Chapter 9

Information Flow Control

This chapter discusses the specification and enforcement of *information flow policies*. Such policies concern whether the initial values of certain variables may directly or indirectly affect the values of certain other variables and/or may affect program termination. The variables might correspond to regions of memory, files, input channels, or output channels. For enforcing confidentiality, an information flow policy would specify that values stored in certain variables not be affected by secrets; for enforcing integrity, it would specify that values stored in certain variables not be affected by values derived from untrusted sources.

9.1 Labels Specifying Information Flow Policies

An information flow policy for a program (i) employs a *label assignment* $\Gamma(\cdot)$ to associate a label $\Gamma(v)$ with each program variable v and (ii) gives a partial order[1] $\sqsubseteq$ (with complement $\not\sqsubseteq$) on the set Λ of possible labels.

- $\Gamma(v) \sqsubseteq \Gamma(w)$ specifies that the initial value of variable v is allowed to affect the value of variable w at designated points during executions.
- $\Gamma(v) \not\sqsubseteq \Gamma(w)$ specifies that the initial value of variable v is not allowed to affect the value of variable w at designated points during executions.

What is considered a "designated point" during an execution depends on the information flow policy. With some information flow policies, the designated

[1] A *relation* ρ on a set *Vals* is a subset of $\{\langle a, b\rangle \mid a, b \in \mathit{Vals}\}$. Its *complement* $\not\rho$ is the set $\{\langle a, b\rangle \mid a, b \in \mathit{Vals}\} - \rho$. A *partial order* ρ on *Vals* is a relation on *Vals* that satisfies the following properties, where (as is conventional) infix notation $a\,\rho\,b$ is used for $\langle a, b\rangle \in \rho$.

Reflexive: $a\,\rho\,a$ for all $a \in \mathit{Vals}$.
Transitive: $a\,\rho\,b$ and $b\,\rho\,c$ implies $a\,\rho\,c$ for all $a, b, c \in \mathit{Vals}$.
Antisymmetric: $a\,\rho\,b$ and $b\,\rho\,a$ implies $a = b$ for all $a, b \in \mathit{Vals}$.

A partial order ρ does not have to relate all pairs of elements $a, b \in \mathit{Vals}$, so if $a \not\rho\ b$ holds it is possible that neither $a\,\rho\,b$ nor $b\,\rho\,a$ holds.

points are the final states of terminating executions; with others, the designated points are all intermediate states produced during terminating and nonterminating executions.

For each label $\lambda \in \Lambda$, partial order $\sqsubseteq$ partitions the set V of program variables into subsets

$$V_{\sqsubseteq\lambda} = \{v \mid \Gamma(v) \sqsubseteq \lambda\} \qquad V_{\not\sqsubseteq\lambda} = \{v \mid \Gamma(v) \not\sqsubseteq \lambda\}$$

where the initial value of no variable from $V_{\not\sqsubseteq\lambda}$ is allowed to affect the value of any variable from $V_{\sqsubseteq\lambda}$ at designated points during an execution. These partitions facilitate formulating information flow policies in which the restrictions that are being imposed on access to each variable v also apply to all variables storing values affected by v. With such policies

$$(\forall \lambda, \lambda' \in \Lambda\colon\ \lambda \sqsubseteq \lambda' \Rightarrow R(\lambda) \preceq R(\lambda')) \tag{9.1}$$

holds, where $R(\lambda)$ denotes the restrictions imposed on access to variables that have a label λ and relation $R(\lambda) \preceq R(\lambda')$ asserts that $R(\lambda')$ is more restrictive than $R(\lambda)$ or, equivalently, compliance with $R(\lambda')$ implies compliance with $R(\lambda)$.

An obvious question is whether satisfying (9.1) is ever useful. Why couldn't we just use a fine-grained implementation of ordinary access control in order to restrict principals from reading or writing individual variables? Such access control restrictions, however, could not prevent a program that is authorized to read a secret variable x and to write a public variable y from copying x to y. But an information flow policy satisfying (9.1) would prohibit such leaks. This is because $\Gamma(x) \sqsubseteq \Gamma(y)$ must hold for a principal to write y after reading x, so from (9.1), we conclude that restrictions on x and y must satisfy $R(\Gamma(x)) \preceq R(\Gamma(y))$. Therefore, accesses to y also must comply with $R(\Gamma(x))$. If $R(\Gamma(x))$ specifies that only a select set of principals are allowed to read secrets and, thus, are allowed to read x, then $R(\Gamma(y))$ cannot allow additional principals to read y.

9.1.1 Labels for Expressions

$\Gamma(\cdot)$ gives labels to variables, but programs involve expressions, too. The label $\Gamma_{\mathcal{E}}(E)$ that we associate with an expression E enables an information flow policy (i) to specify the variables and expressions that E is allowed to affect and (ii) to specify the variables and expressions that are allowed to affect E. Since uniary operators and infix binary operators can be viewed as syntactic sugar for function applications, no generality is lost if we limit consideration here to expressions constructed from constants, variables, and function applications $f(E_1, E_2, \ldots, E_n)$ having arguments E_i that are themselves expressions.

Constants. The label $\Gamma_{\mathcal{E}}(c)$ that we associate with a constant c ought not preclude an assignment statement from storing c into any variable. Therefore, we require that $\Gamma_{\mathcal{E}}(c) \sqsubseteq \Gamma(v)$ hold for any constant c and any variable v. That

leads to the definition

$$\Gamma_{\mathcal{E}}(c)\text{: } \bot_{\Lambda} \text{ for any constant } c \tag{9.2}$$

where $\bot_{\Lambda}$ satisfies

$$\bot_{\Lambda} \in \Lambda \;\; \wedge \;\; (\forall \lambda \in \Lambda\text{: } \bot_{\Lambda} \sqsubseteq \lambda). \tag{9.3}$$

Variables. The label $\Gamma_{\mathcal{E}}(v)$ that we associate with an expression that is a variable v should have the same restrictions as $\Gamma(v)$. That is achieved with the following definition.

$$\Gamma_{\mathcal{E}}(v)\text{: } \Gamma(v) \text{ for any variable } v. \tag{9.4}$$

Function Applications. Whether the value of $f(E_1, E_2, \ldots, E_n)$ is affected by the value of its argument E_i depends on f. The conservative choice for label $\Gamma_{\mathcal{E}}(f(E_1, E_2, \ldots E_n))$ would be a label that works for any function f. Such a label would allow (but not require) each argument E_i to affect the value of $f(E_1, E_2, \ldots E_n)$:

$$\Gamma_{\mathcal{E}}(E_i) \sqsubseteq \Gamma_{\mathcal{E}}(f(E_1, E_2, \ldots E_n)) \;\; \text{ for } 1 \le i \le n. \tag{9.5}$$

So (9.5) is the goal when devising a definition for $\Gamma_{\mathcal{E}}(f(E_1, E_2, \ldots E_n))$.

A value that is at least as large as any member of a set is called an *upper bound* for that set; a *least upper bound* is an upper bound that is not larger than any other upper bound. One way to satisfy (9.5) is by defining label $\Gamma_{\mathcal{E}}(f(E_1, E_2, \ldots E_n))$ to be an upper bound of set $\{\Gamma_{\mathcal{E}}(E_1), \Gamma(E_2), \ldots, \Gamma_{\mathcal{E}}(E_n)\}$. Among the upper bounds, the least upper bound is the best choice for label $\Gamma_{\mathcal{E}}(f(E_1, E_2, \ldots E_n))$, because it allows the value of $f(E_1, E_2, \ldots E_n)$ to affect more variables.

Least upper bounds for finite subsets $\{\lambda_1, \lambda_2, \ldots, \lambda_n\} \subseteq \Lambda$ having partial orders $\sqsubseteq$ typically are specified by using the idempotent, commutative, and associative *join* operator $\sqcup$, which satisfies the following axioms.

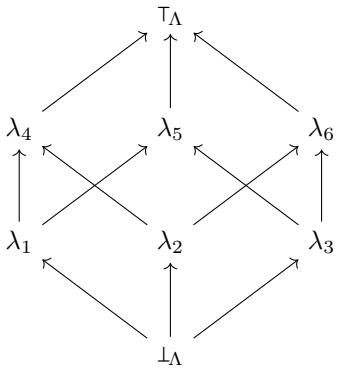
$$\lambda_i \sqsubseteq (\lambda_1 \sqcup \lambda_2 \sqcup \cdots \sqcup \lambda_n) \qquad \text{for } 1 \le i \le n \tag{9.6}$$

$$(\lambda_1 \sqsubseteq \lambda \;\wedge\; \lambda_2 \sqsubseteq \lambda \;\wedge\; \cdots \;\wedge\; \lambda_n \sqsubseteq \lambda) \;\Rightarrow\; (\lambda_1 \sqcup \lambda_2 \sqcup \cdots \sqcup \lambda_n) \sqsubseteq \lambda \tag{9.7}$$

Axiom (9.6) requires $\lambda_1 \sqcup \lambda_2 \sqcup \cdots \sqcup \lambda_n$ to be an upper bound for $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$, and axiom (9.7) requires $\lambda_1 \sqcup \lambda_2 \sqcup \cdots \sqcup \lambda_n$ to be a least upper bound. We can ensure that Λ contains least upper bound $\lambda_1 \sqcup \lambda_2 \sqcup \cdots \sqcup \lambda_n$ for any subset $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ of Λ simply (i) by adding to Λ an element $\top_{\Lambda}$ that satisfies $\lambda \sqsubseteq \top_{\Lambda}$ for all $\lambda \in \Lambda$, and (ii) by defining $\lambda \sqcup \lambda'$ to equal $\top_{\Lambda}$ for every pair of labels λ and λ' where $\lambda \sqcup \lambda' \notin \Lambda$ had been satisfied.

Figure 9.1 depicts a set of labels and some least upper bounds. Notice, not all labels in Λ are related by $\sqsubseteq$—for example, $\lambda_1 \not\sqsubseteq \lambda_6$ holds. The figure also illustrates that the converse[2] of (9.7) does not always hold—we have that $\lambda_6 \sqsubseteq \lambda_4 \sqcup \lambda_3$ holds (since $\lambda_4 \sqcup \lambda_3$ is $\top_{\Lambda}$) but neither $\lambda_6 \sqsubseteq \lambda_4$ nor $\lambda_6 \sqsubseteq \lambda_3$ holds.

[2]The *converse* of $P \Rightarrow Q$ is $Q \Rightarrow P$.



$\lambda_4 = \lambda_1 \sqcup \lambda_2$
$\lambda_5 = \lambda_1 \sqcup \lambda_3$
$\lambda_6 = \lambda_2 \sqcup \lambda_3$
$\top_\Lambda = \lambda_1 \sqcup \lambda_2 \sqcup \lambda_3$

$\lambda \longrightarrow \lambda'$ denotes $\lambda \sqsubset \lambda'$

Figure 9.1: Examples of $\sqcup$ for $\Lambda = \{\bot_\Lambda, \lambda_1, \ldots, \lambda_6, \top_\Lambda\}$

Given axiom (9.6), a definition for $\Gamma_{\mathcal{E}}(f(E_1, E_2, \ldots E_n))$ satisfying (9.5) can be constructed by using $\sqcup$

$$\Gamma_{\mathcal{E}}(f(E_1, E_2, \ldots E_n)): \quad \bigsqcup_{1 \le i \le n} \Gamma_{\mathcal{E}}(E_i) \tag{9.8}$$

where we define

$$\bigsqcup_{i \in \mathcal{I}} \lambda_i: \quad \begin{cases} \bot_\Lambda & \text{if } \mathcal{I} = \varnothing \\ \lambda_{i_1} \sqcup \lambda_{i_2} \sqcup \cdots \sqcup \lambda_{i_n} & \text{if } \mathcal{I} = \{i_1, i_2, \ldots, i_n\} \end{cases} \tag{9.9}$$

Combining (9.2), (9.4), and (9.8), we get the following definition for the label $\Gamma_{\mathcal{E}}(E)$ given to an expression E.

$$\Gamma_{\mathcal{E}}(E): \quad \begin{cases} \bot_\Lambda & \text{if } E \text{ is a constant } c \\ \Gamma(v) & \text{if } E \text{ is a variable } v \\ \bigsqcup_{1 \le i \le n} \Gamma_{\mathcal{E}}(E_i) & \text{if } E \text{ is } f(E_1, E_2, \ldots, E_n) \end{cases} \tag{9.10}$$

Useful corollaries of (9.10) include the following, where $Vars(expr)$ is the set of variables referenced in $expr$.

$$\Gamma_{\mathcal{E}}(E) \;=\; \bigsqcup_{v \in Vars(expr)} \Gamma(v) \tag{9.11}$$

$$(\Gamma_{\mathcal{E}}(E) \not\sqsubseteq w) \;\Rightarrow\; (\exists v \in Vars(E):\ v \not\sqsubseteq w) \tag{9.12}$$

9.1.2 Λ_{LH}: No Read Up or Write Down

Set $\Lambda_{\mathsf{LH}} = \{\mathsf{L}, \mathsf{H}\}$ of labels, along with the partial order $\sqsubseteq$ and join $\sqcup$ as defined by Figure 9.2, are often used in discussing information flow policies.

- For specifying confidentiality, variables storing public information are given label L, and variables storing secret information are given label H. Secret values are then prohibited from affecting public values, because $\mathsf{H} \not\sqsubseteq \mathsf{L}$ holds.

$\sqsubseteq$	L	H
L	$\sqsubseteq$	$\sqsubseteq$
H	$\not\sqsubseteq$	$\sqsubseteq$

(a) Definition of $\sqsubseteq$

$\sqcup$	L	H
L	L	H
H	H	H

(b) Definition of $\sqcup$

Figure 9.2: Definitions of $\sqsubseteq$ and $\sqcup$ for $\Lambda_{\mathsf{LH}} = \{\mathsf{L}, \mathsf{H}\}$

- For specifying integrity, variables storing trusted information are given label L and variables storing untrusted information are given label H. Untrusted values are then prohibited from affecting trusted values, because $\mathsf{H} \not\sqsubseteq \mathsf{L}$ holds.

You may find it counterintuitive to be using the same label (H) both for untrusted values and for secret values. Here is way to reconcile these interpretations. According to (9.1), uses of variables with label H should be more restricted than uses of variables with label L. For confidentiality, the added restrictions limit the propagation of secrets. For integrity, the added restrictions limit the propagation of untrusted information.

Some find it helpful to think of an information flow from a variable with label L (Low) to a variable with label H (High) as information flowing "up", and they think of an information flow from a variable with label H to a variable with label L as information flowing "down". Using that metaphor, two consequences of the prohibitions implied by $\mathsf{H} \not\sqsubseteq \mathsf{L}$ are:

- *No read up.* The value obtained by reading a variable with label H cannot be written to a variable with label L.
- *No write down.* The value written to a variable with label L cannot be read from a variable with label H.

So the information flow policy that Figure 9.2 specifies might be succinctly described by "no read up; no write down".

9.1.3 Noninterference Policies

Noninterference policies are a class of information flow policies that use counterfactual arguments[3] for defining whether one variable affects another. The counterfactual argument defines "v affects w" to hold in a program S if executing S with different initial values for v can cause the value of w to differ.

Noninterference policies are intended to prevent a so-called λ-*observer* (for any $\lambda \in \Lambda$) from learning about the initial values of variables in $V_{\not\sqsubseteq\lambda}$ by reading variables in $V_{\sqsubseteq\lambda}$. The capabilities of different threats are modeled by making different assumptions about when λ-observers can access the variables in $V_{\sqsubseteq\lambda}$. Some noninterference policies assume that λ-observers only have access to the

[3] With a *counterfactual argument*, multiple hypothetical starting points or sets of assumptions are the basis for justiying the conclusion.

initial and final states of an execution; other noninterference policies assume that λ-observers can access intermediate states, too. And some noninterference policies assume that λ-observers are also capable of detecting that an execution is non-terminating or that an execution have been blocked by an enforcement mechanism.

9.2 Termination Insensitive Noninterference

Termination Insensitive Noninterference (TINI) policies prohibit the values of variables from $V_{\not\sqsubseteq\lambda}$ in initial states from affecting the values of variables from $V_{\sqsubseteq\lambda}$ in final states of terminating executions, for all labels $\lambda \in \Lambda$. So if TINI is being enforced then the initial and final values of variables from $V_{\sqsubseteq\lambda}$ in terminating executions reveal nothing about the initial values of variables from $V_{\not\sqsubseteq\lambda}$.[4] TINI policies are intended for settings that satisfy the following.

- *Batch.* A λ-observer can read variables in $V_{\sqsubseteq\lambda}$ before and after, but not during, execution of a program.
- *Asynchronous.* A λ-observer cannot distinguish a non-terminating execution from a terminating execution that has not yet terminated.

TINI policies can be formally defined by using a predicate $\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ti}} \mathcal{W}$ that holds if, in terminating executions by S, the initial values of variables in the set $\mathcal{V}$ do not affect the final values of variables in the set $\mathcal{W}$.

> **Termination Insensitive Noninterference (TINI).** For a deterministic program S with a set V of variables, having labels from a set Λ with partial order $\sqsubseteq$:
>
> $$(\forall \lambda \in \Lambda\colon\ V_{\not\sqsubseteq\lambda} \overset{S}{\not\rightarrow}_{\mathsf{ti}} V_{\sqsubseteq\lambda}) \qquad \square$$

The formal definition for $\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ti}} \mathcal{W}$ uses a function $[\![S]\!](s)$ that characterizes the result produced by executing S from a state s.

$$[\![S]\!](s)\colon \begin{cases} s' & \text{if execution of } S \text{ in state } s \text{ terminates in state } s' \\ \Uparrow & \text{if execution of } S \text{ in state } s \text{ is non-terminating} \end{cases} \tag{9.13}$$

The formal definition of $\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ti}} \mathcal{W}$ also uses a predicate that is satisfied when two states give the same values to variables in some set $\mathcal{V}$. For a state s, we

[4] TINI policies thus ignore leaks that become possible if an observer can reliably predict that some execution will be nonterminating. For example, determining that an execution of

```
while x = 0 do skip end
```

will be nonterminating implies that $x = 0$ was *true* in the initial state. *Termination sensitive noninteference* (TSNI) policies strengthen TINI policies to account for observers that can detect if an execution is nonterminating.

write $s.v$ to denote the value of a variable v in a state s, and we define the value of a variable v in *state projection* $s|_{\mathcal{V}}$ as follows.

$$s|_{\mathcal{V}}.v\text{:}\quad \begin{cases} s.v & \text{if } v \in \mathcal{V} \\ ? & \text{otherwise .} \end{cases} \tag{9.14}$$

State projections provide a straightforward way to define predicates that asserts two states give the same values to the variables in $\mathcal{V}$ or to the variables in complement $\overline{\mathcal{V}}$ comprising the variables in $Vars(S) - \mathcal{V}$.

$$s =_{\mathcal{V}} s'\text{:}\quad s|_{\mathcal{V}} = s'|_{\mathcal{V}} \qquad\qquad s =_{\overline{\mathcal{V}}} s'\text{:}\quad s|_{\overline{\mathcal{V}}} = s'|_{\overline{\mathcal{V}}}$$

We then have the following formal definition for predicate $\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ti}} \mathcal{W}$, where $Init_S$ denotes the set of initial states of a program S.

$$\begin{aligned} \mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ti}} \mathcal{W}\text{:}\quad & (\forall s, s' \in Init_S\text{:}\ s =_{\overline{\mathcal{V}}} s' \wedge [\![S]\!](s) \neq \Uparrow \wedge [\![S]\!](s') \neq \Uparrow \\ & \qquad \Rightarrow [\![S]\!](s) =_{\mathcal{W}} [\![S]\!](s')) \end{aligned} \tag{9.15}$$

So predicate $\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ti}} \mathcal{W}$ specifies a requirement on the states produced by terminating executions started in initial states s and s', where s and s' can give different values to one or more variables in $\mathcal{V}$ but give the same values to variables not in $\mathcal{V}$. That requirement is that final states $[\![S]\!](s)$ and $[\![S]\!](s')$ must satisfy $[\![S]\!](s) =_{\mathcal{W}} [\![S]\!](s')$, which implies that the final values of variables in $\mathcal{W}$ have not been affected by any differences in starting states s and s'. Since initial states s and s' differ only in the values for variables in $\mathcal{V}$, a counterfactual argument has been made to establish that the different values for variables in $\mathcal{V}$ did not affect the values of variables in $\mathcal{W}$.

9.2.1 TINI in Action

Consider a TINI policy specified by set Λ_{LH} of labels with partial order $\sqsubseteq$ of Figure 9.2. For a deterministic program S with variables V, this information flow policy specifies the following, derived from the above formalization of TINI by replacing λ in $V_{\not\sqsubseteq\lambda} \overset{S}{\not\rightarrow}_{\mathsf{ti}} V_{\sqsubseteq\lambda}$ with the possible values: L and H.

$$V_{\not\sqsubseteq\mathsf{L}} \overset{S}{\not\rightarrow}_{\mathsf{ti}} V_{\sqsubseteq\mathsf{L}} \quad\wedge\quad V_{\not\sqsubseteq\mathsf{H}} \overset{S}{\not\rightarrow}_{\mathsf{ti}} V_{\sqsubseteq\mathsf{H}} \tag{9.16}$$

By expanding $\overset{S}{\not\rightarrow}_{\mathsf{ti}}$ according to definition (9.15), we obtain the following restrictions on the initial and final states of the terminating executions by S:

$$\begin{aligned} & (\forall s, s' \in Init_S\text{:}\ \ s =_{\overline{V}_{\not\sqsubseteq\mathsf{L}}} s' \ \wedge\ [\![S]\!](s) \neq \Uparrow \ \wedge\ [\![S]\!](s') \neq \Uparrow \\ & \qquad\qquad \Rightarrow\ [\![S]\!](s) =_{V_{\sqsubseteq\mathsf{L}}} [\![S]\!](s')) \\ \wedge\ & (\forall s, s' \in Init_S\text{:}\ \ s =_{\overline{V}_{\not\sqsubseteq\mathsf{H}}} s' \ \wedge\ [\![S]\!](s) \neq \Uparrow \ \wedge\ ;[\![S]\!](s') \neq \Uparrow \\ & \qquad\qquad \Rightarrow\ [\![S]\!](s) =_{V_{\sqsubseteq\mathsf{H}}} [\![S]\!](s')) \end{aligned} \tag{9.17}$$

$\Gamma(in)$	$\Gamma(out)$	$V_{\sqsubseteq \mathsf{L}}$	$V_{\not\sqsubseteq \mathsf{L}}$	$V_{\sqsubseteq \mathsf{H}}$	$V_{\not\sqsubseteq \mathsf{H}}$	$out := in?$
L	L	$\{in, out\}$	$\varnothing$	$\{in, out\}$	$\varnothing$	$\surd$
L	H	$\{in\}$	$\{out\}$	$\{in, out\}$	$\{\varnothing\}$	$\surd$
H	L	$\{out\}$	$\{in\}$	$\{in, out\}$	$\{\varnothing\}$	$\times$
H	H	$\varnothing$	$\{in, out\}$	$\{in, out\}$	$\varnothing$	$\surd$

Figure 9.3: Possible information flow policies for $out := in$

Because the following hold

$$\overline{V}_{\not\sqsubseteq \mathsf{L}} = V_{\mathsf{L}} \qquad V_{\sqsubseteq \mathsf{L}} = V_{\mathsf{L}} \qquad \overline{V}_{\not\sqsubseteq \mathsf{H}} = V_{\sqsubseteq \mathsf{H}} \qquad V_{\sqsubseteq \mathsf{H}} = V_{\mathsf{L}} \cup V_{\mathsf{H}}$$

(9.17) is equivalent to:

$$\begin{array}{l} (\forall s, s' \in \mathit{Init}_S:\ \ s =_{V_{\mathsf{L}}} s' \ \wedge\ [\![S]\!](s) \neq \Uparrow \ \wedge\ [\![S]\!](s') \neq \Uparrow \\ \qquad\qquad \Rightarrow\ [\![S]\!](s) =_{V_{\mathsf{L}}} [\![S]\!](s')) \\ \wedge\ \ (\forall s, s' \in \mathit{Init}_S:\ \ s =_{V_{\mathsf{L}} \cup V_{\mathsf{H}}} s' \ \wedge\ [\![S]\!](s) \neq \Uparrow \ \wedge\ [\![S]\!](s') \neq \Uparrow \\ \qquad\qquad \Rightarrow\ [\![S]\!](s) =_{V_{\mathsf{L}} \cup V_{\mathsf{H}}} [\![S]\!](s')) \end{array} \tag{9.18}$$

We have that $V_{\mathsf{L}} \cup V_{\mathsf{H}} = V$ holds, since every variable is assigned a label from Λ_{LH}. Therefore, predicate $s =_{V_{\mathsf{L}} \cup V_{\mathsf{H}}} s'$ in (9.18) is equivalent to predicate $s = s'$. So the second quantified formula of (9.18) is satisfied due to the assumption that S is deterministic—terminating executions of deterministic program that start from the same states always produce the same states. The second quantified formula of (9.18) is thus equivalent to *true*, and we conclude that TINI policy (9.18) is specifying:

$$\begin{array}{l} (\forall s, s' \in \mathit{Init}_S:\ \ (s =_{V_{\mathsf{L}}} s' \ \wedge\ [\![S]\!](s) \neq \Uparrow \ \wedge\ [\![S]\!](s') \neq \Uparrow\ ) \\ \qquad\qquad \Rightarrow\ \ [\![S]\!](s) =_{V_{\mathsf{L}}} [\![S]\!](s')) \end{array}$$

States satisfying $s =_{V_{\mathsf{L}}} s'$ may differ in the values of variables in V_{H} but must agree on the values of variables in V_{L}. That means (9.18) implies that the values of variables in V_{H} in initial states may not affect on the values of variables in V_{L} in final states or, equivalently, that the values of variables with label H are prohibited from affecting the values of variables with label L. So if this TINI policy is enforced, then variables with label H can store information that we do not want leaked to variables with label L.

To illustrate further, consider the simple program: $out := in$. This program is deterministic, it always terminates, and the value of in in initial states affects the value of out in final states. The final column in Figure 9.3 summarizes whether program $out := in$ satisfies the TINI policy that results from the labeling that each row gives for variables in and out. There is a $\surd$ in the final column if the TINI policy defined by the row is satisfied by execution of $out := in$. The third row has an $\times$ in the final column because that TINI policy is violated by execution of $out := in$. This violation should not be surprising—TINI prohibits

executions where a variable having label H affects a variable having label L, and here *in* has label H but *out* has label L.

Some implications of various TINI policies might be surprising. Consider variables x_{L} and x_{H}, with $\Gamma(x_{\mathsf{L}}) = \mathsf{L}$ and $\Gamma(x_{\mathsf{H}}) = \mathsf{H}$. The following program shows that a TINI policy can be violated by assignment statements where the expressions are constants.

$$\texttt{if } x_{\mathsf{H}} = 0 \texttt{ then } x_{\mathsf{L}} := 1 \texttt{ else } x_{\mathsf{L}} := 2 \texttt{ fi} \tag{9.19}$$

The next program slightly changes the `else` alternative.

$$\texttt{if } x_{\mathsf{H}} = 0 \texttt{ then } x_{\mathsf{L}} := 1 \texttt{ else } x_{\mathsf{L}} := 1 \texttt{ fi} \tag{9.20}$$

The TINI policy is not violated, because the same assignment to x_{L} is executed for any value of x_{H}.

Two final programs illustrate that TINI policies are not necessarily violated if assignment statements store into variables labeled L from variables labeled H. In this program

$$x_{\mathsf{L}} := x_{\mathsf{H}};\ x_{\mathsf{L}} := 63; \tag{9.21}$$

the TINI policy is satisfied, since the final value of x_{L} is not affected by the value of x_{H} (although an intermediate value is).

This last program satisfies the TINI policy if B does not mention x_{L} or x_{H}, even though a variable with label H affects a variable with label L in the body of the `while`.

$$\texttt{while } B \texttt{ do } x_{\mathsf{L}} := x_{\mathsf{H}} \texttt{ end} \tag{9.22}$$

If B is initially *true* then B will remain *true* (because the only variable changed in the loop body is not mentioned in B), so the `while` never terminates. The TINI policy is satisfied because TINI policies impose no restrictions on non-terminating executions. If B is initially *false*, then the TINI policy is satisfied because the loop body is never executed, so problematic assignment statement $x_{\mathsf{L}} := x_{\mathsf{H}}$ is never executed.

9.2.2 TINI Enforcement

To be concrete in our discussions about enforcement of noninterference policies, Figure 9.4 gives the grammer for IMP, a simple imperative programming language. Instead of including variable declarations, an IMP program will be accompanied by a function $\Gamma(\cdot)$ giving a label for each variable. Expressions *expr* in IMP programs are constructed from constants, variables, operators, and functions, as discussed in §9.1.1. Finally, we write "ℓ_i: S" to indicate that a *statement label* ℓ_i names the control point associated with the start of statement S. Statement labels also provide a way to refer to the statement at that control point. No statement label will appear more than once in a program, and statement labels are disjoint from the labels in Λ.

$$
\begin{array}{rcl}
\mathit{stmt} & ::= & \texttt{skip} \\
& & \mid \mathit{var} := \mathit{expr} \\
& & \mid \texttt{if } \mathit{expr} \texttt{ then } \mathit{stmt} \texttt{ else } \mathit{stmt} \texttt{ fi} \\
& & \mid \texttt{while } \mathit{expr} \texttt{ do } \mathit{stmt} \texttt{ end} \\
& & \mid \mathit{stmt};\ \mathit{stmt}
\end{array}
$$

Figure 9.4: Syntax for IMP programs

$$
\begin{array}{l}
\ell_1\colon S_1 \\
\ell_2\colon \texttt{if } G_2 \texttt{ then } \ell_3\colon S_3 \\
\qquad\qquad \texttt{else } \ell_4\colon \texttt{while } G_4 \texttt{ do} \\
\qquad\qquad\qquad\qquad \ell_5\colon S_5 \texttt{ end;} \\
\qquad\qquad\quad \ell_6\colon S_6 \\
\texttt{fi;} \\
\ell_7\colon S_7
\end{array}
$$

ℓ_i	$\Theta_S(\ell_i)$
ℓ_1	$\varnothing$
ℓ_2	$\varnothing$
ℓ_3	$\{G_2\}$
ℓ_4	$\{G_2\}$
ℓ_5	$\{G_2, G_4\}$
ℓ_6	$\{G_2\}$
ℓ_7	$\varnothing$

Figure 9.5: $\Theta_S(\ell_i)$ for a program S

Assignment statements $\mathit{var} := \mathit{expr}$ are the way an IMP program changes the value of a variable; *var* is called the *target*, and *expr* is called the *source*. IMP provides two kinds of *control-flow statements*: `if` statements and `while` statements. Each control-flow statement has a *guard* and a *body*. The guard is a Boolean expression; the body comprises one or more statements. With an `if` statement, the body has a `then` alternative and an `else` alternative; the value of the guard determines which alternative is executed. With a `while` statement, the value of the guard determines whether the body is executed for another iteration or, instead, execution of the `while` statement terminates.

For each statement label ℓ in a program S, there is a set $\Theta_S(\ell)$ that contains the guard for those control-flow statements having a body that includes statement ℓ. So each guard in $\Theta_S(\ell)$ can affect whether statement ℓ will be reached during execution of S. Figure 9.5 gives $\Theta_S(\cdot)$ for an example program. Notice, $\Theta_S(\ell)$ contains multiple guards when ℓ is nested within multiple control-flow statements. Guards in $\Theta_S(\ell)$, however, are not the only guards that can affect whether statement ℓ will be reached during an execution of a program S. Figure 9.5 gives an example. We have that $G_4 \notin \Theta_S(\ell_7)$ holds even though G_4 could affect whether ℓ_7 will be reached—G_4 affects whether `while` statement ℓ_4 terminates, and if that `while` statement does not terminate then ℓ_7 will not be reached.

An execution of S that violates TINI must terminate and must execute some assignment statement. There are two ways that executing an assignment statement $\ell\colon w := \mathit{expr}$ could violate TINI. With an *explicit flow*, the violation

is caused by some variable in *expr*. Explicit flows are prevented if the following holds.

$$\Gamma_{\mathcal{E}}(\mathit{expr}) \sqsubseteq \Gamma(w) \tag{9.23}$$

With an *implicit flow*, the violation is caused by some guard G that affects whether the assignment is executed and that does not satisfy $\Gamma_{\mathcal{E}}(G) \sqsubseteq \Gamma(w)$. So implicit flows are prevented if none of those guards exists, because the following holds.

$$\left(\bigsqcup_{G \in \Theta_S(\ell)} \Gamma_{\mathcal{E}}(G) \right) \;\sqsubseteq\; \Gamma(w). \tag{9.24}$$

Therefore, the following check establishes that executing an assignment statement does not cause explicit or implicit flows.

> **Safe Assignment Statements.** For each assignment statement ℓ: $w := \mathit{expr}$ that S executes, check that the following holds.
>
> $$\left(\Gamma_{\mathcal{E}}(\mathit{expr}) \;\sqcup\; \left(\bigsqcup_{G \in \Theta_S(\ell)} \Gamma_{\mathcal{E}}(G) \right) \right) \;\sqsubseteq\; \Gamma(w) \qquad \square$$

Safe Assignment Statements is *conservative*—a program that complies will satisfy TINI, but a program that does not comply might nevertheless satisfy TINI. One reason a program could be wrongly rejected is that definition (9.10) for $\Gamma_{\mathcal{E}}(\cdot)$ ignores the semantics of expressions. For example, if variables v and w satisfy $\Gamma(v) \not\sqsubseteq \Gamma(w)$ then the program $w := v - v$ does not satisfy Safe Assignment Statements because $\Gamma_{\mathcal{E}}(v - v) = \Gamma(v)$ and, therefore, $\Gamma_{\mathcal{E}}(v - v) \sqsubseteq \Gamma(w)$ does not hold. However, program $w := v - v$ does satisfy TINI, since the final value of w is the same for all initial values of v.

A second reason programs are wrongly rejected is that Safe Assignment Statements ignores context. Program (9.20) on page 251 is rejected due to `if` statement guard $x_{\mathsf{H}} = 0$. Yet this program satisfies TINI because the `then` and the `else` alternatives each store the same constant into x_{L} for any initial value of x_{H}. A different effect of context is seen in program (9.21), where an assignment statement $x_{\mathsf{L}} := x_{\mathsf{H}}$ that does not satisfy Safe Assignment Statements is followed by an assignment statement $x_{\mathsf{L}} := 63$ that overwrites the problematic update.

9.2.3 Enforcing TINI with Types

A type-safe programming language will have a set of *typing rules* for deriving *type-correct* programs. The typing rules ensure that all executions of type-correct programs are guaranteed to be well behaved. You are doubtless familiar with typing rules to ensure that only certain kinds of values are stored in each program variable and are arguments to only certain operations. Such typing rules might, for example, prohibit performing arithmetic operations on variables

storing character strings. In this section, we give typing rules that ensure type-correct programs satisfy TINI.

To assert that a program or statement S is type-correct, we will use a *judgement*

$$\Gamma, \gamma \vdash_{\mathsf{ti}} S \tag{9.25}$$

where *typing context* Γ is a function that gives a label $\Gamma(v) \in \Lambda$ to each variable $v \in \mathit{Vars}(S)$, and *control context* γ is a label from Λ.[5] Judgements that satisfy certain constraints are defined to be *valid for TINI*.

> **Valid Judgements for TINI.** A judgement $\Gamma, \gamma \vdash_{\mathsf{ti}} S$ for a program S with variables V is *valid for TINI* if and only if
>
> (i) S satisfies TINI: $(\forall \lambda \in \Lambda\colon\ V_{\not\sqsubseteq \lambda} \overset{S}{\not\rightarrow}_{\mathsf{ti}} V_{\sqsubseteq \lambda})$
>
> (ii) Target w of every assignment statement in S satisfies: $\gamma \sqsubseteq \Gamma(w)$. □

Requirement (i) should not be surprising, given our goal of having type-safe programs comply with TINI. Requirement (ii) might be. It enables valid judgements for statement compositions to be derived from valid judgements for the component statements, as follows. Consider a statement $\ell_S\colon S$ that is part of a larger program T, and suppose $\Gamma, \gamma \vdash_{\mathsf{ti}} S$ is valid. For $\Gamma, \gamma \vdash_{\mathsf{ti}} S$ to be valid for TINI, requirement (i) implies that TINI is satisfied by $\ell_S\colon S$ and, therefore, no assignment statement $w := \mathit{expr}$ in $\ell_S\colon S$ violates Safe Assignment Statements. If, in addition,

$$\left(\bigsqcup_{G \in \Theta_T(\ell)} \Gamma_{\mathcal{E}}(G) \right) \ \sqsubseteq\ \gamma \tag{9.26}$$

has been proved then requirement (ii) implies that $\Gamma_{\mathcal{E}}(G) \sqsubseteq \Gamma(w)$ holds for every guard G in $\Theta_T(\ell)$. That means TINI cannot be violated by implicit flows from guards in T that affect whether $\ell_S\colon S$ is executed. So if we have proved (9.26) then Safe Assignment Statements for program $\ell_S\colon S$ implies that Safe Assignment Statements for program T also holds for all of the assignment statements in S. As we shall see, the typing rules require that appropriate instances of (9.26) be proved when statements are composed.

Typing Rules. Each typing rule R is specified with a schema:

$$\text{R:}\ \frac{H_1,\ H_2,\ \ldots,\ H_n}{\Gamma, \gamma \vdash_{\mathsf{ti}} S}$$

The schema serves as a template for deriving the rule's *conclusion* $\Gamma, \gamma \vdash_{\mathsf{ti}} S$ by mechanically transforming some or all of the rule's *hypotheses* $H_1, H_2, \ldots, H_n$. By design, the conclusion of a typing rule will be a judgement that is valid for TINI if each of the rule's hypotheses is valid.

[5] Consistent with the IMP syntax given in Figure 9.4, we use "statement" and "program" interchangeably in the following discussions.

$$\text{SKIP:}\ \frac{}{\Gamma, \gamma \vdash_{\mathsf{ti}} \texttt{skip}} \qquad \text{ASSIGN:}\ \frac{\gamma \sqcup \Gamma_{\mathcal{E}}(expr) \sqsubseteq \Gamma(w)}{\Gamma, \gamma \vdash_{\mathsf{ti}} w := expr}$$

$$\text{IF:}\ \frac{\Gamma_E(expr) = \lambda, \quad \Gamma, \gamma \sqcup \lambda \vdash_{\mathsf{ti}} S, \quad \Gamma, \gamma \sqcup \lambda \vdash_{\mathsf{ti}} S'}{\Gamma, \gamma \vdash_{\mathsf{ti}} \texttt{if } expr \texttt{ then } S \texttt{ else } S' \texttt{ fi}}$$

$$\text{WHILE:}\ \frac{\Gamma_E(expr) = \lambda, \quad \Gamma, \gamma \sqcup \lambda \vdash_{\mathsf{ti}} S}{\Gamma, \gamma \vdash_{\mathsf{ti}} \texttt{while } expr \texttt{ do } S \texttt{ end}} \qquad \text{SEQ:}\ \frac{\Gamma, \gamma \vdash_{\mathsf{ti}} S, \quad \Gamma, \gamma \vdash_{\mathsf{ti}} S'}{\Gamma, \gamma \vdash_{\mathsf{ti}} S;\ S'}$$

Figure 9.6: Typing rules for TINI compliance

Figure 9.6 gives the typing rules. An IMP program S is considered type-correct if judgement $\Gamma, \bot_\Lambda \vdash S$ can be derived using these typing rules, because having $\Gamma, \bot_\Lambda \vdash S$ be valid for TINI implies that S satisfies TINI. So, the typing rules give a way to enforce TINI for a program S: use the typing rules to check whether S is type-correct. IMP programs that are type-correct are allowed to execute; other programs are not allowed to execute.

The typing rules work by ensuring that no assignment statement violates Safe Assignment Statements. Looking at an example is a good way to see how this is checked by the rules. Consider the following possible conclusion of rule IF, where S' denotes some statement.

$$\Gamma, \bot_\Lambda \ \vdash_{\mathsf{ti}} \ S\colon \texttt{if } B \texttt{ then } \ell\colon w := expr \texttt{ else } S' \texttt{ fi} \tag{9.27}$$

To derive this judgement requires having a derivation for each hypothesis of rule IF. The second hypothesis requires a derivation of the following.

$$\Gamma,\ \bot_\Lambda \sqcup \Gamma_E(B) \ \vdash_{\mathsf{ti}} \ \ell\colon w := expr$$

Rule ASSIGN must be used to derive this judgement, and the required hypothesis for that derivation is satisfied provided the following holds

$$\bot_\Lambda \sqcup \Gamma_{\mathcal{E}}(B) \sqcup \Gamma_{\mathcal{E}}(expr) \sqsubseteq \Gamma(w),$$

which is equivalent to $\Gamma_{\mathcal{E}}(B) \sqcup \Gamma_{\mathcal{E}}(expr) \sqsubseteq \Gamma(w)$. For program S, $\Theta_S(\ell)$ is $\{B\}$ and, therefore, the following holds.

$$\Gamma_{\mathcal{E}}(B) \ = \ \bigsqcup_{G \in \Theta_S(\ell)} \Gamma_{\mathcal{E}}(G)$$

So we have showed that the derivation of (9.27) requires:

$$\left(\bigsqcup_{G \in \Theta_S(\ell)} \Gamma_{\mathcal{E}}(G) \right) \sqcup \Gamma_{\mathcal{E}}(expr) \ \sqsubseteq \ \Gamma(w)$$

This is what Safe Assignment Statements requires for $\ell\colon w := expr$, since whether ℓ executes is affected by guard B of the `if` statement (and by no other guards).

1. $\Gamma_{\mathcal{E}}(0) = \mathsf{L}$ — definition (9.10) of $\Gamma_{\mathcal{E}}(\cdot)$, since $\bot_{\{\mathsf{L},\mathsf{H}\}} = \mathsf{L}$.
2. $\Gamma(m) = \mathsf{H}$ — assumption.
3. $((\mathsf{L} \sqcup \mathsf{H}) \sqcup \mathsf{L}) \sqsubseteq \mathsf{H}$ — definitions of $\sqcup$ and $\sqsubseteq$ in Figure 9.2.
4. $\Gamma, \mathsf{L} \sqcup \mathsf{H} \vdash_{\mathsf{ti}} m := 0$ — ASSIGN with 1, 2, 3.
5. $\Gamma_{\mathcal{E}}(y) = \mathsf{H}$ — definition (9.10) of $\Gamma_{\mathcal{E}}(\cdot)$, since $\Gamma(y) = \mathsf{H}$ by assumption.
6. $((\mathsf{L} \sqcup \mathsf{H}) \sqcup \mathsf{H}) \sqsubseteq \mathsf{H}$ — definitions of $\sqcup$ and $\sqsubseteq$ in Figure 9.2.
7. $\Gamma, \mathsf{L} \sqcup \mathsf{H} \vdash_{\mathsf{ti}} m := y$ — ASSIGN with 5, 2, 6.
8. $\Gamma_{\mathcal{E}}(x \le y) = \mathsf{H}$ — definition (9.10) of $\Gamma_{\mathcal{E}}(\cdot)$, since $\Gamma_{\mathcal{E}}(x \le y) = (\Gamma(x) \sqcup \Gamma(y)) = (\mathsf{L} \sqcup \mathsf{H}) = \mathsf{H}$.
9. $\Gamma, \mathsf{L} \vdash_{\mathsf{ti}} \texttt{if } x \le y \texttt{ then } m := 0 \texttt{ else } m := y \texttt{ fi}$ — IF with 8, 4, 7.

Figure 9.7: Example of Hilbert-style proof format

Proof Formats. Various formats can be used for presenting derivations of judgements. Each format has advantages and disadvantages. To illustrate the different formats, we use each to give the type-correctness derivation for the following judgement

$$\Gamma, \mathsf{L} \vdash_{\mathsf{ti}} \texttt{if } x \le y \texttt{ then } m := 0 \texttt{ else } m := y \texttt{ fi} \tag{9.28}$$

assuming $\Gamma(x) = \mathsf{L}$, $\Gamma(y) = \mathsf{H}$, and $\Gamma(m) = \mathsf{H}$ hold, Λ is $\{\mathsf{L}, \mathsf{H}\}$, and the rules for evaluating expressions involving $\sqsubseteq$ and $\sqcup$ are those given in Figure 9.2.

Hilbert-Style Proof Format. Figure 9.7 gives a type-correctness derivation that is formatted as a list of sequentially numbered *steps*. Each step comprises a formula F (often, a judgement) and a justification J. When F is a judgement, J names a typing rule R and lists the numbers for earlier steps that discharge hypotheses needed to derive F by using R. Sometimes the validity of a hypothesis is given as part of the justification rather than by referencing an earlier step. Such inline justifications are used by steps 1, 2, 3, 5, and 8 of Figure 9.7.

Derivation Tree Proof Format. Figure 9.8 gives the type-correctness derivation as a series of derivation trees. A *derivation tree* vertically stacks instances of typing rules, positioning the conclusion of one rule to appear as a hypothesis for another rule. Three derivation trees appear in Figure 9.8. Tags (DT1 and DT2) on the first two derivation trees allow their conclusions to be used for discharging hypotheses in the third derivation tree. Many people prefer reading derivation trees over reading the lists of steps in the Hilbert-style proof format, because derivation trees graphically show dependencies between steps. Derivation trees are a natural format when working with pen and paper, but few text formatters facilitate their construction.

Hierarchical Proof Format. A combination of Hilbert-style proofs and derivation trees is to present a list of judgements, but do so hierarchically. This format is illustrated in Figure 9.9. Here, justifications for the hypotheses needed to infer the judgement of step n are listed after that step, indented, and numbered

$$\text{ASSIGN:}\ \frac{((\mathsf{L} \sqcup \mathsf{H}) \sqcup \Gamma_{\mathcal{E}}(0)) \sqsubseteq \mathsf{H}}{\Gamma, \mathsf{L} \sqcup \mathsf{H} \vdash_{\mathsf{ti}} m := 0} \tag{DT1}$$

$$\text{ASSIGN:}\ \frac{((\mathsf{L} \sqcup \mathsf{H}) \sqcup \Gamma_{\mathcal{E}}(y)) \sqsubseteq \mathsf{H}}{\Gamma, \mathsf{L} \sqcup \mathsf{H} \vdash_{\mathsf{ti}} m := y} \tag{DT2}$$

$$\text{IF:}\ \frac{\Gamma_{\mathcal{E}}(x \le y) = \mathsf{H}, \quad \text{DT1:}\ \dfrac{\ldots}{\Gamma, \mathsf{L} \sqcup \mathsf{H} \vdash_{\mathsf{ti}} m := 0}, \quad \text{DT2:}\ \dfrac{\ldots}{\Gamma, \mathsf{L} \sqcup \mathsf{H} \vdash_{\mathsf{ti}} m := y}}{\Gamma, \mathsf{L} \vdash_{\mathsf{ti}} \texttt{if}\ x \le y\ \texttt{then}\ m := 0\ \texttt{else}\ m := y\ \texttt{fi}}$$

Figure 9.8: Example of derivation tree proof format

1. $\Gamma, \mathsf{L} \vdash_{\mathsf{ti}} \texttt{if}\ x \le y\ \texttt{then}\ m := x\ \texttt{else}\ m := y\ \texttt{fi}$ IF with 1.1, 1.2, and 1.3.
   - 1.1. $\Gamma_{\mathcal{E}}(x \le y) = \mathsf{H}$ — $\Gamma_{\mathcal{E}}(x \le y) = (\Gamma(x) \sqcup \Gamma(y)) = (\mathsf{L} \sqcup \mathsf{H}) = \mathsf{H}$.
   - 1.2. $\Gamma, \mathsf{L} \sqcup \mathsf{H} \vdash_{\mathsf{ti}} m := 0$ — ASSIGN with 1.2.1 and 1.2.2.
     - 1.2.1. $\Gamma_{\mathcal{E}}(0) = \mathsf{L}$ — Definition (9.10) of $\Gamma_{\mathcal{E}}(\cdot)$, since $\bot_{\{\mathsf{L},\mathsf{H}\}} = \mathsf{L}$.
     - 1.2.2. $\Gamma(m) = \mathsf{H}$ — Assumption.
     - 1.2.3. $((\mathsf{L} \sqcup \mathsf{H}) \sqcup \mathsf{L}) \sqsubseteq \mathsf{H}$ — Definitions of $\sqcup$ and $\sqsubseteq$ in Figure 9.2.
   - 1.3. $\Gamma, \mathsf{L} \sqcup \mathsf{H} \vdash_{\mathsf{ti}} m := y$ — ASSIGN with 1.3.1, 1.3.2, and 1.3.3.
     - 1.3.1 $\Gamma_{\mathcal{E}}(y) = \mathsf{H}$ — Assumption.
     - 1.3.2. $\Gamma(m) = \mathsf{H}$ — Assumption.
     - 1.3.3. $((\mathsf{L} \sqcup \mathsf{H}) \sqcup \mathsf{H}) \sqsubseteq \mathsf{H}$ — Definitions of $\sqcup$ and $\sqsubseteq$ in Figure 9.2.

Figure 9.9: Example of hierarchically presented proof format

by appending sequence numbers to n to get $n.1$, $n.2$, etc. Arbitrary levels of nesting are permitted. With this format, indentation helps readers to see the steps that support a conclusion, but without the distraction of how each of those steps is being justified. The format also enables the reader to focus on the reasoning used to support the justification for any given step.

9.2.4 Dynamic Enforcement of TINI

For *dynamic enforcement* of TINI, we employ a reference monitor.[6] When an execution is about to perform an action that would violate TINI, the reference monitor blocks further progress and deletes the program state. Given the Batch and Asynchronous assumptions (see page 248), blocked executions are indistinguishable to λ-observers from nonterminating executions. Since TINI imposes no constraints on nonterminating executions, TINI also imposes no constraints on blocked executions. Provided all other terminating executions do not violate TINI then the enforcement we seek is achieved.

[6] See Chapter 11 for a detailed treatment of reference monitors.

The only way for a terminating execution of an IMP program S to violate TINI is by executing an assignment statement that does not satisfy Safe Assignment Statements (page 253). Therefore, the reference monitor checks Safe Assignment Statements whenever an assignment statement is about to execute. To perform such a check for an assignment statement ℓ: $w := expr$ in some program S, the reference monitor needs labels $\Gamma(w)$, $\Gamma_{\mathcal{E}}(expr)$, and $\bigsqcup_{G \in \Theta_S(\ell)} \Gamma_{\mathcal{E}}(G)$. These labels can be derived using $\Gamma(\cdot)$ as follows.

- $\Gamma(w)$ and $\Gamma_{\mathcal{E}}(expr)$ can be calculated by the reference monitor if (i) reaching each assignment statement ℓ: $w := expr$ is an event that causes the reference monitor to be invoked, and (ii) the name of target w and names of variables referenced in $expr$ are delivered to the reference monitor as part of that event.

- $\bigsqcup_{G \in \Theta_S(\ell)} \Gamma_{\mathcal{E}}(G)$ can be calculated by the reference monitor if (i) reaching and exiting `if` or `while` statements are events that cause the reference monitor to be invoked and (ii) the code for each of these events (`if` G, `fi`, `while` G, or `end`) is delivered to the reference monitor with that event.

Figure 9.10 gives the code for such a reference monitor, $\mathcal{R}_{TI}$. That code uses `require`(B) statements, where B is restricted to being a Boolean expression involving the labels of variables and expressions. Execution of `require`(B) evaluates B. If B evaluates to *false* then the reference monitor deletes the state and blocks further progress of the program that was being executed when the reference monitor was invoked; if B evaluates to *true* then execution is allowed to proceed.

The code for reference monitor $\mathcal{R}_{TI}$ is invoked and checks Safe Assignment Statements whenever an assignment statement is reached in the program S being monitored. To facilitate this checking, the reference monitor also is invoked so that it can update a stack[7] $sl[S]$ whenever S reaches or exits a control-flow statement. We are assuming that stack $sl[S]$ is allocated and initialized to empty when S starts execution. The updates to $sl[S]$ described in Figure 9.10 ensure that

$$sl[S].\texttt{top} = \bigsqcup_{G \in \Theta_S(\ell)} \Gamma_{\mathcal{E}}(G) \tag{9.29}$$

holds whenever an assignment statement (say) ℓ: $w := expr$ is about to execute. Therefore, the value of $sl[S].\texttt{top}$, along with $\Gamma(w)$ and $\Gamma_{\mathcal{E}}(expr)$, can be used by the reference monitor when checking Safe Assignment Statements for that assignment statement.

What $\mathcal{R}_{TI}$ Enforces. Consider a program S executing with $\mathcal{R}_{TI}$ present, resulting in the combined program that we represent using the notation $\mathcal{R}_{TI} \triangleright S$.

[7] For a stack st, we use operations $st.\texttt{push}(v)$ to insert value v, $st.\texttt{pop}()$ to remove the most recently added value, and a function $st.\texttt{top}()$ that returns the most recently added value.

when monitored program S reaches ℓ	reference monitor code for action that is performed
ℓ: $w := expr$	`require`$(sl[S].\texttt{top} \sqcup \Gamma_{\mathcal{E}}(expr) \sqsubseteq \Gamma(w))$
ℓ: `if` G `then` ...	$sl[S].\texttt{push}(\ sl[S].\texttt{top} \sqcup \Gamma_{\mathcal{E}}(G)\ )$
... `fi`; ℓ: ...	$sl[S].\texttt{pop}()$
ℓ: `while` G `then` ...	$sl[S].\texttt{push}(\ sl.\texttt{top} \sqcup \Gamma_{\mathcal{E}}(G)\ )$
... `end`; ℓ: ...	$sl[S].\texttt{pop}()$

Figure 9.10: Reference monitor $\mathcal{R}_{TI}$ for TINI

By construction, each terminating execution of $\mathcal{R}_{TI} \rhd S$ corresponds to a terminating execution by S in which every assignment statement in that execution satisfies Safe Assignment Statements.

$\mathcal{R}_{TI} \rhd S$ satisfies TINI, by definition, if the following holds.

$$(\forall \lambda \in \Lambda\colon\ V_{\not\in\lambda} \overset{\mathcal{R}_{TI} \rhd S}{\not\longrightarrow_{\mathsf{ti}}} V_{\sqsubseteq\lambda}) \tag{9.30}$$

If (9.30) does not hold then, according to definition (9.15) of $\mathcal{V} \overset{S}{\not\longrightarrow_{\mathsf{ti}}} \mathcal{W}$, there must be initial states s and s' of terminating executions that agree on the initial values of variables in $V_{\sqsubseteq\lambda}$ but do not agree on the final values of those variables. We can show that this scenerio is impossible by assuming that such a problematic pair of terminating executions exists and using them to derive a contradiction.

If two executions of $\mathcal{R}_{TI} \rhd S$ do not have the same final values for variables in $V_{\sqsubseteq\lambda}$, then there must be an earliest state where their values for variables in $V_{\sqsubseteq\lambda}$ disagree. The disagreement must be caused by an assignment statement that was affected by a value in some variable outside of $V_{\sqsubseteq\lambda}$, since the two executions agreed on values for $V_{\sqsubseteq\lambda}$ in all previous states. However, such an assignment statement would have violated Safe Assignment Statements, so execution would be blocked before reaching that assignment statement, which contradicts the assumption that we started with terminating executions.

But observe that if S does not satisfy TINI then there must be terminating executions of S that become nonterminating executions of $\mathcal{R}_{TI} \rhd S$. This smaller set of terminating executions for $\mathcal{R}_{TI} \rhd S$ exhibits added correlations among variables in the initial states of terminating executions. When variables are correlated, the value of one can be used to predict the value of others, potentially compromising confidentiality. We might have hoped that the terminating executions of $\mathcal{R}_{TI} \rhd S$ would reveal to a λ-observer nothing about the initial values of variables in $V_{\not\in\lambda}$. But it is not the case—even though $\mathcal{R}_{TI} \rhd S$ satisfies TINI.

TINI is satisfied by $\mathcal{R}_{TI} \triangleright S$ because TINI is concerned only with terminating executions, and differences in the initial values of variables in $V_{\not\sqsubseteq\lambda}$ are not detectable by reading variables from $V_{\sqsubseteq\lambda}$ in the final states of terminating executions by $\mathcal{R}_{TI} \triangleright S$. But a less-stringent form of confidentiality is being enforced because of the smaller set of possible initial values for $V_{\not\sqsubseteq\lambda}$. Confidentiality for nonterminating executions is not affected, though, because we have been assuming that a λ-observer cannot determine whether an execution being observed will terminate. Therefore, such an observer could not make inferences about the initial value of variables in $V_{\not\sqsubseteq\lambda}$ during executions of $\mathcal{R}_{TI} \triangleright S$ that have not (yet) terminated. In short, characterizing the enforcement that $\mathcal{R}_{TI}$ provides is subtle.

9.2.5 Comparison of TINI Enforcement Mechanisms

One difference between type-correctness and dynamic enforcement is overhead. Type-correctness incurs overhead before a program is executed but imposes no runtime overhead. With dynamic enforcement, transfers of control to reference monitor $\mathcal{R}_{TI}$ incur the overhead of a context switch, and executing the $\mathcal{R}_{TI}$ actions takes time. However, $\mathcal{R}_{TI}$ only checks an assignment statement when that statement is reached during an execution, so potentially fewer assignment statements need to be checked (although each assignment statement is checked each time it is executed).

Another difference between type-correctness and dynamic enforcement is permissiveness. Type-correctness blocks execution of any IMP program containing any statement that would violate TINI if executed in isolation. $\mathcal{R}_{TI}$ is less conservative—it allows some terminating executions by programs that are not type-correct. The following programs illustrate, where $\Gamma(x_{\mathsf{L}}) = \mathsf{L}$ and $\Gamma(x_{\mathsf{H}}) = \mathsf{H}$ hold.

$$\texttt{if } even(x_{\mathsf{H}}) \texttt{ then } x_{\mathsf{L}} := 1 \texttt{ else skip fi} \tag{9.31}$$

$$\texttt{if } even(x_{\mathsf{L}}) \texttt{ then } x_{\mathsf{L}} := x_{\mathsf{H}} \texttt{ else skip fi} \tag{9.32}$$

Neither program is type-correct, and both programs violate TINI. Yet $\mathcal{R}_{TI}$ allows executions of (9.31) and (9.32) that do not attempt to execute an assignment statement (since each of the assignment statements in those programs violates Safe Assignment Statements). In particular, $\mathcal{R}_{TI}$ allows executions of (9.31) that start in states where $even(x_{\mathsf{H}})$ is *false*, and $\mathcal{R}_{TI}$ allows executions of (9.32) that start in states where $even(x_{\mathsf{L}})$ is *false*.

Some executions of (9.31) and (9.32) are blocked by $\mathcal{R}_{TI}$, though, resulting in a smaller set of terminating executions where x_{H} is necessarily odd in the initial states of all terminating executions. TINI nevertheless is enforced, because TINI is concerned only with terminating executions and any differences in the (odd) initial values for x_{H} in terminating executions are hidden from an L-observer reading x_{L}.

Could different typing rules allow substantial improvements in premissiveness when we use type-correctness for enforcement? The typing rules of Fig-

ure 9.6, for example, reject a program if it contains a problematic assignment statement that cannot be reached in any terminating execution. If the typing rules could identify and ignore such unreachable assignment statements then more programs would be type-correct and, thus, more programs would be executable. However, to determine that a statement is reachable would require the typing rules to determine whether `while` loops are guaranteed to terminate and whether the guards for a collection of `if` statements all could hold during one execution. The undecidability of the halting problem implies no algorithm can exist to make such inferences. Since the set of typing rules is defining an algorithm (which is used for type checking), we must conclude that inferences we would need about statement reachability cannot be incorporated into typing rules.

9.3 Progress Sensitive Noninterference (PSNI)

Malware concurrently executing on the same computer as some program S might be able to provide observers with information about the sequence of the intermediate states produced as S executes. The malware would have the same label λ as whatever program it co-opted, so it is a λ-observer. If nothing is known about the execution speed of S, then the scenerio can be characterized as follows.

- *Interactive.* A λ-observer can read variables in $V_{\sqsubseteq\lambda}$ before, during, and after execution of a program S with variables V.
- *Nontermination Detection.* A λ-observer can detect nontermination of program S.

All terminating and nonterminating executions are visible to λ-observers in such settings.

A generalization of TINI that is well suited for this setting is Progress Sensitive Noninterference (PSNI). PSNI specifies that the variables $V_{\sqsubseteq\lambda}$ in the sequence of intermediate states produced by executing S are not affected by the initial values of variables in $V_{\not\sqsubseteq\lambda}$. So PSNI specifies limits on what is revealed about the initial values of variables in $V_{\not\sqsubseteq\lambda}$ to a λ-observer monitoring the values of variables $V_{\sqsubseteq\lambda}$ during execution.

The following program illustrates how information about the initial values of variables from $V_{\not\sqsubseteq\lambda}$ might be revealed through the sequence of values assigned to variables in $V_{\sqsubseteq\lambda}$ during an execution.

$$\begin{array}{l} \ell_1\colon y := 0;\ \ell_2\colon z := 0; \\ \ell_3\colon \texttt{if}\ x = 0\ \texttt{then}\ \ell_4\colon y := 1;\ \ell_5\colon z := 2 \\ \qquad\qquad\quad \texttt{else}\ \ell_6\colon x := 4;\ \ell_7\colon z := 2;\ \ell_8\colon y := 1 \\ \ell_9\colon \end{array} \tag{9.33}$$

Notice that the initial value of x determines the order of updates to y and z in the `if` statement. So if $\Gamma(x) = \mathsf{H}$, $\Gamma(y) = \mathsf{L}$, and $\Gamma(z) = \mathsf{L}$ hold, then the sequence of values assigned to variables $y, z \in V_{\sqsubseteq\mathsf{L}}$ in intermediate states reveals

$$\sigma\colon \begin{bmatrix} x \mapsto 0 \\ y \mapsto ? \\ z \mapsto ? \\ pc \mapsto \ell_1 \end{bmatrix} \begin{bmatrix} x \mapsto 0 \\ y \mapsto 0 \\ z \mapsto ? \\ pc \mapsto \ell_2 \end{bmatrix} \begin{bmatrix} x \mapsto 0 \\ y \mapsto 0 \\ z \mapsto 0 \\ pc \mapsto \ell_3 \end{bmatrix} \begin{bmatrix} x \mapsto 0 \\ y \mapsto 0 \\ z \mapsto 0 \\ pc \mapsto \ell_4 \end{bmatrix} \begin{bmatrix} x \mapsto 0 \\ y \mapsto 1 \\ z \mapsto 0 \\ pc \mapsto \ell_5 \end{bmatrix} \begin{bmatrix} x \mapsto 0 \\ y \mapsto 1 \\ z \mapsto 2 \\ pc \mapsto \ell_9 \end{bmatrix}$$

$$\sigma'\colon \begin{bmatrix} x \mapsto 1 \\ y \mapsto ? \\ z \mapsto ? \\ pc \mapsto \ell_1 \end{bmatrix} \begin{bmatrix} x \mapsto 1 \\ y \mapsto 0 \\ z \mapsto ? \\ pc \mapsto \ell_2 \end{bmatrix} \begin{bmatrix} x \mapsto 1 \\ y \mapsto 0 \\ z \mapsto 0 \\ pc \mapsto \ell_3 \end{bmatrix} \begin{bmatrix} x \mapsto 1 \\ y \mapsto 0 \\ z \mapsto 0 \\ pc \mapsto \ell_6 \end{bmatrix} \begin{bmatrix} x \mapsto 4 \\ y \mapsto 0 \\ z \mapsto 0 \\ pc \mapsto \ell_7 \end{bmatrix} \begin{bmatrix} x \mapsto 4 \\ y \mapsto 0 \\ z \mapsto 2 \\ pc \mapsto \ell_8 \end{bmatrix} \begin{bmatrix} x \mapsto 0 \\ y \mapsto 1 \\ z \mapsto 2 \\ pc \mapsto \ell_9 \end{bmatrix}$$

Figure 9.11: Sequences σ and σ' of intermediate states for executions of (9.33)

information about the initial value of variable $x \in V_{\not\in \mathsf{L}}$. Yet program (9.33) satisfies TINI, because the final values of y and z are the same for all initial values of x.

For specifying PSNI, we define a predicate $\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \mathcal{W}$ that holds if and only if executions of S from initial states that differ only in the values of variables in $\mathcal{V}$ produce indistinguishable sequences of values for the variables in $\mathcal{W}$. As an example, if S is program (9.33) then $\{x\} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \{y, z\}$ would not hold, because different initial values for x can cause executions where the sequences of values assigned to variables y and z during one execution can differ from the sequences assigned during another. Figure 9.11 illustrates by giving two *execution traces*, each specifying an initial state followed by the intermediate states produced during an execution of program (9.33). There, we write "$x \mapsto val$" when defining a state that maps variable name x to a value val, and we use ? to represent an unknown or uninitialized value. So execution trace σ describes the execution from an initial state where $x = 0$ holds, and execution trace σ' describes execution from an initial state where $x = 1$ holds.

For $\{x\} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \{y, z\}$ to hold, the sequences of updates to the values for y and z in σ is required to be indistinguishable from that sequence in σ'. We can check this requirement by constructing *projected execution traces* $\langle \sigma|_{\{y,z\}} \rangle$ and $\langle \sigma'|_{\{y,z\}} \rangle$, which give the sequences of changed values for y and z.

$$\langle \sigma|_{\{y,z\}} \rangle\colon \begin{bmatrix} y \mapsto ? \\ z \mapsto ? \end{bmatrix} \begin{bmatrix} y \mapsto 0 \\ z \mapsto ? \end{bmatrix} \begin{bmatrix} y \mapsto 0 \\ z \mapsto 0 \end{bmatrix} \begin{bmatrix} y \mapsto 1 \\ z \mapsto 0 \end{bmatrix} \begin{bmatrix} y \mapsto 1 \\ z \mapsto 2 \end{bmatrix} \tag{9.34}$$

$$\langle \sigma'|_{\{y,z\}} \rangle\colon \begin{bmatrix} y \mapsto ? \\ z \mapsto ? \end{bmatrix} \begin{bmatrix} y \mapsto 0 \\ z \mapsto ? \end{bmatrix} \begin{bmatrix} y \mapsto 0 \\ z \mapsto 0 \end{bmatrix} \begin{bmatrix} y \mapsto 0 \\ z \mapsto 2 \end{bmatrix} \begin{bmatrix} y \mapsto 1 \\ z \mapsto 2 \end{bmatrix} \tag{9.35}$$

The fourth states of (9.34) and (9.35) are different, so $\{x\} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \{y, z\}$ does not hold—information about the initial value of x would be revealed to an observer that is monitoring y and z.

Construction of a projected execution trace $\langle \sigma|_{\mathcal{V}} \rangle$ involves two steps: (i) variables not in $\mathcal{V}$ are deleted from each state of σ and (ii) runs of identical states in the result are collapsed into a single state. We formalize this as follows. For a sequence σ states, define $\sigma[i]$ to be its i^{th} state, $\sigma[i..]$ to be the suffix starting with $\sigma[i]$, and the *destutter* operation $\langle \sigma \rangle$ to be:

$$\langle \sigma \rangle : \begin{cases} \sigma[1] & \text{if } len(\sigma) = 1 \ \vee \ (\forall 1 < i \leq len(\sigma) : \ \sigma[1] = \sigma[i]) \\ \sigma[1] \ \langle \sigma[2..] \rangle & \text{if } len(\sigma) > 1 \ \wedge \ \sigma[1] \neq \sigma[2] \\ \langle \sigma[2..] \rangle & \text{otherwise} \end{cases}$$

A *trace projection* $\sigma|_{\mathcal{V}}$ is the sequence of state projections for the states in σ:

$$\sigma|_{\mathcal{V}} : \ \ \sigma[1]|_{\mathcal{V}} \ \sigma[2]|_{\mathcal{V}} \ \ldots$$

And predicate $\sigma =_{\mathcal{V}} \sigma'$ is defined to hold if projected executions traces $\langle \sigma|_{\mathcal{V}} \rangle$ and $\langle \sigma'|_{\mathcal{V}} \rangle$ are the same length and have identical i^{th} states, for each i

$$\sigma =_{\mathcal{V}} \sigma' : \quad \langle \sigma|_{\mathcal{V}} \rangle = \langle \sigma'|_{\mathcal{V}} \rangle$$

We are collapsing runs of identical states when forming a projected execution trace $\langle \sigma|_{\mathcal{V}} \rangle$ because it results in the sequence of states that would be seen by an observer monitoring $\mathcal{V}$, since state transitions are not detectable to such an observer unless the value of some variable in $\mathcal{V}$ changes. Notice, a projected execution trace $\langle \sigma|_{\mathcal{V}} \rangle$ might have finite length even though σ has infinite length—this case arises with a nonterminating loop where all assignment statements in the loop body update variables that are not visibile to the observer.

Our formal definition for $\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \mathcal{W}$ employs a function $[\![S]\!]^{\mathsf{tr}}(s)$ that, for a deterministic program S, maps an initial state s to an execution trace that begins with s and is followed by the sequence of intermediate states that would be produced by executing S. Of interest are the pairs of projected execution traces for executions that start in initial states s and s' satisfying $s =_{\overline{\mathcal{V}}} s'$.

$$\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \mathcal{W} : \quad (\forall s, s' \in \mathit{Init}_S : \ s =_{\overline{\mathcal{V}}} s' \ \Rightarrow \ [\![S]\!]^{\mathsf{tr}}(s) =_{\mathcal{W}} [\![S]\!]^{\mathsf{tr}}(s')) \tag{9.36}$$

So for $\mathcal{V} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \mathcal{W}$ to hold, different initial values for variables in $\mathcal{V}$ must result in indistinguishable projected execution sequences for variables in $\mathcal{W}$.

To illustrate the use of definition (9.36), we check whether $\{x\} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \{y, z\}$ holds when S is program (9.33). $\overline{\mathcal{V}}$ is $\{y, z\}$ (since $\mathcal{V}$ is $\{x\}$) and $\mathcal{W}$ is $\{y, z\}$, so we get:

$$(\forall s, s' \in \mathit{Init}_S : \ s =_{\{y,z\}} s' \ \Rightarrow \ [\![S]\!]^{\mathsf{tr}}(s) =_{\{y,z\}} [\![S]\!]^{\mathsf{tr}}(s')) \tag{9.37}$$

The earlier claim that $\{x\} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \{y, z\}$ does not hold would be confirmed by showing that (9.37) does not hold. The initial states of σ and σ' in Figure 9.11

do satisfy antecedent $s =_{\{y,z\}} s'$, but the consequent $[\![S]\!]^{\mathsf{tr}}(s) =_{\{y,z\}} [\![S]\!]^{\mathsf{tr}}(s')$ of (9.37) does not hold since (9.34) and (9.35) are different. We have confirmed that $\{x\} \overset{S}{\not\rightarrow}_{\mathsf{ps}} \{y,z\}$ does not hold.

We now have all the building blocks needed to specify that the values of variables $V_{\not\sqsubseteq\lambda}$ in initial states are not allowed to affect the values of variables $V_{\sqsubseteq\lambda}$ in the intermediate states or in the final state produced by execution of a program S.

> **Progress Sensitive Noninterference (PSNI).** For a deterministic program S with a set V of variables having labels from a set Λ with a partial order $\sqsubseteq$:
>
> $$(\forall \lambda \in \Lambda\colon\ V_{\not\sqsubseteq\lambda} \overset{S}{\not\rightarrow}_{\mathsf{ps}} V_{\sqsubseteq\lambda}) \qquad \square$$

9.3.1 PSNI Enforcement

TINI ignores nonterminating executions. PSNI does not and, consequently, additional implicit flows can arise. For example, in IMP program

$$x := 0;\ \ \texttt{while}\ y = 0\ \texttt{do skip end};\ \ \ell\colon x := 23$$

assignment statement ℓ is not in the body of the `while` statement, so (by definition) guard $y = 0$ is not in $\Theta_S(\ell)$. That guard, nevertheless, affects whether assignment statement ℓ is reached: ℓ is not reached unless the `while` statement terminates. Also, this implicit flow is not prevented by Safe Assignment Statements (page 253).

For IMP programs in which all `while` statements always terminate, compliance with Safe Assignment Statements does enforce PSNI, because $\Theta_S(\ell)$ for such a program S does contain all of the guards that affect whether ℓ executes. So one approach to enforcing PSNI would be to reject any IMP program that includes nonterminating `while` statements and to check the remaining programs for compliance with Safe Assignment Statements. But this is not feasible. The undecidability of the halting problem implies that typing rules cannot exist for identifying `while` statements that do not always terminate. For the same reason, no test that a reference monitor could employ during execution would be able to predict nontermination of a `while` statement that has been reached or has started executing.

So rather than deal with `while` statements that are not guaranteed to terminate, we define a new programming language IMP$^-$ that instead of `while` statements offers `for`-loop statements that always terminate. In a syntactically correct IMP$^-$ `for`-loop statement

$$\texttt{for}\ w := \mathit{expr}\ \textbf{to}\ \mathit{expr}'\ \texttt{do}\ S\ \texttt{end} \tag{9.38}$$

w must be an integer variable, *expr* and *expr′* must be integer-valued expressions, and body S must be an IMP$^-$ statement that does not contain any assignment statements with w as a target or with any variable in $\mathit{Vars}(\mathit{expr}')$ as

a target. In executions of such a `for`-loop statement, body S is executed zero or more times. This behavior is simulated by the following IMP code.

$$w := expr;\ \texttt{while}\ w \le expr'\ \texttt{do}\ S;\ w := w + 1\ \texttt{end} \tag{9.39}$$

The guard for `for`-loop statement statement (9.61) is $expr \le w \le expr'$, since this predicate holds each time S starts executing. Therefore,

$$expr \le w \le expr' \in \Theta_T(\ell)$$

holds for every statement ℓ in body S of a `for`-loop statement that appears in a program T.

9.3.2 Enforcing PSNI with Types

Type-correctness can be used to enforce PSNI. We employ a class of judgements $\Gamma, \gamma \vdash_{\mathsf{ps}} S$ whose validity is defined as follows.

Valid Judgements for PSNI. A judgement $\Gamma, \gamma \vdash_{\mathsf{ps}} S$ for a program S with variables V is *valid for PSNI* if and only if

(i) S satisfies PSNI: $(\forall \lambda \in \Lambda\colon\ V_{\not\in\lambda} \overset{S}{\not\longrightarrow}_{\mathsf{ps}} V_{\sqsubseteq\lambda})$

(ii) Target w of every assignment statement in S satisfies: $\gamma \sqsubseteq \Gamma(w)$. □

So if $\Gamma, \bot_\Lambda \vdash_{\mathsf{ps}} S$ is valid for PSNI then a program S satisfies PSNI.

An IMP$^-$ program S is type-correct if the judgement $\Gamma, \bot_\Lambda \vdash_{\mathsf{ps}} S$ can be derived using the typing rules in Figure 9.17. Each of those typing rules derives a judgement that is valid for PSNI whenever all of the rule's hypotheses are valid. The set of typing rules in Figure 9.17 is obtained by replacing rule WHILE in Figure 9.6 with a rule FOR to handle `for`-loop statements. In rule FOR, $Tgts(S)$ denotes the set of variables that are the targets of assignment statements in S. The hypotheses of rule FOR imply that (i) $w := expr$ and $w := w + 1$ used in implementation (9.39) of the `for`-loop statement each complies with the Safe Assignment Statements, and (ii) the iterations eventually end because variable w and variables in upper bound $expr'$ are not updated in body S of the `for`-loop statement.

9.3.3 Enforcing PSNI at Runtime

By monitoring intermediate states of an execution, a λ-observer with knowledge of the source code might be able to ascertain when an execution has become blocked by a runtime enforcement mechanism. This prompts us to adopt a conservative stance for runtime enforcement of PSNI, adding one further stipulation to the Interactive and Nontermination Detection assumptions (on page 261):

- *Blocking Detection.* A λ-observer can detect when execution of a program becomes blocked by a runtime enforcement mechanism.

$$\text{SKIP:}\ \frac{}{\Gamma, \gamma \vdash_{\mathsf{ps}} \texttt{skip}} \qquad \text{ASSIGN:}\ \frac{\gamma \sqcup \Gamma_{\mathcal{E}}(expr) \sqsubseteq \Gamma(w)}{\Gamma, \gamma \vdash_{\mathsf{ti}} w := expr}$$

$$\text{IF:}\ \frac{\Gamma_E(expr) = \lambda, \quad \Gamma, \gamma \sqcup \lambda \vdash_{\mathsf{ps}} S, \quad \Gamma, \gamma \sqcup \lambda \vdash_{\mathsf{ps}} S'}{\Gamma, \gamma \vdash_{\mathsf{ps}} \texttt{if}\ expr\ \texttt{then}\ S\ \texttt{else}\ S'\ \texttt{fi}} \qquad \text{SEQ:}\ \frac{\Gamma, \gamma \vdash_{\mathsf{ps}} S, \quad \Gamma, \gamma \vdash_{\mathsf{ps}} S'}{\Gamma, \gamma \vdash_{\mathsf{ps}} S;\ S'}$$

$$\text{FOR:}\ \frac{\begin{array}{c} w \notin \mathit{Tgts}(S), \quad \mathit{Tgts}(S) \cap \mathit{Vars}(expr') = \varnothing, \\ \Gamma_{\mathcal{E}}(expr) = \lambda, \quad \Gamma_{\mathcal{E}}(expr') = \lambda', \quad \Gamma(w) = \lambda'', \\ \gamma \sqcup \lambda \sqcup \lambda' \sqsubseteq \lambda'', \quad \Gamma, \gamma \sqcup \lambda \sqcup \lambda' \sqcup \lambda'' \vdash_{\mathsf{ps}} S \end{array}}{\Gamma, \gamma \vdash_{\mathsf{ps}} \texttt{for}\ w := expr\ \textbf{to}\ expr'\ \texttt{do}\ S\ \texttt{end}}$$

Figure 9.12: Typing rules for PSNI compliance

Unfortunately, detection of blocking can result in problematic implicit flows to λ-observers, as the following IMP$^-$ program illustrates. Assume that $\bot_\Lambda = \mathsf{L}$, $\Gamma(i_\mathsf{L}) = \mathsf{L}$, $\Gamma(x_\mathsf{H}) = \mathsf{H}$, and $\mathsf{H} \not\sqsubseteq \mathsf{L}$ hold.

$$\begin{array}{ll} S\colon & \texttt{for}\ i_\mathsf{L} := 0\ \textbf{to}\ N\ \texttt{do} \\ & \quad \texttt{if}\ x_\mathsf{H} = i_\mathsf{L}\ \texttt{then}\ \ell_S\colon x_\mathsf{L} := i_\mathsf{L} \\ & \qquad\qquad\ \texttt{else skip} \\ & \quad \texttt{fi} \\ & \texttt{end} \end{array} \tag{9.40}$$

Assignment statement ℓ_S does not comply with Safe Assignment Statements because of the implicit flow from x_H in `if` statement guard $x_\mathsf{H} = i_\mathsf{L}$. Therefore, a runtime monitor must block execution of S before it reaches ℓ_S. This execution blocking does not suffice, though. An L-observer that is monitoring i_L and that detects the execution blocking can infer the initial value of x_H—it is the last observed value of i_L.

A runtime monitor for enforcing PSNI therefore must not only prevent explicit flows and implicit flows. It also must prevent *detection flows*, where the knowledge that execution has blocked allows a λ-observer to learn information about the initial value of a variable in $V_{\not\sqsubseteq\lambda}$. Notice, variables in $V_{\sqsubseteq\lambda}$ are not affected by detection flows, so PSNI per se has not been violated. Nevertheless, since the goal of enforcing PSNI is to prevent λ-observers from learning about the initial values of variables in $V_{\not\sqsubseteq\lambda}$, detection flows must be prevented.

In general, detection flows can occur only if (i) a λ-observer can determine that execution blocked when some statement ℓ was reached, and (ii) $\Gamma(v) \not\sqsubseteq \lambda$ holds where $v \in \mathit{Vars}(G)$ and $G \in \Theta_S(\ell)$ hold. With program (9.56), for example, $\Theta_S(\ell_S)$ contains the guard $x_\mathsf{H} = i_\mathsf{L}$ and, therefore, blocking an execution upon reaching ℓ_S creates a detection flow from x_H to an L-observer.

To enforce PSNI at runtime without creating detection flows requires invoking a reference monitor $\mathcal{R}_{PI}$ in order to

(i) block execution in anticipation of executing an assignment statement that does not comply with Safe Assignment Statements, but

(ii) avoid detection flows by not blocking execution upon reaching a statement ℓ unless $\Gamma_{\mathcal{E}}(G) = \bot_{\Lambda}$ holds for all guards G in $\Theta_S(\ell)$.

Ideally, unnecessary blocking also would be avoided, but doing that is not necessary for enforcing PSNI.

Reference monitor $\mathcal{R}_{TI}$ in Figure 9.10 satisfies (i) but not (ii). TINI ignores blocked executions that abort, so satisfying (ii) is not required for $\mathcal{R}_{TI}$ to enforce TINI. For $\mathcal{R}_{PI}$ to satisfy both (i) and (ii), $\mathcal{R}_{PI}$ would have to block execution upon reaching a control-flow statement S that contains a problematic assignment statement. But a reference monitor must, by definition, operate ignorant of code not yet reached. Consequently, $\mathcal{R}_{PI}$ would have to be conservative and block execution upon reaching any control-flow statement having a guard G that satisfies $\Gamma_{\mathcal{E}}(G) \neq \bot_{\Lambda}$. PSNI would be enforced and detection flows would be prevented. Most executions would be blocked, though, making $\mathcal{R}_{PI}$ even less permissive than type-checking. Moreover, the constraints on reference monitors make it impossible for any reference monitor to do significantly better.

We can do better. With *hybrid enforcement*, a reference monitor is permitted to perform a static analysis of the monitored program. Consequently, whenever a control-flow statement S is reached, a hybrid enforcement mechanism can decide whether to block execution by using knowledge of the statements that appear in the body of S. That knowledge allows the following scheme for avoiding detection flows.

> **Hybrid Enforcement of PSNI.** Upon reaching a control-flow statement with guard G and body S, the enforcement mechanism blocks execution if S contains an assignment statement that not satisfy Safe Assignment Statements.[8] □

This enforcement mechanism, for example, would block (9.56) when the `for` statement is first reached, because the body of that `for` statement contains problematic assignment statement ℓ_S.

Notice, Hybrid Enforcement of PSNI is more permissive than type-checking. Also, with Hybrid Enforcement of PSNI, fewer control-flow statements cause aborts than $\mathcal{R}_{PI}$ does, since Hybrid Enforcement of PSNI does not require $\Gamma_{\mathcal{E}}(G) = \bot_{\Lambda}$ to hold for guards G that will affect whether execution is blocked (i.e., reference monitor requirement (ii) above).

[8] This restriction is sometimes called *no sensitive upgrade*.

