Proving noninterference for a while-language using small-step operational semantics

Andrew Myers

March 2011*

This is a tutorial on how to prove the soundness of a security type system in the context of a simple language, using small-step semantics. The original proof of soundness for such a language [VSI96] was done using big-step semantics. Small-step semantics is interesting because it is more compatible with language features such as concurrency and nondeterminism, allowing it to "scale up" to more interesting languages. But it also presents some added challenges.

1 Syntax

The language is a simple while-language similar to IMP [Win93], but with security typing. Let us call it W.

$$
\begin{aligned}
x &\in \textbf{Var} \\
M &\in \textbf{Var} \to \mathbb{Z} \\
a\ (\in \textbf{AExp}) &::= n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \\
t\ (\in \mathbb{T}) &::= \textbf{true} \mid \textbf{false} \\
b\ (\in \textbf{BExp}) &::= t \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \neg b \mid a_1 = a_2 \mid a_1 \leq a_2 \\
c\ (\in \textbf{Com}) &::= \textbf{skip} \mid x := a \mid c_1;\ c_2 \mid \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \mid \textbf{while}\ b\ \textbf{do}\ c
\end{aligned}
$$

2 Operational semantics

For simplicity, big-step semantics are used for arithmetic and boolean expressions.

Arithmetic Expressions

The relation $\langle M, a\rangle \Downarrow n$ means a evaluates to integer n in memory M.

*This tutorial was originally written in November 2009 for the Marktoberdorf Summer School on Logics and Languages for Reliability and Security. It was updated in March 2011 to clarify the completeness argument, and a few small corrections were made in July 2012 for use in the Oregon Programming Languages Summer School.

$$\frac{}{\langle M, n\rangle \Downarrow n} \qquad \frac{}{\langle M, x\rangle \Downarrow M(x)}$$

$$\frac{\langle M, a_1\rangle \Downarrow n_1 \quad \langle M, a_2\rangle \Downarrow n_2 \quad n_3 = n_1 \oplus n_2}{\langle M, a_1 \oplus a_2\rangle \Downarrow n_3}$$

Boolean Expressions

Analogous to arithmetic expressions.

Commands

$\langle$M, **skip**$\rangle$ is a final configuration. We write $f[x := y]$ for the function g such that $g(x) = y$ and $\forall x' \neq x.g(x') = f(x')$.

$$\frac{\langle M, a\rangle \Downarrow n}{\langle M, x := a\rangle \longrightarrow \langle M[x := n], \textbf{skip}\rangle} \qquad \frac{}{\langle M, \textbf{skip}; c\rangle \longrightarrow \langle M, c\rangle}$$

$$\frac{\langle M, c_1\rangle \longrightarrow \langle M', c_1'\rangle}{\langle M, c_1; c_2\rangle \longrightarrow \langle M', c_1'; c_2\rangle}$$

$$\frac{\langle M, b\rangle \Downarrow \textbf{true}}{\langle M, \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2\rangle \longrightarrow \langle M, c_1\rangle} \qquad \frac{\langle M, b\rangle \Downarrow \textbf{false}}{\langle M, \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2\rangle \longrightarrow \langle M, c_2\rangle}$$

$$\frac{}{\langle M, \textbf{while } b \textbf{ do } c\rangle \longrightarrow \langle M, \textbf{if } b \textbf{ then } (c; \textbf{while } b \textbf{ do } c) \textbf{ else skip}\rangle}$$

3 Static semantics

Labels are drawn from a pointed lattice $\mathcal{L}$ with a bottom element $\bot$. The mapping Γ : **Var** $\to \mathcal{L}$ gives the unique label of each variable name.

Arithmetic typing

The judgment $\vdash a : L$ means a has label L.

$$\frac{}{\vdash x : \Gamma(x)} \qquad \frac{}{\vdash n : \bot} \qquad \frac{\vdash a_1 : L_1 \quad \vdash a_2 : L_2}{\vdash a_1 \oplus a_2 : L_1 \sqcup L_2}$$

Boolean typing

The judgment $\vdash b : L$ means b has label L.

$$\frac{}{\vdash \textbf{true} : \bot} \qquad \frac{}{\vdash \textbf{false} : \bot} \qquad \frac{\vdash a_1 : L_1 \quad \vdash a_2 : L_2}{\vdash a_1 = a_2 : L_1 \sqcup L_2}$$

Command typing

The judgment $pc \vdash c$ means that c is well-typed with the program-counter label pc. We write $\vdash c$ to mean $\bot \vdash c$.

$$\frac{}{pc \vdash \textbf{skip}} \qquad \frac{\vdash a : L' \quad L' \sqcup pc \sqsubseteq \Gamma(x)}{pc \vdash x := a}$$

$$\frac{pc \vdash c_1 \quad pc \vdash c_2}{pc \vdash c_1; c_2} \qquad \frac{\vdash b : L' \quad pc \sqcup L' \vdash c_1 \quad pc \sqcup L' \vdash c_2}{pc \vdash \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2}$$

$$\frac{\vdash b : L' \quad pc \sqcup L' \vdash c}{pc \vdash \textbf{while}\ b\ \textbf{do}\ c}$$

Notice that the pc label increases inside **if** and **while** to account for the label on the guard expression b.

4 Memory equivalence

We use a distinguished label L ("low") to define what is visible to the low observer. It is assumed that the low observer can see memory locations x where $\Gamma(x) \sqsubseteq L$. These are the *low variables*.

We write $M_1 \approx_L M_2$ to mean that memories M_1 and M_2 are indistinguishable to the low observer. That is, they agree on all locations x visible to the low observer.

If a label L' is not visible—that is, $L' \not\sqsubseteq L$, we say that L' is *high*. A variable with such a label is a high variable.

5 Noninterference

We say that a program c satisfies noninterference [GM82] if equivalent initial memories produce equivalent final memories:

$$M_1 \approx_L M_2 \wedge \langle M_1, c\rangle \longrightarrow^* \langle M_1', \textbf{skip}\rangle \wedge \langle M_2, c\rangle \longrightarrow^* \langle M_2', \textbf{skip}\rangle \implies M_1' \approx_L M_2'$$

This is a *termination-insensitive* notion of noninterference, because it says nothing about what happens if one of the two evaluations diverges.

6 Soundness

The security type system is sound if all well-typed programs satisfy noninterference:

$$\vdash c \wedge M_1 \approx_L M_2 \wedge \langle M_1, c\rangle \longrightarrow^* \langle M_1', \textbf{skip}\rangle \wedge \langle M_2, c\rangle \longrightarrow^* \langle M_2', \textbf{skip}\rangle \Rightarrow M_1' \approx_L M_2'$$

7 Proving noninterference for W

Proving noninterference in a small-step setting seems to require augmenting the operational semantics in some way. (This is not true of big-step (natural) semantics or of denotational semantics.) The reason is that small-step evaluation does not preserve enough information about the structure of the program being executed.

We want to reason about two different executions of a program c. To accomplish this we will syntactically mark the parts of the program that are allowed to be different in the two executions. We write $[c]$ to indicate a command c that may be different. We also write $[n]$ and $[t]$ to indicate arithmetic and boolean values that may be different. The result of evaluating an arithmetic expression is either a number n or a bracketed number $[n]$, and similarly with boolean expressions. The resulting augmented language we will call [W].

The noninterference technique shown here combines ideas from work by Volpano, Smith, and Irvine [VSI96], by Zdancewic and Myers [ZM02] and by Pottier and Simonet [PS03]. Like Zdancewic's approach, it establishes an equivalence between two parallel executions operating on different initial states. Like Pottier and Simonet's approach, it introduces a simple syntactic construct (brackets) to keep track of high subexpressions.

7.1 Equivalence on memories

We augment memories to be allowed to map (high) variables to bracketed results. Then two memory locations are equivalent if they contain corresponding values: either both are the same integer or both are bracketed integers:

$$\frac{}{n \approx n} \qquad\qquad \frac{}{[n_1] \approx [n_2]}$$

We have the corresponding rules for booleans too. Now, we can define another equivalence relation $M_1 \approx M_2$ on augmented memories:

$$M_1 \approx M_2 \iff \forall x. M_1(x) \approx M_2(x)$$

For any two equivalent standard memories $M_1 \approx_L M_2$, there are clearly two augmented memories $M_1' \approx M_2'$ that agree with the standard memories everywhere except for brackets. We simply put brackets around the values of all high variables. Conversely, for any two equivalent augmented memories where brackets are used only for high variables, there are two equivalent standard memories.

Given a [W] memory M, let $\lfloor M \rfloor$ represent its projection to an original memory by removing all brackets. We write $\vdash M$ if the values of all high (and only high) variables have brackets, and say that such a memory is well-formed. Then if $M_1 \approx M_2$ and $\vdash M_1$ and $\vdash M_2$, the projections of these well-formed memories are low-equivalent: $\lfloor M_1 \rfloor \approx_L \lfloor M_2 \rfloor$.

7.2 Augmented syntax

We allow brackets around values and commands:

$$
\begin{aligned}
a &::= \ldots \mid [n] \\
b &::= \ldots \mid [t] \\
c &::= \ldots \mid [c]
\end{aligned}
$$

7.3 Equivalence on commands

Two commands are equivalent if they are equal modulo bracketed commands:

$$\frac{}{\textbf{skip} \approx \textbf{skip}} \qquad \frac{a \approx a'}{x := a \approx x := a'}$$

$$\frac{c_1 \approx c_1' \quad c_2 \approx c_2'}{c_1; c_2 \approx c_1'; c_2'} \qquad \frac{b \approx b' \quad c_1 \approx c_1' \quad c_2 \approx c_2'}{\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \approx \textbf{if } b' \textbf{ then } c_1' \textbf{ else } c_2'}$$

$$\frac{b \approx b' \quad c \approx c'}{\textbf{while } b \textbf{ do } c \approx \textbf{while } b' \textbf{ do } c'} \qquad \frac{}{[c_1] \approx [c_2]}$$

7.4 Augmented operational semantics

We extend the operational semantics to propagate brackets. Nothing interesting happens from the computational perspective with brackets. They are just a syntactic marker.

Arithmetic

$$\frac{}{\langle M, [n] \rangle \Downarrow [n]} \qquad \frac{\langle M, a_1 \rangle \Downarrow [n_1] \quad \langle M, a_2 \rangle \Downarrow n_2 \quad n = n_1 \oplus n_2}{\langle M, a_1 \oplus a_2 \rangle \Downarrow [n]}$$

$$\frac{\langle M, a_1 \rangle \Downarrow n_1 \quad \langle M, a_2 \rangle \Downarrow [n_2] \quad n = n_1 \oplus n_2}{\langle M, a_1 \oplus a_2 \rangle \Downarrow [n]} \qquad \frac{\langle M, a_1 \rangle \Downarrow [n_1] \quad \langle M, a_2 \rangle \Downarrow [n_2] \quad n = n_1 \oplus n_2}{\langle M, a_1 \oplus a_2 \rangle \Downarrow [n]}$$

Booleans

Follows the same pattern as arithmetic.

Commands

$$\frac{\langle M, c \rangle \longrightarrow \langle M', c' \rangle}{\langle M, [c] \rangle \longrightarrow \langle M', [c'] \rangle} \qquad \frac{}{\langle M, [\textbf{skip}] \rangle \longrightarrow \langle M, \textbf{skip} \rangle}$$

$$\frac{\langle M, b \rangle \Downarrow [\textbf{true}]}{\langle M, \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \rangle \longrightarrow \langle M, [c_1] \rangle} \qquad \frac{\langle M, b \rangle \Downarrow [\textbf{false}]}{\langle M, \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \rangle \longrightarrow \langle M, [c_2] \rangle}$$

We also need to replace the original assignment rule to make sure that memories stay well-formed when a low value is assigned to a high variable. A bracket is added in this case.

$$\frac{\langle M, a\rangle \Downarrow n \quad \Gamma(x) \not\sqsubseteq L}{\langle M, x := a\rangle \longrightarrow \langle M[x := [n]], \textbf{skip}\rangle} \qquad \frac{\langle M, a\rangle \Downarrow n \quad \Gamma(x) \sqsubseteq L}{\langle M, x := a\rangle \longrightarrow \langle M[x := n], \textbf{skip}\rangle}$$

$$\frac{\langle M, a\rangle \Downarrow [n]}{\langle M, x := a\rangle \longrightarrow \langle M[x := [n]], \textbf{skip}\rangle}$$

7.5 Static semantics

We augment the static semantics of the language to account for brackets as follows:

$$\frac{L' \not\sqsubseteq L}{\vdash [n] : L'} \qquad \frac{L' \not\sqsubseteq L}{\vdash [t] : L'}$$

$$\frac{L' \vdash c \quad pc \sqsubseteq L' \quad L' \not\sqsubseteq L}{pc \vdash [c]}$$

These rules ensure that the type system treats bracketed expressions as "high".

7.6 Noninterference in [W]

We can express the security enforced by the [W] type system as the following noninterference condition:

Lemma 1 (Noninterference of [W])

$$\begin{aligned} &\vdash c \wedge \vdash M_1 \wedge \vdash M_2 \wedge M_1 \approx M_2 \\ &\wedge \langle M_1, c\rangle \longrightarrow^* \langle M_1', \textbf{skip}\rangle \\ &\wedge \langle M_2, c\rangle \longrightarrow^* \langle M_2', \textbf{skip}\rangle \\ &\Rightarrow M_1' \approx M_2' \end{aligned}$$

7.7 Completeness of [W]

To use Lemma 1 to prove noninterference for the original language W, we first need to establish a correspondence between the two languages. Every new evaluation rule corresponds to an evaluation rule of the original language, except with extra brackets. Therefore, every step that is performed in the original language can also be performed in the augmented language, and vice versa.

We need to know that any full evaluation of a program in the original language can be simulated in the augmented language, perhaps with extra brackets. Given that c is a [W] command, let us use the notation $\lfloor c \rfloor$ to denote removal of all brackets from c in

the obvious way, yielding a command from W. Using this notation, we can express the completeness of [W] with respect to W:

Lemma 2 (Completeness of [W])

$$\vdash c \wedge \langle \lfloor M \rfloor, \lfloor c \rfloor \rangle \longrightarrow^* \langle M', \textbf{skip} \rangle \Rightarrow \exists M''. \langle M, c \rangle \longrightarrow^* \langle M'', \textbf{skip} \rangle \wedge M' = \lfloor M'' \rfloor$$

Note that the evaluation step on the left-hand side of the implication is a W step, whereas the evaluation on the right-hand side is a [W] step.

This completeness theorem says nothing about divergent evaluations in the source language. These evaluations don't matter because we are proving termination-insensitive noninterference.

Assuming that we have Lemma 2 (it is proved by induction on the number of steps in the evaluation of $\langle \lfloor M \rfloor, \lfloor c \rfloor \rangle$, but the proof is currently left to the reader), we prove noninterference for the original language (see Section 5) as follows.

7.8 Noninterference of W

We start by assuming the three premises of the noninterference condition:

$$M_1 \approx_L M_2 \qquad \langle M_1, c \rangle \longrightarrow^* \langle M_1', \textbf{skip} \rangle \qquad \langle M_2, c \rangle \longrightarrow^* \langle M_2', \textbf{skip} \rangle \tag{1}$$

and our goal is to prove $M_1' \approx_L M_2'$.

Because $M_1 \approx_L M_2$, we know from Section 7.1 that there exist two [W] memories M_3 and M_4 such that the following conditions all hold:

$$\begin{array}{cc} \lfloor M_3 \rfloor = M_1 & \lfloor M_4 \rfloor = M_2 \\ \vdash M_3 & \vdash M_4 \\ M_3 \approx M_4 & \end{array} \tag{2}$$

Because $\langle M_1, c \rangle \longrightarrow^* \langle M_1', \textbf{skip} \rangle$ and $\lfloor M_3 \rfloor = M_1$, completeness tells us there exists some M_1'' such that $\langle M_3, c \rangle \longrightarrow^* \langle M_1'', \textbf{skip} \rangle$, and $M_1' = \lfloor M_1'' \rfloor$. Similarly, there exists M_2'' such that $\langle M_4, c \rangle \longrightarrow^* \langle M_2'', \textbf{skip} \rangle$, and $M_2' = \lfloor M_2'' \rfloor$. Conditions (2) allow us to use noninterference on [W] (Lemma 1), obtaining $M_1'' \approx M_2''$. And therefore $M_1' \approx_L M_2'$, proving the desired noninterference result for W.

Now it just remains to prove Lemmas 1 and 2. This is where the real work happens.

8 Proving noninterference for [W]

8.1 Some important lemmas

Lemma 3 *Low arithmetic expressions always evaluate to ordinary integers (without brackets):*

$$\vdash M \wedge \vdash a : L' \wedge L' \sqsubseteq L \Rightarrow \exists n. \langle M, a \rangle \Downarrow n$$

Proof. By structural induction on a.

- Case $a = n$: trivial
- Case $a = x$: By cases on x being low or high.
  - Case x is low ($\Gamma(x) \sqsubseteq L$):
    $M(x) = n$ for some n, because $\vdash M$.
    Therefore $\langle M, a\rangle \Downarrow n$.
  - Case x is high:
    This contradicts $L' \sqsubseteq L$.
- Case $a = a_1 + a_2$: From $\vdash a : L'$, we know $\vdash a_1 : L_1$ and $\vdash a_2 : L_2$ where $L' = L_1 \sqcup L_2$.

  By the induction hypothesis, there exist n_1, n_2 such that $\langle M, a_1\rangle \Downarrow n_1$ and $\langle M, a_2\rangle \Downarrow n_2$.

  Therefore $\langle M, a\rangle \Downarrow n$ where $n = n_1 + n_2$.

Lemma 4 *Low boolean expressions always evaluate to ordinary truth values (without brackets):*

$$\vdash M \wedge \vdash b : L' \wedge L' \sqsubseteq L \Rightarrow \exists t.\langle M, b\rangle \Downarrow t$$

Proof. By structural induction on b, similarly to the proof of Lemma 3.

8.2 Subsumption

Lemma 5 (PC Subsumption) *If a command can be typed under a given program counter label pc, it can also be typed under a lower label pc':*

$$pc \vdash c \wedge pc' \sqsubseteq pc \implies pc' \vdash c$$

Proof. By rule induction on the typing derivation $pc \vdash c$.

- case $pc \vdash$ **skip**: Trivial, since $pc' \vdash$ **skip** for any pc'.
- case $pc \vdash x := a$:

  From typing rule, have: $\vdash a : L'$ and $L' \sqsubseteq \Gamma(x)$ and $pc \sqsubseteq \Gamma(x)$. Since $pc' \sqsubseteq pc$, we have $pc' \sqsubseteq \Gamma(x)$ too, so $pc' \vdash x := a$.
- case $pc \vdash c_1; c_2$:

  From typing, have $pc \vdash c_1$ and $pc \vdash c_2$.

  From induction hypothesis, $pc' \vdash c_1$ and $pc' \vdash c_2$. Therefore we can derive $pc' \vdash c_1; c_2$.

- case $pc \vdash$ **if** b **then** c_1 **else** c_2:

  From typing, have $\vdash b : L'$ and $pc \sqcup L' \vdash c_1$ and $pc \sqcup L' \vdash c_2$.

  Since $pc' \sqcup L' \sqsubseteq pc \sqcup L$, the induction hypothesis gives us $pc' \sqcup L' \vdash c_1$ and $pc' \sqcup L' \vdash c_2$. Therefore we can derive $pc' \vdash$ **if** b **then** c_1 **else** c_2.

- case $pc \vdash$ **while** b **do** c':

  From typing, have $\vdash b : L'$ and $pc \sqcup L' \vdash c'$.

  Since $pc' \sqcup L' \sqsubseteq pc \sqcup L$, the induction hypothesis gives us $pc' \sqcup L' \vdash c'$.

  Therefore we can derive $pc' \vdash$ **while** b **do** c'.

8.3 Preservation

We need to know that [W] preserves typing during evaluation. This is also known as subject reduction.

Lemma 6 (Preservation)

$$\vdash M \wedge pc \vdash c \wedge \langle M, c\rangle \longrightarrow \langle M', c'\rangle \implies \vdash M' \wedge pc \vdash c'$$

Proof. By rule induction on the derivation of $\langle M, c\rangle \longrightarrow \langle M', c'\rangle$.

- case $\langle M, [c]\rangle \longrightarrow \langle M[c']\rangle$. (augmented rule)

  From the evaluation rule, we have $\langle M, c\rangle \longrightarrow \langle M', c'\rangle$.

  From the typing rule, there is some high L' such that $L' \vdash c$ and $pc \sqsubseteq L'$.

  We can assume the induction hypothesis, so $L' \vdash c'$ and $\vdash M'$.

  Therefore we can derive $pc \vdash [c']$.

- case $\langle M, [\textbf{skip}]\rangle \longrightarrow \langle M, \textbf{skip}\rangle$.

  This case is trivial because $pc \vdash$ **skip** and we already have $\vdash M$.

- case $\langle M, x := a\rangle \longrightarrow \langle M[x := n], \textbf{skip}\rangle$.

  We have $pc \vdash$ **skip** trivially, but also need to show $\vdash M[x := n]$. The variable x must be low for this evaluation rule to apply, so after changing its mapping in M to an ordinary integer, the memory will still be well-formed.

- case $\langle M, x := a\rangle \longrightarrow \langle M[x := [n]], \textbf{skip}\rangle$. (augmented rule)

  From the typing derivation, we know $\vdash a : L'$ where $L' \sqsubseteq \Gamma(x)$.

  If $\Gamma(x)$ is high, the memory M will still be well-formed after the mapping for x is changed to a bracketed integer.

  And $\Gamma(x)$ cannot be low. If it were, then L' would be low too. But by the low arithmetic lemma, a cannot evaluate to a bracketed integer.

- case $\langle M, \textbf{skip}; c_1\rangle \longrightarrow \langle M, c_1\rangle$.

  We have the required $pc \vdash c_1$ from $pc \vdash$ **skip**; c_1, and of course $\vdash M$ still holds.

- case $\langle M, c_1; c_2\rangle \longrightarrow \langle M', c_1'; c_2\rangle$.

  From evaluation rule, $\langle M, c_1\rangle \longrightarrow \langle M', c_1'\rangle$.

  From typing rule, $pc \vdash c_1$.

  By induction hypothesis, $pc \vdash c_1'$ and $\vdash M'$.

  Therefore we can also derive $pc \vdash c_1'; c_2$ as required.

- cases $\langle M, \textbf{if}\, b\, \textbf{then}\, c_1\, \textbf{else}\, c_2\rangle \longrightarrow \langle M, c_1\rangle$, $\langle M, \textbf{if}\, b\, \textbf{then}\, c_1\, \textbf{else}\, c_2\rangle \longrightarrow \langle M, c_2\rangle$.

  From the evaluation rule, $\langle M, b\rangle \Downarrow \textbf{true}$ (respectively, $\langle M, b\rangle \Downarrow \textbf{false}$).

  From the typing rule, we have $\vdash b : L'$ and $pc \sqcup L' \vdash c_1$ and $pc \sqcup L' \vdash c_2$.

  From Lemma 5, $pc \vdash c_1$ and $pc \vdash c_2$.

  Therefore both branches preserve typing.

- cases $\langle M, \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\rangle \longrightarrow \langle M, [c_1]\rangle$ and $\langle M, \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\rangle \longrightarrow \langle M, [c_2]\rangle$.

  These are the rules that apply when $\langle M, b\rangle \Downarrow [\textbf{true}]$ or $\langle M, b\rangle \Downarrow [\textbf{false}]$ respectively.

  From the typing rule, we have $\vdash b : L'$ and $pc \sqcup L' \vdash c_1$ and $pc \sqcup L' \vdash c_2$.

  From Lemma 4, L' must be high, so $pc \sqcup L'$ is also high. Also, $pc \sqsubseteq pc \sqcup L'$. Therefore, $pc \vdash [c_1]$ and $pc \vdash [c_2]$, as required.

- case $\langle M, \textbf{while}\ b\ \textbf{do}\ c'\rangle \longrightarrow \langle M, \textbf{if}\ b\ \textbf{then}\ (c; \textbf{while}\ b\ \textbf{do}\ c)\ \textbf{else skip}\rangle$:

  A typing derivation for the right-hand side must look like the following:

$$\dfrac{\dfrac{A}{\vdash b : L'} \quad \dfrac{\dfrac{B}{pc \sqcup L' \vdash c} \quad \dfrac{\dfrac{A}{\vdash b : L'} \quad \dfrac{B}{pc \sqcup L' \sqcup L' \vdash c}}{pc \sqcup L' \vdash \textbf{while}\ b\ \textbf{do}\ c}}{pc \sqcup L' \vdash c; \textbf{while}\ b\ \textbf{do}\ c} \quad \dfrac{}{pc \sqcup L' \vdash \textbf{skip}}}{\textbf{if}\ b\ \textbf{then}\ c'; \textbf{while}\ b\ \textbf{do}\ c'\ \textbf{else skip}}$$

  But we have derivations A and B by the typing rule for **while** b **do** c'.

Lemma 7 (High-step lemma) *A step of a command typable with high pc results in an equivalent memory.*

$$pc \not\sqsubseteq L \wedge pc \vdash c \wedge \langle M, c\rangle \longrightarrow \langle M', c'\rangle \Rightarrow M \approx M'$$

Proof. By rule induction on $\langle M, c\rangle \longrightarrow \langle M', c'\rangle$.

- cases $\langle M, [\textbf{skip}]\rangle \longrightarrow \langle M, \textbf{skip}\rangle$, $\langle M, \textbf{skip}; c'\rangle \longrightarrow \langle M, c'\rangle$, $\langle M, \textbf{if}\, b\, \textbf{then}\, c_1\, \textbf{else}\, c_2\rangle \longrightarrow \langle M, c\rangle$, $\langle M, \textbf{if}\ b\ \textbf{then}\ c_3\ \textbf{else}\ c_4\rangle \longrightarrow \langle M, c\rangle$, $\langle M, \textbf{if}\ b\ \textbf{then}\ c_3\ \textbf{else}\ c_4\rangle \longrightarrow \langle M, [c_3]\rangle$ and $\langle M, \textbf{if}\ b\ \textbf{then}\ c_3\ \textbf{else}\ c_4\rangle \longrightarrow \langle M, [c_4]\rangle$, $\langle M, \textbf{while}\ b\ \textbf{do}\ c_1\rangle \longrightarrow \langle M, \textbf{if}\ b\ \textbf{then}\ (c_1; \textbf{while}\ b\ \textbf{do}\ c_1)\ \textbf{else skip}\rangle$:

  Trivial, because $M' = M$.

- case $\langle M, x := a_1\rangle \longrightarrow \langle M[x := n], \textbf{skip}\rangle$:

  By the typing rule, $pc \sqsubseteq \Gamma(x)$, so x is high ($\Gamma(x) \not\sqsubseteq L$). So this case cannot happen.

- case $\langle M, x := a_1\rangle \longrightarrow \langle M[x := [n]], \textbf{skip}\rangle$:

  This case only happens when x is high, so clearly $M \approx M[x := [n]]$.

- case $\langle M, c_1; c_2\rangle \longrightarrow \langle M', c_1'; c_2\rangle$.

  From the evaluation rule, $\langle M, c_1\rangle \longrightarrow \langle M', c_1'\rangle$.

  Therefore $M \approx M'$ by the induction hypothesis.

- case $\langle M, [c_1]\rangle \longrightarrow \langle M', [c_1']\rangle$:

  From the evaluation rule, $\langle M, c_1\rangle \longrightarrow \langle M', c_1'\rangle$.

  From the typing rule, $L' \vdash c_1$ where L' is high.

  Therefore, $M \approx M'$, by the induction hypothesis.

8.4 Simple expression equivalence lemmas

Lemma 8 (Arithmetic preserves equivalence)

$$a_1 \approx a_2 \wedge M_1 \approx M_2 \wedge \langle M_1, a_1\rangle \Downarrow a \Rightarrow \exists a'.\langle M_2, a_2\rangle \Downarrow a' \wedge a \approx a'$$

Lemma 9 (Boolean evaluation preserves equivalence)

$$b_1 \approx b_2 \wedge M_1 \approx M_2 \wedge \langle M_1, b_1\rangle \Downarrow b \Rightarrow \exists b'.\langle M_2, b_2\rangle \Downarrow b' \wedge b \approx b'$$

These two lemmas are proved in the same way. We show just the arithmetic one.

Proof. By rule induction on $\langle M_1, a_1\rangle \Downarrow a$.

- case $\langle M_1, n_1\rangle \Downarrow n_1$. Then $a_2 = n_1$, and $\langle M_2, a_2\rangle \Downarrow n_1$.
- case $\langle M_1, x\rangle \Downarrow n_1$. Then $M_1(x) = n_1$, so $M_2(x) = n_1$, so $\langle M_2, x\rangle \Downarrow n_1$.
- case $\langle M_1, x\rangle \Downarrow [n_1]$. Then $M_2(x) = [n_2]$ for some n_2, and $\langle M_2, x\rangle \Downarrow [n_2]$. But $[n_1] \approx [n_2]$.
- case $\langle M_1, a_1' + a_1''\rangle \Downarrow a$. Then a_2 is $a_2' + a_2''$ where $a_1' \approx a_2'$ and $a_2' \approx a_2''$. From the evaluation rule, $\langle M_1, a_1'\rangle \Downarrow a_3'$ and $\langle M_1, a_1''\rangle \Downarrow a_3''$.

  By the induction hypothesis, $\langle M_2, a_2'\rangle \Downarrow a_4'$ where $a_3' \approx a_4'$ and $\langle M_2, a_2''\rangle \Downarrow a_4''$ where $a_3'' \approx a_4''$.

  Consider cases on a_3' and a_3''.

  - case $a_3' = n_1'$ and $a_3'' = n_1''$. Then a_1 evaluates to their sum, but $a_4' = n_1'$ and $a_4'' = n_1''$, so a_2 does too.
  - case $a_3' = [n_1']$ and $a_3'' = n_1''$, or $a_3' = n_1'$ and $a_3'' = [n_1'']$, or $a_3' = [n_1']$ and $a_3'' = [n_1'']$. In all these cases, $a = [n]$ where $n = n_1' + n_1''$, and so does a'.
- case $\langle M_1, [n]\rangle \Downarrow [n]$. From $a_1 \approx a_2$, we know $a_2 = [n']$. Then $\langle M_2, a_2\rangle \Downarrow [n']$, and $[n] \approx [n']$ as required.

8.5 Unwinding lemma

The noninterference theorem on [W] (Lemma 1) talks about an evaluation taking an arbitrary number of steps. We prove it by induction on the number of steps taken on the left side. We need an *unwinding lemma* [GM84] to show that equivalence and well-formedness are preserved in each step. More precisely, for each step taken by one of the two equivalent programs, either the other program can take some number of steps to reach an equivalent state, or it diverges ($\langle M_2, c_2\rangle \Uparrow$) inside a bracket:

Lemma 10 (Unwinding)

$$\begin{array}{c} \vdash M_1 \wedge \vdash M_2 \wedge \vdash c_1 \wedge \vdash c_2 \\ \wedge\, M_1 \approx M_2 \wedge c_1 \approx c_2 \wedge \langle M_1, c_1\rangle \longrightarrow \langle M_1', c_1'\rangle \\ \Longrightarrow \\ (\exists M_2', c_2'. \langle M_2, c_2\rangle \longrightarrow^* \langle M_2', c_2'\rangle \wedge M_1' \approx M_2' \wedge c_1' \approx c_2') \\ \vee\, (\langle M_2, c_2\rangle \Uparrow \wedge \exists c.\ c_2 = [c]) \end{array}$$

Notice that we don't include

$$\vdash c_1' \wedge \vdash c_2' \wedge \vdash M_1' \wedge \vdash M_2'$$

in the consequent, even though we'll need those to be able to string instances of the unwinding lemma together. Fortunately, the Preservation lemma gives us these parts of the invariant already.

Proof. By rule induction on $\langle M_1, c_1\rangle \longrightarrow \langle M_1', c_1'\rangle$.

- case $\langle M_1, [c_3]\rangle \longrightarrow \langle M_1', [c_3']\rangle$.

  From $c_1 \approx c_2$, we know $c_2 = [c_4]$. And $[c_3'] \approx [c_4]$, so $c_2 \approx [c_3']$. From the evaluation rule, we know $\langle M_1, c_3\rangle \longrightarrow \langle M_1', c_3'\rangle$. From the typing rule, we know $L' \vdash c_3$ for high L'. Therefore, from the high-step lemma, we have $M_1' \approx M_1$. Therefore we choose $\langle M_2', c_2'\rangle = \langle M_2, c_2\rangle \approx \langle M_1', [c_3']\rangle$.

- case $\langle M_1, [\textbf{skip}]\rangle \longrightarrow \langle M_1, \textbf{skip}\rangle$.

  From $c_1 \approx c_2$, we know $c_2 = [c_4]$. If c_2 diverges, we are done with $c = c_4$. Let us consider the case that c_2 does not diverge. The only way for c_2 not to diverge is for c_4 to evaluate to **skip**, in which case we have $\langle M_2, [c_4]\rangle \longrightarrow^* \langle M_2', [\textbf{skip}]\rangle \longrightarrow \langle M_2', \textbf{skip}\rangle$. By the high-step lemma (and by induction on the number of evaluation steps, $M_2' \approx M_2 \approx M_1$. So we can choose $c_2' = \textbf{skip}$.

- case $\langle M_1, x := a_1\rangle \longrightarrow \langle M_1[x := n], \textbf{skip}\rangle$:

  From $c_1 \approx c_2$, we know c_2 is $x := a_2$ for some a_2 where $a_1 \approx a_2$.

  From the evaluation rule we know $\langle M, a_1\rangle \Downarrow n$.

  By Lemma 8, we have $\langle M, a_2\rangle \Downarrow n$.

  If $M_1 \approx M_2$, then $M_1[x := n] \approx M_2[x := n]$.

- case $\langle M_1, x := a_1\rangle \longrightarrow \langle M[x := [n]], \textbf{skip}\rangle$:

  This case is proved the same way as the previous one, using Lemma 8.

- case $\langle M_1, \textbf{skip}; c_1'\rangle \longrightarrow \langle M_1, c_1'\rangle$.

  Command c_2 must also have form $\textbf{skip}; c_2'$ where $c_1' \approx c_2'$. So $\langle M_2, c_2\rangle \longrightarrow \langle M_2, c_2'\rangle$, preserving equivalence on memories and commands as required.

- case $\langle M_1, c_3; c_4\rangle \longrightarrow \langle M_1', c_3'; c_4\rangle$. Command c_2 must have form $c_5; c_6$ where $c_3 \approx c_5$ and $c_4 \approx c_6$. From the evaluation rule we have $\langle M_1, c_3\rangle \longrightarrow \langle M_1', c_3'\rangle$. By the induction hypothesis, we know from $c_3 \approx c_5$ that $\langle M_2, c_5\rangle \longrightarrow \langle M_2', c_5'\rangle$ where $M_1' \approx M_2'$. From the evaluation rule, $\langle M_2, c_5; c_6\rangle \longrightarrow \langle M_2', c_5'; c_6\rangle$. So the case is satisfied with $c_2' = c_5'; c_6$.

- cases $\langle M_1, \textbf{if}\ b\ \textbf{then}\ c_3\ \textbf{else}\ c_4\rangle \longrightarrow \langle M_1, c_3\rangle$ and $\langle M_1, \textbf{if}\ b\ \textbf{then}\ c_3\ \textbf{else}\ c_4\rangle \longrightarrow \langle M_1, c_4\rangle$.

  From the equivalence $c_1 \approx c_2$, we know c_2 has the form $\textbf{if}\, b'\, \textbf{then}\, c_5\, \textbf{else}\, c_6$ where $b \approx b'$, $c_3 \approx c_5$, and $c_4 \approx c_6$. Since b and b' can't be bracketed, they must be the same.

  We know that either $\langle M_1, b\rangle \Downarrow t$ (where t is either **true** or **false**), or $\langle M_1, b\rangle \Downarrow [t]$. Actually, we can't get a bracketed value as a result because then we'd use a different evaluation rule for **if** (the next one). But if $\langle M_1, b\rangle \Downarrow t$, then by Lemma 9, b' must evaluate to the same value t. Therefore c_1 and c_2 take the same branch. From $c_3 \approx c_5$ and $c_4 \approx c_6$, we know that in either case $c_1' \approx c_2'$.

- cases $\langle M_1, \textbf{if}\ b\ \textbf{then}\ c_3\ \textbf{else}\ c_4\rangle \longrightarrow \langle M_1, [c_3]\rangle$ and $\langle M_1, \textbf{if}\ b\ \textbf{then}\ c_3\ \textbf{else}\ c_4\rangle \longrightarrow \langle M_1, [c_4]\rangle$.

  This rule is used when $\langle M_1, b\rangle \Downarrow [t]$. As in the previous case, we know c_2 has the form $\textbf{if}\ b\ \textbf{then}\ c_5\ \textbf{else}\ c_6$ where $c_3 \approx c_5$, and $c_4 \approx c_6$. By Lemma 9, we know that $\langle M_2, b\rangle \Downarrow [t']$. Depending on t', either $\langle M_2, c_2\rangle \longrightarrow \langle M_2, [c_5]\rangle$ or $\langle M_2, c_2\rangle \longrightarrow \langle M_2, [c_6]\rangle$. That is, c_1' is either $[c_3]$ or $[c_4]$, and c_2' is either $[c_5]$ or $[c_6]$. In any case we have $c_1' \approx c_2'$.

- case $\langle M, \textbf{while}\ b\ \textbf{do}\ c_3\rangle \longrightarrow \langle M, \textbf{if}\ b\ \textbf{then}\ (c_3; \textbf{while}\ b\ \textbf{do}\ c_3)\ \textbf{else skip}\rangle$:

  We have $c_2 = \langle M_2, \textbf{while}b\textbf{do}c_4\rangle$ where $c_3 \approx c_4$. Therefore $c_2' = \textbf{if}b\textbf{then}(c_4; \textbf{while}b\textbf{do}c_4)\textbf{elseskip}$, and $M_2' = M_2$. Clearly $c_1' \approx c_2'$.

8.6 Acknowledgments

Aslan Askarov, Danfeng Zhang, and Éric Tanter gave helpful feedback and caught some bugs.

References

[GM82] Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20, April 1982.

[GM84] Joseph A. Goguen and Jose Meseguer. Unwinding and inference control. In *Proc. IEEE Symposium on Security and Privacy*, pages 75–86, April 1984.

[PS03] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1), January 2003.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 3rd edition, 1993.

[ZM02] Steve Zdancewic and Andrew C. Myers. Secure information flow via linear continuations. *Higher Order and Symbolic Computation*, 15(2–3):209–234, September 2002.