Math. Struct. in Comp. Science (2015), *vol.* 25, *pp.* 320–360. © Cambridge University Press 2014
doi:10.1017/S0960129513000625 First published online 10 November 2014

Hidden-Markov program algebra with iteration

ANNABELLE MCIVER[†], LARISSA MEINICKE[‡] and
CARROLL MORGAN[§]

[†]*Dept Comp Sci, Macquarie University, NSW, Australia*
Email: annabelle.mciver@mq.edu.au
[‡]*Dept Comp Sci, Univ Queensland, Qld, Australia*
Email: l.meinicke@uq.edu.au
[§]*School Comp Sci and Eng, Univ NSW, NSW, Australia*
Email: carrollm@cse.unsw.edu.au

Received January 2011; revised July 2013

We use hidden Markov models to motivate a quantitative compositional semantics for noninterference-based security with iteration, including a refinement- or 'implements' relation that compares two programs with respect to their information leakage; and we propose a program algebra for source-level reasoning about such programs, in particular as a means of establishing that an 'implementation' program leaks no more than its 'specification' program.

This joins two themes: we extend our earlier work, having iteration but only *qualitative* (Morgan 2009), by making it quantitative; and we extend our earlier *quantitative* work (McIver *et al.* 2010) by including iteration.

We advocate stepwise refinement and source-level program algebra – both as conceptual reasoning tools and as targets for automated assistance. A selection of algebraic laws is given to support this view in the case of quantitative noninterference; and it is demonstrated on a simple iterated password-guessing attack.

1. Introduction: extant theory and practices

Hidden Markov models, or HMMs, extend Markov processes by supposing that the process state is not directly visible: only certain observations of it can be made (Jurafsky and Martin 2000). How HMMs motivate a quantitative noninterference-security program semantics is our principal topic: the hidden state of the HMM has 'high security' and the observations that the HMM allow have 'low security.'

Program algebra is the manipulation of program texts themselves, i.e. as syntax and according to algebraic rules laid down beforehand, with the aim of showing equivalence or ordering with respect to a so-called 'refinement' relation (Section 2) between one program and another. That requires a semantics, and proofs of the elementary rules wrt. that semantics. Furthermore, these rules must be preserved by context in order for true algebra to be possible: in programming semantics, that last is called *compositionality*. This represents an 'up front' cost for reasoning about program behaviours. When that cost has been paid however, just once, then the benefits accrue forever after – every time an equality or refinement can be shown syntactically without 'descending' into the semantics.

The significance of *iteration* is that its proper treatment, via suprema of chains, makes interesting demands on the semantic machinery already set-up for straight-line, quantitative noninterference programs (Aldini and Pierro 2004; McIver *et al.* 2010).

Our first specific contribution extends an existing (but recent (McIver *et al.* 2010)) compositional semantics for straight-line quantitative noninterference security, one with a novel two-level 'hyper-distribution' semantics, by showing how hypers (for short) – previously introduced without detailed motivation – are in fact directly suggested by the mathematical machinery of HMMs (Section 3). Our second contribution adds iterating programs to that (Section 6), requiring thus a treatment of nontermination and fixed points: this would be straightforward were it not for the fact that supremum-completeness, on which fixed-points' existence usually relies, does *not* appear to hold.

Our third contribution (Section 7) is to show how, in spite of the incompleteness, we can via a more-specialized 'termination order' retain discrete distributions for the treatment of loops: that gives a simpler theory than (the more general) measures would require. Nevertheless, our further goal of extending compositional-closure (McIver *et al.* 2010) to iteration does seem to require measures: at that point, there is no escape (Section 12).

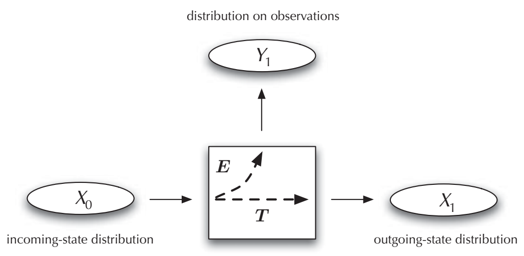
Our final contribution is a selection of algebraic laws (Section 9), and the treatment of an example (Section 10) illustrating the style of reasoning we hope they will facilitate.

2. Program algebra and refinement

Algebra is powerful, and it is general; and it is especially useful in program verification where algebra's feature of *compositionality* allows the reuse that simplifies verification tasks. *Program* algebra in particular provides equalities or refinements (see below) that, although proved in isolation between program fragments, can then be reused freely within arbitrary contexts, drastically simplifying correct-by-construction and/or post-hoc verification arguments.

A *refinement* ordering between programs is weaker than equality: it defines the relationship that must hold between specifications and their implementations in a given application domain (Wirth 1971; Morgan 1994; Back and von Wright 1998). In special applications, such as noninterference security, the refinement relation is adjusted – usually made more restrictive – to take further aspects into account: here it will be the possible release of high-level information. Thus secure refinement checks not only (non)termination, but also compares programs to see which one releases more information about hidden, high-security variables: it is more distinguishing than standard program refinement.

For example, take integer variables v, h and suppose that v is visible (low-security) whereas h is hidden (high-security). Furthermore, assume an attacker with 'perfect recall', i.e. one who remembers visible variables' values even if they are subsequently overwritten. (We explain this assumption in Sections 4.2 and 12 below.) Then we would expect the refinement $(\mathsf{v}{:=}\,\mathsf{h}{\div}2; \mathsf{v}{:=}\,\mathsf{v}{\div}2) \sqsubseteq \mathsf{v}{:=}\,\mathsf{h}{\div}4$ but, crucially, not the reverse. On the left, observing the first assignment to v as well as the second (perfect recall) allows us to distinguish $\mathsf{h}{=}1$ from $\mathsf{h}{=}3$; but on the right we cannot do that. The right-hand program is a refinement of the left-hand one because it is more secure; with an appropriate security-refinement algebra we would show this syntactically (Section 9.3).


distribution on observations
Y_1
E
X_0
T
X_1
incoming-state distribution
outgoing-state distribution

By *a priori* we mean that the distribution X_1 is determined statically, from information "already" available and in particular is not derived from an actual execution.

Fig. 1. A hidden Markov model, *a priori* view.

3. Hidden Markov models and hyper-distributions

3.1. *Basic structure of HMMs*

A HMM comprises a set $\mathcal{X}$ of states, a set $\mathcal{Y}$ of observations, and two stochastic matrices $\boldsymbol{T}, \boldsymbol{E}$ (Jurafsky and Martin 2000): the *transition* probabilities $\boldsymbol{T}$ give for any two states $x_{\{0,1\}}\in\mathcal{X}$ the (conditional) probability $\boldsymbol{T}(x_1|x_0)$ that a transition will end in final state x_1 given that it began in initial state x_0; and the *emission* probabilities give for any state x_0 and observation $y_1\in\mathcal{Y}$ the probability $\boldsymbol{E}(y_1|x_0)$ that y_1 will be emitted, and thus observed, given the initial state x_0. Typically an HMM is analysed over a number of steps $i = 0, 1, \ldots$ from some initial distribution X_0 over $\mathcal{X}$, so that a succession of states $x_1, x_2, \ldots$ and observations $y_1, y_2, \ldots$ occurs, where each x_i related to x_{i+1} by $\boldsymbol{T}$ and to y_{i+1} by $\boldsymbol{E}$.

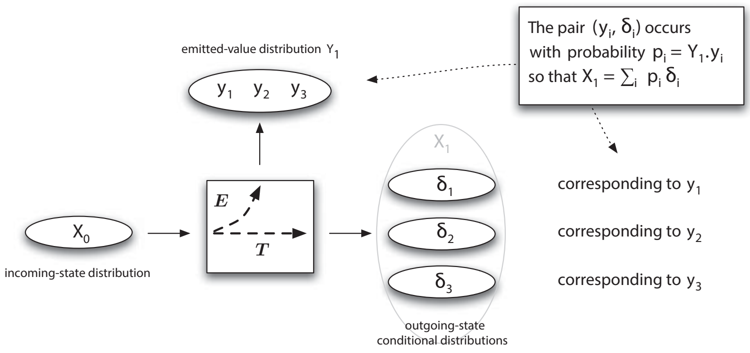
We assume finitely many states in the state space, and thus use discrete distributions throughout.[†]

We illustrate a single step in Figure 1. With X_0 the distribution of incoming state x_0, the distribution X_1 of outgoing states x_1 is the multiplication of X_0 as a row-vector by $\boldsymbol{T}$ as a matrix, that is $\Pr(X_1{=}x_1) := \sum_{x_0} \Pr(X_0{=}x_0)\boldsymbol{T}(x_1|x_0)$. Similarly, the distribution Y_1 of observations y_1 is given by a (matrix) multiplication amounting to $\Pr(Y_1{=}y_1) := \sum_{x_0} \Pr(X_0{=}x_0)\boldsymbol{E}(y_1|x_0)$. The 'hidden' essence of the model is that though we cannot see the incoming x_0's and the outgoing x_1's directly, still the observation of y_1 tells us something about each if we do know the incoming state distribution X_0 and the matrices $\boldsymbol{T}, \boldsymbol{E}$.

3.2. *A priori and a posteriori distributions on the state-space $\mathcal{X}$*

The *a priori* distribution on the HMM's input is X_0, and the *a posteriori* distribution on the input can be calculated from $\boldsymbol{T}, \boldsymbol{E}$, and X_0 in the usual way, via Bayes' formula, once we have observed y_1.

[†] The countably infinite supports mentioned in the introduction occur, eventually, in spite of this finiteness assumption (Section 8.4).


The pair (y_i, δ_i) occurs
with probability $p_i = Y_1.y_i$
so that $X_1 = \sum_i p_i \delta_i$
emitted-value distribution Y_1
y_1 y_2 y_3
E
T
X_0
incoming-state distribution
X_1
δ_1
δ_2
δ_3
outgoing-state conditional distributions
corresponding to y_1
corresponding to y_2
corresponding to y_3

By *a posteriori* we mean that the conditional distributions $\delta_{\{1,2,3\}}$ are deduced *after* observation of the emitted values $y_{\{1,2,3\}}$, and represent a revision of the *a priori* knowledge of the outgoing state as represented in the X_1 of Fig. 1.

Fig. 2. A hidden Markov model, *a posteriori* view.

But we concentrate instead on the output. The *a priori* distribution of outgoing x_1 is X_1 as calculated in Section 3.1 above. Its *a posteriori* distribution is conditioned on the emitted y_1 actually observed: it too is determined by the usual Bayes formula

$$\Pr(X_1{=}x_1|Y_1{=}y_1) := \frac{\sum_{x_0} \Pr(X_0{=}x_0)\boldsymbol{E}(y_1|x_0)\boldsymbol{T}(x_1|x_0)}{\sum_{x_0} \Pr(X_0{=}x_0)\boldsymbol{E}(y_1|x_0)}, \qquad (1)$$

that is the (joint) probability that x_1, y_1 both occurred divided by the overall (marginal) probability that y_1 occurred. Thus, *before* we observe any y_1 we believe the distribution of outgoing x_1 to be X_1, and *after* we observe y_1 we believe that distribution to be as (1). This view is illustrated in Figure 2.

3.3. *The attacker's point of view: an equivalent representation*

Although the matrices $\boldsymbol{T}, \boldsymbol{E}$ determine the HMM completely, we suggest that from the point of view of an attacker trying to determine the state of the HMM, it would be more useful to consider a different (but equivalent) formulation: the effect of one step from a known initial distribution X_0 is a joint distribution over observations in $\mathcal{Y}$ and their corresponding outgoing conditional distributions over $\mathcal{X}$: this structure thus comprises values Δ of type $\mathbb{D}(\mathcal{Y}{\times}\mathbb{D}\mathcal{X})$, where we write $\mathbb{D}\mathcal{X}$ and similar for the type of *discrete distributions* over $\mathcal{X}$, thus one-summing functions of type $\mathcal{X}{\rightarrow}[0,1]$. That is, each Δ gives for a pair (y_1, δ_1) in $\mathcal{Y}{\times}\mathbb{D}\mathcal{X}$ the probability that an attacker will observe y_1 and will conclude from it that x_1 has *a posteriori* distribution δ_1.

We call such Δ-values *hyper-distributions*, or just *hypers*. Since Δ is a joint distribution (jointly over $\mathcal{Y}$ and $\mathbb{D}\mathcal{X}$), we can speak of its left- and right-marginal distributions: the left-marginal distribution $\overleftarrow{\Delta}$ is of type $\mathbb{D}\mathcal{Y}$, and is in fact just Y_1 from above. That is, the distribution Y_1 of emitted observations is recovered as $\overleftarrow{\Delta}$.

The right-marginal distribution $\overrightarrow{\Delta}$ of the hyper is more interesting: it is of type $\mathbb{D}^2\mathcal{X}$ and, although it *averages* to the outgoing state distribution X_1 (in the sense shown in Figure 2), most of the popular (conditional) information-entropy measurements are likely to decrease, becoming less than the entropy of X_1 itself: that decrease quantifies the 'leak' that the emissions of Y_1 represent. For example, the *conditional Shannon entropy* of $\overrightarrow{\Delta}$, defined $\sum_{\delta:\mathbb{D}\mathcal{X}} \overrightarrow{\Delta}(\delta)\mathrm{H}(\delta)$ over the possible *a posteriori* distributions δ, is no more than $\mathrm{H}(X_1)$, the Shannon entropy of the *a priori* outgoing distribution X_1 itself.[†]

Thus, the denotational-style semantic representation we extract from HMM-theory is the *hyper-distribution* of type $\mathbb{D}(\mathcal{Y}{\times}\mathbb{D}\mathcal{X})$, a nesting of one distribution within another. As we will see, this allows us to equip the semantic space with a 'refinement' partial order; but it is *security* refinement, so that for hypers $\Delta_{\{0,1\}}$ one can speak of whether Δ_0 is more or less secure than Δ_1 or, if not, whether they are perhaps simply security-incomparable.

3.4. *A probabilistic monad*

A further benefit of conventional denotational techniques is our access to computational monads (Giry 1981; Moggi 1989; van Breugel 2005), simplifying the presentation considerably.

From here on, we use a dot '.' for function application, rather than parentheses $(\cdot)$, writing thus $f.x$ rather than $f(x)$. For curried functions, we will usually have $f.x.y$ rather than e.g. either $f(x,y)$ or $f(x)(y)$.[‡] As a result, given distribution $X{:}\mathbb{D}\mathcal{X}$ the probability it assigns to $x{:}\mathcal{X}$ is simply $X.x$, that is $\Pr(X{=}x)$ but written more compactly and taking advantage of the fact that X is just a function (of type $\mathcal{X}{\to}[0,1]$). The same economy accrues for random variables.

The monad structure for computations (Moggi 1989) supposes a triple $(\mathbb{K}, \boldsymbol{\eta}, \boldsymbol{\mu})$ where $\mathbb{K}$ is an endofunctor on a given category, and $\boldsymbol{\eta}, \boldsymbol{\mu}$ are natural transformations satisfying certain coherence conditions. An example of this is the *Giry monad* (Giry 1981), typically used for probabilistic computations; in its general form, its functor takes an object $(\Omega, \mathcal{B}_\Omega)$ comprising a set Ω and a sigma-algebra $\mathcal{B}_\Omega$ on it to the set of probability measures on $(\Omega, \mathcal{B}_\Omega)$, endowed with a suitable sigma-algebra of its own, induced from the given $\mathcal{B}_\Omega$.

Working here with discrete measures, our use of the monad will be modest and we will use suggestive names for its components, based on its specialization to discrete distributions and functional programming. In particular,

functor $\mathbb{D}$ – Given set $\mathcal{S}$ write $\mathbb{D}\mathcal{S}$ for the set of discrete distributions over $\mathcal{S}$.

push-forward map – Given two sets $\mathcal{X}, \mathcal{Y}$ and a function $f{:}\mathcal{X}{\to}\mathcal{Y}$ write $\mathbb{D}f$, the action of the functor on the function, as $\mathsf{map}.f{:}\mathbb{D}\mathcal{X}{\to}\mathbb{D}\mathcal{Y}$.[§] In the probability literature this is

[†] For distribution X in $\mathbb{D}\mathcal{X}$ such that $\Pr(X{=}x_i)$ is p_i, the Shannon entropy $\mathrm{H}(X)$ of X is given by $-\Sigma_i\, p_i \ln p_i$. As remarked, a number of other security-based definitions of entropy give the same inequality (Köpf and Basin 2007).

[‡] An advantage of this is that it distinguishes function application from the many other uses of parentheses, and produces self-contained expressions thus of less clutter. In this respect we compare $\mathrm{H}.X = -\Sigma_x\, X.x \ln(X.x)$ with the conventional presentation of Shannon entropy in Footnote † with its indices i and temporary names p_i.

[§] This is consistent with the definition of map in functional programming.

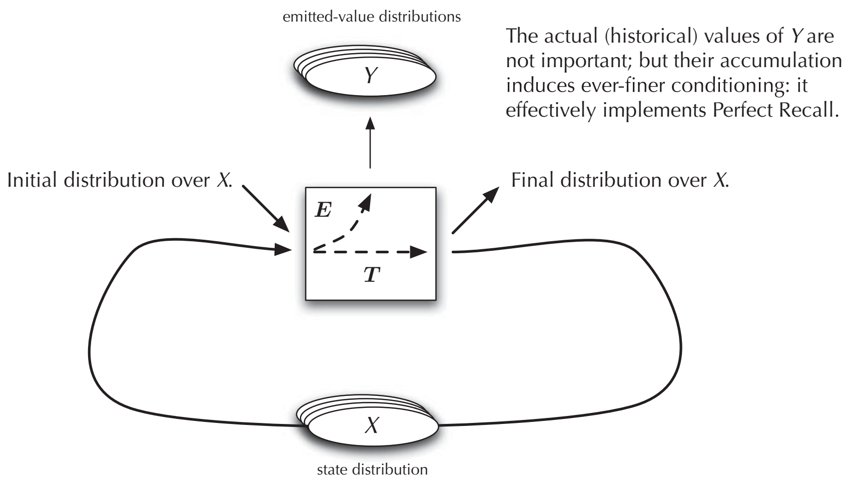

emitted-value distributions
Y
The actual (historical) values of Y are not important; but their accumulation induces ever-finer conditioning: it effectively implements Perfect Recall.
Initial distribution over X.
E
T
Final distribution over X.
X
state distribution

Fig. 3. A hidden Markov model: iteration accumulates leakage.

called the *push-forward*, defined for any $X{:}\mathbb{D}\mathcal{X}$ and $y{:}\mathcal{Y}$ in the discrete case as

$$\mathsf{map}.f.X.y := X.(f^{-1}.y) \;=\; (\textstyle\sum x{:}\mathcal{X} \mid f.x{=}y \bullet X.x).$$

multiplication avg – The multiplication (natural) transformation $\boldsymbol{\mu}{:}\mathbb{D}^2\mathcal{X}{\to}\mathbb{D}\mathcal{X}$ averages the distributions in its argument distribution-of-distributions, to give a distribution again. We write that as avg for 'average' and in the discrete case for $\boldsymbol{X}{:}\mathbb{D}^2\mathcal{X}$ it is defined for any $x{:}\mathcal{X}$ as

$$\mathsf{avg}.\boldsymbol{X}.x := (\textstyle\sum X{:}\mathbb{D}\mathcal{X} \bullet \boldsymbol{X}.X{\times}X.x).$$

Kleisli composition via lifting – For two functions $f{:}\mathcal{X}{\to}\mathbb{D}\mathcal{Y}$ and $g{:}\mathcal{Y}{\to}\mathbb{D}\mathcal{Z}$, the *lift* of g, written g^*, is defined to be the functional composition $\mathsf{avg}\circ\mathsf{map}.g$ of type $\mathbb{D}\mathcal{Y}{\to}\mathbb{D}\mathcal{Z}$ so that the Kleisli composition g after f is expressed $g^*{\circ}f$ in the usual way.

Similar definitions and notations apply for the monad $\underline{\mathbb{D}}$ associated with the *partial distributions* that sum to no more than one (Jones and Plotkin 1989; van Breugel 2005).

The immediate benefit from this monadic structure is the sequential composition of two HMMs, written say $H_1;H_2$, so that the final *a posteriori* distribution takes the observations from the first as well as the second HMM observables into account. If we take the type of $H_{\{1,2\}}$ to be $\mathcal{Y}{\times}\mathbb{D}\mathcal{X} \to \mathbb{D}(\mathcal{Y}{\times}\mathbb{D}\mathcal{X})$, then we want our definition of $H_1;H_2$ to have similar type, thus giving it the same features as a single HMM provided that the observations from both of its components are considered: the general case is illustrated in Figure 3.

For our single composition $H_1;H_2$, the outgoing result of H_1 is an *a posteriori* hyper which is then presented 'en bloc' as input to H_2. Via the type constructor, that intermediate hyper is a partitioning of some flattened distribution of type $\mathcal{Y}{\times}\mathbb{D}\mathcal{X}$ according to the observables emitted by H_1; [†] each partition of that flattened distribution – itself of type $\mathcal{Y}{\times}\mathbb{D}\mathcal{X}$ – is separately input to H_2, but after the final output is produced the partitioning is

[†] We call it a distribution simply to avoid a proliferation of names. In fact, it is isomorphically a distribution of type $\mathbb{D}(\mathcal{Y}{\times}\mathcal{X})$ whose left marginal is a point distribution.

'reassembled' in the final *a posteriori* distribution, thus neatly taking both the observations from $H_{\{1,2\}}$ into account. Crucially this partitioning and reassembling is done according to the original weightings, which is what allows us to use the monad:

$$\mathcal{Y}{\times}\mathbb{D}\mathcal{X} \xrightarrow{H_1} \mathbb{D}(\mathcal{Y}{\times}\mathbb{D}\mathcal{X}) \xrightarrow{\mathsf{map}.H_2} \mathbb{D}^2(\mathcal{Y}{\times}\mathbb{D}\mathcal{X}) \xrightarrow{\mathsf{avg}} \mathbb{D}(\mathcal{Y}{\times}\mathbb{D}\mathcal{X}).$$

Here, the original input type $\mathcal{Y}{\times}\mathbb{D}\mathcal{X}$ is transformed as we suggest by H_1 to $\mathbb{D}(\mathcal{Y}{\times}\mathbb{D}\mathcal{X})$, which then in its partitioned form is passed to H_2 and, via the $\mathsf{map}/\mathsf{avg}$ construction, that partition is reassembled after the action of H_2 on its components. That supplies our definition for $H_1;H_2$, thus also of type $\mathcal{Y}{\times}\mathbb{D}\mathcal{X} \to \mathbb{D}(\mathcal{Y}{\times}\mathbb{D}\mathcal{X})$. Note we do not need $\mathcal{Y}^2$ to 'combine' the observations of the two separate HMMs, an important advantage of this presentation: in Section 4.2(2,4) this is explained further.

4. Quantitative noninterference security for programs

4.1. *Noninterference via hidden and visible-variables; atomicity*

Take a simple programming model comprising a finite set $\mathcal{H}$ of hidden states, ranged over by (high-security) program variable(s) h, and a finite set $\mathcal{V}$ of visible states, ranged over by (low-security) program variable(s) v. The state space overall is thus the product $\mathcal{V}{\times}\mathcal{H}$, and our program texts refer to variables v,h.†

Observers of the program's execution can see v, but they cannot see h. Attackers of the program try to learn about h's final values, or at least their distribution, by observing v's values as execution of the program proceeds.

We begin with assignment statements as a basis: a simultaneous assignment is written $\mathsf{v},\mathsf{h}{:=}\,V,H$, allowing both expressions V,H to refer to the initial values of v,h without worrying about which one is updated first: the two expressions V,H may contain variables of either kind, or both. We base the assignment on a probabilistic-choice syntax $\mathsf{x}{:\in}\,X$ that means 'choose the new value of x according to the distribution X', where X itself is a distribution-valued expression that possibly depends on the initial value of x as well as other variables.

To keep track of variables in the generic semantic definitions, we write the distribution-valued expressions as explicit functions applied to them, so arriving at $\mathsf{v},\mathsf{h}{:\in}\; E.\mathsf{v}.\mathsf{h}, T.\mathsf{v}.\mathsf{h}$ as our basic simultaneous probabilistic assignment to the two variables; actual program texts of course simply use expressions over v,h in which such functions might occur. Thus $E.\mathsf{v}.\mathsf{h}$ is a distribution, depending on the initial values of v,h, according to which v's new value is chosen. Similarly $T.\mathsf{v}.\mathsf{h}$ is the distribution for the choice of h's new value. The statement is *atomic* in the sense that only its results are accessible, not how they were computed.

† For simplicity we are assuming that multiple hidden or visible variables are collected separately within vectors h or v, i.e. that v,h are the only variables present; but we won't clutter the presentation with the 'overhooks' for that. The program texts can refer of course to individual elements of the vectors, which references are interpreted as projections etc. in the usual way.

Ordinary, non-probabilistic assignments can be written $\mathsf{v},\mathsf{h}{:=}\,E.\mathsf{v}.\mathsf{h},\,T.\mathsf{v}.\mathsf{h}$ in which E, T now give values rather than distributions; they are clearly the special case of the above for point distributions, and so do not need a separate treatment.

4.2. *Connecting HMMs and the programming model; perfect recall*

The connection is made by identifying $\mathcal{V}$ with $\mathcal{Y}$, and $\mathcal{H}$ with $\mathcal{X}$. The visible v corresponds to the emitted observations y, thus to a sort of 'output buffer'; and the hidden h corresponds to the state x, passed from one program fragment through sequential composition to the next one.

A slight generalization is that the probabilistic choices can be influenced by the immediately previous emitted value (by y from the previous step, whose value is the initial value of v for this step), whereas in an HMM this is typically not done. This is only a notational convenience, since clearly the HMM's state spaces can be elaborated to allow the same freedom; but such conveniences are a part of adapting the HMM framework to programming practice.

Further elaborating the adaptation, we make the following remarks:

1. A typical sequential program will execute many individual atomic steps successively. The outgoing state from one step will be both its final value of h, fed-in automatically as the incoming state of the next step, *and* the last emitted observable value, found in v.
2. Although the observations emitted from each step will successively overwrite earlier values in v, the conditioning observations of those earlier outputs caused is *not* lost: it is preserved by the $\mathsf{map}/\mathsf{avg}$ composition. Thus, the partitioning expressed by the growing support-set of the outer $\mathbb{D}$ becomes finer on each step, so that deductions made by an attacker's having seen an earlier v are never forgotten. This is called *perfect recall* (Halpern and O'Neill 2002).
3. The distribution $E.\mathsf{v}.\mathsf{h}$ from which v's final value is chosen corresponds to the stochastic matrix $\boldsymbol{E}$ of the HMM. In effect, the h in $E.\mathsf{v}.\mathsf{h}$ selects the row of $\boldsymbol{E}$ that gives the distribution from which y, that is from which v is chosen. Similarly, the distribution $T.\mathsf{v}.\mathsf{h}$ from which h's final value is chosen corresponds to the stochastic matrix $\boldsymbol{T}$. The programs' access to v is why we include $(\mathcal{Y}\times)$ in the state.
4. The *a priori* view of the program is the extent to which we can determine the distribution of the final values of h by knowing the incoming distribution of v,h and the program text. The *a posteriori* view reflects the extra information about h finally that we have once we actually execute the program and note the successive emissions in v that occur during that execution. However, the *values* of those emissions need not be remembered: only the conditioning they induce is important. That is why we do not need 'sequence of $\mathcal{Y}$' in our state, in spite of perfect recall: the recall is expressed in the outer $\mathbb{D}$.

5. ***Refinement* increases entropy compositionally**

Our advocacy of stepwise refinement (Wirth 1971) for development of quantitatively noninterference secure programs suggest comparisons of specifications S with implementations I. Say that S is 'Shannon-refined' by I, writing $S \preceq_{\mathrm{se}} I$, just when for every incoming distribution of hidden values h the *a posteriori* conditional Shannon entropy produced by I, for h, is at least that produced by S (as at 4. above). Stepwise refinement wrt. Shannon entropy requires transitivity of $(\preceq_{\mathrm{se}})$ obviously; but it also would require that $S \preceq_{\mathrm{se}} I$ imply $\mathcal{C}(S) \preceq_{\mathrm{se}} \mathcal{C}(I)$ for any context $\mathcal{C}$ – that is, it should be *compositional.* And it is not, in general.

Define analogously $(\preceq_{\mathrm{br}})$ for comparing conditional Bayes Risk of outputs, in the same way; it is not compositional either. [†]

The refinement relation $(\sqsubseteq)$ introduced earlier (McIver *et al.* 2010), and extended here for iteration, in fact *is* compositional; and furthermore, it implies both $(\preceq_{\mathrm{se}})$ and $(\preceq_{\mathrm{br}})$. (Counter-examples for compositionality of $(\preceq_{\mathrm{se}})$ and $(\preceq_{\mathrm{br}})$ are given in the extended version of that work.) We now explain refinement.

5.1. *Comparing hyper-distributions*

We begin for simplicity with an entirely hidden state $\mathcal{X}$ (i.e. without $\mathcal{Y}$) thus having hypers $\mathbb{D}^2\mathcal{X}$. We ask whether, for (each) fixed observation y_1, one HMM 'reveals more' than the other in a sense made precise as follows.

A hyper Δ_S in $\mathbb{D}^2\mathcal{X}$, produced say as the output of one HMM, is 'refined by' another hyper Δ_I, produced by another HMM of the same type, if two distributions $\delta_S^{\{1,2\}}\colon\mathbb{D}\mathcal{X}$ in the support of Δ_S can be merged to form a single distribution δ_I in what becomes Δ_I. This merging increases a variety of (conditional) entropies, including the two mentioned above. We say that one HMM is entropy-refined by another when for corresponding inputs and corresponding values of emitted observables their outgoing hypers are refinement-related.

For an example, we restrict to Booleans T,F and write $\{\!\{x^{@p}, y^{@q}\ldots, z^{@r}\}\!\}$ for the discrete distribution assigning probabilities $p,q,\ldots,r$ to values $x,y,\ldots,z$: it is partial or total depending on whether $p{+}q{+}\cdots{+}r$ equals 1. Suppose the specification hyper Δ_S contains two distributions $\delta_S^1 := \{\!\{\mathsf{T}^{@\frac{1}{3}}, \mathsf{F}^{@\frac{2}{3}}\}\!\}$ and $\delta_S^2 := \{\!\{\mathsf{T}^{@\frac{1}{2}}, \mathsf{F}^{@\frac{1}{2}}\}\!\}$ with probabilities $p^1 := 1/4$ and $p^2 := 1/3$ respectively: thus Δ_S is partial and can itself be written $\{\!\{\delta_S^{1\,@\frac{1}{4}}, \delta_S^{2\,@\frac{1}{3}}, \ldots\}\!\}$. We first calculate a weighted merge as follows:

— Scale $\delta_S^{\{1,2\}}$ by their respective probabilities $p^{\{1,2\}}$ in Δ_S to get partial distributions $\{\!\{\mathsf{T}^{@\frac{1}{12}}, \mathsf{F}^{@\frac{1}{6}}\}\!\}$ and $\{\!\{\mathsf{T}^{@\frac{1}{6}}, \mathsf{F}^{@\frac{1}{6}}\}\!\}$.
— Add those together pointwise to get $\{\!\{\mathsf{T}^{@\frac{1}{4}}, \mathsf{F}^{@\frac{1}{3}}\}\!\}$.
— Normalize to get $\delta_I := \{\!\{\mathsf{T}^{@\frac{3}{7}}, \mathsf{F}^{@\frac{4}{7}}\}\!\}$ with probability $p := 7/12$ in Δ_I.

Then, we refine Δ_S by removing the two distributions $\delta_S^{\{1,2\}}$ (total weight 7/12) and replacing them by their weighted merge, the single δ_I (of the same weight), to give Δ_I. All the other points in the support of Δ_S would carry over unchanged into Δ_I; but of course this process can be repeated, since refinement is to be transitive. We see at (2) below that general entropy refinements are achieved by merges of more than two sources, having multiple targets and by 'pre-splitting' sources proportionally to allow them to participate in more than one merge: the essential idea is as given here.

[†] The Bayes risk is the largest guaranteed chance that one guess of h is *incorrect*.

5.2. *Preliminary definition of entropy refinement*

Distributions over our states in $\mathcal{X}$, i.e. in $\mathbb{D}\mathcal{X}$, are called *inner* distributions or just 'inners'. Distributions over inners, i.e. in $\mathbb{D}\mathbb{D}\mathcal{X} = \mathbb{D}^2\mathcal{X}$, are hypers, as we have seen. If we want to concentrate on the 'outer' $\mathbb{D}$ of a hyper, we refer to that as the *outer* distribution, or just 'outer'. We will (briefly) need distributions of hypers $\mathbb{D}^3\mathcal{X}$, called *super* distributions or just 'supers'.

Definition 5.1 (entropy refinement (preliminary definition)). Let the state-space be $\mathcal{X}$, a finite set and consider two hypers $\Delta_{\{S,I\}}{:}\mathbb{D}^2\mathcal{X}$. We say that Δ_S is *entropy refined* by Δ_I, written $\Delta_S{\preceq}\Delta_I$, iff there is a super $\mathbf{\Delta}{:}\mathbb{D}^3\mathcal{X}$ such that

$$\Delta_S = \mathsf{avg}.\mathbf{\Delta} \qquad \text{and} \qquad \mathsf{map}.\mathsf{avg}.\mathbf{\Delta} = \Delta_I .$$

We return to our example, hyper Δ_S now with three inners $\delta_S^1 {:=} \{\!\{\mathsf{T}^{@\frac{1}{3}}, \mathsf{F}^{@\frac{2}{3}}\}\!\}$ and $\delta_S^2 {:=} \{\!\{\mathsf{T}^{@\frac{1}{2}}, \mathsf{F}^{@\frac{1}{2}}\}\!\}$ and $\delta_S^3 {:=} \{\!\{\mathsf{T}^{@1}\}\!\}$ with probabilities $p^1 {:=} 1/4$ and $p^2 {:=} 1/3$ and $p^3 {:=} 5/12$ respectively, where the third inner is chosen to bring the (outer's) sum to 1, i.e. to make it total.

Now to reach the entropy refinement of Δ_S given by hyper Δ_I, we merge the first two inners and simply carry the third through. The mediating super $\mathbf{\Delta}$ contains the two hypers

— hyper $\Delta^1 {:=} \{\!\{\delta_S^{1\,@\frac{3}{7}}, \delta_S^{2\,@\frac{4}{7}}\}\!\}$ with probability $7/12$ in $\mathbf{\Delta}$ and

— hyper $\Delta^2 {:=} \{\!\{\delta_S^{3\,@1}\}\!\}$ with probability $5/12$ in $\mathbf{\Delta}$,

so that $\Delta_S = \mathsf{avg}.\mathbf{\Delta}$, for example because

$$\mathsf{avg}.\mathbf{\Delta}.\delta_S^1 = 3/7{\times}7/12 = 1/4 = p^1 = \Delta_S.\delta_S^1 .$$

From Definition 5.1 the hyper Δ_I is therefore given by $\mathsf{map}.\mathsf{avg}.\mathbf{\Delta}$, that is

— inner distribution $\mathsf{avg}.\Delta^1 = \mathsf{avg}.\{\!\{\delta_S^{1\,@\frac{3}{7}}, \delta_S^{2\,@\frac{4}{7}}\}\!\} = \{\!\{\mathsf{T}^{@\frac{3}{7}}, \mathsf{F}^{@\frac{4}{7}}\}\!\}$
with probability $7/12$ and

— inner distribution $\mathsf{avg}.\Delta^2 = \mathsf{avg}.\{\!\{\delta_S^{3\,@1}\}\!\} = \delta_S^3$ itself, carried through
with probability $5/12$ as we expected.

A second example of entropy refinement is given at (2) below.

It can be shown (McIver *et al.* 2010) that refinement is indeed a partial order: reflexivity is obvious, anti-symmetry follows from an entropy-based argument or alternatively from 'colour mixing' (Sonin 2008). Its transitivity can be shown using matrices, or by a monadic approach using general properties of map and avg and specific properties of the probabilistic functor.

6. Iteration, refinement chains and incompleteness

Iteration and entropy refinement taken together impose new demands on our semantic space: the existence of a least program, and closure under limits. These demands are imposed since iterations are usually defined via least fixed points whose existence is trivial if the program space forms a *cpo* under the refinement ordering (Tarski 1955). Since we have not yet introduced non-termination, the space $(\mathbb{D}^2\mathcal{X}, \preceq)$ has no least element. Even more significant however, as we show below, is that not all of its non-empty entropy-refinement chains have a supremum.

As a result, the usual technique of defining iterations via refinement-least fixed points will not obviously apply – even after extending the space and its ordering to incorporate non-terminating behaviours – and we will have to do something slightly different (Section 7.3).

6.1. *An example of incompleteness*

Define again $\mathcal{X} := \{\mathsf{T}, \mathsf{F}\}$, and let δ_p be the inner $\{\!\{\mathsf{T}^{@p}, \mathsf{F}^{@1-p}\}\!\}$, alternatively written $\mathsf{T}\,{}_p\!\oplus \mathsf{F}$, for any $0 \leqslant p \leqslant 1$. Form the sequence of hypers

$$\begin{array}{lcl} \Delta_1 &:=& \{\!\{\delta_0^{@\frac12}, \delta_1^{@\frac12}\}\!\} \\ \Delta_2 &:=& \{\!\{\delta_0^{@\frac14}, \delta_{1/2}^{@\frac12}, \delta_1^{@\frac14}\}\!\} \\ \Delta_3 &:=& \{\!\{\delta_0^{@\frac18}, \delta_{1/4}^{@\frac14}, \delta_{1/2}^{@\frac14}, \delta_{3/4}^{@\frac14}, \delta_1^{@\frac18}\}\!\} \\ & & \ldots \end{array} \qquad (2)$$

in $\mathbb{D}^2\mathcal{X}$ whose pattern should be evident.

From Definition 5.1 we see that each of these hypers is an entropy refinement of the preceding: for example to get from Δ_2 to Δ_3 we first 'pre-split' Δ_2 into smaller pieces

$$\{\!\{\underset{\downarrow}{\delta_0^{@\frac18}}, \underbrace{\delta_0^{@\frac18}, \quad \delta_{1/2}^{@\frac18}}_{\text{merge}}, \underset{\downarrow}{\delta_{1/2}^{@\frac14}}, \underbrace{\delta_{1/2}^{@\frac18}, \quad \delta_1^{@\frac18}}_{\text{merge}}, \underset{\downarrow}{\delta_1^{@\frac18}}\}\!\}$$

and then merge the selected inners as explained above, that is

$$\delta_0^{@\frac18} + \delta_{1/2}^{@\frac18} \;=\; \delta_{1/4}^{@\frac14} \quad \text{and} \quad \delta_{1/2}^{@\frac18} + \delta_1^{@\frac18} \;=\; \delta_{3/4}^{@\frac14}$$

to give Δ_3 when we allow the un-merged distributions simply to carry through.[†]

[†] Using our formal definition Definition 5.1 introduces a super $\mathbf{\Delta}$ to mediate the entropy refinement $\Delta_2 \preceq \Delta_3$; it is given by

normalize the columns
to give hypers of $\mathbf{\Delta}$

$$\Delta_2 \left\{ \left| \begin{array}{c} \delta_0^{@\frac14} \\ \delta_{1/2}^{@\frac12} \\ \delta_1^{@\frac14} \end{array} \right| \right. \xleftarrow{\text{avg}} \left| \begin{array}{c||c||c||c||c} \delta_0^{@\frac18} & \delta_0^{@\frac18} & & & \\ & \delta_{1/2}^{@\frac18} & \delta_{1/2}^{@\frac14} & \delta_{1/2}^{@\frac18} & \\ & & & \delta_1^{@\frac18} & \delta_1^{@\frac18} \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \delta_0^{@\frac18} & \delta_{1/4}^{@\frac14} & \delta_{1/2}^{@\frac14} & \delta_{3/4}^{@\frac14} & \delta_1^{@\frac18} \end{array} \right| \;\; \text{map.avg}$$

$$\underbrace{\delta_0^{@\frac18} \;\; \delta_{1/4}^{@\frac14} \;\; \delta_{1/2}^{@\frac14} \;\; \delta_{3/4}^{@\frac14} \;\; \delta_1^{@\frac18}}_{\Delta_3}$$

Now, by symmetry any hyper Δ that was a refinement-limit of the chain (2) would have to be uniform (except possibly for the endpoints) but with a countably infinite support, since the supports of the chains' *elements* grow without bound – and uniform, infinite and discrete distributions do not exist. The actual limit of that refinement chain is in fact the *measure* over the distributions δ_p given by taking p uniformly from $[0, 1]$, and that is outside our space $\mathbb{D}^2\mathcal{X}$. Writing $\mathbb{M}$ for 'measure' (informally, i.e. without being specific about the sigma-algebra) we find our limit in $\mathbb{M}\mathbb{D}\mathcal{X}$ rather than $\mathbb{D}^2\mathcal{X}$.

6.2. *Dealing with the incompleteness: proper measures*

Pursuing the $\mathbb{M}\mathbb{D}\mathcal{X}$ strategy suggested above would lead us through steps like these:

1. Define a metric over $\mathbb{D}\mathcal{X}$, i.e. provide a distance function between (discrete) distributions. For reasons we explain in Section 12 we would choose the Kantorovich metric (Deng and Du 2009) which is advocated for this kind of application anyway (van Breugel 2005).
2. Generate the Borel algebra from the Kantorovich metric.
3. Define refinement between hypers that are proper measures, a generalization of the 'split/merge' of Definition 5.1 and explained in our previous work (McIver *et al.* 2010) for the discrete case.
4. Observe that the resulting, more general semantic space $\mathbb{D}\mathcal{X} \to \mathbb{M}\mathbb{D}\mathcal{X}$ allows (still) a monadic treatment of sequential composition.
5. Define the program semantics Section 8 in that more sophisticated space.

But we do not do that here. Instead, in this report we limit our extensions to *just what will suffice* for the quantitative security of iterative programs, including making refinement-based comparisons between them, as part of our general programme of expanding the scope of this approach to deal with realistic situations. In fact, we will see that refinement chains generated by the fixed-point definition of loops *do* have a refinement-sup in our space – that is, 'loop-approximant chains' are a strict subset of all possible refinement chains, and do not in particular contain examples like (2) above.

Thus we can remain within the space of discrete hypers $\mathbb{D}^2\mathcal{H}$, which allows a drastic simplification (compared with $\mathbb{M}\mathbb{D}\mathcal{H}$) of the presentation. How this is done is the topic of the next section: we will be using *partial* discrete distributions to represent nontermination, thus concentrating on a space $\mathcal{V}\times\mathbb{D}\mathcal{H} \to \underline{\mathbb{D}}(\mathcal{V}\times\mathbb{D}\mathcal{H})$ for denotations of programs.

7. Semantics for the HMM-interpretation of secure iterating programs

7.1. *Denotations of programs*

Here we give a precise construction of a semantic space, and the interpretation of a small programming language in it. The language includes probability, visible versus hidden variables and iteration.

The two merges of inners referred to in the main text occur in columns 2,4; the columns 1,3,5 are the inners that carry through unchanged from Δ_2 to Δ_3.

For *noninterference* we imagine a finite underlying state space of two parts, named $\mathcal{V}$ and $\mathcal{H}$ where $\mathcal{V}$ is the 'visible' part of the state and $\mathcal{H}$ is its 'hidden' part. Because the $\mathcal{H}$ part is hidden our underlying state space will not be simply the Cartesian product of those two components, but rather the set $\mathcal{S} := \mathcal{V} \times \mathbb{D}\mathcal{H}$ comprising the product of the visible part $\mathcal{V}$ (as is) and the *distributions* $\mathbb{D}\mathcal{H}$ over the hidden part.

For *nontermination* we consider program outputs to be of type $\underline{\mathbb{D}}\mathcal{S}$, that is $\underline{\mathbb{D}}(\mathcal{V} \times \mathbb{D}\mathcal{H})$, the *partial* distributions over $\mathcal{S}$ – this represents a slight generalization of the type suggested above for programs in that the partiality (the one-deficit) is used to describe the probability of the program's failing to terminate (Jones and Plotkin 1989; He *et al.* 1997; Morgan *et al.* 1996; McIver and Morgan 2005). As before, we call elements of $\underline{\mathbb{D}}\mathcal{S}$ hypers, referring if necessary to *partial* hypers when the distinction is important. Thus $\mathcal{S} \rightarrow \underline{\mathbb{D}}\mathcal{S}$, that is $\mathcal{V} \times \mathbb{D}\mathcal{H} \rightarrow \underline{\mathbb{D}}(\mathcal{V} \times \mathbb{D}\mathcal{H})$ is the type we propose for programs: from an initial state (v, δ) in $\mathcal{S}$ a program determines a partial distribution on $\mathcal{S}$, i.e. a distribution whose supports have structure (v', δ'), as its final output.

Recall from Section 3.4 that we write function application as $f.x$, with '.' associating to the left. Operators without their operands are written between parentheses, as $(\preceq)$ for example.[†]

7.2. *The entropy refinement order between programs*

As usual our orders on programs will be the pointwise orders on their results. Our first order is the *Entropy Refinement* set out at Definition 5.1, adapted to deal with partial hypers and to take the $\mathcal{V}$ portion of the state into account.

Definition 7.1 (entropy refinement (generalizing Definition 5.1)). Let the state space be $\mathcal{S} = \mathcal{V} \times \mathbb{D}\mathcal{H}$, with $\mathcal{V}, \mathcal{H}$ both finite, and define $Q : \mathcal{S} \rightarrow \mathbb{D}(\mathcal{V} \times \mathcal{H})$ with $Q.(v, \delta).(v', h')$ equal to $\delta.h'$ if $v = v'$, otherwise zero.[‡]

For two hypers $\Delta_{\{S,I\}} : \underline{\mathbb{D}}\mathcal{S}$, we say that Δ_S is *entropy refined* by Δ_I, writing $\Delta_S \preceq \Delta_I$, just when $\mathsf{map}.Q.\Delta_S \preceq \mathsf{map}.Q.\Delta_I$ according to our preliminary definition Definition 5.1 of entropy refinement, but taking our $\mathcal{V} \times \mathcal{H}$, here, all at once as just $\mathcal{X}$ there and generalizing $\mathsf{map}, \mathsf{avg}$ to partial distributions in the obvious way.

Like the preliminary definition, $(\preceq)$ defines a partial order on hypers. Note that a consequence of this definition is that entropy refinement does not change the distribution of the visible variables: that is if $\Delta_S \preceq \Delta_I$, then in fact $\overleftarrow{\Delta_S} = \overleftarrow{\Delta_I}$ where we recall that $\overleftarrow{\Delta}$ is the

[†] The latter (known as *sections* in functional programming) allows us easily to write expressions relating operators themselves, such as the succinct $(<) \subseteq (\leqslant)$ stating that less than is a subset of less-than-or-equals as a relation. Thus, the former 'dot' convention distinguishes function application from sections as well.
As a further example (though not needed in this report), as part of the definition of the Giry monad one defines *evaluation functions* E_B that, given a measure μ as argument, return μ applied to the measurable set B as the result. With sections and the 'dot' convention one writes directly $(.B)$ for this function: the well-established syntactical rules for sections then ensure that $E_B(\mu) = (.B).\mu = \mu.B$ automatically. A separate introduction, definition and explanation of the E_B notation is not necessary.

[‡] More succinctly, this is defining the product distribution $Q.(v, \delta) := \{\!\{v\}\!\} \times \delta$.

left-marginal distribution of the product distribution $\Delta{:}\mathbb{D}(\mathcal{V}\times\mathbb{D}\mathcal{H})$. Similarly, the *a priori* distribution of h associated with each value of v is left unchanged.

We now address the incompleteness issue raised in Section 6.1.

7.3. *The termination refinement order between programs*

We follow an approach that allows us to distinguish between chains produced by iteration and those produced by refinement more generally (Nelson 1989; Roscoe 1992): we use a stronger order for which our space *is* complete.

For a partial hyper $\Delta{:}\underline{\mathbb{D}}\mathcal{S}$, the probability that it terminates is just its total weight, written $\sum\Delta$; equivalently, the amount by which it fails to sum to 1 is its probability of nontermination. We define a partial order that allows increase of termination only, as follows:

Definition 7.2 (termination refinement). For $\Delta_{\{S,I\}}$ in $\underline{\mathbb{D}}\mathcal{S}$, we say that Δ_S is termination-refined by Δ_I, written $\Delta_S{\leqslant}\Delta_I$, just when for all $s=(v,\delta)$ in $\mathcal{S}$ we have $\Delta_S.s \leqslant \Delta_I.s$. This is simply the pointwise extension of $(\leqslant)$ on the real-valued probabilities.

Our space has a least element, and is trivially closed under sup-chains in this termination order, since the probabilities themselves are bounded above (by 1) and below (by 0). We will in due course show that the fixed-point definition of iteration generates termination chains, and so the completeness here will give us just the well definedness we need. That is, we will rely on

Lemma 7.1 (termination completeness). Let $\Delta_0{\leqslant}\Delta_1{\leqslant}\cdots$ be a $(\leqslant)$-chain of hypers in $\underline{\mathbb{D}}\mathcal{S}$. Then the chain has a $(\leqslant)$-least upper bound $\bigvee_i \Delta_i$ in $\underline{\mathbb{D}}\mathcal{S}$.

Proof. Completeness of $[0,1]$. □

Note that everywhere-terminating programs are maximal in this *cpo*.

7.4. *Secure refinement between programs*

The primary order of interest on our space, secure refinement, allows both entropy refinement and termination refinement:

Definition 7.3 (secure refinement). Given (partial) hypers $\Delta_{\{S,I\}}$ in $\underline{\mathbb{D}}\mathcal{S}$, we define *Secure Refinement* as the composition of the two other orders: first termination refinement, and then entropy refinement. We have $\Delta_S{\sqsubseteq}\Delta_I$ just when there is an intermediate hyper Δ such that $\Delta_S{\leqslant}\Delta$ and $\Delta{\preceq}\Delta_I$.

Observe trivially that $(\leqslant)$ is a strengthening of $(\sqsubseteq)$, by reflexivity of $(\preceq)$. Like termination and entropy refinement, secure refinement is a partial order on hypers. Reflexivity holds trivially from that of $(\leqslant)$ and $(\preceq)$. The transitivity of $(\sqsubseteq)$ follows from transitivity of the two other orders, plus the fact that $(\sqsubseteq) \supseteq (\leqslant){\circ}(\preceq)$. For antisymmetry we reason that if $A \sqsubseteq C$ and $C \sqsubseteq A$ then there must exist a B and D such that $A \leqslant A{+}B \preceq C$ and $C \leqslant C{+}D \preceq A$. From reflexivity of $(\leqslant)$ and transitivity of $(\sqsubseteq)$ we then have that both

$A{+}B \sqsubseteq A$ and $C{+}D \sqsubseteq C$, and thus both C and D must be zero since $(\sqsubseteq)$ cannot decrease the overall weight of a hyper. From this we have that $A{\preceq}C$ and $C{\preceq}A$, hence $A = C$ by antisymmetry of $(\preceq)$.

The definition of program refinement is the pointwise extension of the above, that is

Definition 7.4 (secure program refinement). Let S, I be programs' meanings of type $\mathcal{S}{\to}\mathbb{D}\mathcal{S}$. We say that $S{\sqsubseteq}I$ just when for all initial states $s{:}\mathcal{S}$ we have $S.s \sqsubseteq I.s$ according to Definition 7.3.

7.5. *Least fixed points in $\mathcal{S}{\to}\mathbb{D}\mathcal{S}$: getting around incompleteness*

The normal approach to fixed-point semantics for loops would be to show that a loop defines a $(\sqsubseteq)$-continuous functional $\mathcal{L}$ over the program space $\mathcal{S}{\to}\mathbb{D}\mathcal{S}$, and then to take the $(\sqsubseteq)$-supremum of the chain $\amalg \sqsubseteq \mathcal{L}.\amalg \sqsubseteq \mathcal{L}^2.\amalg\cdots$ where $\amalg$ is the least program, the one producing the output hyper of zero weight for all inputs.

Here instead we show that a loop defines a $(\leqslant)$-continuous functional $\mathcal{L}$, and then take the $(\leqslant)$-supremum of the chain $\amalg \leqslant \mathcal{L}.\amalg \leqslant \mathcal{L}^2.\amalg\ldots$. Its well definedness follows from Lemma 7.1; its relevance is justified by the following lemma.

Lemma 7.2 (equivalence of fixed points). Let partial orders $(\leqslant)$ and $(\sqsubseteq)$ be defined over some space $\mathcal{X}$, and let $\mathcal{L}$ be an endofunction on $\mathcal{X}$. Suppose further that $(\leqslant) \subseteq (\sqsubseteq)$, that is that $(\leqslant)$ implies $(\sqsubseteq)$.

If a $(\leqslant)$-least (resp. greatest) fixed point of $\mathcal{L}$ exists, then also a $(\sqsubseteq)$-least (resp. greatest) fixed point of $\mathcal{L}$ exists, and in fact they are equal.

Proof. Let x be the $(\leqslant)$-least fixed point of $\mathcal{L}$. Then for any (other) fixed-point x' of $\mathcal{L}$ we have $x{\leqslant}x'$ and so – by assumption – also $x{\sqsubseteq}x'$. Thus x is a $(\sqsubseteq)$-lower-bound for all fixed points; but it is a fixed point itself. Therefore it is the $(\sqsubseteq)$-least fixed point as well. (The same argument holds for greatest.) □

In the next section, we will introduce our language to express and reason about secure programming, extending our previous work with iteration. We use Definition 7.2 for the semantics for loops, relying on Lemma 7.2 to ensure that it is also well defined as a least fixed point in the security order; we do, of course, need to show that the assumption of $(\leqslant)$-continuity is satisfied by the semantic definitions we give. In the conclusion, we shall return to the question of $(\sqsubseteq)$-limits more generally, i.e. those which are not restricted to $(\leqslant)$-limits.

8. Programing language

Having tied-down the details of our semantic space, we can now give our programs' denotations via structural induction; however there are two potential sources of complexity in what we present. The first, conceptual, is the two-level structure that we motivated in the sections above, the partial distributions that themselves are taken over *other* conditional, or sometimes even *a posteriori* distributions.

The second is notational: standard constructions like conditionals and push-forward are now generated by program fragments that, as a rule, are expressions over free variables (i.e. the variables of the program) rather than (pure) mathematical functions themselves. This leads to uncomfortable expositions like ‘$\Pr(D|E)$ where distribution $D(x)$ is given by $\cdots x \cdots$ and predicate $E(x)$ holds just when $\cdots x \cdots$’. Although these are easy to understand (being well-established notations), they are hard to manipulate algebraically in specific cases where D, E are determined by some computer program.

We now introduce specialized notation to streamline our semantic definitions.

8.1. *Distribution comprehensions*

Recall that the *support* $\lceil\delta\rceil$ of distribution $\delta{:}\underline{\mathbb{D}}\mathcal{X}$ is those elements $x{:}\mathcal{X}$ with $\delta.x{\neq}0$; naturally for $\delta{:}\underline{\mathbb{D}}\mathcal{X}$ we have $\lceil\delta\rceil{\subseteq}\mathcal{X}$. The *weight* of δ is written $\sum\delta$, defined $(\sum x{:}\mathcal{X} \bullet \delta.x)$ so that full distributions have weight 1. Distributions can be scaled and summed according to the usual pointwise extension of multiplication and addition to real-valued functions, provided the outcomes are again distributions.

Given a non-empty finite set $\mathcal{X}$ we write $\lfloor\mathcal{X}\rfloor$ for the uniform distribution over $\mathcal{X}$, that is the uniform distribution $\delta{:}\mathbb{D}\mathcal{X}$ such that $\lceil\delta\rceil{=}\mathcal{X}$.

8.1.1. *Enumerated distributions and expected values.*

These are notations for enumerated distributions, i.e. those in which the support is explicitly listed (cf. set enumerations that list a set's elements):

- **empty.** The empty, or zero subdistribution has empty support and assigns probability zero to all elements: we write it $\{\!\!\{\}\!\!\}$.
- **multiple.** We write $\{\!\!\{x^{@p}, y^{@q}, \ldots, z^{@r}\}\!\!\}$ for the distribution assigning probabilities $p, q, \ldots, r$ to elements $x, y, \ldots, z$ respectively, with $p{+}q{+}\cdots{+}r \leqslant 1$. Provided $p, q, \ldots, r > 0$, the support is therefore the set $\{x, y, \ldots, z\}$.
- **point.** The distribution concentrated on a single element x is written $\{\!\!\{x\}\!\!\}$, i.e. abbreviating $\{\!\!\{x^{@1}\}\!\!\}$ whose support is $\{x\}$.
- **uniform.** When explicit probabilities are omitted they are implicitly uniform: thus $\{\!\!\{x, y, z\}\!\!\}$ is $\{\!\!\{x^{@\frac{1}{3}}, y^{@\frac{1}{3}}, z^{@\frac{1}{3}}\}\!\!\}$.
- **binary, and distributed uniform.** For a two-element distribution we write $x\ {}_p{\oplus}\ y$ for $\{\!\!\{x^{@p}, y^{@1-p}\}\!\!\}$, and in the uniform case we can write $x{\oplus}y{\oplus}\cdots{\oplus}z$ for $\{\!\!\{x, y, \ldots, z\}\!\!\}$.

For expected values of random variables that are written as expressions, we have

- **expected value.** We write $(\mathcal{E}\, d{:}\delta \bullet E)$ for the *expected value* $\sum_{d:\lceil\delta\rceil}(\delta.d{\times}E)$ of expression E, interpreted as a random variable in d, over distribution δ.
  If E is Boolean, then it is taken to be 1 if E holds and 0 otherwise, so that the expected value is then just the combined probability in δ of all elements d satisfying E. If necessary for clarity we will write $[E]$ to indicate E's conversion from Boolean to $0, 1$; when possible, however, we omit it (to reduce proliferation of brackets).

8.1.2. *Distribution comprehensions, conditioning and* a posteriori *values.*

As for set comprehensions, with distribution comprehensions we describe a distribution by giving a rule for forming it, i.e. its supporting elements and the probabilities they have. Here are the common cases:

- **map, push-forward.** When f in Section 3.4 is given as an expression E of type $\mathcal{Y}$, with free variable x say, then for the push-forward distribution $\mathsf{map}.f.\delta$ we write the comprehension $\{\!\!\{ x{:}\delta \bullet E \}\!\!\}$ where for $y{:}\mathcal{Y}$ we define
$$\{\!\!\{ x{:}\delta \bullet E \}\!\!\}.y := (\mathcal{E}\, x{:}\delta \bullet E{=}y).$$
Recall from above that the Boolean value $E{=}y$ is to be converted implicitly to 0, 1 in this case.
- **conditional distribution.** Given a distribution $\delta{:}\underline{\mathbb{D}}\mathcal{X}$ and a Boolean expression R in free variable x, we write $\{\!\!\{ x{:}\delta \mid R \}\!\!\}$ for the distribution obtained by conditioning δ on the set (the event) that R represents as a predicate in x. Thus for $x'{:}\mathcal{X}$ we have
$$\{\!\!\{ x{:}\delta \mid R \}\!\!\}.x' := \delta.x' \times [R'] \;/\; (\mathcal{E}\, x{:}\delta \bullet [R]), \tag{3}$$
where R' is R with x replaced by x' and here, for clarity, with $[\cdot]$ we make the conversions to 0, 1 explicit.
- ***a posteriori* values.** Finally, for *Bayesian belief revision* suppose δ is an *a priori* distribution over some $\mathcal{X}$ and let expression R (not Boolean) in free variable x in $\mathcal{X}$ be the probability of a certain observable result if that x is chosen. Then $\{\!\!\{ x{:}\delta \mid R \}\!\!\}$ is the *a posteriori* distribution (revising δ) when that result actually occurs. The definition is as for (3) immediately above, but using just R rather than $[R]$.
Note that R can be scaled without affecting the value of this expression, so wlog it can be made one-summing as x varies: this makes it easier to interpret as a probabilistic outcome that triggers Bayesian belief revision.
- **general distribution comprehension.** We can combine all the above possibilities by writing $\{\!\!\{ x{:}\delta \mid R \bullet E \}\!\!\}$, for distribution δ, real expression R (in x) and expression E (also in x) to mean
$$(\mathcal{E}\, x{:}\delta \bullet R{\times}\{\!\!\{E\}\!\!\}) \;/\; (\mathcal{E}\, x{:}\delta \bullet R) \tag{4}$$
where, first, an expected value is formed in the numerator by scaling and adding point-distribution $\{\!\!\{E\}\!\!\}$ as a real-valued function: this gives another (sub-)distribution. The scalar denominator then conditions on R.
A missing E is implicitly x itself. If R is omitted, then $(R\times)$ is removed from the numerator, and the denominator is removed altogether. (When δ is a full distribution, this happens automatically by assuming a missing R to be 1, or equivalently Boolean *true*.)

As a concrete example we recall the puzzle

> In families with two children of equally and independently distributed gender, if one child is a boy what is the chance that the other is too?

Encoding *boy*, *girl* as Booleans T, F we write $\{\!\!\{ x, y{:}\mathsf{T}{\oplus}\mathsf{F} \mid x{\vee}y \bullet x{\wedge}y \}\!\!\}$ for the distribution of the pushed-forward function *both boys* $(x{\wedge}y)$ over the *iid gender joint-distribution* of the two children $(x, y{:}\mathsf{T}{\oplus}\mathsf{F})$ conditioned on the event *at least one boy* $(x{\vee}y)$. It works out as

$$
\begin{aligned}
& \{\!\!\{ x,y{:}\mathsf{T}{\oplus}\mathsf{F} \mid x{\vee}y \bullet x{\wedge}y \}\!\!\} \\
= \;& (\mathcal{E}\, x,y{:}\mathsf{T}{\oplus}\mathsf{F} \bullet [x{\vee}y]{\times}\{\!\!\{x{\wedge}y\}\!\!\}) \;/\; (\mathcal{E}\, x,y{:}\mathsf{T}{\oplus}\mathsf{F} \bullet [x{\vee}y]) \\
= \;& (1{\times}\{\!\!\{\mathsf{T}\}\!\!\}/4 + 1{\times}\{\!\!\{\mathsf{F}\}\!\!\}/4 + 1{\times}\{\!\!\{\mathsf{F}\}\!\!\}/4 + 0{\times}\{\!\!\{\mathsf{F}\}\!\!\}/4) \;/\; (3/4) \\
= \;& \{\!\!\{\mathsf{T}^{@\frac{1}{4}}, \mathsf{F}^{@\frac{1}{2}}\}\!\!\} \;/\; (3/4) \\
= \;& \mathsf{T}\;{}_{1/3}{\oplus}\;\mathsf{F},
\end{aligned}
$$

that is (as we know) that the *a posteriori* probability of 'both boys' is 1/3. This is the kind of calculation that specific programs' semantics generate.

8.2. *Program semantics for the HMM core: revelations*

We recall from Section 3.1 that a HMM is determined by two stochastic kernels (matrices) $\boldsymbol{T},\boldsymbol{E}$. In programming terms the $\boldsymbol{T}$ represents a probabilistic assignment to our hidden variable h; we deal with that at *Choose prob. hidden* in Section 8.3 below.

The $\boldsymbol{E}$ on the other hand releases information (about h) in what we call a 'revelation' – observables our attacker can see McIver and Morgan (2008). It has two forms, the second a generalization of the first.

In the two definitions below, and further, we write E.v.h as an *expression* in which program variables v, h might occur free. The same convention applies to D, G, p for distributions, (Boolean) guards and probabilities resp.

Program type	Program text P	Semantics $[\![P\,]\!].(v,\delta)$
Reveal value	**reveal** E.v.h	$\{\!\!\{h{:}\delta \bullet (v, \{\!\!\{h'{:}\delta \mid E.v.h'{=}E.v.h\}\!\!\})\}\!\!\}$
	Expression E.v.h takes its value in some type $\mathcal{X}$ representing observations an attacker can make. The command reveals a value x depending on v, h. Neither v nor h is changed by this; but the outgoing distribution of the hidden h is conditioned on the basis of the x revealed. Note that x is not stored; but because of perfect recall an attacker can remember it.	
Reveal choice	**reveal** D.v.h	$\{\!\!\{h{:}\delta; x{:}D.v.h \bullet (v, \{\!\!\{h{:}\delta \mid D.v.h.x\}\!\!\})\}\!\!\}$
	Expression D.v.h is now more generally of type $\mathbb{D}\mathcal{X}$, so that for $x{:}\mathcal{X}$ we have D.v.h.x as a probability. The command calculates that distribution, and then chooses some value x according to those probabilities; that value x is then revealed. As before, variables v, h are not changed; but the distribution of h is conditioned on the fact that x was revealed. *Reveal value* is the special case **reveal** $\{\!\!\{E$.v.h$\}\!\!\}$ of *Reveal choice*.	

8.3. *Semantics of syntactically atomic commands*

Syntactically atomic commands are regarded as *semantically* atomic in the sense that the only information they leak is what the final value of the visible v allows to be deduced about the final value of h with knowledge of the program text. Thus for example v:= h leaks everything about h, since v's final value is evidently the same as h's; yet v:= 0×h reveals nothing, even though at some point in an internal register the value of h might have been accessible. In this sense, the syntactic atoms are the atoms of observation also: within them neither perfect recall nor implicit flow make sense.

We determine the semantics of these atomic commands systematically. Using 'classical', i.e. without-noninterference probabilistic sequential semantics (Kozen 1985, etc.) gives a straightforward meaning to atomic commands' actions on a state space $\mathcal{S}$ as functions of type $\mathcal{S}{\to}\underline{\mathbb{D}}\mathcal{S}$ taking an initial distribution to a (sub-)distribution of final states. If we abstract from noninterference properties by considering v to be hidden (as well as h), and set $\mathcal{S}{:=}\mathcal{V}{\times}\mathcal{H}$ then we have a ready-made classical semantics for the syntactic atoms we are dealing with here.

The initial 'state' will be a pair (v,δ) in $\mathcal{V}{\times}\mathbb{D}\mathcal{H}$. We therefore reuse '$Q$' from Definition 7.1 to express this as the joint distribution $\{\!\!\{v\}\!\!\}{\times}\delta$ of type $\mathbb{D}(\mathcal{V}{\times}\mathcal{H})$, that is $\mathbb{D}\mathcal{S}$. To apply a command with semantics of type $\mathcal{S}{\to}\underline{\mathbb{D}}\mathcal{S}$ to that, we use lifting (Section 3.4) so that the result of this classical interpretation is again of type $\underline{\mathbb{D}}(\mathcal{V}{\times}\mathcal{H})$, and we convert this back to the noninterference output-type $\underline{\mathbb{D}}(\mathcal{V}{\times}\mathbb{D}\mathcal{H})$ by analogy with 'revealing v' according to the semantics above – since knowledge of v's final value is all that escapes an atomic command. Following *Reveal value* from above, we define

$$\mathsf{rv}.\Delta := \{\!\!\{\ (v,h){:}\,\Delta \bullet (v, \{\!\!\{(v',h'){:}\,\Delta \mid v{=}v' \bullet h'\}\!\!\})\ \}\!\!\}.^{\dagger} \tag{5}$$

The result of the procedure above – convert incoming $\mathcal{V}{\times}\mathbb{D}\mathcal{H}$ to $\mathbb{D}(\mathcal{V}{\times}\mathcal{H})$, then apply lifted classical semantics; then apply rv to the result – is summarized below. Observe that neither program **abort**, nor assertions are necessarily useful for writing specific programs, but our focus is on reasoning *about* programs, in particular algebraically, and for that these commands play a prominent role.

Program type[‡]	Program text P	Semantics $[\![P\,]\!].(v,\delta)$
Least element	**abort**	$\{\!\!\{\}\!\!\}$
	This is the program that simply fails to terminate: for every input it produces the empty subdistribution as output. In our refinement order, as a specification it allows all possible implementations (i.e. that **abort** $\sqsubseteq S$ for all S) – essentially playing the role of '0' in arithmetic.	
Identity	**skip**	$\{\!\!\{(v,\delta)\}\!\!\}$
	The 'do nothing' command simply converts its input to a point-hyper on output, i.e. reproduces its input with probability one.	
Assertion	$\{p.\mathsf{v}.\mathsf{h}\}$	$\{\!\!\{(v,\{\!\!\{h{:}\,\delta \mid p.v.h\}\!\!\})^{@(\mathcal{E}\,h'{:}\,\delta\,\bullet\,p.v.h')}\}\!\!\}$

[†] We justify (5) informally by noting that it is what results from replacing hidden h in the *rhs* of *Reveal value* by the hidden pair (v,h) and considering the expression $E.\mathsf{v}.\mathsf{h}$ to be simply v.

[‡] The most general form of atomic assignment is the *Simultaneous choice* mentioned earlier, whose semantics can be deduced as for the others from its classical behaviour. Since it is seldom needed, however, we omit its definition for brevity.

An assertion gives directly in $p.\mathsf{v}.\mathsf{h}$ the probability of the command's termination. With probability $1{-}p$ the assertion behaves as **abort**.
When with probability p it *does* terminate, however, it conditions the hidden value's distribution δ on the fact it did so: that is δ is revised to reflect that the abort did not occur. The visible variable v is unaffected in this case.

Assign to visible $\mathsf{v}{:=}E.\mathsf{v}.\mathsf{h}$ $\quad \{\!\!\{\ h{:}\delta \bullet (E.v.h, \{\!\!\{h'{:}\delta \mid E.v.h'{=}E.v.h\}\!\!\})\ \}\!\!\}$

The command's effect is to assign the *rhs*-value to v *but also* to condition the hidden distribution on the fact that h can produce the value observed to have been put into v.

Assign to hidden $\mathsf{h}{:=}E.\mathsf{v}.\mathsf{h}$ $\quad \{\!\!\{\ (v, \{\!\!\{h{:}\delta \bullet E.v.h\}\!\!\})\ \}\!\!\}$

The command does not change v, but maps the hidden incoming distribution of h through $E.\mathsf{v}$ considered as a function of (incoming) h to produce the resulting distribution on (outgoing) h.

Choose prob. visible $\mathsf{v}{:\in}D.\mathsf{v}.\mathsf{h}$

$\{\!\!\{\ v'{:}(\mathcal{E}\, h{:}\delta \bullet D.v.h) \bullet (v', \{\!\!\{h'{:}\delta \mid D.v.h'.v'\}\!\!\})\ \}\!\!\}$

Expression $D.\mathsf{v}.\mathsf{h}$ is a *distribution* on $\mathcal{V}$, and the choice of v's new value is made according to it. It generalizes *Assign to visible*, since the latter can be written $\mathsf{v}{:\in}\{\!\!\{E.\mathsf{v}.\mathsf{h}\}\!\!\}$.

Choose prob. hidden $\mathsf{h}{:\in}D.\mathsf{v}.\mathsf{h}$ $\quad \{\!\!\{\ (v, (\mathcal{E}\, h{:}\delta \bullet D.v.h))\ \}\!\!\}$

Expression $D.\mathsf{v}.\mathsf{h}$ is now a distribution on $\mathcal{H}$, and the choice of h's new value is made according to it. It generalizes *Assign to hidden*, since the latter can be written $\mathsf{h}{:\in}\{\!\!\{E.\mathsf{v}.\mathsf{h}\}\!\!\}$.

As a syntactic convenience, when we are using the more general 'choose' form of either command, but the *rhs*'s distribution is written out using ($\oplus$) rather than as a $\{\!\!\{\}\!\!\}$-style comprehension, we use the conventional assignment symbol ($:=$) so that e.g. we can write $\mathsf{v}{:=}\mathsf{T}{\oplus}\mathsf{F}$ for flipping a fair Boolean coin.

As an example of the algebraic utility of *Assertion*, we note that distinguished commands **abort** and **skip** are special cases of assertions, so that **skip** $=\{\mathsf{T}\}$ and **abort** $=\{\mathsf{F}\}$. Further, the semantics of *Reveal choice* can be given more compactly – assuming $D.\mathsf{v}.\mathsf{h}$ has type $\mathbb{D}\mathcal{X}$ – as

$$[\![\textbf{reveal } D.\mathsf{v}.\mathsf{h}]\!].(v,\delta) = (\textstyle\sum x{:}\mathcal{X} \bullet [\![\{D.\mathsf{v}.\mathsf{h}.x\}]\!].(v,\delta)). \tag{6}$$

That formulation makes it easy to reason about revelations in terms of more primitive commands. We also have that assignments to visible variables that may depend on h may be represented more simply in terms of those that do not:

$$[\![\mathsf{v}{:\in}D.\mathsf{v}.\mathsf{h}]\!].(v,\delta) = (\textstyle\sum v'{:}\mathcal{V} \bullet [\![\{D.\mathsf{v}.\mathsf{h}.v'\};\mathsf{v}{:=}v']\!]). \tag{7}$$

As we will see in the next section, assertions also play an important role in the specification of probabilistic choice and conditionals.

8.4. *Semantics of compound commands: implicit flow*

Compound commands are in fact the simplest to define, since they are treated almost as they would be for classical semantics. The only adjustment is to insert conditioning assertions on program branch-points to enforce *implicit flow*, that is that information escapes by observation of the outcome of conditionals.

Program type	Program text P	Semantics $[\![P]\!].(v,\delta)$
Composition	$P_1 ; P_2$	$[\![P_2]\!]^*.([\![P_1]\!].(v,\delta))$
	Sequential composition is interpreted as Kleisli composition (Section 3.4).	
General prob. choice	$P_L \,{}_{p.\mathsf{v}.\mathsf{h}}\!\oplus P_R$	$[\![\{p.\mathsf{v}.\mathsf{h}\} ; P_L]\!].(v,\delta) + [\![\{1{-}p.\mathsf{v}.\mathsf{h}\} ; P_R]\!].(v,\delta)$
	Expression $p.\mathsf{v}.\mathsf{h}$ is evaluated to a probability of the command's taking its left branch; otherwise it takes the right. The attacker can observe which branch was taken: this is reflected in the conditioning assertions at the beginning of each branch.	
Conditional choice	**if** $G.\mathsf{v}.\mathsf{h}$ **then** P_T **else** P_F **fi**	$[\![\{G.\mathsf{v}.\mathsf{h}\} ; P_T]\!].(v,\delta) + [\![\{\neg G.\mathsf{v}.\mathsf{h}\} ; P_F]\!].(v,\delta)$
	This is a specialization of the previous *General probabilistic choice* to the case where the probability is always either 1 (go left) or 0 (go right). Again the conditioning assertions guard each branch.	
Iteration	**while** $p.\mathsf{v}.\mathsf{h}$ **do** P **od**	the ($\leqslant$)-least fixed point of $\mathcal{L}$,
	applied to (v,δ), where $\mathcal{L}$ is the unique endofunction on the space $\mathcal{S}{\rightarrow}\underline{\mathbb{D}}\mathcal{S}$ of programs' meanings such that for any program L we have $[\![P ; L_{p.\mathsf{v}.\mathsf{h}}\!\oplus\mathbf{skip}]\!] = \mathcal{L}.[\![L]\!]$. As for *Conditional choice*, the loop guard is a probability determined by the program variables v,h, with as a special case Booleans T,F interpreted as 1 (enter the loop) or 0 (terminate the loop).	

For iteration we are taking the usual least-fixed-point approach except, for the reasons explained above, we use a special *termination order* ($\leqslant$) for the chain of iterates. For this we need the (usual) technical results of continuity of our program contexts.

Lemma 8.1 (continuity of program contexts). Any context $\mathcal{C}(\cdot)$, constructed in the programming language above, satisfies $\mathcal{C}(\bigvee_i P_i) = \bigvee_i \mathcal{C}(P_i)$ for non-empty ($\leqslant$)-chains $\bigvee_i P_i$.

Proof. Because the termination order is so simple (unlike the entropy order), being essentially pointwise less-than-or-equals, this result can be verified by structural induction using the fact that scalar multiplication and countable addition are continuous over non-empty $\leqslant$-chains in the real interval $[0,1]$. For example, to show continuity of sequential composition in its left argument, we can reason that if $[\![P]\!] = \bigvee_i [\![P_I]\!]$, then

$$(\textstyle\bigvee_i [\![P_i ; R]\!]).(v,\delta)$$

$$
\begin{array}{ll}
= (\bigvee_i [\![P_i\,; R]\!].(v,\delta)) & \text{“limits defined pointwise”} \\
= (\bigvee_i (\sum(v',\delta') \colon \lceil [\![R]\!].(v,\delta)\rceil \bullet [\![R]\!].(v,\delta).(v',\delta') * [\![P_i]\!].(v',\delta'))) & \text{“composition”} \\
= & \text{“}\leqslant\text{-continuity of countable addition and scalar multiplication”} \\
\quad (\sum(v',\delta') \colon \lceil [\![R]\!].(v,\delta)\rceil \bullet [\![R]\!].(v,\delta).(v',\delta') * (\bigvee_i [\![P_i]\!].(v',\delta'))) & \\
= (\sum(v',\delta') \colon \lceil [\![R]\!].(v,\delta)\rceil \bullet [\![R]\!].(v,\delta).(v',\delta') * ([\![P]\!].(v',\delta'))) & \text{“}[\![P]\!] = \bigvee_i [\![P_i]\!]\text{”} \\
= (\bigvee_i [\![P\,; R]\!]).(v,\delta). & \text{“composition”}
\end{array}
$$

Continuity of probabilistic and conditional choices in either argument are equally straightforward. To show that $[\![\textbf{while } p.\mathsf{v}.\mathsf{h} \textbf{ do } P \textbf{ od}]\!] = \bigvee_i [\![\textbf{while } p.\mathsf{v}.\mathsf{h} \textbf{ do } P_i \textbf{ od}]\!]$ in the case $[\![P]\!] = \bigvee_i [\![P_i]\!]$, we reason that

$$[\![\textbf{while } p.\mathsf{v}.\mathsf{h} \textbf{ do } P \textbf{ od}]\!] = \bigvee_j f^j.[\![\textbf{abort}]\!] \qquad \text{for } f.[\![X]\!] := [\![P\,; X \;{}_{p.\mathsf{v}.\mathsf{h}}{\oplus}\; \textbf{skip}]\!]$$

by continuity of function f (which follows inductively by continuity of sequential composition and probabilistic choice and the fact that $\leqslant$ is a cpo in our program semantics). Subsequently it is enough to show by induction that each $f^j.[\![\textbf{abort}]\!]$ is continuous in P, and for this continuity of sequential composition and probabilistic choice again suffices. □

Importantly, each of our compound operators are monotonic with respect to their arguments and the secure refinement order ($\sqsubseteq$), meaning that we may reason compositionally about the correctness of programs.

We note that the infinitary definition of loop semantics can produce a result hyper of countably infinite support: an example is the program whose iteration has probability 1/2 of completing on every instance but, if it does not, releases another morsel of information about hidden h. It is

$$
\begin{array}{l}
\textbf{hid } \mathsf{h}\colon \{0,1,2\}; \\
\quad \mathsf{h} := 0 \oplus 1 \oplus 2; \\
\quad \textbf{while } 1/2 \textbf{ do reveal } A \;{}_{h/2}{\oplus}\; B \textbf{ od},
\end{array}
$$

where A, B are arbitrary distinct constants.

Theorem 8.1 (monotonicity of compound commands). Each of the commands listed above is monotonic with respect to their program arguments and the refinement order.

8.5. *Local and multiple variables; hidden correlations*

To this point we have had just two variables, visible v and hidden h, and have been assuming for simplicity that they are all the variables in the program. In practice however each of $\mathcal{V}, \mathcal{H}$ will each comprise many variables, represented in the usual Cartesian way. Thus, if we have variables $\mathsf{a}{:}\mathcal{A}, \mathsf{b}{:}\mathcal{B}, \mathsf{c}{:}\mathcal{C}, \mathsf{d}{:}\mathcal{D}$ with the first two a, b visible and the last two c, d hidden, then $\mathcal{V}$ is $\mathcal{A}\times\mathcal{B}$ and $\mathcal{H}$ is $\mathcal{C}\times\mathcal{D}$ so that the state space is $\mathcal{A}\times\mathcal{B}\times\mathbb{D}(\mathcal{C}\times\mathcal{D})$. Assignments and projections are handled as normal.

Thus we allow local variables, both visible and hidden, which extend the state as described above: within the scope of a visible local-variable declaration $|\![\![$ **vis** $\mathsf{x}\!:\!\mathcal{X}\cdots]\!]\!|$, the $\mathcal{V}_{\text{local}}$ used is $\mathcal{X}\times\mathcal{V}_{\text{global}}$. Hidden variables are similar.[†]

Note however that because for simplicity we have been assuming that v,h are in fact *all* the variables in the program, i.e. that they stand for vectors of variables implicitly, our semantics above establishes the equality of the two fragments $\mathsf{v}\!:=\mathsf{h};\ \mathsf{v},\mathsf{h}\!:=0,0$ and $\mathsf{v},\mathsf{h}\!:=0,0$, reflecting our deliberate concentration on h's *final* value (Morgan 2006, 2009) in order to extend conventional refinement (Morgan 1994; Back and von Wright 1998) that does the same. In this case h's initial value's being revealed on the left has no bearing on our knowledge or ignorance of its final value and so does not introduce a difference in meaning between the two fragments shown.

If however there are other hidden variables, not mentioned but still in scope as might happen within a local block or within the context of extra declarations, then our semantics must be slightly more general, in particular recognizing that the v or h appearing on the left of an assignment is just one component of a vector of visible resp. hidden variables.

Technically this is handled by extending our hidden distribution to type $\mathbb{D}\mathcal{H}^2$, which tracks correlations with initial values. For simplicity we do not do that here, since in fact any program in which hiddens are not assigned-to (as in our examples and case studies) can be treated with the simpler $\mathbb{D}\mathcal{H}$-style semantics.

9. Algebra of HMM-style programs

The programming language introduced in Section 8, interpreted over the hyper-based semantics, admits a program algebra allowing the proof of general refinements between programs. In this section we present some of the foundational laws of this program algebra, which are then illustrated in Sections 9.8 and 10, via an example based on password guessing.

9.1. *General principles and scoping laws; referential transparency*

As for classical programs, it is possible to replace expressions by other expressions of equal value in context so that, for example, referential transparency gives

$$\mathsf{v}\!:=E.\mathsf{v}.\mathsf{h};\{\mathsf{T}\} \;=\; \mathsf{v}\!:=E.\mathsf{v}.\mathsf{h};\{\mathsf{v}{=}E.\mathsf{v}.\mathsf{h}\}.$$

It is also possible to move program fragments in and out of local scopes provided variable bindings are respected. Since empty scopes are equivalent to **skip**, i.e.

$$\textbf{skip} \;=\; |\![\![\ \textbf{vis}\ \mathsf{v}'\!:\!\mathcal{V}\ ]\!]\!| \;=\; |\![\![\ \textbf{hid}\ \mathsf{h}'\!:\!\mathcal{H}\ ]\!]\!|, \tag{8}$$

it is possible to introduce fresh variables of any constant type. We may also introduce assignments to scope-terminated variables as long as they do not reveal information about

[†] Implicitly local variables are assumed to be initialized by a uniform choice over their finite state space. In our examples however, we always initialize local variables explicitly, to avoid confusion.

the hidden state:

$$|\![\![\;\mathbf{vis}\; \mathsf{v}'\!:\!\mathcal{V};\;\cdots\;]\!]\!| \;=\; |\![\![\;\mathbf{vis}\; \mathsf{v}'\!:\!\mathcal{V};\;\cdots\;;\mathsf{v}'\!:\in D.\mathsf{v}.\mathsf{v}'\;]\!]\!|, \tag{9}$$

$$|\![\![\;\mathbf{hid}\; \mathsf{h}'\!:\!\mathcal{H};\;\cdots\;]\!]\!| \;=\; |\![\![\;\mathbf{hid}\; \mathsf{h}'\!:\!\mathcal{H};\;\cdots\;;\mathsf{h}'\!:\in D.\mathsf{v}.\mathsf{h}.\mathsf{h}'\;]\!]\!|. \tag{10}$$

As an example of the interaction of local scopes and visibility we have

$$\begin{array}{ll}
& [\![\; \mathbf{reveal}\; D.\mathsf{v}.\mathsf{h}\;]\!] \\
= & (\sum v'\!:\!\mathcal{X} \bullet [\![\{D.\mathsf{v}.\mathsf{h}.v'\}]\!]) \qquad \text{“represent revelation using assertions (6)”} \\
= & (\sum v'\!:\!\mathcal{X} \bullet [\![\; |\![\![\;\mathbf{vis}\; \mathsf{v}'\!:\!\mathcal{X}; \{D.\mathsf{v}.\mathsf{h}.v'\}; \mathsf{v}'\!:=\!v'\;]\!]\!|\;]\!]) \qquad \text{“introduce fresh variable terminated by a secure assignment”} \\
= & [\![\; |\![\![\;\mathbf{vis}\; \mathsf{v}'\!:\!\mathcal{X};\; \mathsf{v}'\!:\in D.\mathsf{v}.\mathsf{h}\;]\!]\!|\;]\!], \qquad \text{“shift scope and represent visible assignment using assertions (7)”}
\end{array}$$

i.e. that a revelation is effectively an assignment to a temporary visible variable: because of perfect recall, the revealed value is not forgotten; but because the temporary variable is declared within a block, it is effectively erased.

9.2. *Assertions*

We present here some basic properties of assertions that will be used to justify algebraic laws for more complex statements such as revelations and probabilistic choices. First, we have that assertions satisfy the following equivalence,

$$\{p_1.\mathsf{v}.\mathsf{h}\};\{p_2.\mathsf{v}.\mathsf{h}\} \;=\; \{p_1.\mathsf{v}.\mathsf{h} \times p_2.\mathsf{v}.\mathsf{h}\} = \{p_2.\mathsf{v}.\mathsf{h}\};\{p_1.\mathsf{v}.\mathsf{h}\} \tag{11}$$

and are thus commutative under sequential composition. Constant assertions also commute over arbitrary programs, so that

$$\{p\};S \;=\; S;\{p\}. \tag{12}$$

Since assertions referring to h may condition the hidden state, from the definition of secure refinement (Definition 7.3) we have

$$(\sum n \bullet [\![\{p_n.\mathsf{v}.\mathsf{h}\}]\!]) \;\sqsubseteq\; [\![\{\sum n \bullet p_n.\mathsf{v}.\mathsf{h}\}]\!], \tag{13}$$

for $(\sum n \bullet p_n.\mathsf{v}.\mathsf{h}) \leqslant 1$. Using this we can calculate that $\mathbf{skip}\;_{p.\mathsf{v}.\mathsf{h}}\!\oplus\;\mathbf{skip} \sqsubseteq \mathbf{skip}$ since from implicit flow the *lhs* reveals $p.\mathsf{v}.\mathsf{h}$ but the *rhs* reveals nothing.

On the other hand, additions of assertions that refer only to the *visible* state reveal nothing, and thus (13) can be strengthened to equality, giving

$$(\sum n \bullet [\![\{p_n.\mathsf{v}\}]\!]) \;=\; [\![\{\sum n \bullet p_n.\mathsf{v}\}]\!], \tag{14}$$

whence $\mathbf{skip}\;_{p.\mathsf{v}}\!\oplus\;\mathbf{skip} = \mathbf{skip}$.

Using this algebra of assertions for Booleans $G_{\{1,2\}}.\mathsf{v}.\mathsf{h}$ we have

$$\{G_1.\mathsf{v}.\mathsf{h}\};\{G_2.\mathsf{v}.\mathsf{h}\} \;=\; \{G_1.\mathsf{v}.\mathsf{h} \wedge G_2.\mathsf{v}.\mathsf{h}\}, \tag{15}$$

and so the *Boolean assertions* are idempotent, that is $\{G.\mathsf{v}.\mathsf{h}\};\{G.\mathsf{v}.\mathsf{h}\} = \{G.\mathsf{v}.\mathsf{h}\}$, and complements under composition so that $\{G.\mathsf{v}.\mathsf{h}\};\{\neg G.\mathsf{v}.\mathsf{h}\} = \mathbf{abort}$. When all Boolean

G_n.v.h are disjoint we also have

$$(\sum n\colon[1..N] \bullet \{G_n.\mathsf{v}.\mathsf{h}\}) \sqsubseteq \{\bigvee n\colon[1..N] \bullet G_n.\mathsf{v}.\mathsf{h}\}, \tag{16}$$
$$(\sum n\colon[1..N] \bullet \{G_n.\mathsf{v}\}) = \{\bigvee n\colon[1..N] \bullet G_n.\mathsf{v}\}. \tag{17}$$

9.3. *Basic laws for revelations*

A single **reveal** releases information but changes no variable. Using refinement we can with **reveal** $D_1 \sqsubseteq$ **reveal** D_2 express that revealing D_2 leaks no more information than revealing D_1 would have. The refinement between programs means this statement applies for *any* incoming distribution.

We write **reveal** (E_1, E_2) for the release of two pieces of information, one defined by expression E_1 and the other defined by expression E_2. For example **reveal** (h **mod** 2, h **mod** 3) releases information about both h's divisibility by 2 and 3: this is more informative than releasing just one, giving the refinement

$$\textbf{reveal}\ (\mathsf{h}\ \textbf{mod}\ 2, \mathsf{h}\ \textbf{mod}\ 3) \sqsubseteq \textbf{reveal}\ \ \mathsf{h}\ \textbf{mod}\ 2. \tag{18}$$

As we shall see, this and a number of other laws can be derived from a single general refinement rule which effectively states that any released information can be concealed somewhat by distributing it stochastically.

Lemma 9.1 (basic reveal refinement). Let D.v.h be a distribution over some $\mathcal{X}$ and F be a stochastic matrix (which can depend on v) giving for each element of $\mathcal{X}$ a distribution over some other type $\mathcal{Y}$. Then we have

$$\textbf{reveal}\ D.\mathsf{v}.\mathsf{h} \sqsubseteq \textbf{reveal}\ \ D.\mathsf{v}.\mathsf{h} \otimes F.\mathsf{v},$$

where $(\otimes)$ is defined by $(D.\mathsf{v}.\mathsf{h} \otimes F.\mathsf{v}).y := (\sum x\colon\mathcal{X} \bullet D.\mathsf{v}.\mathsf{h}.x \times F.\mathsf{v}.x.y)$.[†]

Proof. We reason as follows:

$$\begin{array}{lll}
 & [\![\textbf{reveal}\ D.\mathsf{v}.\mathsf{h}]\!] & \\
= & (\sum x\colon\mathcal{X} \bullet [\![\{D.\mathsf{v}.\mathsf{h}.x\}]\!]) & \text{“define revelation using assertions (6)”} \\
= & (\sum x\colon\mathcal{X} \bullet [\![\{D.\mathsf{v}.\mathsf{h}.x\}; \{(\sum y\colon\mathcal{Y} \bullet F.\mathsf{v}.x.y)\}]\!]) & \text{“distribution } F.\mathsf{v} \text{ is full;} \\
 & & \{1\} = \textbf{skip} \text{ is unit of composition (33)”} \\
= & (\sum x, y\colon\mathcal{X}, \mathcal{Y} \bullet [\![\{D.\mathsf{v}.\mathsf{h}.x\}; \{F.\mathsf{v}.x.y\}]\!]) & \text{“(14); Kleisli composition} \\
 & & \text{distributes over addition”} \\
= & (\sum x, y\colon\mathcal{X}, \mathcal{Y} \bullet [\![\{D.\mathsf{v}.\mathsf{h}.x \times F.\mathsf{v}.x.y\}]\!]) & \text{“(11)”} \\
\sqsubseteq & (\sum y\colon\mathcal{Y} \bullet [\![\{(\sum x\colon\mathcal{X} \bullet D.\mathsf{v}.\mathsf{h}.x \times F.\mathsf{v}.x.y)\}]\!]) & \text{“(13)”} \\
= & [\![\,\textbf{reveal}\ D.\mathsf{v}.\mathsf{h} \otimes F.\mathsf{v}\,]\!]. & \text{“define revelation using assertions (6)”}
\end{array}$$

[†] If D.v.h and F.v are expressed as matrices then $(\otimes)$ is matrix multiplication.

□

As an example of Lemma 9.1 we suppose h is Boolean, and that we have a revelation behaving as follows. If h is T then T is emitted with probability $1/4$ and F with probability $3/4$; if h is F then F is emitted unconditionally. We write this **reveal** $D.\mathsf{h}$ (omitting the $.\mathsf{v}$ in this simple case) via the D-matrix

$$\begin{array}{cc} & \begin{array}{cc} \mathsf{T} & \mathsf{F} \end{array} \quad \leftarrow \text{emitted value} \\ \begin{array}{c} \mathsf{h}{=}\mathsf{T} \\ \mathsf{h}{=}\mathsf{F} \end{array} & \begin{pmatrix} 1/4 & 3/4 \\ 0 & 1 \end{pmatrix} \end{array} \tag{19}$$

Now we can condition on the emitted value, so defining a partition on any incoming state: for example if the incoming state s is $(v, \{\!\{\mathsf{T}@\frac{1}{2}, \mathsf{F}@\frac{1}{2}\}\!\})$ then $[\![\,\mathbf{reveal}\ D.\mathsf{h}\,]\!].s = \{\!\{(v, \{\!\{\mathsf{T}\}\!\})@\frac{1}{8}, (v, \{\!\{\mathsf{T}@\frac{3}{7}, \mathsf{F}@\frac{4}{7}\}\!\})@\frac{7}{8}\}\!\}$ expressing the fact that T is emitted only if h is T thus completely revealing h in this case; however this happens only $1/8$ of the time; the remaining $7/8$ of the time h is only partly revealed, with the *a posteriori* distribution's being merely F-skewed.

Now, suppose that the process is overlaid by another process F (again omitting $.\mathsf{v}$) which obscures the information emitted by **reveal** $D.\mathsf{h}$ by changing the values stochastically:

$$\begin{array}{cc} & \begin{array}{cc} \mathsf{T} & \mathsf{F} \end{array} \quad \leftarrow \text{new emitted value} \\ \begin{array}{c} \text{emission from } D.\mathsf{h} \text{ was } \mathsf{T} \\ \text{emission from } D.\mathsf{h} \text{ was } \mathsf{F} \end{array} & \begin{pmatrix} 1 & 0 \\ 1/2 & 1/2 \end{pmatrix} \end{array} \tag{20}$$

Overall, the value actually emitted by the combination is determined by the product of the matrices in (19) and (20), that is

$$\begin{pmatrix} 1/4 & 3/4 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ 1/2 & 1/2 \end{pmatrix} = \begin{pmatrix} 5/8 & 3/8 \\ 1/2 & 1/2 \end{pmatrix}$$

which for the chosen incoming distribution gives that $[\![\,\mathbf{reveal}\ D.\mathsf{h} \otimes F\,]\!].s$ is $\{\!\{(v, \{\!\{\mathsf{T}@\frac{5}{9}, \mathsf{F}@\frac{4}{9}\}\!\})@\frac{9}{16}, (v, \{\!\{\mathsf{T}@\frac{3}{7}, \mathsf{F}@\frac{4}{7}\}\!\})@\frac{7}{16}\}\!\}$, leaking less than $[\![\,\mathbf{reveal}\ D.\mathsf{h}\,]\!].s$.

Now Lemma 9.1 justifies (18) with F as the projection function onto the first component. Other rules can be derived similarly:

Lemma 9.2 (simple reveal rules).

$$\mathbf{reveal}\ k = \mathbf{skip} \tag{21}$$

$$\mathbf{reveal}\ D.\mathsf{v}.\mathsf{h} \sqsubseteq \mathbf{skip} \tag{22}$$

$$\mathbf{reveal}\ \mathsf{h} \sqsubseteq \mathbf{reveal}\ D.\mathsf{v}.\mathsf{h} \tag{23}$$

$$\mathbf{reveal}\ G.\mathsf{v}.\mathsf{h} = \mathbf{reveal}\ \neg G.\mathsf{v}.\mathsf{h} \tag{24}$$

$$\mathbf{reveal}\ (E_1.\mathsf{v}.\mathsf{h}, E_2.\mathsf{v}.\mathsf{h}) \sqsubseteq \mathbf{reveal}\ E_1.\mathsf{v}.\mathsf{h} \tag{25}$$

$$\mathbf{reveal}\ (E.\mathsf{v}.\mathsf{h}, E.\mathsf{v}.\mathsf{h}) = \mathbf{reveal}\ E.\mathsf{v}.\mathsf{h} \tag{26}$$

Proof. The first is a consequence of the equivalent definition of revelations in terms of assertions and the rest are consequences of it and Lemma 9.1. For example (22) follows by defining $F.\mathsf{v}$ to be constant; and (23) follows by defining $F.\mathsf{v}$ in Lemma 9.1 to be

$D.\mathsf{v}.\mathsf{h}$; (24) follows by defining $F.\mathsf{v}$ to swap the values T and F. Finally (25, 26) follow by defining $F.\mathsf{v}$ to be the projection function. $\square$

With the apparatus so far, the example in Section 2 could be sketched[†]

$$\begin{array}{lll}
 & \mathsf{v}{:=}\,\mathsf{h}\div 2;\ \mathsf{v}{:=}\,\mathsf{v}\div 2 & \\
= & |\![\![\ \textbf{vis}\ \mathsf{v}';\ \mathsf{v}'{:=}\,\mathsf{h}\div 2; \mathsf{v}{:=}\,\mathsf{v}'\div 2\ ]\!]| & \text{“classical reasoning with visibles and scopes”} \\
= & |\![\![\ \textbf{vis}\ \mathsf{v}';\ \mathsf{v}'{:=}\,\mathsf{h}\div 2; \mathsf{v}{:=}\,(\mathsf{h}\div 2)\div 2\ ]\!]| & \text{“referential transparency Section 9.1”} \\
= & |\![\![\ \textbf{vis}\ \mathsf{v}';\ \mathsf{v}'{:=}\,\mathsf{h}\div 2\ ]\!]|; \mathsf{v}{:=}\,\mathsf{h}\div 4 & \text{“shrink scope; arithmetic”} \\
= & \textbf{reveal}\ \mathsf{h}\div 2;\ \mathsf{v}{:=}\,\mathsf{h}\div 4 & \text{“revelation equivalence Section 9.1”} \\
\sqsubseteq & \mathsf{v}{:=}\,\mathsf{h}\div 4. & \text{“(22), that } \textbf{reveal}\ \mathsf{h}\div 2 \sqsubseteq \textbf{skip}\text{”}
\end{array}$$

9.4. *Reveals in sequence*

When two or more HMM's are executed sequentially, where the outputs from one are 'fed into' another, an observer is able to preserve information from earlier executions to add to information learned by observing later executions. The basic rule expressing the total amount of information leaked is set out next.

Lemma 9.3 (sequential reveals). Let $D_1.\mathsf{v}.\mathsf{h}$ and $D_2.\mathsf{v}.\mathsf{h}$ be distributions over some $\mathcal{X}$ and $\mathcal{Y}$ respectively. Then we have

$$\textbf{reveal}\ D_1.\mathsf{v}.\mathsf{h}; \textbf{reveal}\ D_2.\mathsf{v}.\mathsf{h} = \textbf{reveal}\ (D_1{\times}D_2).\mathsf{v}.\mathsf{h},$$

where $(D_1{\times}D_2).\mathsf{v}.\mathsf{h}$ is the joint distribution over ordered pairs of $\mathcal{X}\times\mathcal{Y}$, defined as usual so that $(D_1{\times}D_2).\mathsf{v}.\mathsf{h}.(x,y)$ is $D_1.\mathsf{v}.\mathsf{h}.x \times D_2.\mathsf{v}.\mathsf{h}.y$.

Proof. This follows directly from the definition of **reveal** $D.\mathsf{v}.\mathsf{h}$ and sequential composition:

$$\begin{array}{lll}
 & [\![\ \textbf{reveal}\ D_1.\mathsf{v}.\mathsf{h}; \textbf{reveal}\ D_2.\mathsf{v}.\mathsf{h}\ ]\!] & \\
= & (\sum x,y{:}\,\mathcal{X},\mathcal{Y} \bullet [\![\{D_1.\mathsf{v}.\mathsf{h}.x\};\{D_2.\mathsf{v}.\mathsf{h}.y\}]\!]) & \text{“revelations as summations (6)} \\
 & & \text{composition distributes addition”} \\
= & (\sum x,y{:}\,\mathcal{X},\mathcal{Y} \bullet [\![\{D_1.\mathsf{v}.\mathsf{h}.x \times D_2.\mathsf{v}.\mathsf{h}.y\}]\!]) & \text{“(11)”} \\
= & [\![\ \textbf{reveal}\ (D_1{\times}D_2).\mathsf{v}.\mathsf{h}\ ]\!]. & \text{“represent assertion summation as revelation (6)”}
\end{array}$$

$\square$

This rule says that we can simplify two successive reveals into a single reveal where the external values are gathered together and the residual probabilities aggregated as expected, so that overall the result is as though a single HMM had been executed, albeit with a modified stochastic matrix. Using this basic rule we can prove the following:

[†] With only a selection of laws, sometimes we must omit details in the calculations.

Lemma 9.4 (simple sequential rules).

$$\textbf{reveal } D_1.\mathsf{v}.\mathsf{h}; \textbf{reveal } D_2.\mathsf{v}.\mathsf{h} = \textbf{reveal } D_2.\mathsf{v}.\mathsf{h}; \textbf{reveal } D_1.\mathsf{v}.\mathsf{h} \tag{27}$$

$$\textbf{reveal } E_1.\mathsf{v}.\mathsf{h}; \textbf{reveal } E_2.\mathsf{v}.\mathsf{h} = \textbf{reveal } (E_1.\mathsf{v}.\mathsf{h}, E_2.\mathsf{v}.\mathsf{h}) \tag{28}$$

$$\textbf{reveal } \mathsf{h}; \textbf{reveal } D.\mathsf{v}.\mathsf{h} = \textbf{reveal } \mathsf{h} \tag{29}$$

Proof. Using Lemma 9.3, equation (27) follows from the underlying commutativity, and (28) follows from the fact that **reveal** $E.\mathsf{v}.\mathsf{h}$ equals **reveal** $\{\!\{E.\mathsf{v}.\mathsf{h}\}\!\}$. For (29) we have from (22) and (33) that **reveal** h; **reveal** $D.\mathsf{v}.\mathsf{h} \sqsubseteq$ **reveal** h. For refinement in the other direction we reason

$$\begin{array}{cll}
 & \textbf{reveal } \mathsf{h} & \\
= & \textbf{reveal } (\mathsf{h}, \mathsf{h}) & \text{“(26)”} \\
= & \textbf{reveal } \mathsf{h}; \textbf{reveal } \mathsf{h} & \text{“(28)”} \\
\sqsubseteq & \textbf{reveal } \mathsf{h}; \textbf{reveal } D.\mathsf{v}.\mathsf{h}. & \text{“(22)”}
\end{array}$$

□

The rules in Lemma 9.4 formalize our intuition about successive reveals. For example (27) says that the information can be revealed in any order, that revealing two different expressions in succession is the same as revealing a pair containing both expressions (28), and that once h has been revealed entirely then there is nothing more to reveal (29).

The following lemma lists further properties explaining how assertions and revelations interact via sequential composition.

Lemma 9.5 (assertions and revelations in sequence).

$$\{p.\mathsf{v}.\mathsf{h}\}; \textbf{reveal } D.\mathsf{v}.\mathsf{h} = \textbf{reveal } D.\mathsf{v}.\mathsf{h}; \{p.\mathsf{v}.\mathsf{h}\} \tag{30}$$

$$\{G.\mathsf{v}.\mathsf{h}\} = \{G.\mathsf{v}.\mathsf{h}\}; \textbf{reveal } G.\mathsf{v}.\mathsf{h} \tag{31}$$

Proof: The first equivalence is shown using a similar proof to that of Lemma 9.3. For (31) we show

$$\begin{array}{cll}
 & [\![\{G.\mathsf{v}.\mathsf{h}\}]\!] & \\
= & [\![\{G.\mathsf{v}.\mathsf{h}\}]\!] + [\![\{\mathsf{F}\}]\!] & \text{“}\textbf{abort}\text{ is zero of program addition”} \\
= & [\![\{G.\mathsf{v}.\mathsf{h}\}; \{G.\mathsf{v}.\mathsf{h}\}]\!] + [\![\{G.\mathsf{v}.\mathsf{h}\}; \{\neg G.\mathsf{v}.\mathsf{h}\}]\!] & \text{“separate Boolean assertions (15)”} \\
= & [\![\{G.\mathsf{v}.\mathsf{h}\}; \textbf{reveal } G.\mathsf{v}.\mathsf{h}]\!]. & \text{“composition over addition; (6)”}
\end{array}$$

The first (30) states that revelations and assertions commute, while the second (31) says that after asserting predicate $G.\mathsf{v}.\mathsf{h}$, no more information can be leaked by revealing the value of $G.\mathsf{v}.\mathsf{h}$.

9.5. *Reveals in choice*

In a probabilistic choice between two reveal statements, an observer may witness both which revelation was executed as well as the outcome of that statement. We can combine such a choice into a single reveal statement.

Lemma 9.6 (choices between reveals). Let $D_L.\mathsf{v.h}$ and $D_R.\mathsf{v.h}$ be distributions over $\mathcal{X}$ and $p.\mathsf{v.h}$ be a probability; let

$$\mathcal{X}_2 := \quad \mathsf{Lft}\ \mathcal{X}\ +\ \mathsf{Rgt}\ \mathcal{X}$$

be the discriminated union of two copies of $\mathcal{X}$, with injection functions therefore of type $\mathsf{Lft}, \mathsf{Rgt}\colon \mathcal{X}\to\mathcal{X}_2$. We have that

$$\begin{array}{rl} & \textbf{reveal } D_L.\mathsf{v.h} \\ {}_{p.\mathsf{v.h}}\oplus & \textbf{reveal } D_R.\mathsf{v.h} \end{array} = \begin{array}{rl} \textbf{reveal} & \mathsf{map.Lft.}(D_L.\mathsf{v.h}) \\ {}_{p.\mathsf{v.h}}\oplus & \mathsf{map.Rgt.}(D_R.\mathsf{v.h}), \end{array}$$

where the injection-functions' 'tagging' of the two distributions has effectively given them disjoint supports.

Proof. Let $D'_L.\mathsf{v.h}, D'_R.\mathsf{v.h}$ be respectively $\mathsf{map.Lft.}(D_L.\mathsf{v.h}), \mathsf{map.Rgt.}(D_R.\mathsf{v.h})$. We have then

$$\begin{array}{lll}
 & [\![\ \textbf{reveal } D_L.\mathsf{v.h}\ {}_{p.\mathsf{v.h}}\oplus\ \textbf{reveal } D_R.\mathsf{v.h}\ ]\!] & \\
= & [\![\ \textbf{reveal } D'_L.\mathsf{v.h}\ {}_{p.\mathsf{v.h}}\oplus\ \textbf{reveal } D'_R.\mathsf{v.h}\ ]\!] & \text{“Lemma 9.1”} \\
= & \quad [\![\ \{p.\mathsf{v.h}\}; \textbf{reveal } D'_L.\mathsf{v.h}\ ]\!] & \text{“probabilistic choice”} \\
 & +\ [\![\ \{1-p.\mathsf{v.h}\}; \textbf{reveal } D'_R.\mathsf{v.h}\ ]\!] & \\
= & \sum_{x:\mathcal{X}} & \text{“revelations are additions of assertions (6);} \\
 & \quad [\![\{p.\mathsf{v.h}\}; \{D'_L.\mathsf{v.h.}(\mathsf{Lft}.x)\}]\!] & \text{additive distributivity} \\
 & +\ [\![\{1-p.\mathsf{v.h}\}; \{D'_R.\mathsf{v.h.}(\mathsf{Rgt}.x)\}]\!] & \text{of Kleisli composition”} \\
= & (\sum x_2 : \mathcal{X}_2 \bullet [\![\{(D'_L.\mathsf{v.h}\ {}_{p.\mathsf{v.h}}\oplus\ D'_R.\mathsf{v.h}).x_2\}]\!]) & \text{“(11); and } D'_L.\mathsf{v.h.}(\mathsf{Rgt}.x), \\
 & & D'_R.\mathsf{v.h.}(\mathsf{Lft}.x) \text{ both zero”} \\
= & [\![\ \textbf{reveal } D'_L.\mathsf{v.h}\ {}_{p.\mathsf{v.h}}\oplus\ D'_R.\mathsf{v.h}\ ]\!]. & \text{“addition of assertions as revelation (6)”}
\end{array}$$

□

From this lemma we may derive the following laws concerning revelations and probabilistic choice.

Lemma 9.7 (simple choice rules). For probability $p.\mathsf{v.h}$ and distributions $D_1.\mathsf{v.h}$ and $D_2.\mathsf{v.h}$ we have that

$$(\textbf{reveal } D_L.\mathsf{v.h}\ \ {}_{p.\mathsf{v.h}}\oplus\ \textbf{reveal } D_R.\mathsf{v.h})\ \sqsubseteq\ \textbf{reveal } (D_L.\mathsf{v.h}\ \ {}_{p.\mathsf{v.h}}\oplus\ D_R.\mathsf{v.h}).$$

However if the support of $D_L.\mathsf{v.h}$ and $D_R.\mathsf{v.h}$ are disjoint, then the refinement relation is an equality.
Proof: This follows from Lemmas 9.1 and 9.6.

From (21) and Lemma 9.7 we have, for example, that

$$\textbf{skip}\ {}_{p.\mathsf{v.h}}\oplus\ \textbf{skip} = (\textbf{reveal } \mathsf{T}\ {}_{p.\mathsf{v.h}}\oplus\ \textbf{reveal } \mathsf{F}) = \textbf{reveal } (\mathsf{T}\ {}_{p.\mathsf{v.h}}\oplus\ \mathsf{F}),$$

thus illustrating the information leakage due to implicit flow.

9.6. *Composition, probabilistic choice and conditionals*

As well as being monotonic in both their program arguments (Theorem 8.1), sequential composition and probabilistic choice – of which conditional choice is a special case – satisfy the following basic laws corresponding to classical probabilistic equalities (McIver and Morgan 2005).

Lemma 9.8 (basic composition and choice laws). For all programs S, T and R and probabilities $p.\mathsf{v.h}$ we have the following properties hold.

$$S;(T;R) = (S;T);R \tag{32}$$
$$\mathbf{skip};S = S;\mathbf{skip} = S \tag{33}$$
$$\mathbf{abort};S = S;\mathbf{abort} = \mathbf{abort} \tag{34}$$
$$(S\ {}_{p.\mathsf{v.h}}\!\oplus T) = (T\ {}_{1-p.\mathsf{v.h}}\!\oplus S) \tag{35}$$
$$(S\ {}_{p.\mathsf{v.h}}\!\oplus T);R = (S;R\ {}_{p.\mathsf{v.h}}\!\oplus T;R) \tag{36}$$
$$(S\ {}_{1}\!\oplus T) = S \tag{37}$$

Additionally, for any R satisfying both $R;\{p.\mathsf{v.h}\} = \{p.\mathsf{v.h}\};R$ and $R;\{1-p.\mathsf{v.h}\} = \{1-p.\mathsf{v.h}\};R$ we have that R distributes from the left into a choice with probability $p.\mathsf{v.h}$:

$$R;(S\ {}_{p.\mathsf{v.h}}\!\oplus T) = (R;S\ {}_{p.\mathsf{v.h}}\!\oplus R;T). \tag{38}$$

Since both assertions (11) and reveals (30) commute over assertions, Equation (38) gives us that they distribute to the right over arbitrary probabilistic (and conditional) choices. Additionally, commutativity of constant assertions over all statements (12) means that all programs distribute over choices in which the probability is constant. We can also derive, for example, the following properties:

$$S\ {}_{p.\mathsf{v.h}}\!\oplus S \sqsubseteq S \tag{39}$$
$$S\ {}_{p.\mathsf{v}}\!\oplus S = S \tag{40}$$
$$\mathbf{if}\ G\ \mathbf{then}\ S\ \mathbf{else}\ T = \mathbf{reveal}\ G;\mathbf{if}\ G\ \mathbf{then}\ S\ \mathbf{else}\ T \tag{41}$$

The first two follow from left distributivity (21),(33),(36) and Lemma 9.7. The last follows from (31) and (38).

9.7. *Rules for general iteration*

Recall that we write **while** $p.\mathsf{v.h}$ **do** S **od** for a general iteration of S, with probability $p.\mathsf{v.h}$ of exiting the loop on each iteration. From Theorem 8.1 we have that such loops are monotonic on their program argument. Additionally, from its least-fixed-point semantics we have

Lemma 9.9 (fixed point rule). If $(S;W)\ {}_{p.\mathsf{v.h}}\!\oplus\ \mathbf{skip} = W$ then we have the refinement $(\mathbf{while}\ p.\mathsf{v.h}\ \mathbf{do}\ S\ \mathbf{od}) \sqsubseteq W$.

Proof. From the Tarski fixed-point theorem (Tarski 1955) wrt the order ($\preccurlyeq$), and that the loop is a least fixed point, we have immediately

$$(S\,;W)\,{}_{p.\mathsf{v.h}}\!\oplus\,\mathbf{skip}\ \preccurlyeq\ W \qquad \text{implies} \qquad (\mathbf{while}\ p.\mathsf{v.h}\ \mathbf{do}\ S\ \mathbf{od})\ \preccurlyeq\ W\,. \tag{42}$$

The result then follows immediately from the two inclusions $(=) \subseteq (\preccurlyeq) \subseteq (\sqsubseteq)$. □

In a specification task, however, the goal is typically to implement a specification by an iteration, i.e. to establish a refinement in the opposite direction. For terminating iterations we have this rule:

Corollary 9.1 (termination iteration). If **while** $p.\mathsf{v.h}$ **do** S **od** terminates with probability one, and $(S\,;W)\,{}_{p.\mathsf{v.h}}\!\oplus\,\mathbf{skip} = W$, then **while** $p.\mathsf{v.h}$ **do** S **od** $= W$.

Proof. We adapt the proof of Lemma 9.9, noting that if the loop terminates it is ($\preccurlyeq$)-maximal and hence the *rhs* ($\preccurlyeq$) in (42) must in fact be an equality. □

Termination is usually shown by exhibiting a probabilistic variant over the state (Hart *et al.* 1983; Morgan 1996; McIver and Morgan 2005); a straightforward simple case is when the loop's exit probability is bounded away from zero, in particular **while** k **do** $\cdots$ for any constant $k < 1$.

9.8. *Small example: one guess at a password*

We have a hidden password p chosen from three possibilities $\mathcal{P}{:=}\{p_1, p_2, p_2\}$. This fragment describes an attacker's single guess, uniformly chosen:

$$|\![\ \mathbf{vis}\ \mathsf{g};\ \mathsf{g}{:\in}\,\{\!\{p_1, p_2, p_3\}\!\};\ \mathbf{reveal}\ \mathsf{g{=}p}\ ]\!|.$$

Local visible value g is chosen from the uniform distribution $\{\!\{p_1, p_2, p_3\}\!\}$, and then it is used as a guess. Note that if the guess is correct, then T is revealed which – in itself – does not reveal the password's value: that latter is *then* learned by deduction, from the program's code and the fact that g is visible. If g had been hidden, we would know only that the guess had succeeded, but still not the value of p.

We now show how algebra can be used to convert 'operational' descriptions like the above into less obvious but more calculationally convenient forms, in this case a single **reveal** statement; and in Section 10 we will see how useful this equivalence turns out to be. For now, we reason

$$|\![\ \mathbf{vis}\ \mathsf{g};\ \mathsf{g}{:\in}\,\{\!\{p_1, p_2, p_3\}\!\};\ \mathbf{reveal}\ \mathsf{g{=}p}\ ]\!|$$

```
=   |[ vis g;                                    "split visible choice using (7);
       g:= $p_1$ ⊕ g:= $p_2$ ⊕ g:= $p_3$;                 note choices (⊕) are uniform
       reveal  g=p                                       by convention, i.e. ($_{1/3}\oplus$)
    ]|                                                                in this case"
```

$$
\begin{array}{lll}
= & |\!|\![\ \mathbf{vis}\ \mathsf{g}; & \text{“left distributivity (36)”} \\
 & \quad\ \ \mathsf{g}{:=}p_1;\mathbf{reveal}\ \mathsf{g}{=}\mathsf{p}; & \\
 & \quad \oplus\ \mathsf{g}{:=}p_2;\mathbf{reveal}\ \mathsf{g}{=}\mathsf{p}; & \\
 & \quad \oplus\ \mathsf{g}{:=}p_3;\mathbf{reveal}\ \mathsf{g}{=}\mathsf{p} & \\
 &]\!|\!| &
\end{array}
$$

$$
\begin{array}{lll}
= & |\!|\![\ \mathbf{vis}\ \mathsf{g}; & \text{“replace expressions by those of equal value (Section 9.1);} \\
 & \quad\ \ \mathsf{g}{:=}p_1;\mathbf{reveal}\ p_1{=}\mathsf{p}; & \text{e.g. in the first branch } \mathsf{g}{:=}p_1 \text{ establishes} \\
 & \quad \oplus\ \mathsf{g}{:=}p_2;\mathbf{reveal}\ p_2{=}\mathsf{p}; & \text{that } p_1{=}\mathsf{g}\text{, so that } \mathsf{g} \text{ can be} \\
 & \quad \oplus\ \mathsf{g}{:=}p_3;\mathbf{reveal}\ p_3{=}\mathsf{p} & \text{replaced by } p_1 \text{ in the } \mathbf{reveal}\text{”} \\
 &]\!|\!| &
\end{array}
$$

$$
\begin{array}{lll}
= & \quad\ \ |\!|\![\ \mathbf{vis}\ \mathsf{g};\mathsf{g}{:=}p_1\ ]\!|\!|;\ \mathbf{reveal}\ p_1{=}\mathsf{p} & \text{“shift scope (Section 9.1), since } \mathsf{g} \text{ is} \\
 & \oplus\ |\!|\![\ \mathbf{vis}\ \mathsf{g};\mathsf{g}{:=}p_2\ ]\!|\!|;\ \mathbf{reveal}\ p_2{=}\mathsf{p} & \text{no longer free in the } \mathbf{reveal}\text{’s”} \\
 & \oplus\ |\!|\![\ \mathbf{vis}\ \mathsf{g};\mathsf{g}{:=}p_3\ ]\!|\!|;\ \mathbf{reveal}\ p_3{=}\mathsf{p} &
\end{array}
$$

$$
\begin{array}{lll}
= & \mathbf{reveal}\ p_1{=}\mathsf{p}\ \oplus\ \mathbf{reveal}\ p_2{=}\mathsf{p}\ \oplus\ \mathbf{reveal}\ p_3{=}\mathsf{p} & \text{“(9), (8) and (33)”} \\
= & \mathbf{reveal}\ (p_1,p_1{=}\mathsf{p})\ \oplus\ \mathbf{reveal}\ (p_2,p_2{=}\mathsf{p})\ \oplus\ \mathbf{reveal}\ (p_3,p_3{=}\mathsf{p}) & \text{“Lemma 9.1”} \\
= & \mathbf{reveal}\ \ (p_1,p_1{=}\mathsf{p}) \oplus (p_2,p_2{=}\mathsf{p}) \oplus (p_3,p_3{=}\mathsf{p}), & \text{“Lemma 9.7”}
\end{array}
$$

giving a single **reveal** whose expression part we manipulate further, at (46) below. Note that it is the appeal to (7) that relies on g's being visible: if it were not, then the implicit flow introduced by the first step would represent a leak, invalidating the equality.

10. Extended example: iterative reasoning

We now demonstrate our treatment of iteration, reusing the simple password-guessing attack within a loop.

10.1. *A password attack: specification*

We assume a set of passwords $\mathcal{P}$ and a hidden variable $\mathsf{p}{:}\mathcal{P}$ containing the (current) password; let $\mathbb{P}_N\mathcal{P}$ be the set of all size-N subsets of $\mathcal{P}$. A typical attack would be to choose one of those sets of potential passwords, and then to try them all in a 'bulk attack' as in the program fragment

$$|\!|\![\ \mathbf{vis}\ \mathsf{G};\ \ \mathsf{G}{:\in}\ \lfloor\mathbb{P}_N\mathcal{P}\rfloor;\ \mathbf{reveal}\ \{\mathsf{p}\}{\cap}\mathsf{G}\ ]\!|\!|. \tag{43}$$

(We omit the typing $\mathsf{G}{:}\mathbb{P}\mathcal{P}$, to reduce clutter.) The statement $\mathsf{G}{:\in}\lfloor\mathbb{P}_N\mathcal{P}\rfloor$ makes a uniform choice of size-N subset of $\mathcal{P}$, assigning it to G. We are assuming that N is strictly less than the size P of $\mathcal{P}$.

The **reveal** $\{\mathsf{p}\}{\cap}\mathsf{G}$ reveals either $\{\mathsf{p}\}$, if the attack succeeds, or the empty set $\varnothing$ if it does not. That is, the outcome of fragment (43) above is either to say 'the hidden password is p' (a successful attack, revealing $\{\mathsf{p}\}$) or 'the hidden password is not in G' (an unsuccessful attack, revealing $\varnothing$) since, in the latter case we do know the visible attack-set G even though the attack failed. As a specification, it abstracts from precisely how the passwords are tried, in what order, or whether possibly repeated: it says only that they *are* tried.

Now suppose the incoming distribution of p is some $\pi : \mathbb{D}\mathcal{P}$; then the program fragment above produces an output hyper $\Pi : \mathbb{D}^2\mathcal{P}$ comprising a distribution of distributions over $\mathcal{P}$. (Note that the output hyper contains no G component, because G is local.) If we calculated this with our semantics (although we omit the calculations here), we would find two kinds of inners in its support, namely

success. A p-indexed family of point inner distributions $\{\!\{p\}\!\}$ each itself with outer probability $N(\pi.p)/P$, the probability $\pi.p$ that p was the password, but multiplied by the probability N/P that it was in the uniformly chosen attack-set G of size N.

failure. A G-indexed set of inner distributions of support-size $P-N$, each such distribution derived by conditioning π on *not* being in the set G and having outer probability $(1-\pi.G)/\mathsf{C}^P_N$, the probability that this particular G was chosen for the attack-set multiplied by the probability that the password was not in it.

As a check, we note that the outer probabilities sum to one, as they should since the specification program is terminating: we have

$$\begin{array}{ll}
 & \sum_p N(\pi.p)/P + \sum_G (1-\pi.G)/\mathsf{C}^P_N \\
 & \sum_p N(\pi.p)/P + \sum_G 1/\mathsf{C}^P_N - \sum_G \pi.G/\mathsf{C}^P_N \\
= & N/P + \mathsf{C}^P_N/\mathsf{C}^P_N - \sum_p \mathsf{C}^{P-1}_{N-1}(\pi.p)/\mathsf{C}^P_N \\
= & 1.
\end{array}$$

Finally, if for example we assume that the incoming distribution π is uniform over $\mathcal{P}$, then the Bayes Risk before the attack is $1-1/P$ and, after the attack, it has been reduced to the conditional Risk $P \times N(\pi.p)/P \times 0 + \mathsf{C}^P_N \times ((1-N/P)/\mathsf{C}^P_N) \times (1-1/(P-N))$, that is reduced to $1-(N+1)/P$.

10.2. *A password attack: implementation*

We suppose a simple-minded actual attacker who chooses single passwords uniformly at random, possibly with repetition and, after each attack, has some fixed probability c of giving up. This would be described by the fragment

```
while c do
  |[ vis g;  g:∈ 𝒫;  reveal g=p ]|          (44)
od.
```

A complete analysis of (44) is combinatorially complex, having an output hyper comprising inner distributions over subsets of $\mathcal{P}$ of all possible sizes and – as such – would be difficult to reason about within a larger system. More practical would be to determine, once and for all, whether (44) is an implementation of (43), i.e. that it is at least as secure as (43) and then ever after to use the simpler (43) in larger analyses. Since (44) is parametrized by c, we might in fact ask

What is the largest value of probability c for which (43) $\sqsubseteq$ (44)?

10.3. *Example refinement analysis: the simplest case*

To illustrate the approach, we address the above question in the very simple case where $\mathcal{P}=\{p_1,p_2,p_3\}$ is of size 3, and our specification describes a 'bulk attack' of size $N=1$.[†] Thus, we are asking for the largest c that achieves the refinement

$$\begin{array}{ll} |\!|\![\ \textbf{vis}\ \mathsf{g};\ \mathsf{g}{:}{\in}\,\mathcal{P};\ \textbf{reveal}\ \mathsf{g}{=}\mathsf{p}\ ]\!|\!| \sqsubseteq & \textbf{while}\ c\ \textbf{do} \\ & \quad |\!|\![\ \textbf{vis}\ \mathsf{g};\ \mathsf{g}{:}{\in}\,\mathcal{P};\ \textbf{reveal}\ \mathsf{g}{=}\mathsf{p}\ ]\!|\!| \\ & \textbf{od}. \end{array} \tag{45}$$

We do this in two stages: the first is to hypothesize a parametrized straight-line equivalent for the loop, then synthesizing a condition on the parameters that makes it satisfy the fixed-point equation of Corollary 9.1.

As in Lemma 9.6, we introduce a discriminated union $\mathcal{P}^? := \mathsf{is}\ \mathcal{P} + \mathsf{isn't}\ \mathcal{P} + \mathsf{nix}$ which, used in **reveal** commands, will allow us to reveal what p is, what it is not, and – for algebraic convenience – to reveal nothing at all.

In our simple case here of $\mathcal{P}$ having just three elements, therefore $\mathcal{P}^?$ has seven. Further exploiting $\mathcal{P}$'s size of three, for any p in $\mathcal{P}$ we write p_+ for one of the values p is not, and p_- for the other. With this approach we can express *lhs* (45) without its local block and the guess variable g: for that, we return to our example calculation of Section 9.8 giving **reveal** $(p_1,p_1{=}\mathsf{p}) \oplus (p_2,p_2{=}\mathsf{p}) \oplus (p_3,p_3{=}\mathsf{p})$. We can recode this directly using Lemma 9.1: it becomes just

$$\textbf{reveal}\ \{\!\{\mathsf{is\ p},\ \mathsf{isn't\ p}_+,\ \mathsf{isn't\ p}_-\}\!\}. \tag{46}$$

We return to the synthesis of the loop's straight-line equivalent, supposing it has the form

$$\textbf{reveal}\ \{\!\{\mathsf{is\ p}^{@x},\ \mathsf{isn't\ p_+}^{@\frac{y}{2}},\ \mathsf{isn't\ p_-}^{@\frac{y}{2}},\ \mathsf{nix}^{@z}\}\!\} \tag{47}$$

for some probabilities $x{+}y{+}z = 1$ that we have to determine. This reveals what p is with probability x, what p is not with probability $y/2{+}y/2 = y$; and with probability z it reveals nothing at all.

Our synthesizing equality is then given by Corollary 9.1, because the loop with its constant c terminates; that is we require

$$\begin{array}{lll} & \textbf{reveal}\ \{\!\{\mathsf{is\ p}^{@x},\ \mathsf{isn't\ p_+}^{@\frac{y}{2}},\ \mathsf{isn't\ p_-}^{@\frac{y}{2}},\ \mathsf{nix}^{@z}\}\!\} & \big\}\leftarrow (47) \\ = & \quad\textbf{reveal}\ \{\!\{\mathsf{is\ p},\ \mathsf{isn't\ p}_+,\ \mathsf{isn't\ p}_-\}\!\}; & \longleftarrow\ \text{loop body} \\ & \quad\textbf{reveal}\ \{\!\{\mathsf{is\ p}^{@x},\ \mathsf{isn't\ p_+}^{@\frac{y}{2}},\ \mathsf{isn't\ p_-}^{@\frac{y}{2}},\ \mathsf{nix}^{@z}\}\!\} & \big\}\leftarrow (47) \\ {}_c\oplus & \quad\textbf{skip}, & \longleftarrow\ \text{loop exit} \end{array}$$

[†] In this simple case a bulk attack of size $N{=}2$ is uninteresting, because it would reveal everything: either what the password is (if $\mathsf{p}{\in}\mathsf{G}$) or two values that it is not (if $\mathsf{p}{\notin}\mathsf{G}$). In the latter case we would deduce p's value anyway, by elimination.

whose right-hand side we can simplify with the revelation laws from Section 9, in particular Lemmas 9.1, 9.3 and 9.7. That gives

$$\textbf{reveal}\ \left\{\!\!\left\{\begin{array}{ll} \textsf{is p} & @\ c(x+2y/3+z/3), \\ \textsf{isn't p}_+ & @\ c(y/6+z/3), \\ \textsf{isn't p}_- & @\ c(y/6+z/3), \\ \textsf{nix} & @\ 1-c \end{array}\right\}\!\!\right\},$$

and that should be equal to the left-hand side, the original (47). Since $z = 1-c$ trivially, we concentrate $\textsf{p}_+$ case to obtain $y/2 = c(y/6 + 2(1-c)/3)$, so that $y = 2c(1-c)/(3-c)$, whence $x = c-y = c(1+c)/(3-c)$.[†]

10.4. *Establishing the c-optimal refinement: the second stage*

We now want to find the largest value of c that allows

$$\begin{array}{ll} & |\!|\![\ \textbf{vis}\ \textsf{g};\ \textsf{g}{:}{\in}\,\mathcal{P};\ \textbf{reveal}\ \textsf{g=p}\ ]\!|\!| \\ \sqsubseteq & \textbf{reveal}\ \{\!\{\textsf{is p}^{@x},\ \textsf{isn't p}_+{}^{@\frac{y}{2}},\ \textsf{isn't p}_-{}^{@\frac{y}{2}},\ \textsf{nix}^{@z}\}\!\} \end{array}$$

where x, y, z have the c-determined values calculated above: we recall the remark above at (46) about formulating our specification as a simple revelation, without needing a local variable $\textsf{g}$. That gives the equivalent goal

$$\begin{array}{ll} & \textbf{reveal}\ \{\!\{\textsf{is p}^{@\frac{1}{3}},\ \textsf{isn't p}_+{}^{@\frac{1}{3}},\ \textsf{isn't p}_-{}^{@\frac{1}{3}}\}\!\} \\ \sqsubseteq & \textbf{reveal}\ \{\!\{\textsf{is p}^{@x},\ \textsf{isn't p}_+{}^{@\frac{y}{2}},\ \textsf{isn't p}_-{}^{@\frac{y}{2}},\ \textsf{nix}^{@z}\}\!\} \end{array}$$

† Working through this and extracting the arithmetic results in the following table:

↓ effective joint revelation			
is p, is p	with prob. $x/3$	equivalent to revealing just	is p
is p, isn't p_+	with prob. $y/6$	equivalent to revealing just	is p
is p, isn't p_-	with prob. $y/6$	equivalent to revealing just	is p
is p, nix	with prob. $z/3$	equivalent to revealing just	is p
isn't p_+, is p	with prob. $x/3$	equivalent to revealing just	is p
isn't p_+, isn't p_+	with prob. $y/6$	equivalent to revealing just	isn't p_+
isn't p_+, isn't p_-	with prob. $y/6$	equivalent to revealing just	is p
isn't p_+, nix	with prob. $z/3$	equivalent to revealing just	isn't p_+
isn't p_-, is p	with prob. $x/3$	equivalent to revealing just	is p
isn't p_-, isn't p_+	with prob. $y/6$	equivalent to revealing just	is p
isn't p_-, isn't p_-	with prob. $y/6$	equivalent to revealing just	isn't p_-
isn't p_-, nix	with prob. $z/3$	equivalent to revealing just	isn't p_- .

The probabilities in the text come from adding the final column in groups.

For this we refer to Lemma 9.1, whose D is effectively the *lhs* above: written as a matrix, it would be

$$\begin{array}{c|ccccccc} & \text{is } p_1 & \text{isn't } p_1 & \text{is } p_2 & \text{isn't } p_2 & \text{is } p_3 & \text{isn't } p_3 & \text{nix} \\ \hline p_1 & 1/3 & 0 & 0 & 1/3 & 0 & 1/3 & 0 \\ p_2 & 0 & 1/3 & 1/3 & 0 & 0 & 1/3 & 0 \\ p_3 & 0 & 1/3 & 0 & 1/3 & 1/3 & 0 & 0 \end{array} . \qquad (48)$$

We need a $7{\times}7$ stochastic matrix F, that is a function $\mathcal{P}^? \rightarrow \mathbb{D}\mathcal{P}^?$ which, when multiplied after D, gives the *rhs* above, that is

$$\begin{array}{c|ccccccc} & \text{is } p_1 & \text{isn't } p_1 & \text{is } p_2 & \text{isn't } p_2 & \text{is } p_3 & \text{isn't } p_3 & \text{nix} \\ \hline p_1 & x & 0 & 0 & y/2 & 0 & y/2 & z \\ p_2 & 0 & y/2 & x & 0 & 0 & y/2 & z \\ p_3 & 0 & y/2 & 0 & y/2 & x & 0 & z \end{array} .$$

The columns of the latter must be interpolations of columns of the former, thus the first *rhs* column $[x, 0, 0]$ cannot contain non-zero contributions from any other than the first *lhs* column $[1/3, 0, 0]$.† Hence $x \leqslant 1/3$ and, since we are trying to maximize c we maximize x also by setting $x := 1/3$.‡ Similar reasoning then establishes that the second *rhs* column $[0, y/2, y/2]$ must be obtained by taking proportion $3y/2$ of the second *lhs* column; and then the last *rhs* column is made by combining proportions $1 - 3y/2$ of each of columns 1,3,5 on the *lhs*.

Since $x{=}1/3$ entails $c(1{+}c)/(3{-}c){=}1/3$, that is $c \approx 0.53$, we have established our desired (45) with c taking that value (or less), independently of the distribution with which the hidden p might have been chosen.

11. Related work

11.1. *HMMs, algebra and noninterference*

HMMs (Jurafsky and Martin 2000) have a long history and many practical applications; their conceptual connection to noninterference suggests that their algorithmic methods might be of use here. That is, extant HMM techniques could be used for efficient numerical calculation of whether some $\boldsymbol{T}_S, \boldsymbol{E}_S$, a specification, was secure enough for our purposes: once that was done, the refinement relation established via program-algebra could ensure that an implementation $\boldsymbol{T}_I, \boldsymbol{E}_I$ was at least as secure as that *without* requiring a second numerical calculation. The advantage of this is that the first calculation, over a smaller and more abstract system, is likely to be much simpler than the second would have been.

† We write transposed columns horizontally as rows between brackets $[\cdot]$ instead of parentheses.
‡ The function $x := c(1{+}c)/(3{-}c)$ is monotonic for $0 \leqslant c \leqslant 1$.

There are techniques based on the manipulation of 'graphical models' to represent Bayesian networks in alternate equivalent ways: these are similar in spirit to our algebraic manipulations (Bishop 2006), although there the motivation is usually to find more efficient algorithms.

The application of HMMs to noninterference security is recent: originally, noninterference was qualitative (Goguen and Meseguer 1984). Probabilistic noninterference (Smith 2003; Chatzikokolakis *et al.* 2007) is a generalization of that idea to provide weaker statements concerning an attacker's ability to guess high security state by observing the behaviour and pattern of observables. Variations of the idea have been studied extensively for concurrent systems (Sabelfeld and Sands 2000; Volpano and Smith 1998) and taking computational issues into account (Backes and Pfitzmann 2002).

The definition of our space $\mathbb{D}(\mathcal{V}\times\mathbb{D}\mathcal{H})$ and its refinement order draws inspiration from constructions and techniques already present in the literature. The monad is Giry/Kantorovich (Giry 1981; van Breugel 2005), and the refinement order is related to the theory of inhomogeneous Markov Chains (Cohn 1974).

11.2. *Compositionality, information theory and assorted entropies*

A compelling approach to quantitative security is to use information-theoretic measures to compare the (e.g. Shannon) entropy of the hidden variables' *a priori* distribution (e.g. their incoming values) and their *a posteriori* distribution once the program has executed (Chatzikokolakis and Palamidessi 2007; Chatzikokolakis *et al.* 2007; Alvim *et al.* 2010); recently this has been applied to iterating programs as well (Malacaria 2010; Mu and Clark 2009). But *compositionality* is crucial: given that one program is more secure than another according to some entropy-based criterion, how do we know that inequality is preserved in a larger context?

We have shown earlier (McIver *et al.* 2010) that refinement has two key properties for compositional entropy-based reasoning: it is preserved by contexts; and it implies non-decrease for an assortment of entropies, including Shannon entropy, guessing entropy, Bayes risk and marginal guesswork. Perhaps it applies to others (Braun *et al.* 2009).

Thus, our work here is part of a larger program to unite earlier work in quantitative information flow (or escape) (Alvim *et al.* 2010; Köpf and Basin 2007) in channels, as models of computation, with a *denotational* presentation of program semantics based on HMMs including a *compositional* refinement relation that compares these quantitative measures between programs, specifically between specifications and their purported implementations. By considering iterations, we are extending our own earlier work (McIver *et al.* 2010) in a way that relates to others' work on quantitative information flow from iterations (Malacaria 2010; Mu and Clark 2009) much as in the way described above.

Compositionality 'within' a program addresses the question of whether security established for a component is preserved when embedded in a larger context (Braun *et al.* 2008). Compositionality 'between' programs, as we do here, addresses the question of whether two programs' relative security is preserved when they are both placed in the same context: this latter is less common.

A representative example of others' doing so is recent work by Yasuoka and Terauchi (2010) in which computational hardness is analysed. They consider deterministic sequential straight-line programs i.e. without probabilistic or demonic choice and without loops, but that nevertheless operate in a quantitative context (i.e. having input *distributions* rather than simply input values). Since the programs have no probabilistic choices, those authors are able to reduce the (analogue of) the secure-refinement relation to the qualitative noninterference comparison of the programs. This is a special case of our general conjecture concerning the promotion of qualitative results to quantitative results provided demonic choice is replaced by uniform choice (McIver *et al.* 2010, Section 8.1): if there is no demonic choice, there is nothing to replace and so the program is unchanged.

These authors look for a relation guaranteeing the correct entropy ordering (for all incoming distributions) wrt a selection of entropies, as we do, and they address the computational hardness of validating that relationship in particular cases. We address with compositional *closure* the additional question of how weak such a relation can be (McIver *et al.* 2010).

12. Summary, conclusions, prospects

Earlier we built the core of a programming algebra for probability and noninterference: here we have extended it to include iteration and nontermination; and we solved the technical problem of incompleteness, that arose in the process, by introducing a simpler 'termination' order that allowed us to remain with discrete distributions. Further, we have shown how the semantics is related to HMMs, an existing consensus of how such application domain should be handled and analysed.

The formalist rigour of program semantics, however, can make unusual demands on traditional mathematical presentations: a programming language is interpreted inductively in a structured space equipped with operators corresponding to the constructors of that language. In particular, sequential programs with any kind of nondeterminism (whether demonic, probabilistic or some other) are often interpreted as functions of type $\mathcal{S}\to\mathbb{K}\mathcal{S}$ where $\mathcal{S}$ is the state space and $\mathbb{K}$ is some type constructor (or functor) expressing the nondeterminism. Thus, our first contribution in detail was to (re-)interpret HMMs in this style (in Section 3), where $\mathcal{S}$ was $\mathcal{V}\times\mathbb{D}\mathcal{H}$ and $\mathbb{K}$ became $\underline{\mathbb{D}}$ (in Section 3.4). We made some small programming-motivated extensions to the HMM model, in particular adding *visible* variables to the state so that the most recent observation is carried forward into the next operation. The second extension was allowing *iterations* and hence, potentially, computations that might not terminate (thus $\underline{\mathbb{D}}$ rather than $\mathbb{D}$).

In constructing the semantic operations we built-in *perfect recall* and *implicit flow*, which are security assumptions about the power of the attacker. This can be controversial: in general one can choose to impose these or not. We *did* impose them because we have argued extensively elsewhere (Morgan 2006, 2009) that a compositional definition of program refinement is not possible otherwise[†]. Perfect recall in particular, however, does seem a good fit for HMMs independently of the refinement argument, since the knowledge

[†] We did not have space to repeat those arguments here.

gained from observations, once emitted from the output-side of an HMM, cannot be expunged from the attacker's repertoire by any kind of overwriting subsequently.

Our second contribution was to work-around ($\sqsubseteq$)-incompleteness by using an alternative, more specialised order ($\preccurlyeq$), showing that a program algebra including iteration is feasible (Section 9); and our third contribution was to argue by example that the resulting source-level reasoning is promising (Section 10).

There are two immediate prospects for further work. One that in practice we would like to answer questions like the one posed in Section 10.2 for general guesses of size N and large password spaces $\mathcal{P}$, and many other similar. For this we would need tool support both for the semantics (i.e. given a program, determine its meaning) and for establishing refinement (i.e. whether this meaning refined by that one) in a probabilistic setting (Katoen *et al.* 2010; McIver *et al.* 2009).

The other prospect is to complete our semantic space to proper measures, in fact to follow the approach outlined in Section 6.2. Beyond compositionality of ($\sqsubseteq$) we want its compositional *closure*, already achieved for straight-line programs, guaranteeing that the refinement relation is not unnecessarily strong; but that argument required (analytical) closure/compactness of a set of finite, discrete probability distributions in a metric space (McIver *et al.* 2010); and to do that here, with the extra feature of iterations that generate chains of approximants, seems to make the move to measures inevitable.

Finally, our longer-term aim is to add demonic choice to the model for e.g. demonic scheduling that takes into account what the adversary can, and cannot see Alvim *et al.* (2010). We have done this for qualitative systems (Morgan 2006, 2009) and we have earlier combined demonic and probabilistic choice *without* hiding (He *et al.* 1997; Morgan *et al.* 1996; McIver and Morgan 2005). The technique of convex closure, useful for that, generates uncountably many interpolated distributions: it is a second reason we are likely to need measures, and so we hope to exploit the structures developed for this paper at that later point.

References

Wikipedia (2011) `en.wikipedia.org`.

Aldini, A. and Pierro, A. D. (2004) A quantitative approach to noninterference for probabilistic systems. *Electronic Notes in Theoretical Computer Science* **99** 155–182.

Alvim, M., Andres, M. and Palamidessi, C. (2010) Information flow in interactive systems. In: Proceedings 21st CONCUR. *Springer Lecture Notes in Computer Science* **6269** 102–116.

Ash, R. B. (1972) *Real Analysis and Probability*, Academic Press.

Back, R. J. and von Wright, J. (1998) *Refinement Calculus: A Systematic Introduction*, Springer.

Backes, M. and Pfitzmann, B. (2002), Computational probabilistic non-interference. In: 7th European Symposium on Research in Computer Security. *Lecture Notes in Computer Science* **2502** 1–23.

Bishop, C. (2006) *Pattern Recognition and Machine Learning*, Information Science and Statistics, Springer.

Braun, C., Chatzikokolakis, K. and Palamidessi, C. (2008) Compositional methods for information-hiding. In: Proceeding FOSSACS'08. *Springer Lecture Notes in Computer Science* **4962** 443–457.

Braun, C., Chatzikokolakis, K. and Palamidessi, C. (2009) Quantitative notions of leakage for one-try attacks. In: Proceedings MFPS. *Electronic Notes in Theoretical Computer Science* **249** 75–91.

Chatzikokolakis, K. and Palamidessi, C. (2007) A framework for analyzing probabilistic protocols and its application to the partial secrets exchange. *Theoretical Computer Science* **389** (3) 512–27.

Chatzikokolakis, K., Palamidessi, C. and Panangaden, P. (2007) Probability of error in information-hiding protocols. In: *Proceedings CSF*, IEEE Computer Society 341–354.

Cohn, H. (1974) A ratio limit theorem for the finite nonhomogeneous markov chains. *Israel Journal of Mathematics* **19** 329–334.

Deng, Y. and Du, W. (2009) Kantorovich metric in computer science: A brief survey. *Electronic Notes in Theoretical Computer Science* **253** (3) 119–133.

Dudley, R. (2002) *Real Analysis and Probability*, Cambridge University Press.

Giry, M. (1981) A categorical approach to probability theory. In: Categorical Aspects of Topology and Analysis. *Springer Lecture Notes in Mathematics* **915** 68–85.

Goguen, J. and Meseguer, J. (1984) Unwinding and inference control. In: *Proceedings IEEE Symposium on Security and Privacy*, IEEE Computer Society 75–86.

Halmos, P. (1973) The legend of von Neumann. *The American Mathematical Monthly* **80** (4) 382–394.

Halpern, J. and O'Neill, K. (2002) Secrecy in multiagent systems. In: *Proceedings 15th IEEE Computer Security Foundations Workshop* 32–46.

Hart, S., Sharir, M. and Pnueli, A. (1983) Termination of probabilistic concurrent programs, *ACM Transactions on Programming Language and Systems* **5** 356–380.

He, J., Seidel, K. and McIver, A. (1997) Probabilistic models for the guarded command language. *Science of Computer Programming* **28** 171–192.

Jones, C. and Plotkin, G. (1989) A probabilistic powerdomain of evaluations. In: *Proceedings of the IEEE 4th Annual Symposium on Logic in Computer Science*, Los Alamitos, Calif. Computer Society Press 186–195.

Jurafsky, D. and Martin, J. (2000) *Speech and Language Processing*, Prentice Hall International.

Katoen, J. P., McIver, A. K., Meinicke, L. A. and Morgan, C. C. (2010) Linear-invariant generation for probabilistic programs: Automated support for proof-based methods. In: *Proceedings Static Analysis Symposium*, Springer 390–406.

Köpf, B. and Basin, D. (2007) An information-theoretic model for adaptive side-channel attacks. In: *Proceedings 14th ACM Conference Computer and Communications Security.*

Kozen, D. (1985) A probabilistic PDL. *Journal of Computer and System Sciences* **30** (2) 162–178.

Malacaria, P. (2010) Risk assessment of security threats for looping constructs. *Journal of Computer Security* **18** (2) 191–228.

McIver, A., Gonzalia, C. and Morgan, C. (2009) Probabilistic affirmation and refutation: Case studies. In: *Proceedings Automatic Program Verification.*

McIver, A., Meinicke, L. and Morgan, C. (2010) Compositional closure for Bayes risk in probabilistic noninterference. In: Abramsky S. *et al.*, (eds.) Proceedings ICALP 2010. *Lecture Notes in Computer Science* **6199** 223–235. Extended abstract.

McIver, A. and Morgan, C. (2005) *Abstraction, Refinement and Proof for Probabilistic Systems*, Monographs in Computer Science, Springer, New York.

McIver, A. and Morgan, C. (2008) A calculus of revelations. Presented at VSTTE Theories Workshop. Available at `www.cs.york.ac.uk/vstte08/`.

McShane, E. (1937) Jensen's inequality. *Bulletin of the American Mathematical Society* **43** (8) 521–527.

Moggi, E. (1989) Computational lambda-calculus and monads. In: *Proceedings 4th Symposium LiCS* 14–23.

Morgan, C. (1994) *Programming from Specifications*, 2nd edition, Prentice-Hall. Available at `web.comlab.ox.ac.uk/oucl/publications/books/PfS/`.

Morgan, C. (1996) Proof rules for probabilistic loops. In: Jifeng, H., Cooke, J. and Wallis, P. (eds.) *Proceedings BCS-FACS 7th Refinement Workshop*, Workshops in Computing, Springer. Available at `ewic.bcs.org/conferences/1996/refinement/papers/paper10.htm`.

Morgan, C. (2005) Of probabilistic *wp* and *CSP*. In: Abdallah, A., Jones, C. and Sanders, J., (eds.) *Communicating Sequential Processes: The First 25 Years*, Springer.

Morgan, C. (2006) The shadow knows: Refinement of ignorance in sequential programs. In: Uustalu, T. (ed.) *Mathematics of Program Construction*, **4014** 359–378. Springer. (Treats *Dining Cryptographers*.)

Morgan, C. (2009) The shadow knows: Refinement of ignorance in sequential programs. *Science of Computer Programming* **74** (8) 629–653. (Treats *Oblivious Transfer*.)

Morgan, C. (2010) Compositional noninterference from first principles. *Formal Aspects of Computing* 1–24.

Morgan, C., McIver, A. and Seidel, K. (1996) Probabilistic predicate transformers. *ACM Transactions on Programming Language and Systems* **18** (3) 325–353. `doi.acm.org/10.1145/229542.229547`.

Mu, C. and Clark, D. (2009) Quantitative analysis of secure information flow via probabilistic semantics. In: *Proceedings ARES'09*, IEEE 49–57.

Nelson, G. (1989) A generalization of Dijkstra's calculus. *ACM Transactions on Programming Language and Systems* **11** (4) 517–561.

Roscoe, A. (1992) An alternative order for the failures model. *Journal of Logic Computation* **2** (5) 557–577.

Sabelfeld, A. and Sands, D. (2000) Probabilistic noninterference for multi-threaded programs. In: *13th IEEE Computer Security Foundations Workshop (CSFW'00)*, 200–214.

Smith, G. (2003) Probabilistic noninterference through weak probabilistic bisimulation. In: *16th IEEE Computer Security Foundations Workshop (CSFW'03)*.

Sonin, I. (2008) The decomposition-separation theorem for finite nonhomogeneous markov chains and related problems. In: *Markov Processes and Related Topics: A Festschrift for Thomas G Kurtz*, volume 4, IMS 1–15.

Tarski, A. (1955) A lattice-theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics* **5** 285–309.

van Breugel, F. (2005) The metric monad for probabilistic nondeterminism. Draft available at `http://www.cse.yorku.ca/~franck/research/drafts/monad.pdf`.

Volpano, D. and Smith, G. (1998) Probabilistic noninterference in a concurrent language. In: *11th IEEE Computer Security Foundations Workshop (CSFW'98)*, 34–43.

Wirth, N. (1971) Program development by stepwise refinement. *Communications of the ACM* **14** (4) 221–227.

Worrell, J. (2010) Private communication.

Yasuoka, H. and Terauchi, T. (2010) Quantitative information flow – verification hardness and possibilities. In: *Proceedings 23rd IEEE CSF Symposium*, 15–27.