Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 259 (2009) 123–141

www.elsevier.com/locate/entcs

How to Brew-up a Refinement Ordering

Carroll Morgan[1,2]

School of Computer Science and Engineering
University of New South Wales
Sydney, Australia

Abstract

Fifty years ago there were few mathematical models of program semantics, perhaps none. Now there is probably a new one created every day. How do we do it? How *should* we do it?
In our community we understand the utility of the refinement order, and we believe that each fresh semantics should come equipped with one. Although refinement's general principles are well understood, it is still not so easy to see just what the order should be in any particular case. Thus one of the things we should do is be clear about what the criteria really are for refinement orders.
Recently invented is the Shadow Semantics for non-interference -style security of sequential programs *including a refinement order*. Using that as an example, I give here a rational reconstruction of how a refinement order can be "brewed-up" for a specific purpose; the aim of the exercise is to extract general lessons about how that can be done.

Keywords: The Shadow, refinement, security, ignorance, semantics, non-interference, refinement paradox, compositionality, full abstraction.

1 Introduction: *Refinement and implication*

Refinement $S \sqsubseteq I$ holds by definition just when "Specification S has good property G" implies "Implementation has property G" or, equivalently, "I has bad property B" implies "S has property B," with the implications holding in each case for all good/bad properties of a certain kind decided beforehand. Thus refinement is just implication, either forward of reverse: the problem is only in deciding "what kind of properties, exactly?" What G's and/or B's do we consider? And how do we decide whether or not a specification or implementation has such a property?

The initial step is subjective, determined by the phenomena to be studied: functional behaviour, probability, security, concurrency... But that's only a beginning; and the aim of this note is to show by example what technical tools and points of

[1] I acknowledge the support of the Australian Research Council Grant DP0879529.

[2] Email: carrollm@cse.unsw.edu.au

doi:10.1016/j.entcs.2009.12.021

view contribute to "brewing-up" a fully developed refinement order once that first step has been taken.

Inherited from implication are refinement's partial-order characteristics: reflexivity, antisymmetry and transitivity. Further properties are monotonicity and compositionality. The practical significance of *transitivity* is well known, that if you refine a program today, and refine it further tomorrow and again the day after, then by the weekend you still have a refinement of what you started with. This is one of the reasons we insist that it be a partial order. *Reflexivity* is trivial, since any program S or I has all the properties of itself, whether good or bad. *Antisymmetry* we return to later (as full abstraction).

Monotonicity is actually a property of functions, rather than of partial orders: we say that f is monotonic with respect to $\sqsubseteq$ just when $x \sqsubseteq x'$ implies $f.x \sqsubseteq f.x'$ for all x, x'. [3] In the space of programs (i.e. of their texts) monotonicity means that $S \sqsubseteq I$ implies $\mathcal{C}(S) \sqsubseteq \mathcal{C}(I)$ for all program-contexts $\mathcal{C}$. In the space of program meanings, we would say that the function f is the denotation of a program context (corresponding to $\mathcal{C}(\cdot)$), and the argument x fills the "hole" $(\cdot)$ in the context with the meaning of a program (such as S or I). Either way, the practical significance of monotonicity is that if you refine your own small part (from x to x') of a big program $(f.x)$ then what you get $(f.x')$ has not destroyed the work (f) of all your colleagues: it's still a refinement overall.

Compositionality is in the broader community more well known than monotonicity, but it is related: it means that the meaning of a compound is determined by the meanings of its components. If we use the conventional denotational brackets $[\![\cdot]\!]$ to convert syntax to semantics, compositionality thus means that whenever $[\![P]\!] = [\![Q]\!]$ we have $[\![\mathcal{C}(P)]\!] = [\![\mathcal{C}(Q)]\!]$ also. But monotonicity (in its second, semantic form) already says that $[\![P]\!] \sqsubseteq [\![Q]\!]$ gives us $[\![\mathcal{C}]\!].[\![P]\!] \sqsubseteq [\![\mathcal{C}]\!].[\![Q]\!]$, and vice versa, so that compositionality follows from refinement's other two partial-order properties of reflexivity and anti-symmetry. [4]

The above properties are basic guidelines for how refinement orders should behave; but the underlying issue is not so much the order but the semantics it sits in. That in turn depends crucially on the observations you allow yourself to make or –to put it another way– the observations you consider sufficiently important to be worth paying for. [5]

A popular starting point for finding the *semantic sweet-spot* in a new system is to conduct thought experiments between pairs of programs, asking whether or not they should be distinguished (are worth distinguishing) — with that being done *before* the semantics is set in concrete.

With our background of course we would rather ask the question whether one

[3] We write $f.x$ for function f applied to argument x. It reduces the number of parentheses needed in complicated expressions.

[4] Actually it follows anyway from the fact that $[\![\mathcal{C}]\!]$ is meaningful on its own, i.e. is actuall some function f that can be applied to program meanings.

[5] The currency could be real money –the salary of the *RA* who makes the semantics for you– or it could be more abstractly the time wasted down the decades with an observational repertoire that turns out to be pointlessly complex. On the other hand, it could also be the lives lost if that repertoire is too simple.

program is *refined by* another; and that is where we will begin in §2 immediately following. Where we will end –some time later– is the refinement order given by (6) below. The "rational reconstruction" of the path, from beginning to end, will try to show by example the kinds of issues that are encountered when designing any refinement order, and how they can be dealt with.

The context for our case study will be the recently developed Shadow Semantics for non-interference style security of sequential programs. Although sequential-program semantics [5,10,4], refinement [8,12,2,1,15] and the idea of non-interference -style security [6] were all established long ago, what impeded their being brought together was the *Refinement Paradox* [11]. In a simple form it is this:

> If a secret can have two values then it is a refinement (classically) to replace that by a secret which can have only one value: demonic nondeterminism has been reduced. Yet that refinement has destroyed the security property: something that can have only one value is no longer a secret.

With that in mind, we now turn to our experiments.

2 Gedanken Experiments

"The common goal of a thought experiment is to explore the potential consequences of the principle in question." [6]

For us, the principle in question is that the secure refinement that we are trying to define/discover should have the characteristics necessary for software development to be practical. It's important to carry out our experiments with respect to those principles *before* committing to a detailed design of the semantics and its accompanying refinement relation: not only will the experiments' outcomes help you to formulate your design; but –with luck– they will prevent you at an early stage from constructing a bad design which, because it took so much hard work, you will then be reluctant to throw away.

Here are some of the principles (among others [18]):

- Refinement is monotonic.
- All specific classical refinements remain valid if they do not involve security.
- All general "structural" classical refinements remain valid, whether they involve security or not.
- Expressions equal in a context can be exchanged without effect (referential transparency).

These characteristics have some surprising consequences. Here are just two:

Suppose that our program has just two variables v, h over $\{0,1\}$ with v visible and h hidden. *Does program* $v{:=}\,h; v{:=}\,0$ *reveal* h *to an observer?* We reason

$$(v{:=}\,h; v{:=}\,0);\ (v{:=}\,0 \sqcap v{:=}\,1)$$

[6] This is taken from the Wikipedia entry for "Gedanken experiments."

$$
\begin{array}{lll}
= & v{:=}\,h;\ (v{:=}\,0; (v{:=}\,0 \sqcap v{:=}\,1)) & \text{“structural equality: associativity”} \\
= & v{:=}\,h;\ (v{:=}\,0 \sqcap v{:=}\,1) & \text{“classical equality not involving security”} \\
\sqsubseteq & v{:=}\,h; \mathbf{skip} & \text{“classical refinement; monotonicity”} \\
= & v{:=}\,h\ , & \text{“structural equality: } \mathbf{skip} \text{ is the identity”}
\end{array}
$$

so that since the implementation $v{:=}\,h$ doesn't conceal h –which we regard as a *bad* property– we conclude that $v{:=}\,h; v{:=}\,0; (v{:=}\,0 \sqcap v{:=}\,1)$, the specification, must also have failed to do so. We must therefore give our semantics the property of *perfect recall* [7].

Another experiment is *whether program* $h{:=}\,0 \sqcap h{:=}\,1$ *reveals h.* We reason

$$
\begin{array}{lll}
 & (h{:=}\,0 \sqcap h{:=}\,1);\ (v{:=}\,0 \sqcap v{:=}\,1) & \\
= & & \text{“structural equality: distribution } \sqcap \text{ and ;”} \\
 & (h{:=}\,0; (v{:=}\,0 \sqcap v{:=}\,1))\ \sqcap\ (h{:=}\,1; (v{:=}\,0 \sqcap v{:=}\,1)) & \\
\sqsubseteq & (h{:=}\,0; v{:=}\,0)\ \sqcap\ (h{:=}\,1; v{:=}\,1) & \text{“classical refinement; monotonicity”} \\
= & (h{:=}\,0; v{:=}\,h)\ \sqcap\ (h{:=}\,1; v{:=}\,h) & \text{“referential transparency”} \\
= & (h{:=}\,0 \sqcap h{:=}\,1); v{:=}\,h\ . & \text{“reverse above steps”}
\end{array}
$$

From that we see that we can, by refinement, introduce a statement that reveals h explicitly. Our model must therefore allow the adversary to *observe the program flow*, because that is the only way operationally he could have discovered h in this case. Similar reasoning shows that

$$\mathbf{if}\ E\ \mathbf{then\ skip\ else\ skip\ fi}$$

reveals the Boolean value E.

Our experiments tell us that our semantics –whatever it turns out to be– should be rich enough to record past values of visible variables and of the program counter. Similar experiments give further insights, and the more time spent here the better off we are likely to be later: think of this as the “requirements capture” phase of our project to build a semantics.

Now we move to some initial design.

3 The execution model: sequences of program-states

We continue with just two variables v, h with v visible and h hidden; yet the experiments above have already suggested what these two terms “visible” and “hidden” should mean if we are going to allow refinement and non-interference -style security to coëxist:

An attacker executes our program and strives to discover 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).

- The *atomicity* of the program commands determines the points 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, each trace being 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_0, h_0, p_0), is then a set of (potential) traces of (v, h, p)-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. [7]

4 Equivalence of traces expresses the attacker's capabilities

Again appealing to the Gedanken Experiments §2, we consider two traces to be equivalent as follows. Suppose the types of v, h, p are $\mathcal{V}, \mathcal{H}, \mathcal{P}$ resp., so that the type of a trace $t\colon \boldsymbol{\mathcal{T}}$ is the set of sequences $\mathsf{seq}.(\mathcal{V}{\times}\mathcal{H}{\times}\mathcal{P})$; for such a t, write $\boldsymbol{v}.t, \boldsymbol{h}.t, \boldsymbol{p}.t$ for the projections onto types $\mathsf{seq}.\mathcal{V}$, $\mathsf{seq}.\mathcal{H}$ and $\mathsf{seq}.\mathcal{P}$ resp.

Then we say two traces $t_{\{1,2\}}$ are equivalent, writing $t_1{\sim}t_2$, just when we have both $\boldsymbol{v}.t_1{=}\boldsymbol{v}.t_2$ and $\boldsymbol{p}.t_1{=}\boldsymbol{p}.t_2$, 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). 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 of 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.

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 $\{t'\colon \boldsymbol{\mathcal{T}} \mid t \sim t'\}$. 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.

[7] That constructs a very simple Kripke model where the accessibility relation is the equivalence that relates "worlds" indistinguishable to an attacker.

5 The nature of possible attacks determines refinement

In classical correctness, the aim is to have implementations give the "correct" answer; with nondeterminism, of course there might be many correct answers. But not all answers should be correct (within the output variable's type), since otherwise there'd be no point in running the program. An *attack*, in the classical sense, is an attempt to show that an implementation does not meet its aim — thus a demonstration of an implementation's having produced an incorrect answer. The answer is observed in an actual run, and it is shown to be incorrect by examination of the specification's *source code*, proving by doing so that the specification could never have delivered that answer. The definition of refinement is *always* that if an implementation fails a test then it must be possible for the specification also to fail that same test. That depends in turn on the definition of a test, and is what we defined above for classical sequential correctness.

Taking the two previous paragraphs together, then, we have immediately that in our context refinement implies that any output value v possible for the implementation must also be possible for the specification. This is of course the classical refinement relation for nondeterministic programs, excluding h-outputs however because the attacker cannot see them.

For so-called *secure* refinement we add a second form of verification activity, this time to do with h. The model of §4 is necessary to formulate it, because that model is how we captured the idea that an attacker can gather information while the program is running: by using sequences and the equivalence, we can formulate his use –at the end of execution– of the information that he gathered along the way.

Here our test is a subset X of h's type $\mathcal{H}$; the attacker observes the implementation's execution and on a particular run, with its particular resolution of nondeterminism, he finds he is able to deduce *conclusively* that the final value of h is in that set X, even though at no stage could he see h itself, and not at the end in particular. For example (the most elementary), the program $h{:=}\,0$ is susceptible to the attack $X{=}\{0\}$ because the attacker's examination of the source code is enough to establish that $h{\in}X$ finally: he doesn't even have to run the program. The program $h{:=}\,0 \sqcap h{:=}\,1$ is susceptible to the same attack, even though h might finally be different from 0, because on the occasions it *is* equal to 0 the attacker will know it because he observed the $\sqcap$-branch along the way. The program $h{:\in}\,\{0,1\}$ is however not susceptible to the $X{=}\{0\}$ -attack, because its nondeterminism is atomic and the attacker cannot observe it: a single step in the debugger takes him directly from before- to after the statement, and the resolution of "$:\in$" is hidden.

Having defined a test for security, we should be able to find a corresponding definition of secure refinement to follow. 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 the values that an attacker must concede h could possibly have at the end of this run. [8] We call H the *shadow* of h because, in effect,

[8] We are here of course relying on the fact that the constructions in §4 correctly modelled the attackers

it records the values of h resulting from $:\in$ -outcomes that could have been taken on this run, even if in fact they were not.

Taking the two previous paragraphs together, then, we have immediately that in our context secure refinement implies that for any output shadow H_I possible for the implementation there must be a possible "supporting" output shadow H_S for the specification, where *supporting* means precisely that $H_S \subseteq H_I$. This is exactly what is needed to establish that for any H_I, X with $H_I \subseteq X$, a successful X-attack on the implementation, there must also be a successful X-attack possible on the specification: there must also be an output shadow H_S with $H_S \subseteq X$.

Taking the above all together, we have our first tentative definition of secure refinement in total. Specification S is refined by implementation I, written $S \sqsubseteq I$ as usual, just when

(1)
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 $H_S \subseteq H_I$ by some final shadow H_S produced by S.

We will see in a moment that this is actually not yet the right definition: it needs to be strengthened. But it is a good first step.

6 Compositionality and monotonicity

We saw earlier that *compositionality* is the property that the meaning of a compound can be deduced from the meanings of its components alone. In some cases compositionality is not easy to achieve: an informal example is as follows [16].

Parents' eye colour, on its own, cannot accurately predict the distribution of eye colour among their children: some brown-eyed parents are virtually certain to produce brown-eyed children; other 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 (as far as we know). 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 "domi-

observational capabilities.

nant" 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.

The problem with our first definition (1) of refinement is that, although it correctly captures our focus on secure refinement (cf. our focus on eye-colour), it is not compositional. We will explore that via a series of examples.

6.1 First example, and consequential adjustment of refinement

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

Consider the context $(\cdot\,;v{:=}\,h)$, which produces respectively the programs $h{:=}\,0; v{:=}\,h$ and $h{:\in}\,\{0,1\}; v{:=}\,h$. The second of those can produce the v-output 1; but the first cannot. Thus the second program *in context* does not refine the first (in that same context). Our informal 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 could then see after all. So we adjust our definition of refinement to account for that, getting

(2)

classical v — Any final visible v that can be produced by I can also be produced by S,

classical h — Any final hidden h 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 $H_S{\subseteq}H_I$ by some final shadow H_S produced by S.

6.2 Second example, and...

Our second example concerns $h{:\in}\,\{0,1\}; v{:=}\,h$ and $h{:\in}\,\{0,1\}; v{:=}\,1{-}h$. According to (our revised) (2) they are equal (i.e. inter-refinable): both produce as output-v's of 0 and 1; both produce output-h's of 0 and 1; and both produce output shadows 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-output of 0 or 2, while the second produces only 1. We fix that by coupling the attacker's potential observations of v, h, giving

(3)

classical v, h — Any final visible v, h 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 $H_S{\subseteq}H_I$ by some final shadow H_S produced by S.

6.3 Third example...

Our third example concerns the programs $(v{:=}\,0; h{:=}\,0 \sqcap 1) \sqcap (v{:=}\,1; h{:\in}\,\{0,1\})$ and $(v{:=}\,0; h{:\in}\,\{0,1\}) \sqcap (v{:=}\,1; h{:=}\,0 \sqcap 1)$ where, to reduce clutter, we are abbreviating $h{:=}\,0 \sqcap h{:=}\,1$ as $h{:=}\,0 \sqcap 1$ — note therefore that this "syntactic-sugared" assignment is not atomic. [9] According to (3) the two programs are equal: both produce as output-(v,h)'s all four possibilities from $\mathcal{V}{\times}\mathcal{H}$; and both produce all possible output shadows, thus $\{0\}$, $\{1\}$ and $\{0,1\}$. And yet...

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

(4) **hybrid** v, h, H — Any final triple (v,h,H_I) produced by I can be supported via a triple (v,h,H_S), produced by S, with $H_S{\subseteq}H_I$.

This version of refinement is –finally– correct; in the next section we explore how to justify that claim, and what its implications are.

7 Judging refinement's suitability; inducing a semantics

At the end of the last section we claimed that (4) was "correct." That means, first of all, that it is sufficient to distinguish programs according to our two original criteria at (1). That is because if two programs differ according to (1) then they certainly differ according to (4) as well. (If two people have different eye-colours, then certainly they have different eye-colour alleles.)

Second, our program contexts are monotonic with respect to (4). 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 monotonic separately, and then concluding inductively that all finite contexts built from those constructs are monotonic too. (Here we have an advantage over Mendel: he cannot proceed by induction to "all possible people." Instead our belief in compositionality of alleles rests on a careful observation of the process.)

The two properties above are essential. Desirable is as well is a third property that our refinement definition is as simple as possible. (The above two properties can be understood to be saying "... but not too simple.") To show the optional property of simplicity we argue that if any programs are distinguished by our sophisticated (4) then there is a *context* within which they can in fact be distinguished by our original (1). (Alleles are as simple as it gets for explaining 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 suppose that program I yields a triple $(\hat{v}, \hat{h}, H_I)$ but there is no $H_S{\subseteq}H_I$ such that program S yields a triple $(\hat{v}, \hat{h}, H_S)$. We conclude from (4) that $S \not\sqsubseteq I$. How do we convince ourselves that that is a distinction worth making?

[9] That is, our syntactic convention is that $h{:\in}\,\{0,1\}$ is atomic but $h{:=}\,0 \sqcap 1$ is not.

The program-in-context $(I;\,\textbf{if}\ v{\neq}\hat{v} \vee h{\neq}\hat{h}\ \textbf{then}\ h{:}{\in}\,\mathcal{H}\ \textbf{fi})$ can produce a shadow H_I, by assumption; but the matching $(S;\,\textbf{if}\ v{\neq}\hat{v} \vee h{\neq}\hat{h}\ \textbf{then}\ h{:}{\in}\,\mathcal{H}\ \textbf{fi})$ cannot produce a supporting $H_S{\subseteq}H_I$, which shows it is indeed worth distinguishing according to our original criterion (1). (The point of the **if**... is to "smother" the other (v,h)-outputs of S so that they cannot supply the needed supporting H_S.[10])

(5) With our refinement definition now secured, we **induce our programs' semantics**, given an initial state, to be just the set of attacks that can succeed against it. This is the standard approach, ideal when it can be applied, because then refinement boils down to simple reverse-inclusion of those sets: if $S \sqsubseteq I$ then any attack that succeeds against I must (potentially) succeed against S. Because our semantics has the optional simplicity-property, we can say that it is *fully abstract.*

8 Distilling the essence of traces leaves The Shadow Semantics

The model of §3 was chosen subjectively, [11] being made comprehensive enough 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 §§5–7, and in particular the final definition of refinement (4), determined for us just what information we needed to keep (alleles) and, by implication, what we could throw away. With that, and with (5) in mind, we determine the semantics of a command by taking an initial state, generating all traces (§3), calculating the equivalence (§4), keeping only the triples (v,h,H) that we need for compositionality of refinement (4), and "up-closing" in the third, shadow component, because if an attack "I think h is in X" succeeds, then so will any (other) attack X' with $X{\subseteq}X'$. Thus a command P's semantics $[\![P]\!]$ is of type $\mathcal{V}{\to}\mathcal{H}{\to}\mathbb{P}\mathcal{H} \to \mathbb{P}(\mathcal{V}{\times}\mathcal{H}{\times}\mathbb{P}\mathcal{H})$, with the "$\mathbb{P}$" on the output side reflecting possible nondeterminism, [12] and an *up-closure* condition that if $(v',h',H') \in [\![P]\!].v.h.H$ and $H'{\subseteq}H''$, then also $(v',h',H'') \in [\![P]\!].v.h.H$.

(6) 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 $\mathcal{H}$ of h be $\{0,1,2\}$ and consider the program $h{:}{\in}\{0,1\} \sqcap h{:}{\in}\{1,2\}$ which, seen more closely, becomes

$$_{[p_0]}\quad (_{[p_1]}\ h{:}{\in}\{0,1\}\ _{[p_2]})\ \sqcap\ (_{[p_3]}\ h{:}{\in}\{1,2\}\ _{[p_4]})\quad _{[p_5]}\ .$$

[10] A special case is when H_I is all of $\mathcal{H}$, where our smothering trick fails; but then program S can have no $(\hat{v},\hat{h},H_S)$ output at all. So we use the classical context $(;\ \textbf{if}\ v{=}\hat{v} \wedge h{=}\hat{h}\ \textbf{then}\ v{:=}1\ \textbf{else}\ v{:=}0\ \textbf{fi})$ which sets v to 0 after executing S; after I however v can be set to 1.

[11] That is, we have only a subjective notion of its being the right one: there is no rigorous way we can prove that.

[12] This is effectively the same as $(\mathcal{V}{\times}\mathcal{H}{\times}\mathbb{P}\mathcal{H}) \to \mathbb{P}(\mathcal{V}{\times}\mathcal{H}{\times}\mathbb{P}\mathcal{H})$, but the "spread-out" Curried arguments generate less parenthesis-clutter in use. Another familiar alternative is $(\mathcal{V}{\times}\mathcal{H}{\times}\mathbb{P}\mathcal{H}) \leftrightarrow (\mathcal{V}{\times}\mathcal{H}{\times}\mathbb{P}\mathcal{H})$; but this makes it messier to state the healthiness conditions subsequently.

with the bracketed notations being program-counter values. From initial state (v_0, h_0, p_0) this generates the four traces

$$\begin{array}{ll}
\langle (v_0,h_0,p_0),(v_0,h_0,p_1),(v_0,0,p_2),(v_0,0,p_5)\rangle\ , & \text{in equivalence class } A\text{, say}\\
\langle (v_0,h_0,p_0),(v_0,h_0,p_1),(v_0,1,p_2),(v_0,1,p_5)\rangle\ , & \text{also in class } A\\
\langle (v_0,h_0,p_0),(v_0,h_0,p_3),(v_0,1,p_4),(v_0,1,p_5)\rangle\ , & \text{in class } B\\
\langle (v_0,h_0,p_0),(v_0,h_0,p_3),(v_0,2,p_4),(v_0,2,p_5)\rangle\ , & \text{in class } B
\end{array}$$

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

$$\begin{array}{ll}
(v_0,0) & \text{in class } A\\
(v_0,1) & \text{in class } A\\
(v_0,1) & \text{in class } B\\
(v_0,2) & \text{in class } B\ \ ,
\end{array}$$

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-value v_0 the two H-sets –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:

$$(7)\qquad \begin{array}{l}(v_0,0,\{0,1\})\\(v_0,1,\{0,1\})\\(v_0,1,\{1,2\})\\(v_0,2,\{1,2\})\end{array} \qquad \text{and} \qquad \begin{array}{l}(v_0,0,\{0,1,2\})\\(v_0,1,\{0,1,2\})\\(v_0,2,\{0,1,2\})\end{array}$$

If we concentrate on the minimal (left-hand) triples in (7), 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 "unknown knowns."

In other words, at the beginning of the run he cannot predict the resolution of the first nondeterminism $\sqcap$; 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 $:\in$ was resolved (in either case) — but 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.

9 Healthiness conditions

In the previous section (§8) we finally managed to distil a "lean" semantics (sets of triples) from the operationally motivated guesswork (§3) with which we began. 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; it was motivated by the notion that if a given attack succeeds, then so should any stronger attack. Thus although the space of program results, the "square type," is $\mathbb{P}(\mathcal{V}\times\mathcal{H}\times\mathbb{P}\mathcal{H})$, we do not consider all possible sets (i.e. elements of the powerset) to be healthy: only the up-closed ones are. [13]

That technique of up-closure is a very common feature of program semantics. For example, in sequential relational semantics $S\rightarrow\mathbb{P}S_\perp$ modelling both nontermination (with $\perp$) and nondeterminism (with $\mathbb{P}$), 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 $s_1 \sqsubset s_2$ just when $s_1{=}\perp{\neq}s_2$. [14]

Similarly in sequential *and probabilistic* relational semantics [9,19,14], where the output sets contain subdistributions Δ over S, up-closure requires that if subdistribution Δ can be produced by a program and we have $\Delta{<}\Delta'$, then we consider also Δ' to be produced (potentially) by that program. [15]

Finally, in the failures model of *CSP* a process is considered to be a set of trace-failure pairs (s, X), the attack 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 (5), 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 eschewing up-closure is the Z-schema [8], whose type

[13] 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.)

[14] More precisely, it is the Smyth powerdomain-order on $\mathbb{P}S_\perp$ generated from the underlying flat order on $S_\perp$ itself [21]. Fluffing-up's being a specific instance of the general Smyth-technique is the reason it works so well.

[15] A *subdistribution* sums to no more than one, rather than to one exactly: any deficit represents the probability of nontermination (abort). Thus when $\Delta{<}\Delta'$ the greater subdistribution assigns at least the same probability to all proper outcomes, and strictly less probability to abort [19,14].

(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 $Z_1{\sqsubseteq}Z_2$ just when

$$\mathsf{dom}.Z_1 \subseteq \mathsf{dom}.Z_2 \quad \wedge \quad (\mathsf{dom}.Z_1){\lhd}Z_2 \subseteq Z_1 \ .$$

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

Another example where up-closure is not used is weakest preconditions, whose semantic space is $\mathbb{P}S \to \mathbb{P}S$ 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 $wp.P.(post_1 \wedge post_2) = wp.P.post_1 \wedge wp.P.post_2$ — and this is because they are constructed from the underlying, and more operational relational semantics. For a relational program (semantics) $r \in S \to S_\perp$ we define $wp.r.post$ to be $\{s{:}\,S \cdot \left(\forall s'{:}\,S \mid s'{\in}r.s \cdot s'{\in}post\right)\}$, where here we are interpreting predicate *post* as the subset of S it denotes. Now $wp.r$ is of type $\mathbb{P}S \to \mathbb{P}S$, as it should be; but function *wp* is itself not surjective. That is, the image of the set $S \to S_\perp$ through the function *wp* is exactly the subset of $\mathbb{P}S \to \mathbb{P}S$ 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 [19,14].

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 of 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 $\mathbb{P}(\mathcal{V}{\times}\mathcal{H}{\times}\mathbb{P}\mathcal{H})$. 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 $R{:=}\,[\![P]\!].v.h.H$ of triples we must have that

(8)
(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 $(v', h'', H') \in R$.

It is because of the necessity condition that we should drop up-closure. If in Example (7) the type $\mathcal{H}$ 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 (4) 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 (8) a higher priority.

10 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.

10.1 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 **hid** h the local block

$$[\![\ \textbf{vis}\ v'; \textbf{hid}\ h';\quad h' {:\in} \{0,1\}; v' {:=} h \oplus h'\ ]\!] \tag{9}$$

is equivalent to **skip** — it changes nothing; and it reveals nothing. This is not surprising: a coin h is (already) hidden; a second temporary hidden coin h' is introduced and flipped; and it is published in 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 [3,17] and the Oblivious Transfer [20,18]) 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 [15] or a generalised substitution [1]:

$$v', h' {:} [v' \oplus h' = h]$$

$$\text{or} \qquad \mathit{ANY}\ v_1, h_1\ \ \mathit{WHERE}\ v_1 \oplus h_1 = h\ \ \mathit{THEN}\ v', h' {:=} v_1, h_1\ \ \mathit{END}\ .$$

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 (9), giving

$$[\![\ \textbf{vis}\ v'; \textbf{hid}\ h';\quad v' {:\in} \{0,1\}; h' {:=} h \oplus v'\ ]\!]\ . \tag{10}$$

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 our semantics identifies $v' {:\in} \{0,1\}$

and $v' := 0 \sqcap 1$ when v' is visible, whether local or not.

The question is then whether (9) and (10) really are equal. They should be, by transitivity of equality since they are both decompositions of $(v' \oplus h') := h$. On the other hand, in (9) the nondeterminism is hidden, yet in (10) it's visible. *Obviously* that means they are different — but in fact they are the same. [16]

10.2 Revelations, and "effectively visible" values

The *revelation command* **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 $[\![$ **vis** v; $v := E$ $]\!]$. We say that a value E is *effectively visible* at a point in a program just when adding a statement **reveal** E at that point does not change the program's meaning [13]. If in particular a hidden h is effectively visible at some point, then in reasoning about the program at that point we can treat 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 $\star$ in the program fragment $v := h \star$ because the two programs $v := h$ and $v := h$; **reveal** h are equal. That's no surprise. But it is also true that the two programs $v := h$ and **reveal** h; $v := h$ are equal, and so do we conclude that h is effectively visible at $\star$ in $\star\ v := h$? *Obviously* variable h is not effectively visible *before* it has been assigned to v — but in fact it is.

10.3 Effective visibility and nondeterminism

Let's accept h's effective visibility at $\star$ in $\star\ v := h$ — perhaps reasoning that, although the attacker does not actually know h at the time $\star$ 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 $\star$ (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 $\star$ ($v := h \sqcap$ **skip**). Is h effectively visible this time? After all, it is no longer certain that it will be revealed by the assignment, since the **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 $v := h \sqcap$ **skip** and **reveal** h; ($v := h \sqcap$ **skip**) — and *obviously* they are equal, for the reasons just mentioned. But in fact they are not.

10.4 Local blocks and cover-ups

As our last example we modify the previous one by making v local: is h effectively visible in $\star$ $[\![$ **vis** v; $v := h \sqcap$ **skip** $]\!]$? *Obviously* making v local has no effect, since it

[16] This might not surprise everyone; but it surprised us. Unfortunately you can't surprise all of the people all of the time.

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 §10.3 it was not.

11 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 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.

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 formal method itself. Our story is an example of *Meta Formal Methods*.

Acknowledgement

Thanks to Eerke Boiten for careful reading and helpful comments, and to John Derrick (as well) for providing the opportunity and incentive to write this account.

References

[1] Abrial, J.-R., "The B Book: Assigning Programs to Meanings," Cambridge University Press, 1996.

[2] Back, R.-J., *On the correctness of refinement steps in program development*, Report A-1978-4, Dept Comp Sci, Univ Helsinki (1978).

[3] Chaum, D., *The Dining Cryptographers problem: Unconditional sender and recipient untraceability*, J. Cryptol. **1** (1988), pp. 65–75.

[4] Dijkstra, E., "A Discipline of Programming," Prentice-Hall, 1976.

[5] Floyd, R., *Assigning meanings to programs*, in: J. Schwartz, editor, *Mathematical Aspects of Computer Science*, number 19 in Proc Symp Appl Math., American Mathematical Society, 1967 pp. 19–32.

[6] Goguen, J. and J. Meseguer, *Unwinding and inference control*, in: *Proc IEEE Symp on Security and Privacy*, 1984, pp. 75–86.

[7] Halpern, J. and K. O'Neill, *Secrecy in multiagent systems*, in: *Proc 15th IEEE Computer Security Foundations Workshop*, 2002, pp. 32–46.

[8] Hayes, I., "Specification Case Studies," Prentice-Hall, 1987, `http://www.itee.uq.edu.au/~ianh/Papers/SCS2.pdf`.

[9] He, J., K. Seidel and A. McIver, *Probabilistic models for the guarded command language*, Science of Computer Programming **28** (1997), pp. 171–92.

[10] Hoare, C., *An axiomatic basis for computer programming*, Comm ACM **12** (1969), pp. 576–80, 583.

[11] Jacob, J., *Security specifications*, in: *IEEE Symposium on Security and Privacy*, 1988, pp. 14–23.

[12] Jones, C., "Systematic Software Development using VDM," Prentice-Hall, 1986.

[13] McIver, A., *The secret art of computer programming*, in: *Proc. ICTAC 2009*, 2009, invited presentation.

[14] McIver, A. and C. Morgan, "Abstraction, Refinement and Proof for Probabilistic Systems," Tech Mono Comp Sci, Springer, New York, 2005.
URL http://www.cse.unsw.edu.au/~carrollm/arp/

[15] Morgan, C., "Programming from Specifications," Prentice-Hall, 1994, second edition, web.comlab.ox.ac.uk/oucl/publications/books/PfS/.

[16] Morgan, C., *Of probabilistic wp and CSP*, in: A. Abdallah, C. Jones and J. Sanders, editors, *Communicating Sequential Processes: The First 25 Years*, Springer, 2005 .

[17] Morgan, C., *The Shadow Knows: Refinement of ignorance in sequential programs*, in: T. Uustalu, editor, *Math Prog Construction*, Springer **4014** (2006), pp. 359–78, treats *Dining Cryptographers.*

[18] Morgan, C., *The Shadow Knows: Refinement of ignorance in sequential programs*, Science of Computer Programming **74** (2009), treats *Oblivious Transfer.*

[19] Morgan, C., A. McIver and K. Seidel, *Probabilistic predicate transformers*, ACM Trans Prog Lang Sys **18** (1996), pp. 325–53,
doi.acm.org/10.1145/229542.229547.

[20] Rivest, R., *Unconditionally secure commitment and oblivious transfer schemes using private channels and a trusted initialiser*, Technical report, M.I.T. (1999), //theory.lcs.mit.edu/~rivest/Rivest-commitment.pdf.

[21] Smyth, M., *Power domains*, Jnl Comp Sys Sci **16** (1978), pp. 23–36.

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

The healthiness conditions (8) 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 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

$$R \quad = \quad \mathsf{addHid}.P \quad := \quad \{v, h, H \mid (v, H){\in}P \wedge h{\in}H\} \ .$$

Because of reflexivity (8), this construction addHid does not miss any triples; because of necessity it doesn't add too many. The type of programs' semantics would then become the simpler $\mathcal{V}{\to}\mathbb{P}\mathcal{H}{\to}\mathbb{P}(\mathcal{V}{\times}\mathbb{P}\mathcal{H})$; but of course the refinement relation would have to change.

Luckily we do not have to torture ourselves by inventing the pair-appropriate refinement relation $\sqsubseteq_2$ all over again from first principles, retracing the long route for the triple-appropriate relation $\sqsubseteq_3$ (call it) that we finally defined at (4). Instead we simply say that $P \sqsubseteq_2 P'$ just when $\mathsf{addHid}.P \sqsubseteq_3 \mathsf{addHid}.P'$.

We can work out the definition of $\sqsubseteq_2$ by calculation. Writing P_v for the projection $\{H \cdot (v, H){\in}P\}$, we have

$P \sqsubseteq_2 P'$

iff $\quad \mathsf{addHid}.P \sqsubseteq_3 \mathsf{addHid}.P'$ "definition $\sqsubseteq_2$"

iff $\quad \{v,h,H \mid H{\in}P_v \wedge h{\in}H\} \;\sqsubseteq_3\; \{v,h,H \mid H{\in}P'_v \wedge h{\in}H\}$ "definition addHid"

iff "definition $\sqsubseteq_3$"

$$\left(\forall v,h,H' \mid H'{\in}P'_v \wedge h{\in}H' \cdot \left(\exists H \mid H{\in}P_v \cdot h{\in}H \wedge H{\subseteq}H'\right)\right)$$

iff "rearrange quantifiers"

$$\left(\forall v,H'{:}\,P'_v \cdot \left(\forall h{:}\,H' \cdot \left(\exists H \mid H{\in}P_v \cdot h{\in}H \wedge H{\subseteq}H'\right)\right)\right)$$

iff for all v we have $\quad \left(\forall h{\in}H'{:}\,P'_v \cdot \left(\exists H{:}\,P_v \cdot h{\in}H{\subseteq}H'\right)\right)$, "compact"

which means that for every v, each set in P'_v must be the union of some collection of sets occurring in P_v.

This refinement relation $\sqsubseteq_2$ is not as simple as $\sqsubseteq_3$ because each pair in the implementation could potentially need support from a *collection* of pairs in the specification (a collection whose H_S components' union equals the single H_I 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 $H_{\{1,2\}}$ by a single shadow $H_1{\cup}H_2$ is exactly what happens when we replace a visible choice $\sqcap$ by a hidden choice $:\in$ so that e.g. $h{:=}\,0 \sqcap 1$, with its two shadows $\{0\}$ and $\{1\}$, refines to $h{:\in}\,\{0,1\}$ with its single shadow $\{0,1\}$. [17]

Thus $(\sqcap) \sqsubseteq (:\in)$ is the "essence" of security refinement.

Perhaps this is simpler overall?

As a final example, we ask whether $h{:\in}\,\{0,1\} \sqcap h{:\in}\,\{2,3\}$ is refined by $h{:\in}\,\{0,1,2\}$. *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 $(v_0, 2, \{0,1,2\})$ on the right has no support on the left: on the one hand, neither $(v_0, 0, \{0,1\})$ nor $(v_0, 1, \{0,1\})$ will do because, although $\{0,1\} \subseteq \{0,1,2\}$, their h-components 0,1 do not include 2; on the other hand, $(v_0, 2, \{2,3\})$ will not do because $\{2,3\} \not\subseteq \{0,1,2\}$.

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

[17] 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 Brief comments on the interesting examples

B.1 On §10.1

Operationally, the first formulation (9) 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 (10) 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 §10.2

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 at his leisure. The point is that he can deduce the value h had at the point indicated, even if that deduction is later.

B.3 On §10.3

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

$$h{:\in}\,\mathcal{H}{-}\{v\};\quad v{:=}\,h \sqcap \textbf{skip};\qquad \textbf{if}\ v{=}h\ \textbf{then}\ h{:\in}\,\mathcal{H}\ \textbf{fi}$$

$$\text{and}\quad h{:\in}\,\mathcal{H}{-}\{v\};\quad \textbf{reveal}\ h; (v{:=}\,h \sqcap \textbf{skip});\quad \textbf{if}\ v{=}h\ \textbf{then}\ h{:\in}\,\mathcal{H}\ \textbf{fi}\ .$$

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 §10.4

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 **skip**.