The Shadow Knows:

Refinement of ignorance in sequential programs

UNSW-CSE-TR-0522

Carroll Morgan[1]
School of Computer Science and Engineering,
University of New South Wales
carrollm@cse.unsw.edu.au

30 November 2005

[1]Supported as an Australian Professorial Fellow by the Australian Research Council

Abstract

Sequential-program state can be separated into "visible" and "hidden" parts [4] in order to allow knowledge-based reasoning [8] about the hidden values. *Ignorance-preserving refinement* should ensure that observing the visible part of an *implementation* can reveal no more about the hidden part than could be revealed by its *specification*.

Possible applications are zero-knowledge protocols, or security contexts where the "high-security" state is considered hidden and the "low-security" state is considered visible.

Rather than checking for ignorance preservation at each stage, we suggest program-algebraic "refinement rules" that *preserve ignorance by construction* [7, 15].

The Dining Cryptographers (DC) [3] is a motivating example, in which ignorance of certain variables (coins) is intended to contribute to an ignorance property of the overall protocol. Our algebra is powerful enough to derive DC, while retaining soundness by avoiding (*e.g.*) the *Refinement Paradox* [13].

We formulate and justify general principles about which refinement rules should be retained, for algebraic power and utility, and which should be discarded, for soundness.

1 Introduction

Traditionally, rigorous sequential-program development is based on an operational model of state-to-state relations, a program logic of Hoare-triples $\{\Phi\}\ P\ \{\Psi\}$ [12] or weakest preconditions [5], built above the model and consistent with it, and finally a "refinement" algebra [2, 18] of (in-)equalities between programs, in turn consistent with both the model and the logic. A specification S is *refined by* an implementation I, written $S \sqsubseteq I$, just when I preserves all logically-expressible properties of S.

Ignorance is (for us) what observer's does not know about the parts of the program state he cannot see directly. We partition the state into a "visible" part v which the observer can see and a "hidden" part h which he cannot, and we consider a known program operating over v, h: from the final value of v, what can he deduce about the final value of h? If the program is $v{:=}\,0$, what he knows after about h is just what he knew before; if it is $v{:=}\,h \bmod 2$, he has learned h's parity; and if it is $v{:=}\,h$ he has learned h's value exactly.

Traditional refinement does not preserve ignorance. Assume v, h to have type T: the program "choose v from T" is refinable into "set v to h" — it is simply a reduction of demonic nondeterminism. That refinement $v{:\in}\,T \sqsubseteq v{:=}\,h$ is called the "Refinement Paradox" [13] because it does not preserve ignorance: program $v{:\in}\,T$ tells us nothing about h, whereas $v{:=}\,h$ tells us everything.

Although we want to employ traditional refinement because it is familiar, we cannot use it "as is" for ignorance-preservation because it would be unsound.

Our first contribution is to propose the following principles for excluding from the algebra just those refinements inconsistent with ignorance-preservation:

Pr1 *Retain all "v-only" refinements* — It would be impractical to search an entire program for hidden variables in order to validate local "visible-only" reasoning in which the hiddens are not mentioned.

Pr2 *Retain all "structural" refinements* — 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 being moved around: they are valid *en bloc*. It would be impractical to have to check through the fragments' interiors (including *e.g.* procedure calls) to validate such familiar structural rearrangements.

Pr3 *Examine each "explicit-h" refinement on its merits* — Those that preserve ignorance will be retained; the others (*e.g.* the Paradox) will be excluded.

Our main contribution is then to extend the model and logic of sequential programming (only slightly) to realise the above principles: Principles Pr1,Pr2 will hold unconditionally; and, for Pr3, putative explicit-h refinements can be checked individually. Sec. 2.3 illustrates exclusion; Sec. 7 illustrates retention.

A potential application is zero-knowledge- or security-sensitive protocols, where ignorance is of course of great importance. Our final contribution is to treat the Dining Cryptographer's protocol (DC) in that respect (Sec. 8).

2 Realising the refinement-algebra of ignorance

The great advantage of having our goals expressed algebraically it that we can conduct early (and intellectually inexpensive) *gedanken* experiments that inform the later model construction. *Does program* $v{:=}\,h; v{:\in}\,T$ *reveal* h*?* Yes it does, because $v{:\in}\,T \sqsubseteq \text{SKIP}$ (Pr1); sequential composition ";" is $\sqsubseteq$-monotonic (Pr2); and SKIP is the identity (Pr2). Thus

$$v{:=}\,h; v{:\in}\,T \quad \sqsubseteq \quad v{:=}\,h; \text{SKIP} \quad = \quad v{:=}\,h \ ,$$

and the implementation (*rhs*) fails to conceal h: so the specification must have failed also. Hence our model must have "perfect recall" [10], because escape (*lhs*) of h in $v{:=}\,h$ is not "erased" by the overwriting $v{:\in}\,T$ of v. That, as well as compelling advice from Moses and Engelhardt, suggests the Logic of Knowledge.

The standard model for knowledge-based reasoning [8] is based on possible "runs" of a system and participating agents' ignorance of how they have interleaved; 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: informally, we think of this as an observer of the system, 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 (its dual) knowledge, and *decrease of ignorance* is sufficient to exclude an otherwise acceptable refinement.

2.1 The model as a Kripke structure

We are given a sequential program text, including a notion of atomicity: unless stated otherwise, each *syntactically* atomic element of the program changes the program counter when it is executed. 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 now we suppose we have just two (untyped) 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 $\overline{v}, \overline{h}$, together with a history-sequence $\overline{p}$ of the program counter; the observer can see $\overline{v}, \overline{p}$ but not $\overline{h}$. From $\overline{p}$, after $\mathrm{S1}; (\mathrm{S2} \sqcap \mathrm{S3}); \mathrm{S4}$ we can "remember" for example which of S2 or S3 was executed earlier.

The possible runs of a system S comprise all sequences of global states that could be produced by the successive execution of atomic steps from some initial v_0, h_0, including all outcomes resulting from demonic choice (both $\sqcap$ and $:\in$).

If the current state is $(\overline{v}, \overline{h}, \overline{p})$, then the set of *possible* states is the set of triples $(\overline{v}, \overline{h}_1, \overline{p})$ that S can produce 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, which depends on S, v_0, h_0.

Thus from $\overline{p}$ the observer knows the execution trace; from $\overline{v}$ he knows the successive v values; but of hiddens he knows only h_0 directly. Fig. 1 illustrates this viewpoint with some small examples: in Case 1.1 we are indifferent to the two forms of choice because h does not occur in them (Pr1); however Cases 1.(2,3) contain h explicitly, and the traditional rules don't necessarily apply (Pr3).

2.2 A logic of knowledge and ignorance

Our logical language is first-order predicate formulae Φ, interpreted conventionally over the variables of the program, augmented with a "knows" modal operator so that $\mathrm{K}\Phi$ holds in this state just when Φ itself holds in *all* states accessible via $\sim$ from this one, and its dual $\mathrm{P}\Phi$ defined $\neg\mathrm{K}(\neg\Phi)$ which holds in this state just when Φ itself holds in *some* state accessible from this one. Occurrences of variable names in Φ give their current values

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 observer (so we write "we know" *etc.*)

	Program	Informal commentary
1.1	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 is still h_0, though we cannot see it.
1.2	$h{:\in}\{0,1\}$	We know that h is either 0 or 1, but we don't know which; we see that v is v_0.
1.3	(two atomic statements) $h{:=}\,0 \sqcap h{:=}\,1$	We know the value of h, because from the program-counter history we know which of the atomic $h{:=}\,0$ or $h{:=}\,1$ was executed.
1.4	$h{:\in}\{0,1\}$; $v{:=}\,0 \sqcap v{:=}\,1$	We don't know whether h is 0 or it is 1: even the $\sqcap$-demon cannot see the hidden variable.
1.5	$h{:\in}\{0,1\}$; $v{:\in}\{h, 1{-}h\}$	Though the choice of v refers to h it reveals no information, since the statement is atomic.
1.6	$h{:\in}\{0,1\}$; $v{:=}\,h \sqcap v{:=}\,1{-}h$	Here h is revealed, because we know which of the two atomic assignments to v was executed.
1.7	$h{:\in}\{0,1,2,3\}$; $v{:=}\,h$	We see v; we deduce h since we can see $v{:=}\,h$ in the program text.
1.8	$h{:\in}\{0,1,2,3\}$; $v{:=}\,h \bmod 2$	We see v; from that either we deduce h is 0 or 2, or that h is 1 or 3.
1.9	$h{:\in}\{0,1,2,3\}$; $v{:=}\,h \bmod 2$; $v{:=}\,0$	We see v is 0; but our deductions about h are as for 1.8, because we saw v's earlier value.

Figure 1: Examples of ignorance, informally interpreted

so that variable v (resp. h) is bound to $\mathsf{last}.\overline{v}$ (resp. $\mathsf{last}.\overline{h}$). Earlier values in $\overline{v}, \overline{h}$ are not available directly, nor is $\overline{p}$; however they indirectly influence the modalities via their effect on $\sim$. Fig. 2 illustrates the logic with our earlier examples in Fig. 1.

2.3 Refinement, and the paradox

Traditional refinement $\sqsubseteq$ between programs allows the reduction of demonic nondeterminism, as in $v{:\in}\{0,1\} \sqsubseteq v{:=}\,0$.[2] It is a partial order over the program lattice [2] and, as such, satisfies $\mathrm{S1} \sqcap \mathrm{S2} \sqsubseteq \mathrm{S1}$ in general; and it is induced by the chosen program logic, so that $\mathrm{S1} \sqsubseteq \mathrm{S2}$ just when all expressible properties of S1 are preserved in S2.

Our expressible properties will be traditional Hoare-style triples (equivalently Dijkstra-style weakest preconditions) over formulae whose truth is preserved by increase of ignorance: all modalities K occur negatively; all modalities P occur positively. We say that such occurrences of modalities are *ignorant*; and a formula is said to be ignorant just when all its modalities are.

Definition 0.1 *A triple* $\{\Phi\}\ \mathrm{S}\ \{\psi\}$ *is* valid *just when any system* S1 *for which* Φ *is a valid conclusion (as in Fig. 2) if extended to a system* $\mathrm{S1;S}$ *becomes a system for which* ψ *is a valid conclusion.*

The Refinement Paradox [13] is an issue because traditional refinement allows the "secure" $v{:\in}\,T$ to be refined to the "insecure" $v{:=}\,h$ as an instance of reduction of demonic nondeterminism. The extended logic avoids the paradox: in particular, in Sec. 6 we show that the property $\{\mathrm{P}(h{=}c)\}\ v{:\in}\,T\ \{\mathrm{P}(h{=}c)\}$ is valid, but that property $\{\mathrm{P}(h{=}c)\}\ v{:=}\,h\ \{\mathrm{P}(h{=}c)\}$ is not valid. An operational argument is given there also.

3 The $(\mathsf{v}, \mathsf{h}, \mathsf{H})$ interpretation of the logic

Our logical language (Sec. 2.2) is first-order augmented with a modal operator so that $\mathrm{K}\Phi$ is read "certainly Φ" [8, 3.7.2]. We give the language function- (including constant-) and relation symbols as needed, among which we distinguish the (program-variable) symbols *visibles* in V and *hiddens* in H; as well there are the usual (logical) variables in L over which we allow $\forall, \exists$ quantification. The visibles, hiddens and variables are collectively the *scalars* $\mathrm{X} \mathrel{\widehat{=}} \mathrm{V} \cup \mathrm{H} \cup \mathrm{L}$.

[2] It also allows elimination of divergence, which we do not treat here.

In each case we interpret the formula after execution of the program from initial values v_0, h_0. "Valid conclusion" means *true in all final states* and "Invalid conclusion" means *false in some final state*.

	Program	Valid conclusion	Invalid conclusion
2.1	both $v{:\in}\{0,1\}$ and $v{:=}\,0 \sqcap v{:=}\,1$	$v \in \{0,1\}$	$v = 0$
2.2	$h{:\in}\{0,1\}$	P(h=0)	K(h=0)
2.3	$h{:=}\,0 \sqcap h{:=}\,1$	$h \in \{0,1\}$	P(h=0)
2.4	$h{:\in}\{0,1\}$; $v{:=}\,0 \sqcap v{:=}\,1$	P(v=h)	K(v≠h)
2.5	$h{:\in}\{0,1\}$; $v{:\in}\{h, 1{-}h\}$	P(h=0) In fact Program 2.5 equals Program 2.4.	P(v=0)
2.6	$h{:\in}\{0,1\}$; $v{:=}\,h \sqcap v{:=}\,1{-}h$	$v \in \{0,1\}$ But Program 2.6 differs from Program 2.5.	P(h=0)
2.7	$h{:\in}\{0,1,2,3\}$; $v{:=}\,h$	K(v=h)	P(v≠h)
2.8	$h{:\in}\{0,1,2,3\}$; $v{:=}\,h \bmod 2$	v=0 ⇒ P($h{\in}\{2,4\}$)	P(h=1) ∧ P(h=2)
2.9	$h{:\in}\{0,1,2,3\}$; $v{:=}\,h \bmod 2$; $v{:=}\,0$	P($h{\in}\{1,2\}$) The $v{:=}\,0$ is an unsuccessful "cover up".	v=0 ⇒ P($h{\in}\{2,4\}$)

- In 2.3 the invalidity is because $\sqcap$ might resolve to the right: then h=0 is impossible.
- In 2.6 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 is impossible.
- In 2.8 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 $v{:=}\,h \bmod 2$ leaves us in no doubt about h's parity; we cannot simultaneously consider both 1 and 2 to be possible.
- In 2.9 the invalidity is because the *final* value of v does not indicate h's parity.

Figure 2: Examples of ignorance logic, informally interpreted

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,hiddens respectively; we write their types $\mathrm{V}\!\twoheadrightarrow\!\mathsf{D}$ and $\mathrm{H}\!\twoheadrightarrow\!\mathsf{D}$.

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

A *state* $(\mathsf{v}, \mathsf{h}, \mathsf{H})$ comprises a visible- v, hidden- h and *shadow*- part H; the last, in $\mathbb{P}.(\mathcal{H}\!\twoheadrightarrow\!\mathsf{D})$, is a set of valuations over hiddens only. We require that $\mathsf{h} \in \mathsf{H}$.[3]

We define truth of Φ at $(\mathsf{v}, \mathsf{h}, \mathsf{H})$ under valuation w by induction, writing $(\mathsf{v}, \mathsf{h}, \mathsf{H}), \mathsf{w} \models \Phi$. Let t be the term-valuation built inductively from the valuation $\mathsf{v} \triangleleft \mathsf{h} \triangleleft \mathsf{w}$. Then we have the following [*op. cit.*, 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 $(\mathsf{t}.\mathrm{T}_1, \cdots, \mathsf{t}.\mathrm{T}_k)$ 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 $(\mathsf{v}, \mathsf{h}, \mathsf{H}), \mathsf{w} \triangleleft \langle \mathrm{L} \mapsto \mathsf{d} \rangle \models \Phi$ 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 $(\mathsf{v}, \mathsf{h}, \mathsf{H}) \models$ when w is empty, and $\models \Phi$ when $(\mathsf{v}, \mathsf{h}, \mathsf{H}) \models \Phi$ for all $\mathsf{v}, \mathsf{h}, \mathsf{H}$ 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 have $\models \Phi \Rightarrow \mathrm{P}\Phi$.

4 The "reduced" $(\mathsf{v}, \mathsf{h}, \mathsf{H})$ operational model

In Sec. 2.1 programs took initial states v_0, h_0 to sets of run-triples $(\overline{v}, \overline{h}, \overline{p})$, and the program text induced an equivalence relation $\sim$ over them. In Sec. 3 we interpreted a modal logic over $(\mathsf{v}, \mathsf{h}, \mathsf{H})$ triples.

[3] Our state corresponds to Fagin's *Kripke structure and state* together [*loc. cit.*]; but our use of Kripke structures is extremely limited. Not only do we make the Common-Domain Assumption, but we do not allow the structure to vary between worlds except for the interpretation h of hiddens.

The initial state is $(\mathsf{v}_0, \mathsf{h}_0, \{\mathsf{h}_0\})$.

	Program	Final states in the "reduced" $(\mathsf{v}, \mathsf{h}, \mathsf{H})$ model	
3.1	both $v{:\in}\{0,1\}$ and $v{:=}0 \sqcap v{:=}1$	$(0, h_0, \{h_0\})$, $(1, h_0, \{h_0\})$	
3.2	$h{:\in}\{0,1\}$	$(v_0, 0, \{0,1\})$, $(v_0, 1, \{0,1\})$	
3.3	$h{:=}0 \sqcap h{:=}1$	$(v_0, 0, \{0\})$, $(v_0, 1, \{1\})$	
3.4	$h{:\in}\{0,1\}$; $v{:=}0 \sqcap v{:=}1$	$(0,0,\{0,1\})$, $(0,1,\{0,1\})$, $(1,0,\{0,1\})$, $(1,1,\{0,1\})$	
3.5	$h{:\in}\{0,1\}$; $v{:\in}\{h, 1{-}h\}$	$(0,0,\{0,1\})$, $(1,0,\{0,1\})$, $(0,1,\{0,1\})$, $(1,1,\{0,1\})$	Thus this and 3.4 are equal.
3.6	$h{:\in}\{0,1\}$; $v{:=}h \sqcap v{:=}1{-}h$	$(0,0,\{0\})$, $(1,0,\{0\})$, $(0,1,\{1\})$, $(1,1,\{1\})$	But this one differs.
3.7	$h{:\in}\{0,1,2,3\}$; $v{:=}h$	$(0,0,\{0\})$, $(1,1,\{1\})$, $(2,2,\{2\})$, $(3,3,\{3\})$	
3.8	$h{:\in}\{0,1,2,3\}$; $v{:=}h \bmod 2$	$(0,0,\{0,2\})$, $(1,1,\{1,3\})$, $(0,2,\{0,2\})$, $(1,3,\{1,3\})$	
3.9	$h{:\in}\{0,1,2,3\}$; $v{:=}h \bmod 2$; $v{:=}0$	$(0,0,\{0,2\})$, $(0,1,\{1,3\})$, $(0,2,\{0,2\})$, $(0,3,\{1,3\})$ The final $v{:=}0$ does not affect H.	

In (3.9) partial information about h remains, represented by two possibilities for H of $\{0,2\}$ and $\{1,3\}$, even though $v{=}0$ in all outcomes.

Figure 3: Examples (Figs. 1,2) revisited: a relational interpretation

The two are brought together by giving a "reduced" relational operational model for programs, over such triples. That is, we augment the usual relational state (v, h) with the shadow H; but we avoid the full detail contained in Sec. 2's run-sequences (whose virtue however was the comparison with the standard knowledge framework [8]).

Thus programs take *input* reduced states to *output sets* of reduced states, as we illustrate in Fig. 3 whose final states are such that the valid conclusions (Fig. 2) are true (Sec. 3) over the whole output set and the invalid conclusions are false for at least one output element.

The correspondence between the two operational models is summarised in the following theorem:

Theorem 1 The run-sequence- (Sec. 2.1) and reduced- (this section) operational models correspond via the abstraction

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

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

Distribution through P however is affected by that modality's implicitly quantifying over hidden variables: if e is a hidden variable, then $[e\backslash E]\mathrm{P}\Phi$ is just Φ; and if E contains hidden variables, the substitution does not distribute into $\mathrm{P}\Phi$ 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 $[\Downarrow E]\mathrm{P}\Phi \mathrel{\widehat{=}} \mathrm{P}(E \wedge \Phi)$; hidden variables in E are *not* renamed.

Set hidden	$[h{\leftarrow}E]$	Distributes through all operators, including P, with renaming as necessary for $\forall, \exists$ (but not P).
Set shadow	$[h{\Leftarrow}E]$	Distributes through all classical operators, with renaming; has no effect on classical atomic formulae.

For modal formulae we have $[h{\Leftarrow}E]\mathrm{P}\Phi \mathrel{\widehat{=}} \mathrm{P}(\exists h' : E \cdot [h\backslash h']\Phi)$.

Figure 4: Technical predicate transformers

Proof: The straightforward proof [17] shows that the abstraction is possible because programs cannot refer to the full run-sequences directly; what they *can* refer to —the current values of v, h— is captured in the abstraction. The shadow H is used by the modal-logic semantics to come (Sec. 5).[5] □

5 Weakest-precondition modal semantics

For practical reasoning, we introduce a weakest-precondition *logical* semantics to support the assertional style of Def. 0.1. It corresponds to the operational semantics of Sec. 4 (and hence via Thm. 1 to the original sequence-semantics also), given the interpretation in Sec. 3 of the modal formulae.

The logical semantics is given in two layers, in Fig. 4 and Fig. 5, because most programs generate *both* a modal- and a classical-style transformer, which distribute differently through postconditions.

Visible and hidden variables have separate declarations VIS v and HID h respectively. Declarations within a local scope do not affect visibility: a global hidden variable cannot be seen by the observer; a local visible variable can.

[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 $\mathsf{last}.\overline{h}'$".

[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.

Logical variable e is fresh.

Identity	$wp.\text{SKIP}.\Psi$	$\widehat{=}$	Ψ
Assign to visible	$wp.(v{:=}E).\Psi$	$\widehat{=}$	$[e\backslash E]\,[\Downarrow e{=}E]\,[v\backslash e]\,\Psi$
Choose visible	$wp.(v{:\in}E).\Psi$	$\widehat{=}$	$(\forall e{:}\,E \cdot [\Downarrow e{\in}E]\,[v\backslash e]\,\Psi)$
Assign to hidden	$wp.(h{:=}E).\Psi$	$\widehat{=}$	$[h{\leftarrow}E]\,\Psi$
Choose hidden	$wp.(h{:\in}E).\Psi$	$\widehat{=}$	$(\forall e{:}\,E \cdot [h\backslash e]\,[h{\Leftarrow}E]\,\Psi)$
Demonic choice	$wp.(\text{S1} \sqcap \text{S2}).\Psi$	$\widehat{=}$	$wp.\text{S1}.\Psi \;\wedge\; wp.\text{S2}.\Psi$
Composition	$wp.(\text{S1};\text{S2}).\Psi$	$\widehat{=}$	$wp.\text{S1}.(wp.\text{S2}.\Psi)$

Conditional

$$wp.(\textbf{if } E \textbf{ then } \text{S1} \textbf{ else } \text{S2} \textbf{ fi}).\Psi \;\widehat{=}\; E \Rightarrow [\Downarrow E]\,wp.\text{S1}.\Psi \;\wedge\; \neg E \Rightarrow [\Downarrow \neg E]\,wp.\text{S2}.\Psi$$

Declare visible	$wp.(\text{VIS } v).\Psi$	$\widehat{=}$	$(\forall e \cdot [v\backslash e]\,\Psi)$	} Note that both these substitutions propagate within modalities in Ψ.
Declare hidden	$wp.(\text{HID } h).\Psi$	$\widehat{=}$	$(\forall e \cdot [h{\leftarrow}e]\,\Psi)$	

The *wp*-definitions apply whether the postcondition Ψ is ignorant or not.

Figure 5: Weakest-precondition modal semantics

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 assume *wlog* that modalities are not nested, since we can remove nestings via $\models \text{P}\Phi \equiv (\exists c \cdot [h\backslash c]\Phi \wedge \text{P}(h{=}c))$.

The congruence of the logical- and operational semantics justifies the connection between weakest preconditions and assertions.

Theorem 2 For all formulae Φ, Ψ, we have that

$\models \Phi \Rightarrow wp.\mathrm{S}.\Psi$ (as in Figs. 4,5) iff $\{\Phi\}\ \mathrm{S}\ \{\Psi\}$ (as in Def. 0.1).

Proof: The full proof [17] relies on a syntactic translation of v, h program fragments (Fig. 5) into v, h, H fragments, the operational semantics (which we have here only sketched), and a corresponding translation of modal formulae into ordinary first-order formulae, in both cases introducing an explicit H. In effect our language and logic are both regarded as syntactic sugar for a more basic form. For example (recall Example 2.8),

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

Then the normal *wp*-semantics [5] is used over the explicit v, h, H program fragments, and the resulting preconditions are translated back from the pure first-order $(\exists h{:}\, H \cdots)$-form into the modal P-form. □

The *wp*-logic has the following significant features:

1. All visible-variable-only program refinements (hence equalities) are preserved (Pr1).
2. All refinements relying only on Demonic choice, Composition, Identity ("structural") are preserved (Pr2).
3. The transformers defined in Fig. 5 distribute conjunction, as standard transformers do [5]. Thus complicated postconditions can be treated piecewise.
4. Non-modal postconditions can be treated using traditional semantics [5, 12], even if the program contains hidden variables.
5. Because of (3,4) the use of the modal semantics can be restricted to only the modal conjuncts of a postcondition.

From (4) we never *add* refinements. An example of (5) occurs in the Dining Cryptographers derivation (Fig. 6).

6 Avoiding the Refinement Paradox

In this section we see an example of a refinement's being excluded (Pr3).

Operationally, for programs S, S' we have $\mathrm{S} \sqsubseteq \mathrm{S}'$ just when for some initial (v_0, h_0, H_0) every possible outcome (v', h', H') of S' has $v = v' \wedge h = h' \wedge H \subseteq H'$ for some outcome (v, h, H) of S.[6] Thus, recalling Fig. 3, we have

[6] This is the *Smyth* powerdomain-order over an underlying refinement on single triples that allows the H-component —*i.e.* ignorance— to increase [22].

postcondition $p \Rightarrow \langle\!\langle p_1\!:\mathbb{B} \rangle\!\rangle$

through $wp.(p\!:=p_0 \oplus p_1 \oplus p_2)$ gives "Assign visible"

$[e\backslash p_0 \oplus p_1 \oplus p_2]\ [\Downarrow e = p_0 \oplus p_1 \oplus p_2]\ [p\backslash e]\ (p \Rightarrow \langle\!\langle p_1\!:\mathbb{B} \rangle\!\rangle)$

$\equiv$ $[e\backslash p_0 \oplus p_1 \oplus p_2]\ [\Downarrow e = p_0 \oplus p_1 \oplus p_2]\ (e \Rightarrow \langle\!\langle p_1\!:\mathbb{B} \rangle\!\rangle)$ "substitute"

$\equiv$ $[e\backslash p_0 \oplus p_1 \oplus p_2]\ [\Downarrow e = p_0 \oplus p_1 \oplus p_2]\ (e \Rightarrow (\forall b\!:\mathbb{B} \cdot \mathrm{P}(p_1{=}b)))$ "expand $\langle\!\langle\cdot\rangle\!\rangle$"

$\equiv$ $[e\backslash p_0 \oplus p_1 \oplus p_2]\ (e \Rightarrow (\forall b\!:\mathbb{B} \cdot \mathrm{P}(e = p_0 \oplus p_1 \oplus p_2\ \wedge\ p_1{=}b)))$ "Shrink shadow"

$\equiv$ $[e\backslash p_0 \oplus p_1 \oplus p_2]\ (e \Rightarrow (\forall b\!:\mathbb{B} \cdot \mathrm{P}((p_0 \oplus p_1 \oplus p_2)\ \wedge\ p_1{=}b)))$ "e in antecedent"

$\Leftarrow$ $\mathrm{P}((p_0{=}p_2) \wedge p_1)\ \wedge\ \mathrm{P}((p_0 \oplus p_2) \wedge \neg p_1)$ "drop antecedent; instantiate b"

$\Leftarrow$ $\mathrm{P}(\neg p_0 \wedge \neg p_2 \wedge p_1)\ \wedge\ \mathrm{P}(\neg p_0 \wedge p_2 \wedge \neg p_1)$ "P is $\Rightarrow$-monotonic"

$\Leftarrow$ $\langle\!\langle p_0, p_1, p_2\!:\mathbb{B} \mid \sum_i p_i \leq 1 \rangle\!\rangle$. "abbreviation"

Because *wp* is conjunctive (Sec. 5) we can deal with postcondition conjuncts separately; the standard part is obvious from ordinary *wp*; and by symmetry we can concentrate *wlog* on the p_1 case for the remainder.

Figure 6: Adequacy of the cryptographers' specification (Sec. 8)

for example (3.3) $\sqsubseteq$ (3.2), (3.6) $\sqsubseteq$ (3.5) and $((3.7); v\!:=0) \sqsubseteq$ (3.9). Apropos the paradox we see that $v\!:\in T \not\sqsubseteq v\!:=h$ because the former's final states are $\{e\!:T \cdot (e, h_0, H_0)\}$ whereas the latter's are just $\{\ (h_0, h_0, \{h_0\})\ \}$ and, even supposing $h_0 \in E$, still in general $H_0 \not\subseteq \{h_0\}$.

Fig. 7 shows how *wp*-logic avoids the Refinement Paradox.

7 The Encryption Lemma

In this section we see an example of a refinement's being retained (Pr3).

When a hidden secret is encrypted with a hidden key and published as a visible message, the intention is that observers 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 *DC* (Dining Cryptographers') protocol, where a secret (whether some cryptographer paid) is encrypted (via exclusive-or) with a key (a hidden Boolean coin) and becomes a message (is announced aloud).

We examine this simple situation in the ignorance logic; the resulting formalisation will provide one of the key steps in the *DC* derivation of Sec. 8.

Lemma 2.1 Let $s\!:S$ be a secret, $k\!:K$ a key, and $\oslash$ an encryption method so that $s \oslash k$ is the encryption of s. In a context HID s we have the refinement

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

	$wp.(v{:\in}\,E).(\mathrm{P}(h{=}c))$		$wp.(v{:=}\,h).(\mathrm{P}(h{=}c))$
$\equiv$	"Choose visible"	$\equiv$	"Assign to visible"
	$(\forall e{:}\,E \cdot [\Downarrow e{\in}E]\,[v\backslash e]\,\mathrm{P}(h{=}c))$		$[e\backslash h]\,[\Downarrow e{=}h]\,[v\backslash e]\,\mathrm{P}(h{=}c)$
$\equiv$	"v not free"	$\equiv$	"v not free; Shrink shadow"
	$(\forall e{:}\,E \cdot [\Downarrow e{\in}E]\,\mathrm{P}(h{=}c))$		$[e\backslash h]\,\mathrm{P}(e{=}h \wedge h{=}c)$
$\equiv$	"Shrink shadow"	$\equiv$	$[e\backslash h]\,\mathrm{P}(e{=}c \wedge h{=}c)$ "$h{=}c$"
	$(\forall e{:}\,E \cdot \mathrm{P}(e{\in}E \wedge h{=}c)))$	$\equiv$	"h not free in $e{=}c$"
$\equiv$	"h not free in $e{\in}E$"		$[e\backslash h]\,(e{=}c \wedge \mathrm{P}(h{=}c))$
	$(\forall e{:}\,E \cdot e{\in}E \wedge \mathrm{P}(h{=}c))$	$\equiv$	$h{=}c \wedge \mathrm{P}(h{=}c)$ "e not free"
$\equiv$	$\mathrm{P}(h{=}c)$. "e not free in $\mathrm{P}(h{=}c)$"	$\equiv$	$h{=}c$. "$\models \Phi \Rightarrow \mathrm{P}\Phi$"

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

The right-hand side shows that $h{=}c$ is the weakest 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 E contains no h, the fragment $v{:\in}\,E$ can be replaced by $v{:=}\,h$ only if we know h already.

Figure 7: Avoiding the Refinement Paradox, seen logically

which expresses that publishing the encryption as a message m reveals nothing about the secret s, provided the *Key-Complete Condition* (1) of Fig. 8 (KCC) is satisfied and the key k is not revealed.[7]

Proof: The calculation is given in Fig. 8. Informally we note that the KCC tells us that for every message $s \oslash k$ revealed in m, every *potential* value s' of s has a possible (different) key $k'{:}\,K$ that would produce the same message. Since all values of $k'{:}\,K$ are possible, all the potential values s' for s are also (still) possible. □

8 Deriving the Dining Cryptographers' Protocol

The Dining Cryptographers Protocol (DC) is an example of ignorance preservation [3]. In the original formulation, three cryptographers have finished their meal, and ask the waiter for the bill: he says it has already been paid; they know that the payer is either one of them or is the *NSA*. They devise a protocol to decide which — without however revealing the payer in the former case .

Each two cryptographers flip a Boolean coin, hidden from the third cryptographer; and each announces the exclusive-or $\oplus$ of the two coins he sees

[7] The Key-Complete Condition is very strong, requiring as many keys as messages; yet it applies to the DC protocol, where both are just one bit.

postcondition $s{=}B \ \wedge \ \mathrm{P}(s{=}C)$

through $wp.(m{:=}\, s \oslash k)$ gives "Assign visible"

$\quad [e\backslash s \oslash k]\ [\Downarrow e = s \oslash k]\ [m\backslash e]\ (s{=}B \ \wedge \ \mathrm{P}(s{=}C))$

$\equiv \quad s{=}B \ \wedge \ [e\backslash s \oslash k]\ \mathrm{P}(e = s \oslash k \ \wedge \ s{=}C)$ "m not free; Shrink shadow"

$\equiv \quad s{=}B \ \wedge \ [e\backslash s \oslash k]\ \mathrm{P}(e = C \oslash k \ \wedge \ s{=}C)$ "$s{=}C$"

through $wp.(k{:\in}\, K)$ gives "Choose hidden"

$\quad (\forall e{:}\, K \cdot [k\backslash e]\ [k{\Leftarrow}K]\ (s{=}B \ \wedge \ [e\backslash s \oslash k]\ \mathrm{P}(e = C \oslash k \ \wedge \ s{=}C)))$

$\equiv \quad s{=}B \ \wedge \ (\forall e{:}\, K \cdot [k{\Leftarrow}K]\ [k\backslash e]\ [e\backslash s \oslash k]\ \mathrm{P}(e = C \oslash k \ \wedge \ s{=}C))$ "distribute"

$\equiv \quad s{=}B \ \wedge \ (\forall e{:}\, K \cdot [k{\Leftarrow}K]\ [e\backslash s \oslash e]\ \mathrm{P}(e = C \oslash k \ \wedge \ s{=}C))$ "k hidden"

$\equiv \quad s{=}B \ \wedge \ (\forall e{:}\, K \cdot [e\backslash s \oslash e]\ [k{\Leftarrow}K]\ \mathrm{P}(e = C \oslash k \ \wedge \ s{=}C))$ "swap"

$\equiv \quad s{=}B \ \wedge \ (\forall e{:}\, K \cdot [e\backslash s \oslash e]\ \mathrm{P}(\exists k'{:}\, K \cdot e = C \oslash k' \ \wedge \ s{=}C))$ "Set shadow"

$\equiv \quad s{=}B \ \wedge \ (\forall e{:}\, K \cdot [e\backslash s \oslash e]\ (\exists k'{:}\, K \cdot e = C \oslash k') \ \wedge \ \mathrm{P}(s{=}C))$ "distribute"

$\equiv \quad s{=}B \ \wedge \ \mathrm{P}(s{=}C) \ \wedge \ (\forall e{:}\, K \cdot (\exists k'{:}\, K \cdot s \oslash e = C \oslash k'))$ "distribute"

We use a subsidiary lemma [17] that SKIP $\sqsubseteq$ S provided S does not change v or the actual or possible values for h: *i.e.* it is sufficient that for all A, B, C we have

$$\{v{=}A \wedge h{=}B \wedge \mathrm{P}(h{=}C)\} \quad \mathrm{S} \quad \{v{=}A \wedge h{=}B \wedge \mathrm{P}(h{=}C)\}\ ,$$

where v, h are the (vectors of) all variables in context.
In Lem. 2.1 the context is HID s (and no v), giving the initial calculation above. Because neither m nor k is free in its final line, concluding the calculation by applying the remaining commands VIS m, HID k has no effect. Finally, assuming the precondition, $\forall$-quantifying over B, C, s in their type S, then renaming e, C to k, s', leaves only

$$KCC\text{—} \qquad (\forall s, s'{:}\, S; k{:}\, K \cdot (\exists k'{:}\, K \cdot s \oslash k = s' \oslash k'))\ , \tag{1}$$

which we call the *Key-Complete Condition* for encryption $\oslash$ with key-set K.

Figure 8: Deriving the Key-Complete Condition for the Encryption Lemma (Sec. 7)

and whether-he-paid. The exclusive-or of the three announcements, known to all, is true iff some cryptographer paid; but it reveals nothing about which one did.

We model this with global Boolean variables p (some cryptographer paid) and p_i (Cryptographer i paid): a typical specification would include $p = p_0 \oplus p_1 \oplus p_2$ as a *post*-condition. If we take the waiter as observer, then from his point of view the postcondition might also include $p \Rightarrow \langle\!\langle p_0 \colon \mathbb{B} \rangle\!\rangle \wedge \langle\!\langle p_1 \colon \mathbb{B} \rangle\!\rangle \wedge \langle\!\langle p_2 \colon \mathbb{B} \rangle\!\rangle$, where in general for hidden (vector) h we introduce this abbreviation:

- *Complete ignorance* $\quad \langle\!\langle h \colon E \mid \Phi \rangle\!\rangle \;\widehat{=}\; (\forall e \colon E \cdot [h \backslash e]\Phi \Rightarrow \mathrm{P}(h{=}e)) \;,$ [8]

with omitted Φ defaulting to TRUE, expressing ignorance of h's value beyond its membership in $\{h \colon E \mid \Phi\}$. Thus in this case, even if p holds, still the waiter is to know nothing about whether p_0, p_1 or p_2 hold individually.

As a specification *pre*-condition we would find $\langle\!\langle p_0, p_1, p_2 \colon \mathbb{B} \mid \sum_i p_i \leq 1 \rangle\!\rangle$ expressing (with an abuse of notation) that the waiter considers any combination possible provided at most one p_i holds. Putting pre- and post- together, the suitability of a specification S could be expressed

$$\{\langle\!\langle p_0, p_1, p_2 \colon \mathbb{B} \mid \sum_i p_i \leq 1 \rangle\!\rangle\} \;\; \mathrm{S} \;\; \{p = p_0 \oplus p_1 \oplus p_2 \wedge p \Rightarrow \begin{pmatrix} & \langle\!\langle p_0 \colon \mathbb{B} \rangle\!\rangle \\ \wedge & \langle\!\langle p_1 \colon \mathbb{B} \rangle\!\rangle \\ \wedge & \langle\!\langle p_2 \colon \mathbb{B} \rangle\!\rangle \end{pmatrix}\} \;, \tag{2}$$

and Fig. 6 shows it indeed is satisfied when S is the assignment $p{:=}\, p_0 \oplus p_1 \oplus p_2$. (Compare Halpern and O'Neill's specification [11]: ours is less expressive because we deal with only one agent at a time.)

Rather than prove (2) for an implementation directly —which could be complex— we can *use program algebra to manipulate a specification for which (2) has already been established.* An implementation reached via ignorance-preserving refinement steps requires no further proof of ignorance-preservation [15].

A derivation of the *DC* protocol, from the specification of Fig. 6, is given in Fig. 9. To illustrate the possibility of different viewpoints, we observe as Cryptographer 0 —rather than the waiter— which makes p_0 visible rather than hidden and thus alters our global context to VIS p, p_0; HID p_1, p_2: we can see the final result, and we can "see" whether *we* paid; but we cannot see directly whether Cryptographers 1 or 2 paid.

[8] Naturally we have $\models \langle\!\langle h \colon E \rangle\!\rangle \Rightarrow \mathrm{P}(h{\in}E)$, but in fact the latter is strictly weaker: for example, program $h{:\in}\, \{0, 1\}$ establishes $\mathrm{P}(h{\in}\{1, 2\})$ but not $\langle\!\langle h \colon \{1, 2\} \rangle\!\rangle$.

The global context VIS p, p_0; HID p_1, p_2 is Cryptographer 0's viewpoint.

$p{:=}\, p_0 \oplus p_1 \oplus p_2$

$=$ **skip**; "SKIP is identity"
$p{:=}\, p_0 \oplus p_1 \oplus p_2$

$\sqsubseteq$ "Encryption Lemma"
$[\![$ **vis s_1, c_1; hid c_2** $\cdot$
$c_2{:}\in \mathbb{B}$;
$s_1{:=}\, c_1 \oplus p_1 \oplus c_2$ $]\!]$;
$p{:=}\, p_0 \oplus p_1 \oplus p_2$

$=$ "move into block; typed declaration"
$[\![$ VIS s_1, c_1; HID **$c_2{:}\,\mathbb{B}$** $\cdot$
$s_1{:=}\, c_1 \oplus p_1 \oplus c_2$
$p{:=}\, p_0 \oplus p_1 \oplus p_2$ $]\!]$

$=$ "new inner block equals SKIP"
$[\![$ VIS s_1, c_1; HID $c_2{:}\,\mathbb{B}$ $\cdot$
$s_1{:=}\, c_1 \oplus p_1 \oplus c_2$
$p{:=}\, p_0 \oplus p_1 \oplus p_2$
$[\![$ **vis s_0, s_2, c_0** $\cdot$
$s_0{:=}\, c_0 \oplus p_0 \oplus c_1$;
$s_2{:=}\, p \oplus s_0 \oplus s_1$ $]\!]$ $]\!]$

$=$ "reorder"
$[\![$ VIS **s_0**, s_1, **s_2**, **c_0**, c_1; HID $c_2{:}\,\mathbb{B}$ $\cdot$
$s_0{:=}\, c_0 \oplus p_0 \oplus c_1$;
$s_1{:=}\, c_1 \oplus p_1 \oplus c_2$
$p{:=}\, p_0 \oplus p_1 \oplus p_2$
$s_2{:=}\, p \oplus s_0 \oplus s_1$ $]\!]$

$=$ "Boolean algebra"
$[\![$ VIS s_0, s_1, s_2, c_0, c_1; HID $c_2{:}\,\mathbb{B}$ $\cdot$
$s_0{:=}\, c_0 \oplus p_0 \oplus c_1$;
$s_1{:=}\, c_1 \oplus p_1 \oplus c_2$
$p{:=}\, p_0 \oplus p_1 \oplus p_2$
$s_2{:=}\, c_2 \oplus p_2 \oplus c_0$ $]\!]$

$=$ "reorder"
$[\![$ VIS s_0, s_1, s_2, c_0, c_1; HID $c_2{:}\,\mathbb{B}$ $\cdot$
$s_0{:=}\, c_0 \oplus p_0 \oplus c_1$;
$s_1{:=}\, c_1 \oplus p_1 \oplus c_2$
$s_2{:=}\, c_2 \oplus p_2 \oplus c_0$
$p{:=}\, p_0 \oplus p_1 \oplus p_2$ $]\!]$

$=$ "Boolean algebra"
$[\![$ VIS s_0, s_1, s_2, c_0, c_1; HID $c_2{:}\,\mathbb{B}$ $\cdot$
$s_0{:=}\, c_0 \oplus p_0 \oplus c_1$;
$s_1{:=}\, c_1 \oplus p_1 \oplus c_2$
$s_2{:=}\, c_2 \oplus p_2 \oplus c_0$
$p{:=}\, s_0 \oplus s_1 \oplus s_2$ $]\!]$.

The derivation begins at upper-left with the specification, ending with the implementing protocol at right. **Bold text** highlights changes.

The "typed declaration" HID $c_2{:}\,\mathbb{B}$ $\cdot$ abbreviates HID $c_2 \cdot c_2{:}\in \mathbb{B}$.

Local variables c_i (coins) and s_i (Cryptographer i *said*) for $i{:}\,0, 1, 2$ are introduced during the derivation, which depends principally on Key-Completeness and the Boolean algebra of $\oplus$.
In our use of Lem. 2.1 (Encryption) the message (m) is s_1, the secret (s) is p_1, the encryption ($\oslash$) is $\cdot(\oplus\, c_1 \oplus)\cdot$ —which satisfies the Key-Complete Condition (1) for both values of the visible c_1— and the key (k) is c_2.
Other refinements, such as moving statements into blocks where there is no capture, and swapping of statements that do not share variables, are examples of Pr1 and Pr2.

Figure 9: Deriving the Dining Cryptographers' Protocol

9 Contributions, comparisons and conclusions

Consider this (invalid) refinement, in which we use integers rather than Booleans:

$$p{:=}\, p_0 \oplus p_1 \oplus p_2 \quad \sqsubseteq^{?} \quad [\![\text{ VIS } s_0, s_1, s_2, c_0, c_1 : \{0,1\}; \text{ HID } c_2 : \{0,1\} \cdot s_0{:=}\, c_0 + p_0 - c_1; \; s_1{:=}\, c_1 + p_1 - c_2 \; s_2{:=}\, c_2 + p_2 - c_0 \; p{:=}\, (s_0 + s_1 + s_2) \bmod 2 \;]\!] \; . \tag{3}$$

If $c_0, p_0, c_1 = 1, 1, 0$, then visible s_0 will be 2 and p_0=1 is revealed.
It shouldn't be.

The coins c_i cancel just as in Fig. 9, but this time additively.

Our contribution is to disallow this refinement, and others like it.[9]

More generally our contribution is to have altered the rules for refinement of sequential programs, just enough, so that ignorance of hidden variables is preserved. We derive correct protocols (Fig. 9), but no longer incorrect ones (3).

Halpern and O'Neill apply the Logic of Knowledge to secrecy [10] and anonymity [11]. Compared to their work, 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 [8]; and our programs are not concurrent. They treat *DC*, as do Engelhardt, Moses & van der Meyden [6], and van der Meyden & Su [23].

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 security comes from regarding hidden variables as "high-security" and visible variables as "low-security", and concentrating on program semantics rather than *e.g.* extension via syntactic annotations: thus we take the *extensional* view [4] of *non-interference* [9] where security properties are deduced directly from the semantics of a program [20, III-A]. Recent examples of this include elegant work by Leino *et al.* [14] and Sabelfeld *et al.* [21].

Again we have specialised severely —we do not consider lattices, nor divergence (*e.g.* infinite loops), nor concurrency, nor probability. However our "agenda" of Refinement, the Logic of Knowledge and Algebra has induced four interesting differences from the usual semantic approaches to security:

[9] One role of Formal Methods is to *prevent* people from writing programs, those that are unintentionally wrong.

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 *DC specification* is not secure: it reveals whether the cryptographers (collectively) paid or not. To attempt to prove the *DC implementation* (absolutely) secure is therefore inappropriate.

   However, if we did wish to prove absolute security we would simply require refinement of an absolutely secure specification (*e.g.* SKIP, or $v{:\in}\, T$).

2. *We concentrate on* final- *rather than initial hidden values.* This is induced by the Kripke structure, which models what other states are possible "now".

   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 operate on a local hidden copy, thrown away at the end of the block. Thus $[\![\ \text{HID}\ h'{:}\,\{h\} \cdot h'{:=}\,0\ ]\!]$ becomes secure (for us too), and $[\![\ \text{HID}\ h'{:}\,\{h\} \cdot v{:=}\,h'; h'{:\in}\,T\ ]\!]$ becomes insecure.

   A direct comparison with *non-interference* considers the relational semantics R of a program, operating 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 value. Operational reasoning (as Sec. 6) [17] then shows that

$$\textit{Absolute security} — \qquad (\forall v, h, h'{:}\,T \cdot R.v.h = R.v.h')$$

   is necessary and sufficient, which is non-interference for R exactly [14, 21].

3. *We insist on perfect recall.* This is induced by our algebraic principles (recall the *gedanken* experiment of Sec. 2), 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.

   It is also a side-effect of (thread) concurrency [10],[20, IV-B], but has different causes. We are concerned with ignorance-preservation during program *development*; the concurrency problem occurs during program *execution*.

   The "label creep" [20, II-E] caused by perfect recall, where the build-up of un-erasable leaks makes the program eventually useless, is miti-

gated because our knowledge of the *current* hidden values can decrease (via *e.g.* $h{:\in}\, T$), even though knowledge of *initial-* (or even previous) values cannot.

4. *We do not require "low-view determinism"* [20, IV-B]. This is induced by our explicit policy of retaining abstraction and determining exactly when we can "refine it away" (and when we cannot). The approach of Roscoe and others instead requires low-level behaviour to be deterministic [19].

We conclude that *ignorance refinement* is able to handle examples of simple design — even though their significance may be far from simple. Because *wp*-logic for ignorance retains most structural features of traditional *wp*, we expect that loops and their invariants, divergence, and concurrency via *e.g. action systems* [1] will be feasible extensions.

Adding probability via modal "expectation transformers" [16] is a longer-term goal, but will require a satisfactory treatment of conditioning (the probabilistic version of *Shrink shadow*) in that context.

Acknowledgements

Thanks to Yoram Moses and Kai Engelhardt for suggesting the problem and the approach, and to them, Annabelle McIver, Jeff Sanders and Susan Stepney for comments and suggestions. Thanks also to Michael Clarkson and Chenyi Zhang for references.

References

[1] R.-J.R. Back and R. Kurki-Suonio. Decentralisation of process nets with centralised control. In *2nd ACM SIGACT-SIGOPS Symp. Principles of Distributed Computing*, pages 131–42, 1983.

[2] R.-J.R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer Verlag, 1998.

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

[4] E.S. Cohen. Information transmission in sequential programs. *ACM SIGOPS Operatings Systems Review*, 11(5):133–9, 1977.

[5] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall International, Englewood Cliffs, N.J., 1976.

[6] K. Engelhardt, Y. Moses, and R. van der Meyden. Kai and Yoram's DC paper.

[7] K. Engelhardt, R. van der Meyden, and Y. Moses. A refinement theory that supports reasoning about knowledge and time. In R. Nieuwenhuis and A. Voronkov, editors, *LPAR*, volume 2250 of *LNCS*, pages 125–141. Springer Verlag, 2001.

[8] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning about Knowledge*. MIT Press, 1995.

[9] J.A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc. IEEE Symp. on Security and Privacy*, pages 75–86, 1984.

[10] J.Y. Halpern and K.R. O'Neill. Secrecy in multiagent systems. In *Proc. 15th IEEE Computer Security Foundations Workshop*, pages 32–46, 2002.

[11] J.Y. Halpern and K.R. O'Neill. Anonymity and information hiding in multiagent systems. In *Proc. 16th IEEE Computer Security Foundations Workshop*, 2003.

[12] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–80, 583, October 1969.

[13] J. Jacob. Security specifications. In *IEEE Symposium on Security and Privacy*, pages 14–23, 1988.

[14] K.R.M Leino and R. Joshi. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–38, 2000.

[15] Heiko Mantel. Preserving information flow properties under refinement. In *Proc. IEEE Symp. Security and Privacy*, pages 78–91, 2001.

[16] Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Technical Monographs in Computer Science. Springer Verlag, New York, 2004.

[17] C.C. Morgan. Appendix to *The Shadow Knows*.

[18] C.C. Morgan. *Programming from Specifications*. Prentice-Hall, second edition, 1994.

[19] A. W. Roscoe, J.C.P Woodcock, and L. Wulf. Non-interference through determinism. *Journal of Computer Security*, 4(1):27–54, 1996.

[20] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE Jnl. Selected Areas Comm.*, 21(1), 2003.

[21] A. Sabelfeld and D. Sands. A PER model of secure information flow. *Higher-Order and Symbolic Computation*, 14:59–91, 2110.

[22] M.B. Smyth. Power domains. *Jnl. Comp. Sys. Sciences*, 16:23–36, 1978.

[23] R. van der Meyden and K. Su. Symbolic model checking the knowledge of the Dining Cryptographers. In *Proc. 17th IEEE Works. Computer Security Foundations*, pages 280–91, 2004.