Compositional noninterference

from first principles

Carroll Morgan

Abstract. The recently invented Shadow Semantics for noninterference-style security of sequential programs avoids the Refinement Paradox by inhibiting the reduction of demonic nondeterminism in those cases where it would compromise security.

The construction (originally) of the semantic domain for The Shadow, and the interpretation of programs in it, relied heavily on intuition, guesswork and the advice of others. That being so, it is natural after the fact to try to reconstruct an idealised “inevitable” path from first principles to where we actually ended up: not only does one learn (more) about semantic principles by doing so, but the “rational reconstruction” helps to expose the choices there were and to legitimise the decisions that resolved them.

Unlike our other papers on noninterference, this one does not contain a significant case study: instead its aim is to provide the most accessible account we can of the methods we use and why our model, in its details, has turned out the way it has. In passing, it might give some insight into the general role and significance of compositionality and testing-with-context for program semantics.

Finally, a technical contribution here is a new “Transfer Principle” that captures uniformly a large class of classical refinements that remain valid when noninterference is taken into account.

Keywords: Security, refinement, noninterference, Refinement Paradox, compositionality, testing semantics.

1. Introduction

Noninterference analysis partitions a program state into high- and low-security portions, and determines to what extent high-security values can be deduced from low-security observations [7]. It is attractive to combine that with refinement, the reduction of demonic nondeterminism that allows program development to proceed through layers of abstraction. But that is impeded by the so-called Refinement Paradox [12] — the fact that reduction of demonic nondeterminism can turn a secure program into an insecure one.

For example, let \( v, h \) be variables of type \( \{0, 1\} \), and let the first be low-security while the second is high-security; suppose \( v : \in \{0, 1\} \) is a command that assigns to \( v \) a value chosen demonically from \( \{0, 1\} \). Classical refinement allows that command to be refined to \( v := h \); but “secure refinement” (to be defined) probably should not allow it: the first command \( v : \in \{0, 1\} \) does at least look secure, since it does not refer
to \( h \); yet the second command \( v := h \) does not look secure at all. Indeed it could hardly reveal the hidden \( h \) more explicitly.

In earlier work the *Shadow Semantics* [18, 20] proposed a compromise, a solution to the Paradox in which nondeterminism associated with low-security variables could after all be reduced, but only if it did not consequentially reduce nondeterminism of high-security variables at the same time. Thus \( v : \in \{0, 1\} \) could be refined to \( v := 0 \), because that reduction of nondeterminism in the visible \( v \) has no implications for an observer’s knowledge of the hidden \( h \). But our problematic refinement above, to \( v := h \), would *not* be allowed because an observer of the low-security \( v \) could deduce a consequential reduction in the nondeterminism of \( h \) as well.

In this report we give a rational reconstruction of how The Shadow achieved this compromise, appealing as much as possible to general techniques of semantics, refinement and compositionality as we can. Our aim is both to provide specifically a compelling account of how noninterference and refinement can be reconciled and at the same time, by proceeding “from first principles,” to illustrate how general techniques play out in practice. Part of our contribution here is this new presentation, since earlier descriptions of The Shadow have been much more ad hoc.

Our (first) principal technical contribution in this report is the identification of an Elementary-Testing Order for noninterference-style security, and the proof that the secure refinement we define is its compositional closure (§7.3) in the semantics we choose: this elementary testing will help us conceptually to get from classical- to secure refinement. Our second technical contribution is the formulation and proof of a Transfer Principle (§10) which characterises uniformly a large number of classical refinements that are also secure refinements.

**Notation** Throughout we write function application with a dot, that is \( f.x \) rather than \( f(x) \), since it reduces the clutter of parentheses, and we use the *Eindhoven Style* of comprehension–quantifier, bound variable(s) and their type(s), range, expression– because of its uniformity and its nice calculational properties.\(^1\)

Program variables like \( v, h \) will be written in *Sans Serif* to distinguish them from the values like \( v, h \), possibly with decorations, that those variables might assume. We refer to low-security variables as *visible* variables, typically \( v, w \), and high-security variables will be *hidden*, typically \( h \).

Binary-relation symbols, when they appear without all their operands, are written between parentheses (·) in the style of the sections of functional programming. Conventional refinement we call *classical*, and write it –unconventionally– as \( (\leq) \) to distinguish it from the novel *secure refinement* order, which we will write as \( (\subseteq) \).

### 2. Refinement is adversarial

Design-time refinement is potentially an attack: indeed the classical refinement \( v : \in \{0, 1\} \leq v := h \) could not make this clearer. Our only protection against colleagues’ well meaning refinements –our unwitting adversaries– is a careful formulation of secure refinement so that \( v : \in \{0, 1\} \nsubseteq v := h \), in spite of its being allowed classically as above. This example is after all the essence of the Refinement Paradox, solved by recognising that there must be some programs \( S, I \) for which \( S \leq I \) but \( S \nsubseteq I \).

In a spirit of informal reconnaissance, we use this design-time adversarial view to look for properties that secure refinement (\( \subseteq \)) is *likely* to have, without claiming any rigour in the process (yet). General principles of semantics (i.e. theory) and software development (i.e. practice) will play a role as we go along.

Throughout this section we assume all variables to have type \( \{0, 1\} \).

### 2.1. Monotonicity

Let \( C(\cdot) \) be a context in our programming language (yet to be defined), so that \( C(X) \) is program \( X \) placed in that context. Monotonicity of (classical) refinement is the property that \( S \leq I \) implies \( C(S) \leq C(I) \) for all contexts, and it is what allows the stepwise refinement in which a program is correctly developed globally via many local steps [25]. For (the same) practical reasons, secure refinement should have that property too: we adopt the

---

\(^1\) For example, given sets \( X, Y \) and function \( f : X \rightarrow Y \) the set \( \{ x.X; y.Y \mid y = f.x \cdot \{x'.X \mid y = f.x'\} \} \) is the maximal partition of \( X \) into non-empty subsets that agree on \( f \).
Principle of monotonicity If $S \sqsubseteq I$ then for all contexts $C(\cdot)$ we have $C(S) \sqsubseteq C(I)$.

Naturally the force of this property depends on the contexts that are allowed: the more powerful our programming language –from which $C$ is constructed– the more demanding Monotonicity will be.

2.2. Program flow and atomicity

In classical program development a demonic nondeterministic choice is written $X \sqcap Y$, acting between two program fragments $X, Y$. It expresses a design-time abstraction, that we do not care which of $X$ or $Y$ is chosen for further development; and it expresses equally well an opportunity for run-time nondeterminism, where an implementation may choose between executing $X$ or executing $Y$, even possibly making different choices on different occasions. In both cases the nondeterminism is considered to be “demonic” in the sense that as careful designers we must expect the worst: if we say we don’t care, we must accept what we get.

That $X \sqcap Y$ can be refined by $X$, or by $Y$, is so well known and widely accepted that we expect it to hold in secure development too. Consider therefore the choice $h := 0 \sqcap h := 1$ between $h := 0$ and $h := 1$. From the above we must expect it, as a specification, to be refined both by implementation $h := 0$ and by implementation $h := 1$; we take $h := 0$ for our example, so that what we are considering is the secure refinement
\[ h := 0 \sqcap h := 1 \sqsubseteq h := 0 \ . \] (1)

Now implementation $h := 0$ allows an observer to deduce that the final value $h'$ of $h$ is 0, and he can do that simply by looking at the source code. But specification $h := 0 \sqcap h := 1$ does not seem to allow that deduction, since its $h'$ finally can be 0 or 1.

How then can (1) be a secure refinement, given that variable $h$ is supposed to be hidden and yet more can be deduced on the right than on the left? (2)

In order to legitimise Refinement (1), we are forced to revise our understanding of $h := 0 \sqcap h := 1$, treating it as if its program-flow were visible. That is, we imagine running it and we pretend that we can observe the resolution of its demonic choice ($\sqcap$). When it resolves to the left, we know that $h$ is about to be assigned 0 and that finally $h'$ will be 0. We adopt that as a feature of our approach, the

Principle of Implicit Flow In the presence of refinement, a program’s noninterference properties should be evaluated as if the program-flow were visible.

This compromise answers our question at (2) above: the example (1) can be (indeed is) a secure refinement because, although the implementation $h := 0$ reveals that $h'$ is 0, the specification $h := 0 \sqcap h := 1$ is no better: it might reveal the same. (Also, on any particular run, it might not — but since the choice is demonic, the mere possibility is enough.)

It is in fact another of refinement’s adversarial properties. We are not supposing that the run-time environment actually does contain an adversary that can observe program flow, although it might — rather we are saying that if we wish to use a secure-refinement relation with reasonable properties then we must adopt Implicit Flow and act as if there were such an adversary.

Having introduced the observation of program flow, we are forced to consider atomicity, since otherwise we have an infinite regress: can we see the movement of quarks? We will prevent the regress by arranging that expression evaluation is atomic.

2.3. Referential transparency, extension and perfect recall

We begin by noting that we have the classical refinement $v \in\{0,1\} \leq w$ even though the expression $\{0,1\}$ suggests that perhaps $v := 0$ and $v := 1$ should be the only possibilities. That in fact they are not the only ones is due to the

Principle of Referential Transparency Expressions equal in context can be exchanged without effect.

No matter what $w$‘s value might be, the two sets $\{0,1\}$ and $\{w, \neg w\}$ are equal; and so Referential Transparency makes the statements $v \in \{0,1\}$ and $v \in \{w, \neg w\}$ equal also. With the choice written in the latter form, we can see $v := w$ and $v := \neg w$ as obvious further possibilities for refinement.
Having to accept this classical refinement \( v \in \{0, 1\} \leq v := w \), we should expect the secure refinement \( v \in \{0, 1\} \sqsubseteq v := w \) as well, from the

**Principle of Extension** If programs \( S, I \) contain only visible variables, then \( S \leq I \) implies \( S \sqsubseteq I \).

The practical reason for Extension is that one designer, having already established a classical refinement in his part of a system, does not expect that refinement to be thrown into doubt simply because security is being considered by another designer in a different part. Putting it another way, we could say that we expect designers to engage in the more intricate security-refinement reasoning only when hidden variables are actually present in the code before them. Otherwise they should be able to continue refining (classically) as they have always done.

Now consider the specification \( w := h; v \in \{0, 1\}; w := 0 \) in which hidden \( h \) is copied to visible \( w \) but is later overwritten by \( w := 0 \). The intermediate statement \( v \in \{0, 1\} \) does not refer to hidden variables, and so by Extension it can be securely refined to \( v := w \). By Monotonicity that would refine the whole fragment to an implementation \( w := h; v := w; w := 0 \), and now \( h \) is revealed after all in the final value of \( v \) — in spite of the fact that, in the specification, variable \( v \) does not interact with \( h \) or \( w \) at all. Thus the assignment \( w := h \) must be considered to be the culprit, and we have the

**Principle of Perfect Recall** In the presence of refinement, a program’s noninterference properties should be evaluated as if the values of visible variables can be observed even if subsequently overwritten.

Again this is an adversarial property of refinement. And again, we are not supposing that the run-time environment actually does contain an adversary with perfect recall, although it might — rather we are saying that if we wish to use a secure refinement relation with reasonable properties then we must act as if there were such an adversary.  

2.4. Summary

We stress that the above remarks are informal, as they must be since we have not introduced a secure semantics in this presentation (yet); and so all of our conclusions are only opinions. Other decisions would lead to other “principles” and consequentially to other models of non-interference style security than the one we are about to explain.

What we are illustrating however is the value of an initial reconnaissance, an exploration of the structural attributes our semantics should have if we are going to be satisfied with it given our refinement-oriented point of view. Not only does this help to avoid later disappointment (and much wasted work); but it will guide the construction of the semantics itself.

3. The elementary-testing order

3.1. The definition of elementary testing is by negotiation

When we try to formulate convincing criteria for whether some specification \( S \) is implemented by some \( I \), we aim to agree with our clients on the tests that can be applied to \( S \) and to \( I \) — we agree in fact on the circumstances in which a client would be entitled to reject an implementation \( I \) of the specification \( S \), the very \( S \) we presumably had adopted jointly as part of a contract. Once such criteria have been negotiated, we must accept that if such a test fails for our \( I \) but it can be shown that it would always have succeeded for his \( S \), then the contract has been broken — and we can expect to see him in court.

An example is what we will call an elementary classical test: say that a program \( X \) has exhibited \( v = v' \) if the final value of \( v \) after \( X \) has run is explicitly observed to have that value \( v' \). For example, a client suspicious of our handling of his specification \( v := 1+1 \) might try to catch the implementation exhibiting \( v' = 3 \) — he is trying to find a \( v' \) such that \( I \) can exhibit \( v' \) but \( S \) never can. This simple test leads ultimately to the

---

2 It does mean though that our implementations can be used in run-time environments where such adversaries are actually present. For example, if we describe a distributed implementation with a sequential program, then Perfect Recall and Implicit Flow are real, run-time risks caused by the observations possible for individual agents.

To extend these notions to noninterference, we propose the following elementary test for security. Say that \( X \) has \textit{leaked} \( h = h' \) if, by observation of program flow and all intermediate values assumed by visible variables (even if subsequently overwitten), it can be \textit{deduced} that the final value of \( h \) has that value \( h' \).

Note that we are not allowed to observe the variable \( h \) directly; instead we use our indirect observations — and of course the source code of \( I \), since otherwise the intermediate values and implicit flows would make no sense.

For example, a client suspicious of our handling of his specification \( h : \in \{ 0..9 \} \) might try to catch the implementation leaking \( h' = 9 \) — and he would succeed if our implementation contained the code \( v := h \div 9 \) and, on a particular run, the nondeterminism in \( h : \in \{ 0..9 \} \) resolved to \( h := 9 \) — for then \( v \) will be observed to be 1 (even if subsequently overwritten), and the only value of \( h \) that can cause that is 9.

We now define the elementary-testing order for noninterference to be the combination of the two tests above:

\[ S \mathrel{\triangleleft} I \text{ if } \forall I, C \in \mathcal{C}(S) \mathrel{\triangleright} C(I). \]

\[ S \mathrel{\triangleright} I \text{ if } \forall S, C \in \mathcal{C}(S) \mathrel{\triangleleft} C(I). \]

**Definition 1. Elementary-Testing Order for noninterference** We say that \( S \mathrel{\triangleleft} I \), that \( S \) and \( I \) are in the elementary-testing order (\( \triangleleft \)) for noninterference, just when from some initial state

—\textit{functional testing} If implementation \( I \) can exhibit \( v = v' \) by direct observation, then so can its specification \( S \).

—\textit{security testing} If implementation \( I \) can leak \( h = h' \) by indirect observation and deduction, then so can its specification \( S \).

We stress that the point of proposing and negotiating this elementary-testing order is for us as designers to be sure to satisfy our customers. The refinement order (yet to be determined) will be \textit{our} tool for that; the elementary-testing order is \textit{their} tool for judging our efforts. With that in mind, we want to choose a refinement order to satisfy the

**Principle of soundness** If \( S \mathrel{\triangleleft} I \) then for all contexts \( C \) we have \( C(S) \mathrel{\triangleleft} C(I) \).

Thus, with \textit{Soundness}, if a designer achieves \( S \mathrel{\triangleleft} I \) then there can be no context \( C \) such that \( C(I) \) will disappoint a customer who was expecting behaviour conforming to \( C(S) \).

### 3.2. Examples of the elementary-testing order

The most obvious elementary-testing-order violation is \( h : \in \{ 0, 1 \} \not\leq h := 0 \), because on the right \( h \)'s final value 0 is leaked by inspection of the source code, without even running the program, but on the left we can never be sure of \( h \)'s final value exactly even if we do run the program, since we're not allowed to look at \( h \) directly in the final state. \textit{Implicit Flow} does not help, because the choice of the assignment to \( h \) is done atomically on the left: there is no non-trivial program flow to give anything away. That distinguishes it from our earlier example \( h := 0 \cap h := 1 \) where the program flow reveals which branch of the choice \( (\cap) \) is taken, and hence by deduction which value \( h \) will receive. Thus we can note — in passing — that \( h : \in \{ 0, 1 \} \) and \( h := 0 \cap h := 1 \) are different.

On the other hand, we do have \( v : \in \{ 0, 1 \} \leq v := 0 \), since the only final value produced on the right, that is \( v' := 0 \), is one which the left could indeed produce as well. In fact \( v : \in \{ 0, 1 \} \) and \( v := 0 \cap v := 1 \) are equal: even though atomicity disables implicit-flow deductions just as above, an attacker doesn’t need them in this case because \( v \)'s value is directly observable anyway once the program fragment is done.

More interesting is the case where \( S \) is \( h : \in \{ 0, 1, 2 \} \) and \( I \) is \( h : \in \{ 0, 1 \} \). Here we have \( S \mathrel{\triangleleft} I \) because \( h \)'s final value cannot be deduced exactly in either case. On the other hand, if we define context \( C \) to be \( (\vdash h := h \div 2) \), then we have \( C(S) \not\leq C(I) \) because \( h' \) can be deduced on the right (must be 0) but not on the left (could be 0 or 1). Thus from \textit{Soundness} we know that in spite of \( S \mathrel{\triangleleft} I \) we must accept \( S \not\leq I \), and this refinement order — once we figure out what it is — will protect us from the naïve belief that we might have implemented \( h : \in \{ 0, 1, 2 \} \) by \( h : \in \{ 0, 1 \} \) simply because of their \((\leq)\)-relationship. After all, our customer could have context \( (\vdash h := h \div 2) \) waiting for us, and we might know nothing about it.
4. The execution model: sequences of program-states and their equivalence

The examples of §2 suggested that to deal with refinement we must assume a run-time adversary with both perfect recall and access to implicit flow. We now use that intuition (for that is all it was) to begin to build our security model.

We imagine that an attacker executes our program and strives to deduce the final value of the hidden variable h. In doing so he uses a run-time debugger, setting breakpoints and single-stepping as he wishes: whenever the program is paused for inspection, the exact point in the source code is indicated, and thus as well as the names of all variables in scope (including the hidden ones). We make these two points:

- The atomicity of the program commands determines the locations at which breakpoints can be inserted or (equivalently) the size of single-steps the attacker can instruct the debugger to take. Pauses are allowed only between atoms, and never within them.
- The visibility of the program variables determines whether the debugger reveals variable values for inspection. When execution is paused, hovering the cursor over an in-scope visible-declared variable reveals its value; hovering over a hidden-declared value reveals nothing.

We provide the raw material for modelling these attack-capabilities by recording entire traces of a program’s execution: for simplicity we will have just the two variables v, h. Each trace is then a sequence of triples comprising values of the visible variable v, the hidden variable h and the program counter p. The program’s meaning, based on an initial state (v, h, p), is then a set of (potential) traces of triples, with the multiplicity of traces (if present) being due to nondeterminism in the program text.

The attacker’s capabilities are then expressible in terms of an equivalence relation on these traces, which we define as follows. Suppose the types of v, h, p are V, H, P resp., so that the type of a trace \( t: \mathcal{T} \) is the set of sequences \( \text{seq}(V \times H \times P) \); for such a \( t \), write \( v.t, h.t, p.t \) for the projections onto types \( \text{seq} V, \text{seq} H \) and \( \text{seq} P \) respectively.

Then we say two traces \( t_{1,2} \) are equivalent, writing \( t_1 \sim t_2 \), just when we have both \( v.t_1 = v.t_2 \) and \( p.t_1 = p.t_2 \), that is when they have the same sequence of v’s (since we allow perfect recall) and the same sequence of p’s (since we allow observation of the program flow).\(^3\) Since a trace includes its final element, this implies that equivalent traces have reached the same point in the program (in fact, by the same route), and have the same value \( v’ \) in v at that point. They do not have to have the same sequence of h’s or indeed even the same final h’, since the attacker cannot observe that directly.

The attacker’s capabilities are then that, when the program is paused after some trace \( t \), in fact he does not know \( t \) itself. Instead, he knows the equivalence class \([t]_\sim \) that \( t \) inhabits, that is the set \( \{ t’: \mathcal{T} \mid t \sim t’ \} \). Note that \( t \) itself is in that set: but it is quite likely not the only member, and the bigger the set is the bigger the attacker’s ignorance of what actually has occurred in that particular run.

5. Elementary testing and semantics constrains refinement

Having built our semantic space, we now take our first steps towards finding a (≤)-sound refinement order.

On a particular run, the trace produced will be \( \sim \)-equivalent possibly to a number of others, and the model of §4 has been constructed so that the \( h’ \)-values found at the ends of all those equivalent traces are collectively a set \( H’ \) comprising all and exactly the values that an attacker must concede h could possibly have at the end of this run. We regard this as an extra variable \( H \) called the Shadow of h because, in effect, it records the values of h resulting from \((\leq)\)-outcomes that could have been taken on this run — even if, in fact, this time they weren’t. Now from our elementary-testing order we conclude immediately that if the implementation can produce a singleton shadow \( \{h’\} \), then the specification must be able to produce the same singleton shadow \( \{h’\} \): the presence of the singleton shadow for the implementation means that \( I \) can be caught with \( h’ \); and elementary testing then requires the same to be true of \( S \).

\(^3\) That constructs a very simple Kripke model where the accessibility relation is the equivalence that relates “worlds” indistinguishable to an attacker.
Together with classical elementary testing, this gives us our first tentative definition of secure refinement in total. Specification \( S \) is “tentatively” refined by implementation \( I \), written \( S \sqsubseteq_3 I \), just when

- **classical**— Any final visible \( v' \) that can be produced by \( I \) can also be produced by \( S \) and
- **secure**— Any final singleton shadow \( \{h'\} \) that can be produced by \( I \) can also be produced by \( S \).

We will see in a moment that this is actually not the right definition: it needs to be strengthened. But it is based directly on elementary testing (\( \sqsubseteq \)), and that makes it a good first step.

## 6. Compositionality

**Compositionality** is the property that the meaning of a compound can be deduced from the meanings of its components alone. In some cases it is not easy to achieve: an informal example is as follows [17].

Parents’ eye colour, on its own, cannot accurately predict the distribution of eye colour among their children: some brown-eyed parents may produce one blue-eyed child in four. This means means that if we abstract so severely that equality between people is (just) equality of their eye-colours, then any programming language that includes “having children” as one of its language features is not compositional with respect to that equality.

On the other hand, equality of full genetic profile, though still an abstraction because it ignores phenotype, is an equality for which having children is compositional. If \( M, M' \) are twin brothers and \( F, F' \) twin sisters, then the children of \( M + F \) and the children of \( M' + F' \) will have identical distributions of eye colour (in the limit of very many children) even if \( M, F \) love Italian opera but \( M', F' \) are Goths. The genotype abstracts from taste in music; but it is still sufficiently discriminating for eye-colour.

But full genotype is far too discriminating if all we are concerned with is eye-colour: the ideal situation is one where the equality is as simple as possible while retaining compositionality. For eye-colour, the crucial idea (Mendel) is “dominant” and “recessive” characteristics (*alleles*). Two people are equal just when their eye-colour alleles are the same — and the “operator” for having children is compositional for that.

Monotonicity is related to compositionality. If we define equivalence (\( \equiv \)) to be inter-refinability —so that \( P \equiv Q \) just when \( P \sqsubseteq Q \) and \( Q \sqsubseteq P \)— then monotonicity implies compositionality with respect to that equivalence: we just apply it in both directions. Alternatively we say that (\( \equiv \)) is a congruence with respect to contexts.

Now the problem with our first, tentative definition of refinement (\( \sqsubseteq_3 \)) is that, although it correctly captures our focus on secure refinement (cf. our focus on eye-colour), it turns out to be neither sound nor monotonic — and so it cannot not lead to a compositional equivalence. We now explore that issue via a series of examples; and we fix it.

### 6.1. First example, and consequential adjustment of refinement

Our first example we saw already in §3.2, where we had effectively \( h : \in \{0, 1, 2\} \sqsubseteq_3 h : \in \{0, 1\} \) but

\[
\begin{align*}
\text{h:} & : \in \{0, 1, 2\}; \text{h:= h ÷ 2 \not\in} \text{ h:} : \in \{0, 1\}; \text{h:= h ÷ 2}.
\end{align*}
\]

This shows that (\( \sqsubseteq_3 \)) is not sound; and we fix this by considering not-necessarily-singleton shadows.

Suppose \( I \) produces shadow \( H'_I \), and suppose that there is no shadow \( H'_S \) produced by \( S \) with \( H'_S \sqsubseteq H'_I \).

Define a function \( f \) that takes all of \( H'_I \) to 0 but everything else to 1. Then \( I; h := f.h \) can produce shadow \( \{0\} \) but \( S; h := f.h \) never can, since any shadow \( H'_S \) containing a value \( h' \) with \( f.h' = 0 \) must also contain some (other) \( h'' \) with \( f.h'' = 1 \). Otherwise we’d have \( H'_S \sqsubseteq H'_I \), violating our assumption.

So we adjust our definition of refinement to account for that, getting

- **classical**— Any final visible \( v' \) that can be produced by \( I \) can also be produced by \( S \) and
- **secure**— Any final shadow \( H'_I \) that can be produced by \( I \) can be supported via \( H'_S \sqsubseteq H'_I \) by some \( \text{final shadow} \ H'_S \) produced by \( S \).

We call this refinement relation (\( \sqsubseteq_5 \)). It improves (\( \sqsubseteq_3 \)), and it does solve the problem posed by (4) — but it is still not good enough.
6.2. Second example, and consequential adjustment...

Our second example concerns the programs \( h:=0 \) and \( h\in\{0,1\} \). According to (5), the second is a \((\subseteq_5)\)-refinement of the first: the classical criterion is satisfied because the final \( v' \) is in both cases whatever the initial value \( v \) was; and the security criterion is satisfied because \( \{0\}\subseteq\{0,1\} \). And yet...

Consider the context \((-;v:=h)\), which produces respectively the programs \( h:=0; v:= h \) and \( h\in\{0,1\}; v:= h \). The second of those can produce the output \( v'=1 \), but the first cannot; and this can be detected even by a classical elementary test. Thus \((\subseteq_5)\) isn’t sound either.

Our lesson from this example is that \( h \) is relevant (after all), even though an attacker cannot see it, because it could in a larger context be assigned subsequently to \( v \) which an attacker then could see after all. So we adjust our definition of refinement a second time, now getting

\[
\text{classical } v^- \quad \text{Any final visible } v' \text{ that can be produced by } I \text{ can also be produced by } S,
\]

\[
\text{classical } h^- \quad \text{Any final hidden } h' \text{ that can be produced by } I \text{ can also be produced by } S \quad \text{and}
\]

\[
\text{secure}^- \quad \text{Any final shadow } H'_I \text{ that can be produced by } I \text{ can be supported via } H'_S \subseteq H'_I \text{ by some final shadow } H'_S \text{ produced by } S.
\]

We call this \((\subseteq_6)\). But we must still go further.⁴

6.3. Third example, and...

Our third example concerns \( h\in\{0,1\}; v:= h \) and \( h\in\{0,1\}; v:= 1\not\leftarrow h \). According to \((\subseteq_6)\) they are equal (that is inter-refinable): both produce \( v' \)'s of 0 and 1; both produce \( h' \)'s of 0 and 1; and both produce shadows \( H' \) of \( \{0\} \) and \( \{1\} \). Note in the last case the shadows are singleton, not \( \{0,1\} \) as a casual inspection of the code’s \( h\in\{0,1\} \) might suggest, because the assignment to \( v \) in both cases reveals \( h \). And yet...

In this case the context \((-;v:=v+h)\) distinguishes the two programs, since the first in context produces \( v' \) of 0 or 2, while the second produces only \( v'=1 \). We fix that by coupling the attacker’s potential observations of \( v, h \), giving

\[
\text{classical } v, h^- \quad \text{Any final visible } (v', h') \text{ that can be produced by } I \text{ can also be produced by } S \quad \text{and}
\]

\[
\text{secure}^- \quad \text{Any final shadow } H'_I \text{ that can be produced by } I \text{ can be supported via } H'_S \subseteq H'_I \text{ by some final shadow } H'_S \text{ produced by } S'.
\]

This gives us \((\subseteq_7)\); but we are still not quite there.

6.4. Fourth example...

Our fourth example concerns the programs

\[
(v:= 0; h:= 0 \not\leftarrow 1) \cap (v:= 1; h\in\{0,1\}) \quad \text{and} \quad (v:= 0; h\in\{0,1\}) \cap (v:= 1; h:= 0 \not\leftarrow 1)
\]

where, to reduce clutter, we are abbreviating \( h:= 0 \not\leftarrow 1 \) as \( h:= 0 \not\leftarrow 1 \) — note therefore that this “syntactic-sugared” assignment is not atomic.⁵ According to \((\subseteq_7)\) the two programs are equivalent: both produce \( (v', h') \)'s of all four possibilities from \( V\times H \); and both produce all possible output shadows, thus \( \{0\}, \{1\} \) and \( \{0,1\} \). And yet...

These two are distinguished by the context \((-; \text{if } v=0 \text{ then } h\in\{0,1\} \text{ fi}) \). In the first case the only output shadow is \( \{0,1\} \); but in the second case, both \( \{0\} \) and \( \{1\} \) are still possible as well. This we fix by coupling all three observations together, giving

\[
\text{hybrid } v, h, H^- \quad \text{Any final triple } (v', h', H'_I) \text{ produced by } I \text{ can be supported via a triple } (v', h', H'_S) \text{, produced by } S, \text{ that is with } H'_S \subseteq H'_I.
\]

This version of refinement \((\subseteq_8)\) is the one we choose.

⁴ Our addition of this criterion for \( h \), because of compositionality of sequential composition, separates us from some other approaches [3, 2].

⁵ That is, our syntactic convention is that \( h\in\{0,1\} \) is atomic but \( h:= 0 \not\leftarrow 1 \) is not.
6.5. Summary

In the examples above, the tentative definitions of refinement were shown to be unsound, except for the last one ($\sqsubseteq_8$), by finding contexts that were steadily sneakier. The fix in each case was to restrict each new definition even more than its predecessor, so that in fact we had as relations a sequence of inclusions

$$(\sqsubseteq_3) \supseteq (\sqsubseteq_5) \supseteq (\sqsubseteq_6) \supseteq (\sqsubseteq_7) \supseteq (\sqsubseteq_8) ,$$

with a claim finally that $(\sqsubseteq_8)$ is the definition we want. But what makes us stop there? Why don’t we just take $(\sqsubseteq)$ to be the trivial order (=)? That too would be sound.

These are the questions we address in the next section.

7. Compositional closure

7.1. Completeness of a refinement relation

At the end of the last section we claimed that $(\sqsubseteq_8)$ is the definition we want. That means, first of all, that it must be sound. This is proved not by running out of counter-examples in the style of §6, nor by examining all of the infinitely many possible contexts, but rather by proving that each of the program-language constructors is sound separately, and then concluding inductively that all finite contexts built from those constructs are sound too.

Our problem here, though, is that we still do not have a programming language pinned down, with a semantics: we do not yet know the set of contexts that are possible. To that extent we are still proceeding by intuition, gradually improving our constructions from several directions at once and hoping that they will be consistent when they finally meet.

So let us assume that when we do finally have our programming language interpreted in the semantics of §4, a structural induction will show that indeed $(\sqsubseteq_8)$ is sound. In that spirit of optimism, we call it just $(\sqsubseteq)$ from now on.

But that still leaves the question of why we didn’t choose the trivial (=), evidently sound, to be our refinement relation. The obvious answer is that it would be useless: relating programs only to themselves, it could never be used for development.

We approach this conceptual problem via the

**Principle of completeness** If $S \not\sqsubseteq I$ then for some context $C$ we have $C(S) \not\sqsubseteq C(I)$.

Informally, Completeness means that if we reject a refinement we must have a good reason.

As an example, note that alleles are complete for eye-colour if for any two people $M, M'$ with different eye-colour alleles we can find a single $F$ such that $M+F$ and $M'+F$ can yield different child-distributions of actual eye-colour. Thus, with Completeness, if we reject a proposed implementation because we find $S \not\sqsubseteq I$, then we must be able to find a context $C$ so that $C(I)$ would disappoint our customer if he had been expecting $C(S)$.

Since (=) is not complete, we no longer need to worry about that extreme case.

7.2. Definition of compositional closure

If a refinement relation $(\sqsubseteq)$ is both $(\preceq)$-sound and $(\succeq)$-complete, we say that $(\sqsubseteq)$ is the **compositional closure** of $(\lessgtr)$.\(^6\) Once we fix $(\lessgtr)$ which —recall— is by negotiation, the compositional closure is determined uniquely by the program semantics, and to that extent it is not negotiable. We can see that easily by supposing there are two candidates $(\sqsubseteq_{(a,b)})$ for the closure of $(\lessgtr)$. Then we have

$$S \sqsubseteq_{(a,b)} I \Rightarrow C(S) \lessgtr C(I) \quad \text{for all contexts } C$$

"soundness of $(\sqsubseteq_{(a,b)})$"

$$S \sqsubseteq_{(a,b)} I$$

"completeness of $(\sqsubseteq_{(a,b)})$"

\(^6\) Closure usually suggests adding more elements until some collective condition is achieved (such as inclusion of all limit points) — but here we are taking elements away from $(\lessgtr)$ to reach $(\sqsubseteq)$. In fact we think of the closure to be operating on our ability to **distinguish** programs via tests-in-context, so that it is actually $(\preceq)$ that is being closed to give $(\sqsubseteq)$.\(^6\)
and vice versa. Furthermore, we can show that wrt. any \((\subseteq)\) the compositional closure is the smallest complete relation, and it is the biggest sound relation.\(^7\)

Finally we can show that a compositional closure must be monotonic, arguing directly from its general definition: we reason

\[
C(S) \not\subseteq C(I) \quad \text{for some } C \\
\Rightarrow D(C(S)) \not\subseteq D(C(I)) \quad \text{for some } D \\
\Rightarrow S \not\subseteq I.
\]

"completeness"

"soundness wrt. context \(D \supseteq C\)"

That being so, we introduce the

**Principle of Consistency** If \(S \sqsubseteq I\) then \(S \leq I\).

Now Soundness implies Consistency via the identity context; and Monotonicity with Consistency implies Soundness trivially. Thus with Completeness we have that Soundness can be equivalently formulated as Consistency plus Monotonicity. The latter presentation is slightly easier to prove, since it better suits a structural induction.

### 7.3. Our \((\sqsubseteq)\) is the compositional closure of \((\subseteq)\)

We now argue —still in advance of having a precise semantics— that our relation \((\sqsubseteq)\) is likely to be the compositional closure of \((\subseteq)\). We have already (optimistically) assumed it to be sound.

For completeness, suppose that program \(I\) yields a triple \((v', h', H'_1)\) but there is no \(H'_2 \subseteq H'_1\) such that program \(S\) yields a triple \((v', h', H'_2)\). Thus we conclude from (8) that \(S \not\sqsubseteq I\).

Can we find a context \(C\) such that also \(C(S) \not\subseteq C(I)\) ?

Assume wlog. that type \(\mathcal{H}\) has at least two values \(h_{0,1}\). Define a function \(f: \mathcal{H} \rightarrow \mathcal{H}\) such that \(f.h = h_0\) for all \(h: H'_1\) and is \(h_1\) otherwise, and then let context \(C\) be \(\{;\) if \((v, h) = (v', h')\) then \(h := f.h \) else \(h \in \{h_0, h_1\}\ fi\)\).

Now \(C(I)\) produces final shadow \(\{h_0\}\) when \(I\) itself produces the triple \((v', h', H'_1)\), as we assumed it can. But \(C(S)\) cannot do that: for if \(S\) produces some \((v'', h'', H'')\) with \((v'', h'') \neq (v', h')\), then the final shadow is \(\{h_0, h_1\}\), which is not \(\{h_0\}\) alone; and when \(S\) produces \((v', h', H'_2)\) for any \(H'_2\), we cannot get final shadow \(\{h_0\}\) either, since by assumption \(H'_2 \not\subseteq H'_1\).

Thus we have found our context, and shown completeness.

With a good will, we have now tied down in (8) a definition of refinement \((\sqsubseteq)\) that suits the choice of elementary testing \((\subseteq)\) we chose so long ago in Def. 1: the refinement relation is small enough to be sound, but big enough to be complete. Moreover, we will see in §10 to come that it is big enough to satisfy a remarkably general “transfer principle” that gives us for free a characterisation of many important secure refinements that are analogues of classical ones.

Before that, however, we will have to repay the “good will” by finding a semantics that justifies the various informal proofs we have done above; we come to that in §9.4.

### 8. Distilling the essence of traces leaves The Shadow Semantics

The state-sequence model of §4 was chosen to reflect the informal (but crucial) issues discovered in the preliminary experiments of §2. In fact it contains far more information than we need (like a genotype).

The subsequent analyses of §§6.1–6.4, and in particular the final definition of refinement (8), determined for us just what information we needed to keep (alleles) and, by implication, what we could throw away. With that in mind, we determine the semantics of a command by taking an initial state, generating all traces

\(^7\) If \((\subseteq_{a,b})\) are both complete wrt. \((\subseteq)\) and \(\neg(S ((\subseteq_a) \cap (\subseteq_b)) I)\) then wlog. \(S \not\subseteq I\) whence by completeness of \((\subseteq_a)\) we have \(C(S) \not\subseteq C(I)\) for some \(C\). Hence \(((\subseteq_a) \cap (\subseteq_b))\) is also complete, and so there is a smallest relation \((\sqsubseteq)\) complete wrt. \((\subseteq)\). If that \((\sqsubseteq)\) is not sound, then for some \(S, I, C\) we have \(S \sqsubseteq I\) but \(C(S) \not\subseteq C(I)\). Remove relationship \((S, I)\) from \((\sqsubseteq)\) and that is strictly smaller, but still complete — a contradiction.

If \((\subseteq_{a,b})\) are both sound and \(S ((\subseteq_a) \cup (\subseteq_b)) I\) then wlog. \(S \subseteq I\) whence by soundness of \((\subseteq_a)\) we have \(C(S) \subseteq C(I)\) for any \(C\). Hence \(((\subseteq_a) \cup (\subseteq_b))\) is also sound, and so there is a largest relation \((\sqsupseteq)\) sound wrt. \((\subseteq)\). If that \((\sqsupseteq)\) is not complete, then for some \(S, I\) we have \(S \not\sqsubseteq I\) but \(C(S) \subseteq C(I)\) for all \(C\). Add relationship \((S, I)\) to \((\sqsubseteq)\) and that is strictly bigger, but still sound — a contradiction.
and calculating the equivalence (§4), keeping only the triples \((v, h, H)\) that we need for refinement (7.3), and “up-closing” in the third, shadow component so that we can express refinement as reverse set inclusion. Thus a command \(P\)’s semantics \([P]\) is of type \(V\rightarrow H\rightarrow \mathcal{P}(V\times H\times \mathcal{P}H)\), with the powerset “\(\mathcal{P}\)” on the output side reflecting possible nondeterminism, and an up-closure condition that if \((v', h', H') \in [P]v.h.H\) and \(H' \subseteq H''\), then also \((v', h', H'') \in [P]v.h.H\).

And now, finally, we can say that a specification \(S\) is refined by an implementation \(I\) just when for all \(v, h, H\) in their types we have \([S]v.h.H \supseteq [I]v.h.H\).

Here are some examples. Let the type \(H\) of \(h\) be \(\{0, 1, 2\}\) and consider the program \(h:\in \{0, 1\} \cap h:\in \{1, 2\}\) which, seen more closely, becomes
\[
[p] \quad \langle p_1 \mid h:\in \{0, 1\} \cap p_2 \rangle \cap \langle p_3 \mid h:\in \{1, 2\} \cap p_4 \rangle \mid p_5
\]
with the bracketed notations being program-counter values. From initial state \((v, h, p)\) this generates the four

\[
\begin{align*}
&((v, h, p), (v, h, p_1), (v, 0, p_2), (v, 0, p_5)) , & \text{in equivalence class } A, \text{ say} \\
&((v, h, p), (v, h, p_1), (v, 1, p_2), (v, 1, p_5)) , & \text{also in class } A \\
&((v, h, p), (v, h, p_3), (v, 1, p_4), (v, 1, p_5)) , & \text{in class } B \\
&((v, h, p), (v, h, p_3), (v, 2, p_4), (v, 2, p_5)) , & \text{in class } B
\end{align*}
\]
with two \(\sim\)-equivalence classes \(A, B\) as indicated. If now we “distil the essence,” by keeping only the final
\((v, h)\) values while retaining the equivalence, we get
\[
\begin{align*}
& (v, 0) \quad \text{in class } A \\
& (v, 1) \quad \text{in class } A \\
& (v, 1) \quad \text{in class } B \\
& (v, 2) \quad \text{in class } B
\end{align*}
\]
where the relation is no longer an equivalence (but it doesn’t matter), because we have thrown away some information and thus have lost transitivity. This gives for our one-and-only initial \(v\) the two \(H'\)-sets, that is the two shadows \(\{0, 1\}\) and \(\{1, 2\}\) which (interestingly enough) are not disjoint. Thus our program produces from the given initial state four “minimal” final-state triples plus three others generated by up-closure:

\[
\begin{align*}
& (v, 0, \{0, 1\}) \quad (v, 0, \{0, 1, 2\}) \\
& (v, 1, \{0, 1\}) \quad (v, 1, \{0, 1, 2\}) \\
& (v, 1, \{1, 2\}) \quad (v, 2, \{0, 1\}) \\
& (v, 2, \{1, 2\}) \quad (v, 2, \{0, 1, 2\})
\end{align*}
\]

If we concentrate on the minimal (left-hand) triples in (10), we learn that after observing the above program an attacker will either know that \(h'\) is in \(\{0, 1\}\) or he will know that it is in \(\{1, 2\}\) even though before the program runs he will not know which of those things he will later know: in that sense they are “not-yet-known knowns.”

In other words, at the beginning of the run he cannot predict the yet-to-occur resolution of the first, non-atomic nondeterminism (\(\cap\)) but by the end of the run he will have seen which way it went. Nevertheless, which ever way that was, he will not have seen how the immediately following atomic (\(\cap\)) was resolved (in either case) — he will know, from the source code and single-stepping, over which set (either \(\{0, 1\}\) or \(\{1, 2\}\)) that second resolution occurred.

The right-hand triples, generated by closure, tell us that whatever happens the attacker will know that \(h'\) is in \(\{0, 1, 2\}\).

We see here, incidentally, the role of atomicity: an atomic command (like \(h:\in \{0, 1\}\)) has a single entry and exit and —even if the command contains nondeterminism— a single-stepping attacker will be taken straight from its entry to its exit, observing nothing about the choice taken along the way.

---

8 This is effectively the same as \((V\times H\times \mathcal{P}H)\rightarrow \mathcal{P}(V\times H\times \mathcal{P}H)\), but the “spread-out” Curried arguments generate less parenthesis-clutter in use. Another familiar alternative is \((V\times H\times \mathcal{P}H)\rightarrow (V\times H\times \mathcal{P}H)\); but this makes it messier to state the healthiness conditions subsequently.
9. A programming language and its semantics

9.1. Syntax

The syntax of a simple programming language is given in Fig. 1. It includes visible demonic choice (\(\cap\)) as well as hidden, atomic choice (\(:\in\)). As we have already remarked, the former is directly observable and the latter is not. The declaration-annotations \texttt{vis} and \texttt{hid} determine the visibility of variables.

9.2. Semantics defined via translation to a classical program

The semantics is given in Fig. 2 via a translation of the secure syntax into classical program syntax including an explicit new variable \(H\).

Visible- and hidden local variables can be declared within blocks; note that scope does not affect visibility: (even) a global hidden variable cannot be seen by the adversary; (even) a local visible variable can.

Brackets \([\cdot]\) (for brevity) or equivalently \texttt{begin \cdot end} (in this section only, for clarity) introduce a local scope that initially extends either \(v\) or \(h, H\) as appropriate for the declarations the brackets introduce, and
finally projects away the local variables as the scope is exited. In the translation for visibles this treatment is the usual one: thus we have
\[
\left[\text{begin vis } \nu; \ P \ \text{end}\right] := \ \text{begin var } \nu; \ [ \ P \ ] \ \text{end}.
\]
For hidden variables however we arrange for \(H\) to be implicitly extended by the declaration; thus we have
\[
\left[\text{begin hid } h; \ P \ \text{end}\right] := \ \text{begin var } h; : = H \times \mathcal{H}; [ \ P \ ]; H := H | \text{end},
\]
where we invent the notation \(H|\) for the projection of the product \(H \times \mathcal{H}\) back to \(H\).

We make the following observations on the translations of Fig. 2 before turning to an example:

- **Assign to visible** “shrinks” the shadow to just those values still consistent with the value the visible reveals: the adversary, knowing the outcome and the program code, concludes that the other values are no longer possible. This is consistent with **Perfect Recall**.
- **Choose visible** is a generalisation of that.

- **Assign to hidden** sets the shadow to all values that could have resulted from the current shadow; again, **Choose hidden** is a generalisation.

- **Demonic choice** and **Composition** retain their usual definitions. Note in particular that the former induces nondeterminism among possible shadows \(H\) as well.

- The **Conditional** shrinks the shadow on each branch to just those values consistent with being on that branch, thus representing the adversary’s observing which branch was taken. This is consistent with **Implicit Flow**.

- **The Shadow** \(H\) is a single powerset over the tuple of all hidden variables (rather than a tuple of separate powersets). In this way any correlations between hidden variables are captured.

### 9.3. Examples of semantics

We give two examples, with the second being a refinement of the first. We begin with program \(P\) defined
\[
h \in \{0, 1\}; (v := h \sqcap v := h + 1).
\]
From Fig. 2 we have
\[
\left[\begin{array}{l}
\begin{array}{l}
h \in \{0, 1\}; (v := h \sqcap v := h + 1) \\
\end{array}
\end{array}\right] = \left[\begin{array}{l}
\begin{array}{l}
h \in \{0, 1\}; [v := h \sqcap v := h + 1] \\
\end{array}
\end{array}\right] = \left[\begin{array}{l}
\begin{array}{l}
h \in \{0, 1\}; (\{v := h\} \sqcap \{v := h + 1\}) \\
\end{array}
\end{array}\right],
\]
\[
\text{“composition”}
\]
\[
\text{“demonic choice”}
\]
\[
\text{“choose hidden; rename bound variable”}
\]
\[
\text{“classical simplification”}
\]
\[
\text{“assign to visible; rename bound variables”}
\]
\[
\text{“remove temporary } e; \text{ classical simplification”}
\]
\[
\text{“classical simplification”}
\]
By inspection of the final, simplified classical program we see that the possible final state-triples are just these four:
\[
(0, 0, \{0\}) \quad (1, 0, \{0\}) \quad (1, 1, \{1\}) \quad (2, 1, \{1\})\quad (11)
\]
Because the Shadow $H$ is singleton, the hidden $h$ can be deduced in each case.

Now we consider program $Q$ defined
\[
h :\in\{0,1\}; v :\in\{h,h+1\},
\]
in which the second command $v :\in\{h,h+1\}$ is a strict refinement of the earlier $v := h \land v := h+1$ that was in $P$, in spite of our passing observation in §3.2 that $v :\in\{0,1\}$ and $v := 0 \land v := 1$ are equal. Our current cases are distinguished by the fact that we have hidden variables on the right, whereas earlier in §3.2 we did not: in §10 we get further insight into this important point.

In the meantime, from Fig. 2 we now have
\[
\begin{align*}
1 & = h :\in\{0,1\}; [v :\in\{h,h+1\}] \\
2 & = h :\in\{0,1\}; H := \bigcup\{h': H \cdot \{0,1\}\}; [v :\in\{h,h+1\}] \\
3 & = h :\in\{0,1\}; H := \{0,1\}; [v :\in\{h,h+1\}] \\
4 & = h :\in\{0,1\}; e :\in\{h,h+1\}; H := \{h': H \mid e \in\{h',h'+1\}\}; v := e \\
5 & = h :\in\{0,1\}; e :\in\{h,h+1\}; H := \{h', \{0,1\} \mid e \in\{h',h'+1\}\}; v := e \\
6 & = h := 0; e := 0; H := \{0\}; v := e \\
\land & h := 0; e := 1; H := \{0,1\}; v := e \\
\land & h := 1; e := 1; H := \{0,1\}; v := e \\
\land & h := 1; e := 2; H := \{1\}; v := e \\
7 & = v, h := 0, 0; H := \{0\} \\
\land & v, h := 1, 0; H := \{0,1\} \\
\land & v, h := 1, 1; H := \{0,1\} \\
\land & v, h := 2, 1; H := \{1\},
\end{align*}
\]
and here we see that the possible final state-triples are
\[
(0,0,\{0\}) (1,0,\{0,1\}) (1,1,\{0,1\}) (2,1,\{1\}).
\]
(12)
The difference between $P$ and $Q$ is now evident: when $v$ is 1 finally, in $P$ we can deduce $h$’s final value; but in $Q$ we cannot. The other two cases do not reveal a difference, since when $v$ is finally 0 we know $h$ is 0 because $0 \not\in \{1,1+1\}$ means it cannot be 1; and when $v$ is finally 2 we know $h$ is 1 because $2 \not\in \{0,0+1\}$ means it cannot be 0.

The triples of (12) are a refinement of those in (11) because for example triple $(1,0,\{0,1\})$ of the latter is supported by $(1,0,\{0\})$ in the former. The reverse is not true, so the refinement $P \sqsubseteq Q$ is strict.

9.4. Repaying the “good will”

The semantics of Fig. 2 can be shown to establish Monotonicity of $(\sqsubseteq)$, by a structural induction; and Consistency from §7.2 is trivial. Thus our refinement $(\sqsubseteq)$ is sound with respect to elementary testing $(\sqsubseteq)$ over this semantics.

The argument for Completeness in §7.3 is legitimised by fact that our informal reasoning there, about the specific context we constructed, is supported by the semantics we have just defined. Thus our refinement is complete with respect to elementary testing over this semantics.

Knowing in advance the detailed properties that our semantics had to fulfil, from above, supplied in effect a fine-grained target for our definitions, much more help than we would have had without our initial (and
extensive) explorations: we knew that this relation had to satisfy monotonicity; that that principle had to hold; and that those other informal arguments about program behaviour had to turn out to be accurate.

That’s the value of guesswork and intuition — which however must be paid for in the end.

10. The Transfer Principle

10.1. Statement of the principle, and examples

In §2 we pointed out the adversarial nature of refinement, supporting the view that secure refinement (⊆) was going to be more restrictive than classical refinement (≤): Soundness pushed us in that direction, and we saw in §6.1ff how more and more putative refinements had to be forbidden. But we found in §6.5 that it could not be too restrictive without crippling the refinement process altogether.

We then appealed to Completeness in §7.1 for a push back in the opposite direction, preserving a large number of refinements by insisting that refinement should be denied only if there was a good reason, a context that could show that forbidding it was necessary because of elementary testing.

Thus Soundness and Completeness struck a balance, in effect equally opposing forces leading us to an equilibrium where the definition of (⊆) was found: it is the compositional closure of (≤) with respect to our semantics.

In this section we show how remarkably effective that equilibrium turns out to be: we characterise a large collection of classical refinements that carry over unchanged into the secure semantics. That is done with the

Transfer Principle Let $C_1(X,Y,\cdots,Z)$ and $C_2(X,Y,\cdots,Z)$ be two contexts involving only visible variables, and satisfying the classical refinement

$$C_1(X,Y,\cdots,Z) \leq C_2(X,Y,\cdots,Z)$$

for all classical program fragments $X,Y,\cdots,Z$ — even those in which further visible variables not appearing already in $C_{1,2}$ might occur. Then the secure refinement

$$C_1(X,Y,\cdots,Z) \sqsubseteq C_2(X,Y,\cdots,Z)$$

holds for all program fragments $X,Y,\cdots,Z$ over both visible and hidden variables.

With Transfer we aim to codify the designer’s classical intuition about what should still be true even for secure refinement. Here are some examples of the Transfer Principle in action:

- (from §2.2) To show that $X \sqcap Y \sqsubseteq X$, choose $C_1(X,Y):=X \sqcap Y$ and $C_2(X,Y):=X$.
- (from §2.3) To show Extension, that $S \sqsubseteq I$ implies $S \sqsubseteq I$ whenever $S,I$ are over visible variables only, choose $C_1() := S$ and $C_2() := I$.
- To show that skip is a (left) identity, choose $C_1(X):=\text{skip}; X$ and $C_2(X):=X$.
- To show associativity of composition, choose $C_1(X,Y,Z):=(X(Y)); Z$ and $C_2(X,Y,Z):=(X; Y); Z$.
- To show distributivity of $\sqcap$ and composition, choose

$$C_1(X,Y,Z):=X \sqcap (Y; Z) \quad \text{and} \quad C_2(X,Y,Z):=(X; Y) \sqcap (Y; Z)$$

and

$$C_1(X,Y,Z):=X; (Y \sqcap Z) \quad \text{and} \quad C_2(X,Y,Z):=(X; Y) \sqcap (Y; Z).$$

There are many more examples.

Here, in contrast, are several cases in which Transfer does not apply:

- The context parameters $X,Y,\cdots,Z$ must be program fragments, not expressions: otherwise we could reason incorrectly that $v: \in X \sqsubseteq X$ in general. We have already seen that $v: \in \{h, h+1\} \not\sqsubseteq v:=h$.
- The classical refinement must hold even over parameters with extra visible variables: otherwise we could reason incorrectly that $v: \in V \sqsubseteq X$ in general. (Note that $v: \in V \sqsubseteq X$ does not hold if $X$ assigns to some $w$.) We have already seen that $v: \in V \not\sqsubseteq v:=h$. 

The distributivity of conditional and composition

\[
\text{if } E \text{ then } X \text{ else } Y; Z \subseteq \text{ if } E \text{ then } X; Z \text{ else } Y; Z \text{ fl}
\]
cannot be shown by Transfer when \(E\) contains hidden variables. Still it is true: but a bespoke proof is required.

10.2. Proof of the Transfer Principle

Suppose the classical refinement

\[
C_1(X, Y, \ldots, Z) \leq C_2(X, Y, \ldots, Z)
\]
holds for all classical program fragments \(X, Y, \ldots, Z\), and let \(P, Q, \ldots, R\) be specific program fragments possibly containing hidden variables. Using Fig. 2 we can translate any \(C(P, Q, \ldots, R)\) into a classical program containing an extra variable \(H\) (The Shadow): but it is a property of that translation that visible-only program constructions are translated into themselves: hence we have

\[
[C(P, Q, \ldots, R)] = C([P], [Q], \ldots, [R]).
\]

Now we have that \(C_1([P], [Q], \ldots, [R]) \leq C_2([P], [Q], \ldots, [R])\), since from our assumption (13) we can make the instantiation \(X, Y, \ldots, Z := [P], [Q], \ldots, [R]\) by interpreting the latter as classical programs over \(v, h, H\). Appealing to (14) for \(C_{(1,2)}\) then gives us the classical refinement

\[
[C_1(P, Q, \ldots, R)] \leq [C_2(P, Q, \ldots, R)],
\]
that is that for any \((v, h, H)\) and \((v', h', H')\) with \((v', h', H') \in [C_1(P, Q, \ldots, R)]\), \((v, h, H)\) we must have also \((v', h', H'') \in [C_1(P, Q, \ldots, R)]\), \((v, h, H)\). But this implies secure refinement, since for that we need only the weaker \((v', h', H'' \subseteq H')\) for some \(H'' \subseteq H'\). Thus we have

\[
C_1(P, Q, \ldots, R) \subseteq C_2(P, Q, \ldots, R)
\]
as required.

11. Healthiness conditions

In the earlier section (§8) we finally distilled a “lean” semantics (sets of triples) from the operationally motivated guesswork (§4) with which we began (sets of sequences of triples). In order that refinement could be simple reverse subset-inclusion, we included the up-closure requirement, a so-called “healthiness condition” on the sets of outputs we are prepared to consider. Thus although the space of program results, the “square type,” is \(P(v \times H \times PH)\), we do not consider all possible sets (i.e., elements of the powerset) to be healthy: only the up-closed ones are.

That technique of up-closure is a very common feature of program semantics. For example, in sequential relational semantics \(S \rightarrow PS_\perp\) modelling both nontermination (with \(\perp\)) and nondeterminism (with \(P\)), it is conventional to “fluff-up” the output sets so that if \(\perp\) is in the set, so is every other element of the state-space \(S\). This is simply up-closure over the flat domain \(S_\perp\) where \(s_1 \subseteq s_2\) just when \(s_1 = \perp \neq s_2\).

Similarly in sequential and probabilistic relational semantics [10, 21, 15], where the output sets contain

\footnote{Here we rely on \(X, Y, \ldots, Z\) being programs, not expressions: we have not defined \([\cdot, \cdot]\) for expressions on their own.}

\footnote{Here we use the fact that \(X, Y, \ldots, Z\) must be allowed contain extra visible variables, not appearing in \(C_{(1,2)}\): in this case, that extra variable is \(H\).}

\footnote{Encouraged by a referee, I tracked this possibly unfamiliar term down with the help of Bernard Sufrin (email Sep. ’09). He wrote that the term square sets was coined by Abrial (∼1981) to mean those “freely constructed” from givens with product and power only. These are the “essentials” from within which all other describable sets were “separated” by predicates. (It’s an informal, descriptive notion, not intended (e.g.) to compete with the cumulative hierarchy.)}

\footnote{More precisely, it is the Smyth powerdomain-order on \(PS_\perp\) generated from the underlying flat order on \(S_\perp\) itself [24]. Fluffing-up’s being a specific instance of the general Smyth-technique is the reason it works so well.}
subdistributions $\Delta$ over $S$, up-closure requires that if subdistribution $\Delta$ can be produced by a program and we have $\Delta \subset \Delta'$, then we consider also $\Delta'$ to be produced (potentially) by that program.\footnote{A subdistribution sums to no more than one, rather than to one exactly: any deficit represents the probability of nontermination (abort). Thus when $\Delta \subset \Delta'$ the greater subdistribution assigns at least the same probability to all proper outcomes, and strictly less probability to abort [21, 15].}

Finally, in the failures model of CSP a process is considered to be a set of trace-failure pairs $(s, X)$, the test in this case being the claim that “after trace $s$ the offer $X$ can be refused.” Clearly in that case also $(s, X')$ can be refused whenever $X' \subseteq X$, and so the refusal sets are down-closed — which is indeed up-closed if you are upside-down.

In all three cases, refinement is reverse subset-inclusion as at (9), the effect — and indeed the point — of using up-closure.

But up-closure is not always used, and thus refinement is not always as simple as reverse subset-inclusion. Nevertheless it is always equivalent to that, if you look at things in a certain way; it’s just that sometimes looking at things in another way is more convenient.

A well known example without up-closure is the $Z$-schema [9], whose type (interpreting the predicate as a relation) is $S \leftrightarrow S$ with $S$ being the set denoted by the signature. For $Z_{1,2} \in S \leftrightarrow S$ we have $Z_1 \subseteq Z_2$ just when

$$\text{dom.} \ Z_1 \subseteq \text{dom.} \ Z_2 \quad \land \quad (\text{dom.} \ Z_1) \triangleleft Z_2 \subseteq Z_1.$$  

That’s certainly more complicated than reverse subset-inclusion; but avoiding explicit mention of $\bot$ is simpler for specifications, one of $Z$’s primary purposes.

Another example without up-closure is weakest preconditions, whose semantic space is $\mathcal{P}S \rightarrow \mathcal{P}S$ taking post-conditions to the weakest preconditions that guarantee them; it is not the case that $wp.P. post = pre$ and $pre' \Rightarrow pre$ implies $wp.P. post = pre'$ as well, although clearly in this case the up-closure is extremely close by. (Think of Hoare triples.)

We too will abandon up-closure, for reasons to do with another form of healthiness condition: but what we are going to do will be equivalent to what we have already done. (That is, we are not going to alter our definition of refinement any further.)

Weakest preconditions have a healthiness condition conjunctivity, that for two postconditions $post_{1,2}$ we have $wp.P.(post_1 \land post_2) = wp.P.post_1 \land wp.P.post_2$ — and this is because they are constructed from the underlying, and more operational relational semantics. For a relational program (semantics) $r \in S \rightarrow S$, we define $wp.r.post$ to be $\{s:S \mid (\forall s':S \mid s' \in r.s \cdot s' \in post )\}$, where here we are interpreting predicate post as the subset of $S$ it denotes. Now $wp.r$ is of type $\mathcal{P}S \rightarrow \mathcal{P}S$, as it should be; but function $wp$ is itself not surjective. That is, the image of the set $S \rightarrow S_1$ through the function $wp$ is exactly the subset of $\mathcal{P}S \rightarrow \mathcal{P}S$ containing the conjunctive functions only, and that is not all of them. Thus this kind of healthiness condition is (paradoxically) a “scar” left over from the way in which the semantic denotation was constructed (in this case, via the $wp$ function). The “sub-linearity” of probabilistic predicate transformers is exactly the same kind of scar [21, 15].

In fact there are two scars left by the way in which we construct the Shadow Semantics (of triples) from the operational semantics (of traces). The first is that in any output triple $(v, h, H)$ we must have $h \in H$, because the equivalence relation $\sim$, from which the shadow $H$ was constructed, is reflexive. That is (A) the actual value $h$ of $h$ recorded in the triple $(v, h, H)$ must be one of the potential values the attacker considers to be possible.

The second is that if $(v, h, H)$ is a possible output-triple, and we have $h' \in H$ for some (other) $h'$, then also $(v, h', H)$ must be a possible output triple. That is (B) a reasonable attacker will never consider $h'$ to be a possibility unless inspection of the source-code shows that the program could actually produce it.

Note that the explanations (A,B) are not the causes of the two healthiness conditions: the cause is that the process of distilling triples from traces (§8) is not surjective onto $\mathcal{P}(\mathbb{V} \times H \times \mathcal{P}H)$. The explanations are only post-hoc “sanity checks” that we are still heading in the right direction. The two conditions are that for any output-set $R := \{P|h.h.H$ of triples we must have that

(A) reflexivity If $(v', h', H') \in R$ then $h' \in H'$.

(B) necessity If $(v', h', H') \in R$ and $h'' \in H'$ then also $(v', h'', H') \in R$.  

\[15\]
It is because of the necessity condition that we should drop up-closure. If in Example (10) the type \( H \) had been bigger, say \( \{0,1,2,3\} \), then up-closure would add triples like \( (v_0,0,\{0,1,2,3\}) \) which in respect of the potential \( h \)-value 3 violates necessity: there is no other triple \( (v_0,3,\{0,1,2,3\}) \). And indeed there should not be one, because the program cannot assign 3 to \( h \) under any circumstances.

Note the definition (8) of refinement has not changed; it is just that once we drop up-closure it is no longer conveniently equivalent to reverse subset-inclusion. We give the importance of necessity (15) a higher priority.

12. Some interesting examples

The point of working out a semantics carefully, especially in a new and unfamiliar domain, is that it clarifies and instructs your intuition in situations where that intuition might be somewhat stretched. Especially enjoyable are program-pairs that are “obviously” different, yet the semantics says they are the same, or pairs that are “obviously” the same but the semantics says they differ. In this section we present a few of those.

12.1. The Encryption Lemma, and Specification Statements

What we call the Encryption Lemma is based on properties of exclusive-or \( \oplus \): it is that in the context of a global hidden Boolean \( h \) the local block

\[
\text{vis } v; \text{hid } h; \quad h' \in \{0,1\}; v' := h \oplus h'
\]

(16)
is equivalent to \textit{skip} — it changes nothing; and it reveals nothing. This is not surprising: a global coin \( h \) is (already) hidden; a second local hidden coin \( h' \) is introduced and flipped; and it is published in local \( v' \) whether the two coins show the same face. Obviously that reveals nothing about the original coin \( h \) since the temporary coin \( h' \) is hidden and, indeed, thrown away at the end of the local block.

This idiom is used so often (e.g. in both the Dining Cryptographers [5, 18] and the Oblivious Transfer [23, 20]) that it’s nice to have a concise notation for it; and so we write \( (v' \oplus h') := h \) for the contents of the block, an abbreviation for a specification statement [16] or a generalised substitution [1]:

\[
v', h'; (v' \oplus h') = h
\]

or

\[
\text{ANY } v_1, h_1 \text{ WHERE } v_1 \oplus h_1 = h \text{ THEN } v', h' := v_1, h_1 \text{ END .}
\]

That is, we set \( v' \) and \( h' \), nondeterministically, so that their exclusive-or equals \( h \); and that nondeterminism is hidden, because this command is atomic.

The interesting issue is what such specification statements mean in general when hidden variables are involved: we could, after all, decompose our example specification statement in the complementary way to (16), giving

\[
\text{vis } v; \text{hid } h; \quad v' \in \{0,1\}; h' := h \oplus v'
\]

(17)

Here the nondeterminism is visible, because it’s being applied to visible \( v' \), and an attacker can see \( v' \) even though it’s local. In fact, as remarked earlier our semantics identifies \( v' \in \{0,1\} \) and \( v' := 0 \cap 1 \) when \( v' \) is visible, whether local or not.

The question is then whether (16) and (17) really are equal. They should be, by transitivity of equality since they are both equal to \( (v' \oplus h') := h \). On the other hand, in (16) the nondeterminism is hidden, yet in (17) it’s visible. \textit{Obviously} that means they are different — but in fact they are the same.

12.2. Revelations, and “effectively visible” values

The \textit{revelation command} \texttt{reveal E} publishes the value of \( E \) for all to see, but changes no variable in doing so: it’s equivalent by definition to the local block \[ \text{vis } v; \quad v := E \]. We say that a value \( E \) is \textit{effectively visible} at a point in a program just when adding a statement \texttt{reveal E} at that point does not change the

\footnote{This might not surprise everyone; but it surprised us. Unfortunately you can’t surprise all of the people all of the time.}
program’s meaning \[13\]. If in particular a hidden variable \(h\) is effectively visible at some point, then in reasoning about the program at that point we can treat expressions involving \(h\) as if it were declared visible (instead of hidden, as it actually was declared).

Thus for example hidden variable \(h\) is effectively visible at \(*\) in the program fragment \(v := h \ast\) because the two programs \(v := h\) and \(v := h; \text{reveal}\ h\) are equal. That’s no surprise. But it is also true that the two programs \(v := h\) and \(\text{reveal}\ h; v := h\) are equal, and so do we conclude that \(h\) is effectively visible at \(*\) in \(*\ v := h\)? Obviously variable \(h\) is not effectively visible before it has been assigned to \(v\) — but in fact it is.

12.3. Effective visibility and nondeterminism

Let’s accept \(h\)’s effective visibility at \(*\) in \(*\ v := h\) — perhaps reasoning that, although the attacker does not actually know \(h\) at the time \(*\) is reached, he can reason as if he does simply by waiting one more command before he starts drawing conclusions retrospectively about the program state at it was at \(*\) (now in the past). It makes no difference when those conclusions are drawn, and the eventual release of \(h\) is inevitable: the statement \(v := h\) cannot be avoided.

With that in mind, we exercise the semantics a bit more by asking about \(*\ (v := h \sqcap \text{skip})\). Is \(h\) effectively visible this time? After all, it is no longer certain that it will be revealed by the assignment, since the \text{skip} command might be executed instead. Nevertheless, since \(\sqcap\) is demonic we should conclude the worst, that \(h\) is effectively visible, simply because the demon might make it so. That depends —by definition— on the equality of the two fragments \(v := h \sqcap \text{skip}\) and \(\text{reveal}\ h; (v := h \sqcap \text{skip})\) — and obviously they are equal, for the reasons just mentioned. But in fact they are not.

12.4. Local blocks and cover-ups

As our last example we make \(v\) local: is \(h\) effectively visible in \(*\ [\ \text{vis} v'; v' := h \sqcap \text{skip} ]\)? Obviously making \(v\) local has no effect, since it can be seen regardless of the fact that its having held \(h\)’s value is “covered-up” when it is discarded at the end of the local block. (Locality has no effect on visibility.) But in fact \(h\) is effectively visible in this example, even though in §12.3 it was not.

13. Conclusion: Meta Formal Methods

We’ve told a story about constructing a refinement order and a model for non-interference in sequential programs. Naturally it didn’t actually happen that way. But in telling the story we are just doing Formal Methods “one level up.”

The same kind of story-telling occurs in the careful construction of computer programs, especially ones made using Formal Methods. There are many mistakes and blind alleys, and however much we would like to believe that a program is developed hand-in-hand with its proof via inexorable steps that are never undone, in practice the only “never” is that such developments almost never happen.

What we do aim for in Formal Methods is to set our standards of reasoning so high that we will be able to invent a story, afterwards, of how the program might have been constructed in that inexorable way if only we had been clever enough. This is not for us to convince others of how smart we are — rather it is to allow others to convince themselves that we have not been stupid on their behalf.

What we have done in this note is of exactly that same type, and serves the same purpose except that it in this case we are constructing not a program but a small formal method itself. Our story is an example of Meta Formal Methods.

Acknowledgements

Thanks to Eerke Boiten for careful reading and helpful comments on the original version \[19\], and to John Derrick (as well) for providing the opportunity and incentive to write this revised account.
References


A. Epilogue: an even leaner semantics, and its derived refinement order

The healthiness conditions \((A,B)\) of (15) mean that the \(h\) component of our triples is redundant: keeping just \((v,H)\) -pairs is equivalent. To get from triples to pairs, obviously we just project the \(h\)-component away. In the other direction, from a set of pairs \(P\) we deduce uniquely the set of triples \(R\) it must have come from via

\[
R = \text{addHid}.P := \{v, h, H \mid (v, H) \in P \land h \in H\}.
\]

Because of reflexivity (15A), this construction \text{addHid} does not miss any triples; because of necessity (15B) it doesn’t add too many. The type of programs’ semantics would then become the simpler \(V\rightarrow\mathbb{P}H \rightarrow \mathbb{P}(V \times \mathbb{P}H)\); but of course the refinement relation would have to change.

Luckily we do not have to torture ourselves by inventing a pair-appropriate refinement relation \(\subseteq_{18}\), call it, all over again from first principles, retracing the long route for the triple-appropriate relation \(\subseteq_{8}\) that we finally chose. Instead we simply say that \(P \subseteq_{8} P'\) just when \text{addHid}.P \subseteq_{8} \text{addHid}.P'.

We can work out a direct definition of \(\subseteq_{18}\) by calculation. Writing \(P_x\) for the projection \(\{H \cdot (v,H) \in P\}\), we have
Also it generalises more easily to our single shadow conflict with. –furthermore– we can retain that union-closure because there are no other healthiness conditions for it to condition of union-closure on the pairs, the refinement relation becomes reverse subset-inclusion again and components’ union equals the single could potentially need support from a collection that is that for every v, each set H in P_v must be the union of some collection of sets H occurring in P_v.

This refinement relation (⊆) is not as simple as (⊆) was, because each v-pair in the implementation could potentially need support from a collection of v-pairs in the specification (a collection whose components’ union equals the single H in question). On the other hand, if we impose a single healthiness condition of union-closure on the pairs, the refinement relation becomes reverse subset-inclusion again and–furthermore– we can retain that union-closure because there are no other healthiness conditions for it to conflict with.

Part of the intuitive appeal of the union-closure definition is that replacing two shadows H_1 ∪ H_2 by a single shadow H_1’ ∪ H_2’ is exactly what happens when we replace a visible choice (∈) by a hidden choice (,:) so that e.g. h := 0 ∪ 1, with its two shadows {0} and {1}, refines to h ∈ {0, 1} with its single shadow {0, 1}. Also it generalises more easily to our probabilistic-shadow semantics [14].

Thus (∈) ⊆ (:) is the “essence” of non-interference refinement.

Perhaps this is simpler overall?

As a final example, we ask whether h ∈ {0, 1} ∩ h ∈ {2, 3} is refined by h ∈ {0, 1, 2}. Obviously it is, because the potential h-outputs on the right are included in those on the left, and the Shadow on the right contains one of those on the left.

Using triples, we find nevertheless that it is not, because triple (v, 2, {0, 1, 2}) on the right has no support on the left: on the one hand, neither (v, 0, {0, 1}) nor (v, 1, {0, 1}) will do because, although {0, 1} ⊆ {0, 1, 2}, their h-components 0,1 do not include 2; on the other hand, (v, 2, {2, 3}) will not do because {2, 3} ∉ {0, 1, 2}.

Using pairs we find that, although {0, 1, 2} is a subset of the union of shadows {0, 1} ∪ {2, 3}, it is not the union exactly. Thus the views agree (as they should): the refinement fails either way.

B. Brief comments on the interesting examples

B.1. On §12.1

Operationally, the first formulation (16) seems secure because it involves an act whose outcome is unpredictable (the flip) and whose result we cannot see (the coin h is concealed). All that happens in the second formulation (17) is that the emphasis is placed on the hiding (the final assignment to h is pointless, since it is hidden and local), and we see that the burden of unpredictability is shifted to the secret h. Either way, we learn nothing.

B.2. On §12.2

Here the confusion is introduced by the informal use of English: of course h is not actually visible at that earlier point; but the attacker is not obliged to reason in real time. He can just take a whole slew of observations and puzzle them out later, at his leisure. The point is that he can deduce the value h had at the point indicated, even if that deduction is made well after the program has terminated.

\footnote{It also suggests a normal form for secure programs: all visible demonic choices are taken first, then hidden choices and then, finally, deterministic program fragments.}
B.3. On §12.3

The difference between \( v := h \sqcap \text{skip} \) and \( \text{reveal} \ h; (v := h \sqcap \text{skip}) \) is seen by considering the two programs in a larger context: we compare

\[
\begin{align*}
\text{h} ; & \in \mathcal{H} - \{v\}; \quad v := h \sqcap \text{skip}; \quad \text{if } v = h \text{ then } h ; & \in \mathcal{H} \text{ fi} \\
\text{and} \quad \text{h} ; & \in \mathcal{H} - \{v\}; \quad \text{reveal} \ h; (v := h \sqcap \text{skip}); \quad \text{if } v = h \text{ then } h ; & \in \mathcal{H} \text{ fi}.
\end{align*}
\]

The leading statement puts us in a context where we know that \( h \) and \( v \) differ, but that is all; the trailing statement tries to detect the escape of \( h \)'s value into \( v \), via the test \( v = h \), and executes a “cover up” command \( h ; \in \mathcal{H} \) if necessary.

We can now see that the second program is worse than the first, because it releases \( h \)'s value and (demonically) still escapes detection. The first program cannot escape detection, and the cover-up code ensures that either way we know little about \( h \) at its conclusion.

B.4. On §12.4

This last example highlights the difference between the attacker and the programmer. In App. B.3 we saw that the possible escape of \( h \) via the command \( v := h \) could be compensated for by defensive programming subsequently: it is as if the “thief” of \( h \)'s value left muddy footprints by altering \( v \)'s value in the process; and those footprints can be tested for later with program code.

But if \( v \) is local, then the thief’s footprints are left in snow — which melts at the end of the block. There is no defensive programming subsequently that can detect whether \( h \) escaped and so, this time, we must indeed assume the worst: that (demonically) the \( v := h \) was executed and not the \( \text{skip} \).