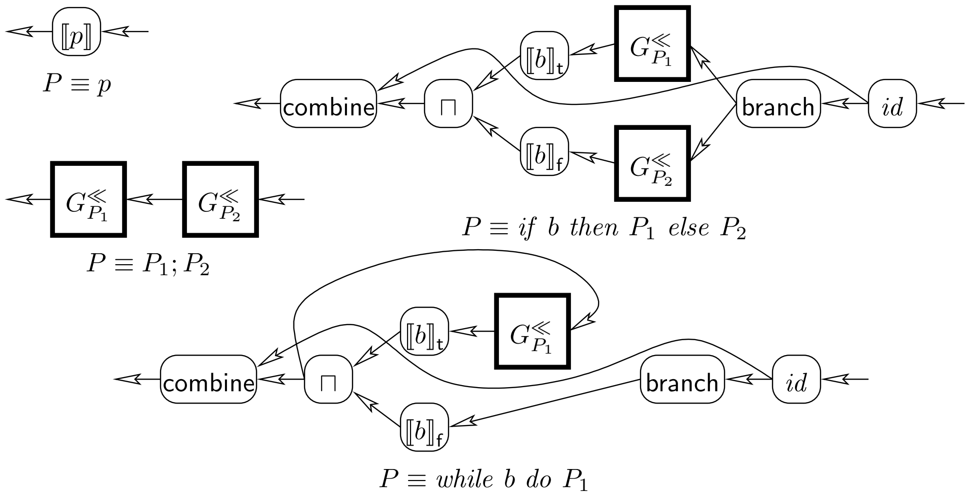

Fig. 2. Schematic control-flow graph $G^{\ll}_P$ of a program P for backward must-analyses

The *direct* and *inverse* control-flow graphs $G^{\gg}_P$ and $G^{\ll}_P$ of a program P are defined inductively over the structure of P, the definitions are given in Figure 1 and Figure 2. The sets of nodes and edges, as well as the mappings ar, σ and τ and the edges e^{i} and e^{f} are presented there in a straightforward manner. The transfer functions are depicted inside the nodes; id denotes the identity function on D (and is only there, because it is convenient if every control-flow graph has a unique initial edge, just as we have required). The incoming edges of a node are numbered from top to bottom.

We see that in addition to the transfer functions $[\![p]\!]$, $[\![b]\!]_{\mathsf{t}}$, $[\![b]\!]_{\mathsf{f}}$ and $\sqcup$ / $\sqcap$, given by the semantics of primitive constructs or the lattice D itself, the control-flow graph also requires certain semantic functions $\mathsf{branch} : D \to D$ and $\mathsf{combine} : D \times D \to D$; they must also be monotone and upwards (for forward analyses) or downwards (for backwards analyses) ω-continuous. The nodes labeled with branch and $\mathsf{combine}$ are used to inform the program analysis about the control dependencies of the nodes in the control-flow graph; this information is necessary for some kinds of analyses. If the goal of an analysis is to determine the properties of program points (all kinds of value analyses are in this class), then the nodes labeled with branch and combine are not really necessary. Simple joining of abstract states coming from different branches suffices because, at the confluence point, we have to take the disjunction of the properties on branches and this corresponds to the join on D. The situation is the same for the analyses that attempt to determine the properties of program paths (for example, where certain variables are used or defined)—disjunction (or join) has to be taken at confluence points. In such cases, branch may simply be the identity function and combine the second projection.

However, there are some analyses, most notably analyses for information flow, that do not belong to either class. These analyses do not attempt to find properties, but to make sure that all program runs are somehow similar (and to find the level of similarity). In principle, we want to find the most precise equivalence relation from a given set of equivalence relations over program runs, such that it relates any two runs that are possible for the given program. If some equivalence relation is suitable for one set of program runs (say, all runs over a certain program path), and the same equivalence relation is also suitable for some other set of program runs (all runs over some other program path), then this is not yet sufficient for a run from the first set and a run from the second set to be related. Some extra processing is necessary when merging those paths and we have found that a form of pre- and post-processing given by Figures 1 and 2 suffices for the examples that we are going to present. We believe that this form is sufficiently general and covers more-or-less every program analysis that we may want to execute. The combine-functions have appeared before in CFG-based analyses for secure information flow [5,6].

The post-processing function combine is given some extra information about the state before running the branches. This control flow graph edge from before the branches to combine does not really make the framework any more powerful, as the same information could be propagated through the branches by suitably modifying the domain D. But this modification may overly complicate the domain and the transfer functions.

Abstract denotations are associated with arbitrary programs by a forward or a backward analysis in the following way. For any control graph G, we define a functional $\Phi_G : D \to ((E \to D) \to (E \to D))$ by

$$\Phi_G(d)(\iota)(e) = \begin{cases} d & \text{if } e = e^{\mathrm{i}} \\ \ell(v)(\iota(e_1), \ldots, \iota(e_{ar(v)})), & \text{otherwise} \end{cases}$$

where $v = \sigma(e)$ and $e_i = \tau^{-1}(v, i)$.

The abstract denotation $[\![P]\!]_{\gg} : D \to D$ of a program P defined by a forward analysis is $[\![P]\!]_{\gg}(d) = \mathit{lfp}(\Phi_{G_P^{\gg}}(d))(e^{\mathrm{f}}_{G_P^{\gg}})$, where the least fixedpoint exists since $E \to D$ is an ω-complete upper semilattice and $\Phi_{G_P^{\gg}}(d)$ is upward ω-continuous. $[\![P]\!]_{\gg}$ is easily verified to be monotone and upward ω-continuous.

The abstract denotation $[\![P]\!]_{\ll} : D \to D$ assigned by a backward analysis to a program P is $[\![P]\!]_{\ll}(d) = \mathit{gfp}(\Phi_{G_P^{\ll}}(d))(e^{\mathrm{f}}_{G_P^{\ll}})$; the greatest fixedpoint exists since $E \to D$ is an ω-complete lower semilattice and $\Phi_{G_P^{\ll}}(d)$ is downward ω-continuous. $[\![P]\!]_{\ll} : D \to D$ is also monotone and downward ω-continuous.

As next we state some lemmas about the intermediate results of computing the least or greatest fixed points of $\Phi_G(d)$; these lemmas are needed in the next section. These lemmas basically say that the when we compute the fixed point of $\Phi_G(d)$ then we are also computing the fixed points of $\Phi_{G'}(d')$ where G' is a subgraph of G and d' is the is the abstract value on the initial edge of G'. Hence these lemmas expose the compositionality in the computation of $\mathit{lfp}(\Phi_G(d))$. We concentrate on the case where D is an ω-complete *upper* semilattice and the primitive semantic functions are all monotone and *upward* continuous. That is, we concentrate on $[\![P]\!]_{\gg}$; the results for $[\![P]\!]_{\ll}$ are analogous. For a control flow graph G and a mapping $\iota : E \to D$ we define the mappings $\iota_i^{\uparrow G} : E \to D$ by

$$\forall e \in E : \iota_0^{\uparrow G}(e) = \bot \qquad \forall i \in \mathbb{N} : \iota_{i+1}^{\uparrow G} = \Phi_G(d)(\iota_i^{\uparrow G}) \ .$$

We have $\iota_i^{\uparrow G} \sqsubseteq \iota_j^{\uparrow G}$ whenever $i \leq j$ because $\Phi_G(d)$ is monotone and $\iota_0^{\uparrow G} \sqsubseteq \iota_1^{\uparrow G}$. The join operation $\sqcup$ on D is extended pointwise to mappings from E to D.

Lemma 1 (Kleene's fixed point theorem) $\mathit{lfp}(\Phi_G(d)) = \bigsqcup_{i \in \mathbb{N}} \iota_i^{\uparrow G}$.

Also, for a mapping $\kappa : E \to D$ define the mappings $\iota_i^{\kappa \uparrow G}$ by

$$\iota_0^{\kappa \uparrow G} = \kappa \qquad \forall i \in \mathbb{N} : \iota_{i+1}^{\kappa \uparrow G} = \iota_i^{\kappa \uparrow G} \sqcup \Phi_G(d)(\iota_i^{\kappa \uparrow G})$$

and let $\mathit{lfp}^{\kappa}(\Phi_G(d)) = \bigsqcup_{i \in \mathbb{N}} \iota_i^{\kappa \uparrow G}$; this join exists because of ω-completeness of D. By Lemma 1, $\mathit{lfp}(\Phi_G(d)) = \mathit{lfp}^{\lambda e.\bot}(\Phi_G(d))$.

Lemma 2 *Let $\kappa, \kappa' : E \to D$ be such that $\kappa \sqsubseteq \kappa' \sqsubseteq \mathit{lfp}^{\kappa}(\Phi_G(d))$. Then $\mathit{lfp}^{\kappa}(\Phi_G(d)) = \mathit{lfp}^{\kappa'}(\Phi_G(d))$.*

PROOF. For all $i \in \mathbb{N}$ we have

$$\iota_i^{\kappa \uparrow G} \sqsubseteq \iota_i^{\kappa' \uparrow G} \sqsubseteq \mathit{lfp}^{\kappa}(\Phi_G(d)),$$

this follows trivially from the monotonicity of $\Phi_G(d)$ and the definition of the fixed points. Taking the joins over all i gives us

$$lfp^{\kappa}(\Phi_G(d)) \sqsubseteq lfp^{\kappa'}(\Phi_G(d)) \sqsubseteq lfp^{\kappa}(\Phi_G(d)),$$

giving us the equality of least fixed points. $\square$

Lemma 3 *Let $G_P^{\gg}$ be the control flow graph of a program P, let P' be a syntactic subprogram of P and let $G_{P'}^{\gg}$ be a subgraph of $G_P^{\gg}$ that is the control flow graph of P'. The initial and final edges $\mathrm{e}^{\mathrm{i}}_{G_{P'}^{\gg}}$ and $\mathrm{e}^{\mathrm{f}}_{G_{P'}^{\gg}}$ are also edges of $G_P^{\gg}$. Let E' be the set of edges of $G_{P'}^{\gg}$. Let $i \in \mathbb{N}$ be fixed. Let $\kappa_i : E' \to D$ be the restriction of $\iota_i^{\uparrow G_P^{\gg}}$ to E'. Let $\hat{d} = \iota_i^{\uparrow G_P^{\gg}}(\mathrm{e}^{\mathrm{i}}_{G_{P'}^{\gg}})$ Then*

$$\kappa_i \sqsubseteq lfp(\Phi_{G_{P'}^{\gg}}(\hat{d})) \ . \tag{1}$$

PROOF. Define the mappings $\hat{\iota}_j : E' \to D$ by

$$\forall e \in E' : \hat{\iota}_0 = \bot \qquad \forall j \in \mathbb{N} : \hat{\iota}_{j+1} = \Phi_{G_{P'}^{\gg}}(\hat{d})(\hat{\iota}_j) \ . \tag{2}$$

By induction over j, we show that $\kappa_j \sqsubseteq \hat{\iota}_j$ for all $j \leq i$. If $j = 0$ then $\kappa_0 = \hat{\iota}_0 = \lambda e.\bot$. If $j > 0$ then

$$\hat{\iota}_j(e^{\mathrm{i}}_{G_{P'}^{\gg}}) = \hat{d} = \iota_i^{\uparrow G_P^{\gg}}(e^{\mathrm{i}}_{G_{P'}^{\gg}}) \sqsupseteq \iota_j^{\uparrow G_P^{\gg}}(e^{\mathrm{i}}_{G_{P'}^{\gg}}) = \kappa_j(e^{\mathrm{i}}_{G_{P'}^{\gg}}) \ .$$

If $e \in E' \backslash \{e^{\mathrm{i}}_{G_{P'}^{\gg}}\}$ then let v be the source node of e and let $e_1, \ldots, e_k$ be the incoming edges of v. The edges $e_1, \ldots, e_k$ also belong to E'. We have

$$\hat{\iota}_j(e) = \ell(v)(\hat{\iota}_{j-1}(e_1), \ldots, \hat{\iota}_{j-1}(e_k)) \sqsupseteq \ell(v)(\kappa_{j-1}(e_1), \ldots, \kappa_{j-1}(e_k)) = \kappa_j(e),$$

where the inequality is by the induction assumption and monotonicity of $\ell(v)$. Finally, we get (1) from

$$\kappa_i \sqsubseteq \hat{\iota}_i \sqsubseteq \bigsqcup_{j \in \mathbb{N}} \hat{\iota}_j = lfp(\Phi_{G_{P'}^{\gg}}(\hat{d})),$$

where the last inequality comes from Lemma 1. $\square$

Corollary 4 *With the same notations as in Lemma 3, we have*

$$[\![P']\!]_{\gg}(\iota_i^{\uparrow G_P^{\gg}}(e^{\mathrm{i}}_{G_{P'}^{\gg}})) \sqsupseteq \iota_i^{\uparrow G_P^{\gg}}(e^{\mathrm{f}}_{G_{P'}^{\gg}}) \ . \tag{3}$$

PROOF. Inequation (3) is obtained by applying the mappings in (1) to $e^{\mathrm{f}}_{G_{P'}^{\gg}}$. $\square$

Lemma 5 *Let v be a node in a control flow graph G. Let $e_1, \dots, e_{ar(v)}$ be the incoming edges of v (i.e. $\tau(e_j) = (v, j)$) and let e be the outgoing edge of v (i.e. $\sigma(e) = v$). Then $\ell(v)(\iota_i^{\uparrow G}(e_1), \dots, \iota_i^{\uparrow G}(e_{ar(v)})) \sqsupseteq \iota_i^{\uparrow G}(e)$ for all $i \in \mathbb{N}$.*

PROOF. $\ell(v)(\iota_i^{\uparrow G}(e_1), \dots, \iota_i^{\uparrow G}(e_{ar(v)})) = \iota_{i+1}^{\uparrow G}(e) \sqsupseteq \iota_i^{\uparrow G}(e)$. $\square$

Lemma 6 *With the same notations as in Lemma 3, we have*

$$[\![P']\!]_{\gg}(\mathit{lfp}(\Phi_{G_P^{\gg}}(d))(e^{\mathrm{i}}_{G_{P'}^{\gg}})) = \mathit{lfp}(\Phi_{G_P^{\gg}}(d))(e^{\mathrm{f}}_{G_{P'}^{\gg}}) \ . \tag{4}$$

PROOF. The inequation "$\sqsupseteq$" in (4) can be obtained by taking the join over all $i \in \mathbb{N}$ in (3). To demonstrate the inequation "$\sqsubseteq$" define the mappings $\hat{\iota}_j : E' \to D$ as in (2) where we take $\hat{d} = \mathit{lfp}(\Phi_{G_P^{\gg}}(d))(e^{\mathrm{i}}_{G_{P'}^{\gg}})$. By induction on j we can show that $\hat{\iota}_j(e) \sqsubseteq \mathit{lfp}(\Phi_{G_P^{\gg}}(d))(e)$, for all $e \in E'$. Indeed, $\hat{\iota}_0(e) = \bot$ for all e, this establishes the basis of the induction. If $j > 0$ and $e = e^{\mathrm{i}}_{G_{P'}^{\gg}}$ then $\hat{\iota}_j(e) = \hat{d} = \mathit{lfp}(\Phi_{G_P^{\gg}}(d))(e)$. If $e \in E' \backslash \{e^{\mathrm{i}}_{G_{P'}^{\gg}}\}$ then let v be the source of e and $e_1, \dots, e_k$ be the incoming edges of v; the edges $e_1, \dots, e_k$ also belong to E'. We have

$$\begin{aligned}\hat{\iota}_j(e) &= \ell(v)(\hat{\iota}_{j-1}(e_1), \dots, \hat{\iota}_{j-1}(e_k)) \\ &\sqsubseteq \ell(v)(\mathit{lfp}(\Phi_{G_P^{\gg}}(d))(e_1), \dots, \mathit{lfp}(\Phi_{G_P^{\gg}}(d))(e_k)) = \mathit{lfp}(\Phi_{G_P^{\gg}}(d))(e),\end{aligned}$$

where the inequation comes from the induction assumption and the last equation from the properties of fixed points. Taking the join over all j gives us

$$\mathit{lfp}(\Phi_{G_{P'}^{\gg}}(\hat{d}))(e) = \bigsqcup_{j \in \mathbb{N}} \hat{\iota}_j(e) \sqsubseteq \mathit{lfp}(\Phi_{G_P^{\gg}}(d))(e)$$

for all $e \in E'$. This is the inequation "$\sqsubseteq$" in (4) for $e = e^{\mathrm{f}}_{G_{P'}^{\gg}}$. $\square$

3 Type Systems

We are now ready to proceed to our type systems. We present a schematic type system for both forward and backward analyses. Type systems for concrete analyses are obtained as instances.

In both cases, types are of the form $d_1 \longrightarrow d_2$ where $d_1, d_2 \in D$, reflecting the idea that denotations of programs are transformers of (standard or non-standard) states. The intended meaning of a typing judgement $P : d_1 \longrightarrow d_2$ is that if the state before running P can be described by d_1, then the state

$$\frac{d_1 \sqsubseteq d_1' \quad \vdash_{\gg} P : d_1' \longrightarrow d_2' \quad d_2' \sqsubseteq d_2}{\vdash_{\gg} P : d_1 \longrightarrow d_2} \text{(sub)}$$

$$\frac{}{\vdash_{\gg} p : d \longrightarrow [\![p]\!](d)} \text{(prim)}$$

$$\frac{\vdash_{\gg} P_1 : d \longrightarrow d' \quad \vdash_{\gg} P_2 : d' \longrightarrow d''}{\vdash_{\gg} P_1; P_2 : d \longrightarrow d''} \text{(seq)}$$

$$\frac{\vdash_{\gg} P_1 : [\![b]\!]_{\mathsf{t}}(\mathsf{branch}(d)) \longrightarrow d' \quad \vdash_{\gg} P_2 : [\![b]\!]_{\mathsf{f}}(\mathsf{branch}(d)) \longrightarrow d'}{\vdash_{\gg} \mathit{if}\ b\ \mathit{then}\ P_1\ \mathit{else}\ P_2 : d \longrightarrow \mathsf{combine}(d, d')} \text{(if)}$$

$$\frac{\vdash_{\gg} P_1 : [\![b]\!]_{\mathsf{t}}(d') \longrightarrow d' \quad \mathsf{branch}(d) \sqsubseteq d'}{\vdash_{\gg} \mathit{while}\ b\ \mathit{do}\ P_1 : d \longrightarrow \mathsf{combine}(d, [\![b]\!]_{\mathsf{f}}(d'))} \text{(while)}$$

Fig. 3. Schematic type system corresponding to forward may-analyses

after is described by d_2. This is independent of the direction of the analysis, i.e., the implication sign in the type always indicates the direction of running the program.

Also in both cases, the type system is extracted from the definition of control-flow graphs for programs. The typing rule for each statement-construct is read off from the corresponding clause in the definition of control-flow graphs. The rule (sub) is a subsumption rule combining the usual subsumption rule with structural subtyping for implications.

The type system derived from a forward analysis is presented in Figure 3. All rules, except (while), are self-explanatory. In (while), d' is a form of a loop invariant—it corresponds to the abstract state at the back-edge in the direct control-flow graph of the while-statement. The normal use of the type system for analyzing a program is this: given a program P and an abstract state d, find an abstract state d' such that $P : d \longrightarrow d'$. Inspecting the rule set, one can see that the process is immediate except for two aspects: (1) it is not *a priori* clear if and when it is necessary to apply (sub) and (2) applying (while) takes guessing a suitable invariant.

The type system is equivalent to the inducing analysis in the sense of the following theorem.

Theorem 7 *For any $P \in \mathbf{Stm}$ and $d_1, d_2 \in D$ it holds that $\vdash_{\gg} P : d_1 \longrightarrow d_2$ iff $[\![P]\!]_{\gg}(d_1) \sqsubseteq d_2$.*

PROOF. Both directions are proved by induction over the structure of the program P; or over the derivation tree of $\vdash_{\gg} P : d_1 \longrightarrow d_2$, which is equivalent.

Consider the 'only if' direction first. Let P be a program such that $\vdash_{\gg} P : d_1 \longrightarrow d_2$ holds. Consider the last rule that was applied to derive $\vdash_{\gg} P : d_1 \longrightarrow d_2$.

If the rule was (sub) then we also have $\vdash_{\gg} P : d'_1 \longrightarrow d'_2$ for some $d'_1 \sqsupseteq d_1$ and $d'_2 \sqsubseteq d_2$. Hence we have

$$[\![P]\!]_{\gg}(d_1) \sqsubseteq [\![P]\!]_{\gg}(d'_1) \sqsubseteq d'_2 \sqsubseteq d_2$$

where the first inequality is given by the monotonicity of $[\![P]\!]_{\gg}$ and the second by the induction hypothesis applied to $\vdash_{\gg} P : d'_1 \longrightarrow d'_2$.

If the last rule was (prim) then P is a primitive statement p. We have $d_1 = d$ and $d_2 = [\![p]\!](d) = [\![P]\!]_{\gg}(d_1)$ for some $d \in D$.

If the last rule was (seq) then P is of the form $P_1; P_2$. Also, there exists $d' \in D$, such that $\vdash_{\gg} P_1 : d_1 \longrightarrow d'$ and $\vdash_{\gg} P_2 : d' \longrightarrow d_2$. The induction hypothesis gives us $[\![P_1]\!]_{\gg}(d_1) \sqsubseteq d'$ and $[\![P_2]\!]_{\gg}(d') \sqsubseteq d_2$. Consider the computation of $\mathit{lfp}(\Phi_{G_P^{\gg}}(d_1))$; let $\iota_i^{\uparrow G_P^{\gg}}$ be defined as in the end of Section 2. Let e be the edge in $G_P^{\gg}$ that is between $G_{P_1}^{\gg}$ and $G_{P_2}^{\gg}$.

Let $i \geq 1$. Corollary 4, when applied to the subprogram P_1, gives us

$$\iota_i^{\uparrow G_P^{\gg}}(e) \sqsubseteq [\![P_1]\!]_{\gg}(\iota_i^{\uparrow G_P^{\gg}}(e^{\mathrm{i}})) = [\![P_1]\!]_{\gg}(d_1) \sqsubseteq d' \ .$$

Applying Corollary 4 to the subprogram P_2 gives

$$\iota_i^{\uparrow G_P^{\gg}}(e^{\mathrm{f}}) \sqsubseteq [\![P_2]\!]_{\gg}(\iota_i^{\uparrow G_P^{\gg}}(e)) \sqsubseteq [\![P_2]\!]_{\gg}(d') \sqsubseteq d_2 \ .$$

Hence by Lemma 1, $[\![P]\!]_{\gg}(d_1) = \mathit{lfp}(\Phi_{G_P^{\gg}}(d_1))(e^{\mathrm{f}}) \sqsubseteq d_2$.

If the last rule that was applied to derive $\vdash_{\gg} P : d_1 \longrightarrow d_2$ was (if) then P is of the form *if* b *then* P_1 *else* P_2. Let us give the following names to the following edges in $G_P^{\gg}$ (see Figure 1).

- e_{1a} and e_{2a} are the edges from the node labeled with branch to the nodes labeled with $[\![b]\!]_{\mathsf{t}}$ and $[\![b]\!]_{\mathsf{f}}$, respectively;
- e_{1b} [resp. e_{2b}] is the initial edge of $G_{P_1}^{\gg}$ [resp. $G_{P_2}^{\gg}$];
- e_{1c} [resp. e_{2c}] is the final edge of $G_{P_1}^{\gg}$ [resp. $G_{P_2}^{\gg}$];
- e_3 [resp. e_4] is the edge from the node labeled with $\sqcup$ [resp id] to the node labeled with $\mathsf{combine}$.

The induction hypothesis gives us $[\![P_1]\!]_{\gg}([\![b]\!]_{\mathsf{t}}(\mathsf{branch}(d))) \sqsubseteq d'$ and $[\![P_2]\!]_{\gg}([\![b]\!]_{\mathsf{f}}(\mathsf{branch}(d))) \sqsubseteq d'$. Let $i \geq 1$. Using Lemma 5 and Corollary 4, as well as the induction hypothesis and the monotonicity of semantic functions, we can derive

$$\begin{aligned}\iota_i^{\uparrow G_P^{\gg}}(e^{\mathrm{f}}) &\sqsubseteq \mathsf{combine}(\iota_i^{\uparrow G_P^{\gg}}(e_4), \iota_i^{\uparrow G_P^{\gg}}(e_3)) \\ &\sqsubseteq \mathsf{combine}(d, \iota_i^{\uparrow G_P^{\gg}}(e_{1c}) \sqcup \iota_i^{\uparrow G_P^{\gg}}(e_{2c})) \\ &\sqsubseteq \mathsf{combine}(d, [\![P_1]\!]_{\gg}(\iota_i^{\uparrow G_P^{\gg}}(e_{1b})) \sqcup [\![P_2]\!]_{\gg}(\iota_i^{\uparrow G_P^{\gg}}(e_{2b}))) \\ &\sqsubseteq \mathsf{combine}(d, [\![P_1]\!]_{\gg}([\![b]\!]_{\mathsf{t}}(\iota_i^{\uparrow G_P^{\gg}}(e_{1a}))) \sqcup [\![P_2]\!]_{\gg}([\![b]\!]_{\mathsf{f}}(\iota_i^{\uparrow G_P^{\gg}}(e_{2a})))) \\ &\sqsubseteq \mathsf{combine}(d, [\![P_1]\!]_{\gg}([\![b]\!]_{\mathsf{t}}(\mathsf{branch}(d))) \sqcup [\![P_2]\!]_{\gg}([\![b]\!]_{\mathsf{f}}(\mathsf{branch}(d)))) \\ &\sqsubseteq \mathsf{combine}(d, d' \sqcup d') = \mathsf{combine}(d, d')\ .\end{aligned}$$

Hence by Lemma 1, $[\![P]\!]_{\gg}(d) = \mathit{lfp}(\Phi_{G_P^{\gg}}(d))(e^{\mathrm{f}}) \sqsubseteq \mathsf{combine}(d, d')$.

If the last rule that was applied to derive $\models_{\gg} P : d_1 \longrightarrow d_2$ was (while) then P is of the form *while* b *do* P_1. Let us give the following names to the following edges in $G_P^{\gg}$ (see Figure 1).

- e_1 [resp. e_2; e_5; e_6; e_7] is the edge from the node labeled with branch [resp. $\sqcup$; $\sqcup$; $[\![b]\!]_{\mathsf{f}}$; id] to the node labeled with $\sqcup$ [resp. $[\![b]\!]_{\mathsf{t}}$; $[\![b]\!]_{\mathsf{f}}$; $\mathsf{combine}$; $\mathsf{combine}$];
- e_3 [resp. e_4] is the initial [resp. final] edge of $G_{P_1}^{\gg}$.

According to the rule (while) and the induction assumption there exists $d' \in D$, such that $\mathsf{branch}(d) \sqsubseteq d'$ and $[\![P_1]\!]_{\gg}([\![b]\!]_{\mathsf{t}}(d')) \sqsubseteq d'$. Using mathematical induction over i, we show that for all $i \in \mathbb{N}$, $\iota_i^{\uparrow G_P^{\gg}}(e_4) \sqsubseteq d'$. Indeed, $\iota_0^{\uparrow G_P^{\gg}}(e_4) = \perp$ and

$$\begin{aligned}\iota_{i+1}^{\uparrow G_P^{\gg}}(e_4) &\sqsubseteq [\![P_1]\!]_{\gg}([\![b]\!]_{\mathsf{t}}(\iota_{i+1}^{\uparrow G_P^{\gg}}(e_2))) \\ &= [\![P_1]\!]_{\gg}([\![b]\!]_{\mathsf{t}}(\iota_i^{\uparrow G_P^{\gg}}(e_1) \sqcup \iota_i^{\uparrow G_P^{\gg}}(e_4))) \\ &\sqsubseteq [\![P_1]\!]_{\gg}([\![b]\!]_{\mathsf{t}}(\mathsf{branch}(d) \sqcup d')) = [\![P_1]\!]_{\gg}([\![b]\!]_{\mathsf{t}}(d')) \sqsubseteq d',\end{aligned}$$

here we have besides the induction assumption $\iota_i^{\uparrow G_P^{\gg}}(e_4) \sqsubseteq d'$ also used Lemma 5 and Corollary 4.

Let $i \geq 1$. Using Lemma 5 and Corollary 4, as well as the induction hypothesis and the monotonicity of semantic functions, we can derive

$$\begin{aligned}\iota_i^{\uparrow G_P^{\gg}}(e^{\mathrm{f}}) &\sqsubseteq \mathsf{combine}(\iota_i^{\uparrow G_P^{\gg}}(e_7), \iota_i^{\uparrow G_P^{\gg}}(e_6)) \\ &\sqsubseteq \mathsf{combine}(d, [\![b]\!]_{\mathsf{f}}(\iota_i^{\uparrow G_P^{\gg}}(e_5))) \\ &\sqsubseteq \mathsf{combine}(d, [\![b]\!]_{\mathsf{f}}(\iota_i^{\uparrow G_P^{\gg}}(e_1) \sqcup \iota_i^{\uparrow G_P^{\gg}}(e_4))) \\ &\sqsubseteq \mathsf{combine}(d, [\![b]\!]_{\mathsf{f}}(\mathsf{branch}(d) \sqcup d')) = \mathsf{combine}(d, [\![b]\!]_{\mathsf{f}}(d'))\ .\end{aligned}$$

Hence by Lemma 1, $[\![P]\!]_{\gg}(d) = \mathit{lfp}(\Phi_{G_P^{\gg}}(d))(e^{\mathrm{f}}) \sqsubseteq \mathsf{combine}(d, [\![b]\!]_{\mathsf{f}}(d'))$.

For showing the 'if' direction, let P be a program and $d \in D$. We show that $\models_{\gg} P : d \longrightarrow [\![P]\!]_{\gg}(d)$; the claim of the theorem in its full generality is then obtained by applying the typing rule (sub).

If P is a primitive statement p then we have $\vdash_{\gg} p : d \longrightarrow [\![p]\!](d)$ by the rule (prim). Also, $[\![p]\!]_{\gg}(d) = [\![p]\!](d)$.

If P is $P_1; P_2$ then let $d'' = [\![P]\!]_{\gg}(d)$ and let e be the edge of $G_P^{\gg}$ that connects $G_{P_1}^{\gg}$ and $G_{P_2}^{\gg}$. Let $d' = \mathit{lfp}(\Phi_{G_P^{\gg}}(d))(e)$. Applying Lemma 6 to P_1 and P_2 we get $[\![P_1]\!]_{\gg}(d) = d'$ and $[\![P_2]\!]_{\gg}(d') = d''$. By the induction assumption, we can derive $\vdash_{\gg} P_1 : d \longrightarrow d'$ and $\vdash_{\gg} P_2 : d' \longrightarrow d''$. These derivations together with an application of the rule (seq) give us the necessary derivation for P.

If P is *if* b *then* P_1 *else* P_2 then let certain edges in $G_P^{\gg}$ be given the same names as in the proof for *if*-direction. For such named edge e_{x} let $d_{\mathrm{x}} = \mathit{lfp}(\Phi_{G_P^{\gg}}(d))(e_{\mathrm{x}})$. We have $d_{1a} = d_{2a} = \mathsf{branch}(d)$, $d_{1b} = [\![b]\!]_{\mathsf{t}}(\mathsf{branch}(d))$, $d_{2b} = [\![b]\!]_{\mathsf{f}}(\mathsf{branch}(d))$ by the properties of fixed points; $d_{1c} = [\![P_1]\!]_{\gg}([\![b]\!]_{\mathsf{t}}(\mathsf{branch}(d)))$, $d_{2c} = [\![P_2]\!]_{\gg}([\![b]\!]_{\mathsf{f}}(\mathsf{branch}(d)))$ by Lemma 6; $[\![P]\!]_{\gg}(d) = \mathsf{combine}(d, d_{1c} \sqcup d_{2c})$. Let $d' = d_{1c} \sqcup d_{2c}$. By induction assumption we derive $\vdash_{\gg} P_1 : [\![b]\!]_{\mathsf{t}}(\mathsf{branch}(d)) \longrightarrow d_{1c}$ and $\vdash_{\gg} P_2 : [\![b]\!]_{\mathsf{f}}(\mathsf{branch}(d)) \longrightarrow d_{2c}$. Using the typing rule (sub) we can derive $\vdash_{\gg} P_1 : [\![b]\!]_{\mathsf{t}}(\mathsf{branch}(d)) \longrightarrow d'$ and $\vdash_{\gg} P_2 : [\![b]\!]_{\mathsf{f}}(\mathsf{branch}(d)) \longrightarrow d'$. These derivations together with an application of the rule (if) give us the necessary derivation for P.

If P is *while* b *do* P_1 then let certain edges in $G_P^{\gg}$ be given the same names as in the proof for *if*-direction. Define the values d_{x} as above. Let $d' = d_2$. Then $\mathsf{branch}(d) \sqsubseteq d'$ because $d' = \mathsf{branch}(d) \sqcup d_4$. By Lemma 6, $[\![P_1]\!]_{\gg}([\![b]\!]_{\mathsf{t}}(d')) = d_4$. Hence, by induction assumption, we can derive $\vdash_{\gg} P_1 : [\![b]\!]_{\mathsf{t}}(d') \longrightarrow d_4$ and, using the rule (sub), also $\vdash_{\gg} P_1 : [\![b]\!]_{\mathsf{t}}(d') \longrightarrow d'$. By the properties of fixed points we have $[\![P]\!]_{\gg}(d) = \mathsf{combine}(d, [\![b]\!]_{\mathsf{f}}(d'))$ and the corresponding type can be derived with the help of the rule (while). $\square$

We see that, given an initial abstract state, the type system can approve as final any abstract state that is weaker (containing more concrete states) than the final abstract state computed by the analysis. The analysis gives the strongest (smallest) postcondition of a precondition, whereas the type system accepts any postcondition.

The type system corresponding to backward analyses is given in Figure 4. (Remember that the direction of the arrow still indicates the direction of running the program.) The rules are different, since inverse control-flow graphs differ from direct ones. But the ideas are still exactly the same. Here the loop invariant d' corresponds to the abstract state at the back-edge of the inverse control-flow graph of the while-statement. The normal use of the type system to analyze a program is this: given a program P and an abstract state d, find an abstract state d' such that $P : d' \longrightarrow d$.

$$\frac{d_1 \sqsubseteq d_1' \quad \vdash_{\ll} P : d_1' \longrightarrow d_2' \quad d_2' \sqsubseteq d_2}{\vdash_{\ll} P : d_1 \longrightarrow d_2} \text{(sub)}$$

$$\frac{}{\vdash_{\ll} p : [\![p]\!](d) \longrightarrow d} \text{(prim)}$$

$$\frac{\vdash_{\ll} P_1 : d'' \longrightarrow d' \quad \vdash_{\ll} P_2 : d' \longrightarrow d}{\vdash_{\ll} P_1; P_2 : d'' \longrightarrow d} \text{(seq)}$$

$$\frac{\vdash_{\ll} P_1 : d_{\mathrm{t}} \longrightarrow \mathsf{branch}(d) \quad \vdash_{\ll} P_2 : d_{\mathrm{f}} \longrightarrow \mathsf{branch}(d)}{\vdash_{\ll} \mathit{if}\ b\ \mathit{then}\ P_1\ \mathit{else}\ P_2 : \mathsf{combine}(d, [\![b]\!]_{\mathsf{t}}(d_{\mathrm{t}}) \sqcap [\![b]\!]_{\mathsf{f}}(d_{\mathrm{f}})) \longrightarrow d} \text{(if)}$$

$$\frac{\vdash_{\ll} P_1 : d' \longrightarrow [\![b]\!]_{\mathsf{t}}(d') \sqcap [\![b]\!]_{\mathsf{f}}(\mathsf{branch}(d))}{\vdash_{\ll} \mathit{while}\ b\ \mathit{do}\ P_1 : \mathsf{combine}(d, [\![b]\!]_{\mathsf{t}}(d') \sqcap [\![b]\!]_{\mathsf{f}}(\mathsf{branch}(d))) \longrightarrow d} \text{(while)}$$

Fig. 4. Schematic type system corresponding to backward must-analyses

Again we have a perfect match between the type system and the analysis; the proof is the same as in the forward case:

Theorem 8 *For any $P \in \mathbf{Stm}$ and $d_1, d_2 \in D$ it holds that $\vdash_{\ll} P : d_1 \longrightarrow d_2$ iff $d_1 \sqsubseteq [\![P]\!]_{\ll}(d_2)$.*

Here the type system can identify as initial any abstract state that is stronger than the initial state computed by the analysis. While the analysis delivers the weakest precondition of a postcondition, the type system approves any precondition.

It is now also clear why we wanted forward analyses to be "may" and backward analyses to be "must": this gives the subsumption rules their normal shape: the implication construct is covariant in its succedent and contravariant in the antecedent and not the other way around.

4 Examples

To demonstrate the generality of the framework given in Section 2 and the mechanics of our type systems, we now show how some concrete data-flow analyses can be expressed within it and discuss the type systems that our method delivers for them. We present the analyses for constant propagation, very busy expressions, and secure information flow. And we show even Hoare logic to be an instance of our framework.

In the examples below, we assume that the programming language is defined over a supply **Var** of arithmetic variables, that primitive statements p have the form $x := \mathit{aop}(x_1, \ldots, x_k)$ where $x, x_1, \ldots, x_k \in \mathbf{Var}$ and aop is an arithmetic operator with arity k, and Boolean expressions have the form $\mathit{bop}(x_1, \ldots, x_k)$ where bop is an Boolean operator with arity k. In the following, we let o range over both aop and bop.

Constant propagation is a forward may-analysis. The aim is to determine for all program points the values of as many variables as possible. In the context of type systems, we are given the initial values of some variables and want to determine the final values of as many variables as possible.

Let $\mathbb{Z}^\top = \mathbb{Z} \mathbin{\dot\cup} \{\top\}$, where $\top$ is the largest element and other elements are incomparable. The abstract state space D for constant propagation is $(\mathbf{Var} \to \mathbb{Z}^\top)_\perp$, where $X_\perp$ denotes the set $X \mathbin{\dot\cup} \{\perp\}$ with $\perp$ the smallest element and all other elements ordered in the same way as in X, and the order on $\mathbf{Var} \to \mathbb{Z}^\top$ is defined pointwise.

The analysis works in the forward direction. For ease of exposition, let $aop \in \mathbb{Z} \cup \{+, \times\}$ and $bop \in \{\leq, =\}$. Here the elements of $\mathbb{Z}$ are nullary operators and the rest are binary. If $d = \perp$ then $[\![x := o(x_1, \ldots, x_k)]\!](d) = \perp$ and $[\![o(x_1, \ldots, x_k)]\!]_{\mathsf{t}}(d) = [\![o(x_1, \ldots, x_k)]\!]_{\mathsf{f}}(d) = \perp$. Otherwise

$$[\![x := n]\!](d) = d[x \mapsto n]$$

$$[\![x := y + z]\!](d) = \begin{cases} d[x \mapsto d(y) + d(z)], & d(y), d(z) \in \mathbb{Z} \\ d[x \mapsto \top], & \text{otherwise} \end{cases}$$

$$[\![x := y \times z]\!](d) = \begin{cases} d[x \mapsto 0], & d(y) = 0 \text{ or } d(z) = 0 \\ d[x \mapsto d(y) \cdot d(z)], & d(y), d(z) \in \mathbb{Z}\backslash\{0\} \\ d[x \mapsto \top], & \text{otherwise} \end{cases}$$

$$[\![x \leq y]\!]_{\mathsf{t}}(d) = \begin{cases} \perp, & d(x), d(y) \in \mathbb{Z} \text{ and } d(x) > d(y) \\ d, & \text{otherwise} \end{cases}$$

$$[\![x \leq y]\!]_{\mathsf{f}}(d) = \begin{cases} \perp, & d(x), d(y) \in \mathbb{Z} \text{ and } d(x) \leq d(y) \\ d, & \text{otherwise} \end{cases}$$

$$[\![x = y]\!]_{\mathsf{t}}(d) = \begin{cases} \perp, & d(x), d(y) \in \mathbb{Z} \text{ and } d(x) \neq d(y) \\ d[x \mapsto d(y)], & d(y) \in \mathbb{Z} \text{ and } d(x) = \top \\ d[y \mapsto d(x)], & d(x) \in \mathbb{Z} \text{ and } d(y) = \top \\ d, & \text{otherwise} \end{cases}$$

$$[\![x = y]\!]_{\mathsf{f}}(d) = \begin{cases} \perp, & d(x), d(y) \in \mathbb{Z} \text{ and } d(x) = d(y) \\ d, & \text{otherwise} \end{cases}$$

Finally, $\mathsf{branch}(d) = d$ and $\mathsf{combine}(d, d') = d'$. The corresponding type system is given in Figure 5. Fig. 5 may seem quite different from Fig. 3 but in reality this is not so. The denotations of primitive statements and boolean expressions are defined through case analysis. We have chosen to present the different cases as different rules. We could have presented single rules for primitive

$$\frac{d_1 \sqsubseteq d_1' \quad \vdash_{\gg} P : d_1' \longrightarrow d_2' \quad d_2' \sqsubseteq d_2}{\vdash_{\gg} P : d_1 \longrightarrow d_2} \ (\text{sub})$$

$$\frac{}{\vdash_{\gg} x := o(x_1, \dots, x_k) : d \longrightarrow d[x \mapsto \top]} \ (\text{prim}_1)$$

$$\frac{}{\vdash_{\gg} x := n : d \longrightarrow d[x \mapsto n]} \ (\text{prim}_2)$$

$$\frac{d(y) \in \mathbb{Z} \quad d(z) \in \mathbb{Z}}{\vdash_{\gg} x := o(y, z) : d \longrightarrow d[x \mapsto [\![o]\!](d(y), d(z))]} \ (\text{prim}_3)$$

$$\frac{d(y) = 0 \text{ or } d(z) = 0}{\vdash_{\gg} x := y \times z : d \longrightarrow d[x \mapsto 0]} \ (\text{prim}_4)$$

$$\frac{\vdash_{\gg} P_1 : d \longrightarrow d' \quad \vdash_{\gg} P_2 : d' \longrightarrow d''}{\vdash_{\gg} P_1; P_2 : d \longrightarrow d''} \ (\text{seq})$$

$$\frac{\vdash_{\gg} P_1 : d \longrightarrow d' \quad \vdash_{\gg} P_2 : d \longrightarrow d'}{\vdash_{\gg} \mathit{if}\ o(x, y)\ \mathit{then}\ P_1\ \mathit{else}\ P_2 : d \longrightarrow d'} \ (\text{if}_1)$$

$$\frac{d(x) \in \mathbb{Z} \quad d(y) \in \mathbb{Z} \quad [\![o]\!](d(x), d(y)) \quad \vdash_{\gg} P_1 : d \longrightarrow d'}{\vdash_{\gg} \mathit{if}\ o(x, y)\ \mathit{then}\ P_1\ \mathit{else}\ P_2 : d \longrightarrow d'} \ (\text{if}_2)$$

$$\frac{d(x) \in \mathbb{Z} \quad d(y) \in \mathbb{Z} \quad \neg[\![o]\!](d(x), d(y)) \quad \vdash_{\gg} P_2 : d \longrightarrow d'}{\vdash_{\gg} \mathit{if}\ o(x, y)\ \mathit{then}\ P_1\ \mathit{else}\ P_2 : d \longrightarrow d'} \ (\text{if}_3)$$

$$\frac{d(x) \in \mathbb{Z} \quad d(y) = \top \quad \vdash_{\gg} P_1 : d[y \mapsto d(x)] \longrightarrow d' \quad \vdash_{\gg} P_2 : d \longrightarrow d'}{\vdash_{\gg} \mathit{if}\ x = y\ \mathit{then}\ P_1\ \mathit{else}\ P_2 : d \longrightarrow d'} \ (\text{if}_4)$$

$$\frac{d(y) \in \mathbb{Z} \quad d(x) = \top \quad \vdash_{\gg} P_1 : d[x \mapsto d(y)] \longrightarrow d' \quad \vdash_{\gg} P_2 : d \longrightarrow d'}{\vdash_{\gg} \mathit{if}\ x = y\ \mathit{then}\ P_1\ \mathit{else}\ P_2 : d \longrightarrow d'} \ (\text{if}_5)$$

$$\frac{\vdash_{\gg} P : d' \longrightarrow d' \quad d \sqsubseteq d'}{\vdash_{\gg} \mathit{while}\ o(x, y)\ \mathit{do}\ P : d \longrightarrow d'} \ (\text{while}_1)$$

$$\frac{\vdash_{\gg} P : d' \longrightarrow d' \quad d \sqsubseteq d' \quad d'(x) \in \mathbb{Z} \quad d'(y) \in \mathbb{Z} \quad [\![o]\!](d'(x), d'(y))}{\vdash_{\gg} \mathit{while}\ o(x, y)\ \mathit{do}\ P : d \longrightarrow \bot} \ (\text{while}_2)$$

$$\frac{d(x) \in \mathbb{Z} \quad d(y) \in \mathbb{Z} \quad \neg[\![o]\!](d(x), d(y))}{\vdash_{\gg} \mathit{while}\ o(x, y)\ \mathit{do}\ P : d \longrightarrow d} \ (\text{while}_3)$$

$$\frac{\vdash_{\gg} P : d'[y \mapsto d'(x)] \longrightarrow d' \quad d \sqsubseteq d' \quad d'(x) \in \mathbb{Z} \quad d'(y) = \top}{\vdash_{\gg} \mathit{while}\ x = y\ \mathit{do}\ P : d \longrightarrow d'} \ (\text{while}_4)$$

$$\frac{\vdash_{\gg} P : d'[x \mapsto d'(y)] \longrightarrow d' \quad d \sqsubseteq d' \quad d'(y) \in \mathbb{Z} \quad d'(x) = \top}{\vdash_{\gg} \mathit{while}\ x = y\ \mathit{do}\ P : d \longrightarrow d'} \ (\text{while}_5)$$

Fig. 5. Type system from constant propagation analysis

statements, if-statements and while-statements by directly following Fig. 3, thus obtaining the same five rules as in Fig. 3, but then the case analysis would have been present in these rules, making them quite unreadable. Also, in some rules for if- and while-statements we have left out some premises of the form $\vdash_{\gg} P : \bot \longrightarrow d$. The reason is, that a premise in this form is always true. Indeed, $\vdash_{\gg} P : \bot \longrightarrow \bot$ can be easily inferred for any $P \in \mathbf{Stm}$.

We stress that this "tweaking" of rules in Fig. 5 in comparison with the general

$$\frac{d_1 \subseteq d_1' \quad \vdash_{\ll} P : d_1' \longrightarrow d_2' \quad d_2' \subseteq d_2}{\vdash_{\ll} P : d_1 \longrightarrow d_2} \text{(sub)}$$

$$\frac{}{\vdash_{\ll} x := o(x_1, \ldots, x_k) : (d \backslash \mathbf{Exp}_x) \cup \{o(x_1, \ldots, x_k)\} \longrightarrow d} \text{(prim)}$$

$$\frac{\vdash_{\ll} P_1 : d \longrightarrow d' \quad \vdash_{\ll} P_2 : d' \longrightarrow d''}{\vdash_{\ll} P_1; P_2 : d \longrightarrow d''} \text{(seq)}$$

$$\frac{\vdash_{\ll} P_1 : d_{\mathrm{t}} \longrightarrow d \quad \vdash_{\ll} P_2 : d_{\mathrm{f}} \longrightarrow d}{\vdash_{\ll} \mathit{if}\ o(x_1, \ldots, x_k)\ \mathit{then}\ P_1\ \mathit{else}\ P_2 : (d_{\mathrm{t}} \cap d_{\mathrm{f}}) \cup \{o(x_1, \ldots, x_k)\} \longrightarrow d} \text{(if)}$$

$$\frac{\vdash_{\ll} P : d' \longrightarrow (d' \cap d) \cup \{o(x_1, \ldots, x_k)\}}{\vdash_{\ll} \mathit{while}\ o(x_1, \ldots, x_k)\ \mathit{do}\ P : (d' \cap d) \cup \{o(x_1, \ldots, x_k)\} \longrightarrow d} \text{(while)}$$

Fig. 6. Type system from very busy expressions analysis

rules in Fig. 3 is not inherently necessary but is done solely for presentation here. We want to show that intuitive data-flow analyses lead to type systems whose rules correspond to our intuition, too.

4.2 Very Busy Expressions

Very busy expressions [7] is a backward must-analysis. The aim of this analysis is to determine, for each program point, which expressions will certainly be computed after the execution has passed that point and before any of the variables in these expressions are redefined. The results of this analysis may be used to decide whether it is worthwhile to precompute some expressions.

Let **Exp** be the set of all expressions, i.e. terms $o(x_1, \ldots, x_k)$. The underlying lattice D for very busy expressions analysis is $\mathcal{P}(\mathbf{Exp})$ (the set of all subsets of **Exp**) with the natural order. For each $x \in \mathbf{Var}$ let $\mathbf{Exp}_x \subseteq \mathbf{Exp}$ be the set of all these expressions that contain x. The analysis works in the backward direction. We define

$$[\![x := o(x_1, \ldots, x_k)]\!](d) = (d \backslash \mathbf{Exp}_x) \cup \{o(x_1, \ldots, x_k)\}$$

$$[\![o(x_1, \ldots, x_k)]\!]_{\mathsf{t}}(d) = [\![o(x_1, \ldots, x_k)]\!]_{\mathsf{f}}(d) = d \cup \{o(x_1, \ldots, x_k)\},$$

$\mathsf{branch}(d) = d$ and $\mathsf{combine}(d, d') = d'$. This gives us the analysis in [7]. The equivalent type system is given in Figure 6.

4.3 Secure Information Flow

Next we turn to secure information flow. The classical secure information flow analysis in the spirit of Denning and Denning [8] is a forward may-analysis. Let $\mathcal{L} = (\mathcal{L}, \vee, \wedge, \mathbf{1}, \mathbf{0})$ be a finite lattice of *security classes* for variables. The goal of the analysis is to verify that the initial value of some variable x may only affect the final value of a variable y if the final security class of y is at least as great as the initial security class of x.

Let $\mathbf{Var}' = \mathbf{Var}\dot{\cup}\{\mathsf{pc}\}$. The abstract state space D of the analysis is $\mathbf{Var}' \to \mathfrak{L}$ where the ordering is defined pointwise. Given $d \in D$ describing the security classes at the start of the program the program analysis attempts to find for each variable the security class that bounds from above the information that may flow to its final value. The security class $d(\mathsf{pc})$ contains an upper bound on the information that was used to make the decisions (at branching points) that caused the execution to reach the current program point; $d(\mathsf{pc})$ should be equal to $\mathbf{0}$ at the start of the program. The notation pc is a mnemonic for "program counter".

The analysis works in the forward direction. The semantics of statements and expressions is defined by

$$[\![x := o(x_1, \ldots, x_k)]\!](d) = d[x \mapsto d(x_1) \vee \cdots \vee d(x_k) \vee d(\mathsf{pc})]$$

$$[\![o(x_1, \ldots, x_k)]\!]_{\mathsf{t}}(d) = [\![o(x_1, \ldots, x_k)]\!]_{\mathsf{f}}(d) = d[\mathsf{pc} \mapsto d(x_1) \vee \cdots \vee d(x_k) \vee d(\mathsf{pc})] \ .$$

That is, the security level of the affected quantity (a variable or the program counter) is updated with the join of security levels of affecting quantities.

We see that the explicit flows in the program (assigning a variable to another) are handled directly by the analysis. The implicit flows (one variable controlling whether something is assigned to another one) are handled by increasing of the security class of the program counter. At the points where the conditional statements end, the $\mathsf{combine}$-function restores the security class of pc to the one before the conditional statement: $\mathsf{combine}(d, d') = d'[\mathsf{pc} \mapsto d(\mathsf{pc})]$. The branch-function is just the identity mapping.

The resulting type system is given in Figure 7. Actually, if we followed Figure 3 to the letter, then we would have had the following rule for *while*:

$$\frac{\vdash_{\gg} P : d'[\mathsf{pc} \mapsto d'(x_1) \vee \cdots \vee d'(x_k) \vee d'(\mathsf{pc})] \longrightarrow d' \quad d \sqsubseteq d'}{\vdash_{\gg} \mathit{while}\ o(x_1, \ldots, x_k)\ \mathit{do}\ P : d \longrightarrow d'[\mathsf{pc} \mapsto d(\mathsf{pc})]} \tag{5}$$

(in the conclusion both $[\![o(x_1, \ldots, x_k)]\!]_{\mathsf{f}}$ and $\mathsf{combine}$ change the value of $d'(\mathsf{pc})$ but $\mathsf{combine}$ is the last one). To get the rule (while) in Figure 7, we replace d' to the right of the arrow in the premise of (5) with $d'[\mathsf{pc} \mapsto d'(x_1) \vee \cdots \vee d'(x_k) \vee d'(\mathsf{pc})]$ which we can then rename into d' on both sides of the arrow. But now the new d' cannot be arbitrary: $d'(\mathsf{pc})$ must be at least $d'(x_i)$ for all $i \in \{1, \ldots, k\}$. The rule (5) and the modified rule (while) of Figure 7 are

$$\frac{d_1 \sqsubseteq d_1' \quad \vdash_{\gg} P : d_1' \longrightarrow d_2' \quad d_2' \sqsubseteq d_2}{\vdash_{\gg} P : d_1 \longrightarrow d_2}\ \text{(sub)}$$

$$\frac{}{\vdash_{\gg} x := o(x_1, \ldots, x_k) : d \longrightarrow d[x \mapsto d(x_1) \vee \cdots \vee d(x_k) \vee d(\mathsf{pc})]}\ \text{(prim)}$$

$$\frac{\vdash_{\gg} P_1 : d \longrightarrow d' \quad \vdash_{\gg} P_2 : d' \longrightarrow d''}{\vdash_{\gg} P_1; P_2 : d \longrightarrow d''}\ \text{(seq)}$$

$$\frac{\begin{array}{c}\vdash_{\gg} P_1 : d[\mathsf{pc} \mapsto d(x_1) \vee \cdots \vee d(x_k) \vee d(\mathsf{pc})] \longrightarrow d' \\ \vdash_{\gg} P_2 : d[\mathsf{pc} \mapsto d(x_1) \vee \cdots \vee d(x_k) \vee d(\mathsf{pc})] \longrightarrow d'\end{array}}{\vdash_{\gg} \mathit{if}\ o(x_1, \ldots, x_k)\ \mathit{then}\ P_1\ \mathit{else}\ P_2 : d \longrightarrow d'[\mathsf{pc} \mapsto d(\mathsf{pc})]}\ \text{(if)}$$

$$\frac{\vdash_{\gg} P : d' \longrightarrow d' \quad d \sqsubseteq d' \quad \forall i : d'(x_i) \leq d'(\mathsf{pc})}{\vdash_{\gg} \mathit{while}\ o(x_1, \ldots, x_k)\ \mathit{do}\ P : d \longrightarrow d'[\mathsf{pc} \mapsto d(\mathsf{pc})]}\ \text{(while)}$$

Fig. 7. Type system from forward information flow analysis

mutually derivable. Indeed, we can derive (while) from (5) by

$$\frac{\dfrac{\dfrac{\forall i : d'(x_i) \leq d'(\mathsf{pc})}{d'[\mathsf{pc} \mapsto d'(x_1) \vee \cdots \vee d'(x_k) \vee d'(\mathsf{pc})] = d'} \quad \vdash_{\gg} P : d' \longrightarrow d'}{\vdash_{\gg} P : d'[\mathsf{pc} \mapsto d'(x_1) \vee \cdots \vee d'(x_k) \vee d'(\mathsf{pc})] \longrightarrow d'}\ \text{(sub)} \qquad d \sqsubseteq d'}{\vdash_{\gg} \mathit{while}\ o(x_1, \ldots, x_k)\ \mathit{do}\ P : d \longrightarrow d'[\mathsf{pc} \mapsto d(\mathsf{pc})]}\ \text{(5)}\ .$$

Denote $d'' = d'[\mathsf{pc} \mapsto d'(x_1) \vee \cdots \vee d'(x_k) \vee d'(\mathsf{pc})]$; then $\forall i : d''(x_i) \leq d''(\mathsf{pc})$ and $d' \sqsubseteq d''$. We can derive (5) from (while) by first deriving

$$\frac{\dfrac{\vdash_{\gg} P : d'' \longrightarrow d' \quad \overline{\overline{d' \sqsubseteq d''}}}{\vdash_{\gg} P : d'' \longrightarrow d''}\ \text{(sub)} \quad \dfrac{d \sqsubseteq d' \quad \overline{\overline{d' \sqsubseteq d''}}}{d \sqsubseteq d''} \quad \overline{\overline{\forall i : d''(x_i) \leq d''(\mathsf{pc})}}}{\vdash_{\gg} \mathit{while}\ o(x_1, \ldots, x_k)\ \mathit{do}\ P : d \longrightarrow d''[\mathsf{pc} \mapsto d(\mathsf{pc})]}\ \text{(while)}$$

and then using the equality $d''[\mathsf{pc} \mapsto d(\mathsf{pc})] = d'[\mathsf{pc} \mapsto d(\mathsf{pc})]$ to the right of the arrow in the conclusion of this derivation (technically, this is also an application of the rule (sub)).

4.4 A Backward Secure Information Flow Analysis

We now look at an alternative secure information flow analysis, which is backward and must. The abstract state space D of this analysis is the same as that of the forwards analysis just considered. Given $d \in D$ describing the security classes at the end of the program the program analysis attempts to find for each variable the maximum initial security class, such that the initial value of that variable does not flow to the final value of any variable with a lower security class. The security class $d(\mathsf{pc})$ holds a similar upper bound for the decisions influencing whether an assignment to a variable is executed or not; it should equal $\mathbf{1}$ at the end of the program (i.e. at the beginning of the analysis).

Let us define an auxiliary function $la : \mathcal{P}(\mathbf{Var}') \times D \times \mathcal{L} \to D$ by

$$la(X, d, c)(x) = \begin{cases} d(x) \wedge c, & x \in X \\ d(x), & x \notin X \end{cases} .$$

We see that la is used to lower the security classes of the variables in X to the level of at most c. The semantics of statements $[\![x := o(x_1, \dots, x_k)]\!]$ is defined by

$$[\![x := o(x_1, \dots, x_k)]\!](d) = la(\{x_1, \dots, x_k, \mathsf{pc}\}, d[x \mapsto \mathbf{1}], d(x)) .$$

That is, before the statement $x := o(x_1, \dots, x_k)$ the variable x is dead hence the secrecy of its value is of no importance. As the values of x_i may influence the final values of the variables, they cannot be more secret than x is after the statement. The same holds for implicit flows through pc. The semantics of Boolean expressions is

$$[\![o(x_1, \dots, x_k)]\!]_{\mathsf{t}}(d) = [\![o(x_1, \dots, x_k)]\!]_{\mathsf{f}}(d) = la(\{x_1, \dots, x_k\}, d, d(\mathsf{pc})) .$$

At the beginning of a branching construct, the security classes of the variables x_i of its guard $o(x_1, \dots, x_k)$ should be lowered to the level of security classes that the variables assigned to in the branches have at the end of the branching construct. The meet of these security classes is collected into $d(\mathsf{pc})$. We do not want to collect anything more into $d(\mathsf{pc})$, for example the security classes of variables that are assigned to after the end of the branching construct. Hence we let the branch-function reset the security class of pc. We define $\mathsf{branch}(d) = d[\mathsf{pc} \mapsto \mathbf{1}]$. The security class of pc has to be restored afterwards, so we put $\mathsf{combine}(d, d') = d'[\mathsf{pc} \mapsto d(\mathsf{pc}) \wedge d'(\mathsf{pc})]$. Recall that d was the abstract value at the incoming edge of the corresponding branch-node (in the inverse control flow graph).

The type system derived from this analysis is given in Figure 8. But again we have simplified some of the rules. If we had followed Figure 4 literally, then the rule for *while*-statements would have been

$$\frac{\vdash_{\ll} P : d' \longrightarrow d'[x_i \mapsto d'(x_i) \wedge d'(\mathsf{pc})]_{i=1}^{k} \sqcap d[\mathsf{pc} \mapsto \mathbf{1}]}{\vdash_{\ll} \mathit{while}\ o(x_1, \dots, x_k)\ \mathit{do}\ P : d'[x_i \mapsto d'(x_i) \wedge d'(\mathsf{pc})]_{i=1}^{k} \sqcap d \longrightarrow d} . \quad (6)$$

Indeed, if $d(\mathsf{pc}) = \mathbf{1}$ (as set by branch) then $[\![b]\!]_{\mathsf{f}}(d) = d$. In the conclusion of the rule, the application of $\mathsf{combine}$ undoes the setting of $d(\mathsf{pc})$ to $\mathbf{1}$. We modify this rule by denoting $d'[x_i \mapsto d'(x_i) \wedge d'(\mathsf{pc})]_{i=1}^{k}$ by d', giving us

$$\frac{\vdash_{\ll} P : d' \longrightarrow d' \sqcap d[\mathsf{pc} \mapsto \mathbf{1}] \quad \forall i : d'(x_i) \leq d'(\mathsf{pc})}{\vdash_{\ll} \mathit{while}\ o(x_1, \dots, x_k)\ \mathit{do}\ P : d' \sqcap d \longrightarrow d} . \quad (7)$$

$$\frac{d_1 \sqsubseteq d_1' \quad \vdash_{\ll} P : d_1' \longrightarrow d_2' \quad d_2' \sqsubseteq d_2}{\vdash_{\ll} P : d_1 \longrightarrow d_2} \text{(sub)}$$

$$\frac{}{\begin{array}{l}\vdash_{\ll} x := o(x_1, \ldots, x_k) : \\ \qquad d[\mathsf{pc} \mapsto d(\mathsf{pc}) \wedge d(x), x \mapsto \mathbf{1}][x_i \mapsto d(x_i) \wedge d(x)]_{i=1}^k \longrightarrow d\end{array}} \text{(prim)}$$

$$\frac{\vdash_{\ll} P_1 : d'' \longrightarrow d' \quad \vdash_{\ll} P_2 : d' \longrightarrow d}{\vdash_{\ll} P_1; P_2 : d'' \longrightarrow d} \text{(seq)}$$

$$\frac{\vdash_{\ll} P_1 : d' \longrightarrow d[\mathsf{pc} \mapsto \mathbf{1}] \quad \vdash_{\ll} P_2 : d' \longrightarrow d[\mathsf{pc} \mapsto \mathbf{1}] \quad \begin{array}{c}\forall i : d'(x_i) \leq d'(\mathsf{pc}) \\ d'(\mathsf{pc}) \leq d(\mathsf{pc})\end{array}}{\vdash_{\ll} \mathit{if}\ o(x_1, \ldots, x_k)\ \mathit{then}\ P_1\ \mathit{else}\ P_2 : d' \longrightarrow d} \text{(if)}$$

$$\frac{\vdash_{\ll} P : d' \longrightarrow d' \quad \forall i : d'(x_i) \leq d'(\mathsf{pc}) \quad d' \sqsubseteq d[\mathsf{pc} \mapsto \mathbf{1}]}{\vdash_{\ll} \mathit{while}\ o(x_1, \ldots, x_k)\ \mathit{do}\ P : d'[\mathsf{pc} \mapsto d'(\mathsf{pc}) \wedge d(\mathsf{pc})] \longrightarrow d} \text{(while)}$$

Fig. 8. Type system from backward information flow analysis

The new d' cannot be completely arbitrary—$d'(x_i)$ must be smaller or equal to $d'(\mathsf{pc})$. This side condition is added to the rule. The original d' is then just something that is larger than the new d'. As the original d' only occurs on the left side of the arrow in the premise we can safely replace it with the new d'. Abbreviate $d'' = d'[x_i \mapsto d'(x_i) \wedge d'(\mathsf{pc})]_{i=1}^k$, then $d'' \sqsubseteq d'$ and $\forall i : d''(x_i) \leq d''(\mathsf{pc})$. We can derive (7) from (6) by

$$\frac{\dfrac{\forall i : d'(x_i) \leq d'(\mathsf{pc})}{d' = d''} \quad \dfrac{\dfrac{\vdash_{\ll} P : d' \longrightarrow d' \sqcap d[\mathsf{pc} \mapsto 1] \quad \dfrac{\forall i : d'(x_i) \leq d'(\mathsf{pc})}{d' = d''}}{\vdash_{\ll} P : d' \longrightarrow d'' \sqcap d[\mathsf{pc} \mapsto 1]} \text{(sub)}}{\vdash_{\ll} \mathit{while}\ o(x_1, \ldots, x_k)\ \mathit{do}\ P : d'' \sqcap d \longrightarrow d} \text{(6)}}{\vdash_{\ll} \mathit{while}\ o(x_1, \ldots, x_k)\ \mathit{do}\ P : d' \sqcap d \longrightarrow d} \text{(sub)}$$

and (6) from (7) by

$$\frac{\dfrac{\overline{\overline{d'' \sqsubseteq d'}} \quad \vdash_{\ll} P : d' \longrightarrow d'' \sqcap d[\mathsf{pc} \mapsto 1]}{\vdash_{\ll} P : d'' \longrightarrow d'' \sqcap d[\mathsf{pc} \mapsto 1]} \text{(sub)} \quad \overline{\overline{\forall i : d''(x_i) \leq d''(\mathsf{pc})}}}{\vdash_{\ll} \mathit{while}\ o(x_1, \ldots, x_k)\ \mathit{do}\ P : d'' \sqcap d \longrightarrow d} \text{(7)} \quad .$$

We further modify (7) by denoting $d' \sqcap d[\mathsf{pc} \mapsto \mathbf{1}]$ by d'. This gives us the rule (while) in Figure 8. Now we also require that the new d' is smaller or equal to $d[\mathsf{pc} \mapsto \mathbf{1}]$. Abbreviate $d^* = d' \sqcap d[\mathsf{pc} \mapsto 1]$, $d^\dagger = d'[\mathsf{pc} \mapsto d'(\mathsf{pc}) \wedge d(\mathsf{pc})]$ and

$d^{\ddagger} = d^*[\mathsf{pc} \mapsto d^*(\mathsf{pc}) \wedge d(\mathsf{pc})]$. We can derive (while) from (7) by

$$
\dfrac{\dfrac{d' \sqsubseteq d[\mathsf{pc} \mapsto \mathbf{1}]}{d^{\dagger} = d' \sqcap d} \qquad \dfrac{\dfrac{\vdash_{\ll} P : d' \longrightarrow d' \qquad \dfrac{d' \sqsubseteq d[\mathsf{pc} \mapsto \mathbf{1}]}{d' = d^*}}{\vdash_{\ll} P : d' \longrightarrow d^*}\,(\mathrm{sub}) \qquad \forall i : d'(x_i) \leq d'(\mathsf{pc})}{\vdash_{\ll} \mathit{while}\ o(x_1, \ldots, x_k)\ \mathit{do}\ P : d' \sqcap d \longrightarrow d}\,(7)}{\vdash_{\ll} \mathit{while}\ o(x_1, \ldots, x_k)\ \mathit{do}\ P : d^{\dagger} \longrightarrow d}\,(\mathrm{sub})
$$

and (7) from (while) by

$$
\dfrac{\overline{\overline{d' \sqcap d = d^{\ddagger}}} \qquad \dfrac{\dfrac{\overline{\overline{d^* \sqsubseteq d'}} \quad \vdash_{\ll} P : d' \longrightarrow d^*}{\vdash_{\ll} P : d^* \longrightarrow d^*}\,(\mathrm{sub}) \qquad \dfrac{\forall x_i : d'(x_i) \leq d'(\mathsf{pc})}{\forall x_i : d^*(x_1) \leq d^*(\mathsf{pc})} \qquad \overline{\overline{d^* \sqsubseteq d[\mathsf{pc} \mapsto \mathbf{1}]}}}{\vdash_{\ll} \mathit{while}\ o(x_1, \ldots, x_k)\ \mathit{do}\ P : d^{\ddagger} \longrightarrow d}\,(\mathrm{while})}{\vdash_{\ll} \mathit{while}\ o(x_1, \ldots, x_k)\ \mathit{do}\ P : d' \sqcap d \longrightarrow d}\,(\mathrm{sub})
$$

.

Also, according to Figure 4 the rule (if) should have been

$$
\dfrac{\vdash_{\ll} P_1 : d' \longrightarrow d[\mathsf{pc} \mapsto \mathbf{1}] \quad \vdash_{\ll} P_2 : d' \longrightarrow d[\mathsf{pc} \mapsto \mathbf{1}]}{\begin{array}{l}\vdash_{\ll} \mathit{if}\ \ o(x_1, \ldots, x_k)\ \mathit{then}\ P_1\ \mathit{else}\ P_2 : \\ \qquad\qquad d'[x_i \mapsto d'(x_i) \wedge d'(\mathsf{pc})]_{i=1}^{k}[\mathsf{pc} \mapsto d(\mathsf{pc}) \wedge d'(\mathsf{pc})] \longrightarrow d\end{array}} \tag{8}
$$

Here we denote $d'[x_i \mapsto d'(x_i) \wedge d'(\mathsf{pc})]_{i=1}^{k}[\mathsf{pc} \mapsto d(\mathsf{pc}) \wedge d'(\mathsf{pc})]$ by d', this gives us the rule (if) in Figure 8. We have to add the conditions $d'(x_i) \leq d'(\mathsf{pc}) \leq d(\mathsf{pc})$ for the new d', because for the original d', these conditions held for $d'[x_i \mapsto d'(x_i) \wedge d'(\mathsf{pc})]_{i=1}^{k}[\mathsf{pc} \mapsto d(\mathsf{pc}) \wedge d'(\mathsf{pc})]$. The mutual derivability of (if) and (8) is obvious.

4.5 Hoare Logic

Hoare logic can be derived as a "type system" according to our method from the following backward analysis (essentially a weakest precondition calculus). We define D to be the set of all predicates on states, i.e., $D = (\mathbf{Var} \to \mathbb{Z}) \to \mathbb{B}$ and $\sqsubseteq$ to be pointwise implication, so $\phi \sqsubseteq \psi$ iff $\phi\, s$ implies $\psi\, s$ for any state s. We define $[\![x := a]\!]\, \phi = \lambda s.\, \phi\, (s[x \mapsto [\![a]\!]\, s])$, $[\![b]\!]_{\mathsf{t}} \phi = \lambda s.\, [\![b]\!]\, s \Rightarrow \phi\, s$, $[\![b]\!]_{\mathsf{f}}\, \phi = \lambda s.\, \neg [\![b]\!]\, s \Rightarrow \phi\, s$, $\mathsf{branch}(\phi) = \phi$, $\mathsf{combine}(\phi, \phi') = \phi'$.

We get that $[\![P]\!]_{\ll}(\psi) = \mathit{wp}(\psi)$ where wp means the weakest precondition in the conventional sense. The corresponding "type system" is presented in Figure 9, which is deductively equivalent to the standard formulation of Hoare logic.

It is also possible to derive a formulation of Hoare logic from a forward analysis

$$\frac{\phi_1 \models \phi_1' \quad \vdash_{\ll} P : \phi_1' \longrightarrow \phi_2' \quad \phi_2' \models \phi_2}{\vdash_{\ll} P : \phi_1 \longrightarrow \phi_2} \text{(sub)}$$

$$\frac{}{\vdash_{\ll} x := a : \phi[x \mapsto a] \longrightarrow \phi} \text{(prim)}$$

$$\frac{\vdash_{\ll} P_1 : \phi'' \longrightarrow \phi' \quad \vdash_{\ll} P_2 : \phi' \longrightarrow \phi}{\vdash_{\ll} P_1; P_2 : \phi'' \longrightarrow \phi} \text{(seq)}$$

$$\frac{\vdash_{\ll} P_1 : \phi_{\mathrm{t}} \longrightarrow \phi \quad \vdash_{\ll} P_2 : \phi_{\mathrm{f}} \longrightarrow \phi}{\vdash_{\ll} \mathit{if}\ b\ \mathit{then}\ P_1\ \mathit{else}\ P_2 : (b \Rightarrow \phi_{\mathrm{t}}) \wedge (\neg b \Rightarrow \phi_{\mathrm{f}}) \longrightarrow \phi} \text{(if)}$$

$$\frac{\vdash_{\ll} P : \phi' \longrightarrow (b \Rightarrow \phi') \wedge (\neg b \Rightarrow \phi)}{\vdash_{\ll} \mathit{while}\ b\ \mathit{do}\ P : (b \Rightarrow \phi') \wedge (\neg b \Rightarrow \phi) \longrightarrow \phi} \text{(while)}$$

Fig. 9. "Type system" from standard weakest precondition calculus (= Hoare logic)

$$\mathfrak{P}(P, T, d_1) = \begin{cases} \mathfrak{P}(P, T', d_1') & R \text{ is (sub)} \\ [] & R \text{ is (prim)} \\ \mathfrak{P}(P_1, T_1', d_1) \cdot \mathfrak{P}(P_2, T_2', d') & R \text{ is (seq)} \\ \mathfrak{P}(P_1, T_1', [\![b]\!]_{\mathsf{t}}(\mathsf{branch}(d_1))) \cdot \mathfrak{P}(P_2, T_2', [\![b]\!]_{\mathsf{f}}(\mathsf{branch}(d_1))) & \\ & R \text{ is (if)} \\ d' \cdot \mathfrak{P}(P_1, T', [\![b]\!]_{\mathsf{t}}(d')) & R \text{ is (while)}, \end{cases}$$

here R is the last rule in T; T' is the rest of the tree T (T_1' and T_2', if R has two descendants); b, P_1, P_2, d' and d_1' have the same meaning as in the rules.

Fig. 10. Certificate for $\vdash_{\gg} P : d_1 \longrightarrow d_2$ with type inference tree T

(a strongest postcondition calculus). We refrain from presenting the details here.

5 Certification and Checking of Typings

In this section we describe how to convince another party that a program P has a given type $d_1 \longrightarrow d_2$. We assume that we know a type derivation T with the root $\vdash_{\gg} P : d_1 \longrightarrow d_2$ (we concentrate on the type system in Figure 3, the system in Figure 4 is handled similarly). The checker only has to compare types and compute primitive semantic functions.

The *certificate (proof)* for $\vdash_{\gg} P : d_1 \longrightarrow d_2$ has the form $(d_1, \mathfrak{P}(P, T, d_1), d_2)$ with an easily constructible $\mathfrak{P}(P, T, d_1) \in D^*$. The function $\mathfrak{P}$ is defined in Figure 10.

Let us now describe the *checking (verification) algorithm* $\mathfrak{V}$. It takes as arguments a program P, an abstract state d_1' and a certificate $\mathsf{p} \in D^*$ and either fails or returns an abstract state d_2' and $\mathsf{p}' \in D^*$. Failure implies that the proof

$$
\begin{array}{ll}
\textbf{case } P \textbf{ of} & \\
\quad p: & \textbf{return } (\llbracket p \rrbracket(d), \mathsf{p}) \\
\quad P_1; P_2: & (d', \mathsf{p}') \leftarrow \mathfrak{V}(P_1, d, \mathsf{p}) \\
& \textbf{return } \mathfrak{V}(P_2, d', \mathsf{p}') \\
\quad \textit{if } b \textit{ then } P_1 \textit{ else } P_2: & (d_1, \mathsf{p}') \leftarrow \mathfrak{V}(P_1, \llbracket b \rrbracket_{\mathsf{t}}(\mathsf{branch}(d)), \mathsf{p}) \\
& (d_2, \mathsf{p}'') \leftarrow \mathfrak{V}(P_2, \llbracket b \rrbracket_{\mathsf{f}}(\mathsf{branch}(d)), \mathsf{p}') \\
& \textbf{return } (\mathsf{combine}(d, d_1 \sqcup d_2), \mathsf{p}'') \\
\quad \textit{while } b \textit{ do } P_1: & d' \leftarrow \textbf{head}(\mathsf{p}) \\
& \textbf{check}(d' \sqsupseteq \mathsf{branch}(d)) \\
& (d'', \mathsf{p}') \leftarrow \mathfrak{V}(P_1, \llbracket b \rrbracket_{\mathsf{t}}(d'), \textbf{tail}(\mathsf{p})) \\
& \textbf{check}(d'' \sqsubseteq d') \\
& \textbf{return } (\mathsf{combine}(d, \llbracket b \rrbracket_{\mathsf{f}}(d')), \mathsf{p}')
\end{array}
$$

Fig. 11. The checking algorithm $\mathfrak{V}(P, d, \mathsf{p})$

is not accepted. To verify the proof (d_1, p, d_2) for the program P, one invokes $\mathfrak{V}(P, d_1, \mathsf{p})$, receives (d_2', p') and accepts if $d_2' \sqsubseteq d_2$. The function $\mathfrak{V}$ is defined in Figure 11. Here **check** checks that its argument is true, and fails otherwise. The function **head** also fails on empty input.

Lemma 9 *If $d_1, d_1' \in D$ and p is such that $d_1 \sqsupseteq d_1'$ and $\mathfrak{V}(P, d_1, \mathsf{p}) = (d_2, \mathsf{p}')$ for some $d_2 \in D$ and p' then also $\mathfrak{V}(P, d_1', \mathsf{p}) = (d_2', \mathsf{p}')$ for some $d_2' \sqsubseteq d_2$.*

PROOF. Induction over the call tree of $\mathfrak{V}$. If the invocation of $\mathfrak{V}$ makes no further calls to $\mathfrak{V}$ then P is a primitive statement p and $d_2 = \llbracket p \rrbracket(d_1)$, $d_2' = \llbracket p \rrbracket(d_1')$; $d_2' \sqsubseteq d_2$ because of the monotonicity of $\llbracket p \rrbracket$. If further calls to $\mathfrak{V}$ are made then the arguments to these calls are computed monotonically from the value of d. Hence the results of these calls are also monotonic with respect to d. If the program P is a *while*-loop then $\mathfrak{V}$ contains some checks whether a value d' read from the proof p is larger than d or some value computed monotonically from d. If these checks pass then they pass also if d is made smaller. The right component of the return value of $\mathfrak{V}$ is just the "unconsumed" part of the proof p, it does not depend on d. □

Theorem 10 *If $\vdash_{\gg} P : d_1 \longrightarrow d_2$ is derivable and T is the corresponding derivation, then the invocation of $\mathfrak{V}(P, d_1, \mathfrak{P}(P, T, d_1) \cdot \mathsf{p}')$ does not fail for any p', and it returns a pair of the form (d_2', p') where $d_2' \sqsubseteq d_2$.*

PROOF. Let $\mathsf{p} = \mathfrak{P}(P, T, d_1)$. Induction over the structure of the inference tree T. Consider the last applied rule R. If R is (sub) then $\mathfrak{V}(P, d_1, \mathsf{p}) = \mathfrak{V}(P, d_1', \mathsf{p} \cdot \mathsf{p}')$ for some $d_1' \sqsubseteq d_1$ and the claim of the theorem follows from the induction assumption and Lemma 9. If R is (prim) then $\mathfrak{V}$ always succeeds, does not consume any part of the proof, and returns $\llbracket p \rrbracket(d_1)$ which is less than or equal to d_2 by Theorem 7. If R is (seq) and P is $P_1; P_2$ then

$\mathfrak{V}$ is first invoked on P_1; by the induction assumption, this invocation consumes precisely the prefix $\mathfrak{P}(P_1, T_1', d_1)$ of the proof and returns an abstract value $d'' \in D$, such that $d'' \sqsubseteq d'$, $\models_{\gg} P_2 : d' \longrightarrow d_2$ and T_2' is the corresponding inference tree. Afterwards $\mathfrak{V}$ is invoked on P_2 and the invocation consumes the prefix $\mathfrak{P}(P_1, T_2', d')$, succeeds and returns an abstract value $d_2' \sqsubseteq d_2$ by the induction assumption and Lemma 9. Similarly, if R is (if) and P is *if* b *then* P_1 *else* P_2 then the invocation of $\mathfrak{V}$ on P_1 consumes precisely the prefix $\mathfrak{P}(P_1, T_1', [\![b]\!]_{\mathsf{t}}(d_1))$ and the following invocation of $\mathfrak{V}$ on P_2 consumes precisely the prefix $\mathfrak{P}(P_2, T_2', [\![b]\!]_{\mathsf{f}}(d_1))$ of the rest. This invocations return the abstract values $\hat{d_1}$ and $\hat{d_2}$ that are both less than or equal to some d', such that $\mathsf{combine}(d_1, d') \sqsubseteq d_2$. Hence $\mathfrak{V}$ returns a value $d_2' \sqsubseteq d_2$. If R is (while) and P is *while* b *do* P_1 then the first element d' of the proof must be such, that $d' \sqsupseteq d$ (by the construction of $\mathfrak{P}(P, T, d_1)$) and $\models_{\gg} P_1 : [\![b]\!]_{\mathsf{t}}(d') \longrightarrow d'$. By induction assumption, invocation of $\mathfrak{V}$ on P_1 succeeds, consumes the prefix $\mathfrak{P}(P_1, T', [\![b]\!]_{\mathsf{t}}(d'))$ of the proof, and returns some value that is less than or equal to d'. Hence the second check in $\mathfrak{V}$ succeeds as well. The invocation of $\mathfrak{V}$ returns $\mathsf{combine}(d, [\![b]\!]_{\mathsf{f}}(d'))$ which is equal to d_2. $\square$

Theorem 11 *If* p *is such that* $\mathfrak{V}(P, d_1, \mathsf{p}) = (d_2, \mathsf{p}')$*, then* $\models_{\gg} P : d_1 \longrightarrow d_2$*.*

PROOF. We use the proof p to construct an inference tree for $\models_{\gg} P : d_1 \longrightarrow d_2$. The construction proceeds by induction over the structure of P. If P is a primitive statement p then $\mathfrak{V}$ on input P and d_1 returns $d_2 = [\![p]\!](d_1)$ and $\models_{\gg} P : d_1 \longrightarrow d_2$ can be inferred with the rule (prim). If P is $P_1; P_2$ then $\mathfrak{V}$ is successfully invoked on P_1 and d_1, returning d'; and then on P_2 and d', returning d_2. By the induction assumption, $\models_{\gg} P_1 : d_1 \longrightarrow d'$ and $\models_{\gg} P_2 : d' \longrightarrow d_2$ can be inferred, and a further application of the rule (seq) completes the inference of $\models_{\gg} P : d_1 \longrightarrow d_2$. If P is *if* b *then* P_1 *else* P_2 then $\mathfrak{V}$ is successfully invoked on P_1 and $[\![b]\!]_{\mathsf{t}}(d_1)$, returning d_1'; and then on P_2 and $[\![b]\!]_{\mathsf{f}}(d_1)$, returning d_2'. The value d_2 is equal to $\mathsf{combine}(d_1, d_1' \sqcup d_2')$. By induction assumption, we can infer $\models_{\gg} P_1 : [\![b]\!]_{\mathsf{t}}(d_1) \longrightarrow d_1'$ and $\models_{\gg} P_2 : [\![b]\!]_{\mathsf{f}}(d_1) \longrightarrow d_2'$. An application of the rule (if) completes the inference of $\models_{\gg} P : d_1 \longrightarrow d_2$. If P is *while* b *do* P_1 then the first value d' in p must be greater or equal to d_1. When $\mathfrak{V}$ is invoked on P_1 and $[\![b]\!]_{\mathsf{t}}(d_1)$, it returns some $d'' \sqsubseteq d'$, because the second check must also succeed. By induction assumption, $\models_{\gg} P_1 : [\![b]\!]_{\mathsf{t}}(d') \longrightarrow d''$ can be inferred. We can now use the rule (sub) to infer $\models_{\gg} P_1 : [\![b]\!]_{\mathsf{t}}(d') \longrightarrow d'$ and the rule (while) to infer $\models_{\gg} P : d_1 \longrightarrow d_2$. $\square$

The checking algorithm $\mathfrak{V}$ actually "reads" the certificate p only if it encounters a *while*-loop in the program. It must remember the value d' as long as it verifies the loop body because the value of d' is used in a check after that. If space at the checking device is at premium then the proof may repeat the loop invariant d' after the proof for the loop body, i.e. we define

$\mathfrak{P}^\star(\textit{while } b \textit{ do } P_1, T, d_1) = d' \cdot \mathfrak{P}^\star(P_1, T', [\![b]\!]_{\mathsf{t}}(d')) \cdot d'$ for the (while)-rule and as $\mathfrak{P}$ for other rules. Now $\mathfrak{V}$ no longer has to store d' while it verifies the loop body. But $\mathfrak{P}^\star$ may cheat—it may make the two copies of d' different. To avoid this we let $\mathfrak{V}$ store a *cryptographic message digest* of the first copy of d' and compare it to the second copy. Although such solution does not eliminate the storage requirements completely, it considerably reduces them—a digest will typically require 160 to 256 bits [9], considerably less than the entire d'.

Let $H : \{0,1\}^* \rightarrow \{0,1\}^k$ be a *collision-resistant hash function* where k is typically 160 or 256 (as the arguments of H are arbitrary bit-strings, we assume that H can be applied to anything representable). Intuitively (see [10] for precise definitions) this means that it is very hard to find *collisions* for H—pairs (x_1, x_2) where $x_1 \neq x_2$ but $H(x_1) = H(x_2)$. Let $\mathfrak{V}^H(\textit{while } b \textit{ do } P_1, d, \mathsf{p})$ be the following algorithm

$$\begin{aligned}
&d' \leftarrow \mathbf{head}(\mathsf{p}); \mathbf{check}(d' \sqsupseteq \mathsf{branch}(d)); h \leftarrow H(d');\\
&(d'', \mathsf{p}') \leftarrow \mathfrak{V}^H(P_1, [\![b]\!]_{\mathsf{t}}(d'), \mathbf{tail}(\mathsf{p})); \bar{d} \leftarrow \mathbf{head}(\mathsf{p}'); \mathbf{check}(h = H(\bar{d}));\\
&\mathbf{check}(d'' \sqsubseteq \bar{d}); \mathbf{return}\ (\mathsf{combine}(d, [\![b]\!]_{\mathsf{f}}(\bar{d})), \mathbf{tail}(\mathsf{p}'))
\end{aligned}$$

and let $\mathfrak{V}^H$ be defined like $\mathfrak{V}$ for other programs. Using more advanced integrity structures (authenticated search trees [11]), the space requirement per loop may be decreased even more (to the size of a single identifier, 2 or 4 bytes).

Theorem 12 *There exists a polynomial-time algorithm K such that, if p, P and d_1 are such that $\mathfrak{V}^H(P, d_1, \mathsf{p}) = d_2$ but $\not\models\!\!\gg P : d_1 \longrightarrow d_2$ then $K^H(P, d_1, \mathsf{p})$ outputs a collision for H.*

PROOF. The algorithm K executes both $\mathfrak{V}(P, d_1, \mathsf{p})$ (ignoring the second copy of loop invariants) and $\mathfrak{V}^H(P, d_1, \mathsf{p})$ in a lock-step fashion up to the point where the data that they are accessing at this point differs for the first time. A step in either $\mathfrak{V}$ or $\mathfrak{V}^H$ is either a further call to $\mathfrak{V}$ or $\mathfrak{V}^H$, a **return**-statement, or a **check**-statement, except the statement $\mathbf{check}(h = H(\bar{d}))$. This statement, as well as the computation of h or reading the first element of p do not constitute as separate steps, they are grouped with the following statement. We see that the only points where $\mathfrak{V}$ and $\mathfrak{V}^H$ actually access different variables are at the end of handling the programs of the form *while b do P_1*. Namely, in two last statements $\mathfrak{V}$ makes use of the variable d' while $\mathfrak{V}^H$ uses $\bar{d}$. Also, the first difference in the executions of cannot be the **return**-statement of that case, because if $d' \neq \bar{d}$ then this is discovered already at the point where $\mathfrak{V}$ executed $\mathbf{check}(d'' \sqsubseteq d')$ and $\mathfrak{V}^H$ executed $\mathbf{check}(d'' \sqsubseteq \bar{d})$. If K stops the execution of $\mathfrak{V}$ and $\mathfrak{V}^H$ at a point where $d' \neq bard$, but the statement $\mathbf{check}(h = H(\bar{d}))$ succeeded, then it has found two differing values—d' and $\bar{d}$—that are mapped to the same bit-string by H.

The algorithm K outputs the pair $(d', \bar{d})$. $\Box$

6 Related Work

Previous work on relating program analyses and type systems has focused on control-flow analyses (analyses that attempt to show that the program does not make an illegal step) [2,12–14]. These type systems have a more complex structure than the ones considered here and typically feature universally and existentially quantified implications (big meets and joins of implications). The type of a program fragment can contain as much information as the denotational abstract semantics of that fragment. In contrast, a type in our type system corresponds to just one point (pair of abstract states) in the graph of the abstract semantics.

The closest to our work is probably the work of Naik and Palsberg [3]. They consider the same language as we do and their abstract semantics can be cast in our terms as well. But they only consider the case where the underlying lattice D is a powerset (and they do not consider branch and combine functions). And similarly to the systems mentioned above, they employ quantified implications and a type again corresponds to the entire semantics.

The certification of typing of Section 5 is reminiscent of Necula's and Lee's proof-carrying code [15]. Whereas the preliminary versions included the whole proof (derivation tree) in the certificate, later work [16] has concentrated on including only the information that is expensive to determine; this is similar to only including some typing information for loops. The same approach is present in the *lightweight bytecode verification* of Rose [17]—a control-flow graph based approach where the labels of all back-edges are included in the certificate. An implementation and soundness and completeness proofs of this approach are given in [18]. The *abstraction-carrying code* of Albert et al. [19] is promoting exactly the same idea, to make checking the result of a static program analysis easy by shipping the program together with a certificate containing the essential information.

The type systems for secure information flow presented in this paper are similar in their spirit to those of Andrews and Reitman [20] (a logic, really) and Amtoft and Banerjee [21] and more precise than those by Volpano et al. [4]. The reason is that, for us, a type is a pair of security class assignments to variables (corresponding to their security classes before and after a program run), while Volpano et al. insist (for reasons that escape us) that security classes of variables are invariant. During the revision of this paper we learned that a secure information flow type system very similar to ours has also been proposed recently by Hunt and Sands [22]. But their work does not describe any general method for obtaining type systems for data-flow analyses.

7 Conclusions and Future Work

We have shown how imperative data-flow analyses definable in a certain framework can be equivalently cast as type systems, proceeding from a very basic idea about what a type of an imperative program ought to be. The resulting type systems are, to our judgement, extremely intuitive and well-motivated, so we believe we have once more witnessed how useful it can be to have the basic concepts right in any theory. We also believe that our type systems are practically applicable in certified software.

As future work, we plan to further explore the limits of our analysis framework (especially the applications of branch and combine functions) and to see if the framework can be liberalized in interesting ways while keeping the type systems simple. We also plan to study systematic description of data-flow analyses in terms of logics. Yet another direction will be type systems corresponding to analyses for a language with procedures. Here we hope to be able to reuse the main idea of the standard Hoare logic treatment of procedures.

Acknowledgements

We are grateful to the anonymous referees for their suggestions and remarks.

This research was partially supported by the Estonian Science Foundation grants no. 5567 and 6095 and by the EU FP5 IST thematic network project APPSEM II.

References

[1] P. Cousot, Types as abstract interpretations, in: Conf. Record of 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL '97, ACM Press, 1997, pp. 316–331.

[2] J. Palsberg, P. O'Keefe, A type system equivalent to flow analysis, ACM Trans. on Program. Lang. and Syst. 17 (4) (1995) 576–599.

[3] M. Naik, J. Palsberg, A type system equivalent to a model checker, in: S. Sagiv (Ed.), Proc. of 14th Europ. Symp. on on Programming, ESOP 2005, Vol. 3444 of Lect. Notes in Comput. Sci., Springer, 2005, pp. 374–388.

[4] D. Volpano, G. Smith, C. Irvine, A sound type system for secure flow analysis, J. of Comput. Security 4 (2–3) (1996) 167–187.

[5] P. Laud, Semantics and program analysis of computationally secure information flow, in: D. Sands (Ed.), Proc. of 10th Europ. Symp. on Programming, ESOP 2001, Vol. 2028 of Lect. Notes in Comput. Sci., Springer, 2001, pp. 77–91.

[6] P. Laud, Handling encryption in analyses for secure information flow, in: P. Degano (Ed.), Proc. of 12th Europ. Symp. on Programming, ESOP 2003, Vol. 2618 of Lect. Notes in Comput. Sci., Springer, 2003, pp. 159–173.

[7] F. Nielson, H. Nielson, C. Hankin, Principles of Program Analysis, Springer, 1999.

[8] D. Denning, P. Denning, Certification of programs for secure information flow, Comm. of ACM 20 (7) (1977) 504–513.

[9] Secure Hash Standard, Federal Information Processing Standards Publication 180-2 (FIPS PUB 180-2) (1 Aug. 2002).

[10] D. R. Stinson, Cryptography Theory and Practice, 2nd ed., CRC Press, 2002.

[11] A. Buldas, P. Laud, H. Lipmaa, Eliminating counterevidence with applications to accountable certificate management, J. of Comput. Security 10 (3) (2002) 273–296.

[12] N. Heintze, Control-flow analysis and type systems, in: A. Mycroft (Ed.), Proc. of 2nd Int. Static Analysis Symp., SAS '95, Vol. 983 of Lect. Notes in Comput. Sci., Springer, 1995, pp. 189–206.

[13] J. Palsberg, C. Pavlopoulou, From polyvariant flow information to intersection and union types, J. of Funct. Program. 22 (3) (2001) 263–317.

[14] T. Amtoft, F. A. Turbak, Faithful translations between polyvariant flows and polymorphic types, in: G. Smolka (Ed.), Proc. of 9th Europ. Symp. on Programming, ESOP 2000, Vol. 1782 of Lect. Notes in Comput. Sci., Springer, 2000, pp. 26–40.

[15] G. C. Necula, Proof-carrying code, in: Conf. Record of 24th SIGPLAN-SIGACT Symp. on Principles of Programming Lanaguges, POPL 97, ACM Press, 1997, pp. 106–119.

[16] G. C. Necula, S. P. Rahul, Oracle-based checking of untrusted software, in: Conf. Record of 28th SIGPLAN-SIGACT Symp. on Principles of Programming Lanaguges, POPL 2001, ACM Press, 2001, pp. 142–154.

[17] E. Rose, Lightweight bytecode verification, J. of Autom. Reasoning 31 (3–4) (2003) 303–334.

[18] G. Klein, T. Nipkow, Verified lightweight bytecode verification, Concurrency and Computation: Practice and Experience 13 (13) (2001) 1133–1151.

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

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

[21] T. Amtoft, A. Banerjee, Information flow analysis in logical form, in: R. Giacobazzi (Ed.), Proc. of 11th Int. Static Analysis Symp., SAS 2004, Vol. 3148 of Lect. Notes in Comput. Sci., Springer, 2004, pp. 100–115.

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