Science of Computer Programming 74 (2009) 629–653

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

The Shadow Knows: Refinement and security in sequential programs[☆]

Carroll Morgan

School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia

ARTICLE INFO

Article history:
Received 14 February 2007
Received in revised form 9 May 2007
Accepted 22 September 2007
Available online 11 February 2009

Keywords:
Security
Privacy
Hoare logic
Specification
Implementation
Logic of knowledge

ABSTRACT

Stepwise *refinement* is a crucial conceptual tool for system development, encouraging program construction via a number of separate correctness-preserving stages which ideally can be understood in isolation. A crucial conceptual component of *security* is an adversary's ignorance of concealed information. We suggest a novel method of combining these two ideas.

Our suggestion is based on a mathematical definition of "ignorance-preserving" refinement that extends classical refinement by limiting an adversary's access to concealed information: moving from specification to implementation should never increase that access. The novelty is the way we achieve this in the context of sequential programs.

Specifically we give an operational model (and detailed justification for it), a basic sequential programming language and its operational semantics in that model, a "logic of ignorance" interpreted over the same model, then a program-logical semantics bringing those together — and finally we use the logic to establish, via refinement, the correctness of a real (though small) protocol: Rivest's *Oblivious Transfer*. A previous report[*] treated Chaum's *Dining Cryptographers* similarly.

In passing we solve the *Refinement Paradox* for sequential programs.

1. Introduction

Stepwise refinement is an idealised process whereby program- or system development is "considered as a sequence of design decisions concerning the decomposition of tasks into subtasks and of data into data structures" [2]. We say "idealised" because, as is well known, in practice it is almost never fully achieved: the realities of vague- and changing requirements, of efficiency *etc.* simply do not cooperate. Nevertheless as a principle to aspire to, a way of organising our thinking, its efficacy is universally recognised: it is an instance of *separation of concerns*.

A second impediment to refinement (beyond "reality" as above) is its scope. Refinement was originally formalised as a relation between sequential programs [3], based on a state-to-state operational model, with a corresponding logic of Hoare-triples $\{\Phi\}$ S $\{\Psi\}$ [4] or equivalently weakest preconditions $wp.\mathrm{S}.\Psi$ [5]. As a relation, it generates an algebra of (in-)equations between program fragments [6,7].

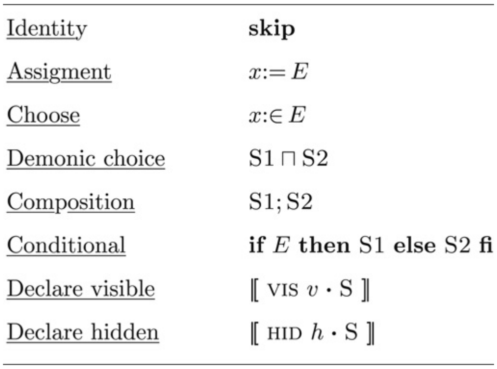
Thus a specification S1 is said to be *refined by* an implementation S2, written S1 $\sqsubseteq$ S2, just when S2 preserves all logically expressible properties of S1. The *scope* of the refinement is determined by that expressivity: originally limited to sequential programs [3], it has in the decades since been extended in many ways (*e.g.* concurrency, real-time and probability).

Security is our concern of scope here. The extension of refinement in this article is based on identifying and reasoning about an adversarial observer's "ignorance" of data we wish to keep secret, to be defined as his uncertainty about the parts of

[☆] This is an extension of an earlier report [C. Morgan, The Shadow Knows: Refinement of ignorance in sequential programs, in: T. Uustalu (Ed.), Proc. of 8th Int. Conf. on Mathematics of Program Construction, MPC 2006, Kuressaare, July 2006, in: Lect. Notes in Comput. Sci., vol. 4014, Springer, Berlin, 2006, pp. 359–378].

E-mail address: carrollm@cse.unsw.edu.au.

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2007.09.003



Identity	**skip**
Assigment	$x{:=}\,E$
Choose	$x{:\in}\,E$
Demonic choice	$\mathrm{S1} \sqcap \mathrm{S2}$
Composition	$\mathrm{S1}; \mathrm{S2}$
Conditional	**if** E **then** S1 **else** S2 **fi**
Declare visible	$[\![\ \textsc{vis}\ v \cdot \mathrm{S}\ ]\!]$
Declare hidden	$[\![\ \textsc{hid}\ h \cdot \mathrm{S}\ ]\!]$

Fig. 1. Programming language syntax.

the program state he cannot see. Thus we consider a program of known source-text, with its state partitioned into a "visible" part v and a "hidden" part h, and we ask

From the initial and final values of visible v, what can an adversary deduce about hidden h? (1)

For example, if the program is $v{:=}\,0$, then what he can deduce afterwards about h is only what he knew beforehand; but if it is $v{:=}\,h \bmod 2$, then he has learned h's parity; and if it is $v{:=}\,h$ then he has learned h's value exactly.

We assume initially (and uncontroversially) that the adversary has at least the abilities implied by (1) above, *i.e.* knowledge of v's values before/after execution and knowledge of the source code. We see below, however, that if we make certain reasonable, practical assumptions about refinement – which we shall justify – then surprisingly we are forced to assume *as well* that the adversary can see both program flow and visible variables' intermediate values: that is, there are increased adversarial capabilities *wrt* ignorance that accrue *as a consequence* of using refinement.[1] Thus refinement presents an increased security risk, a challenge for maintaining correctness: but it is so important, as a design tool, that the challenge is worth meeting.

The problem is in essence that classical refinement [3,6–8] is indeed insecure in the sense that it does not preserve ignorance [9]. If we assume v, h both to have type T, then "choose v from T" is refinable into "set v to h" — as such it is simply a reduction of demonic nondeterminism. But that refinement, which we write $v{:\in}\,T \ \sqsubseteq\ v{:=}\,h$, is called the "Refinement Paradox" (Section 7.3.2) precisely because it does not preserve ignorance: program $v{:\in}\,T$ tells us nothing about h, whereas $v{:=}\,h$ tells us everything. To integrate refinement and security we must address this paradox at least.

Our first contribution is a collection of "refinement principles" that we claim any reasonable *ignorance* refinement algebra [6,7] must adhere to if it is to be practical. Because *basic* principles are necessarily subjective, we give a detailed argument to support our belief that they are important (Section 2).

Our second, and main contribution is to realise those principles: our initial construction for doing so includes a description of the adversary (Section 3), a state-based model incorporating the adversary's capabilities (Section 4.2), a small programming language (Fig. 1) with its operational semantics over that model (Fig. 2) and a thorough treatment of indicative examples (Figs. 3–5).

In support of our construction we give beforehand a detailed rationale for our design (Section 2), and an argument via abstraction that induces our model's features from that rationale by factoring it through Kripke structures (Section 4.1).

Based on our construction we then give a logic for our proposed model (Section 6), a programming-logic based on that and the operational semantics (Sections 7 and 8), and examples of derived algebraic techniques (Sections 9 and 10).

Within the framework constructed as above we interpret *refinement* both operationally (Section 5.3) and logically (Sections 7.2 and 8.1), which interpretations are shown to agree (again Section 7.2) and to satisfy the Principles (Section 8.4).

Ignorance-preserving refinement should be of great utility for developing zero-knowledge- or security-sensitive protocols (at least); and our final contribution (Section 11) is an example, a detailed refinement-based development of Rivest's Oblivious Transfer Protocol [10].

2. Refinement principles: Desiderata for practicality

The general theme of the five *refinement principles* we now present is that local modifications of a program should require only local checking (plus if necessary accumulation of declarative context hierarchically): any wholesale search of the entire source code must be rejected out of hand.

RP0 Refinement is monotonic — If one program fragment is shown in isolation to refine another, then the refinement continues to hold in any context: robustness of local reasoning is of paramount importance for scaling up.

[1] In the contrapositive, what we will argue is that if we integrated refinement and ignorance *without* admitting the adversary's increased capabilities, we would not get a reasonable theory.

RP1 *All classical "visible-variable only" refinements remain valid* — It would be impractical to search the entire program (for hidden variables) in order to validate local reasoning (*i.e.* in which the hiddens elsewhere are not even mentioned).

RP2 *All classical "structural" refinements remain valid* — Associativity of sequential composition, distribution of code into branches of a conditional *etc.* are refinements (actually equalities) that do not depend on the actual code fragments affected: they are *structurally* valid, acting *en bloc*. It would be impractical to have to trawl through their interiors (including *e.g.* procedure calls) to validate such familiar rearrangements. (Indeed, the procedures' interiors might by unavailable due to modularisation.)

RP3 *Some classical "explicit-hidden" refinements become invalid* — This is necessary because, for example, the Refinement Paradox is just such a refinement. Case-by-case reasoning must be justified by a model and a compatible logic.

RT *Referential transparency* — If two expressions are equal (in a declarative context) then they may be exchanged (in that context) without affecting the meaning of the program. This is a crucial component of any mathematical theory with equality.

3. Description of the adversary: *Gedanken* experiments

We initially assume minimal, "weak" capabilities (1) of our adversary, but show via *gedanken* experiments based on our Principles (Section 2) that if we allow refinement then we must assume the "strong" capabilities defined below.

3.1. The weak adversary in fact has perfect recall

We ask *Does program $v{:=}\,h;\ v{:=}\,0$ reveal h* to the weak adversary? According to our principles above, it must; we reason

$$\begin{array}{lll}
 & (v{:=}\,h;\ v{:=}\,0);\ v{:\in}\,T & \\
= & v{:=}\,h;\ (v{:=}\,0;\ v{:\in}\,T) & \text{"}\mathit{RP2}\text{: associativity"} \\
= & v{:=}\,h;\ v{:\in}\,T & \text{"}\mathit{RP1}\text{: visibles only"} \\
\sqsubseteq & v{:=}\,h;\ \mathbf{skip} & \text{"}\mathit{RP1}\text{: visibles only; }\mathit{RP0}\text{"} \\
= & v{:=}\,h, & \text{"}\mathit{RP2}\text{: }\mathbf{skip}\text{ is the identity"}
\end{array} \qquad (2)$$

whence we conclude that, since the implementation ($v{:=}\,h$) fails to conceal h, so must the specification ($v{:=}\,h;\ v{:=}\,0;\ v{:\in}\,T$) have failed to do so. Our model must therefore have *perfect recall* [11], because escape of h into v is not "erased" by the v-overwriting $v{:=}\,0$. That is what allows h to be "copied" by the final $v{:\in}\,T$.[2]

3.2. The weak adversary in fact can observe program flow

Now we ask *Does program $h{:=}\,0\ \sqcap\ h{:=}\,1$ reveal h* to the weak adversary? Again, according to our principles, it must: we reason

$$\begin{array}{lll}
 & (h{:=}\,0 \sqcap h{:=}\,1);\ v{:\in}\,T & \\
= & (h{:=}\,0;\ v{:\in}\,T)\ \sqcap\ (h{:=}\,1;\ v{:\in}\,T) & \text{"}\mathit{RP2}\text{: distribution }\sqcap\text{ and ;"} \\
\sqsubseteq & (h{:=}\,0;\ v{:=}\,0)\ \sqcap\ (h{:=}\,1;\ v{:=}\,1) & \text{"}\mathit{RP1}\text{; }\mathit{RP2}\text{"} \\
= & (h{:=}\,0;\ v{:=}\,h)\ \sqcap\ (h{:=}\,1;\ v{:=}\,h) & \text{"}\mathit{RT}\text{: have }h{=}0\text{ left, }h{=}1\text{ right"} \\
= & (h{:=}\,0 \sqcap h{:=}\,1);\ v{:=}\,h, & \text{"reverse above steps"}
\end{array} \qquad (3)$$

and 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 **if** E **then skip else skip fi** reveals the Boolean value E.

3.3. The strong adversary: A summary

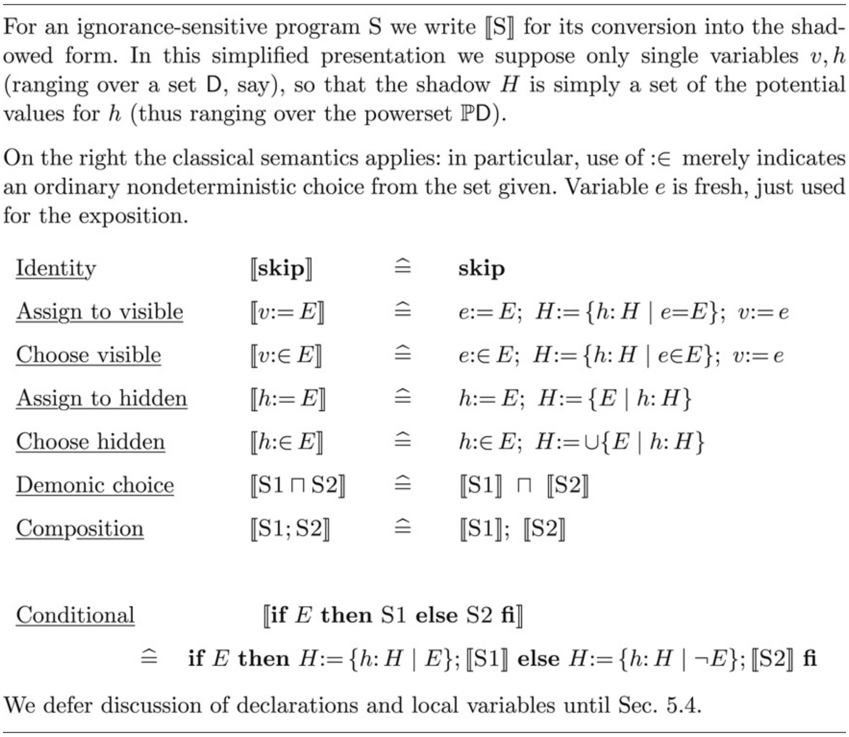
In view of the above, we accept that our of-necessity *strong* adversary must be treated as if he knows at any time what program steps have occurred and what the *visible* variables' values were after each one.

Concerning program flow, we note the distinction between *composite* nondeterminism, written *e.g.* as $h{:=}\,0 \sqcap h{:=}\,1$ and acting *between* syntactic atoms (or larger structures), and *atomic* nondeterminism, written *e.g.* as $h{:\in}\,\{0,\,1\}$ and acting *within* atoms:

- in the composite case, afterwards the adversary knows which of atoms $h{:=}\,0$ or $h{:=}\,1$ was executed, and thus knows the value of h too; yet
- in the atomic case, afterwards he knows only that the effect was to set h to 0 or to 1, and thus knows only that $h{\in}\{0,\,1\}$.

Thus $h{:=}\,0\ \sqcap\ h{:=}\,1$ and $h{:\in}\,\{0,\,1\}$ are different. (Regularity of syntax however allows $v{:=}\,0 \sqcap v{:=}\,1$ and $v{:\in}\,\{0,\,1\}$ as well; but since v is visible, there is no semantic difference between those latter two fragments.)

[2] Adding the extra statement $v{:\in}\,T$ is our algebraic means of detecting information-flow leaks: if a value is accessible to the adversary, then it should be valid to refine $v{:\in}\,T$ so that the value is placed into v.



For an ignorance-sensitive program S we write $[\![\mathrm{S}]\!]$ for its conversion into the shadowed form. In this simplified presentation we suppose only single variables v, h (ranging over a set D, say), so that the shadow H is simply a set of the potential values for h (thus ranging over the powerset $\mathbb{P}\mathsf{D}$).

On the right the classical semantics applies: in particular, use of $:\in$ merely indicates an ordinary nondeterministic choice from the set given. Variable e is fresh, just used for the exposition.

Identity	$[\![\mathbf{skip}]\!]$	$\widehat{=}$	**skip**
Assign to visible	$[\![v{:=}\,E]\!]$	$\widehat{=}$	$e{:=}\,E;\ H{:=}\,\{h{:}\,H \mid e{=}E\};\ v{:=}\,e$
Choose visible	$[\![v{:\in}\,E]\!]$	$\widehat{=}$	$e{:\in}\,E;\ H{:=}\,\{h{:}\,H \mid e{\in}E\};\ v{:=}\,e$
Assign to hidden	$[\![h{:=}\,E]\!]$	$\widehat{=}$	$h{:=}\,E;\ H{:=}\,\{E \mid h{:}\,H\}$
Choose hidden	$[\![h{:\in}\,E]\!]$	$\widehat{=}$	$h{:\in}\,E;\ H{:=}\cup\{E \mid h{:}\,H\}$
Demonic choice	$[\![\mathrm{S1} \sqcap \mathrm{S2}]\!]$	$\widehat{=}$	$[\![\mathrm{S1}]\!]\ \sqcap\ [\![\mathrm{S2}]\!]$
Composition	$[\![\mathrm{S1};\mathrm{S2}]\!]$	$\widehat{=}$	$[\![\mathrm{S1}]\!];\ [\![\mathrm{S2}]\!]$

Conditional $[\![\mathbf{if}\ E\ \mathbf{then}\ \mathrm{S1}\ \mathbf{else}\ \mathrm{S2}\ \mathbf{fi}]\!]$

$$\widehat{=}\quad \mathbf{if}\ E\ \mathbf{then}\ H{:=}\,\{h{:}\,H \mid E\}; [\![\mathrm{S1}]\!]\ \mathbf{else}\ H{:=}\,\{h{:}\,H \mid \neg E\}; [\![\mathrm{S2}]\!]\ \mathbf{fi}$$

We defer discussion of declarations and local variables until Sec. 5.4.

Fig. 2. Operational semantics.

4. A Kripke-based security model for sequential programs

Perfect recall and program flow suggest the Logic of Knowledge and its Kripke models as a suitable conceptual basis for what we want to achieve.

The seminal work on formal logic for knowledge is Hintikka's [12], who used Kripke's possible-worlds semantics for the model: he revived the discussion on a subject which had been a topic of interest for philosophers for millennia. It was first related to multi-agent computing by Halpern and Moses [13], and much work by many other researchers followed. Fagin et al. summarise the field in their definitive introduction [14].

The standard model for knowledge-based reasoning [12–14] is based on possible "runs" of a system and participating agents' ignorance of how the runs have interleaved: although each agent knows the (totality of) the possible runs, a sort of "static" knowledge, he does not have direct "dynamic" knowledge of which run has been taken on any particular occasion. Thus he knows a fact in a given global state (of an actual run) iff that fact holds in all possible global states (allowed by other runs) that have the same local state as his.

For our purposes we severely specialise this view in three ways. The first is that we consider only sequential programs, with explicit demonic choice. As usual, such choice can represent both *abstraction*, that is freedom of an implementor to choose among alternatives (possible refinements), and *ignorance*, that is not knowing which environmental factors might influence run-time decisions.

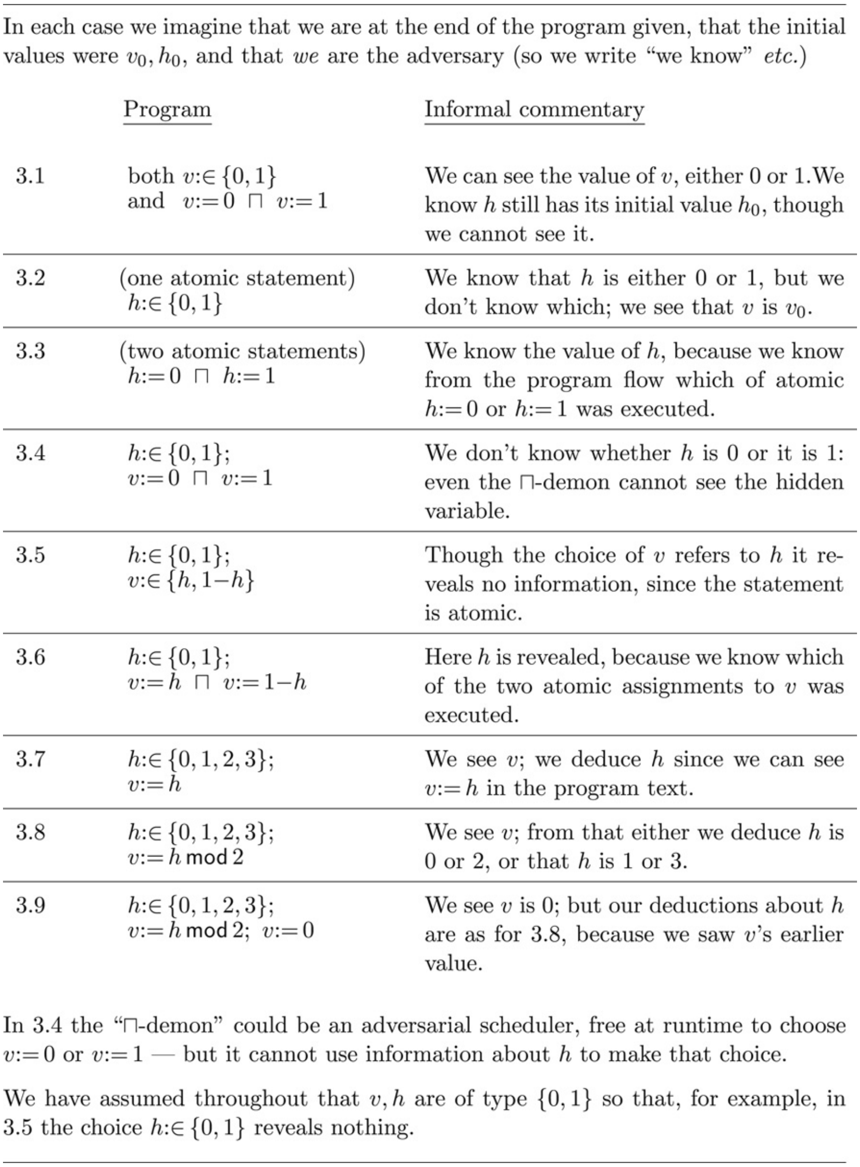
Secondly, we consider only one agent: this is our adversary, whose local state is our system's visible part and who is is trying to learn about (what is for him) the non-local, hidden part.

Finally, we emphasise ignorance rather than knowledge (its dual).

4.1. The model as a Kripke structure

We assume a sequential program text, including a notion of atomicity: unless stated otherwise, each *syntactic* atom changes the program counter when it is executed; semantically, an atom is simply a relation between initial and final states. Demonic choice is either a (non-atomic) choice between two program fragments, thus $\mathrm{S1} \sqcap \mathrm{S2}$, or an (atomic) selection of a variable's new value from some set, thus $x{:\in}\,X$. For simplicity we suppose we have just two (anonymously typed) variables, the visible v and the hidden h.

The global state of the system comprises both v, h variables' current and all previous values, sequences $\overline{v}$, $\overline{h}$, and a history-sequence $\overline{p}$ of the program counter; from Section 3 we assume the adversary can see $\overline{v}$, $\overline{p}$ but not $\overline{h}$. For example, even after



In each case we imagine that we are at the end of the program given, that the initial values were v_0, h_0, and that *we* are the adversary (so we write "we know" *etc.*)

	Program	Informal commentary
3.1	both $v{:\in}\,\{0,1\}$ and $v{:=}\,0 \sqcap v{:=}\,1$	We can see the value of v, either 0 or 1.We know h still has its initial value h_0, though we cannot see it.
3.2	(one atomic statement) $h{:\in}\,\{0,1\}$	We know that h is either 0 or 1, but we don't know which; we see that v is v_0.
3.3	(two atomic statements) $h{:=}\,0 \sqcap h{:=}\,1$	We know the value of h, because we know from the program flow which of atomic $h{:=}\,0$ or $h{:=}\,1$ was executed.
3.4	$h{:\in}\,\{0,1\}$; $v{:=}\,0 \sqcap v{:=}\,1$	We don't know whether h is 0 or it is 1: even the $\sqcap$-demon cannot see the hidden variable.
3.5	$h{:\in}\,\{0,1\}$; $v{:\in}\,\{h, 1{-}h\}$	Though the choice of v refers to h it reveals no information, since the statement is atomic.
3.6	$h{:\in}\,\{0,1\}$; $v{:=}\,h \sqcap v{:=}\,1{-}h$	Here h is revealed, because we know which of the two atomic assignments to v was executed.
3.7	$h{:\in}\,\{0,1,2,3\}$; $v{:=}\,h$	We see v; we deduce h since we can see $v{:=}\,h$ in the program text.
3.8	$h{:\in}\,\{0,1,2,3\}$; $v{:=}\,h \bmod 2$	We see v; from that either we deduce h is 0 or 2, or that h is 1 or 3.
3.9	$h{:\in}\,\{0,1,2,3\}$; $v{:=}\,h \bmod 2$; $v{:=}\,0$	We see v is 0; but our deductions about h are as for 3.8, because we saw v's earlier value.

In 3.4 the "$\sqcap$-demon" could be an adversarial scheduler, free at runtime to choose $v{:=}\,0$ or $v{:=}\,1$ — but it cannot use information about h to make that choice.

We have assumed throughout that v, h are of type $\{0,1\}$ so that, for example, in 3.5 the choice $h{:\in}\,\{0,1\}$ reveals nothing.

Fig. 3. Examples of ignorance, informally interpreted.

S1; (S2 $\sqcap$ S3); S4 has completed he can use $\overline{p}$ to "remember" which of S2 or S3 was executed earlier, and he can use $\overline{v}$ to recall the visible variables' values after whichever it was.

The possible runs of a system S are all sequences of global states that could be produced by the successive execution of atomic steps from some initial v_0, h_0, a tree structure with branching derived from demonic choice (both $\sqcap$ and $:\in$).

If the current state is $(\overline{v}, \overline{h}, \overline{p})$, then the set of *possible* states associated with it is the set of all (other) triples $(\overline{v}, \overline{h}_1, \overline{p})$ that S could (also) have produced from v_0, h_0. We write $(\overline{v}, \overline{h}, \overline{p}) \sim (\overline{v}, \overline{h}_1, \overline{p})$ for this (equivalence) relation of *accessibility*, which depends on S, v_0, h_0.

4.2. An operational model abstracted from the Kripke structure

Because of our very limited use of the Kripke structure, we can take a brutal abstraction of it: programs cannot refer to the full run-sequences directly; what they *can* refer to is just the current values of v, h, and that is all we need keep of them in the abstraction.

For the accessibility relation we introduce a "shadow" variable H, set-valued, which records the possible values of h in all (other) runs that the adversary considers $\sim$-equivalent to (cannot distinguish from) this one; the abstraction to (v, h, H) is thus

	(v,h)-program S	$(\mathsf{v},\mathsf{h},\mathsf{H})$-program $[\![\mathrm{S}]\!]$
4.1a	$v{:\in}\,\{0,1\}$	$e{:\in}\,\{0,1\};\ H{:=}\,\{h{:}\,H \mid e{\in}\{0,1\}\};\ v{:=}\,e$
	and the *rhs* simplifies to $v{:\in}\,\{0,1\}$	
4.1b	$v{:=}\,0\ \sqcap\ v{:=}\,1$	$e{:=}\,0;\ H{:=}\,\{h{:}\,H \mid e{=}0\};\ v{:=}\,e$ $\sqcap\ e{:=}\,1;\ H{:=}\,\{h{:}\,H \mid e{=}1\};\ v{:=}\,e$
	simplifies to $v{:\in}\,\{0,1\}$ again	
4.2	$h{:\in}\,\{0,1\}$	$h{:\in}\,\{0,1\};\ H{:=}\,\cup\{\{0,1\} \mid h{:}\,H\}$
	simplifies to $h{:\in}\,\{0,1\};\ H{:=}\,\{0,1\}$	
4.3	$h{:=}\,0\ \sqcap\ h{:=}\,1$	$h{:=}\,0;\ H{:=}\,\{0 \mid h{:}\,H\}$ $\sqcap\ h{:=}\,1;\ H{:=}\,\{1 \mid h{:}\,H\}$
	simplifies to $h{:\in}\,\{0,1\};\ H{:=}\,\{h\}$	
4.4	$h{:\in}\,\{0,1\};$ $v{:=}\,0\ \sqcap\ v{:=}\,1$	$h{:\in}\,\{0,1\};\ H{:=}\,\{0,1\};$ $v{:\in}\,\{0,1\}$
	simplifies to $h{:\in}\,\{0,1\};\ v{:\in}\,\{0,1\};\ H{:=}\,\{0,1\}$	
4.5	$h{:\in}\,\{0,1\};$ $v{:\in}\,\{h,1{-}h\}$	$h{:\in}\,\{0,1\};\ H{:=}\,\{0,1\};$ $v{:\in}\,\{h,1{-}h\};\ H{:=}\,\{h{:}\,H \mid v{\in}\{h,1{-}h\}\}$
	which is the same as 4.4	
4.6	$h{:\in}\,\{0,1\};$ $v{:=}\,h\ \sqcap\ v{:=}\,1{-}h$	$h{:\in}\,\{0,1\};\ H{:=}\,\{0,1\};$ $v,H{:=}\,h,\{h\}\ \sqcap\ v,H{:=}\,1{-}h,\{h\}$
	simplifies to $h{:\in}\,\{0,1\};\ v{:\in}\,\{0,1\};\ H{:=}\,\{h\}$	
4.7	$h{:\in}\,\{0,1,2,3\};$ $v{:=}\,h$	$h{:\in}\,\{0,1,2,3\};\ H{:=}\,\{0,1,2,3\};$ $v,H{:=}\,h,\{h\}$
	simplifies to $h{:\in}\,\{0,1,2,3\};\ v{:=}\,h;\ H{:=}\,\{h\}$	
4.8	$h{:\in}\,\{0,1,2,3\};$ $v{:=}\,h\,\mathsf{mod}\,2$	$h{:\in}\,\{0,1,2,3\};\ H{:=}\,\{0,1,2,3\};$ $v{:=}\,h\,\mathsf{mod}\,2;\ H{:=}\,\{h{:}\,H \mid v = h\,\mathsf{mod}\,2\}$
	simplifies to $(H{:=}\,\{0,2\} \sqcap H{:=}\,\{1,3\});\ h{:\in}\,H;\ v{:=}\,h\,\mathsf{mod}\,2$	
4.9	$h{:\in}\,\{0,1,2,3\};$ $v{:=}\,h\,\mathsf{mod}\,2;$ $v{:=}\,0$	$h{:\in}\,\{0,1,2,3\};\ H{:=}\,\{0,1,2,3\};$ $v{:=}\,h\,\mathsf{mod}\,2;\ H{:=}\,\{h{:}\,H \mid v = h\,\mathsf{mod}\,2\};$ $v{:=}\,0$
	simplifies to $(H{:=}\,\{0,2\} \sqcap H{:=}\,\{1,3\});\ h{:\in}\,H;\ v{:=}\,0$	

The simplifications are made using classical semantics over v,h,H.

Fig. 4. Operational-semantics examples.

$$v = \mathsf{last}.\overline{v}\ \wedge\ h = \mathsf{last}.\overline{h}\ \wedge\ H = \{\overline{h}' \mid (\overline{v},\overline{h}',\overline{p}) \sim (\overline{v},\overline{h},\overline{p}) \cdot \mathsf{last}.\overline{h}'\}.^{4}$$

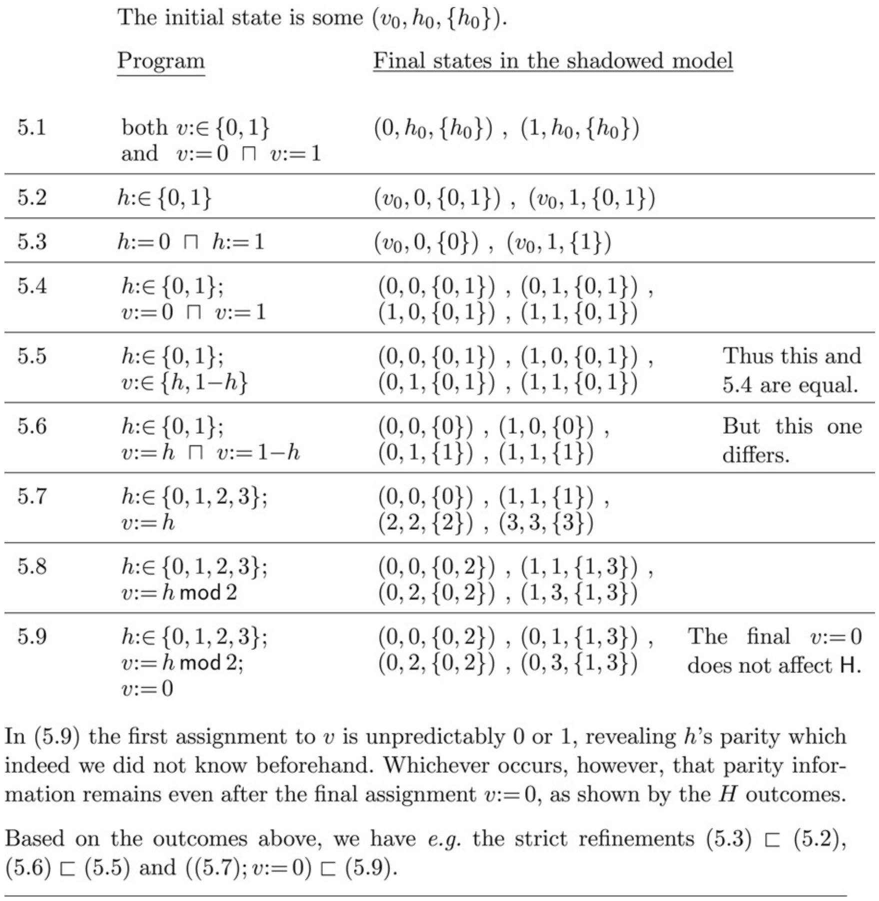
From sequences $\overline{v}, \overline{h}, \overline{p}$ we retain only final values v, h and the induced H.[5]

5. *The Shadow Knows:* An operational semantics

We now use our model to give an ignorance-sensitive operational interpretation of a simple sequential programming language including nondeterminism. To begin with, we continue to assume a state space with just two variables, the visible v and the hidden h. (In general of course there can be many visibles and hiddens.) Our semantics adds a third variable H called the *shadow* of the hidden variable h. The semantics will ensure that, in the sense of Section 4.2 above, the shadow H "knows" the set of values that h has *potentially*.

[4] Read the last as "vary $\overline{h}'$ such that $(\overline{v},\overline{h}',\overline{p}) \sim (\overline{v},\overline{h},\overline{p})$ and take $\mathsf{last}.\overline{h}'$ for each".

[5] In fact the H-component makes h redundant – *i.e.* we can make do with just (v,H) – but this extra "compression" would complicate the presentation subsequently.



The initial state is some $(v_0, h_0, \{h_0\})$.

	Program	Final states in the shadowed model	
5.1	both $v{:\in}\{0,1\}$ and $v{:=}0 \sqcap v{:=}1$	$(0,h_0,\{h_0\})$, $(1,h_0,\{h_0\})$	
5.2	$h{:\in}\{0,1\}$	$(v_0,0,\{0,1\})$, $(v_0,1,\{0,1\})$	
5.3	$h{:=}0 \sqcap h{:=}1$	$(v_0,0,\{0\})$, $(v_0,1,\{1\})$	
5.4	$h{:\in}\{0,1\};$ $v{:=}0 \sqcap v{:=}1$	$(0,0,\{0,1\})$, $(0,1,\{0,1\})$, $(1,0,\{0,1\})$, $(1,1,\{0,1\})$	
5.5	$h{:\in}\{0,1\};$ $v{:\in}\{h,1{-}h\}$	$(0,0,\{0,1\})$, $(1,0,\{0,1\})$, $(0,1,\{0,1\})$, $(1,1,\{0,1\})$	Thus this and 5.4 are equal.
5.6	$h{:\in}\{0,1\};$ $v{:=}h \sqcap v{:=}1{-}h$	$(0,0,\{0\})$, $(1,0,\{0\})$, $(0,1,\{1\})$, $(1,1,\{1\})$	But this one differs.
5.7	$h{:\in}\{0,1,2,3\};$ $v{:=}h$	$(0,0,\{0\})$, $(1,1,\{1\})$, $(2,2,\{2\})$, $(3,3,\{3\})$	
5.8	$h{:\in}\{0,1,2,3\};$ $v{:=}h \bmod 2$	$(0,0,\{0,2\})$, $(1,1,\{1,3\})$, $(0,2,\{0,2\})$, $(1,3,\{1,3\})$	
5.9	$h{:\in}\{0,1,2,3\};$ $v{:=}h \bmod 2;$ $v{:=}0$	$(0,0,\{0,2\})$, $(0,1,\{1,3\})$, $(0,2,\{0,2\})$, $(0,3,\{1,3\})$	The final $v{:=}0$ does not affect H.

In (5.9) the first assignment to v is unpredictably 0 or 1, revealing h's parity which indeed we did not know beforehand. Whichever occurs, however, that parity information remains even after the final assignment $v{:=}0$, as shown by the H outcomes.

Based on the outcomes above, we have *e.g.* the strict refinements $(5.3) \sqsubset (5.2)$, $(5.6) \sqsubset (5.5)$ and $((5.7); v{:=}0) \sqsubset (5.9)$.

Fig. 5. Operational examples of outcomes and refinement.

5.1. Syntax and semantics

The syntax of our example programming language is given in Fig. 1.

The operational semantics is given in Fig. 2, for which we provide the following commentary. In summary, we convert "ignorance-sensitive" (that is v, h-) programs to "ordinary" (that is v, h, H-) programs and then rely on the conventional relational semantics for those.[6] We comment on each case in turn.

- The *identity* **skip** changes no variables, hence has no effect.
- *Assigning to a 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. *Choosing a visible* is a generalisation of that.
- *Assigning to a hidden* sets the shadow to all values that could have resulted from the current shadow; again, *choosing a hidden* is a generalisation.
- *Demonic choice* and *sequential composition* retain their usual definitions. Note in particular that the former induces nondeterminism in the shadow 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.

5.2. Examples of informal- and operational semantics

In Fig. 3 we give informal descriptions of the effects of a number of small program fragments; then in Fig. 4 we apply the semantics above to give the actual translations, showing how they support the informal descriptions.

We begin by noting that $h{:\in}\{0, 1\}$, the simplest example (Fig. 3(.2)) of ignorance, leads us as usual to either of two states, one with $h{=}0$ and the other with $h{=}1$; but since the choice is atomic an adversary cannot tell which of those two states

[6] Our definitions are induced from the abstraction given in Section 4.2.

it is. This is reflected in the induced assignment of {0, 1} to the shadow H, on both branches, shown in the corresponding semantics (4.2).

In contrast, although the program $h{:=}\,0 \;\sqcap\; h{:=}\,1$ (3.3) again leads unpredictably to $h{=}0$ or $h{=}1$, in both cases the semantics (4.3) shows that we have $H{=}\{h\}$ finally, reflecting that the non-atomic program flow has revealed h's value implicitly to the adversary. Thus an operational indication of (degrees of) ignorance is the size of H: the bigger it is, the less is known; and that is what distinguishes these two examples.

5.3. *Operational definition of ignorance-preserving refinement*

Given two states (v_1, h_1, H_1) and (v_2, h_2, H_2) we say that the first is refined by the second just when they agree on their v, h-components and ignorance is only increased in the H-component: that is we define

$$(v_1, h_1, H_1) \sqsubseteq (v_2, h_2, H_2) \quad \widehat{=} \quad v_1{=}v_2 \wedge h_1{=}h_2 \wedge H_1{\subseteq}H_2 .$$

We promote this to *sets* of (v, h, H)-states in the standard (Smyth power-domain [15]) style, saying that we have refinement between two sets S_1, S_2 of states just when every state $s_2{\in}S_2$ is a refinement of some state $s_1{\in}S_1$; that is, we define

$$S_1 \sqsubseteq S_2 \quad \widehat{=} \quad (\forall s_2{:}\, S_2 \cdot (\exists s_1{:}\, S_1 \cdot s_1 \sqsubseteq s_2)).$$

We promote this a second time, now to (v, h, H)-programs, using the standard pointwise-lifting for functions in which S1 $\sqsubseteq$ S2 just when for some initial s the set of possible outcomes S_1 arising via S1 from that s is refined by the set of possible outcomes S_2 arising via S2 from that same s. Examples are given in Fig. 5.

Finally, we say that two (v, h)-programs are related by refinement just when their (v, h, H)-meanings are: that is S1 $\sqsubseteq$ S2 $\widehat{=}$ [[S1]] $\sqsubseteq$ [[S2]].

In summary, we have ignorance-sensitive refinement S1 $\sqsubseteq$ S2 just when for each initial (v, h, H) every possible outcome (v_2, h_2, H_2) of [[S2]] satisfies $v_1{=}v_2 \wedge h_1{=}h_2 \wedge H_1{\subseteq}H_2$ for some outcome (v_1, h_1, H_1) of [[S1]].

5.4. *Declarations and local variables*

In the absence of procedure calls and recursion, we can treat multiple-variable programs over $v_1, \ldots, v_m; h_1, \ldots h_n$, say, as operating over single tuple-valued variables $v{=}(v_1, \ldots, v_m)$; $h{=}(h_1, \ldots h_n)$. A variable v_i or h_i in an expression is actually a projection from the corresponding tuple; as the target of an assignment it induces a component-wise update of the tuple.

Within blocks, visible- and hidden local variables have separate declarations VIS v and HID h respectively. 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 |[·]| (for brevity) or equivalently **begin** · **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 operational semantics for visibles this treatment is the usual one: thus we have [[**begin** VIS v · S **end**]] $\widehat{=}$ **begin var** v · [[S]] **end**. For hidden variables however we arrange for H to be implicitly extended by the declaration; thus we have

$$[\![\, \textbf{begin}\ \textsc{hid}\ h \cdot \mathsf{S}\ \textbf{end}\,]\!] \quad \widehat{=} \quad \textbf{begin var}\ h \cdot \begin{array}{l} H{:=}\,H{\times}\mathsf{D}; \\ [\![\mathsf{S}]\!]; \\ H{:=}\,H{\downarrow} \qquad \textbf{end} \end{array}$$

where D is the set over which our variables – the new h in this case – take their values (Fig. 2), and we invent the notation $H{\downarrow}$ for the projection of the product $H{\times}$D back to H.

6. Modal assertion logic for the shadow model

We now introduce an assertion logic for reasoning over the model of the previous section. Our language will be first-order predicate formulae Φ, interpreted conventionally over the variables of the program, but augmented with a "knows" modal operator [14, 3.7.2] so that KΦ holds in *this* state just when Φ itself holds in *all* (other) states the adversary considers compatible with this one. From our earlier discussion (Section 3) we understand the adversary's notion of compatibility to be, for a given program text, "having followed the same path through that text and having generated the same visible-variable values along the way"; from our abstraction (Section 4.2), we know we can determine that compatibility based on H alone.

The dual modality "possibly" is written PΦ and defined $\neg$K$(\neg\Phi)$; and it is the modality we will use in practice, as it expresses ignorance directly. (Because KΦ seems more easily grasped, however, we explain both.)

6.1. Interpretation of modal formulae

We give the language function- (including constant-) and relation symbols as needed, among which we distinguish the (program-variable) symbols *visibles* in some set V and *hiddens* in H; as well there are the usual (logical) variables in L over which we allow ∀, ∃ quantification. The visibles, hiddens and variables are collectively the *scalars* $\mathrm{X} \mathrel{\widehat{=}} \mathrm{V} \cup \mathrm{H} \cup \mathrm{L}$.

A *structure* comprises a non-empty domain D of values, together with functions and relations over it that interpret the function- and relation symbols mentioned above; within the structure we name the partial functions v, h that interpret visibles and hiddens respectively; we write their types $\mathrm{V} \nrightarrow \mathrm{D}$ and $\mathrm{H} \nrightarrow \mathrm{D}$ (where the "crossbar" indicates the potential partiality of the function).

A *valuation* is a partial function from scalars to D, thus typed $\mathrm{X} \nrightarrow \mathrm{D}$; one valuation w_1 can override another w so that for scalar x we have $(\mathsf{w} \lhd \mathsf{w}_1).\mathsf{x}$ is $\mathsf{w}_1.\mathsf{x}$ if w_1 is defined at x and is w.x otherwise. The valuation $\langle \mathsf{x} \mapsto \mathsf{d} \rangle$ is defined only at x, where it takes value d.

A *state* (v, h, H) comprises a visible- v, hidden- h and *shadow*- part H; the last, in $\mathbb{P}(\mathrm{H} \nrightarrow \mathrm{D})$, is a *set* of valuations over hiddens only.[7] We require that $\mathsf{h} \in \mathsf{H}$.

We define truth of Φ at (v, h, H) under valuation w by induction in the usual style, writing $(\mathsf{v}, \mathsf{h}, \mathsf{H}), \mathsf{w} \models \Phi$. Let t be the term-valuation built inductively from the valuation $\mathsf{v} \lhd \mathsf{h} \lhd \mathsf{w}$. Then we have the following [14, pp. 79,81]:

- $(\mathsf{v}, \mathsf{h}, \mathsf{H}), \mathsf{w} \models \mathrm{R}.\mathrm{T}_1.\cdots.\mathrm{T}_k$ for relation symbol R and terms $\mathrm{T}_1 \cdots \mathrm{T}_k$ iff the tuple $(\mathsf{t}.\mathrm{T}_1, \ldots, \mathsf{t}.\mathrm{T}_k)$ is an element of the interpretation of R.
- $(\mathsf{v}, \mathsf{h}, \mathsf{H}), \mathsf{w} \models \mathrm{T}_1 = \mathrm{T}_2$ iff $\mathsf{t}.\mathrm{T}_1 = \mathsf{t}.\mathrm{T}_2$.
- $(\mathsf{v}, \mathsf{h}, \mathsf{H}), \mathsf{w} \models \neg\Phi$ iff $(\mathsf{v}, \mathsf{h}, \mathsf{H}), \mathsf{w} \not\models \Phi$.
- $(\mathsf{v}, \mathsf{h}, \mathsf{H}), \mathsf{w} \models \Phi_1 \wedge \Phi_2$ iff $(\mathsf{v}, \mathsf{h}, \mathsf{H}), \mathsf{w} \models \Phi_1$ and $(\mathsf{v}, \mathsf{h}, \mathsf{H}), \mathsf{w} \models \Phi_2$.
- $(\mathsf{v}, \mathsf{h}, \mathsf{H}), \mathsf{w} \models (\forall \mathrm{L} \cdot \Phi)$ iff $(\mathsf{v}, \mathsf{h}, \mathsf{H}), \mathsf{w} \lhd \langle \mathrm{L} \mapsto \mathsf{d} \rangle \models \Phi$ for all d in D.
- $(\mathsf{v}, \mathsf{h}, \mathsf{H}), \mathsf{w} \models \mathrm{K}\Phi$ iff $(\mathsf{v}, \mathsf{h}_1, \mathsf{H}), \mathsf{w} \models \Phi$ for all h_1 in H.

We write just $(\mathsf{v}, \mathsf{h}, \mathsf{H}) \models \Phi$ when w is empty, and $\models \Phi$ when $(\mathsf{v}, \mathsf{h}, \mathsf{H}) \models \Phi$ for all v, h, H with $\mathsf{h} \in \mathsf{H}$, and we take advantage of the usual "syntactic sugar" for other operators (including P as $\neg\mathrm{K}\neg$). Thus for example we can show $\models \Phi \Rightarrow \mathrm{P}\Phi$ for all Φ, a fact which we use in Section 8.5. Similarly we can assume *wlog* that modalities are not nested, since we can remove nestings via the validity $\models \mathrm{P}\Phi \equiv (\exists c \cdot [h \backslash c]\Phi \wedge \mathrm{P}(h{=}c))$.

7. A program logic of Hoare-triples

7.1. Pre-conditions and postconditions

We say that $\{\Phi\}$ S $\{\Psi\}$ just when any initial state $(\mathsf{v}, \mathsf{h}, \mathsf{H}) \models \Phi$ must lead via S only to final states $(\mathsf{v}', \mathsf{h}', \mathsf{H}') \models \Psi$; typically Φ is called the *precondition* and Ψ is called the *postcondition*. Fig. 6 illustrates this proposed program logic using our earlier examples from Fig. 3. (Because the example postconditions Ψ do not refer to initial values, their validities in this example are independent of the precondition Φ.)

7.2. Logical definition of ignorance-preserving refinement

We saw in Section 5.3 a definition of refinement given in terms of the model directly. When the model is linked to a programming logic we expect the earlier definition of refinement to be *induced* by the logic, so that S1 $\sqsubseteq$ S2 just when all logically expressible properties of S1 are satisfied S2 also (Section 1). The key is of course "expressible".

Our expressible properties will be traditional Hoare-style triples, *but over formulae whose truth is preserved by increase of ignorance*: those in which all modalities K occur negatively, and all modalities P occur positively. We say that such occurrences of modalities are *ignorant*; and a formula is ignorant just when all its modalities are. Thus in program-logical terms we say that S1 $\sqsubseteq$ S2 just when for all *ignorant* formulae Φ, Ψ we have

$$\begin{array}{r} \text{that the property } \{\Phi\}\ [\![\mathrm{S1}]\!]\ \{\Psi\} \text{ is valid} \\ \text{implies that the property } \{\Phi\}\ [\![\mathrm{S2}]\!]\ \{\Psi\} \text{ is also valid.} \end{array} \tag{4}$$

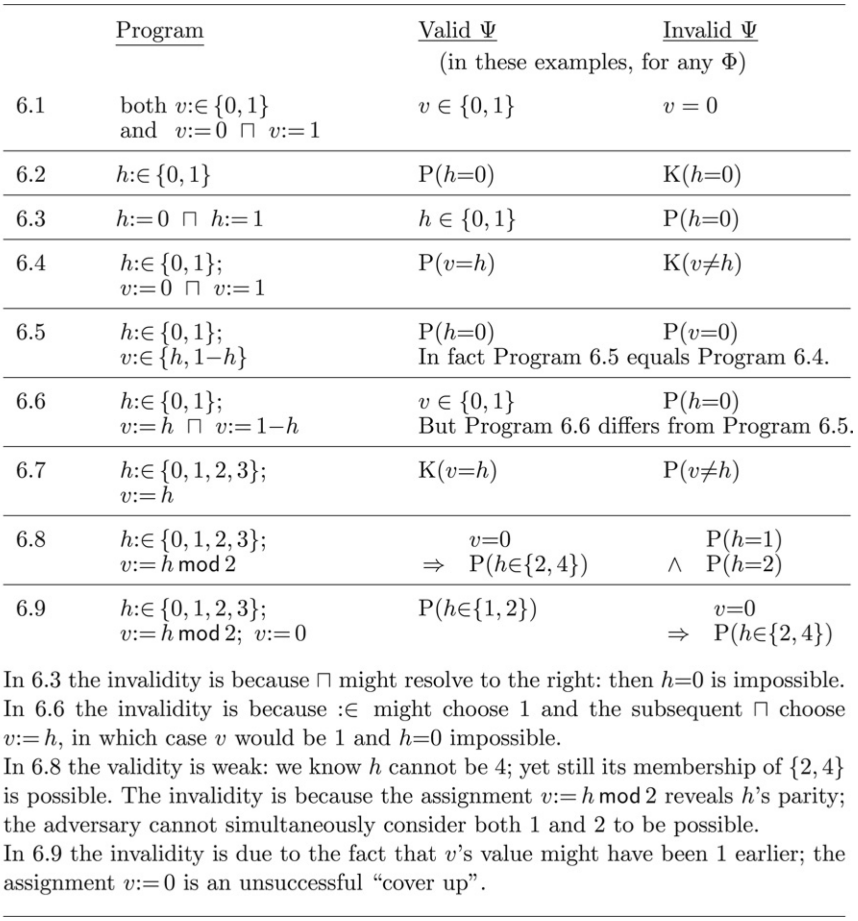
We link these two definitions of refinement in the following lemma.

Lemma 1. *The definitions of ignorance-sensitive refinement given in Section 5.3 and in (4) above are compatible.*

Proof. Fix S1 and S2 and assume for simplicity single variables v, h so that we can work with states (v, h, H) rather than valuations. (Working more generally with valuations, we would define $\mathsf{H}_1 \subseteq \mathsf{H}_2$ to mean $\mathsf{H}_1.\mathsf{x} \subseteq \mathsf{H}_2.\mathsf{x}$ for all hiddens x where both valuations are defined; the proof's extension to this case is conceptually straightforward but notationally cumbersome.) We argue the contrapositive, for a contradiction, in both directions.[8]

[7] To allow for declarations of additional hidden variables, we must make H a set of *valuations* rather than simply a set of values. This is isomorphic to a set of tuples (Section 5.4), but is easier to use in the definition of the logic.

[8] That is we establish $A \Rightarrow B$ via $\neg B \wedge A \Rightarrow \text{FALSE}$.



	Program	Valid Ψ	Invalid Ψ
		(in these examples, for any Φ)	
6.1	both $v{:\in}\{0,1\}$ and $v{:=}0 \sqcap v{:=}1$	$v \in \{0,1\}$	$v = 0$
6.2	$h{:\in}\{0,1\}$	$\mathrm{P}(h{=}0)$	$\mathrm{K}(h{=}0)$
6.3	$h{:=}0 \sqcap h{:=}1$	$h \in \{0,1\}$	$\mathrm{P}(h{=}0)$
6.4	$h{:\in}\{0,1\};$ $v{:=}0 \sqcap v{:=}1$	$\mathrm{P}(v{=}h)$	$\mathrm{K}(v{\neq}h)$
6.5	$h{:\in}\{0,1\};$ $v{:\in}\{h, 1{-}h\}$	$\mathrm{P}(h{=}0)$ In fact Program 6.5 equals Program 6.4.	$\mathrm{P}(v{=}0)$
6.6	$h{:\in}\{0,1\};$ $v{:=}h \sqcap v{:=}1{-}h$	$v \in \{0,1\}$ But Program 6.6 differs from Program 6.5.	$\mathrm{P}(h{=}0)$
6.7	$h{:\in}\{0,1,2,3\};$ $v{:=}h$	$\mathrm{K}(v{=}h)$	$\mathrm{P}(v{\neq}h)$
6.8	$h{:\in}\{0,1,2,3\};$ $v{:=}h \bmod 2$	$v{=}0$ $\Rightarrow\ \mathrm{P}(h{\in}\{2,4\})$	$\mathrm{P}(h{=}1)$ $\wedge\ \mathrm{P}(h{=}2)$
6.9	$h{:\in}\{0,1,2,3\};$ $v{:=}h \bmod 2;\ v{:=}0$	$\mathrm{P}(h{\in}\{1,2\})$	$v{=}0$ $\Rightarrow\ \mathrm{P}(h{\in}\{2,4\})$

- In 6.3 the invalidity is because $\sqcap$ might resolve to the right: then $h{=}0$ is impossible.
- In 6.6 the invalidity is because $:\in$ might choose 1 and the subsequent $\sqcap$ choose $v{:=}h$, in which case v would be 1 and $h{=}0$ impossible.
- In 6.8 the validity is weak: we know h cannot be 4; yet still its membership of $\{2,4\}$ is possible. The invalidity is because the assignment $v{:=}h \bmod 2$ reveals h's parity; the adversary cannot simultaneously consider both 1 and 2 to be possible.
- In 6.9 the invalidity is due to the fact that v's value might have been 1 earlier; the assignment $v{:=}0$ is an unsuccessful "cover up".

Fig. 6. Examples of valid and invalid postconditions.

operational- (Section 5.3) implies logical refinement (Section 7.2)

Suppose we have ignorant modal formulae Φ, Ψ so that $\{\Phi\}\ [\![S1]\!]\ \{\Psi\}$ is valid but $\{\Phi\}\ [\![S2]\!]\ \{\Psi\}$ is invalid. There must therefore be initial $(\mathsf{v}_0, \mathsf{h}_0, \mathsf{H}_0) \models \Phi$ such that final $(\mathsf{v}_2, \mathsf{h}_2, \mathsf{H}_2)$ is possible via $[\![S2]\!]$ yet $(\mathsf{v}_2, \mathsf{h}_2, \mathsf{H}_2) \not\models \Psi$.

Because (we assume for a contradiction) S2 refines S1 operationally, we must have some $(\mathsf{v}_1, \mathsf{h}_1, \mathsf{H}_1)$ via $[\![S1]\!]$ from $(\mathsf{v}_0, \mathsf{h}_0, \mathsf{H}_0)$ with $\mathsf{H}_1{\subseteq}\mathsf{H}_2$ and of course as well $\mathsf{v}_1{=}\mathsf{v}_2$, and similarly $\mathsf{h}_1{=}\mathsf{h}_2$; thus from $(\mathsf{v}_0, \mathsf{h}_0, \mathsf{H}_0) \models \Phi$ and $\{\Phi\}\ [\![S1]\!]\ \{\Psi\}$ we conclude $(\mathsf{v}_2, \mathsf{h}_2, \mathsf{H}_1) \models \Psi$.

But $(\mathsf{v}_2, \mathsf{h}_2, \mathsf{H}_1) \models \Psi$ and $\mathsf{H}_1{\subseteq}\mathsf{H}_2$ implies $(\mathsf{v}_2, \mathsf{h}_2, \mathsf{H}_2) \models \Psi$ because Ψ is ignorant — which gives us our contradiction.

logical- (Section 7.2) implies operational refinement (Section 5.3)

Suppose we have $(\mathsf{v}_2, \mathsf{h}_2, \mathsf{H}_2)$ via $[\![S2]\!]$ from some $(\mathsf{v}_0, \mathsf{h}_0, \mathsf{H}_0)$ but there is no $(\mathsf{v}_2, \mathsf{h}_2, \mathsf{H}_1)$ with $\mathsf{H}_1{\subseteq}\mathsf{H}_2$ via $[\![S1]\!]$ — equivalently, for all $[\![S1]\!]$-outcomes $(\mathsf{v}_2, \mathsf{h}_2, \mathsf{H}_1)$ we have $(\exists h{:}\ \mathsf{H}_1 \mid h \notin \mathsf{H}_2)$.

For convenience let symbols v_0 *etc.* from the state-triples act as constants denoting their values. Define formulae

$$\begin{array}{llll} & \Phi & \widehat{=} & v{=}\mathsf{v}_0 \wedge h{=}\mathsf{h}_0 \wedge (\forall x{:}\ \mathsf{H}_0 \cdot \mathrm{P}(h{=}x)) \\ \text{and} & \Psi & \widehat{=} & v{=}\mathsf{v}_2 \wedge h{=}\mathsf{h}_2 \Rightarrow \mathrm{P}(h{\notin}\mathsf{H}_2). \end{array}$$

Now $\{\Phi\}\ [\![S1]\!]\ \{\Psi\}$ because any initial state with $(\mathsf{v}, \mathsf{h}, \mathsf{H}) \models \Phi$ must have $\mathsf{v}{=}\mathsf{v}_0 {\wedge} \mathsf{h}{=}\mathsf{h}_0 {\wedge} \mathsf{H}{\supseteq}\mathsf{H}_0$, and we observe that expanding the H-component of an initial state can only expand the H component of outcomes. (This is a form of ignorance-monotonicity, established by inspection of Fig. 2.) That is, because all outcomes from $(\mathsf{v}_0, \mathsf{h}_0, \mathsf{H}_0)$ itself via $[\![S1]\!]$ satisfy Ψ, so must all outcomes from any (other) initial such $(\mathsf{v}_0, \mathsf{h}_0, \mathsf{H}) \models \Phi$ satisfy Ψ, since Ψ is ignorant.

But (since we assume logical refinement, for a contradiction) we have $\{\Phi\}\ [\![S1]\!]\ \{\Psi\}$ implies $\{\Phi\}\ [\![S2]\!]\ \{\Psi\}$ — whence also $(\mathsf{v}_2, \mathsf{h}_2, \mathsf{H}_2) \models \Psi$, which is our contradiction (by inspection of Ψ). □

7.3. Two negative examples: Excluded refinements

We now give two important examples of non-refinements: the first appears valid *wrt* ignorance, but is excluded classically; the second is the opposite, valid classically, but excluded *wrt* ignorance.

7.3.1. Postconditions refer to h even though the adversary cannot see it

We start with the observation that we do *not* have $h{:\in}\,\{0,1\}\ ?\!\sqsubseteq\ h{:\in}\,\{0,1,2\}$, even though the ignorance increases and h cannot be observed by the adversary: operationally the refinement fails due to the outcome $(\cdot, 2, \{0,1,2\})$ on the right for which there is no supporting triple $(\cdot, 2, ?)$ on the left; logically it fails because all left-hand outcomes satisfy $h{\neq}2$ but some right-hand outcomes do not.

The example illustrates the importance of direct references to h in our postconditions, *even though h cannot be observed by the adversary*: for otherwise the above refinement *would* after all go through logically. The reason it must not is that, if it did, we would by monotonicity of refinement (*RP0*) have to admit $h{:\in}\,\{0,1\};\ v{:=}\,h\ ?\!\sqsubseteq\ h{:\in}\,\{0,1,2\};\ v{:=}\,h$ as well, clearly false: the outcome $v{=}2$ is not possible on the left, excluding this refinement classically.

7.3.2. The Refinement Paradox

We recall that the Refinement Paradox [9] is an issue because classical refinement allows the "secure" $v{:\in}\,T$ to be refined to the "insecure" $v{:=}\,h$ as an instance of reduction of demonic nondeterminism. We can now resolve it: it is excluded on ignorance grounds.

Operationally, we observe that $v{:\in}\,T\ \not\sqsubseteq\ v{:=}\,h$ because from $(?, h_0, H_0)$ the former's final states are $\{(e, h_0, H_0) \mid e{:}\,T\}$; yet the latter's are just $\{\ (h_0, h_0, \{h_0\})\ \}$ and, even supposing $h_0{\in}T$ so that $e{=}h_0$ is a possibility, still in general $H_0{\not\subseteq}\{h_0\}$ as refinement would require.

Logically, we can prove (Fig. 9 below) that $\{\mathrm{P}(h{=}C)\}\ v{:\in}\,T\ \{\mathrm{P}(h{=}C)\}$ holds while $\{\mathrm{P}(h{=}C)\}\ v{:=}\,h\ \{\mathrm{P}(h{=}C)\}$ does not.

8. Weakest-precondition calculus for (v, h)-semantics directly

Our procedure so far for reasoning about ignorance has been to translate (v, h)-programs into (v, h, H)-programs via the operational semantics of Fig. 2, and then to reason there about refinement either operationally (Section 5.3) or logically (Section 7.2(4)); in view of Lemma 1 we can do either.

We now streamline this process by developing a weakest-precondition-style program logic on the (v, h)-level program text directly, consistent with the above but avoiding the need for translation to (v, h, H)-semantics. At (6) we give the induced (re-)formulation of refinement.

8.1. Weakest preconditions: A review

Given a postcondition Ψ and a program S there are likely to be many candidate preconditions Φ that will satisfy the Hoare-triple $\{\Phi\}$ S $\{\Psi\}$; because it is a property of Hoare-triples that if Φ_1, Φ_2 are two such (for a given S, Ψ), then so is $\Phi_1 \vee \Phi_2$ (and this extends even to infinitely many), in fact there is a so-called *weakest* precondition which is the disjunction of them all: it is written $wp.\mathrm{S}.\Psi$ and, by definition, it has the property

$$\{\Phi\}\ \mathrm{S}\ \{\Psi\} \quad \text{iff} \quad \models \Phi \Rightarrow wp.\mathrm{S}.\Psi. \tag{5}$$

Because the partially evaluated terms wp.S can be seen as functions from postconditions Ψ to preconditions Φ, those terms are sometimes called *predicate transformers*; and a systematic syntax-inductive definition of wp.S for every program S in some language is called *predicate-transformer semantics*. We give it for our language in Fig. 8; it is consistent with the operational semantics of Fig. 2.

Our predicate-transformer semantics for ignorance-sensitive programs is thus derived from the operational semantics and the interpretation in Section 6 of modal formulae. With it comes a wp-style definition of refinement

$$\mathrm{S1} \sqsubseteq \mathrm{S2} \quad \text{iff} \quad \models wp.\mathrm{S1}.\Psi \Rightarrow wp.\mathrm{S2}.\Psi \quad \text{for all ignorant } \Psi \tag{6}$$

which, directly from (5), is consistent with our other two definitions.

8.2. Predicate-transformer semantics for ignorance: Approach

To derive a wp-semantics for the original (v, h)-space, we translate our modal (v, h)-formulae into equivalent classical formulae over (v, h, H), calculate the classical weakest preconditions for *wrt* the corresponding (v, h, H)-programs, and finally translate those back into modal formulae.

From Section 6 we can see that the modality $\mathrm{P}\Phi$ corresponds to the classical formula $(\exists(h_1, \ldots){:}\,H \cdot \Phi)$, where hidden variables $h_1, \ldots$ are all those in scope; for a general formula Ψ we will write $[\![\Psi]\!]$ for the result of applying that translation to all modalities it contains; we continue to assume that modalities are not nested. We write wp for our new transformer semantics (over v, h), using $\mathbf{wp}$ in this section for the classical transformer-semantics (over v, h, H).

As an example of this procedure (recalling Fig. 6.8) we have the following:

$$\begin{array}{rcl} [\![\ v{:=}\,h \bmod 2\ ]\!] & = & v{:=}\,h \bmod 2; \\ & & H{:=}\,\{h{:}\,H \mid v = h \bmod 2\} \\ \text{and}\quad [\![\ v{=}0 \Rightarrow \mathrm{P}(h{\in}\{2,4\})\ ]\!] & = & \quad v{=}0 \\ & & \Rightarrow\ (\exists h{:}\,H \mid h{\in}\{2,4\}). \end{array}$$

Then the classical **wp**-semantics [5] is used over the explicit (v, h, H)-program fragments as follows in this example:

$$v{=}0 \Rightarrow (\exists h{:}\,H \mid h{\in}\{2,4\})$$

through $\mathbf{wp}.(H{:=}\,\{h{:}\,H \mid v = h \bmod 2\})$ gives

$$\begin{array}{lll} & v{=}0 \Rightarrow (\exists h{:}\,\{h{:}\,H \mid v = h \bmod 2\} \mid h{\in}\{2,4\}) & \\ = & v{=}0 \Rightarrow (\exists h{:}\,\{h'{:}\,H \mid v = h' \bmod 2\} \mid h{\in}\{2,4\}) & \text{“rename bound variable”} \\ = & v{=}0 \Rightarrow (\exists h{:}\,H \mid v = h \bmod 2 \wedge h{\in}\{2,4\}) & \text{“flatten comprehensions”} \\ = & v{=}0 \Rightarrow (\exists h{:}\,H \mid v = 0 \wedge h{\in}\{2,4\}) & \text{“arithmetic”} \\ = & v{=}0 \Rightarrow (\exists h{:}\,H \mid h{\in}\{2,4\}) & \text{“predicate calculus”} \end{array}$$

through $\mathbf{wp}.(v{:=}\,h \bmod 2)$ gives

$$h \bmod 2 = 0 \ \Rightarrow\ (\exists h{:}\,H \mid h{\in}\{2,4\})$$

which, translated back into the modal P-form (*i.e.* “un-$[\![\cdot]\!]$’d”) gives the weakest precondition $h \bmod 2 = 0 \Rightarrow \mathrm{P}(h{\in}\{2,4\})$.

Our general tool for the transformers will be the the familiar substitution $[e\backslash E]$ which replaces all occurrences of variable e by the term E, introducing α-conversion as necessary to avoid capture by quantifiers. There are some special points to note, however, if we are applying the substitution over a modal formula Φ and wish to be consistent with the way the substitution would actually be carried out over $[\![\Phi]\!]$.

(1) Substitution into a modality for a hidden variable has no effect; the substitution is simply discarded. Thus $[h\backslash E]\mathrm{P}\Phi$ is just $\mathrm{P}\Phi$.
(2) If x does not occur in Φ then again the substitution is just discarded. Thus $[x\backslash E]\mathrm{P}\Phi$ is just $\mathrm{P}\Phi$ if x does not occur in Φ.
(3) If x, E contain no hiddens, then $[x\backslash E]\mathrm{P}\Phi$ is just $\mathrm{P}([x\backslash E]\Phi)$.
(4) Otherwise the substitution “stalls” at the modality (there is no reduction).

These effects are due to the fact that, although the modality P is implicitly a quantification over hidden variables, we have not listed variables after the “quantifier” as one normally would: so we cannot α-convert them if that is what is necessary to allow the substitution to go through. (This accounts for the stalling in Case 4 and is the price we pay for suppressing the clutter of H and its quantification.) Usually the body Φ of the modality can be manipulated until one of Cases 1–3 holds, and then the substitution goes through after all.

8.3. *Example calculations of atomic commands’ semantics*

We now calculate the *wp*-semantics for *Identity*, *Assign to visible* and *Choose hidden*. Occurrences of v, h in the rules may be vectors of visible- or vectors of hidden variables, in which case substitutions such as $[h\backslash h']$ apply throughout the vector. We continue to assume no nesting of modalities.

8.3.1. *(v, h)-transformer semantics for* **skip**

We must define *wp* in this case so that for all modal formulae Ψ we have

$$[\![\mathit{wp}.\mathbf{skip}.\Psi]\!] \quad = \quad \mathbf{wp}.[\![\mathbf{skip}]\!].[\![\Psi]\!],$$

and that is clearly satisfied by the definition $\mathit{wp}.\mathbf{skip}.\Psi \mathrel{\widehat{=}} \Psi$.

8.3.2. *(v, h)-transformer semantics for assign to visible*

Here we need

$$\begin{array}{rcl} [\![\mathit{wp}.(v{:=}\,E).\Psi]\!] & = & \mathbf{wp}.[\![v{:=}\,E]\!].[\![\Psi]\!] \\ \text{(from Fig. 2)} & = & [e\backslash E][H\backslash\{h{:}\,H \mid e{=}E\}][v\backslash E][\![\Psi]\!], \end{array}$$

where the second equality comes from the operational-semantic definitions of Fig. 2 and classical **wp**. Note that we have *three* substitutions to perform in general.

The middle substitution $[H\backslash\{h{:}\,H \mid e{=}E\}]$ leads to the definition of what we will call a “technical transformer”, *Shrink Shadow*, and it is a feature of our carrying H around without actually referring to it; in our v, h semantics we will write it $[\Downarrow e{=}E]$, where the equality $e{=}E$ expresses the constraint (the “shrinking”) to be applied to the potential values for h as recorded in H.

The general $[\Downarrow \phi]$ is a simple substitution (too); but since it is a substitution for H it affects only the (translated) modal formulae, having no effect on classical atomic formulae (because they do not contain H). Thus for modal formulae (only) we have

$$
\begin{array}{lll}
 & [H\backslash\{h\colon H \mid \phi\}][\![\mathrm{P}\Phi]\!] & \\
= & [H\backslash\{h\colon H \mid \phi\}](\exists h\colon H \cdot \Phi) & \text{“translate”}\\
= & (\exists h\colon \{h'\colon H \mid \phi'\} \cdot \Phi) & \text{“let } \phi' \text{ be } [h\backslash h']\phi \text{ for clarity”}\\
= & (\exists h\colon H \cdot \phi \wedge \Phi) & \text{“simplify”}\\
= & [\![\ \mathrm{P}(\phi \wedge \Phi)\ ]\!] & \text{“retranslate”}\\
= & [\![\ [\Downarrow \phi]\mathrm{P}\Phi\ ]\!]. & \text{“postulated definition } \textit{Shrink Shadow}\text{”}
\end{array}
$$

Note that hidden variables in ϕ are not protected in Shrink Shadow from implicit capture by P.

Collecting this all together gives us

$$
wp.(v{:=}E).\Psi \quad = \quad [e\backslash E][\Downarrow e{=}E][v\backslash E]\Psi,
$$

which will be the definition we give in Fig. 8. We place the synthesised definition of *Shrink Shadow* in Fig. 7.

8.3.3. (v, h)-transformer semantics for Choose hidden

Here we need

$$
\begin{array}{lll}
[\![wp.(h{:\in}\, E).\Psi]\!] & = & \mathbf{wp}.[\![h{:\in}\, E]\!].[\![\Psi]\!]\\
 & = & (\forall h\colon E \cdot [H\backslash\cup\{E \mid h\colon H\}][\![\Psi]\!]),
\end{array}
$$

where again we have referred to Fig. 2. The substitution $[H\backslash\cup\{E \mid h\colon H\}]$ we call *Set Shadow*; in our (v, h)-semantics we write it $[h{\Leftarrow}E]$.

Since *Set Shadow* is again a substitution for H it too affects only the (translated) modal formulae:

$$
\begin{array}{lll}
 & [H\backslash\cup\{E \mid h\colon H\}][\![\mathrm{P}\Phi]\!] & \\
= & [H\backslash\cup\{E \mid h\colon H\}](\exists h\colon H \cdot \Phi) & \text{“translate”}\\
= & [H\backslash\cup\{E \mid h\colon H\}](\exists h'\colon H \cdot \Phi') & \text{“let } \Phi' \text{ be } [h\backslash h']\Phi \text{ for clarity”}\\
= & (\exists h'\colon \cup\{E \mid h\colon H\} \cdot \Phi') & \text{“substitute”}\\
= & (\exists h\colon H \cdot (\exists h' \cdot h'{\in}E \wedge \Phi')) & \text{“convert } \cup \text{ to } \exists\text{”}\\
= & [\![\ \mathrm{P}(\exists h' \cdot h'{\in}E \wedge \Phi')\ ]\!] & \text{“retranslate”}\\
= & [\![\ \mathrm{P}(\exists h\colon E \cdot \Phi)\ ]\!] & \text{“remove primes”}\\
= & [\![\ [h{\Leftarrow}E]\mathrm{P}\Phi\ ]\!]. & \text{“definition } \textit{Set Shadow}\text{”}
\end{array}
$$

Note that h's in E are not captured by the quantifier $(\exists h \cdots)$: rather they are implicitly captured by the modality P. Collecting this together gives us

$$
wp.(h{:\in}\, E).\Psi \quad = \quad (\forall h\colon E \cdot [h{\Leftarrow}E]\Psi)
$$

which again will be the definition we give in Fig. 8. The synthesised definition of *Set Shadow* joins the other *technical transformers* in Fig. 7.

8.4. Adherence to the principles

The *wp*-logic of Figs. 7 and 8 has the following significant features, some of which (1)–(4) bear directly on the principles we set out in Section 2, and some of which (5)–(7) are additional properties desirable in their own right.

(1) *Refinement is monotonic — RP0.*

In view of (6) this requires only a check of Figs. 7 and 8 for the transformers' monotonicity with respect to ⇒.

(2) *All visible-variable-only refinements remain valid — RP1.*

Reference to Fig. 2 confirms that for any visible-only program fragment S its ignorance-sensitive conversion [[S]] has exactly the conventional effect on variable v and no effect on the shadow H (or indeed on h).

(3) *All structural refinements remains valid — RP2.*

At this point we define *structural* to mean "involving only Demonic choice, Composition, Identity and right-hand distributive properties of Conditional".

Such properties can be proved using the operational definitions in Fig. 2, which in the first three cases above are exactly the same as their classical counterparts, merely distributing the relevant operator outwards: that is why their program-algebraic properties are the same.

In the fourth case, Conditional, distribution from the *left* is not considered structural as it requires inspection of the condition E; distribution from the right however remains as in the classical case. Thus we retain for example

$$
(\mathbf{if}\ E\ \mathbf{then}\ \mathrm{S1}\ \mathbf{else}\ \mathrm{S2}\ \mathbf{fi});\ \mathrm{S} \quad = \quad \mathbf{if}\ E\ \mathbf{then}\ (\mathrm{S1};\ \mathrm{S})\ \mathbf{else}\ (\mathrm{S2};\ \mathrm{S})\ \mathbf{fi}.
$$

"Meta-" modality "M" stands for both K and P.

Substitute	$[e\backslash E]$	Replaces e by E, with alpha-conversion as necessary if distributing through $\forall, \exists$.

Distribution through M however is affected by the modalities' implicit quantification over hidden variables: if e is a hidden variable, then $[e\backslash E]\mathrm{M}\Phi$ is just $\mathrm{M}\Phi$; and if E contains hidden variables, the substitution does not distribute into $\mathrm{M}\Phi$ at all (which therefore requires simplification by other means).

Shrink Shadow	$[\Downarrow E]$	Distributes through all classical operators, with renaming; has no effect on classical atomic formulae.

We have $[\Downarrow E]\mathrm{P}\Phi \mathrel{\widehat{=}} \mathrm{P}(E \wedge \Phi)$ and $[\Downarrow E]\mathrm{K}\Phi \mathrel{\widehat{=}} \mathrm{K}(E \Rightarrow \Phi)$; in both cases, hidden variables in E are *not* renamed.

Set hidden	$[h \leftarrow E]$	Distributes through all operators, including M, with renaming as necessary to avoid capture by $\forall, \exists$ (but not M). Replaces h by E.

Set Shadow	$[h \Leftarrow E]$	Distributes through all classical operators, with renaming; has no effect on classical atomic formulae.

For modal formulae $[h \Leftarrow E]\mathrm{P}\Phi \mathrel{\widehat{=}} \mathrm{P}(\exists h\colon E \cdot \Phi)$; and $[h \Leftarrow E]\mathrm{K}\Phi \mathrel{\widehat{=}} \mathrm{K}(\forall h\colon E \cdot \Phi)$; note that h's in E (if any) are not captured by the introduced quantifier.

Fig. 7. Technical predicate transformers.

Identity	$wp.\mathbf{skip}.\Psi$	$\widehat{=}$	Ψ
Assign to visible	$wp.(v{:=}\,E).\Psi$	$\widehat{=}$	$[e\backslash E]\,[\Downarrow e{=}E]\,[v\backslash e]\,\Psi$
Choose visible	$wp.(v{:\in}\,E).\Psi$	$\widehat{=}$	$(\forall e\colon E \cdot [\Downarrow e{\in}E]\,[v\backslash e]\,\Psi)$
Assign to hidden	$wp.(h{:=}\,E).\Psi$	$\widehat{=}$	$[h \leftarrow E]\,\Psi$
Choose hidden	$wp.(h{:\in}\,E).\Psi$	$\widehat{=}$	$(\forall h\colon E \cdot [h \Leftarrow E]\,\Psi)$
Demonic choice	$wp.(\mathrm{S1} \sqcap \mathrm{S2}).\Psi$	$\widehat{=}$	$wp.\mathrm{S1}.\Psi \;\wedge\; wp.\mathrm{S2}.\Psi$
Composition	$wp.(\mathrm{S1};\mathrm{S2}).\Psi$	$\widehat{=}$	$wp.\mathrm{S1}.(wp.\mathrm{S2}.\Psi)$
Conditional	$wp.(\mathbf{if}\ E\ \mathbf{then}\ \mathrm{S1}\ \mathbf{else}\ \mathrm{S2}\ \mathbf{fi}).\Psi$		
		$\widehat{=}$	$E \Rightarrow [\Downarrow E]\,wp.\mathrm{S1}.\Psi \;\wedge\; \neg E \Rightarrow [\Downarrow \neg E]\,wp.\mathrm{S2}.\Psi$
Declare visible	$wp.(\textsc{vis}\ v).\Psi$	$\widehat{=}$	$(\forall e \cdot [v\backslash e]\,\Psi)$
Declare hidden	$wp.(\textsc{hid}\ h).\Psi$	$\widehat{=}$	$(\forall e \cdot [h \leftarrow e]\,\Psi)$

Note that both these substitutions propagate within modalities in Ψ.

Logical variable e is fresh.

The *assign to visible* rule has two components conceptually. The first is of course an assignment of E to v, although this is split into two sections $[e\backslash E] \cdots [v\backslash e]$ so that v's initial- and final values are distinguished in between, necessary should v occur in E.
The second is the "collapse" of ignorance caused by E's value being revealed: the effect of this is inserted by $[\Downarrow e{=}E]$, into the body of modalities.

Fig. 8. Weakest-precondition modal semantics.

$$
\begin{array}{ll}
 & wp.(v{:\in}\,T).(\mathrm{P}(h{=}C)) \\
\equiv & \text{“Choose visible”} \\
 & (\forall e{:}\,T \cdot [\Downarrow e{\in}T]\,[v\backslash e]\,\mathrm{P}(h{=}C)) \\
\equiv & \text{“}v\text{ not free”} \\
 & (\forall e{:}\,T \cdot [\Downarrow e{\in}T]\,\mathrm{P}(h{=}C)) \\
\equiv & \text{“Shrink Shadow”} \\
 & (\forall e{:}\,T \cdot \mathrm{P}(e{\in}T \wedge h{=}C))) \\
\equiv & \text{“}h\text{ not free in }e{\in}T\text{”} \\
 & (\forall e{:}\,T \cdot e{\in}T \wedge \mathrm{P}(h{=}C)) \\
\equiv & \mathrm{P}(h{=}C) \quad \text{“}e\text{ not free in }\mathrm{P}(h{=}C)\text{”}
\end{array}
$$

$$
\begin{array}{ll}
 & wp.(v{:=}\,h).(\mathrm{P}(h{=}C)) \\
\equiv & \text{“Assign to visible”} \\
 & [e\backslash h]\,[\Downarrow e{=}h]\,[v\backslash e]\,\mathrm{P}(h{=}C) \\
\equiv & \text{“}v\text{ not free; Shrink Shadow”} \\
 & [e\backslash h]\,\mathrm{P}(e{=}h \wedge h{=}C) \\
\equiv & [e\backslash h]\,\mathrm{P}(e{=}C \wedge h{=}C) \qquad \text{“}h{=}C\text{”} \\
\equiv & \text{“}h\text{ not free in }e{=}C\text{”} \\
 & [e\backslash h]\,(e{=}C \wedge \mathrm{P}(h{=}C)) \\
\equiv & h{=}C \wedge \mathrm{P}(h{=}C) \qquad \text{“}e\text{ not free”} \\
\equiv & h{=}C \qquad \text{“}\models \Phi \Rightarrow \mathrm{P}\Phi\text{”}
\end{array}
$$

We exploit that $\models \mathrm{P}(\Phi \wedge \Psi) \equiv \Phi \wedge \mathrm{P}\Psi$ when Φ contains no hidden variables.

The right-hand calculation shows that $h{=}C$ is the weakest precondition Φ such that $\{\Phi\}\ v{:=}\,h\ \{\mathrm{P}(h{=}C)\}$, yet $(\mathsf{v}, \mathsf{h}, \mathsf{H}) \models \mathrm{P}(h{=}C) \Rightarrow (h{=}C)$ for all C only when $\mathsf{H} = \{\mathsf{h}\}$. Thus, when the expression T contains no h, the fragment $v{:\in}\,T$ can be replaced by $v{:=}\,h$ only if we know h already.

Fig. 9. Avoiding the Refinement Paradox via *wp*-calculations.

(4) *Referential transparency remains valid — RT.*

The operational definitions show that effect of a program S, if *classically* interpreted over its variables v, h, is the same as the effect of ⟦S⟧ on those variables (*i.e.* ignoring H). Thus two v, h-expressions being equal carries over from S to ⟦S⟧, where we can appeal to classical referential transparency in the v, h, H semantics in order to replace equals by equals.

(5) *The ignorance-sensitive predicate transformers distribute conjunction*, as classical transformers do [5]. Thus complicated postconditions can be treated piecewise.

This can be seen by inspection of Fig. 8.

(6) *Non-modal postconditions can be treated using classical semantics [4,5]*, even if the program contains hidden variables.

This follows from inspection of Figs. 7 and 8 to show that the transformers reduce to their classical definitions when there are no modalities.

(7) *Modal semantics can be restricted to modal conjuncts.*

From (5) we can treat each conjunct separately; from (6) we can use classical semantics on the non-modal ones. Thus the modal semantics is required only for the others.

8.5. Example of *wp*-reasoning

Fig. 9 uses weakest preconditions to support our earlier claim (Section 7.3.2) that the Refinement Paradox is resolved logically: if $\not\models wp.\mathrm{S1}.\Psi \Rightarrow wp.\mathrm{S2}.\Psi$, then from (5) we know $\{wp.\mathrm{S1}.\Psi\}\ \mathrm{S1}\ \{\Psi\}$ holds (perforce) but $\{wp.\mathrm{S1}.\Psi\}\ \mathrm{S2}\ \{\Psi\}$ does not — and that is the situation in Fig. 9.

9. Examples of further techniques

Beyond the general identities introduced above, a number of specific techniques are suggested by the examples and case studies that we address below. Here we introduce some of them, in preparation for the *Encryption Lemma* (Section 10) and the *Oblivious Transfer Protocol* (Section 11).

9.1. Explicit atomicity

If we know that only our initial and final states are observable, why must we assume a "strong" adversary, with perfect recall and access to program flow?

The answer is that if we do *not* make those assumptions then we cannot soundly develop our program by stepwise refinement. But if we are prepared to give up stepwise refinement *in a portion* of our program, then we can *for that portion* assume our adversary is "weak", seeing only initial and final values.

We formalise that via special brackets $\{\{\cdot\}\}$ whose interior will have a dual meaning: at run time, a *weak adversary* is assumed there (an advantage); but, at development time, refinement is *not allowed* there (a disadvantage). Thus it is a trade-off. Intuitively we say that $\{\{S\}\}$ interprets S atomically even when S is compound: it cannot be internally observed; but neither can it be internally refined.

Thus for example $\{\{h{:=}\,0 \sqcap h{:=}\,1\}\}$ is equal to the atomic $h{:\in}\,\{0,1\}$ — in the former, the atomicity brackets prevent an observer's using the program flow at runtime to determine the value of h, and so the $\{\{\cdot\}\}$'d choice-point $\sqcap$ is not the security leak it would normally be. But the price we consequentially pay is that we are not allowed to use refinement there at development-time: for example, although the refinement $(h{:=}\,0 \sqcap h{:=}\,1) \sqsubseteq h{:=}\,0$ is trivial, nevertheless we cannot apply that refinement *within* the brackets $\{\{\cdot\}\}$ to conclude

$$\{\{h{:=}\,0 \sqcap h{:=}\,1\}\} \quad ?\!\sqsubseteq \quad \{\{h{:=}\,0\}\}$$

— and doing that is equivalent to asserting the falsehood $h{:\in}\,\{0,1\}\ ?\!\sqsubseteq h{:=}\,0$. (The *lhs* achieves postcondition $\mathrm{P}(h{=}1)$; the *rhs* does not.)

We do not give the formal definition of $\{\{\cdot\}\}$ here[9]; but we note several useful and reasonable properties of it.

Proposition 1. *If program S is syntactically atomic then* $S = \{\{S\}\}$. □

Proposition 2. *For all programs S we have* $S \sqsubseteq \{\{S\}\}$. □

Proposition 3. *If* either *of the programs $S1, S2$ does not assign to any visible variable, or (more generally) program $S1$'s final visibles are completely determined by its initial visibles, or program $S2$'s initial visibles are completely determined by its final visibles, then* $\{\{S1;S2\}\} = \{\{S1\}\}; \{\{S2\}\}$. □

For Proposition 3, informally, we note that allowing intrusion at a semicolon is harmless when there is nothing (new) for the run-time attacker to see: for example, if $S1$ changes no visible variable, then any visible values revealed at the semicolon were known at the beginning of the whole statement already; if $S2$ does not, then they will be known at the end.

9.2. Referential transparency and classical reasoning

A classical use of referential transparency is to reason that

- A program fragment $S1$ say establishes some (classical) formula ϕ over the program state;
- That ϕ implies the equality of two expressions E and F; and that therefore
- Expression E can be replaced by expression F in some fragment $S2$ that immediately follows $S1$.

The same holds true in our extended context, provided we realise that $S1$ must establish ϕ in *all possible states*: in effect we demand $\mathrm{K}\phi$ rather than just ϕ. In this section we check that this happens automatically, even when visibles and hiddens are mixed, and that thus the classical use of *RT* continues to apply.

Because $\mathrm{K}\phi$ is itself not ignorant, we use it negatively by appealing to "coercions": a *coercion* is written $[\Phi]$ for some formula Φ and is defined[10]

$$\textit{Coerce to } \Phi \ — \qquad wp.[\Phi].\Psi \ \ \widehat{=}\ \ \Phi \Rightarrow \Psi\,. \tag{7}$$

The principal use of a coercion is to formalise the reasoning pattern above, *i.e.* the establishing of some ϕ in all potential states (by $S1$) and its use as a context for *RT* (in $S2$). We begin with using a coercion as a context.

Lemma 2. *For any variable x, whether visible or hidden, and classical formula ϕ such that $\phi \Rightarrow E{=}F$, we have*

$$[K\phi];\, x{:\in}\, E \quad = \quad [K\phi];\, x{:\in}\, F\,.$$

Proof. Appendix. □

Note that the result specialises to assignments, as those are just singleton choices, and that we can view the lemma as formalising *RT*: the truth of $\mathrm{K}\phi$ establishes the context in which $E{=}F$, allowing their use interchangeably.

We now address how coercions are introduced, moved and finally removed. Introduction and removal are easily dealt with, as the following lemma shows that any coercion can be introduced at any point, up to refinement. The coercion [TRUE] can always be removed:

Lemma 3. *We have* [TRUE] = **skip** *and* **skip** $\sqsubseteq [\Phi]$ *for any* Φ.

Proof. Trivial. □

The only problem thus is removing a coercion that is *not* just [TRUE] — and that is done by moving it forward through statements whose combined effect is to establish it, *i.e.* to "make" it true operationally:

[9] The details are available in an addendum [16]. In particular Proposition 1 is important as a statement of consistency, since our definitions in Fig. 8 should be equally valid if presented as lemmas proved directly from the definition of $\{\{\cdot\}\}$.

[10] Coercions are dual to the more familiar *assertions* that allow abnormal termination if they fail (*i.e.* **abort**); in contrast, coercions cause backtracking if they fail (*i.e.* **magic**) [6].

Lemma 4. *For classical formulae ϕ, ψ and program S, we have*

$$\models \phi \Rightarrow wp.S.\psi \quad \textit{implies} \quad S;\, [K\psi] \sqsubseteq [K\phi];\, S.$$

Proof. Appendix. □

The importance of Lemma 4 is that its assumption $\models \phi \Rightarrow wp.S.\psi$ can usually be established by classical reasoning (recall Section 8(6)). In the special case where ϕ and ψ are the same, we say that S *preserves* ϕ and, as a result, can think of "pushing [Kϕ] to the left, through S" as being a refinement.

In summary, the above lemmas work together to provide the formal justification for carrying information from one part of a program to another, as in the following example. Suppose we have a linear fragment

$$S;\ SS;\ x{:=}\,E$$

where, informally, program S establishes some classical property ϕ, that property can be shown by classical reasoning to be preserved by SS, and it implies that $E{=}F$. Then to change E to F in the final statement, we would reason as follows:

$$\begin{array}{lll}
 & S;\ SS;\ x{:=}\,E & \\
\sqsubseteq & S;\ SS;\ [\mathrm{K}\phi];\, x{:=}\,E & \text{“Lemma 3”} \\
= & S;\ SS;\ [\mathrm{K}\phi];\, x{:=}\,F & \text{“assumption; Lemma 2”} \\
= & S;\ \ SS;\, [\mathrm{K}\phi];\ \ x{:=}\,F & \text{“associativity”} \\
\sqsubseteq & S;\ \ [\mathrm{K}\phi];\, SS;\ \ x{:=}\,F & \text{“}SS\text{ preserves }\phi\text{”} \\
= & S;\, [\mathrm{K}\phi];\ \ SS;\ x{:=}\,F & \text{“associativity”} \\
\sqsubseteq & [\textsc{true}];\, S;\ SS;\ x{:=}\,F & \text{“}S\text{ establishes }\phi\text{”} \\
= & S;\ SS;\ x{:=}\,F. & \text{“Lemma 3”}
\end{array}$$

Thus the general pattern is to assume the property ϕ of the state necessary to replace E by F, and then to move it forward, preserved, through the program SS to a point S at which it can be shown to have been established. An example of this is given in Section 10(5).

We have thus checked that the "establish context, then use it" paradigm continues to operate in the normal way, even with mixed visibles and hiddens; therefore we will usually appeal to the above results only informally.

9.3. Hiding leads to refinement

We show that changing a variable from visible to hidden, leaving all else as it is, has the effect of a refinement: informally, that is because the nondeterminism has not been changed while the ignorance can only have increased. More precisely, we show that

$$|[\ \textsc{vis}\ v\ \cdot\ \ S(v)\ ]| \quad \sqsubseteq \quad |[\ \textsc{hid}\ h\ \cdot\ \ S(h)\ ]|$$

for any program $S(v)$ possibly containing references to variable v. This is a *syntactic* transformation, as we can see by comparing the two fragments

$$|[\ \textsc{vis}\ v\ \cdot\ (v{:=}\,0 \sqcap v{:=}\,1);\ h'{:=}\,v\ ]| \ \text{ and } \ |[\ \textsc{vis}\ v\ \cdot\ v{:\in}\,\{0,1\};\ h'{:=}\,v\ ]| \tag{8}$$

in the context of global hidden h'; they are both equivalent semantically to $h'{:=}\,0 \sqcap h'{:=}\,1$ since the assignment to v is visible in each case. However, transforming visible v to hidden h gives respectively

$$|[\ \textsc{hid}\ h\ \cdot\ (h{:=}\,0 \sqcap h{:=}\,1);\ h'{:=}\,h\ ]| \ \text{ and } \ |[\ \textsc{hid}\ h\ \cdot\ h{:\in}\,\{0,1\};\ h'{:=}\,h\ ]|, \tag{9}$$

which differ semantically: the *lhs* remains $h'{:=}\,0 \sqcap h'{:=}\,1$, since the resolution of $\sqcap$ is visible (even though h itself is not); but the *rhs* is now $h'{:\in}\,\{0,1\}$. Note that ((9), *rhs*) is a refinement of ((8), *rhs*) nevertheless, which is precisely what we seek to prove in general.

Lemma 5. *If program $S2$ is obtained from program $S1$ by a syntactic transformation in which some local visible variable is changed to be hidden, then $S1 \sqsubseteq S2$.*

Proof. Routine applications of definitions, as shown in Appendix. □

10. The encryption lemma

In this section we see our first substantial proof of refinement; and we prepare for our treatment of the *OTP* (Oblivious Transfer Protocol).

When a hidden secret is encrypted with a hidden key and the result published as a visible message, the intention is that adversaries ignorant of the key cannot use the message to deduce the secret, even if they know the encryption method. A special case of this occurs in the *OTP*, where a secret is encrypted (via exclusive-or) with a key (a hidden Boolean) and becomes a message (is published). We examine this simple situation in the ignorance logic.

Lemma 6. *Let $s{:}\,S$ be a secret, let $k{:}\,K$ be a key, and let $\oslash$ be an encryption method so that $s \oslash k$ is the encryption of s. Then in a context including* HID *s we have the refinement*

$$\textbf{skip} \quad \sqsubseteq \quad |[\ \textsc{vis}\ m;\ \textsc{hid}\ k \cdot \ \ k{:\in}K;\ m{:=}s\oslash k\ ]|,$$

which expresses that publishing the encryption $s\oslash k$ as a visible message m reveals nothing about the secret s, provided the key k is hidden and the following Key-Complete Condition *is satisfied*[11]*:*

$$KCC - \qquad (\exists\ set\ M \cdot (\forall s{:}\ S \cdot s\oslash K = M)). \tag{10}$$

By $s\oslash K$ we mean $\{s\oslash k \mid k{:}\ K\}$, that is the set formed by $\oslash$-ing the secret s with all possible keys in K. Informally KCC states that the set of possible encryptions is the same for all secrets — it is some fixed set M.

Proof. Assume a context containing the declaration $\textsc{hid}\ s{:}\ S$. Then we reason[12]

$$\begin{array}{lll}
 & \textbf{skip} & \\
= & |[\ \textsc{vis}\ m \cdot m{:\in}M\ ]| & \text{“for any fixed non-empty set } M\text{”} \rightarrow(1)\\
= & |[\ \textsc{vis}\ m,\ v \cdot v{:\in}M;\ m{:=}v\ ]| & \text{“visible-only reasoning”}\\
\sqsubseteq & |[\ \textsc{vis}\ m;\ \textsc{hid}\ h \cdot h{:\in}M;\ m{:=}h\ ]| & \text{“Lemma 5 and renaming”} \rightarrow(2)\\
= & |[\ \textsc{vis}\ m;\ \textsc{hid}\ h \cdot h{:\in}s\oslash K;\ m{:=}h\ ]| & \text{“}s\in S \text{ and } (\forall s{:}\ S \cdot s\oslash K = M)\text{”} \rightarrow(3)\\
 & & \\
\sqsubseteq & & \text{“using atomicity-style reasoning”} \rightarrow(4)\\
 & |[\ \textsc{vis}\ m;\ \textsc{hid}\ h \cdot |[\ \textsc{hid}\ k \cdot k{:\in}K;\ h{:=}s\oslash k\ ]|;\ m{:=}h\ ]| & \\
 & & \\
= & |[\ \textsc{vis}\ m;\ \textsc{hid}\ h,\ k \cdot k{:\in}K;\ h{:=}s\oslash k;\ m{:=}h\ ]| & \text{“rearrange scopes”}\\
= & |[\ \textsc{vis}\ m;\ \textsc{hid}\ h,\ k \cdot k{:\in}K;\ h{:=}s\oslash k;\ m{:=}s\oslash k\ ]| & \text{“}RT\text{”} \rightarrow(5)\\
= & |[\ \textsc{vis}\ m;\ \textsc{hid}\ k \cdot k{:\in}K;\ m{:=}s\oslash k\ ]|. & \text{“}h \text{ now superfluous”} \rightarrow(6)
\end{array}$$

Our only assumption was the KCC at Step (3). □

As a commentary on the proof steps we provide the following:

(1) This and the next step are classical, visible-only reasoning.
(2) The choice from M was visible; now it is hidden.
(3) From s's membership of its type S and the KCC we thus know that $s\oslash K$ and M are equal, whatever value the secret s might have, and so RT applies. This is the crucial condition, and it does not depend on M directly — all that is required is that there be such an M.
(4) To justify this step in detail we introduce atomicity briefly, to simplify the reasoning; then we remove it again. We have

$$\begin{array}{lll}
 & h{:\in}s\oslash K & \\
= & |[\ \textsc{hid}\ k \cdot k{:\in}K\ ]|;\ h{:\in}s\oslash K & \text{“introduced statement is } \textbf{skip}\text{”}\\
= & |[\ \textsc{hid}\ k \cdot k{:\in}K;\ h{:\in}s\oslash K\ ]| & \text{“adjust scopes”}\\
\sqsubseteq & |[\ \textsc{hid}\ k \cdot \{\{k{:\in}K;\ h{:\in}s\oslash K\}\}\ ]| & \text{“Proposition 2”}\\
= & |[\ \textsc{hid}\ k \cdot \{\{k{:\in}K;\ h{:=}s\oslash k;\ k{:\in}K\}\}\ ]| & \text{“classical reasoning”}\\
= & |[\ \textsc{hid}\ k \cdot k{:\in}K;\ h{:=}s\oslash k;\ k{:\in}K\ ]| & \text{“Proposition 3; Proposition 1”}\\
= & |[\ \textsc{hid}\ k \cdot k{:\in}K;\ h{:=}s\oslash k\ ]|. & \text{“}k \text{ local: discard final assignment”}
\end{array}$$

(5) That $(h{:=}s\oslash k;\ m{:=}h) = (h{:=}s\oslash k;\ m{:=}s\oslash k)$ cannot be shown by direct appeal to classical reasoning, simple as it appears, because it mixes hidden (h, s, k) and visible (m) variables. Nevertheless, it is a simple example of the coercion reasoning from Section 9.2; in detail we have

$$\begin{array}{lll}
 & h{:=}s\oslash k;\ \ m{:=}h & \\
\sqsubseteq & h{:=}s\oslash k;\ \ [\mathrm{K}(h = s\oslash k)];\ m{:=}h & \text{“Lemma 3”}\\
= & h{:=}s\oslash k;\ \ [\mathrm{K}(h = s\oslash k)];\ m{:=}s\oslash k & \text{“Lemma 2”}\\
= & h{:=}s\oslash k;\ [\mathrm{K}(h = s\oslash k)];\ \ m{:=}s\oslash k & \text{“associativity”}\\
\sqsubseteq & [\textsc{true}]h{:=}s\oslash k;\ \ m{:=}s\oslash k & \text{“Lemma 4”}\\
= & h{:=}s\oslash k;\ \ m{:=}s\oslash k. & \text{“Lemma 3”}
\end{array}$$

Alternatively we could see it as an application of RT, in the spirit of the comment following Lemma 2.

(6) This is justified by the general equality

$$|[\ \textsc{hid}\ h \cdot h{:\in}E\ ]| \quad = \quad \textbf{skip},$$

which can be proved by direct wp-reasoning. Note however that for visibles $|[\ \textsc{vis}\ v \cdot v{:\in}E\ ]| \neq \textbf{skip}$ in general: let E be $\{h\}$ for example.

[11] The Key-Complete Condition is very strong, requiring as many potential keys as messages; yet it applies to the *OTP*.

[12] This can be done by direct *wp*-calculation also [1]; we thank Lambert Meertens for suggesting it be done in the program algebra.

11. Deriving the Oblivious Transfer Protocol

Rivest's Oblivious Transfer Protocol is an example of ignorance preservation [10].[13] From Rivest's description (paraphrased) the specification is as below; we use $\oplus$ for exclusive-or throughout.

Specification We assume that *Alice* has two messages $m_0, m_1 \colon \mathbb{B}^N$, where we write $\mathbb{B} \mathrel{\widehat{=}} \{0, 1\}$ for the Booleans as bits. The protocol is to ensure that *Bob* will obtain m_c for his choice of $c \colon \mathbb{B}$. But Alice is to have no idea which message Bob chose, and Bob is to learn nothing about Alice's other message $m_{1 \oplus c}$.

Rivest's solution introduces a trusted intermediary *Ted*, who is active only at the beginning of the protocol and, in particular, knows neither Alice's messages $m_{0,1}$ nor Bob's choice c. He participates as follows:

Setup Ted privately gives Alice two random N-bit strings $r_0, r_1 \colon \mathbb{B}^N$; and he flips a bit d, then giving both d and r_d privately to Bob. Ted is now done, and can go home.

Once Ted has gone, the protocol continues between Alice and Bob:

Request Bob chooses privately a bit $c \colon \mathbb{B}$; he wants to obtain m_c from Alice. He sends Alice the bit $e \mathrel{\widehat{=}} c \oplus d$.
Reply Alice sends Bob the values $f_0, f_1 \mathrel{\widehat{=}} m_0 \oplus r_e, m_1 \oplus r_{1 \oplus e}$.
Conclusion Bob now computes $m \mathrel{\widehat{=}} f_c \oplus r_d$.

The whole protocol is summarised in Fig. 10.

Given that there are three principals, we have three potential points of view. We will take Bob's for this example, and the declarations of Fig. 10 therefore become the *context* for our derivation:

$$\begin{array}{lll} \text{Globals:} & \textsc{hid}\ m_0, m_1 \colon \mathbb{B}^N; & \text{— Bob cannot see } m_{0,1}. \\ & \textsc{vis}\ c \colon \mathbb{B}\ \cdot & \\ \text{Locals:} & \textsc{hid}\ r_0, r_1 \colon \mathbb{B}^N; & \text{— Bob cannot see } r_{0,1}. \\ & \textsc{vis}\ r, f_0, f_1 \colon \mathbb{B}^N;\ d, e \colon \mathbb{B}\ \cdot & \end{array} \tag{11}$$

The types are unchanged but the visibility attributes are altered *à la Bob*.[14]

The significance of *e.g.* declaring m_0 to be global while r_0 is local is that local variables are exempted from refinement: a simple classical example in the global context **var** g is the refinement

$$|[\ \mathbf{var}\ l_1 \cdot l_1 := 1;\ g := g + l_1\ ]| \quad \sqsubseteq \quad |[\ \mathbf{var}\ l_2 \cdot l_2 := -1;\ g := g - l_2\ ]|,$$

where the two fragments are in refinement (indeed both are equal to $g := g + 1$) in spite of the fact that the *lhs* does not allow manipulation of l_2 and the *rhs* does not implement $l_1 := 1$. Thus the variables we make local are those in which we have no interest *wrt* refinement.

Rather than prove directly that Rivest's protocol meets some logical pre/post-style specification, instead

> *We use program algebra to manipulate a specification-as-program whose correctness is obvious.*

In this case, the specification is just $m := m_c$, and we think it is obvious that it reveals nothing about $m_{1 \oplus c}$ to Bob: indeed that variable is not even mentioned.[15] Under the declarations of (11), we will show it can be refined to the code of Fig. 10, which is sufficient because (following Mantel [18])

> *An implementation reached via ignorance-preserving refinement steps requires no further proof of ignorance-preservation.*

An immediate benefit of such an approach is that we finesse issues like "if it is known that $m_0 = m_1$ and Bob asks for m_0, then he has learned m_1 — isn't that incorrect?" A pre/post-style direct proof of the implementation would require in the precondition an assertion that m_0 and m_1 were independent, were *not* related in any way, in order to conclude that nothing about $m_{1 \oplus c}$ is revealed. But a refinement-algebraic approach does not require that: it is obvious from the *specification* the role that independence plays in the protocol, and the derivation respects that automatically and implicitly due to its semantic soundness.

We now give the derivation: in the style of Section 10, a commentary on the proof steps follows; in some cases arrows → in the left margin indicate a point of interest. Within Bob's context (11), that is $\textsc{hid}\ m_0, m_1 \colon \mathbb{B}^N;\ \textsc{vis}\ c \colon \mathbb{B}$ we reason

[13] Earlier [1] we derived Chaum's *Dining Cryptographers' Protocol* [17] instead.

[14] For comparison, we note that Alice's point of view is

$$\begin{array}{ll} \textsc{vis}\ m_0, m_1 \colon \mathbb{B}^N;\ \textsc{hid}\ c \colon \mathbb{B}\ \cdot & \text{— Alice cannot see } c. \\ \textsc{vis}\ r_0, r_1, f_0, f_1 \colon \mathbb{B}^N;\ e \colon \mathbb{B};\ \textsc{hid}\ r \colon \mathbb{B}^N;\ d \colon \mathbb{B}\ \cdot & \text{— Alice cannot see } r, d. \end{array}$$

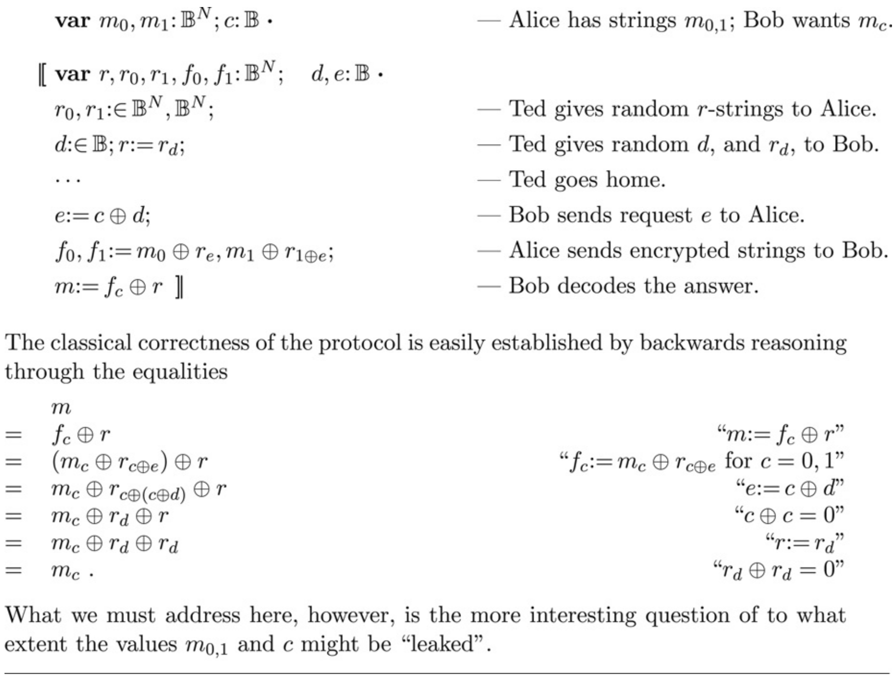
It is not a simple inversion, since the local variables $f_{0,1}$ and e, that is the messages passed between Alice and Bob, are visible to both principals.

[15] When a specification is not so simple, one can elucidate its properties with logical experiments, *i.e.* attempted proofs of properties one believes or hopes it has; our previous report [1] shows that for Chaum's *Dining Cryptographers* [17].

$$
\begin{array}{clll}
 & & m{:=}\,m_c & \text{“specification”} \\[1ex]
 & = & \mathbf{skip};\ m{:=}\,m_c & \text{“Identity”} \\[1ex]
 & = & \quad |[\ \textsc{vis}\ d, e{:}\,\mathbb{B}\ \bullet\ d{:\in}\,\mathbb{B};\ e{:=}\,c \oplus d\ ]|; & \text{“refine } \mathbf{skip}\text{: visible-only reasoning”}\ \rightarrow(7) \\
 & & \quad m{:=}\,m_c & \\[1ex]
 & = & |[\ \textsc{vis}\ d, e{:}\,\mathbb{B}\ \bullet\ d{:\in}\,\mathbb{B};\ e{:=}\,c \oplus d; & \text{“adjust scopes; introduce } \mathbf{skip} \text{ again”} \\
 & & \quad \mathbf{skip}; & \\
 & & \quad m{:=}\,m_c\ \ ]| & \\[1ex]
 & \sqsubseteq & |[\ \textsc{vis}\ d, e{:}\,\mathbb{B}\ \bullet\ d{:\in}\,\mathbb{B};\ e{:=}\,c \oplus d; & \text{“encryption lemma”}\ \rightarrow(8) \\
\rightarrow & & \quad |[\ \textsc{vis}\ f_0, f_1{:}\,\mathbb{B}^N;\ \textsc{hid}\ r_0, r_1 : \mathbb{B}^N\ \bullet & \\
\rightarrow & & \qquad r_0, r_1{:\in}\,\mathbb{B}^N, \mathbb{B}^N; & \\
\rightarrow & & \qquad f_0, f_1{:=}\,m_0 \oplus r_e,\ m_1 \oplus r_{1\oplus e}\ \ ]|; & \\
 & & \quad m{:=}\,m_c\ \ ]| & \\[1ex]
 & = & |[\ \textsc{vis}\ d, e{:}\,\mathbb{B};\ f_0, f_1{:}\,\mathbb{B}^N;\ \textsc{hid}\ r_0, r_1{:}\,\mathbb{B}^N\ \bullet & \text{“adjust scopes”} \\
 & & \quad d{:\in}\,\mathbb{B};\ e{:=}\,c \oplus d; & \\
 & & \quad r_0, r_1{:\in}\,\mathbb{B}^N, \mathbb{B}^N; & \\
 & & \quad f_0, f_1{:=}\,m_0 \oplus r_e,\ m_1 \oplus r_{1\oplus e}; & \\
 & & \quad m{:=}\,m_c\ \ ]| & \\[1ex]
 & = & & \text{“introduce } r\text{: visible-only reasoning”} \\
\rightarrow & & |[\ \textsc{vis}\ d, e{:}\,\mathbb{B};\ f_0, f_1, r{:}\,\mathbb{B}^N;\ \textsc{hid}\ r_0, r_1{:}\,\mathbb{B}^N\ \bullet & \\
 & & \quad d{:\in}\,\mathbb{B};\ e{:=}\,c \oplus d; & \\
 & & \quad r_0, r_1{:\in}\,\mathbb{B}^N, \mathbb{B}^N; & \\
 & & \quad f_0, f_1{:=}\,m_0 \oplus r_e,\ m_1 \oplus r_{1\oplus e}; & \\
 & & \quad m{:=}\,m_c; & \\
\rightarrow & & \quad r{:=}\,f_c \oplus m\ \ ]| & \\[1ex]
 & = & |[\ \textsc{vis}\ d, e{:}\,\mathbb{B};\ f_0, f_1, r{:}\,\mathbb{B}^N;\ \textsc{hid}\ r_0, r_1{:}\,\mathbb{B}^N\ \bullet & \text{“classical reasoning”}\ \rightarrow(9) \\
 & & \quad d{:\in}\,\mathbb{B};\ e{:=}\,c \oplus d; & \\
 & & \quad r_0, r_1{:\in}\,\mathbb{B}^N, \mathbb{B}^N; & \\
 & & \quad f_0, f_1{:=}\,m_0 \oplus r_e,\ m_1 \oplus r_{1\oplus e}; & \\
 & & \quad m{:=}\,m_c; & \\
\rightarrow & & \quad r{:=}\,r_d\ \ ]| & \\[1ex]
 & = & |[\ \textsc{vis}\ d, e{:}\,\mathbb{B};\ f_0, f_1, r{:}\,\mathbb{B}^N;\ \textsc{hid}\ r_0, r_1{:}\,\mathbb{B}^N\ \bullet & \text{“reordering”}\ \rightarrow(10) \\
 & & \quad r_0, r_1{:\in}\,\mathbb{B}^N, \mathbb{B}^N; & \\
 & & \quad d{:\in}\,\mathbb{B}; & \\
 & & \quad r{:=}\,r_d; & \\
 & & \quad e{:=}\,c \oplus d; & \\
 & & \quad f_0, f_1{:=}\,m_0 \oplus r_e,\ m_1 \oplus r_{1\oplus e}; & \\
 & & \quad m{:=}\,m_c\ ]| & \\[1ex]
 & = & |[\ \textsc{vis}\ d, e{:}\,\mathbb{B};\ f_0, f_1, r{:}\,\mathbb{B}^N;\ \textsc{hid}\ r_0, r_1{:}\,\mathbb{B}^N\ \bullet & \text{“classical reasoning”}\ \rightarrow(11) \\
 & & \quad r_0, r_1{:\in}\,\mathbb{B}^N, \mathbb{B}^N; & \\
 & & \quad d{:\in}\,\mathbb{B}; & \\
 & & \quad r{:=}\,r_d; & \\
 & & \quad e{:=}\,c \oplus d; & \\
 & & \quad f_0, f_1{:=}\,m_0 \oplus r_e,\ m_1 \oplus r_{1\oplus e}; & \\
\rightarrow & & \quad m{:=}\,f_c \oplus r\ ]|, &
\end{array}
$$

which concludes the derivation. Given our context (11), we have established that Rivest's protocol reveals no more about the (hidden) variables m_0, m_1 than does the simple assignment $m{:=}\,m_c$ that was our specification.

A technical point is that our manipulation of r – introduced "late" then moved "early" – illustrates a feature which seems to be typical: although r's operational role in the protocol is early (it is used by Ted), its conceptual role is late: in order to manipulate the *rhs* of the assignment to it, we require facts established by the earlier part of the program. Once that is done, it can be moved into its proper operational place. Similarly, the statement $m{:=}\,m_c$, from our specification, was carried through



```
$\mathbf{var}\ m_0, m_1 \colon \mathbb{B}^N; c \colon \mathbb{B}\ \cdot$          — Alice has strings $m_{0,1}$; Bob wants $m_c$.

$[\![\ \mathbf{var}\ r, r_0, r_1, f_0, f_1 \colon \mathbb{B}^N; \quad d, e \colon \mathbb{B}\ \cdot$
   $r_0, r_1 :\in \mathbb{B}^N, \mathbb{B}^N;$                 — Ted gives random $r$-strings to Alice.
   $d :\in \mathbb{B}; r := r_d;$                    — Ted gives random $d$, and $r_d$, to Bob.
   $\cdots$                                — Ted goes home.
   $e := c \oplus d;$                         — Bob sends request $e$ to Alice.
   $f_0, f_1 := m_0 \oplus r_e, m_1 \oplus r_{1 \oplus e};$       — Alice sends encrypted strings to Bob.
   $m := f_c \oplus r\ \ ]\!]$                      — Bob decodes the answer.
```

The classical correctness of the protocol is easily established by backwards reasoning through the equalities

$$\begin{array}{lll} & m & \\ = & f_c \oplus r & \text{“}m := f_c \oplus r\text{”} \\ = & (m_c \oplus r_{c \oplus e}) \oplus r & \text{“}f_c := m_c \oplus r_{c \oplus e} \text{ for } c = 0, 1\text{”} \\ = & m_c \oplus r_{c \oplus (c \oplus d)} \oplus r & \text{“}e := c \oplus d\text{”} \\ = & m_c \oplus r_d \oplus r & \text{“}c \oplus c = 0\text{”} \\ = & m_c \oplus r_d \oplus r_d & \text{“}r := r_d\text{”} \\ = & m_c\ . & \text{“}r_d \oplus r_d = 0\text{”} \end{array}$$

What we must address here, however, is the more interesting question of to what extent the values $m_{0,1}$ and c might be “leaked”.

Fig. 10. Rivests's *Oblivious Transfer Protocol.*

the whole derivation until the program text lying before it was sufficiently developed to justify via *RT* an equals-for-equals substitution in its *rhs*.

The commentary on the derivation is as follows:

(7) The order in which the variables are introduced is not necessarily their final ‘‘execution order’’; rather it is so that the right variables are in scope for subsequent steps. In this case our use of the encryption lemma refers to *e*, which thus must be introduced first.

(8) The visible variable (m) is the $2N$-bit pair f_0, f_1; strictly speaking there should be a single variable f on which a subscript operation $\cdot_i$ is defined, so that the syntax f_i is actually an expression involving two separate variables f and i. Similarly, the key (k) is the $2N$-bit pair r_0, r_1; and the encryption operation $(\oslash)$ is exclusive-or $\oplus$ between $2N$-bit strings, which satisfies *KCC*.

(9) The classical reasoning referred to shows that $r_d = f_c \oplus m$ at this point in the program, allowing an equals-for-equals substitution in the *rhs* of the assignment to r as described in Section 9.2. We could not introduce $r := r_d$ directly in the previous step because that reasoning would not have been visible-only: the expression r_d on the right refers to the hiddens $r_{0,1}$. (See the comment (8) that f_i is an abbreviation; the same is true of r and m.)

(10) We have used the principle that $(x := E;\ y := F) = (y := F;\ x := E)$, given the usual conditions that E contains no y and F no x, to shuffle the assignments around into their final order. (Recall (7) above for why they are out of order.) It holds even when some of the variables are hidden.

(11) The justification here is, as for (9), that classical reasoning (as in Fig. 10) establishes an equality at this point operationally, that $m_c = f_c \oplus r$, and then *RT* applies.

We conclude this example with some general comments on this style of development. The specification $m := m_c$ of the *OTP* is unsatisfactory as a direct implementation *only* because the *rhs* mixes variables of Alice and Bob, two principals who are physically separated: we are following the convention that statements $x_A := E_A$, where the $\cdot_A$ indicates ‘‘uses only variables located with Alice’’, describe computations that Alice undertakes on her own; on the other hand, statements $x_A := E_B$ describes a message E_B constructed by Bob and sent to Alice, received by her in x_A. Thus the difficulty with direct use of the specification is that the expression m_c is neither an E_A nor an E_B, and so there is no single ‘‘place’’ in which it can be evaluated.

Thus we refine ‘‘only’’ in order to solve the locality problem (but open a Pandora's Box in doing so). It could readily be avoided in a Bob's-view implementation such as

```
|[ VIS $m'_0, m'_1 : \mathbb{B}^N\ \cdot$        — Local variables of Bob.
   $m'_0, m'_1 := m_0, m_1;$         — Send both messages to Bob.                (12)
   $m := m'_c$ ]|,                   — Bob takes chosen message.
```

trivially a valid classical refinement and now locality-valid too, since both $m'_{0,1}$ and c are Bob's variables. But then it is just as trivially invalid for ignorance preservation since Bob can deduce $m_{1 \oplus c}$ by examining $m'_{1 \oplus c}$. Fortunately the rules of ignorance refinement prevent us from making this mistake, since in general we have **skip** $\not\sqsubseteq$ $|[\ \text{VIS}\ v \cdot v{:=}\,h\ ]|$, thus invalidating the very step that introduced the $m'_{0,1}$ variables.

12. Contributions, inspirations, comparisons and conclusions

Our contribution is to have altered the rules for refinement of sequential programs, just enough, so that ignorance of hidden variables is preserved: we can still derive correct protocols (Section 11), but can no longer mistakenly propose incorrect ones (Section 7.3).

Thus we allow Stepwise Refinement (Section 1) to be used for the development of security-sensitive sequential programs, avoiding the usual "paradoxes" that in the past have attended this (Section 7.3.2). But we must then adopt a pessimistic stance in which we treat even a "weak" adversary as if he were a "strong" one: refinement comes at a price.

We argued that this price is "non-negotiable" in the sense that any reasonable definition of refinement must pay it (Section 3); but it can be mitigated somewhat by indicating explicitly any portions of the program in which we are prepared to forego the ability to refine: declaring them "atomic" (Section 9.1) protects them from adversarial intrusion (good), but also disallows Stepwise Refinement within them (bad).

Part of the inspiration for our approach was work by Van der Meyden, Engelhardt and Moses who earlier treated the *OTP*, and Chaum's *Dining Cryptographers* [17,1], via a refinement-calculus of knowledge and ignorance [19]. That approach (together with advice from them) is the direct inspiration for what is here.

Compared to the work of Halpern and O'Neill, who apply the Logic of Knowledge to secrecy [11] and anonymity [20], ours is a very restricted special case: we allow just one agent; our $(\mathsf{v}, \mathsf{h}, \mathsf{H})$ model allows only h to vary in the Kripke model [14]; and our programs are not concurrent. What we add back – having specialised away so much – is reasoning in the *wp*-based assertional/sequential style, thus exploiting the specialisation to preserve traditional reasoning patterns where they can apply.

Comparison with *non-interference* [21] comes from regarding hidden variables as "high-security" and visible variables as "low-security", and concentrating on program semantics rather than *e.g.* extra syntactic annotations: thus we take the *extensional* view [22] of non-interference where security properties are deduced directly from the semantics of a program [23, III-A]. Recent examples of this include elegant work by Leino et al. [24] and Sabelfeld et al. [25].

Again we have specialised severely — we do not consider lattices, nor loops (and thus possible divergence), nor concurrency, nor probability. However our "agenda" of Refinement, the Logic of Knowledge, and Program Algebra, has induced four interesting differences from the usual approaches:

(1) *We do not prove "absolute" security of a program.* Rather we show that it is no less secure than another; this is induced by our refinement agenda. After all, the *OTP specification* is not secure in the first place: it reveals one of Alice's messages to Bob. (To attempt to prove the *OTP implementation* absolutely secure is therefore pointless.)

Thus there is similarity of aims with *delimited information release* approaches. In this example borrowed from Sabelfeld and Myers [26, *Average salary*] we suppose two hidden variables declared $\text{HID}\ h_1, h_2$ whose sum we are allowed to release, but not their actual values. Our specification would be the "procedure-" or "method" call[16]

$$\textit{(Avg)} \qquad |[\ \text{VIS}\ v;\ \text{HID}\ p_1, p_2 \cdot p_1, p_2{:=}\,h_1, h_2;\ \ v{:=}\,p_1{+}p_2\ ]|,$$

that reveals that sum. The suggested attack is

$$\textit{(Avg-attack)} \qquad |[\ \text{VIS}\ v;\ \text{HID}\ p_1, p_2 \cdot p_1, p_2{:=}\,\overset{\downarrow}{h_1}, \overset{\downarrow}{h_1};\ \ v{:=}\,p_1{+}p_2\ ]|,$$

which clearly reveals h_1 by using the parameters in an unexpected way. Indeed we have *(Avg)* $\not\sqsubseteq$ *(Avg-attack)* — so we do not allow this either.

Thus our approach to delimited security is to write a specification in which the partial release of information is allowed: refinement then ensures that *no more than that* is released in any implementation.

(2) *We concentrate on* final- *rather than initial hidden values*. This is induced by the Kripke structure of the Logic of Knowledge approach (Section 4), which models what other states are possible "now" (rather than "then").

The usual approach relates instead to hidden *initial* values, so that $h{:=}\,0$ would be secure and $v{:=}\,h;\ h{:\in}\,T$ insecure; for us just the opposite holds. Nevertheless, we could achieve the same effect by operating on a local hidden copy, thrown away at the end of the block. Thus $|[\ \text{HID}\ h'{:}\ \{h\} \cdot h'{:=}\,0\ ]|$ is secure (for both interpretations), and $|[\ \text{HID}\ h'{:}\ \{h\} \cdot v{:=}\,h';\ h'{:\in}\,T\ ]|$ is insecure.

(3) *A direct comparison with non-interference* considers the relational semantics R of a program over $v, h{:}\,T$; the refinement $v{:\in}\,T \sqsubseteq v{:\in}\,R.v.h$ then expresses absolute security for the *rhs* with respect to h's initial (and final) value. We can then reason operationally, as follows.

Fig. 2 shows that from initial $\mathsf{v}, \mathsf{h}, \mathsf{H}$ the possible outcomes on the left are $(t, \mathsf{h}, \mathsf{H})$ for all $t{:}\,T$, and that they are $(t', \mathsf{h}, \{h : \mathsf{H} \mid t'{\in}R.\mathsf{v}.h\})$ on the right for all $t'{:}\,R.\mathsf{v}.\mathsf{h}$. For refinement from that initial state we must therefore have $t'{\in}R.\mathsf{v}.\mathsf{h} \Rightarrow t'{\in}T$, which is just type-correctness; but also

[16] Here we are simulating a function call $Avg\,(h_1, h_2)$ with formal parameters p_1, p_2 and formal return value v. These three variables are local because we are not interested in them directly (they "sit on the stack"): we care only about their effect on what we know about h_1, h_2.

$$t' \in R.\mathsf{v}.\mathsf{h} \quad \Rightarrow \quad \mathsf{H} \subseteq \{h : \mathsf{H} \mid t' {\in} R.\mathsf{v}.h\} \tag{13}$$

must hold, both conditions coming from the operational definition of refinement (Section 5.3). Since (13) constrains all initial states v, h, H, and all t', we close it universally over those variables to give a formula which via predicate calculus and elementary set-theory reduces to the non-interference condition $(\forall \mathsf{v}, \mathsf{h}, \mathsf{h}' \colon T \cdot R.\mathsf{v}.\mathsf{h} = R.\mathsf{v}.\mathsf{h}')$ as usually given for demonic programs [24,25].

(4) *We insist on perfect recall.* This is induced by our algebraic principles (Section 3.1), and thus we consider $v{:=}\,h$ to have revealed *h*'s value at that point, no matter what follows. The usual semantic approach allows instead a subsequent $v{:=}\,0$ to "erase" the information leak.

Perfect recall is also a side-effect of (thread) concurrency [11],[23, IV-B], but has different causes. We are concerned with ignorance-preservation during program *development*; the concurrency-induces-perfect-recall problem occurs during program *execution*. We commented on the link between the two in our discussion of atomicity (Section 9.1).

The "label creep" [23, II-E] caused by perfect recall, where the build-up of un-erasable leaks makes the program eventually useless, is mitigated because our knowledge of the *current* hidden values can decrease (via $h{:\in}\,T$ for example), even though knowledge of *initial*- (or even previous) values cannot.

(5) *We do not require "low-view determinism"* [23, IV-B]. This is induced by our explicit policy of retaining abstraction, and of determining exactly when we can "refine it away" and when we cannot. Roscoe and others instead require low-level behaviour to be deterministic [27].

Our emphasis on refinement continues the spirit of Mantel's work [28,18], but is different in its framework. Like Bossi *et al.* [29], Mantel takes an unwinding-style approach [21] in which the focus is on events and their possible occurrences, with relations between possible traces that express what can or cannot be deduced, about a trace that includes high-security events, based on projections of it from which the high-security events have been erased.

We focus instead primarily on variables' values, and our interest in the traces is not intrinsic but rather is induced temporarily by the refinement principles and their consequences (Section 3). While traces feature in our underlying model (Section 4.1), we abstract almost completely from that (Section 4.2) into a variable (the Shadow variable *H*) which we can then "blend in" to a semi-conventional program-logic formulation of correctness: thus our case-study example (Section 11) *does not refer to traces at all*. Nevertheless other authors' trace-style *abstract criteria* for refinement (*i.e.* preservation of properties), and ours, agree with each other and with the general literature of refinement and its issues [2–9,30].

WE CONCLUDE that *ignorance refinement* is able to handle examples of simple design, at least — even though their significance may be far from simple. Because *wp*-logic for ignorance retains most structural features of classical *wp*, we expect that loops and their invariants, divergence, and concurrency via *e.g. action systems* [31] could all be feasible extensions.

Adding probability via modal "expectation transformers" [30] is a longer-term goal, but will require a satisfactory treatment of conditional probabilities (the probabilistic version of *Shrink Shadow*) in that context.

Acknowledgements

Thanks to Yoram Moses, Kai Engelhardt and Ron van der Meyden for the problem, the right approach and the historical background, and to Anindya Banerjee, Annabelle McIver, Jeff Sanders, Mike Spivey, Susan Stepney, members of IFIP WG2.1 and delegates to Dagstuhl Seminars 06191 (organisers Abrial, Glässer) and 06281 (Broy, Cousot, Misra, O'Hearn) for suggestions.

Michael Clarkson and Chenyi Zhang supplied useful references. Yoram Moses contributed material to Section 4.

Finally I would like to thank the referees of this journal version, and of the earlier report [1], for their very thorough advice.

We acknowledge the support of the Australian Research Council Grant DP034557.

Appendix. Proofs of certain lemmas (Section 9)

Proof of Lemma 2. When *x* is visible v, say, we have for any Ψ that

$$\begin{array}{lll}
 & wp.([\mathrm{K}\phi];\ v{:\in}\,E).\Psi & \\
= & \mathrm{K}\phi \ \Rightarrow\ (\forall e\colon E \cdot [\Downarrow e{\in}E][v\backslash e]\Psi) & \text{“definitions”} \\
= & (\forall e\colon E \cdot \mathrm{K}\phi \Rightarrow [\Downarrow e{\in}E][v\backslash e]\Psi) & \text{“}e\text{ fresh”} \\
= & (\forall e\colon E \cdot \mathrm{K}\phi \Rightarrow [\Downarrow \phi][\Downarrow e{\in}E][v\backslash e]\Psi) & \text{“}\mathrm{K}\phi \Rightarrow (\mathrm{P}\psi \Leftrightarrow \mathrm{P}(\phi \wedge \psi))\text{”} \\
= & (\forall e\colon E \cdot \mathrm{K}\phi \Rightarrow [\Downarrow \phi \wedge e{\in}E][v\backslash e]\Psi) & \text{“merge }\textit{Shrink Shadows}\text{”} \\
= & (\forall e\colon E \cdot \mathrm{K}\phi \Rightarrow [\Downarrow \phi \wedge e{\in}F][v\backslash e]\Psi) & \text{“assumption”} \\
= & wp.([\mathrm{K}\phi];\ v{:=}\,F).\Psi\,. & \text{“symmetric argument”}
\end{array}$$

When *x* is hidden *h*, say, we have for any Ψ that

$$\begin{array}{lll}
 & wp.([\mathrm{K}\phi];\ h{:\in}\, E).\Psi & \\
= & \mathrm{K}\phi\ \Rightarrow\ (\forall h{:}\, E \cdot [h{\Leftarrow}E]\Psi) & \text{“definitions”} \\
= & (\forall h{:}\, E \cdot \mathrm{K}\phi \Rightarrow [h{\Leftarrow}E]\Psi) & \text{“}h\text{ not free in }\mathrm{K}\phi\text{”} \\
= & (\forall h{:}\, E \cdot \mathrm{K}\phi \Rightarrow [h{\Leftarrow}F]\Psi) & \text{“}\mathrm{K}\phi \Rightarrow (\mathrm{P}\psi \Leftrightarrow \mathrm{P}(\phi \wedge \psi))\text{”} \\
= & wp.([\mathrm{K}\phi];\ h{:\in}\, F).\Psi. & \text{“symmetric argument”}
\end{array}$$

Proof of Lemma 4. The proof is by structural induction over programs; we give the base cases here, using Ψ for a general (non-classical) ignorant postcondition formula. For **skip** we have

$$\begin{array}{lll}
 & wp.(\mathbf{skip};\ [\mathrm{K}\psi]).\Psi & \\
= & \mathrm{K}\psi \Rightarrow \Psi & \text{“(7)”} \\
\Rrightarrow & \mathrm{K}\phi \Rightarrow \Psi & \text{“assumption }\phi \Rightarrow \mathbf{skip}.\psi\text{”} \\
= & wp.([\mathrm{K}\phi];\ \mathbf{skip}).\Psi &
\end{array}$$

For *Choose visible* (of which *Assign to visible* is a special case), we have

$$\begin{array}{lll}
 & wp.(v{:\in}\, E;\ [\mathrm{K}\psi]).\Psi & \\
= & (\forall e{:}\, E \cdot [\Downarrow e{\in}E][v\backslash e](\mathrm{K}\psi \Rightarrow \Psi)) & \text{“}\textit{Choose visible}\text{; (7)”} \\
= & (\forall e{:}\, E \cdot \mathrm{K}(e{\in}E \Rightarrow [v\backslash e]\psi) \Rightarrow [\Downarrow e \in E][v\backslash e]\Psi) & \text{“Fig. 7”} \\
\Rrightarrow & (\forall e{:}\, E \cdot \mathrm{K}\phi \Rightarrow [\Downarrow e{\in}E][v\backslash e]\Psi) & \text{“assumption }\phi \Rightarrow wp.(v{:\in}\, E).\psi\text{”} \\
= & \mathrm{K}\phi \Rightarrow (\forall e{:}\, E \cdot [\Downarrow e{\in}E][v\backslash e]\Psi) & \text{“}e\text{ is fresh”} \\
= & wp.([\mathrm{K}\phi];\ v{:\in}\, E).\Psi. & \text{“}\textit{Choose visible}\text{; (7)”}
\end{array}$$

For *Choose hidden* we have

$$\begin{array}{lll}
 & wp.(h{:\in}\, E;\ [\mathrm{K}\psi]).\Psi & \\
= & (\forall h{:}\, E \cdot [h{\Leftarrow}E](\mathrm{K}\psi \Rightarrow \Psi)) & \text{“}\textit{Choose hidden}\text{; (7)”} \\
= & (\forall h{:}\, E \cdot \mathrm{K}(\forall h{:}\, E \cdot \psi) \Rightarrow [h{\Leftarrow}E]\Psi) & \text{“Fig. 7”} \\
\Rrightarrow & (\forall h{:}\, E \cdot \mathrm{K}\phi \Rightarrow [h{\Leftarrow}E]\Psi) & \text{“assumption }\phi \Rightarrow wp.(h{:\in}\, E).\psi\text{”} \\
= & \mathrm{K}\phi \Rightarrow (\forall h{:}\, E \cdot [h{\Leftarrow}E]\Psi) & \text{“}e\text{ is fresh”} \\
= & wp.([\mathrm{K}\phi];\ h{:\in}\, E).\Psi. & \text{“}\textit{Choose hidden}\text{; (7)”}
\end{array}$$

Proof of Lemma 5. The proof is by structural induction. Let $S(x)$ be a program possibly containing references to a free variable x. We show for all ignorant postconditions Φ that

$$\models\ wp^v_x.S(x).\Phi \Rightarrow wp^h_x.S(x).\Phi,$$

where wp^v_x treats x as visible and wp^h_x treats it as hidden, working through the cases of Fig. 8.

Identity Trivial.
Assign to visible/hidden Special case of **Choose visible/hidden**.
Choose visible If x occurs in E then $[\Downarrow e{\in}E]\mathrm{P}\Psi$ will differ in the two cases: when x is visible, it will be free in $[\Downarrow e{\in}E]\mathrm{P}\Psi$ for any $\mathrm{P}\Psi$ occurring (positively) within Φ; when it is hidden, however, it will be captured by the P-modality. Examination of the logical interpretation (Section 6) shows however that the former implies the latter, and this implication will propagate unchanged to the whole of Φ, since $\mathrm{P}\Psi$ occurs positively within it.
Choose hidden By similar reasoning to **Choose visible**, different treatment of $[h{\Leftarrow}E]$ can only result in an implication overall.
Choose x We must compare $(\forall e{:}\, E \cdot [\Downarrow e{\in}E]\,[x\backslash e]\,\Phi)$, for when x is visible, with $(\forall x{:}\, E \cdot [x{\Leftarrow}E]\,\Phi)$ for when x is hidden. Because both $[\Downarrow e{\in}E]$ and $[x{\Leftarrow}E]$ have no effect on the classical part of Φ, we can concentrate on the case $\mathrm{P}\Psi$ — thus we compare

$$(\forall e{:}\, E \cdot \cdots [\Downarrow e{\in}E]\,[x\backslash e]\,\mathrm{P}\Psi) \quad \text{and} \quad (\forall x{:}\, E \cdot \cdots [x{\Leftarrow}E]\,\mathrm{P}\Psi)$$

which, from Fig. 7, is in fact a comparison between

$$(\forall e{:}\, E \cdot \cdots \mathrm{P}(e{\in}E \wedge [x\backslash e]\,\Psi)) \quad \text{and} \quad \cdots \mathrm{P}(\exists e{:}\, E \cdot [x\backslash e]\,\Psi),$$

where the $(\forall x{:}\, E\ \cdot\ \cdots)$ is dropped in the *rhs* because the body contains no free x, and we have introduced the renaming to make it more like the *lhs*. Section 6 now shows, again, that the *lhs* implies the *rhs*.
Demonic choice Trivial (inductively).
Sequential composition Trivial (inductively).
Conditional This too is trivial, because **if** $E \cdots$ can be rewritten using a fresh visible as $|[\textsc{vis}\ b \cdot b{:=}E;\ \mathbf{if}\ b\ \cdots]|$ and it is then handled by the other cases.
Local variables Trivial since the substitution simply distributes through in both cases.

References

[1] C. Morgan, The Shadow Knows: Refinement of ignorance in sequential programs, in: T. Uustalu (Ed.), Proc. of 8th Int. Conf. on Mathematics of Program Construction, MPC 2006, Kuressaare, July 2006, in: Lect. Notes in Comput. Sci., vol. 4014, Springer, Berlin, 2006, pp. 359–378.

[2] N. Wirth, Program development by stepwise refinement, Commun. ACM 14 (4) (1971) 221–227.

[3] R.-J. Back, On the correctness of refinement steps in program development, Report A-1978-4, Dept. of Comput. Sci., Univ. of Helsinki, 1978.

[4] C.A.R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (10) (1969) 576–580. 583.

[5] E. Dijkstra, A Discipline of Programming, Prentice Hall, Englewood Cliffs, 1976.

[6] C. Morgan, Programming from Specifications, 2nd ed., in: Prentice Hall Int. Series in Comput. Sci., Prentice Hall, New York, 1994, URL: http://web.comlab.ox.ac.uk/oucl/publications/books/PfS/.

[7] R.-J. Back, J. von Wright, Refinement Calculus: A Systematic Introduction, Graduate Texts in Computer Science, Springer, New York, 1998.

[8] C. Morgan, T. Vickers (Eds.), On the Refinement Calculus, Formal Approaches to Computing and Information Technology, Springer, London, 1994.

[9] J. Jacob, Security specifications, in: Proc. of 1988 IEEE Symp. on Security and Privacy, S& P '88, Oakland, CA, 1988, IEEE CS Press, Los Alamitos, CA, 1988, pp. 14–23.

[10] R. Rivest, Unconditionally secure commitment and oblivious transfer schemes using private channels and a trusted initialiser, Techn. report, MIT, 1999, URL: http://theory.lcs.mit.edu/~rivest/Rivest-commitment.pdf.

[11] J. Halpern, K. O'Neill, Secrecy in multiagent systems, in: Proc. of 15th IEEE Computer Security Foundations Wksh, CSFW 2002 Cape Breton, NS, June 2002, IEEE CS Press, Los Alamitos, CA, 2002, pp. 32–46.

[12] J. Hintikka, Knowledge and Belief: An Introduction to the Logic of the Two Notions, Cornell University Press, Ithaca, NY, 1962, New edition prepared by V. F. Hendricks and J. Symons (King's College London Publications, London, 2005).

[13] J.Y. Halpern, Y. Moses, Knowledge and common knowledge in a distributed environment, J. ACM 37 (3) (1990) 549–587.

[14] R. Fagin, J. Halpern, Y. Moses, M. Vardi, Reasoning about Knowledge, MIT Press, Cambridge, MA, 1995.

[15] M. Smyth, Power domains, J. Comput. Syst. Sci. 16 (1978) 23–36.

[16] C. Morgan, Unpublished notes "060320 atomicity", 2006, URL: http://www.cse.unsw.edu.au/~carrollm/Notes/.

[17] D. Chaum, The dining cryptographers problem: Unconditional sender and recipient untraceability, J. Cryptol. 1 (1) (1988) 65–75.

[18] H. Mantel, Preserving information flow properties under refinement, in: Proc. of 2001 IEEE Symp. Security and Privacy, S& P 2001, Oakland, CA, May 2001, IEEE CS Press, Los Alamitos, CA, 2001, pp. 78–91.

[19] K. Engelhardt, Y. Moses, R. van der Meyden, Unpublished report, Univ. of New South Wales, Sydney, 2005.

[20] J. Halpern, K. O'Neill, Anonymity and information hiding in multiagent systems, in: Proc. of 16th IEEE Computer Security Foundations Wksh, CSFW 2003, Pacific Grove, CA, June/July 2003, IEEE CS Press, Los Alamitos, CA, 2003, pp. 75–88.

[21] J. Goguen, J. Meseguer, Unwinding and inference control, in: Proc. of 1984 IEEE Symp. on Security and Privacy, S& P '84, Oakland, CA, Apr. 1984, IEEE CS Press, Washington, CA, 1984, pp. 75–86.

[22] E. Cohen, Information transmission in sequential programs, ACM SIGOPS Operating Syst. Rev. 11 (5) (1977) 133–139.

[23] A. Sabelfeld, A. Myers, Language-based information-flow security, IEEE J. Sel. Areas Commun. 21 (1) (2003) 5–19.

[24] K. Leino, R. Joshi, A semantic approach to secure information flow, Sci. Comput. Programming 37 (1–3) (2000) 113–138.

[25] A. Sabelfeld, D. Sands, A PER model of secure information flow, Higher-Order and Symb. Comput. 14 (1) (2001) 59–91.

[26] A. Sabelfeld, A. Myers, A model for delimited information release, in: K. Futatsugi, F. Mizoguchi, N. Yonezaki (Eds.), Revised Papers from Mext-NSF-JSPS Int. Symp. on Software Security: Theories and Systems, 2003, Tokyo, Nov. 2003, in: Lect. Notes in Comput. Sci., vol. 3233, Springer, Berlin, 2004, pp. 174–191.

[27] A. Roscoe, J. Woodcock, L. Wulf, Non-interference through determinism, J. Comput. Secur. 4 (1) (1996) 27–54.

[28] H. Mantel, Possibilistic definitions of security—an assembly kit, in: Proc. of 13th IEEE Computer Security Foundations Wksh, CSFW 2000, Cambridge, July 2000, IEEE CS Press, Los Alamitos, CA, 2000, pp. 185–199.

[29] A. Bossi, R. Focardi, C. Piazza, S. Rossi, Refinement operators and information flow security, in: Proc. of 1st Int. Conf. on Software Engineering and Formal Methods, SEFM 2003, IEEE CS Press, Los Alamitos, CA, 2003, pp. 44–53.

[30] A. McIver, C. Morgan, Abstraction, Refinement and Proof for Probabilistic Systems, in: Technical Monographs in Computer Science, Springer, New York, 2005, URL: http://www.cse.unsw.edu.au/~carrollm/arp/.

[31] R.-J. Back, R. Kurki-Suonio, Decentralisation of process nets with centralised control, in: Proc. of 2nd Ann. ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing, PODC '83, Montreal, Aug. 1983, ACM Press, New York, 1983, pp. 131–142.