On Flow-Sensitive Security Types

Sebastian Hunt
Department of Computing
School of Informatics, City University
London EC1V OHB, UK
seb@soi.city.ac.uk

David Sands
Department of Computer Science and Engineering,
Chalmers University of Technology
Göteborg, Sweden
dave@chalmers.se

Abstract

This article investigates formal properties of a family of semantically sound flow-sensitive type systems for tracking information flow in simple While programs. The family is indexed by the choice of flow lattice.

By choosing the flow lattice to be the powerset of program variables, we obtain a system which, in a very strong sense, subsumes all other systems in the family (in particular, for each program, it provides a principal typing from which all others may be inferred). This distinguished system is shown to be equivalent to, though more simply described than, Amtoft and Banerjee's Hoare-style independence logic (SAS'04).

In general, some lattices are more expressive than others. Despite this, we show that no type system in the family can give better results for a given choice of lattice than the type system for that lattice itself.

Finally, for any program typeable in one of these systems, we show how to construct an equivalent program which is typeable in a simple flow-insensitive system. We argue that this general approach could be useful in a proof-carrying-code setting.

Categories and Subject Descriptors D.3 [*PROGRAMMING LANGUAGES*]; F.3.1 [*LOGICS AND MEANINGS OF PROGRAMS*]: Specifying and Verifying and Reasoning about Programs; F.3.2 [*LOGICS AND MEANINGS OF PROGRAMS*]: Semantics of Programming Languages—Program analysis

General Terms Languages, Security, Theory

Keywords flow-sensitivity, information flow, non-interference, static analysis, type systems

1. Introduction

This article investigates formal properties of a family of flow-sensitive type systems for tracking information flow.

The analysis of information flow in programs has received considerable attention in recent years due to its connection to the problem of secure information flow [SM03]. The classic end-to-end confidentiality policy says that if certain data in a system is considered secret from the perspective of a certain observer of the system, then during computation there should be no information flow from that data to that observer. Denning and Denning [DD77] pioneered the use of program analysis to statically determine if the information flow properties of a program satisfy a certain confidentiality policy.

Most of the more recent work in this area (see [SM03] for an overview) has been based upon the use of *security type systems* to formulate the analysis of secure information flow, and to aid in a rigorous proof of its correctness.

We will focus, like many works in the area, on systems in which secrets are stored in variables. Security *levels* are associated with variables, and this describes the intended secrecy of the contents. The simplest instance of the problem involves two security levels: high (H) which denotes secrets, and low (L) which denotes public data. A partial ordering, $\mathrm{L} \sqsubseteq \mathrm{H}$, denotes that the only permitted information flow is from L to H. The security problem is to verify that there is no dependency between the initial value of the high variables (the secret to which the program has access), and the final value of the low variables (the outputs which are visible to the public).

With respect to the treatment of variables, one feature of almost all recent type based systems is that they are *flow-insensitive*. This means that the order of execution is not taken into account in the analysis. One simple intuition for the notion of flow-insensitivity [NRH99] is that an analysis is flow-insensitive if the results for analysing C_1 ; C_2 are the same as that for C_2 ; C_1. In this respect the analysis of [VSI96] (which can be viewed as a reformulation of Denning and Denning's original analysis) is flow-insensitive. In particular flow-insensitivity of this style of type system means that if a program is to be typed as "secure" then *every* subprogram must also be typed as "secure". So for example the trivial program $l := h$; $l := 0$ where h contains a secret, and the final value of l is low (publicly observable) is considered insecure because the subprogram $l := h$ is insecure.

More generally, flow-insensitivity uses a single abstraction (in this case a single security level) to represent each variable in the program. Flow-sensitivity, on the other hand, increases accuracy by providing a different abstraction at each program point.

Although there are a number of empirical/experimental analyses of the relationship between flow-sensitive and flow-insensitive program analyses (see e.g. [CH95]), there has been very little discussion of this dimension in connection to information flow analysis.

In this article we investigate flow-sensitive typings for a simple While language. We present a family of semantically sound security type systems (parameterised by the choice of flow lattice) which allow the type of a variable to "float", assigning different security types at different points in the program (Section 2).

Although this type system is extremely simple, it turns up some surprises. Our main results are the following:

POPL'06 January 11–13, 2006, Charleston, South Carolina, USA.
Copyright © 2006 ACM 1-59593-027-2/06/0001. . . $5.00.

- Although we can freely choose an arbitrarily complex flow lattice, there is a single "universal" lattice, and hence a single type system, from which all other typings in all other instances can be deduced. In fact, all possible typings in all possible lattices can be obtained from one principal typing in the universal lattice. From the principal typing, we can construct both the strongest (smallest) output typing for a given input typing, and the weakest (largest) input typing for a given output typing. The universal lattice is the powerset of program variables.
- For the universal lattice, we show that the type system is equivalent to Amtoft and Banerjee's Hoare-like logic for program dependence [AB04], which is expressed in terms of input-variable output-variable independence pairs. Because our formulation is based on dependence rather than independence, it is arguably simpler and admits a more straightforward correctness proof, without the need to resort to a non-standard trace semantics.
- In general, some lattices are more expressive than others. For example, in contrast to the two-point lattice $\mathrm{L} \sqsubseteq \mathrm{H}$, a single derivation in the type system for the universal lattice can identify fine-grained inter-variable dependencies of the form "x may depend on the initial value of y but not on z". Despite this variation in expressiveness, we establish in Section 6 an "internal completeness" result which shows that no type system in the family can give better results for a given choice of lattice than the type system for that lattice itself.
- Finally in Section 7 we show that for any program typeable in an instance of the flow-sensitive system, we are able to construct an equivalent program which is typeable in a simple flow-insensitive system. The translation is given by a security-type-directed translation, introducing extra variables. This general approach could be useful in a proof-carrying-code setting where the code consumer can have a simple type system, but the code producer is free to work in a more permissive system and use the translation to provide more easily checked code.

1.1 Related Work

A number of authors have presented flow-sensitive information flow analyses e.g. [CHH02]. Those close in style to a type system formulation include Banâtre *et al* [BBL94], who present a system very similar to that of [AB04], except that all indirect flows are handled in a pre-pass. Andrews and Reitman describe a similar logic [AR80] but did not consider semantic soundness.

In the treatment of information flow analysis of low level code (e.g., [GS05, HS05]), flow-sensitivity arises as an essential component to handle single threaded structures such as stacks and registers, since obviously stacks and registers cannot be assigned a fixed type throughout program execution.

The transformation we present in Section 7 is related to *single static assignment*(SSA)[CFR$^+$89], although the perspective is quite different. We discuss this further in Section 7.6

2. A Family of Type Systems

We work with a simple While language with the usual semantics. Program variables are drawn from a finite set Var. A flow-insensitive type system, such as that in [VSI96], has the following form: each variable is assigned a fixed security level. When assigning an expression to a variable $x := E$, all variables in E must have an equal or lower security level. When assignments take place in loops or conditional branches, to avoid indirect information flows the level of x must be at least as high as the level of any variable in the branching expression.

To allow types to be flow-sensitive, we must allow the type of a variable to "float". For example, taking the two-point flow lattice, when assigning an expression to a variable $x := y+x$, if x has type

$$\text{Skip}\ \frac{}{p \vdash \Gamma\ \{\mathbf{skip}\}\ \Gamma}$$

$$\text{Assign}\ \frac{\Gamma \vdash E : t}{p \vdash \Gamma\ \{x := E\}\ \Gamma[x \mapsto p \sqcup t]}$$

$$\text{Seq}\ \frac{p \vdash \Gamma\ \{C_1\}\ \Gamma' \quad p \vdash \Gamma'\ \{C_2\}\ \Gamma''}{p \vdash \Gamma\ \{C_1\ ;\ C_2\}\ \Gamma''}$$

$$\text{If}\ \frac{\Gamma \vdash E : t \quad p \sqcup t \vdash \Gamma\ \{C_i\}\ \Gamma' \quad i = 1,2}{p \vdash \Gamma\ \{\mathbf{if}\ E\ C_1\ C_2\}\ \Gamma'}$$

$$\text{While}\ \frac{\Gamma \vdash E : t \quad p \sqcup t \vdash \Gamma\ \{C\}\ \Gamma}{p \vdash \Gamma\ \{\mathbf{while}\ E\ C\}\ \Gamma}$$

$$\text{Sub}\ \frac{p_1 \vdash \Gamma_1\ \{C\}\ \Gamma_1'}{p_2 \vdash \Gamma_2\ \{C\}\ \Gamma_2'} \quad p_2 \sqsubseteq p_1, \Gamma_2 \sqsubseteq \Gamma_1, \Gamma_1' \sqsubseteq \Gamma_2'$$

Table 1. Flow-Sensitive Type Rules

L before the assignment and y has type H, then after the assignment x must be considered to have type H.

The flow-sensitive system we define is a family of inference systems, one for each choice of flow lattice $\mathcal{L}$ (where $\mathcal{L}$ may be any finite lattice). For a command C, judgements have the form

$$p \vdash_{\mathcal{L}} \Gamma\ \{C\}\ \Gamma'$$

where $p \in \mathcal{L}$, and Γ, Γ' are type environments of type Var $\to \mathcal{L}$. The inference rules are shown in Table 1. The idea is that if Γ describes the security levels of variables which hold before execution of C, then Γ' will describe the security levels of those variables after execution of C. The type p represents the usual "program counter" level and serves to eliminate indirect information flows; the derivation rules ensure that only variables which end up (in Γ') with types greater than or equal to p may be changed by C. We write $\vdash_{\mathcal{L}} \Gamma\ \{C\}\ \Gamma'$ to mean $\bot_{\mathcal{L}} \vdash_{\mathcal{L}} \Gamma\ \{C\}\ \Gamma'$. We drop the $\mathcal{L}$ subscript from judgements where the identity of the lattice is clear from the context or is not relevant to the discussion.

In some of the derivation rules we write $\Gamma \vdash E : t$ to mean that expression E has type t assuming type environment Γ. Throughout this paper the type of an expression is defined simply by taking the lub of the types of its free variables:

$$\Gamma \vdash E : t \text{ iff } t = \bigsqcup_{x \in \mathrm{fv}(E)} \Gamma(x).$$

This is consistent with the typings used in many systems, though more sophisticated typing rules for expressions would be possible in principle.

3. Semantic Soundness

The type systems satisfy a straightforward non-interference condition: only changes to inputs with types $\sqsubseteq t$ should be visible to outputs with type t. More precisely, given a derivation $\vdash \Gamma\ \{C\}\ \Gamma'$, the final value of a variable x with final type $t = \Gamma'(x)$, should depend at most on the initial values of those variables y with initial types $\Gamma(y) \sqsubseteq t$. Following [HS91, SS01, HR98] we formalise this using equivalence relations.

Definition 3.1. *Let R and S be equivalence relations on stores. We say that program C maps R into S, written $C : R \Rightarrow S$, iff, for all σ, ρ, if $\langle C, \sigma\rangle \Downarrow \sigma'$ and $\langle C, \rho\rangle \Downarrow \rho'$ then $\sigma\ R\ \rho \Rightarrow \sigma'\ S\ \rho'$.*

We note that this is a partial correctness condition: it allows C to terminate on σ but diverge on ρ, even when $\sigma \ R \ \rho$. This reflects the fact that the type systems take no account of the ways in which the values of variables may affect a program's termination behaviour. Given $\Gamma : \mathrm{Var} \to \mathcal{L}$ and $t \in \mathcal{L}$, we write $=_{\Gamma,t}$ for the equivalence relation on stores which relates stores which are equal on all variables having type $\sqsubseteq t$ in environment Γ, thus: $\sigma =_{\Gamma,t} \rho$ iff $\forall x.\Gamma(x) \sqsubseteq t \Rightarrow \sigma(x) = \rho(x)$.

The formal statement of correctness for a derivation $p \vdash \Gamma \ \{C\} \ \Gamma'$ has two parts, one asserting a simple safety property relating to p (as described in Section 2) and the other asserting the non-interference property.

Definition 3.2. *The semantic security relation $p \models_{\mathcal{L}} \Gamma \ \{C\} \ \Gamma'$ holds iff both the following conditions are satisfied:*

1. *For all σ, σ', x, if $\langle C, \sigma\rangle \Downarrow \sigma'$ and $\Gamma'(x) \not\sqsupseteq p$, then $\sigma'(x) = \sigma(x)$.*
2. *For all $t \in \mathcal{L}$, $C : (=_{\Gamma,t}) \Rightarrow (=_{\Gamma',t})$.*

As with $\vdash_{\mathcal{L}}$, we suppress the $\mathcal{L}$ subscript where possible. We write $\models_{\mathcal{L}} \Gamma \ \{C\} \ \Gamma'$ to mean $\perp_{\mathcal{L}} \models_{\mathcal{L}} \Gamma \ \{C\} \ \Gamma'$ (note that condition 1 is vacuous for $p = \perp$).

Theorem 3.3 (Semantic Soundness). $p \vdash_{\mathcal{L}} \Gamma \ \{C\} \ \Gamma' \Rightarrow p \models_{\mathcal{L}} \Gamma \ \{C\} \ \Gamma'$.

The proof for condition 2 of the semantic security relation depends on condition 1, but not vice versa. Proof of condition 1 is by an easy argument that $\Gamma'(x) \not\sqsupseteq p$ implies that C contains no assignments to x. Proof of condition 2 is by induction on the derivation.

The reverse implication, semantic completeness, does *not* hold, as shown by the following:

Example 3.4. *Consider the program $C \stackrel{\text{def}}{=}$ **if** $(h == 0)$ $(l := h)$ $(l := 0)$. This is semantically equivalent to $l := 0$ so it is clear that $\models \Gamma \ \{C\} \ \Gamma$ holds for arbitrary Γ. However, for $\Gamma(h) = \mathrm{H}, \Gamma(l) = \mathrm{L}$, with $\mathrm{L} \sqsubset \mathrm{H}$, $\not\vdash \Gamma \ \{C\} \ \Gamma$, because $\Gamma \vdash (h == 0) : \mathrm{H}$ and the assignments to l force $\Gamma'(l) \sqsupseteq \mathrm{H}$.*

4. The Algorithmic Type System

In this section we introduce a variant of the typing rules in which the weakening rule (Sub) is removed and folded into the If and While rules. The result is a system which calculates the smallest Γ' such that $p \vdash_{\mathcal{L}} \Gamma \ \{C\} \ \Gamma'$. The Skip, Assign and Seq rules are unchanged. The replacement If and While rules are shown in Table 2. The rules are deterministic: given an input type environment exactly one derivation is possible for any given C. (Well, almost. The While rule allows the chain $\Gamma'_0, \Gamma'_1, \cdots, \Gamma'_n$ to be extended arbitrarily by appending unnecessary repetitions of the limit. We may assume that n is chosen minimally.)

Theorem 4.1 (Algorithmic Correctness). *For all $\mathcal{L}$ and for all C:*

1. *For all p, Γ, there exists a unique Γ' such that $p \vdash^{\mathrm{A}}_{\mathcal{L}} \Gamma \ \{C\} \ \Gamma'$ and furthermore, the corresponding function $\mathcal{A}^C_{\mathcal{L}}(p, \Gamma) \mapsto \Gamma'$ is monotone.*
2. *If $p \vdash_{\mathcal{L}} \Gamma \ \{C\} \ \Gamma'$ then $\mathcal{A}^C_{\mathcal{L}}(p, \Gamma) \sqsubseteq \Gamma'$.*
3. *If $p \vdash^{\mathrm{A}}_{\mathcal{L}} \Gamma \ \{C\} \ \Gamma'$ then $p \vdash_{\mathcal{L}} \Gamma \ \{C\} \ \Gamma'$.*

Corollary 4.2. *$\mathcal{A}^C_{\mathcal{L}}(p, \Gamma)$ is the least Γ' such that $p \vdash_{\mathcal{L}} \Gamma \ \{C\} \ \Gamma'$.*

Proof of Algorithmic Correctness. Proof of part 1 of the theorem is by induction on the structure of the command. The interesting case is $\mathbf{while} \ E \ C$. By induction hypothesis, $\mathcal{A}^C_{\mathcal{L}}$ is well-defined and monotone. It follows that the sequences $\Gamma, \Gamma'_1, \Gamma'_2, \ldots$ and $\perp, \Gamma''_0, \Gamma''_1, \ldots$ may be constructed as $\Gamma, F(\Gamma), F^2(\Gamma), \ldots$ and

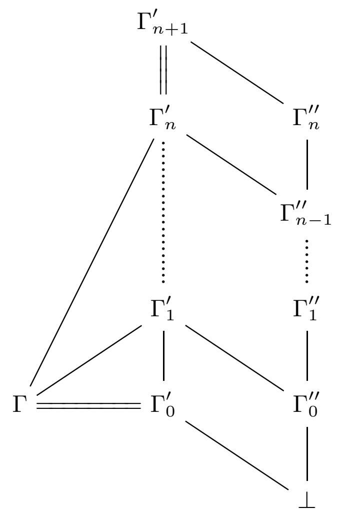


Figure 1. Construction of a Minimal While Typing

$\perp, G(\perp), G^2(\perp), \ldots$, with F and G being monotone functions derived from $\mathcal{A}^C_{\mathcal{L}}$; thus these sequences form the ascending chains shown in Figure 1. The chains have finite height because the lattices are finite, thus n is guaranteed to exist such that $\Gamma'_{n+1} = \Gamma'_n$ and it is then immediate that $\Gamma'_m = \Gamma'_n$ for all $m > n$. Put more succinctly, the While rule specifies Γ'_n as an iterative construction of the least fixed point of a monotone function on a finite lattice.

The proofs of parts 2 and 3 of the theorem are then by straightforward inductions on the $p \vdash_{\mathcal{L}} \Gamma \ \{C\} \ \Gamma'$ derivation and the structure of C, respectively. □

In Section 7 we adapt this version of the type system to define a program transformation which allows the use of conventional fixed-type systems in place of the flow-sensitive ones.

5. A Limiting Case: Dependency Analysis

Given the correctness condition, it is clear that the type systems defined above are calculating dependency relationships between program variables. Intuitively, we might expect to gain the most precise dependency information by choosing the flow lattice $\mathcal{P}(\mathrm{Var})$, which allows us to consider arbitrary sets of variables (including the singleton sets) as distinct types. In Section 6 we explore in detail this question of precision, with some slightly surprising results. Section 6 also formally establishes the special status of the type system for $\mathcal{P}(\mathrm{Var})$; anticipating this, we introduce some terminology:

Definition 5.1. *The* universal lattice *is the flow lattice $\mathcal{P}(\mathrm{Var})$ of sets of program variables. The* universal system *is the corresponding type system.*

In this section we show that the universal system is equivalent to (is, in fact, the De Morgan dual of) Amtoft and Banerjee's Hoare-style independence logic [AB04].

For notational clarity when comparing the universal system with other choices of $\mathcal{L}$, we let Δ, Δ' range over type environments just in the universal system (thus $\Delta, \Delta' : \mathrm{Var} \to \mathcal{P}(\mathrm{Var})$).

$$\text{If}\frac{\Gamma \vdash E : t \quad p \sqcup t \vdash^{\text{A}} \Gamma\ \{C_i\}\ \Gamma'_i \quad i = 1,2}{p \vdash^{\text{A}} \Gamma\ \{\mathbf{if}\ E\ C_1\ C_2\}\ \Gamma'} \quad \Gamma' = \Gamma'_1 \sqcup \Gamma'_2$$

$$\text{While}\frac{\Gamma'_i \vdash E : t_i \quad p \sqcup t_i \vdash^{\text{A}} \Gamma'_i\ \{C\}\ \Gamma''_i \quad 0 \le i \le n}{p \vdash^{\text{A}} \Gamma\ \{\mathbf{while}\ E\ C\}\ \Gamma'_n} \quad \Gamma'_0 = \Gamma,\ \Gamma'_{i+1} = \Gamma''_i \sqcup \Gamma,\ \Gamma'_{n+1} = \Gamma'_n$$

Table 2. Flow-Sensitive Type Rules: Algorithmic Version

5.1 Comparison with Amtoft-Banerjee Hoare Logic

In [AB04], Amtoft and Banerjee define a Hoare-style logic for deducing independence relationships between variables in While programs. Judgements in the logic have the form

$$G \vdash T\ \{C\}\ T'$$

where $G \in \mathcal{P}(\text{Var})$ and $T, T' \in \mathcal{P}(\text{Var} \times \text{Var})$. The idea is roughly as follows. Suppose that C is preceded by some previous computation on the store. We will refer to the value of a variable before this preceding computation as its *original* value. Then a pair $[x\#y]$ in T' represents an assertion that the value of x after C is independent of the original value of y, assuming that all the independence pairs in T are valid for the preceding computation. For ease of comparison, rather than sets of independence pairs T, we present the logic in terms of mappings $\nabla, \nabla' : \text{Var} \to \mathcal{P}(\text{Var})$ (this depends simply on the set isomorphism $A \times B \cong A \to \mathcal{P}(B)$). Thus Amtoft-Banerjee (AB) judgements in our presentation have the form

$$G \vdash \nabla\ \{C\}\ \nabla'$$

The AB derivation rules are shown in Table 3. The ordering $\preceq$ is pointwise reverse subset inclusion, thus:

$$\nabla_1 \preceq \nabla_2 \text{ iff } \forall x \in \text{Var}.\nabla_1(x) \supseteq \nabla_2(x)$$

Note that the ordering used on G is just $\subseteq$, not $\preceq$.

The relationship between the AB logic and the universal system is straightforward: for each Δ there is a corresponding ∇ such that $\nabla(x)$ is the complement of $\Delta(x)$. Where the universal system derives sets of dependencies, the AB logic simply derives the complementary set of independencies. (An AB context set G, on the other hand, corresponds directly to the *same* set p in a $\mathcal{P}(\text{Var})$-derivation.) We use the following notation:

$$\begin{array}{lcl} \overline{\Delta} & \overset{\text{def}}{=} & \nabla,\ \text{where } \nabla(x) = \text{Var} - \Delta(x) \\ \overline{\nabla} & \overset{\text{def}}{=} & \Delta,\ \text{where } \Delta(x) = \text{Var} - \nabla(x) \end{array}$$

Clearly this is an order isomorphism: $\overline{\overline{\Delta}} = \Delta$ and $\Delta_1 \sqsubseteq \Delta_2$ iff $\overline{\Delta_1} \preceq \overline{\Delta_2}$, etc.

Theorem 5.2. *The AB logic and the universal system are De Morgan duals. That is,* $G \vdash \Delta\ \{C\}\ \Delta'$ *is derivable in the universal system iff* $G \vdash \overline{\Delta}\ \{C\}\ \overline{\Delta'}$ *is derivable in the AB logic.*

The proof amounts, essentially, to showing that each AB rule is the dual of the universal system counterpart. This is not quite literally true, since the way some AB rules are formulated builds in the potential for implicit weakening, which must be made explicit using Sub in the corresponding $\mathcal{P}(\text{Var})$-derivation. For example, consider the second side condition on the rule IfAB. If we re-state this in its contrapositive form

$$(\exists x \in \text{fv}(E).w \notin \nabla(x)) \Rightarrow w \in G'$$

it is easily seen that the two side-conditions together amount to

$$G' \supseteq G \cup \bigcup_{x \in \text{fv}(E)} \overline{\nabla}(x) \qquad (1)$$

Note that any subderivation concluding at a premise to IfAB with G' strictly greater than required by (1), can have an instance of SubAB added at the end to make $G' = G \cup \bigcup_{x \in \text{fv}(E)} \overline{\nabla}(x)$. With this caveat, the side condition for IfAB is equivalent to the If premise in the universal system. Similar observations apply to the side conditions for AssignAB and WhileAB.

6. Internal Completeness

In this section we explore a fundamental relationship between different members of our family of flow-sensitive type systems. For simplicity of presentation, we consider only "top-level" typing judgements, ie, those of the form $\vdash \Gamma\ \{C\}\ \Gamma'$ (see Section 6.3 for further remarks on this point). We start by formalising a key notion: the sense in which one typing can be viewed as subsuming another (possibly in a different lattice). Given $\Gamma, \Gamma' : \text{Var} \to \mathcal{L}$, we refer to a pair $\Gamma\ \{\cdot\}\ \Gamma'$ as an $\mathcal{L}$-typing. If $\vdash \Gamma\ \{C\}\ \Gamma'$ we say that typing $\Gamma\ \{\cdot\}\ \Gamma'$ is derivable for C.

Definition 6.1. *An* $\mathcal{L}_1$*-typing* $\Gamma_1\ \{\cdot\}\ \Gamma'_1$ *is said to subsume an* $\mathcal{L}_2$*-typing* $\Gamma_2\ \{\cdot\}\ \Gamma'_2$ *iff, for all commands* C

$$\models_{\mathcal{L}_1} \Gamma_1\ \{C\}\ \Gamma'_1 \Rightarrow\ \models_{\mathcal{L}_2} \Gamma_2\ \{C\}\ \Gamma'_2$$

Note that this is a *semantic* notion of subsumption: one typing subsumes another precisely when the non-interference property specified by the former is stronger - satisfied by fewer programs - than that specified by the latter. As we shall see (Theorem 6.3), the type systems actually faithfully reflect this semantic relationship.

As defined, subsumption appears difficult to verify, since it quantifies over all possible programs. In fact, it suffices to compare the order relationships between the two pairs of type environments:

Theorem 6.2. $\mathcal{L}_1$*-typing* $\Gamma_1\ \{\cdot\}\ \Gamma'_1$ *subsumes* $\mathcal{L}_2$*-typing* $\Gamma_2\ \{\cdot\}\ \Gamma'_2$ *iff, for all* $x, y \in \text{Var}$*:*

$$\Gamma_1(x) \sqsubseteq \Gamma'_1(y) \Rightarrow \Gamma_2(x) \sqsubseteq \Gamma'_2(y)$$

Proof. For the *only if* direction we show the contrapositive. Assume $\Gamma_1(x) \sqsubseteq \Gamma'_1(y)$ and $\Gamma_2(x) \not\sqsubseteq \Gamma'_2(y)$. We must find some command C such that $\models_{\mathcal{L}_1} \Gamma_1\ \{C\}\ \Gamma'_1$ but $\not\models_{\mathcal{L}_2} \Gamma_2\ \{C\}\ \Gamma'_2$. Let $\{z_1, \ldots, z_n\} = \text{Var} - \{y\}$ and let C be the program

$$y := x; z_1 := 0; \cdots; z_n := 0$$

(the use of 0 here is arbitrary, any constant will do). It is then easy to verify that $C : (=_{\Gamma_1,t}) \Rightarrow (=_{\Gamma'_1,t})$ holds for all t but $C : (=_{\Gamma_2,s}) \Rightarrow (=_{\Gamma'_2,s})$ fails for $s = \Gamma'_2(y)$.

For the *if* direction, Assume

$$\begin{array}{ll} \text{(A1)} & \Gamma_1(x) \sqsubseteq \Gamma'_1(y) \Rightarrow \Gamma_2(x) \sqsubseteq \Gamma'_2(y) \\ \text{(A2)} & \models_{\mathcal{L}_1} \Gamma_1\ \{C\}\ \Gamma'_1 \end{array}$$

We have to show, for all $s \in \mathcal{L}_2$, $C : (=_{\Gamma_2,s}) \Rightarrow (=_{\Gamma'_2,s})$. Suppose $\sigma =_{\Gamma_2,s} \rho$ and $\langle C, \sigma\rangle \Downarrow \sigma'$ and $\langle C, \rho\rangle \Downarrow \rho'$ and $\Gamma'_2(y) \sqsubseteq s$. We must show $\sigma'(y) = \rho'(y)$. Now, for any x, $\Gamma_2(x) \sqsubseteq \Gamma'_2(y) \Rightarrow \Gamma_2(x) \sqsubseteq s \Rightarrow \sigma(x) = \rho(x)$. Hence, by (A1), $\Gamma_1(x) \sqsubseteq \Gamma'_1(y) \Rightarrow \sigma(x) = \rho(x)$, thus $\sigma =_{\Gamma_1,t} \rho$, where $t = \Gamma'_1(y)$. Hence, by (A2), $\sigma' =_{\Gamma'_1,t} \rho'$, hence $\sigma'(y) = \rho'(y)$ as required. $\square$

$$\text{AssignAB}\,\frac{}{G \vdash \nabla\ \{x := E\}\ \nabla'} \quad \begin{array}{l}\text{if } \forall y.\forall w \in \nabla'(y).\\ \quad x \neq y \Rightarrow w \in \nabla(y)\\ \quad x = y \Rightarrow w \notin G \wedge \forall z \in \text{fv}(E).w \in \nabla(y)\end{array}$$

$$\text{SeqAB}\,\frac{G \vdash \nabla\ \{C_1\}\ \nabla' \quad G \vdash \nabla'\ \{C_2\}\ \nabla''}{G \vdash \nabla\ \{C_1\ ;\ C_2\}\ \nabla''}$$

$$\text{IfAB}\,\frac{G' \vdash \nabla\ \{C_i\}\ \nabla' \quad i = 1,2}{G \vdash \nabla\ \{\textbf{if}\ E\ C_1\ C_2\}\ \nabla'} \quad \begin{array}{ll}\text{if} & G \subseteq G'\\ \text{and} & w \notin G' \Rightarrow \forall x \in \text{fv}(E).w \in \nabla(x)\end{array}$$

$$\text{WhileAB}\,\frac{G' \vdash \nabla\ \{C\}\ \nabla}{G \vdash \nabla\ \{\textbf{while}\ E\ C\}\ \nabla} \quad \begin{array}{ll}\text{if} & G \subseteq G'\\ \text{and} & w \notin G' \Rightarrow \forall x \in \text{fv}(E).w \in \nabla(x)\end{array}$$

$$\text{SubAB}\,\frac{G_1 \vdash \nabla_1\ \{C\}\ \nabla'_1}{G_2 \vdash \nabla_2\ \{C\}\ \nabla'_2} \quad G_2 \subseteq G_1, \nabla_2 \preceq \nabla_1, \nabla'_1 \preceq \nabla'_2$$

Table 3. Amtoft-Banerjee Hoare Logic

This result shows that the semantic content of a judgement $\vdash \Gamma\ \{C\}\ \Gamma'$ is uniquely determined by the set of pairs $\{(x,y)|\Gamma(x) \sqsubseteq \Gamma'(y)\}$: the smaller this set, the stronger the non-interference property. In fact, these pairs are precisely the dependencies allowed by the typing: if $\Gamma(x) \sqsubseteq \Gamma'(y)$ then the final value of y after executing C *may depend on* the initial value of x. Alternatively, we may consider the contrapositive form of Theorem 6.2, which says that $\Gamma_1\ \{\cdot\}\ \Gamma'_1$ subsumes $\Gamma_2\ \{\cdot\}\ \Gamma'_2$ iff

$$\Gamma_2(x) \not\sqsubseteq \Gamma'_2(y) \Rightarrow \Gamma_1(x) \not\sqsubseteq \Gamma'_1(y)$$

This allows us to understand a typing in terms of *independence* relations (as used by Amtoft and Banerjee). The larger the set $\{(x,y)|\Gamma(x) \not\sqsubseteq \Gamma'(y)\}$, the stronger the non-interference property: if $\Gamma(x) \not\sqsubseteq \Gamma'(y)$ then the final value of y after executing C *must be independent of* the initial value of x.

Now suppose we have an $\mathcal{L}_1$-typing which subsumes an $\mathcal{L}_2$-typing, and suppose we find that the $\mathcal{L}_2$-typing is *not* derivable for C in the $\mathcal{L}_2$-type system. Will it ever be possible to verify the soundness of the $\mathcal{L}_2$-typing for C indirectly, by deriving the subsuming $\mathcal{L}_1$-typing in the $\mathcal{L}_1$-system instead? We might expect this to happen in the case that $\mathcal{L}_1$ has more points, and is therefore able to make more refined dependency distinctions, than $\mathcal{L}_2$. Consider the examples shown in Figure 2, where $\mathcal{L}$ is the four point lattice depicted. It can readily be verified that the $\mathcal{P}(\text{Var})$-typing subsumes the $\mathcal{L}$-typing and both judgements are derivable. However, the $\mathcal{L}$ judgement simply assigns y the most conservative typing in $\mathcal{L}$, whereas the $\mathcal{P}(\text{Var})$ judgement captures the fact that the final value of y may depend on both x and z, but not on the initial value of y. Could it be, that as part of a derivation for some larger program, this fine-grained derivation for y enables us to derive a $\mathcal{P}(\text{Var})$-typing subsuming an $\mathcal{L}$-typing which cannot be derived in the simpler $\mathcal{L}$-system? Surprisingly, the answer is No, as confirmed by the following theorem.

Theorem 6.3 (Internal Completeness). *If $\mathcal{L}_1$-typing $\Gamma_1\ \{\cdot\}\ \Gamma'_1$ subsumes $\mathcal{L}_2$-typing $\Gamma_2\ \{\cdot\}\ \Gamma'_2$ and $\vdash_{\mathcal{L}_1} \Gamma_1\ \{C\}\ \Gamma'_1$, then $\vdash_{\mathcal{L}_2} \Gamma_2\ \{C\}\ \Gamma'_2$.*

Before we can prove the theorem, we need to develop some further machinery. As an additional benefit of this development, we find that, for each command C, there is a principal typing from which all others can be obtained.

6.1 Monotone Renaming of Types

This section establishes a key technical result used in the proof of the Internal Completeness theorem. Roughly speaking, the result says that we can take any derivation and, by consistently renaming the security types, obtain a new one. The notion of renaming is very general and allows us to translate a derivation for one choice of lattice into a derivation for a different lattice; we require only that the renaming function be monotone. Given $\Gamma : \text{Var} \to \mathcal{L}_1$ and a renaming function $f : \mathcal{L}_1 \to \mathcal{L}_2$, we write $f^*(\Gamma) : \text{Var} \to \mathcal{L}_2$ for the pointwise extension of f to Γ, thus $f^*(\Gamma)(x) \stackrel{\text{def}}{=} f(\Gamma(x))$.

Lemma 6.4 (Monotone Renaming). *Let $f : \mathcal{L}_1 \to \mathcal{L}_2$ be monotone. Then $p \vdash_{\mathcal{L}_1} \Gamma\ \{C\}\ \Gamma' \Rightarrow f(p) \vdash_{\mathcal{L}_2} f^*(\Gamma)\ \{C\}\ f^*(\Gamma')$.*

Proof. By induction on the height of the $\mathcal{L}_1$-derivation. We present the Assign and While cases by way of illustration.

Case: Assign. We have an $\mathcal{L}_1$- derivation of the form:

$$\frac{\Gamma \vdash_{\mathcal{L}_1} E : t}{p \vdash_{\mathcal{L}_1} \Gamma\ \{x := E\}\ \Gamma'}$$

where $\Gamma' = \Gamma[x \mapsto p \sqcup t]$. We can construct an $\mathcal{L}_2$- derivation:

$$\frac{f^*(\Gamma) \vdash_{\mathcal{L}_2} E : t'}{f(p) \vdash_{\mathcal{L}_2} f^*(\Gamma)\ \{x := E\}\ f^*(\Gamma)[x \mapsto f(p) \sqcup t']}$$

It suffices to show that $f^*(\Gamma)[x \mapsto f(p) \sqcup t'] \sqsubseteq f^*(\Gamma')$ (since we can then use Sub). By the definitions, $f^*(\Gamma)[x \mapsto f(p) \sqcup t'](y) = f^*(\Gamma')(y)$ for all $y \neq x$ and it remains to show $f(p) \sqcup t' \sqsubseteq f(p \sqcup t)$. Now by monotonicity of f we have

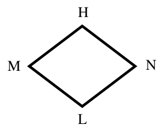
$$t' \stackrel{\text{def}}{=} \bigsqcup_{y \in \text{fv}(E)} f(\Gamma(y)) \sqsubseteq f\left(\bigsqcup_{y \in \text{fv}(E)} \Gamma(y)\right) = f(t) \qquad (2)$$

Finally, using this and monotonicity of f again, we have $f(p) \sqcup t' \sqsubseteq f(p) \sqcup f(t) \sqsubseteq f(p \sqcup t)$.

Case: While. We have an $\mathcal{L}_1$- derivation of the form:

$$\frac{\Gamma \vdash_{\mathcal{L}_1} E : t \quad p \sqcup t \vdash_{\mathcal{L}_1} \Gamma\ \{C\}\ \Gamma}{p \vdash_{\mathcal{L}_1} \Gamma\ \{\textbf{while}\ E\ C\}\ \Gamma}$$

By induction hypothesis we have $f(p \sqcup t) \vdash_{\mathcal{L}_2} f^*(\Gamma)\ \{C\}\ f^*(\Gamma)$. As in the Assign case, we have $f^*(\Gamma) \vdash_{\mathcal{L}_2} E : t'$ and $f(p) \sqcup t' \sqsubseteq$



$\vdash_{\mathcal{P}(\mathrm{Var})}$ $[x:\{x\}, y:\{y\}, z:\{z\}]$ **if** $x\ (y:=z)\ (y:=0)$ $[x:\{x\}, y:\{x,z\}, z:\{z\}]$

$\vdash_{\mathcal{L}}$ $[x:\mathrm{M}, y:\mathrm{L}, z:\mathrm{N}]$ **if** $x\ (y:=z)\ (y:=0)$ $[x:\mathrm{M}, y:\mathrm{H}, z:\mathrm{N}]$

Figure 2. Example Derivations

$f(p \sqcup t)$, allowing us to construct:

$$\dfrac{f^*(\Gamma) \vdash_{\mathcal{L}_2} E : t' \qquad \text{Sub}\ \dfrac{f(p \sqcup t) \vdash_{\mathcal{L}_2} f^*(\Gamma)\ \{C\}\ f^*(\Gamma)}{f(p) \sqcup t' \vdash_{\mathcal{L}_2} f^*(\Gamma)\ \{C\}\ f^*(\Gamma)}}{f(p) \vdash_{\mathcal{L}_2} f^*(\Gamma)\ \{\mathbf{while}\ E\ C\}\ f^*(\Gamma)}$$

□

6.2 Canonical Derivations

Given the Monotone Renaming lemma, we might hope to prove the Internal Completeness theorem by a construction for a suitable monotone renaming function to translate the $\mathcal{L}_1$-derivation into an $\mathcal{L}_2$-derivation for the subsumed typing. However, since an appropriate construction is not immediately obvious[1], we go via an indirect route. We begin our detour by showing how to produce any given derivation $\vdash_{\mathcal{L}} \Gamma\ \{C\}\ \Gamma'$ from a particular form of derivation in the universal system. To do this we construct, for each choice of Γ, an *abstract interpretation* [CC77] which is given by a *pair* of monotone renaming maps:

Definition 6.5. *Given* $\Gamma : \mathrm{Var} \to \mathcal{L}$*, we define the maps* $\alpha_\Gamma : \mathcal{P}(\mathrm{Var}) \to \mathcal{L}$ *and* $\gamma_\Gamma : \mathcal{L} \to \mathcal{P}(\mathrm{Var})$ *by:*

$$\alpha_\Gamma(X) \overset{\text{def}}{=} \bigsqcup_{x \in X} \Gamma(x) \tag{3}$$

$$\gamma_\Gamma(t) \overset{\text{def}}{=} \{x \mid \Gamma(x) \sqsubseteq t\} \tag{4}$$

These maps enjoy a special status. Recall [DP90] that a Galois Connection (GC) between $\mathcal{L}_1$ and $\mathcal{L}_2$ is a pair of maps $\langle\alpha, \gamma\rangle$ with $\alpha : \mathcal{L}_1 \to \mathcal{L}_2$, $\gamma : \mathcal{L}_2 \to \mathcal{L}_1$ and such that $\alpha(s) \sqsubseteq t \iff s \sqsubseteq \gamma(t)$. Key properties of a GC are that α, γ are both monotone, $\alpha \circ \gamma \sqsubseteq \mathrm{id}$, $\gamma \circ \alpha \sqsupseteq \mathrm{id}$, α preserves joins and γ preserves meets. Furthermore, the two component maps uniquely determine each other, thus:

$$\alpha(s) = \bigsqcap\{t \mid s \sqsubseteq \gamma(t)\} \tag{5}$$

$$\gamma(t) = \bigsqcup\{s \mid \alpha(s) \sqsubseteq t\} \tag{6}$$

Lemma 6.6. *For any* $\Gamma : \mathrm{Var} \to \mathcal{L}$*, the pair* $\langle\alpha_\Gamma, \gamma_\Gamma\rangle$ *is a Galois Connection between* $\mathcal{P}(\mathrm{Var})$ *and* $\mathcal{L}$*.*

Our first use of these renaming functions is, given an $\mathcal{L}$-typing $\Gamma\ \{\cdot\}\ \Gamma'$, to construct a typing in the universal system which subsumes it. A central rôle is played by the particular $\mathcal{P}(\mathrm{Var})$ type environment which maps each x to the singleton $\{x\}$. We denote this environment by Δ_0. Thus, for all $x \in \mathrm{Var}$, $\Delta_0(x) \overset{\text{def}}{=} \{x\}$.

Lemma 6.7. $\Delta_0\ \{\cdot\}\ \gamma^*_\Gamma(\Gamma')$ *subsumes* $\Gamma\ \{\cdot\}\ \Gamma'$*.*

Proof. Assume $\Delta_0(x) \subseteq \gamma_\Gamma(\Gamma'(y))$. We must show that $\Gamma(x) \sqsubseteq \Gamma'(y)$. Since $\Delta_0(x) = \{x\}$, the assumption is just $x \in \gamma_\Gamma(\Gamma'(y))$, hence $\Gamma(x) \sqsubseteq \Gamma'(y)$ by definition of γ_Γ. □

It turns out that the two related typings stand or fall together: for any C, the one is derivable if and only if the other is.

[1] Though we can read it off easily enough once we have the proof. It is: $f(s) = \bigsqcup\{\Gamma_2(x) | \Gamma_1(x) \sqsubseteq s\}$.

Lemma 6.8 (Canonical Derivations).
$\vdash_{\mathcal{L}} \Gamma\ \{C\}\ \Gamma' \iff\ \vdash \Delta_0\ \{C\}\ \gamma^*_\Gamma(\Gamma')$

Proof. The proof makes essential use of the Monotone Renaming lemma. For the $\Rightarrow$ direction, Monotone Renaming gives $\vdash \gamma^*_\Gamma(\Gamma)\ \{C\}\ \gamma^*_\Gamma(\Gamma')$. It then suffices to show that $\Delta_0 \subseteq \gamma^*_\Gamma(\Gamma)$, since appending a single use of Sub then gives the required derivation. To show $\Delta_0 \subseteq \gamma^*_\Gamma(\Gamma)$ we must show $x \in \gamma_\Gamma(\Gamma(x))$ for all x, and this is just $\Gamma(x) \sqsubseteq \Gamma(x)$.

For the $\Leftarrow$ direction, Monotone Renaming gives $\vdash \alpha^*_\Gamma(\Delta_0)\ \{C\}\ \alpha^*_\Gamma(\gamma^*_\Gamma(\Gamma'))$. Now, by (5), $\alpha_\Gamma(\{x\})) = \bigsqcap\{t \mid x \in \gamma_\Gamma(t)\} = \bigsqcap\{t \mid \Gamma(x) \sqsubseteq t\} = \Gamma(x)$, thus $\alpha^*_\Gamma(\Delta_0) = \Gamma$. By standard properties of a GC, $\alpha^*_\Gamma(\gamma^*_\Gamma(\Gamma')) \sqsubseteq \Gamma'$. Thus the required derivation follows by appending a single use of Sub. □

Now we can prove the theorem stated at the start of Section 6.

Proof of Internal Completeness. Assume $\mathcal{L}_1$-typing $\Gamma_1\ \{\cdot\}\ \Gamma'_1$ subsumes $\mathcal{L}_2$-typing $\Gamma_2\ \{\cdot\}\ \Gamma'_2$ and $\vdash_{\mathcal{L}_1} \Gamma_1\ \{C\}\ \Gamma'_1$. We must show $\vdash_{\mathcal{L}_2} \Gamma_2\ \{C\}\ \Gamma'_2$ which, by the Canonical Derivations lemma, is equivalent to

$$\vdash \Delta_0\ \{C\}\ \gamma^*_{\Gamma_2}(\Gamma'_2) \tag{7}$$

Furthermore, again by the Canonical Derivations lemma, the existence of our assumed derivation is equivalent to

$$\vdash \Delta_0\ \{C\}\ \gamma^*_{\Gamma_1}(\Gamma'_1) \tag{8}$$

It thus suffices to show

$$\gamma^*_{\Gamma_1}(\Gamma'_1) \sqsubseteq \gamma^*_{\Gamma_2}(\Gamma'_2) \tag{9}$$

and append a single use of Sub to derive (7) from (8). To show (9) we must show $\Gamma_1(y) \sqsubseteq \Gamma'_1(x) \Rightarrow \Gamma_2(y) \sqsubseteq \Gamma'_2(x)$, and this is just the assumed type subsumption, so we are done. □

As we noted above, the use of Galois Connections above is a form of abstract interpretation, and is reminiscent of the study of *complete* abstract interpretations [CC79, GRS00]. We have not explored these connections deeply, but a key difference would appear to be in our use of a different GC for each choice of Γ, rather than a single GC relating all $\mathcal{L}$-derivations to counterpart derivations in the universal system.

6.3 Principal Typings

As an additional corollary of the Canonical Derivations lemma, we find that, for each command C, there is a typing derivable for C from which all others can be inferred, namely

$$\vdash \Delta_0\ \{C\}\ \Delta_C$$

where Δ_C is the smallest Δ' such that $\vdash \Delta_0\ \{C\}\ \Delta'$ (recall that, by Corollary 4.2, this exists and is given by $\mathcal{A}^C_{\mathcal{P}(\mathrm{Var})}(\emptyset, \Delta_0)$). The Canonical Derivations lemma shows that derivability of any given $\vdash_{\mathcal{L}} \Gamma\ \{C\}\ \Gamma'$ is equivalent to $\forall x. \Delta_C(x) \subseteq \gamma^*_\Gamma(\Gamma'(x))$, which unpacks to:

$$y \in \Delta_C(x) \Rightarrow \Gamma(y) \sqsubseteq \Gamma'(x) \tag{10}$$

In fact, we can show that $\Delta_0 \ \{\cdot\} \ \Delta_C$ is a *principal* typing for C, in the sense defined by Wells [Wel02]. Transposed to our setting[2], Wells makes the following definitions:

- A pre-order on typings: $\Gamma_1 \ \{\cdot\} \ \Gamma_1' \le \Gamma_2 \ \{\cdot\} \ \Gamma_2'$ iff $\forall C. \vdash \Gamma_1 \ \{C\} \ \Gamma_1' \Rightarrow \ \vdash \Gamma_2 \ \{C\} \ \Gamma_2'$.
- Principal typings: typing $\Gamma_1 \ \{\cdot\} \ \Gamma_1'$ is *principal for* C iff $\vdash \Gamma_1 \ \{C\} \ \Gamma_1'$, and $\vdash \Gamma_2 \ \{C\} \ \Gamma_2' \Rightarrow \Gamma_1 \ \{\cdot\} \ \Gamma_1' \le \Gamma_2 \ \{\cdot\} \ \Gamma_2'$.

Theorem 6.9 (Principal Typing). *$\Delta_0 \ \{\cdot\} \ \Delta_C$ is principal for C.*

Before proving the theorem we state an easy lemma about subsumption:

Lemma 6.10. *If $\Gamma_1 \ \{\cdot\} \ \Gamma_1'$ subsumes $\Gamma_2 \ \{\cdot\} \ \Gamma_2'$ and $\Gamma' \sqsubseteq \Gamma_1'$, then $\Gamma_1 \ \{\cdot\} \ \Gamma'$ subsumes $\Gamma_2 \ \{\cdot\} \ \Gamma_2'$.*

Proof of Principal Typing. By definition of Δ_C, $\vdash \Delta_0 \ \{C\} \ \Delta_C$. Suppose $\vdash \Gamma \ \{C\} \ \Gamma'$. We must show, for all C', $\vdash \Delta_0 \ \{C'\} \ \Delta_C \Rightarrow \ \vdash \Gamma \ \{C'\} \ \Gamma'$. So suppose $\vdash \Delta_0 \ \{C'\} \ \Delta_C$. By Internal Completeness, it suffices to show that $\Delta_0 \ \{\cdot\} \ \Delta_C$ subsumes $\Gamma \ \{\cdot\} \ \Gamma'$. By Lemma 6.7, $\Delta_0 \ \{\cdot\} \ \gamma^*_\Gamma(\Gamma')$ subsumes $\Gamma \ \{\cdot\} \ \Gamma'$ so, by Lemma 6.10, it suffices to show $\Delta_C \sqsubseteq \gamma^*_\Gamma(\Gamma')$. By the Canonical Derivations lemma (using $\vdash \Gamma \ \{C\} \ \Gamma'$), $\vdash \Delta_0 \ \{C\} \ \gamma^*_\Gamma(\Gamma')$, so by definition of Δ_C, $\Delta_C \sqsubseteq \gamma^*_\Gamma(\Gamma')$. □

As noted earlier, we have restricted attention to typing judgements $p \vdash \Gamma \ \{C\} \ \Gamma'$ with $p = \perp$. While this is appropriate when we wish to consider whole programs, it does not allow us to apply our principal typings result compositionally. We believe the results above extend straightforwardly to the general case, the key step being to adjoin a "program counter variable" to Var, so the universal lattice becomes $\mathcal{P}(\mathrm{Var} + \{\mathrm{pc}\})$.

6.3.1 Polymorphism

The principal typing result above suggests that we should be able to view typings in the universal system as polymorphic, in some sense. In fact, this can be done quite directly: we may take an isomorphic view of $\mathcal{P}(\mathrm{Var})$ which shows typings in the universal system to be polymorphic in the standard sense of types involving type variables. Assume given a set of type variables $\mathrm{TVar} \cong \mathrm{Var}$, ranged over by β. Assume also some particular 1-1 mapping between the two sets: we write β_x for the type variable associated to program variable x. In this view, Δ_0 is a type environment which assigns a unique polymorphic variable β_x to each x. The application of α_Γ to Δ_0 in the proof ($\Leftarrow$) of the Canonical Derivations lemma amounts to an instantiation of the type variables to produce Γ. In general, α_Γ interprets a set T of type variables as the lub of the interpretations of its elements. Thus, in this view, types in the $\mathcal{P}(\mathrm{TVar})$ lattice can be thought of as formal lubs, which can be interpreted as elements in any lattice $\mathcal{L}$ by fixing an interpretation I for each β.

As above, let Δ_C be the smallest Δ' such that $\vdash \Delta_0 \ \{C\} \ \Delta'$. It can be shown that fixing Γ and calculating $\alpha^*_\Gamma(\Delta_C)$ gives us $\mathcal{A}^C_\mathcal{L}(\perp, \Gamma)$, ie the smallest Γ' such that $\vdash \Gamma \ \{C\} \ \Gamma'$. More interestingly, Δ_C may also be used in the reverse direction, to calculate the greatest Γ such that $\vdash \Gamma \ \{C\} \ \Gamma'$ for a given Γ'. The idea is to construct an interpretation $I : \mathrm{TVar} \to \mathcal{L}$ which "unifies" Δ_C and Γ', in the sense that

$$\alpha_I(\Delta_C(x)) \sqsubseteq \Gamma'(x) \tag{11}$$

for all x, where $\alpha_I(T) \overset{\text{def}}{=} \bigsqcup_{\beta \in T} I(\beta)$. The greatest I satisfying this equation for all x is given by

$$I(\beta) \overset{\text{def}}{=} \bigsqcap \{\Gamma'(x) \mid \beta \in \Delta_C(x)\} \tag{12}$$

[2] For this purpose, we view our family as a single type system consisting of the disjoint union of all its members.

The hope is that taking $\Gamma(x) \overset{\text{def}}{=} I(\beta_x)$ should then give us the greatest Γ such that $\vdash \Gamma \ \{C\} \ \Gamma'$. This is borne out by the following:

Proposition 6.11. *Given $\Gamma' : \mathrm{Var} \to \mathcal{L}$, let I be defined as in (12). Then $\Gamma(x) \overset{\text{def}}{=} I(\beta_x)$ is the greatest Γ such that $\vdash \Gamma \ \{C\} \ \Gamma'$.*

Proof. By the Canonical Derivations lemma, it suffices to show that the Γ defined is the greatest such that

$$\vdash \Delta_0 \ \{C\} \ \gamma^*_\Gamma(\Gamma') \tag{13}$$

Firstly, we show that (13) holds by showing that $\gamma_\Gamma(\Gamma'(x)) \supseteq \Delta_C(x)$ for all x. Suppose $\beta_y \in \Delta_C(x)$, then we must show that $\Gamma(y) = I(\beta_y) \sqsubseteq \Gamma'(x)$. This holds because $\beta_y \in \Delta_C(x)$ implies $\Gamma'(x)$ belongs to the set over which the meet is taken in (12).

It remains to show that $\gamma^*_{\Gamma''}(\Gamma') \supseteq \Delta_C \Rightarrow \Gamma'' \sqsubseteq \Gamma$. We show the contrapositive, so suppose $\Gamma'' \not\sqsubseteq \Gamma$. Thus, by (12), for some z, $\beta_x \in \Delta_C(z)$ and $\Gamma''(x) \not\sqsubseteq \Gamma'(z)$, thus $\beta_x \in \Delta_C(z)$ but $\beta_x \notin \gamma_{\Gamma''}(\Gamma'(z))$. □

7. Transformation to Fixed-Types

We have seen that floating types enable more programs to be typed than a standard fixed-type approach. In this section we show that if a program is typeable in the floating type system, then there is an equivalent program which is typeable in a traditional fixed-type system. We show this by construction: we extend the type system so that it also *generates* such a program. Take as an example the following valid judgement for the flow lattice $\mathrm{L} \sqsubseteq \mathrm{H}$, and the type environment $\Gamma = [\, l : \mathrm{L}, h : \mathrm{H}]$:

$$\mathrm{L} \vdash \Gamma \ \{l := h; l := 0; h := 0; l := h\} \ \Gamma$$

A traditional security type system would not be able to handle this example because the level of l becomes temporarily high, and then the level of h becomes low. To systematically transform the program to make it typeable by a fixed-type system, we represent each variable by a family of variables, one for each element of the flow lattice. The idea is that at any point in the computation we will be working with one particular member of the family. Whenever we need to raise the type of a variable from s to t in the original program we represent this in the transformed program by performing an assignment to *move* information from x_s to x_t, and by henceforth working with x_t.

Using this idea, the above program can be represented by the following:

$$l_\mathrm{H} := h_\mathrm{H}; l_\mathrm{L} := 0; h_\mathrm{L} := 0; l_\mathrm{L} := h_\mathrm{L}$$

where h_H and h_L, for example, are distinct variables. The initial inputs l and h are here represented by l_L and h_H respectively. In a flow-insensitive security type system the program is deemed secure because l_L (and h_L) only ever contain "low" information.

7.1 Fixed Variables

To discuss fixed types more precisely it is convenient to introduce a new class of such type-indexed variables into the language:

Definition 7.1. *For any given lattice of types $\mathcal{L}$, define the set of* fixed variables, FVar, *to be the set of type-indexed variables*

$$\mathrm{FVar} \overset{\text{def}}{=} \{x_t \mid x \in \mathrm{Var}, t \in \mathcal{L}.\}$$

To distinguish the fixed variables from the "ordinary" variables we will henceforth refer to the variables in Var *as* floating variables.

So, for example, if we are working in the two-level flow lattice, then for each floating variable x, we have in addition two fixed variables x_L and x_H.

We will now extend the language with fixed-type variables. Their dynamic semantics is just as for floating variables. We are going to present a transformation by adapting the algorithmic version of the type system, but first we must extend it to cover fixed-type variables: we extend the rule for expressions and add a rule for fixed-type assignment. We do not extend the type environments to cover fixed variables since their type is given by their index.

Let $\mathrm{fv}(E)$ denote the free floating variables (as before), and define $\mathrm{ffv}(E)$ to denote the free fixed variables of expression E (and similarly for commands). Then the typing of expressions in the extended language is given by

$$\Gamma \vdash E : t \text{ iff } t = \bigsqcup_{x \in \mathrm{fv}(E)} \Gamma(x) \ \sqcup \bigsqcup_{x_t \in \mathrm{ffv}(E)} t$$

The fixed type rule is simply:

$$\text{Fixed-Assign}\ \frac{\Gamma \vdash E : s \quad s \sqsubseteq t,\ p \sqsubseteq t}{p \vdash^{\mathrm{A}} \Gamma\ \{x_t := E\}\ \Gamma}$$

It is straightforward to extend the soundness arguments to encompass fixed variables.

Note that if we restrict our attention to programs with no free floating variables ($\mathrm{fv}(C) = \emptyset$), then type environments are redundant. We will use metavariable D to range over commands with no free floating variables. We will write $p \vdash D$ to denote $p \vdash^{\mathrm{A}} \Gamma\ \{D\}\ \Gamma$ for arbitrary Γ. It should be straightforward to see that derivations of this form correspond exactly to derivations in e.g. Volpano, Smith and Irvine's system [VSI96], and other Denning-style analyses, although we will not prove this formally.

7.2 Translation

Now we present the translation as an extension of the type system (algorithmic version) to judgements of the form

$$p \vdash_{\mathcal{L}} \Gamma\ \{C \rightsquigarrow D\}\ \Gamma'$$

(we do not decorate $\vdash$ for this system since the form of the judgements readily distinguish them from the previously defined systems). First we need some basic constructions and notations.

Definition 7.2.

1. *For any type environments Γ and Γ', let $\Gamma := \Gamma'$ denote the set*
$$\{x_s := x_t \mid \Gamma(x) = s, \Gamma'(x) = t, s \neq t\}$$
2. *Let S be a set of variable to variable assignment statements. We say that S is* independent *if for any distinct pair $w := x$ and $y := z$ in S, the variables w, x, y and z are all distinct. For independent S, all sequentialisations are semantically equivalent and we let S represent the command obtained by some canonical (but unspecified) sequentialisation.*

Lemma 7.3. $\Gamma := \Gamma'$ *is an independent set of assignments*

Thus we will write $\Gamma := \Gamma'$ to denote the command obtained by some canonical sequentialisation of the assignments.

Definition 7.4. *For any type environment Γ, let E^{Γ} denote the expression obtained by replacing each floating variable x in E with the fixed variable x_s where $s = \Gamma(x)$.*

With these definitions we are ready to introduce the translation. The rules are presented in Table 4.

The basic idea of the translation $p \vdash_{\mathcal{L}} \Gamma\ \{C \rightsquigarrow D\}\ \Gamma'$ is that for any program point in D corresponding to a point in C, for each variable x, only one member of the family $\{x_t\}_{t \in \mathcal{L}}$ will be "in play". The type variables in play at any given program point are given by the type environment at that program point. So for example if $\Gamma(x) = s$ then x_s will be the x-variable in play at the beginning of the execution of D.

Example 7.5. *Since a type derivation is structural in the syntax, for any derivation we can associate a type environment with each program point. Consider the example derivation shown in Figure 3: in the central column we write the environment update (rather than the whole environment) yielding the environment after that program point in the corresponding sub-derivation, and on the right-hand side we write the translated program. The example uses the four point lattice introduced previously (Figure 2).*

It remains to establish two properties of the translated terms:

- *Correctness:* they should be semantically equivalent to the original terms, and
- *Static Soundness:* they should still be typeable.

7.3 Correctness

Correctness means that the input-output behaviour of the program and its translation should be the same. We refer to this as *semantic equivalence*. Since the original programs operate on floating variables, and the translation operates on fixed variables, we must construct a suitable relation between them.

Definition 7.6. *Let σ range over floating variable stores and let ρ range over fixed variable stores. Then for each type environment Γ we define the compatibility relation as*

$$\sigma \sim_{\Gamma} \rho \iff \forall x \in \mathrm{Var}.\sigma(x) = \rho(x_{\Gamma(x)})$$

Theorem 7.7 (Translation Correctness).
If $p \vdash \Gamma\ \{C \rightsquigarrow D\}\ \Gamma'$ then for all σ and ρ such that $\sigma \sim_{\Gamma} \rho$,

- $\langle C, \sigma\rangle \Downarrow \sigma' \Rightarrow \exists \rho'.\langle D, \rho\rangle \Downarrow \rho'$ *and* $\sigma' \sim_{\Gamma'} \rho'$
- $\langle D, \rho\rangle \Downarrow \rho' \Rightarrow \exists \sigma'.\langle C, \sigma\rangle \Downarrow \sigma'$ *and* $\sigma' \sim_{\Gamma'} \rho'$

Proof. See Appendix A.1. □

7.4 Static Soundness

The fact that the translated programs are equivalent to the originals ensures that they have the same security properties, since noninterference is an extensional property. Here we show, more significantly, that the translated program is also typeable – and since it only contains fixed variables this means that it is typeable in a conventional fixed type system.

Lemma 7.8 (Expression Soundness). *If $\Gamma \vdash E : t$ then $\vdash E^{\Gamma} : t$*

Follows directly from the definitions.

Theorem 7.9 (Static Soundness). *If $p \vdash \Gamma\ \{C \rightsquigarrow D\}\ \Gamma'$ then $p \vdash D$*

Proof. See Appendix A.2. □

7.5 Complexity

The transformation increases program size by adding assignments of the form $\Gamma' := \Gamma$. These assignments arise whenever, in the flow-sensitive system, a variable changes its level. Since the only way that a variable can change its level is through an assignment, the size of $\Gamma' := \Gamma$ is bounded by the number (a) of assignment statements in the original program. The number of such assignments that are added to the program is proportional to the number (b) of conditional and while statements. This gives us a bound of $\mathcal{O}(ab)$, i.e., quadratic in the program size. This upper bound is tight, as shown by the following program, where we use the two-point lattice, and initially h is the only variable assigned type H:

```
if $y_1$ then
    if $y_2$ then
        ...
          if $y_n$ then
            if $h$ then $x_1 := 0; \cdots ; x_n := 0$
```

$$\text{Skip-t}\ \frac{}{p \vdash \Gamma\ \{\mathbf{skip} \rightsquigarrow \mathbf{skip}\}\ \Gamma}$$

$$\text{Assign-t}\ \frac{\Gamma \vdash E : t \quad s = p \sqcup t}{p \vdash \Gamma\ \{x := E \rightsquigarrow x_s := E^{\Gamma}\}\ \Gamma[x \mapsto s]}$$

$$\text{Seq-t}\ \frac{p \vdash \Gamma\ \{C_1 \rightsquigarrow D_1\}\ \Gamma' \quad p \vdash \Gamma'\ \{C_2 \rightsquigarrow D_2\}\ \Gamma''}{p \vdash \Gamma\ \{C_1\ ;\ C_2 \rightsquigarrow D_1\ ;\ D_2\}\ \Gamma''}$$

$$\text{If-t}\ \frac{\Gamma \vdash E : t \quad p \sqcup t \vdash \Gamma\ \{C_i \rightsquigarrow D_i\}\ \Gamma'_i \quad i = 1,2}{p \vdash \Gamma\ \{\mathbf{if}\ E\ C_1\ C_2 \rightsquigarrow \mathbf{if}\ E^{\Gamma}\ (D_1\ ;\ \Gamma' := \Gamma_1)\ (D_2\ ;\ \Gamma' := \Gamma_2)\}\ \Gamma'} \quad \Gamma' = \Gamma'_1 \sqcup \Gamma'_2$$

$$\text{While-t}\ \frac{\Gamma'_i \vdash E : t_i \quad p \sqcup t_i \vdash \Gamma'_i\ \{C \rightsquigarrow D_i\}\ \Gamma''_i \quad 0 \le i \le n}{p \vdash \Gamma\ \{\mathbf{while}\ E\ C \rightsquigarrow \Gamma'_n := \Gamma\ ;\ \mathbf{while}\ E^{\Gamma'_n}\ (D_n\ ;\ \Gamma'_n := \Gamma''_n)\}\ \Gamma'_n} \quad \Gamma'_0 = \Gamma,\ \Gamma'_{i+1} = \Gamma''_i \sqcup \Gamma,\ \Gamma'_{n+1} = \Gamma'_n$$

Table 4. Translation to fixed types

Initial typing: $\{[w : \mathrm{L}; x : \mathrm{M}; y : \mathrm{N}; z : \mathrm{H}]\}$

Code	Environment change	Translated code
$\mathbf{if}\ x = 0\ \mathbf{then}\ y := y + 1; w := z$	$[y \mapsto \mathrm{H}; w \mapsto \mathrm{H}]$	$\mathbf{if}\ x_{\mathrm{M}} = 0\ \mathbf{then}\ y_{\mathrm{H}} := y_{\mathrm{M}} + 1; w_{\mathrm{H}} := z_{\mathrm{H}}$ $\mathbf{else}\ y_{\mathrm{H}} := y_{\mathrm{L}}; w_{\mathrm{H}} := w_{\mathrm{L}}$
$\mathbf{while}\ x > 0$		$\mathbf{while}\ x_{\mathrm{M}} > 0$
$z := z + w$		$z_{\mathrm{H}} := z_{\mathrm{H}} + w_{\mathrm{H}}$
$x := x - 1$		$x_{\mathrm{M}} := x_{\mathrm{M}} - 1$
$z := x$	$[z \mapsto \mathrm{M}]$	$z_{\mathrm{M}} := x_{\mathrm{M}}$ $z_{\mathrm{H}} := z_{\mathrm{M}}$

Figure 3. Example translation derivation

where the one-armed conditional is just shorthand for a conditional with **skip** in the else branch. The above program is typeable, where in the final environment, $x_1 \ldots x_n$ have type H. Writing $X_{\mathrm{H}} := X_{\mathrm{L}}$ for the sequence of assignments $x_{1\mathrm{H}} := x_{1\mathrm{L}}; \cdots ; x_{n\mathrm{H}} := x_{n\mathrm{L}}$, the transformed program is:

```
if $y_{1\mathrm{L}}$ then
    if $y_{2\mathrm{L}}$ then
        ...
        if $y_{n\mathrm{L}}$ then
            if $h_{\mathrm{H}}$ then $x_{1\mathrm{H}} := 0; \cdots ; x_{n\mathrm{H}} := 0$
            else $X_{\mathrm{H}} := X_L$
        else $X_{\mathrm{H}} := X_L$
        ...
else $X_{\mathrm{H}} := X_L$
```

It seems likely that there is a more precise bound based on the depth of nesting of loops and conditions, and that such blow ups are unlikely in practice.

7.6 Relation to Single Static Assignment

Our transformation introduces additional variables, and this addition is performed in such a way that a flow-insensitive analysis on the transformed program achieves the same effect as a flow-sensitive analysis on the original. Viewed in this way, our transformation has a similar effect to transformation to *single static assignment form* (SSA) (see e.g. [App98]). Single static assignment is used in the compilation chain to improve and simplify dataflow analyses. It works by the introduction of additional variables in such a way that every variable is assigned-to exactly once. Since there is only one assignment per variable, it follows by construction that there is no need for a flow-sensitive analysis on a program in SSA form, since there is only one program point that can influence the type of a variable.

Our transformation is however rather different from SSA. The transformation we have described uses a flow-sensitive analysis in order to construct the transformed program, whereas SSA's purpose is to avoid the need to perform more complex analyses in the first place. Thus our transformation approach is perhaps not interesting when viewed from a classic compiler-construction perspective.

However, applications such as security are not directly relevant to optimisation and compilation. In a mobile-code setting, a code consumer may demand that the code can be verified to satisfy some information-flow policy. Furthermore, in order to have a small trusted computing base, a small and simple type system is preferable. Transformations of the kind presented here are interesting in this setting because they allow the code producer the benefits of constructing well-typed code in a more expressive system, without requiring the code consumer to verify code with respect to this more complex system[3].

8. Conclusions

We have presented and investigated the formal properties of a family of semantically sound flow-sensitive type systems for tracking

[3] The result of the SSA transformation is not an executable program, since it contains the so-called ϕ-nodes at all join-points, so SSA would be unsuitable for this purpose. However, [ADvRF01] proposes a mobile code representation based on SSA.

information flow in simple While programs. The family is indexed by the choice of flow lattice.

The key results we have shown are that:

- For a given program, all derivations in all members of the family can be inferred from the derivation of a principal typing in the universal system (ie, the type system for the flow lattice $\mathcal{P}(\mathrm{Var})$).
- The universal system is equivalent to Amtoft and Banerjee's Hoare-style independence logic.
- Each member of the family is "complete" with respect to the whole family, in that no member can be used to validate more $\mathcal{L}$-typings than the $\mathcal{L}$-system itself.
- Given a flow-sensitive type derivation for a program, we can systematically transform it to produce a semantically equivalent program which is typeable in a simple flow-insensitive system.

Possible avenues for future work include extending the flow-sensitive systems and program transformation to richer programming languages and deriving more precise complexity results for the program transformation.

Acknowledgments

Many thanks to David Clark for participation in discussions on this work, and for suggesting the worst-case complexity example. Discussions with Sruthi Bandhakavi and Daniel Hedin on alternative formulations of Amtoft and Banerjee's system were particularly useful. Thanks to Josef Sveningsson for drawing our attention to the connection to SSA, and to Niklas Broberg for system support. We also benefited from comments and suggestions from Dennis Walter and the anonymous referees.

References

[AB04] Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In *SAS 2004 (11th Static Analysis Symposium), Verona, Italy, August 2004*, volume 3148 of *LNCS*, pages 100–115. Springer-Verlag, 2004.

[ADvRF01] Wolfram Amme, Niall Dalton, Jeffery von Ronne, and Michael Franz. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 137–147, 2001.

[App98] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, Cambridge, 1998.

[AR80] G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM TOPLAS*, 2(1):56–75, January 1980.

[BBL94] J.-P. Banâtre, C. Bryce, and D. Le Métayer. Compile-time detection of information flow in sequential programs. In *Proc. European Symp. on Research in Computer Security*, volume 875 of *LNCS*, pages 55–73. Springer-Verlag, 1994.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 238–252, January 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[CFR$^+$89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark K. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, 1989.

[CH95] Paul R. Carini and Michael Hind. Flow-sensitive interprocedural constant propagation. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 23–31. ACM Press, 1995.

[CHH02] D. Clark, C. Hankin, and S. Hunt. Information flow for Algol-like languages. *Journal of Computer Languages*, 28(1):3–28, April 2002.

[DD77] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[DP90] B. Davey and H. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[GRS00] Roberto Giacobazzi, Francesco Ranzato, and Francesca Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2):361–416, 2000.

[GS05] S. Genaim and F. Spoto. Information Flow Analysis for Java Bytecode. In R. Cousot, editor, *Proc. of the Sixth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 346–362, Paris, France, January 2005. Springer-Verlag.

[HR98] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 365–377, January 1998.

[HS91] S. Hunt and D. Sands. Binding Time Analysis: A New PERspective. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'91)*, pages 154–164, September 1991. ACM SIGPLAN Notices 26(9).

[HS05] D. Hedin and D. Sands. Timing aware information flow security for a JavaCard-like bytecode. In *First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE '05)*, 2005. To Appear, ENTCS.

[NRH99] F. Nielson, H. Riis Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[SM03] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

[SS01] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher Order and Symbolic Computation*, 14(1):59–91, March 2001. Earlier version in ESOP'99.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[Wel02] J. B. Wells. The essence of principal typings. In *Proc. International Colloquium on Automata, Languages and Programming*, volume 2380 of *LNCS*, pages 913–925. Springer-Verlag, 2002.

A. Proofs from Section 7

A.1 Translation Correctness

In proving the theorem we will make use of the following simple lemmas:

Lemma A.1. *If $\sigma \sim_\Gamma \rho$ then*

1. $\langle \Gamma' := \Gamma, \rho\rangle \Downarrow \rho'$ *where* $\sigma \sim_\Gamma \rho'$
2. $[\![E]\!]\sigma = [\![E^\Gamma]\!]\rho$

Proof. 1. From definition 7.2, the effect of $\Gamma' := \Gamma$ on state ρ can be written as

$$\rho' = \rho[x_{\Gamma'(x)} \mapsto \rho(x_{\Gamma(x)}) \mid x \in \text{Var}]$$

(note that we have ignored the condition $\Gamma(x) \neq \Gamma'(x)$ in the definition of $\Gamma' := \Gamma$ since these are just identity updates). So we have for all $x \in \text{Var}$

$$\begin{aligned}\sigma(x) &= \rho(x_{\Gamma(x)})\\ &= \rho[x_{\Gamma'(x)} \mapsto \rho(x_{\Gamma(x)}) \mid x \in \text{Var}](x_{\Gamma'(x)})\\ &= \rho'(x_{\Gamma'(x)})\end{aligned}$$

and hence $\sigma \sim_{\Gamma'} \rho'$ as required.

2. Straightforward from the definitions.

□

Proof of Translation Correctness. We argue by induction on the derivation in a standard big-step semantics. For collections of independent assignments of the form $\Gamma' := \Gamma$ we somewhat improperly treat them as if they are evaluated in a single atomic step. We illustrate the first part of the theorem, although most steps are in fact reversible, so the proof in the other direction is essentially the same. We focus on the more interesting cases.

Suppose that $p \vdash \Gamma\ \{C \rightsquigarrow D\}\ \Gamma'$, $\sigma \sim_\Gamma \rho$ and that $\langle C, \sigma\rangle \Downarrow \sigma'$. We prove that $\langle D, \rho\rangle \Downarrow \rho'$ where $\sigma' \sim_{\Gamma'} \rho'$ by induction on the derivation of $\langle C, \sigma\rangle \Downarrow \sigma'$ and cases according to the last rule applied:

Case: Assign. We have a derivation of the form

$$\frac{\Gamma \vdash E : t \quad s = p \sqcup t}{p \vdash \Gamma\ \{x := E \rightsquigarrow x_s := E^\Gamma\}\ \Gamma[x \mapsto s]}$$

Suppose that $[\![E]\!]\sigma = V$, and hence that

$$\langle x := E, \sigma\rangle \Downarrow \sigma[x \mapsto V].$$

By Lemma A.1, $[\![E^\Gamma]\!]\sigma = V$ and hence

$$\langle x_\sigma := E^\Gamma, \sigma\rangle \Downarrow \sigma[x \mapsto V].$$

Case: While. There are two cases according to the value of the conditional. We just show the harder case where the last step of the derivation has the form:

$$\frac{[\![E]\!]\sigma = \mathbf{true} \quad \langle C, \sigma\rangle \Downarrow \sigma' \quad \langle \mathbf{while}\ E\ C, \sigma'\rangle \Downarrow \sigma''}{\langle \mathbf{while}\ E\ C, \sigma\rangle \Downarrow \sigma''}$$

We have a translation derivation of the form

$$\frac{\Gamma'_i \vdash E : t_i \quad p \sqcup t_i \vdash \Gamma'_i\ \{C \rightsquigarrow D_i\}\ \Gamma''_i \quad 0 \leq i \leq n}{p \vdash \Gamma\ \{\mathbf{while}\ E\ C \rightsquigarrow \Gamma' := \Gamma\ ;\ \mathbf{while}\ E^{\Gamma'}\ (D\ ;\ \Gamma' := \Gamma'')\}\ \Gamma'}$$

where $\Gamma'_0 = \Gamma$, $\Gamma'_{i+1} = \Gamma''_i \sqcup \Gamma$, $\Gamma'_{n+1} = \Gamma'_n$ and $\Gamma' = \Gamma'_n, \Gamma'' = \Gamma''_n, D = D_n$. Henceforth let W_D denote the subterm $\mathbf{while}\ E^{\Gamma'}\ (D\ ;\ \Gamma' := \Gamma'')$. Assume that $\sigma \sim_\Gamma \rho$. We are required to show that

$$\langle \Gamma' := \Gamma\ ;\ W_D, \rho\rangle \Downarrow \rho'' \text{ where } \sigma \sim_{\Gamma'} \rho''$$

We assemble the following facts in order to construct a derivation for this claim. From Lemma A.1(1) we get

$$\langle \Gamma' := \Gamma, \rho\rangle \Downarrow \rho_1 \text{ where } \sigma \sim_{\Gamma'} \rho_1 \tag{14}$$

and from Lemma A.1(2)

$$\langle E^{\Gamma'}, \rho_1\rangle \Downarrow \mathbf{true}. \tag{15}$$

From the induction hypothesis for the subderivation for C we have

$$\langle D, \rho_1\rangle \Downarrow \rho_2 \text{ where } \sigma' \sim_{\Gamma''} \rho_2, \tag{16}$$

and from Lemma A.1(1),

$$\langle \Gamma' := \Gamma'', \rho_2\rangle \Downarrow \rho_3 \text{ where } \sigma' \sim_{\Gamma'} \rho_3 \tag{17}$$

Finally from the premises of the typing judgement, we also have the weaker judgement:

$$p \vdash \Gamma'\ \{\mathbf{while}\ E\ C \rightsquigarrow \Gamma' := \Gamma\ ;\ W_D\}\ \Gamma'$$

Now we apply the induction hypothesis for the second evaluation premise, with respect to this judgement, to obtain

$$\langle W_D, \rho_3\rangle \Downarrow \rho'' \text{ where } \sigma'' \sim_{\Gamma'} \rho'' \tag{18}$$

Finally from these facts we construct the required derivation, which is given in Figure 4. □

A.2 Static Soundness

The proof is by induction on the structure of the translation derivation, making use of the following simple weakening lemmas:

Lemma A.2.

- *If* $p \vdash D$ *and* $p' \sqsubseteq p$ *then* $p' \vdash D$
- *If* $p \vdash^{\mathrm{A}} \Gamma\ \{C\}\ \Gamma'$ *then for all* x, $\Gamma(x) \neq \Gamma'(x) \Rightarrow p \sqsubseteq \Gamma'(x)$

Proof. The first item is a straightforward induction on the derivation, and we omit the details.

The second item is also by induction on the derivation. We present the three key cases.

Case: Assign. The conclusion of the rule is:

$$p \vdash^{\mathrm{A}} \Gamma\ \{x := E\}\ \Gamma[x \mapsto p \sqcup t]$$

The initial and final type environment only differ (potentially) in x and we see immediately that $p \sqsubseteq \Gamma'(x) = p \sqcup t$.

Case: If. The rule provides a derivation of the form

$$\frac{p \sqcup t \vdash^{\mathrm{A}} \Gamma\ \{C_i\}\ \Gamma'_i \quad i = 1, 2 \quad \Gamma' = \Gamma'_1 \sqcup \Gamma'_2}{p \vdash^{\mathrm{A}} \Gamma\ \{\mathbf{if}\ E\ C_1\ C_2\}\ \Gamma'}$$

The induction hypothesis gives, for $i = 1, 2$,

$$\forall x. \Gamma(x) \neq \Gamma'_i(x) \Rightarrow p \sqsubseteq \Gamma'_i(x)$$

So suppose that for some particular x we have $\Gamma(x) \neq \Gamma'(x)$. Since $\Gamma' = \Gamma'_1 \sqcup \Gamma'_2$ we must have $\Gamma(x) \neq \Gamma'_i(x)$ for either $i = 1$ or $i = 2$ (or both). It follows from the induction hypothesis that $p \sqsubseteq \Gamma'_i(x)$ for this i, and hence that $p \sqsubseteq \Gamma'_1(x) \sqcup \Gamma'_2(x)$ as required.

Case: While. The rule provides a derivation of the form

$$\frac{\Gamma'_i \vdash E : t_i \quad p \sqcup t_i \vdash^{\mathrm{A}} \Gamma'_i\ \{C\}\ \Gamma''_i \quad 0 \leq i \leq n}{p \vdash^{\mathrm{A}} \Gamma\ \{\mathbf{while}\ E\ C\}\ \Gamma'_n}$$

where $\Gamma'_0 = \Gamma$, $\Gamma'_{i+1} = \Gamma''_i \sqcup \Gamma$, $\Gamma'_{n+1} = \Gamma'_n$. The induction hypothesis gives, for $0 \leq i \leq n$, $\Gamma''_i(x) \neq \Gamma'_i(x) \Rightarrow p \sqcup t_i \sqsubseteq \Gamma''_i(x)$.

Assume $\Gamma'_n(x) \neq \Gamma(x)$. Now suppose that $\Gamma''_i(x) = \Gamma'_i(x)$ for all i with $0 \leq i \leq n$: we show that this contradicts the assumption (ie we show that it entails $\Gamma'_n(x) = \Gamma(x)$) by induction on n. The base case is immediate since $\Gamma'_0 = \Gamma$, so consider $n = k + 1$.

$$\dfrac{\dfrac{(14)}{\langle \Gamma' := \Gamma, \rho\rangle \Downarrow \rho_1} \quad \dfrac{\dfrac{(15)}{\langle E^{\Gamma'},\rho_1\rangle \Downarrow \mathbf{true}} \quad \dfrac{\dfrac{(16)}{\langle D,\rho_1\rangle\Downarrow\rho_2}\quad \dfrac{(17)}{\langle \Gamma' := \Gamma'',\rho_2\rangle \Downarrow \rho_3}}{\langle D\ ;\ \Gamma' := \Gamma'', \rho_1\rangle \Downarrow \rho_3} \quad \dfrac{(18)}{\langle W_D,\rho_3\rangle \Downarrow \rho''}}{\langle W_D,\rho_1\rangle \Downarrow \rho''}}{\langle \Gamma' := \Gamma\ ;\ \mathbf{while}\ E^{\Gamma'}\ (D\ ;\ \Gamma' := \Gamma''), \rho\rangle \Downarrow \rho''}$$

Figure 4. Concluding derivation, While case, Theorem 7.7

By construction, $\Gamma'_{k+1}(x) = \Gamma''_k(x) \sqcup \Gamma(x)$ so, by supposition, $\Gamma'_{k+1}(x) = \Gamma'_k(x) \sqcup \Gamma(x)$. But by IH $\Gamma'_k(x) = \Gamma(x)$, hence $\Gamma'_{k+1}(x) = \Gamma(x)$.

We have shown that $\Gamma'_n(x) \neq \Gamma(x)$ implies the existence of some i such that $\Gamma''_i(x) \neq \Gamma'_i(x)$ so, by the induction hypothesis, $p \sqcup t_i \sqsubseteq \Gamma''_i(x)$, hence $p \sqsubseteq \Gamma''_i(x)$. But, as illustrated in Figure 1, $\Gamma''_i \sqsubseteq \Gamma'_n$ holds for all i, so we are done. □

Proof of Static Soundness. We give a couple of illustrative cases.

Case: Assign-t. The derivation is of the form

$$\dfrac{\Gamma \vdash E : t \quad s = p \sqcup t}{p \vdash \Gamma\ \{x := E \rightsquigarrow x_s := E^{\Gamma}\}\ \Gamma[x \mapsto s]}$$

From Lemma 7.8 we have that $\vdash E^{\Gamma} : t$, and thus $p \vdash x_s := E^{\Gamma}$ follows directly from the Assign-fixed axiom.

Case: While-t. Assume the last rule in the inference has the form:

$$\dfrac{\Gamma'_i \vdash E : t_i \quad p \sqcup t_i \vdash \Gamma'_i\ \{C \rightsquigarrow D_i\}\ \Gamma''_i \quad 0 \leq i \leq n}{p \vdash \Gamma\ \{\mathbf{while}\ E\ C \rightsquigarrow \Gamma' := \Gamma\ ;\ \mathbf{while}\ E^{\Gamma'}\ (D\ ;\ \Gamma' := \Gamma'')\}\ \Gamma'}$$

where $\Gamma'_0 = \Gamma,\ \Gamma'_{i+1} = \Gamma''_i \sqcup \Gamma,\ \Gamma'_{n+1} = \Gamma'_n$ and $\Gamma' = \Gamma'_n, \Gamma'' = \Gamma''_n, D = D_n$.

Since the translation system is a conservative extension of the type system, we have a derivation $p \vdash^{A} \Gamma\ \{\mathbf{while}\ E\ C\}\ \Gamma'$, and hence by Lemma A.2 that

$$\forall x. \Gamma(x) \neq \Gamma'(x) \Rightarrow p \sqsubseteq \Gamma'(x).$$

this, together with the fact that $\Gamma \sqsubseteq \Gamma'$ means that every assignment in $\Gamma' := \Gamma$ is typeable, and hence that

$$p \vdash \Gamma' := \Gamma \qquad (19)$$

Similarly with the subderivation $p \sqcup t_n \vdash \Gamma'\ \{C \rightsquigarrow D\}\ \Gamma''$ we get, using Lemma A.2 that

$$\forall x. \Gamma'(x) \neq \Gamma''(x) \Rightarrow p \sqcup t_n \sqsubseteq \Gamma''(x).$$

and in the same manner as above we can conclude that

$$p \sqcup t_n \vdash \Gamma' := \Gamma''. \qquad (20)$$

Furthermore, Lemma 7.8 gives

$$\vdash E^{\Gamma'} : t_n, \qquad (21)$$

and the inductive hypothesis gives us

$$p \sqcup t_n \vdash D. \qquad (22)$$

Putting this together we obtain the concluding derivation

$$\dfrac{\dfrac{(19)}{p \vdash \Gamma' := \Gamma} \quad \dfrac{\dfrac{(21)}{\vdash E^{\Gamma'} : t_n} \quad \dfrac{\dfrac{(22)}{p \sqcup t_n \vdash D} \quad \dfrac{(20)}{p \sqcup t_n \vdash \Gamma' := \Gamma''}}{p \sqcup t_n \vdash D\ ;\ \Gamma' := \Gamma''}}{p \vdash \mathbf{while}\ E^{\Gamma'}\ (D\ ;\ \Gamma' := \Gamma'')}}{p \vdash \Gamma' := \Gamma\ ;\ \mathbf{while}\ E^{\Gamma'}\ (D\ ;\ \Gamma' := \Gamma'')}$$

□