On Flow-Sensitive Security Types

Sebastian Hunt
Department of Computing
School of Informatics, City University
London EC1V 0HB, 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 in-...
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 $1 \subseteq \mathbb{H}$, 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. [CHHI02]. 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 $\text{Var}$. A flow-insensitive type system, such as that in [VS96], 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 $t$ before the assignment and $y$ has type \( \mathbb{H} \), then after the assignment $x$ must be considered to have type \( \mathbb{H} \).

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 $\Gamma \vdash \mathcal{L} \Gamma \{ C \} \Gamma'$ where $\mathcal{L} \in \mathcal{L}$, and $\Gamma, \Gamma'$ are type environments of type $\text{Var} \rightarrow \mathcal{L}$. The inference rules are shown in Table 1. The idea is that if $\Gamma$ describes the security levels of variables which hold before execution of $C$, then $\Gamma'$ will describe the security levels of those variables after execution of $C$. The type $\mathcal{L}$ represents the usual “program counter” level and serves to eliminate indirect information flows; the derivation rules ensure that only variables which end up (in $\Gamma'$) with types greater than or equal to $\mathcal{L}$ may be changed by $C$. We write $\vdash \mathcal{L} \Gamma \{ C \} \Gamma'$ to mean $\vdash \mathcal{L} \Gamma \{ C \} \Gamma'$.

\begin{table}[h]
\centering
\begin{tabular}{ll}
\hline
$\text{Skip}$ & $\Gamma \vdash \mathcal{L} \{ \text{skip} \} \Gamma$
\
$\text{Assign}$ & $\Gamma \vdash \mathcal{L} \{ E : t \} \Gamma$
\
$\text{Seq}$ & $\Gamma \vdash \mathcal{L} \{ C_1 \} \Gamma' \quad \Gamma \vdash \mathcal{L} \{ C_2 \} \Gamma''$
\
$\text{If}$ & $\Gamma \vdash \mathcal{L} \{ E : t \} \Gamma' \quad \Gamma \vdash \mathcal{L} \{ \text{if} \ E \ C_1 C_2 \} \Gamma''$
\
$\text{While}$ & $\Gamma \vdash \mathcal{L} \{ E : t \} \Gamma' \quad \Gamma \vdash \mathcal{L} \{ \text{while} \ E \ C \} \Gamma''$
\
$\text{Sub}$ & $\Gamma_1 \vdash \mathcal{L} \{ C \} \Gamma'_1 \quad \Gamma_2 \vdash \mathcal{L} \{ C \} \Gamma'_2$
\
\hline
\end{tabular}
\caption{Flow-Sensitive Type Rules}
\end{table}

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 $\subseteq \mathcal{L}$ should be visible to outputs with type $\mathcal{L}$. More precisely, given a derivation $\Gamma \vdash \mathcal{L} \{ C \} \Gamma'$, 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 $\Gamma(y) \subseteq \mathcal{L}$. Following [HS91, SS01, HR98] we formalise this using equivalence relations.

**Definition 3.1.** Let $\mathcal{R}$ and $\mathcal{S}$ be equivalence relations on stores. We say that program $\mathcal{C}$ maps $\mathcal{R}$ into $\mathcal{S}$, written $\mathcal{C} : \mathcal{R} \Rightarrow \mathcal{S}$, iff, for all $\sigma, \rho$, if $(\mathcal{C}, \sigma) \downarrow \sigma'$ and $(\mathcal{C}, \rho) \downarrow \rho'$ then $\sigma \mathcal{R} \rho \Rightarrow \sigma' \mathcal{S} \rho'$.
We note that this is a partial correctness condition: it allows $C$ to terminate on $\sigma$ but diverge on $\rho$, 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 $\Gamma : \text{Var} \rightarrow \mathcal{L}$ and $t \in \mathcal{L}$, we write $\equiv_{P, t}$ for the equivalence relation on stores which relates stores which are equal on all variables having type $\subseteq t$ in environment $\Gamma$, thus: $\sigma =_{t, \Gamma} \rho$ iff $\forall x. \Gamma(x) \subseteq t \Rightarrow \sigma(x) = \rho(x)$.

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 $p \vdash_{c} \Gamma \{C\} \Gamma'$ holds if both the following conditions are satisfied:
1. For all $\sigma, \sigma', x$, if $(C, \sigma) \downarrow \sigma'$ and $\Gamma'(x) \not\supseteq p$, then $\sigma'(x) = \sigma(x)$.
2. For all $t \in \mathcal{L}$, $C : (=_{t, \Gamma}) \Rightarrow (=_{t', \Gamma'})$.

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

**Theorem 3.3 (Semantic Soundness).** $p \vdash_{c} \Gamma \{C\} \Gamma' \Rightarrow p \vdash_{c} \Gamma \{C\} \Gamma'$.

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 $\Gamma'(x) \not\supseteq p$ 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 $C = \{ \text{if } (h == 0) \{ l := h \} \}$.

This is semantically equivalent to $l := 0$ so it is clear that $\vdash_{c} \Gamma \{C\} \Gamma'$ holds for arbitrary $\Gamma$. However, for $\Gamma(h) = H$, $\Gamma(l) = L$, with $L \subset H$, $\not\vdash_{c} \Gamma \{C\} \Gamma'$, because $\vdash_{c} \Gamma \{h == 0\} : H$ and the assignments to $l$ force $\Gamma'(l) \not\supseteq H$.

**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 $\Gamma'$ such that $p \vdash_{c} \Gamma \{C\} \Gamma'$. 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 $\Gamma', \Gamma_1', \ldots, \Gamma_n'$ 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 $p, \Gamma$, there exists a unique $\Gamma'$ such that $p \vdash_{c} \Gamma \{C\} \Gamma'$ and furthermore, the corresponding function $A_{c}^{\mathcal{L}}(p, \Gamma) \rightarrow \Gamma'$ is monotone.
2. If $p \vdash_{c} \Gamma \{C\} \Gamma'$ then $A_{c}^{\mathcal{L}}(p, \Gamma) \subseteq \Gamma'$.
3. If $p \vdash_{c} \Gamma \{C\} \Gamma'$ then $p \vdash_{c} \Gamma \{C\} \Gamma'$.

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

**Proof of Algorithmic Correctness.** Proof of part 1 of the theorem is by induction on the structure of the command. The interesting case is **while** $E$ $C$. By induction hypothesis, $A_{c}^{\mathcal{L}}$ is well-defined and monotone. It follows that the sequences $\Gamma, \Gamma_1', \Gamma_2', \ldots$ and $\bot, \Gamma_0', \Gamma_1', \ldots$ may be constructed as $\Gamma, F(\Gamma), F^2(\Gamma), \ldots$ and $\bot, G(\bot), G^2(\bot), \ldots$, with $F$ and $G$ being monotone functions derived from $A_{c}^{\mathcal{L}}$: 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 $\Gamma_n'$ 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 $p \vdash_{c} \Gamma \{C\} \Gamma'$ 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 $\mathcal{P}$(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 $\mathcal{P}$(Var); anticipating this, we introduce some terminology.

**Definition 5.1.** The universal lattice is the flow lattice $\mathcal{P}$(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 $\Delta, \Delta'$ range over type environments just in the universal system (thus $\Delta, \Delta' : \text{Var} \rightarrow \mathcal{P}$(Var)).
\[
\begin{align*}
\text{If} & \quad \Gamma \vdash E : t \quad p \sqcup t \vdash^A \Gamma \quad \{C_i\} \quad \Gamma'_{i} \quad i = 1, 2 \\
\text{While} & \quad \Gamma'_i \vdash E : t_i \quad p \sqcup t_i \vdash^A \Gamma'_i \quad \{C_i\} \quad \Gamma''_i \quad 0 \leq i \leq n \\
& \quad p \vdash^A \Gamma \{\text{while} \ E\} \Gamma''_n
\end{align*}
\]

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 independent of the original value of \(y\), assuming that all the

independence pairs in \(T\) are in order for \(G\) to be the orthogonal

pair. Thus, the order relationships between the two pairs of type enviro

nments: For the

 Definition 6.1.

 An \(L_1\)-typing \(\Gamma_1 \{\cdot\} \Gamma_1'\) is said to subsume an \(L_2\)-typing

\(\Gamma_2 \{\cdot\} \Gamma_2'\) iff, for all commands \(C\)

\[ \vdash_{\epsilon_1} \Gamma_1 \{\cdot\} \Gamma_1' \Rightarrow \vdash_{\epsilon_2} \Gamma_2 \{\cdot\} \Gamma_2' \]

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 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 \(G \vdash \Delta \{C\} \Delta' \) 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 \(P(\text{Var})\)-derivation. For example, consider the second side condition on the rule IFAB. If we re-state this in its contrapositive form

\[ (\exists x \in \text{fv}(E), w \notin \Delta(x) \Rightarrow \exists w \in G') \]

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

\[ G' \supseteq G \cup \bigcup_{x \in \text{fv}(E)} \Delta(x) \]

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 \(G' = G \cup \bigcup_{x \in \text{fv}(E)} \Delta(x)\). 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 dif-

ferent 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 \(G, \Gamma : \text{Var} \to \mathcal{L}\), 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 \(L_1\)-typing \(\Gamma_1 \{\cdot\} \Gamma_1'\) is said to subsume an \(L_2\)-typing \(\Gamma_2 \{\cdot\} \Gamma_2'\) iff, for all commands \(C\)

\[ \vdash_{\epsilon_1} \Gamma_1 \{\cdot\} \Gamma_1' \Rightarrow \vdash_{\epsilon_2} \Gamma_2 \{\cdot\} \Gamma_2' \]

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. \(L_1\)-typing \(\Gamma_1 \{\cdot\} \Gamma_1'\) subsumes \(L_2\)-typing \(\Gamma_2 \{\cdot\} \Gamma_2'\) iff, for all \(x, y \in \text{Var}\):

\[ \Gamma_1(x) \subseteq \Gamma_1'(y) \Rightarrow \Gamma_2(x) \subseteq \Gamma_2'(y) \]

Proof. For the only if direction we show the contrapositive. Assume \(\Gamma_1(x) \equiv \Gamma_1'(y) \) and \(\Gamma_2(x) \equiv \Gamma_2'(y)\). We must find some command \(C\) such that \(\vdash_{\epsilon_1} \Gamma_1 \{C\} \Gamma_1'\) but \(\vdash_{\epsilon_2} \Gamma_2 \{C\} \Gamma_2'\). Let \(\{z_1, \ldots, z_n\} = \text{Var} \setminus \{y\}\) 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 \(C : (=_{\epsilon_1,t}) \Rightarrow (=_{\epsilon_2,t})\) holds for all \(t\) and

\[ \vdash_{\epsilon_2} \Gamma_1 \{C\} \Gamma_1' \]

For the if direction, assume

\[ (A1) \quad \Gamma_1(x) \subseteq \Gamma_1'(y) \Rightarrow \Gamma_2(x) \subseteq \Gamma_2'(y) \\
(A2) \quad \vdash_{\epsilon_1} \Gamma_1 \{C\} \Gamma_1' \]

We have to show, for all \(s \in \mathcal{L}_2, C : (=_{\epsilon_2,s} \Rightarrow (=_{\epsilon_2,t,s})\). Suppose \(\sigma =_{\epsilon_2,s} \rho\) and \((C, \sigma) \downarrow \sigma'\) and \((C, \rho) \downarrow \rho'\) and \(\Gamma_2(y) \subseteq s\). We must show \(\sigma'(y) = \rho'(y)\). Now, for any \(x\), \(\Gamma_2(x) \subseteq \Gamma_2'(y) \Rightarrow \Gamma_2(x) \subseteq s \Rightarrow \sigma(x) = \rho(x)\). Hence, by (A1), \(\Gamma_1(x) \subseteq \Gamma_1'(y) \Rightarrow \sigma(x) = \rho(x)\), thus \(\sigma =_{\epsilon_1,t} \rho\), where \(t = \Gamma_1(y)\). Hence, by (A2), \(\sigma' =_{\epsilon_1,t} \rho'\), hence \(\sigma'(y) = \rho'(y)\) as required. \(\square\)
This result shows that the semantic content of a judgement \( \Gamma \vdash C \mathbf{c} \Gamma' \) 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 \( x \). Alternatively, we may consider the contrapositive form of Theorem 6.2, which says that \( C \mathbf{c} \Gamma \vdash 1 \) subsumes \( 2 \mathbf{c} \Gamma \vdash 1 \) 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 \( L_1 \)-typing which subsumes an \( L_2 \)-typing, and suppose we find that the \( L_2 \)-typing is not derivable for \( C \) in the \( L_2 \)-typing system. Then it is possible to verify the soundness of the \( L_2 \)-typing for \( C \) directly, by deriving the subsuming \( L_1 \)-typing in the \( L_1 \)-typing system instead! We might expect this to happen in the case where \( L_1 \) has more points, and is therefore able to make more refined dependency distinctions, than \( L_2 \). Consider the examples shown in Figure 2, where \( L \) is the four point lattice depicted. It can readily be verified that the \( P(\text{Var}) \)-typing subsumes the \( L \)-typing and both judgements are derivable.

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 \( \Gamma : \text{Var} \to L_1 \) and a renaming function \( f : L_1 \to L_2 \), we write \( f^*(\Gamma) : \text{Var} \to L_2 \) for the pointwise extension of \( f \) to \( \Gamma \), thus \( f^*(\Gamma)(x) \equiv \Gamma(f(x)) \).

**Lemma 6.4** (Monotone Renaming). Let \( f : L_1 \to L_2 \) be monotone. Then \( p \vdash e_1 \Gamma \{ C \} \Gamma \mathbf{c} (p) \vdash e_2 f^*(\Gamma) \{ C \} f^*(\Gamma') \).

**Proof.** By induction on the height of the \( L_1 \)-derivation. We present the Assign and While cases by way of illustration.

**Case: Assign.** We have an \( L_1 \)-derivation of the form:

\[
\Gamma \vdash e_1 : E \mathbf{c} t
\]

where \( \Gamma' = \Gamma[y \mapsto p \cup t] \). We can construct an \( L_2 \)-derivation:

\[
f^*(\Gamma)(y) \vdash e_2 f^*(\Gamma) \{ y := E \} f^*(\Gamma)[x := f(p \cup t' \cup t)]
\]

It suffices to show that \( f^*(\Gamma)(y) \vdash f(p \cup t' \cup t) \subseteq f^*(\Gamma') \) (since we can then use Sub). By the definitions, \( f^*(\Gamma)(y) \vdash f(p \cup t' \cup t) \subseteq f^*(\Gamma')(y) \) for all \( y \neq x \) and it remains to show \( f(p \cup t) \subseteq f(p \cup t) \).

Now by monotonicity of \( f \) we have

\[
t' \equiv \bigcup_{y \in \text{fv}(E)} f^*(\Gamma)(y) \subseteq f^*(\Gamma)[x := f(p \cup t)]
\]

Finally, using this and monotonicity of \( f \) again, we have \( f(p \cup t) \subseteq f(p \cup t) \).

**Case: While.** We have an \( L_1 \)-derivation of the form:

\[
\Gamma \vdash e_1 : E \mathbf{c} \mathbf{w} \mathbf{c} \\Gamma \{ C \} \Gamma
\]

By induction hypothesis we have \( f(p \cup t) \subseteq f(p \cup t) \subseteq f(p \cup t) \).

---

**Table 3. Amtoft-Banerjee Hoare Logic**

<table>
<thead>
<tr>
<th>Context</th>
<th>Rule</th>
<th>Judgement</th>
</tr>
</thead>
</table>
| AssignAB | \( G \vdash \emptyset \{ x := E \} \emptyset' \) | if \( \forall y, z \in \text{fv}(E) \).
| SeqAB | \( G \vdash \{ C \} \emptyset' \) | \( x \neq y \Rightarrow w \in \emptyset(y) \)
| IFAB | \( G \vdash \{ C \} \emptyset' \) | \( x = y \Rightarrow w \notin G \mathbf{c} \mathbf{w} \forall z \in \text{fv}(E) \).
| WhileAB | \( G \vdash \{ \text{while } E C \} \emptyset' \) | \( G \subseteq G' \) and \( w \neq G \Rightarrow \forall x \in \text{fv}(E) \).
| SubAB | \( G_1 \vdash \{ C \} \emptyset' \) | \( G_2 \subseteq G_1 \mathbf{c} \mathbf{w} \subseteq \emptyset_1 \mathbf{w} \subseteq \emptyset_2 \mathbf{w} \).

---

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.
Lemma 6.7. Given \( \Gamma \) a monotone renaming function to translate the \( \mathcal{L} \)-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 \( \vdash \mathcal{E} \Gamma (\{ C \}) \Gamma' \) from a particular form of derivation in the universal system. To do this we construct, for each choice of \( \Gamma \), an abstract interpretation [CC77] which is given by a pair of monotone renaming maps:

\[
\Gamma \quad \text{def} \quad \bigg\{ \begin{array}{ll}
\alpha & \Gamma(x) \\
\gamma & \{ x \mid \Gamma(x) \subseteq t \}
\end{array}
\]

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 \( \alpha : \mathcal{L}_2 \rightarrow \mathcal{L}_1 \) and \( \gamma : \mathcal{L}_1 \rightarrow \mathcal{L}_2 \) and such that \( \alpha(s) \subseteq t \iff s \subseteq \gamma(t) \). Key properties of a GC are that \( \alpha, \gamma \) are both monotone, \( \alpha \circ \gamma \subseteq \mathcal{L}_1, \gamma \circ \alpha \subseteq \mathcal{L}_2 \), \( \alpha \) preserves joins and \( \gamma \) preserves meets. Furthermore, the two component maps uniquely determine each other, thus:

\[
\alpha(s) = \bigvee \{ t \mid s \subseteq \gamma(t) \} \quad \text{(5)}
\]

\[
\gamma(t) = \bigwedge \{ s \mid \alpha(s) \subseteq t \} \quad \text{(6)}
\]

Lemma 6.6. For any \( \Gamma : \text{Var} \rightarrow \mathcal{L} \), the pair \( \langle \alpha, \gamma \rangle \) is a Galois Connection between \( \mathcal{P}^{\text{Var}} \) and \( \mathcal{L} \).

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

Lemma 6.7. \( \Delta_0 \) is a subsumes \( \Gamma \).

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

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.

\[ f = \bigvee \{ \Gamma_2(x) \mid \Gamma_1(x) \subseteq s \} \]

1 Though we can read it off easily enough once we have the proof. It is:

\[ f(s) = \bigvee \{ \Gamma_2(x) \mid \Gamma_1(x) \subseteq s \} \]

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

For the \( \Leftarrow \) direction, Monotone Renaming gives \( \vdash \alpha_l(\Delta_0) \{ C \} \) \( \alpha_l(\gamma_l(\Gamma')) \). Now, by (5), \( \alpha_l(\{ x \}) = \bigvee \{ t \mid x \in \gamma_l(t) \} \) = \( \bigvee \{ t \mid \Gamma(x) \subseteq t \} \) = \( \Gamma(x) \), thus \( \alpha_l(\Delta_0) = \Gamma \). By standard properties of a GC, \( \alpha_l(\gamma_l(\Gamma')) \subseteq \Gamma' \). Thus the required derivation follows by appending a single use of Sub. \( \Box \)

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

Proof of Internal Completeness. Assume \( \mathcal{L}_1 \)-typing \( \Gamma_1 \) \( \{ \} \Gamma_i' \) subsumes \( \mathcal{L}_2 \)-typing \( \Gamma_2 \) \( \{ \} \Gamma_2' \) and \( \vdash \mathcal{E} \Gamma_1 \{ C \} \Gamma_1' \). We must show \( \vdash \mathcal{E} \Gamma_2 \{ C \} \Gamma_2' \) which, by the Canonical Derivations lemma, is equivalent to

\[ \vdash \gamma_l(\Gamma_2) \Gamma_2' \]

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

\[ \vdash \gamma_l(\Gamma_1) \Gamma_1' \]

It thus suffices to show

\[ \gamma_l(\Gamma_1) \subseteq \gamma_l(\Gamma_2) \]

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

1 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, GR50]. 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 \( \Gamma \), 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

\[ \vdash \Delta_0 \{ C \} \Delta \]

where \( \Delta \) is the smallest \( \Delta' \) such that \( \vdash \Delta_0 \{ C \} \Delta' \) (recall that, by Corollary 4.2, this exists and is given by \( \mathcal{A}_P^{\text{Var}}(\emptyset, \Delta_0) \)). The Canonical Derivations lemma shows that derivability of any given \( \vdash \mathcal{E} \Gamma \{ C \} \Gamma' \) is equivalent to \( \forall x. \Delta_C(x) \subseteq \gamma_l(\Gamma'(x)) \), which unpacks to:

\[ y \in \Delta_C(x) \Rightarrow \Gamma(y) \subseteq \Gamma'(x) \]
In fact, we can show that $\Delta_0 \{ \} \Delta_C$ is a principal typing for $C$, in the sense defined by Wells [Wel02]. Transposed to our setting, Wells makes the following definitions:

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

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

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

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

**Proof of Principal Typing.** By definition of $\Delta_C$, $\vdash \Delta_0 \{ C \} \Delta_C$. Suppose $\vdash \Gamma, \{ \} \Gamma'$ as 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 $\Delta_0 \{ \} \Delta_C$ subsumes $\Gamma, \{ \} \Gamma'$. By Lemma 6.7, $\Delta_0 \{ \} \gamma_C(\Gamma')$ subsumes $\Gamma, \{ \} \Gamma'$ so, by Lemma 6.10, it suffices to show $\Delta_C \subseteq \gamma_C(\Gamma')$. By the Canonical Derivations lemma (using $\vdash \Gamma, \{ C \} \Gamma'$), $\vdash \Delta_0 \{ \} \gamma_C(\Gamma')$, so by definition of $\Delta_C$, $\Delta_C \subseteq \gamma_C(\Gamma')$. 

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 adjoint a "program counter variable" to $\Gamma$, so the universal lattice becomes $P(\{ \text{Var} + \{ \text{pc} \} \})$.

### 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 $P(\{ \text{Var} \})$ 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 $T \text{Var} \cong \text{Var}$, ranged over by $\beta$. Assume also some particular 1-1 mapping between the two sets: we write $\beta_\alpha$ for the type variable associated to program variable $x$. In this view, $\Delta_0$ is a type environment which assigns a unique polymorphic variable $\beta_\alpha$ to each $x$. The application of $\alpha_T$ to $\Delta_0$ in the proof (of the) Canonical Derivations lemma amounts to an instantiation of the type variables to produce $\Gamma$. In general, $\alpha_T$ interprets a set $T$ of type variables as the lub of the interpretations of its elements. Thus, in this view, types in the $P(T \text{Var})$ lattice can be thought of as formal lubs, which can be interpreted as elements in any lattice $L$ by fixing an interpretation $I$ for each $\beta$.

As above, let $\Delta_C$ be the smallest $\Delta'$ such that $\vdash \Delta_0 \{ C \} \Delta'$. It can be shown that fixing $\Gamma$ and calculating $\alpha_T(\Delta_C)$ gives us $\Delta_C(\Gamma) \Rightarrow \text{the smallest } \Gamma' \text{ such that } \vdash \Gamma, \{ C \} \Gamma'$. More interestingly, $\Delta_C$ may also be used in the reverse direction, to calculate the greatest $\Gamma$ such that $\vdash \Gamma, \{ C \} \Gamma'$ for a given $\Gamma'$.

The idea is to construct an interpretation $I : T \text{Var} \rightarrow L$ which "unifies" $\Delta_C$ and $\Gamma'$, in the sense that

$$\alpha_I(\Delta_C(x)) \subseteq \Gamma'(x) \quad (11)$$

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

$$I(\beta) \overset{\text{def}}{=} \bigcap \{ \Gamma'(x) | \beta \in \Delta_C(x) \} \quad (12)$$

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

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

**Proof.** By the Canonical Derivations lemma, it suffices to show that the $\Gamma$ defined is the greatest such that

$$\vdash \Delta_0 \{ C \} \gamma_C(\Gamma') \quad (13)$$

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

It remains to show that $\gamma_C(\Gamma'(x)) \supseteq \Delta_C(x)$ and $\Gamma'(x) \subsetneq \Gamma(x)$, thus $\beta_x \in \Delta_C(x)$ but $\beta_x \not\in \gamma_C(\Gamma'(x))$. 

### 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 $l \equiv \top$, and the type environment $\Gamma = \{ l : l, h : h \}$:

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

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:

$$l_n := h_0; l_s := 0; h := 0; l := h_1$$

where $h_0$ and $h_1$, for example, are distinct variables. The initial inputs $l$ and $h$ are here represented by $l_n$ and $h_1$ respectively. In a flow-insensitive security type system the program is deemed secure because $l_n$ (and $h_1$) 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, $F \text{Var}$, to be the set of type-indexed variables

$$F \text{Var} \overset{\text{def}}{=} \{ x_\beta | x \in \text{Var}, \beta \in \mathcal{L} \}$$

To distinguish the fixed variables from the “ordinary” variables we will henceforth refer to the variables in $\text{Var}$ 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 $x_{l_0}$ and $x_{l_1}$.
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 \( \text{fv}(E) \) denote the free floating variables (as before), and define \( \text{ffv}(E) \) 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

\[
\Gamma \vdash E : t \text{ iff } t = \bigcup_{x \in \text{fv}(E)} \Gamma(x) \sqcup \bigcup_{x \in \text{ffv}(E)} \Gamma(x)
\]

The fixed type rule is simply:

\[
\text{Fixed-Assgn} \quad \frac{\Gamma \vdash E : s \quad s \sqsubseteq t, \quad p \sqsubseteq t}{p \vdash E \quad \Gamma} \]

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 \( (\text{ffv}(C) = \emptyset) \), 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 \( p \vdash E \quad \Gamma \) for arbitrary \( \Gamma \). It should be straightforward to see that derivations of this form correspond exactly to derivations in e.g. Volpano, Smith and Irvine’s system [VS96], 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

\[
p \vdash \Gamma \quad C \leadsto D \quad \Gamma'
\]

(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 \( \Gamma \) and \( \Gamma' \), let \( \Gamma := \Gamma' \) denote the set

\[
\{x := x_1 | \Gamma(x) = s, \Gamma'(x) = t, s \neq t\}
\]

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 \( \Gamma \), let \( E\Gamma \) denote the expression obtained by replacing each floating variable \( x \) in \( E \) with the fixed variable \( x \) 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 \( p \vdash \Gamma \quad C \leadsto D \quad \Gamma' \) is that for any program point in \( D \) corresponding to a point in \( C \), for each variable \( x \), only one member of the family \( \{x_1 \in L\} \) 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 \( \sigma \) range over floating variable stores and let \( \rho \) range over fixed variable stores. Then for each type environment \( \Gamma \) we define the compatibility relation as

\[
\sigma \sim \rho \iff \forall x \in \text{Var}(p) \sigma(x) = \rho(x_{\Gamma(x)})
\]

**Theorem 7.7 (Translation Correctness).**

If \( p \vdash \Gamma \quad C \leadsto D \quad \Gamma' \) then for all \( \sigma \) and \( \rho \) such that \( \sigma \sim \rho \),

- \((C, \sigma) \nleftrightarrow \sigma' \Rightarrow \exists \rho', (D, \rho) \nleftrightarrow \rho' \wedge \sigma' \sim \rho' ^{\Gamma'}
- \((D, \rho) \nleftrightarrow \rho' \Rightarrow \exists \sigma', (C, \sigma) \nleftrightarrow \sigma' \wedge \sigma' \sim \rho' ^{\Gamma'}

**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 \( p \vdash \Gamma \quad E \equiv D \quad \Gamma' \) then

Follows directly from the definitions.

**Theorem 7.9 (Static Soundness).**

If \( p \vdash \Gamma \quad C \leadsto D \quad \Gamma' \) then

**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 \( n \) 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 \( H \):

\[
\text{if } y_1 \text{ then } \text{if } y_2 \text{ then } \ldots \text{ if } y_n \text{ then } \text{if } h \text{ then } x_1 := 0; \ldots; x_n := 0
\]
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 \(t\). Writing \(X_n := X_1\) for the sequence of assignments \(x_{1n} := x_1; \ldots ; x_{nn} := x_n\), the transformed program is:

\[
\begin{align*}
\text{if } y_{1n} & \text{ then } y := y + 1; w := z \\
\text{while } x > 0 & \\
\quad z := z + w \\
\quad x := x - 1 \\
\quad z := x \\
\end{align*}
\]

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

\[
\begin{array}{|c|c|c|}
\hline
\text{Code} & \text{Environment change} & \text{Translated code} \\
\hline
\text{if } x = 0 \text{ then } y := y + 1; w := z & [y \mapsto H; w \mapsto H] & \text{if } x_M = 0 \text{ then } y_n := y_{n+1}; w_n := z_n \\
\text{while } x > 0 & \text{else } y_n := y_n; w_n := w_n \\
\quad z := z + w \\
\quad x := x - 1 \\
\quad z := x & & \text{while } x_M > 0 \\
\quad \text{else } X_n := X_L \\
\quad \text{else } X_n := X_L \
\hline
\end{array}
\]

Figure 3. Example translation derivation

where 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 \(\phi\)-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 \( P(V_{\text{var}}) \)).
- 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 \( L \)-typings than the \( L \)-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 Svenningsson 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


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. \( \langle \Gamma' \rangle := \Gamma, \rho \Downarrow \rho' \) where \( \sigma \sim_\Gamma \rho' \)
2. \([E]\sigma = [E^\Gamma]\rho\)

Proof. 1. From definition 7.2, the effect of \( \Gamma' \) := \( \Gamma \) on state \( \rho \) 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 \( x \in \text{Var} \)
\[
\sigma(x) = \rho(x_{\Gamma'(x)}) = \rho(x_{\Gamma(x)}(x_{\Gamma'(x)}))
\]
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 \( p \vdash \Gamma \{C \sim D\} \Gamma', \sigma \sim_\Gamma \rho \) and that \( (C, \sigma) \Downarrow \sigma' \). We prove that \( \langle D, \rho \rangle \Downarrow \rho' \) where \( \sigma' \sim_\Gamma \rho' \) by induction on the derivation of \( (C, \sigma) \Downarrow \sigma' \) and cases according to the last rule applied:

Case: Assign. We have a derivation of the form
\[
\Gamma \vdash E : t \quad s \triangleq p \cup t
\]
\( p \vdash \Gamma \{x := E \mapsto s \} \langle x := E, \sigma \rangle \Downarrow \sigma \).

Suppose that \([E]\sigma = V\) and hence that
\( \langle x := E, \sigma(x) \rangle \Downarrow \sigma \Downarrow V \).

By Lemma A.1. \([E^\Gamma]\sigma = V\) and hence
\( \langle x := E^\Gamma, \sigma(x) \rangle \Downarrow \sigma \Downarrow V \).

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:
\[
\langle E \rangle \sigma = \text{true} \quad (C, \sigma) \Downarrow \sigma' \quad \langle \text{while } E C, \sigma' \rangle \Downarrow \sigma''
\]
We have a translation derivation of the form
\[
\Gamma' \vdash E : t_i \quad p \cup t_i \vdash \Gamma_i \{C \sim D_i\} \Gamma_i' \quad 0 \leq i \leq n
\]
\( p \vdash \Gamma \{\text{while } E C \sim \Gamma : \Gamma \} \quad \text{while } E^{\Gamma'} (D ; \Gamma' := (\Gamma')) \}
\]
where \( \Gamma_0 = \Gamma, \Gamma_{i+1} = \Gamma_i' \cup \Gamma_i \) and \( \Gamma_i' = \Gamma_n' \cup \Gamma' \). Henceforth let \( W_D \) denote the subterm while \( E^\Gamma (D ; \Gamma' := (\Gamma'))\).

Assume that \( \sigma \sim_\Phi \rho \). 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
\[
\langle \Gamma' := \Gamma, \rho \rangle \Downarrow \rho_1 \text{ where } \sigma \sim_{\Gamma'} \rho_1
\]
and from Lemma A.1(2)
\[
\langle E', \rho_1 \rangle \Downarrow \text{true}.
\]
From the induction hypothesis for the subderivation for \( C \) we have
\[
\langle D, \rho_2 \rangle \Downarrow \rho_2 \text{ where } \sigma' \sim_{\Gamma'} \rho_2,
\]
and from Lemma A.1(1),
\[
\langle \Gamma' := \Gamma', \rho_2 \rangle \Downarrow \rho_3 \text{ where } \sigma' \sim_{\Gamma'} \rho_3
\]
Finally from the premises of the typing judgement, we also have the weaker judgement:
\[
p \vdash \Gamma' \{\text{while } E C \sim \Gamma' := \Gamma ; W_D \} \Gamma'
\]
Now we apply the induction hypothesis for the second evaluation premise, with respect to this judgement, to obtain
\[
\langle W_D, \rho_4 \rangle \Downarrow \rho'' \text{ where } \sigma'' \sim_{\Gamma'} \rho''
\]
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.

1. If \( p \vdash D \) and \( \rho' \sqsubseteq p \) then \( p' \sqsubseteq D \)
2. If \( p \vdash^+ \Gamma \{C\} \Gamma' \) 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:
\[
p \vdash^+ \Gamma \{x := E\} \Gamma[x := p \cup t]
\]
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
\[
p \cup t \vdash^+ \Gamma \{C_i\} \Gamma_i \quad i = 1, 2 \quad \Gamma' = \Gamma_i \cup \Gamma_j
\]
The induction hypothesis gives, for \( i = 1, 2, \)
\[\forall x. \Gamma(x) \neq \Gamma'(x) \Rightarrow p \sqsubseteq \Gamma_i'(x) \)
So suppose that for some particular \( x \) we have \( \Gamma(x) \neq \Gamma'(x) \). Since \( \Gamma' = \Gamma_i \cup \Gamma_j \) we must have \( \Gamma_i(x) \neq \Gamma_j(x) \) 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 \( p \sqsubseteq \Gamma_i'(x) \cup \Gamma_j'(x) \) as required.

Case: While. The rule provides a derivation of the form
\[
p \vdash^+ \Gamma \{\text{while } E C\} \Gamma_i
\]
\( \Gamma_i' \vdash E : t_i \quad p \cup t_i \vdash^+ \Gamma_i \{C_i\} \Gamma_i' \quad 0 \leq i \leq n \)
\( p \vdash^+ \Gamma \{\\text{while } E C\} \Gamma_0 \)
\( \Gamma_0 = \Gamma, \Gamma_{i+1} = \Gamma_i' \cup \Gamma_i \)
Assume \( \Gamma_n'(x) \neq \Gamma(x) \). Now suppose that \( \Gamma_n'(x) = \Gamma_i'(x) \) 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 \( n + 1. \)
Putting this together we obtain the concluding derivation
\[(\Gamma' := \Gamma, \rho) \Downarrow p_1\]
\[(\Gamma' := \Gamma, \rho) \Downarrow \text{true}\]  \hspace{3cm} (14)
\[(D, p_1) \Downarrow \rho_2\] \hspace{3cm} (15)
\[(\Gamma' := \Gamma, \rho_2) \Downarrow p_3\] \hspace{3cm} (16)
\[(W_D, p_3) \Downarrow \rho''\] \hspace{3cm} (17)
\[(\Gamma' := \Gamma) ; \text{while } E^{\Gamma''} (D; \Gamma' := \Gamma''), \rho) \Downarrow \rho''\] \hspace{3cm} (18)

Figure 4. Concluding derivation, While case, Theorem 7.7

By construction, \(\Gamma'_k(x) = \Gamma_k(x) \cup \Gamma(x)\) so, by supposition, \(\Gamma'_{k+1}(x) = \Gamma_k(x) \cup \Gamma(x)\). 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, \(p \triangleright t_i \sqsubseteq \Gamma''_i(x)\), 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. \(\Box\)

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

Case: Assign-t. The derivation is of the form
\[
p \vdash \Gamma \{x := E \leadsto s, := E'\} \Gamma[x := s]
\]
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:
\[
p \vdash \Gamma \{\text{while } E C \leadsto \Gamma' := \Gamma ; \text{while } E^{\Gamma'} (D; \Gamma' := \Gamma'')\} \Gamma'
\]
where \(\Gamma'_0 = \Gamma, \Gamma'_{i+1} = \Gamma'_i \sqcup \Gamma, \Gamma'_n = \Gamma'_n\) 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 \(p \vdash^\Gamma \{\text{while } E C\} \Gamma'\), 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
\[
p \vdash \Gamma' := \Gamma \tag{19}\]
Similarly with the subderivation \(p \sqcup t_n \vdash \Gamma' \{C \leadsto D\} \Gamma''\) 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
\[
p \sqcup t_n \vdash \Gamma' := \Gamma''\tag{20}\]
Furthermore, Lemma 7.8 gives
\[
p \vdash E^{\Gamma'} : t_n, \tag{21}\]
and the inductive hypothesis gives us
\[
p \sqcup t_n \vdash D. \tag{22}\]
Putting this together we obtain the concluding derivation
\[
p \vdash E^{\Gamma'} : t_n, \tag{21}\]
\[
p \sqcup t_n \vdash D, \tag{22}\]
\[
p \sqcup t_n \vdash \Gamma' := \Gamma'' \tag{20}\]
\[
p \vdash \Gamma' := \Gamma \tag{19}\]
\[
p \vdash \text{while } E^{\Gamma'} (D; \Gamma' := \Gamma''), \tag{19}\]

\(\Box\)