Quantitative Information Flow with Monads in Haskell

Jeremy Gibbons
University of Oxford, UK

Annabelle McIver
Macquarie University, Australia

Carroll Morgan
University of New South Wales, and Data61 (CSIRO), Australia

Tom Schrijvers
KU Leuven, Belgium

Abstract: Monads are a popular feature of the programming language *Haskell* because they can model many different notions of computation in a uniform and purely functional way. Our particular interest here is the probability monad, which can be –and has been– used to synthesise models for probabilistic programming.

Quantitative Information Flow, or *QIF*, arises when security is combined with probability, and concerns the measurement of the *amount* of information that 'leaks' from a probabilistic program's state to a (usually) hostile observer: that is, not "whether" leaks occur but rather "how much?"

Recently it has been shown that *QIF* can be seen monadically, a 'lifting' of the probability monad from (simply) distributions to *distributions* of distributions — so called "hyper-distributions". Haskell's support for monads therefore suggests a synthesis of an executable model for *QIF*. Here we provide the first systematic and thorough account of doing that: using distributions of distributions to synthesise a model for Quantitative Information Flow in terms of monads in Haskell.

1 Introduction

In contexts where programs have access to or manipulate private information, assessing the control of how that information flows is an essential aspect of the verification task. In some, probably most cases *some* part of the secret must be released for the program to achieve anything useful at all — but it is

the unintended leaks, which could be avoided by more careful programming, that concern us here. Preventing them is enormously challenging, and the idea of quantifying the information that *does* flow was proposed by (Denning, 1982), (Millen, 1987), (Clark et al., 2005b) and others to provide a sound framework to enable the analysis of the severity and impact of such flows. Our model for Quantitative Information Flow, that is *QIF*, was originally expressed in terms of information-theoretic channels, even to the extent of measuring (change in) Shannon Entropy 'by default' as the method of choice (Millen, 1987); and as such it has been used successfully to assess flows related to confidentiality and privacy for relevant operational scenarios. More recently, however, this channel model has been generalised: more general entropies than Shannon's are applicable (Alvim et al., 2012), and flow can be defined for programs rather than just channels (McIver et al., 2010). (Programs can update the private information, whereas channels can only read it.) The key to this generalisation is hyper-distributions (McIver et al., 2010, 2014a) (§3 below) — based on 'hypers', *QIF* in computer programs can be given a monadic semantics that supports many different entropies.

This paper combines the monadic semantics with a second idea: that monads (§2.1 below) abstract and unify common features of "notions of computation" (Moggi, 1991) and that monadic features can be, and have been built into (functional) programming languages (Wadler, 1992). Putting those two ideas together encourages the synthesis of an implementation of a '*QIF*-aware' programming language. We present a prototype of such a language, which we have called "Kuifje".[1] The synthesis also provides an important tool for experimentation with and analysis of how information flows in real programming.

A helpful guide to what we present is the analogous, but more straightforward synthesis that arises from a simpler combination: that simple probabilistic semantics (i.e. without flow) is also monadic (Lawvere, 1962; Giry, 1981). The result is easy implementations of functional probabilistic programming languages (Ramsey and Pfeffer, 2002; Erwig and Kollmansberger, 2006; Kiselyov and Shan, 2009). Our work here benefits from that, because *QIF*-aware programming languages are probabilistic — the 'quantities' in the information flow are derived from the probabilities occurring in the programs and in the probability distributions over the hidden state to which they are applied.

In the development of our functional implementation we use the notion of monoids as a guiding structure. Indeed, we can see both the syntactic- and the

[1] "Kuifje" is the Flemish/Dutch name of Hergé's Tintin, and refers to his hairstyle: a quiff.

semantic domain of a simple programming language as a monoid, the former being free, so that its denotational semantics is then a monoid morphism. The initial-algebra representation of free monoids gives rise to a straightforward implementation of this denotational semantics as a functional-programming-style *fold*. Moreover, with 'fold fusion' (Malcolm, 1990; Hutton, 1999) we can easily derive an efficient implementation of the hyper-distribution semantics from its naïve specification.

2 Background

We begin by describing monads in general (briefly) as they are used in Computer Science (§2.1), how they relate to quantitative information flow (§2.2), how they are instantiated in the functional programming language Haskell (§2.3), and how their Haskell instantiation can be used to build the tools necessary for a probabilistic information-flow-aware programming-language implementation (§2.4). The Haskell-oriented reader may want to skip the theoretical background and jump straight to §2.3.

2.1 Monads

The mathematical structure of monads was introduced to Computer Science by Moggi in order to have a model of computation that was more general than the "gross simplification" of identifying programs with total functions from values to values, a view that "wipes out completely behaviours like non-termination, non-determinism or side effects..." (Moggi, 1991). Here we will be using that generality to capture the behaviour of programs that hide and, complementarily, leak information: and our particular focus will be on using the monadic facilities of the programming language Haskell (Peyton Jones, 2003) to illustrate our ideas (§2.3).

Moggi's insight was to model a "notion of computation" as a monad $\mathbb{T}$, in order to distinguish *plain* values of some type $\mathcal{A}$ from the computations $\mathbb{T}\mathcal{A}$ that *yield* such values. (Later, we will see $\mathbb{T}$ as a Haskell type-constructor.) Thus $\mathbb{T}$ might enrich sets $\mathcal{A}$ of values to $\mathcal{A}_\perp$ by adding an extra element $\perp$ denoting 'not a proper value', perhaps the simplest computational monad; or it might enrich $\mathcal{A}$ to $\mathbb{P}\mathcal{A}$, the subsets of $\mathcal{A}$, for modelling demonic choice; or it might enrich $\mathcal{A}$ to $\mathcal{W}^*\times\mathcal{A}$, pairing with a sequence of $\mathcal{W}$ values, for modelling 'writing' such as to a log file; or it might enrich $\mathcal{A}$ to $\mathbb{D}\mathcal{A}$, the discrete distributions on $\mathcal{A}$ which, below, will be our starting point here. We say "enrich" because in each case the original $\mathcal{A}$ can be found embedded

within $\mathbb{T}\mathcal{A}$: a plain value can always be seen as a degenerate computation. Thus $\mathcal{A}$ is found: within $\mathcal{A}_\perp$ as that subset of computations yielding a proper value; within $\mathcal{W}^*\times\mathcal{A}$ as the computations $(\langle\,\rangle, a)$ in which no writes have yet occurred; within $\mathbb{P}\mathcal{A}$ as the singleton sets $\{a\}$ that represent the (degenerate) demonic choice of only one value, i.e. no choice at all (Hobson's Choice); and $\mathcal{A}$ is found within $\mathbb{D}\mathcal{A}$ as the point distribution $\langle a\rangle$, the probabilistic choice 'certainly a', this time no *probabilistic* choice at all.[2]

For a general monad $\mathbb{T}$, the embeddings illustrated above are all instances of the *unit function*, written η and of type $\mathcal{A}\to\mathbb{T}\mathcal{A}$, so that in the above four cases $\eta\, a$ is (the proper value) a, (the nothing-yet-written) $(\langle\,\rangle, a)$, (the singleton) $\{a\}$, and (the point) $\langle a\rangle$, respectively. Whereas a pure function taking values from $\mathcal{A}$ to values in $\mathcal{B}$ has type $\mathcal{A}\to\mathcal{B}$, Moggi modelled an 'impure function' for a particular notion of computation $\mathbb{T}$ as a so-called *Kleisli arrow* $\mathcal{A}\to\mathbb{T}\mathcal{B}$ — that is, a pure function yielding a computation rather than a plain value. Thus partial functions are modelled as Kleisli arrows $\mathcal{A}\to\mathcal{B}_\perp$, 'writing functions' as Kleisli arrows $\mathcal{A}\to\mathcal{W}^*\times\mathcal{B}$, 'nondeterministic functions' as Kleisli arrows $\mathcal{A}\to\mathbb{P}\mathcal{B}$, and 'probabilistic functions' as Kleisli arrows $\mathcal{A}\to\mathbb{D}\mathcal{B}$.

If a monad $\mathbb{T}$ is to model a "notion of computation", then in particular it had better support sequential composition: if a program f takes a value $a{:}\mathcal{A}$ to some structure in $\mathbb{T}\mathcal{B}$, and it is followed by a compatible program g with the same notion $\mathbb{T}$ of computation, i.e. a Kleisli arrow of type $\mathcal{B}\to\mathbb{T}\mathcal{C}$, then that second program g must *act* as if it has type $\mathbb{T}\mathcal{B}\to\mathbb{T}\mathcal{C}$ if it is to be sequentially composed with f, since the input type of g, the second stage, must in some sense match the output type of the first stage f. This is achieved by defining a *Kleisli-lifting* $(-)^*$, which converts Kleisli arrow $g{:}\mathcal{B}\to\mathbb{T}\mathcal{C}$ to a 'lifted' version g^*, a function of type $\mathbb{T}\mathcal{B}\to\mathbb{T}\mathcal{C}$. Then the intuitive conclusion, that if f goes from $\mathcal{A}$ to $\mathcal{B}$ and g from $\mathcal{B}$ to $\mathcal{C}$ then their composition should go from $\mathcal{A}$ to $\mathcal{C}$, is realised by the *Kleisli composition* $g\bullet f$ of $f{:}\mathcal{A}\to\mathbb{T}\mathcal{B}$ and $g{:}\mathcal{B}\to\mathbb{T}\mathcal{C}$ defined as $g\bullet f = g^*\circ f$ having type $\mathcal{A}\to\mathbb{T}C$, using Kleisli-lifting $(-)^*$ and ordinary functional composition $(\circ)$ together.

A monad $\mathbb{T}$ can be seen as a *functor* which, among other things, means that for any function $h{:}\mathcal{A}\to\mathcal{B}$ there is another function $\mathbb{T}h{:}\mathbb{T}\mathcal{A}\to\mathbb{T}\mathcal{B}$ that behaves in a sensible way (in particular, respecting identity and composition). Then an alternative definition of Kleisli composition is to require that $\mathbb{T}$ have a *multiplication* operation $\mu{:}\mathbb{T}^2\mathcal{C}\to\mathbb{T}\mathcal{C}$. In that case, for Kleisi arrow $g{:}\mathcal{B}\to\mathbb{T}\mathcal{C}$ one can define g^* by $\mu\circ\mathbb{T}g$, and the Kleisli composition $g\bullet f$ of $f{:}\mathcal{A}\to\mathbb{T}\mathcal{B}$ and $g{:}\mathcal{B}\to\mathbb{T}\mathcal{C}$ becomes $\mu\circ\mathbb{T}g\circ f$, thus achieving the same end.

[2] For $a{:}\mathcal{A}$ we write $\langle a\rangle$ for the *point distribution* in $\mathbb{D}\mathcal{A}$ which assigns probability 1 to a and probability 0 to all other elements of $\mathcal{A}$ (like a one-sided die).

With suitable conditions on unit η and multiply μ, the two presentations, one in terms of Kleisli lifting and the other in terms of multiplication, are equivalent and we will use them interchangeably.

The precise conditions for operation $\mathbb{T}$ on sets $\mathcal{A}$ to be a monad are that $\mathbb{T}$ must act also on the functions between those sets, respecting typing (so that $\mathbb{T}f{:}\mathbb{T}\mathcal{A}{\to}\mathbb{T}\mathcal{B}$ when $f{:}\mathcal{A}{\to}\mathcal{B}$), and it must support families of functions $\eta_{\mathcal{A}}{:}\mathcal{A}{\to}\mathbb{T}\mathcal{A}$ and $\mu_{\mathcal{A}}{:}\mathbb{T}^2\mathcal{A}{\to}\mathbb{T}\mathcal{A}$ that satisfy also the following algebraic identities (for $f : \mathcal{A} \to \mathcal{B}$ and $g : \mathcal{B} \to \mathcal{C}$):

(a) $\mathbb{T}id_{\mathcal{A}} = id_{\mathbb{T}\mathcal{A}}$ — $\mathbb{T}$ respects identity.
(b) $\mathbb{T}g \circ \mathbb{T}f = \mathbb{T}(g{\circ}f)$ — $\mathbb{T}$ respects composition.
(c) $\eta_{\mathcal{B}} \circ f = \mathbb{T}f \circ \eta_{\mathcal{A}}$ — η is a natural transformation $1{\to}\mathbb{T}$.
(d) $\mu_{\mathcal{B}} \circ \mathbb{T}^2 f = \mathbb{T}f \circ \mu_{\mathcal{A}}$ — μ is a natural transformation $\mathbb{T}^2{\to}\mathbb{T}$.
(e) $\mu_{\mathcal{A}} \circ \mathbb{T}\mu_{\mathcal{A}} = \mu_{\mathcal{A}} \circ \mu_{\mathbb{T}\mathcal{A}}$ — μ is associative.
(f) $\mu_{\mathcal{A}} \circ \mathbb{T}\eta_{\mathcal{A}} = \mu_{\mathcal{A}} \circ \eta_{\mathbb{T}\mathcal{A}} = id_{\mathcal{A}}$ — η is unit of μ.

Identities (a,b) are the conditions for $\mathbb{T}$ to be a functor, and (c,d) require that the families η and μ form natural transformations, and finally (e,f) are the additional conditions for $(\mathbb{T}, \eta, \mu)$ to form a monad. In particular, it is a straightforward exercise to use these identities to verify that $(\bullet)$ and η form a monoid: we have $h{\bullet}(g{\bullet}f) = (h{\bullet}g){\bullet}f$ and $\eta{\bullet}f = f = f{\bullet}\eta$.

To make those identities more concrete and familiar, we describe in words what the analogous identities would be for the powerset monad $\mathbb{P}$ used to model non-deterministic, i.e. 'demonic' programs. They are (in the same order)

(a) The image of a set X through the identity function id is X itself.
(b) The image of a set X through the composition $g{\circ}f$ is the image through g of the image through f of X.
(c) Making a singleton set $\{x\}$ from x, and then applying f to all elements of that set, is the same as applying f to x first and then making a singleton set from that: in both cases you get the singleton set $\{fx\}$.
(d) Applying f to the *elements of the elements* of a set of sets (that is, applying f 'twice deep'), and then taking the distributed union of the result, is the same as taking the distributed union of the set of sets first, and then applying f to its elements (i.e. once deep). Starting e.g. from $\{\{x_1\}, \{x_2, x_3\}\}$ you get $\{f\,x_1, f\,x_3, f\,x_3\}$ in both cases.
(e) Applying distributed union to each of the elements of a *set of sets of sets* (i.e. *three* nested braces), and then taking the distributed union of that, is the same as taking the distribution union twice. For example, starting from $\{\{\{x_1\}\}, \{\{x_2, x_3\}, \{x_4\}\}\}$ the former gives

$\{\{x_1\}, \{x_2, x_3, x_4\}\}$ and then $\{x_1, x_2, x_3, x_4\}$, and the latter reaches the same result but via $\{\{x_1\}, \{x_2, x_3\}, \{x_4\}\}$ instead.

(f) Converting a set's elements to singleton sets, and then taking the distributed union, is the same as converting the set to a singleton set (of sets) and then taking the distributed union of that — and it is the identity in both cases. The former takes $\{x_1, x_2\}$ to $\{\{x_1\}, \{x_2\}\}$ and then (back) to $\{x_1, x_2\}$; and the latter goes via $\{\{x_1, x_2\}\}$.

2.2 Monads for probabilistic programming

Probabilistic computation is an instance of the more general monadic construction introduced in §2.1 just above, and it provides the essential 'numeric component' of moving from earlier, only qualitative descriptions of information flow (Cohen, 1977; Goguen and Meseguer, 1984) to *Quantitative Information Flow* (Denning, 1982; Millen, 1987; Gray, 1990; Clark et al., 2005b), abbreviated "*QIF*": the study of information leakage, typically from- or between computer programs, where –crucially– the amount of information leaked can be measured and compared. Thus in this section we concentrate on probabilistic computation alone; we move on to information flow in §3.

A (discrete) probabilistic computation over some $\mathcal{A}$ takes an initial state $a{:}\mathcal{A}$ to a final distribution $\alpha{:}\mathbb{D}\mathcal{A}$ — thus it has type $\mathcal{A}{\to}\mathbb{D}\mathcal{A}$. For example with $\mathcal{A}{=}\{h, t\}$ for the head and tail faces of a coin, a computation P that 'seeks heads' might be

$$\begin{array}{rll} Ph = & \langle h \rangle & \text{if heads already, don't flip any more} \\ Pt = & \langle h, t \rangle & \text{if } t \text{ keep flipping} \end{array} \tag{1}$$

where in general we write $\langle a, b, \cdots, z \rangle$ for the uniform distribution over the elements listed inside $\langle \cdots \rangle$. As a special case $\langle a \rangle$ is the point distribution on a.

To seek heads twice we run P twice, i.e. we compose P with itself; but, as already noted more generally, the simple $P{\circ}P$ would be type-incorrect because the second-executed P (the left one) expects an $\mathcal{A}$ but the first P delivers a $\mathbb{D}\mathcal{A}$. Let us write a distribution list $\langle \cdots \rangle$ with superscripts summing to 1 for a discrete distribution with those probabilities (rather than uniform), so that omitted superscripts are assumed to be equal; and we write $\neg p$ for $1{-}p$. Then, following the Kleisli approach, we lift the second P to P^*,

so that its argument is of type $\mathbb{D}\mathcal{A}$, and note that

$$\begin{array}{ll} & P^*\langle h^p, t^{\neg p}\rangle \\ = & \quad \{ \text{ definition } (-)^* \ \} \\ & (\mu \circ \mathbb{D}P)\langle h^p, t^{\neg p}\rangle \\ = & \quad \{ \text{ function composition } \} \\ & \mu(\ \mathbb{D}P\langle h^p, t^{\neg p}\rangle\ ) \\ = & \quad \{ \text{ definition } \mathbb{D}P\text{: see } \dagger \text{ below } \} \\ & \mu\langle\ \langle h\rangle^p, \langle h, t\rangle^{\neg p}\ \rangle \\ = & \quad \{ \text{ definition } \mu \text{ for monad } \mathbb{D}\text{: see } \ddagger \text{ below } \} \\ & \langle\ h^{(1+p)/2}, t^{(1-p)/2}\ \rangle\ , \end{array} \tag{2}$$

so that using P^* in its general form from (2) we can calculate

$$P^2 h = P^*(Ph) = P^*\langle h\rangle = P^*\langle h^1, t^0\rangle = \langle h^{(1+1)/2}, t^{(1-1)/2}\rangle = \langle h^1, t^0\rangle = \langle h\rangle\ ,$$

and again from (2) we have

$$P^2 t = P^*(Pt) = P^*\langle h, t\rangle = P^*\langle h^{1/2}, t^{1/2}\rangle = \langle h^{(1+1/2)/2}, t^{(1-1/2)/2}\rangle = \langle h^{3/4}, t^{1/4}\rangle\ .$$

For "see † below" we note that $\mathbb{D}P$ is in monadic terms the application of functor $\mathbb{D}$ to arrow h; in elementary probability this is the 'push forward' of h (Feller, 1971). For $f{:}\mathcal{A}{\to}\mathcal{B}$ and $\alpha{:}\mathbb{D}\mathcal{A}$ and $b{:}\mathcal{B}$ the *push forward* $\mathbb{D}f$ of f has type $\mathbb{D}\mathcal{A}{\to}\mathbb{D}\mathcal{B}$ and is defined

$$(\mathbb{D}f)\,\alpha\,b \qquad = \qquad \sum_{\substack{a:\,\mathcal{A}\\ fa=b}} \alpha\, a \qquad = \qquad \beta\, b \quad \text{say}\ ,^{3}$$

i.e. so that the probability βb assigned by the $\mathbb{D}\mathcal{B}$-typed distribution $\beta = \mathbb{D}f\,\alpha$ to the element b of B is the total probability assigned by α in $\mathbb{D}\mathcal{A}$ to those a's in $\mathcal{A}$ such that $fa = b$.

For "see ‡ below" we have that the definition of multiply μ for $\mathbb{D}$ is the 'squash' or 'distributed average' that takes a distribution of distributions to a single distribution. With $\mathcal{D}$ as our base set, [4] we take some $\Delta{:}\mathbb{D}^2\mathcal{D}$ and $d{:}\mathcal{D}$ and have that $\mu\Delta \in \mathbb{D}\mathcal{D}$ and [5]

$$\mu\,\Delta\,d \qquad = \qquad \sum_{\delta:\,\Delta} \Delta\delta \times \delta d \quad .^{6} \tag{3}$$

[3] Here we follow the practice of avoiding parentheses as much as possible (because otherwise there would be so many). We don't write $((\mathbb{D}f)\,\alpha)\ b$, because function application associates to the left (and thus would allow even $\mathbb{D}\ f\ \alpha\ b$); and we don't write $\mathbb{D}(f)(\alpha)(b)$, because functional application is (in this area) usually indicated by juxtaposition without parentheses. Note also that we are using currying, where what might be seen as a multi-argument function is instead written as higher-order functions taking their single arguments one at a time from left to right.

[4] This is a temporary departure from $\mathcal{A}, \mathcal{B}$ as typical base sets, because we need to use "capital Greek D" as an element of $\mathbb{D}^2\mathcal{D}$.

[5] Concerning (:) vs. ($\in$) — we use the former to introduce a bound variable, both in text and in formulae. With for example $x{\in}X$ we are instead referring to x, X that are already defined.

The above might seem quite general, particularly when there is a more conventional –and simpler– view of P in (1) above, the 'seek h' operation, as a Markov chain. We now look at that connection, strengthening the intuition for what the probability monad $\mathbb{D}$ and its η, μ are doing, and preparing ourselves for when their generality becomes more useful.

The Markov matrix for the head-seeking P is

$$\begin{array}{c} \\ \text{before } P \end{array} \begin{array}{c} \text{after } P \\ \begin{pmatrix} 1 & 0 \\ 1/2 & 1/2 \end{pmatrix} \end{array}$$

Such matrices are called *stochastic* because their rows sum to one: the rows above are (invisibly) labelled h above t, and the columns h before t; and each element of the matrix gives the probability that one application of P takes that row label to that column label. Thus for example h (upper row) is taken to h (left column) with probability 1; but t (lower row) is taken to h with probability only $1/2$. Now if the initial distribution of the coin faces were $\langle h^p, t^{\neg p} \rangle$, then –as is well known– the final distribution of coin faces is given by the vector-matrix product

$$(p\ \neg p) \begin{pmatrix} 1 & 0 \\ 1/2 & 1/2 \end{pmatrix} \quad = \quad \left(\frac{1+p}{2}\ \frac{1-p}{2} \right) \quad ,$$

agreeing with what was calculated above at (2). And the effect of two P's in succession, that is $P(P\langle h^p, t^{\neg p} \rangle)$, is therefore given by the matrix product

$$\left((p\ \neg p) \begin{pmatrix} 1 & 0 \\ 1/2 & 1/2 \end{pmatrix} \right) \quad \begin{pmatrix} 1 & 0 \\ 1/2 & 1/2 \end{pmatrix}$$

which, because of the associativity of matrix multiplication, can of course be written

$$(p\ \neg p) \quad \left(\begin{pmatrix} 1 & 0 \\ 1/2 & 1/2 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 1/2 & 1/2 \end{pmatrix} \right) \quad = \quad (p\ \neg p) \begin{pmatrix} 1 & 0 \\ 3/4 & 1/4 \end{pmatrix} \quad . \tag{4}$$

Now we can observe that although the type of P itself is $\mathcal{A} \to \mathbb{D}\mathcal{A}$, the type of 'multiply by the square matrix' in the examples above is $\mathbb{D}\mathcal{A} \to \mathbb{D}\mathcal{A}$ — that is, the matrix viewed 'row wise' is of type $\mathcal{A} \to \mathbb{D}\mathcal{A}$ but, viewed as a whole (and using the definition of matrix multiplication) it is of type $\mathbb{D}\mathcal{A} \to \mathbb{D}\mathcal{A}$.

[6] The definition of μ in the powerset monad $\mathbb{P}$ is analogous: it is a distributed union that takes a set of sets to a single set by removing one level of set braces. Specifically, if we think of a set as its characteristic predicate (a function from elements to booleans), then for a set of sets $X{:}\mathbb{P}^2\mathcal{A}$, we have $\mu\, X\, a = (\exists\, x{:}\mathbb{P}\mathcal{A}\,.\, X\, x \wedge x\, a)$ — that is, $a \in \mu\, X$ iff there exists some $x{:}X$ with $a \in x$.

Thus the matrix as a whole corresponds to the Kleisli lifting P^* of its rows P. Then (4) goes on to show that $P^*{\circ}P$ corresponds in turn to the matrix multiplication with itself of the matrix corresponding to P^*. [7]

The wide applicability of matrix algebra comes in part from the fact (fortuitiously) that it has this monadic structure. In general, as Moggi explained, monads contribute to flexibility of general program-algebraic operations as well — and that flexibility is what we use here to explore –and implement– a program algebra for *QIF*.

Summary We will use the discrete-distribution monad $\mathbb{D}$ which takes a set $\mathcal{A}$ to the set $\mathbb{D}\mathcal{A}$ of its discrete distributions. The unit η of $\mathbb{D}$ takes an element of $\mathcal{A}$ to the point distribution $\langle a \rangle$ on a; the multiply μ takes a distribution-of-distributions $\Delta{:}\mathbb{D}^2\mathcal{D}$ to its 'squash' as shown at (3) above, which is however equivalent to the 'smaller' summation over the support $\lceil \Delta \rceil$ of Δ only, thus

$$\mu\,\Delta\,d \qquad = \qquad \sum_{\delta:\,\lceil \Delta \rceil} \Delta\delta \times \delta d \quad , \tag{5}$$

where the *support* of Δ is those $\delta{:}\mathbb{D}\mathcal{D}$ such that $\Delta\delta \neq 0$. (This is important in Haskell because we can then represent distributions as *finite* lists over their support, omitting the possibly infinitely many elements of probability zero.)

2.3 Monads in Haskell

Monads were introduced into Haskell because their mathematical generality translates into expressive power. They are modelled as a type class *Functor* with a method *fmap*, and a subclass *Monad* of *Functor* providing two additional methods *return* and $(\gg\!=)$; the latter is pronounced "bind". Their methods' types are

$$\begin{aligned} &\mathit{fmap} \;\; :: (a \to b) \to m\ a \to m\ b \\ &\mathit{return} :: a \to m\ a \\ &(\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b \end{aligned}$$

where m and a,b correspond to the $\mathbb{T}$ and $\mathcal{A}$,$\mathcal{B}$ of Section 2.1. Function *fmap* is the functorial action of monad m, and *return* is the unit η of monad m, and $(\gg\!=)$ is Kleisli lifting (but with the two arguments swapped). The multiplication μ is modelled by

[7] This corresponds to the general Kleisli identity $(P^*{\circ}P)^* = P^*{\circ}P^*$.

$$join :: m\ (m\ a) \to m\ a$$

As noted above, Kleisli lifting and monad multiplication are interdefinable: therefore we have the further identities

$$\begin{array}{lcl} join\ x & = & x \gg\!= id \\ x \gg\!= f & = & join\ (fmap\ f\ x) \end{array}$$

All of those are just ordinary Haskell functions, defined in plain Haskell code, and it is straightforward to implement additional operations in terms of them. For example, Kleisli lifting with the arguments in the traditional order $(=\!\ll)$ is defined in terms of bind by

$$\begin{array}{l} (=\!\ll) :: (a \to m\ b) \to (m\ a \to m\ b) \\ g =\!\ll x \quad = \quad x \gg\!= g \end{array}$$

and backwards Kleisli composition $(<\!=\!<)$ (written "$\bullet$" in Section 2.1) and its forwards version $(>\!=\!>)$ are defined by

$$\begin{array}{l} (<\!=\!<) :: (b \to m\ c) \to (a \to m\ b) \to (a \to m\ c) \\ g <\!=\!< f \quad = \quad join \circ fmap\ g \circ f \\ (>\!=\!>) :: (a \to m\ b) \to (b \to m\ c) \to (a \to m\ c) \\ f >\!=\!> g \quad = \quad join \circ fmap\ g \circ f \end{array}$$

Clearly they are equal, operationally performing f 'and then' g; it is solely a matter of convenience which way around to write it.

As a final link back to the familiar, we mimic our use of the powerset monad $\mathbb{P}$ in §2.1 to give a similar presentation here of the monad laws in Haskell, but using this time the list monad — call it $\mathbb{L}$. Then the functorial action of $\mathbb{L}$ on a function $f :: a \to b$ is to produce $map\ f$ of type $[a] \to [b]$; unit η of $\mathbb{L}$ takes x to the one-element list $[x]$ containing just x, i.e. it is $(:[])$; and the multiply μ of $\mathbb{L}$ is the function *concat* that 'squashes' a list of lists into a single list. We now have

$$\begin{array}{lcll} id & = & map\ id & \text{— (a)} \\ map\ g \circ map\ f & = & map\ (g \circ f) & \text{— (b)} \\ map\ f \circ (:[]) & = & (:[]) \circ f & \text{— (c)} \\ concat \circ map\ (map\ f) & = & map\ f \circ concat & \text{— (d)} \\ concat \circ map\ concat & = & concat \circ concat & \text{— (e)} \\ concat \circ map\ (:[]) & = & concat \circ (:[]) = id & \text{— (f)} \end{array}$$

We will see see the benefit of all the above generality below, where these properties, and others, are exploited in a monadic treatment of *QIF* and the construction of a Haskell implementation of it.

2.4 Probabilistic programming in Haskell

With §2.2 and §2.3 as building blocks, we can model probabilistic programs in Haskell as Kleisli arrows for the distribution monad $\mathbb{D}$. Thus in this section we recall in more detail the previous work that shows how that is done (Ramsey and Pfeffer, 2002; Erwig and Kollmansberger, 2006; Kiselyov and Shan, 2009). (Note that we are still not addressing *QIF* itself.)

Representation We limit probabilities to the rationals; and our distributions are discrete, represented as (finite) lists of pairs with each pair giving an element of the base type and the probability associated with that element. Usually (but not always) the representation will be *reduced* in the sense that the list includes only elements of the support of the distribution, and each of those exactly once, i.e. excluding both repeated elements and elements with probability zero.

$$\textbf{\textit{type}}\ \mathit{Prob} = \mathit{Rational}$$
$$\textbf{\textit{newtype}}\ \mathit{Dist}\ a = D\ \{\mathit{runD} :: [(a, \mathit{Prob})]\}$$

The 'essence' of the representation is the type $[(a, \mathit{Prob})]$, i.e. lists of pairs. The $D\ \{\mathit{runD} :: \ldots\}$ above is Haskell notation for defining $\mathit{Dist}\ a$ as a record type, with a single field called *runD* that consists of a list of pairs, and introduces function $D :: [(a, \mathit{Prob})] \to \mathit{Dist}\ a$ to wrap up such a list as a record value. The field-name *runD* can be used as a function, to extract the 'bare' list of pairs.

Monad instance The make-probabilistic-monad functor is the type constructor *Dist* above, whose argument type a is the base type of the distribution. The unit *return* of the monad takes an element to the point distribution on that element. The bind operator $(\gg\!=)$ takes a distribution $d :: \mathit{Dist}\ a$ and a distribution-valued function $f :: a \to \mathit{Dist}\ b$ and computes effectively the application of the stochastic matrix f to the initial distribution d. [8]

$$\begin{aligned}&\textbf{\textit{instance}}\ \mathit{Monad}\ \mathit{Dist}\ \textbf{\textit{where}}\\ &\quad \mathit{return}\ x = D\ [(x, 1)]\\ &\quad d \gg\!= f\ = D\ [(y, p \times q) \mid (x, p) \leftarrow \mathit{runD}\ d, (y, q) \leftarrow \mathit{runD}\ (f\ x)]\end{aligned}$$

Note that $(\gg\!=)$ can produce representations that are not reduced, since values of y can be repeated between the supports of different distributions $f\ x$ as x itself varies over the support of d. We will arrange to reduce the representations where necessary.

[8] Think of f as taking a stochastic-matrix row-label to the distribution of column-label probabilities in that row.

Operations Function *uniform* constructs a discrete distribution from a non-empty list of elements, assigning the same probability to each element. (It is not defined on the empty list.)

$$
\begin{array}{l}
\mathit{uniform} :: [a] \to \mathit{Dist}\ a \\
\mathit{uniform}\ l = D\ [(x, 1\ /\ \mathit{length}\ l) \mid x \leftarrow l]
\end{array}
$$

The three-argument function $(-\ {}_{-}\!\oplus\ -)$ takes a probability p and two elements and from them constructs a distribution in which the first element has probability p and the second $1-p$.

$$
\begin{array}{l}
-\ {}_{-}\!\oplus\ - :: \mathit{Prob} \to a \to a \to \mathit{Dist}\ a \\
x\ {}_{p}\!\oplus\ y = D\ [(x, p), (y, 1-p)]
\end{array}
$$

Again, the essence here is the two-element list $[(x, p), (y, \neg p)]$ of course. That is, the distribution $x\ {}_{p}\!\oplus\ y$ generally has two-element support—unless x and y coincide, in which case it is a point distribution (in unreduced form).

Reduction Function *reduction* removes duplicates and zeroes from the concrete representation of a distribution. That does not change the abstract distribution represented, but makes its handling more efficient. In the sequel it will be built-in to monadic compositions (such as $(\gg\!=)$ above, and $(>\!=\!>)$ etc.)

$$
\begin{array}{l}
\mathit{unpackD} :: \mathit{Ord}\ a \Rightarrow \mathit{Dist}\ a \to [(a, \mathit{Prob})] \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{— Recover list representation, reduced.} \\
\mathit{unpackD} = \mathit{removeDups} \circ \mathit{removeZeroes} \circ \mathit{runD} \\
\quad \textbf{\textit{where}} \\
\qquad \mathit{removeZeroes} = \mathit{filter}\ (\lambda(x, p) \to p \neq 0) \\
\qquad \mathit{removeDups} = \mathit{toList} \circ \mathit{fromListWith}\ (+) \\
\mathit{reduction} :: \mathit{Ord}\ a \Rightarrow \mathit{Dist}\ a \to \mathit{Dist}\ a \\
\mathit{reduction} = D \circ \mathit{unpackD} \quad \text{— Unpack and then repack.}
\end{array}
$$

Here (from right to left) function *runD* extracts the essence, namely the list of pairs; then *removeZeroes* and *removeDups* reduce the list; and finally D replaces the reduced list within the constructor, where it began. The Haskell library function *fromListWith* requires an ordering on the elements, so it can make use of a binary search tree in order to remove duplicates in $\mathrm{O}(n \log n)$ time; one could instead require only element equality rather than ordering, at the cost of $\mathrm{O}(n^2)$ time.

$$\text{channel:} \qquad \text{input} \begin{cases} \\ 0 \\ 1 \end{cases} \overbrace{\begin{matrix} 0 & 1 \\ .99 & .01 \\ .02 & .98 \end{matrix}}^{\text{output}} \begin{matrix} \\ \leftarrow \text{sums to 1} \\ \leftarrow \text{sums to 1} \end{matrix}$$

Figure 1 Channel matrix describing not-quite-perfect transmission

$$\text{channel:} \qquad \text{uniform input} \begin{cases} \\ 0\ ^{1}/_{2} \\ 1\ ^{1}/_{2} \end{cases} \overbrace{\begin{matrix} 0 & 1 \\ .495 & .005 \\ .01 & .49 \end{matrix}}^{\text{output}} \leftarrow \text{sums to 1 overall}$$

Figure 2 Joint distribution between input and output, based on Fig. 1 and uniform input

3 *QIF* and hyper-distributions

With the above preliminaries, we can now move to quantitative information flow.

3.1 Background

In the communication channels of (Shannon, 1948), information flows in at the source, is transmitted but possibly with errors introduced, and then flows out at the target. Typically the error-rate is described mathematically as a channel *matrix* that gives explicit probabilities for message corruption. For example, Fig. 1 shows a channel matrix representing correct transmission with high probability only — if a 0 is input, there is a 1% probability that a 1 will come out instead; and for a 1 input, the probability of error is 2%. As noted earlier, the matrix is called *stochastic* because its rows sum to one.

Analyses of channels like Fig. 1 assume a distribution over the inputs, and look at the correlation with the resulting output. For example, if we assume a uniform input distribution on $\{0, 1\}$, then the joint distribution matrix between inputs and outputs would be as in Fig. 2, obtained by multiplying the input probabilities ($^{1}/_{2}$ here) along the rows. Here it's the matrix as a whole that sums to 1, not the individual rows.

The 'information flow' due to such channel matrices is often expressed as

$$\text{channel:} \qquad \text{input} \begin{cases} \\ 0 \\ 1 \end{cases} \overbrace{\begin{matrix} 0 & 1 \\ .51 & .49 \\ .49 & .51 \end{matrix}}^{\text{output}} \begin{matrix} \\ \leftarrow \text{sums to 1} \\ \leftarrow \text{sums to 1} \end{matrix}$$

Figure 3 Channel matrix describing a 1% success-rate

the difference between the Shannon Entropy of the input before- and after the output is observed. [9] In this example the Shannon Entropy beforehand is the entropy of the prior distribution: simply 1 bit, because that distribution is uniform over two values. The Shannon Entropy *afterwards* is however the *conditional* Shannon Entropy obtained by averaging the Shannon Entropies of the posterior distributions, with the weights of that averaging being their probability of occurrence. In Fig. 2 for example, the marginal probability that a 0 is output is .495+.01 = .505 and the conditional distribution on the input is then the normalised column for that output, i.e. that it was 0 with probability ${}^{.495}/_{.505} = .980$ and 1 with probability .020, which distribution has Shannon Entropy .140. Similarly, for output 1 the marginal probability is .495 and the conditional probability on the inputs is 0 with probability ${}^{.005}/_{.495} = .010$ and output 1 with probability .990, with Shannon Entropy .081. The conditional Shannon Entropy overall is thus the weighted average of those two, that is $.505\times.140 + .495\times.081 = .111$, and the number of bits transmitted is the entropy-before vs. conditional-entropy-afterwards difference, that is $1-.111 = .889$ bits in this case. It's a pretty good channel from the communication point of view.

When we look at this from the point of view of computer security however, it's a terrible channel: it has leaked almost all (0.889 bits) of the secret (1 bit). Far better for secrets would be the channel of Fig. 3 that has only a very slight bias induced on the output with respect to the input. A similar calculation for the above shows the conditional Shannon Entropy on the uniform input in this case to be 0.9997, so that this channel leaks only 0.0003 bits. This is more like what we want for our secure programs. (For communication, however, it's Fig. 3 that is a terrible channel.) Thus although the mathematics for flow of communications and for flow of secrets (leaks) from programs is the same, the interpretation is complementary.

In spite of that correspondence, it turns out that Shannon Entropy is sometimes (even often) very far from the best way to measure entropy

[9] The Shannon Entropy of a p vs. $1-p$ distribution is $-(p\lg(p) + (1-p)\lg(1-p))$.

					Shannon Entropy	Bayes Risk	Bayes Vulnerability
password →	A	a	9	%	$H()$	$R()$	$V() = 1-R()$
π_1 probability	$1/2$	$1/2$	0	0	1	$1/2$	$1/2$
π_2 probability	$2/3$	$1/9$	$1/9$	$1/9$	∼1.45	$1/3$	$2/3$

Distribution π_2 has more Shannon Entropy than π_1, but still π_2 is an easier target than π_1 for the one-guess password hacker.

Figure 4 Entropies for two example password distributions

where leaks from computer programs are concerned. (Smith, 2009) argued that in some cases it might be better to measure (in-)security by using instead the maximum probability that the secret could be guessed: an intelligent attacker would guess the secret(s) she knows to have the largest probability: this is called *Bayes Vulnerability*. As an entropy (but not *Shannon* entropy) this is expressed as a real-valued function that takes a distribution to $1 -$ *maximum probability*, where the subtraction from one $(1-)$ is a technical device that ensures increasing disorder leads to increasing entropy: that function is called *Bayes Risk*. On a fixed state space X of size N, the maximum Bayes Risk is $1-1/N$ and –as for Shannon Entropy– occurs on the uniform distribution.

We illustrate the difference with an example of two possible distributions of passwords. In one distribution, people choose alphabetic passwords uniformly, but never use numbers or punctuation. In the other, people mostly choose their friends' names (which are alphabetic) but, among those who do not, the choice is uniform between all other passwords including other alphabetics and those with numbers and punctuation.

To keep the calculations simple, we will suppose the password space is the four-element set $\{\mathsf{A}, \mathsf{a}, \mathsf{9}, \mathsf{\%}\}$ for names, alphabetics generally, numbers and, finally, other punctuation. Suppose that the first distribution has probabilities (in that order) of $(1/2, 1/2, 0, 0)$ and that the second distribution is $\{(2/3, 1/9, 1/9, 1/9)\}$. In Fig. 4 we see a tabulation of the Shannon Entropy H and Bayes Risk R for these two distributions.

Distribution π_1 has Shannon Entropy $H(\pi_1)$=1 and Bayes Risk $R(\pi_1)=1/2$. For π_2 we find $H(\pi_2) = 2/3 \lg(3/2) + 3 \cdot 1/9 \lg(9) \approx 1.45$, and $R(\pi_2) = 1-2/3 = 1/3$. Thus 'from Shannon's point of view' distribution π_2 is more secure than

π_1 — that is $H(\pi_2) > H(\pi_1)$. But Bayes would say the opposite, because $R(\pi_2) < R(\pi_1)$ or, equivalently, we have $V(\pi_2) > V(\pi_1)$, indicating that π_2 is *more* vulnerable, not less. In summary, distribution π_2 has *more* Shannon Entropy, i.e. more information-theoretic disorder than π_1, but still is more vulnerable to a a one-guess attacker than π_1 is.

But it does not stop there: a later development (Alvim et al., 2012) was the further generalisation of entropies to not just two (Shannon, Bayes) but in fact an infinite family of them based on 'gain functions' that captured the economics of the attacker: How much is the secret worth to her? Both Shannon Entropy and Bayes Risk are instances of these gain functions. [10] Then a final generalisation (McIver et al., 2015) (for now) was to recognise that, as functions, gain-function-determined entropies are characterised simply by being concave and continuous over distributions. [11]

Pulling back however from the brink of runaway generalisation, we simply note that, whatever entropy is being used, it is applied to the prior before the channel, and applied conditionally to the resulting joint distribution between input and output determined after the channel. In that latter case, the joint distribution is regarded as a distribution (the output marginal) over posterior distributions obtained by conditioning the input distribution on the event that each particular output has occurred (the normalised joint-matrix columns).

It is 'distributions on posterior distributions' that have been named *hyper-distributions* (McIver et al., 2014a, 2010), a conceptual artefact synthesised by asking "If we allow entropies from a general class of functions f on distributions (including Shannon Entropy and Bayes Risk as special cases), what operation do we carry out to determine the f-leakage of a channel with respect to a particular prior?" The answer is "Apply the entropy to the prior, and take the expected value of the same entropy over the hyper-distribution produced by the channel's acting on that prior; then compare the two numbers obtained." [12]

That is why we focus on hyper-distributions as a unifying concept for *QIF* — it is the common feature of determining leakage in the generalised setting explained above, where there are (infinitely) many possible entropies to consider. In other work we discuss extensively how hyper-distributions

[10] A technical detail for those familiar with (Alvim et al., 2012) — Shannon Entropy requires an infinitary gain funtion, and in fact is more neatly expressed with the complementary 'loss functions' that express how much the attacker has to lose.

[11] This generalisation was introduced in (McIver et al., 2015) as *loss* functions' determining 'uncertainty measures' that were continuous and *concave*.

[12] Popular comparisons include 'additive leakage', where you subtract the former from the latter, and 'multiplicative leakage' where you take the quotient.

$$\text{perfect channel:} \qquad \text{skewed input} \left\{ \begin{array}{c} \overbrace{}^{\text{output}} \\ \begin{array}{cc} 0 & 1 \end{array} \\ \begin{array}{c} 0^{\,0.9} \\ 1^{\,0.1} \end{array} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \end{array} \right.$$

The resulting hyper is $0.9{\times}(1,0) + 0.1{\times}(0,1)$, indicating that with probability 0.9 the adversary will know the input was 0, and with probability 0.1 she will know that it was 1.

Figure 5 Perfect channel, 0-skewed input

$$\text{nearly perfect channel:} \qquad \text{skewed input} \left\{ \begin{array}{c} \overbrace{}^{\text{output}} \\ \begin{array}{cc} 0 & 1 \end{array} \\ \begin{array}{c} 0^{\,0.9} \\ 1^{\,0.1} \end{array} \begin{pmatrix} 1 & 0 \\ .01 & .99 \end{pmatrix} \end{array} \right.$$

Figure 6 Nearly perfect channel, 0-skewed input

lead to a robust notion of a program-refinement–like 'leakage ordering' ($\sqsubseteq$) on *QIF* programs (McIver et al., 2014a, 2010, 2015).

An example of that unification is the comparison of a perfect channel with a nearly perfect channel applied to the same non-uniform prior. We will appeal to two different entropies.

The point is not the precise numbers in the hypers, but rather that those hypers, as an abstraction, contain *all* the information needed to calculate the (conditional) Shannon Entropy, the Bayes Risk, and indeed any other well behaved entropy we require. For the perfect Fig. 5 the conditional Shannon Entropy is $0.9H(1,0) + 0.1H(0,1) = 0$, and the Bayes Risk is $0.9R(1,0) + 0.1R(0,1) = 0$. In both cases the observer is in no doubt about the input's value.

For the nearly perfect Fig. 6 however, the conditional Shannon Entropy is

$$0.901H(0.999, 0.001) + 0.099H(0,1) \qquad = \qquad 0.02 \quad ,$$

and the Bayes Risk is

$$0.901R(0.999, 0.001) + 0.099R(0,1) \qquad = \qquad 0.001 \quad .$$

In this case there is some residual doubt — but not much.

3.2 Hyper-distribution and generalised entropies seen monadically

We now bring our two conceptual threads together: monads and *QIF*.

In §2.1 we saw that monads capture a notion of computation $\mathbb{T}$ by lifting a base-type $\mathcal{A}$ to a type $\mathbb{T}\mathcal{A}$ of computations over $\mathcal{A}$; and in §2.2 we saw how in particular the distribution monad $\mathbb{D}$ lifted a type to the probabilistic computations on that type. Our approach to *QIF* is based on that; but to get the extra expressive power, i.e. to describe not only probability but information flow, we lift base-type $\mathbb{D}\mathcal{A}$, rather than $\mathcal{A}$ itself. The 'ordinary' probabilistic computations of type $\mathcal{A}\to\mathbb{D}\mathcal{A}$ are replaced by computations of type $\mathbb{D}\mathcal{A}\to\mathbb{D}(\mathbb{D}\mathcal{A})$, or $\mathbb{D}\mathcal{A}\to\mathbb{D}^2\mathcal{A}$ of 'information-flow aware' probabilistic computations on $\mathcal{A}$. For that we are using the same monad $\mathbb{D}$ as before, but we are starting 'one level up'. It is the $\mathbb{D}^2\mathcal{A}$ type that we have called the *hyper-distributions* on $\mathcal{A}$, or "hypers" for short.

The first application of $\mathbb{D}$ takes us, as we saw in §2.2 and have seen in other work in this area, to probabilistic computations — but with just the one $\mathbb{D}$ we have only the probabilistic computations that are information-flow *unaware*. For information-flow awareness we need the extra level that hypers provide, so that we can do the before-after comparisons described in §3.1 just above.

We will see that it is the unit η that takes say a distribution $\delta{:}\mathbb{D}\mathcal{A}$ to a particular hyper $\langle\delta\rangle$ that is information-flow *aware*, but in a degenerate sense, as before: it is aware of nothing. In fact it is only the point distribution on the original distribution δ, and so this hyper will in fact denote the (effect of) a probabilistic program that leaks nothing, just as η earlier gave us possibly improper programs that in fact are proper, sequence-writing programs that have not yet written, demonic programs that behave deterministically, and probabilistic programs that use only one-sided dice. This is indeed the kind of embedding that we mentioned before.

The multiplication μ, on the other hand, will now have a more interesting role beyond its technical use in defining Kleisli composition. Given a proper (i.e. not necessarily point) hyper Δ in $\mathbb{D}^2\mathcal{A}$, the squash $\mu\Delta$ of type $\mathbb{D}\mathcal{A}$ is the result of the 'simple' probabilistic program from which all information-flow properties have been removed. In §7.1 we discuss the antecedents of that approach.

Thus monadic *QIF* programs for us will have type $\mathbb{D}\mathcal{A}\to\mathbb{D}^2\mathcal{A}$. If we express a channel C that way, in the style of §3.1, and pick some uncertainty function

f, then on a prior π the uncertainty beforehand is $f\pi$ — and after observing the output of channel C it is $(\mu \circ \mathbb{D}f)(C\pi)$. One then either subtracts or divides one from/by the other to obtain the additive- or multiplicative f-leakage of C applied to π.

But a further advantage of this extra structure, introduced by the 'second $\mathbb{D}$', is that we can represent –within the same type– both

Markov program steps that change the state probabilistically, but leak no information while they do that, and the complementary

Channel program steps that leak information but do not change the state while they do that.

A (pure) Markov program M (for "Markov") will be of type $\mathbb{D}\mathcal{A} \to \mathbb{D}^2\mathcal{A}$ and has the characteristic that for any initial distribution δ we find that $M\delta = \eta\delta'$ for some δ', i.e. that the result of the M is always a *singleton* hyper. And indeed M as a matrix would take row δ, encoding the initial distribution, to row δ' encoding the final distribution. On the other hand, a (pure) channel C again has type $\mathbb{D}\mathcal{A} \to \mathbb{D}^2\mathcal{A}$ (the same as a Markov program), but in addition has the characteristic that $\mu(C\delta) = \delta$, i.e. that the state distribution is not changed: all that is possible is that information can leak.

Recall that the trivial channel that leaks nothing, treating all inputs the same [13] takes any prior to the singleton hyper on that prior: that is, its action is $\pi \mapsto \langle \pi \rangle$, indicating that after the channel has run the adversary knows for sure that the distribution on the input was... just π, which she knew before.

On the other hand, the channel that leaks everything (the identity matrix) takes input π in general to a *non*-singleton hyper whose support is (only) point distributions: for example the prior $(2/3, 1/3)$ is taken by that channel to the hyper $2/3\langle (1,0) \rangle + 1/3\langle (0,1) \rangle$, indicating that with probability $2/3$ the adversary will know for sure that the input was 0 and with probability $1/3$ she will know that it was 1. That is just what a 'leak everything' channel should do.

With the above as a starting point, Kleisli composition of hyper programs allows the two possibilities above to be combined sequentially, while still remaining within the same type, even though we are now one level up at $\mathbb{D}\mathcal{A}$ rather than $\mathcal{A}$: that's what the generality of Kleisli composition does for us automatically. That is, we can *without any further definitions* write $C; M$ for a *QIF*-aware program that leaks information *and then* changes the state;

[13] Concretely this is any channel matrix all of whose rows are equal: the nicest formulation is a 0-column matrix, i.e. there is no output at all; but slightly less shocking is a one-column matrix of all 1's that gives the same output for all inputs. With more than one column, there can clearly be more than one output: but since the rows are the same, the relative frequencies of the outputs gives no information about which input produced them.

and similarly a program $M;C$ changes the state first and then leaks that. [14] Either way, the result is a single function of type $\mathbb{D}\mathcal{A}\to\mathbb{D}^2\mathcal{A}$. Further, the order-theoretic structure of the space $\mathbb{D}\mathcal{A}$, e.g. limits, allow us to introduce smoothly all the apparatus of sequential programming, including loops — with the probabilistic- and the *QIF* features added in 'for free' (McIver et al., 2014b). [15]

4 A concrete programming language *Kuifje*, and its semantics

In §3 we introduced Markov steps and channel steps as program fragments, but abstractly. Both are of type $\mathbb{D}\mathcal{A}\to\mathbb{D}^2\mathcal{A}$ but, as we saw, each has further specific and complementary properties: the Markov step releases no information, and the channel step makes no change to the state. In this section we give concrete notations for these program steps, together with constructions like conditional and iteration for making more substantial programs. The functional components from §2.4 above are used for that.

The presentation takes the form of a sequence of three small languages, of increasing expressivity. We start in §4.1 with a simple imperative command language CL consisting of assignment statements, conditionals, and loops; in §4.2 we add probabilistic assignment statements, yielding the probabilistic command language PCL (essentially *pGCL* of (McIver and Morgan, 2005)); and finally in §4.3 we add observations, yielding our complete *QIF* language Kuifje.

We use an initial-algebra representation for the syntax of these languages because it enables a straightforward implementation of their denotational semantics as *fold*s. Moreover, the representation pays off in §5 where it allows us easily to derive an efficient implementation of the hyper-distribution semantics from its naïve specification.

4.1 Basic language CL

Concrete syntax We will start without probabilistic or *QIF* features, giving a basic *Command Language*, or CL for short, that is just the usual "toy imperative language" in which we can write programs like the following:

[14] Observe that the combination $C;M$ corresponds to a step of a Hidden Markov Model (Baum and Petrie, 1966): in *HMM*'s first some information about the current state is released and then the state is probabilistically updated. Here we arrange for those two effects to be conceptually separated, which allows them to be put together in any combination.

[15] We do not discuss the *QIF* order theory in this paper, however.

```
y := 0;
while (x > 0) {
  y := y + x;
  x := x - 1;
}
```

When this program is run with an initial state where `x` $= 3$, its final state has `x` $= 0$ and `y` $= 0 + 3 + 2 + 1 = 6$.

Abstract syntax The obvious representation of a CL program over a state space of type S is as the type $[\mathit{Instruction}\ S]$ of lists of instructions acting on S, where each instruction is either a state update, a conditional with a guard and two branches, or a loop with a guard and a body:

$$
\begin{aligned}
&\textbf{\textit{type}}\ \mathit{CL}\ s = [\mathit{Instruction}\ s] \\
&\textbf{\textit{data}}\ \mathit{Instruction}\ s \\
&\quad = \mathit{UpdateI}\ (s \to s) \\
&\quad \mid \mathit{IfI}\ (s \to \mathit{Bool})\ (\mathit{CL}\ s)\ (\mathit{CL}\ s) \\
&\quad \mid \mathit{WhileI}\ (s \to \mathit{Bool})\ (\mathit{CL}\ s)
\end{aligned}
$$

However, because the mutual recursion between instructions and programs would cause complications later on, we choose instead a 'continuation-style' representation that is directly recursive. Each of its constructors takes an additional argument representing 'the rest of the program', and there is an additional constructor *Skip* representing the 'empty' program:

$$
\begin{aligned}
&\textbf{\textit{data}}\ \mathit{CL}\ s \\
&\quad = \mathit{Skip} \\
&\quad \mid \mathit{Update}\ (s \to s)\ (\mathit{CL}\ s) \\
&\quad \mid \mathit{If}\ (s \to \mathit{Bool})\ (\mathit{CL}\ s)\ (\mathit{CL}\ s)\ (\mathit{CL}\ s) \\
&\quad \mid \mathit{While}\ (s \to \mathit{Bool})\ (\mathit{CL}\ s)\ (\mathit{CL}\ s)
\end{aligned}
\tag{6}
$$

In particular, $\mathit{CL}\ S$ is isomorphic to $[\mathit{Instruction}\ S]$.

For instance, in the example program above we could use the state

$$\textbf{\textit{data}}\ S = S\ \{\_x :: \mathit{Int}, \_y :: \mathit{Int}\}$$

(a record with two fields), which would allow us to render the above example as follows in Haskell: [16]

[16] Note that notation like $s.y := (s.y + s.x)$, with the obvious meaning, is not pseudo-code, but more familiar rendering of valid Haskell code based on `lens` library[17] operators. We refer the interested reader to this paper's companion code for the details.

[17] `https://hackage.haskell.org/package/lens`

$$
\begin{array}{l}
\mathit{example}_1 :: \mathit{CL}\ S \\
\mathit{example}_1 = \\
\quad \mathit{Update}\ (\lambda s \to s.y := 0) \\
\quad\quad (\mathit{While}\ (\lambda s \to s.x > 0) \\
\quad\quad\quad (\mathit{Update}\ (\lambda s \to s.y := (s.y + s.x)) \\
\quad\quad\quad\quad (\mathit{Update}\ (\lambda s \to s.x := (s.x - 1)) \\
\quad\quad\quad\quad\quad \mathit{Skip})) \\
\quad\quad \mathit{Skip})
\end{array}
\tag{7}
$$

We now discuss the constructions individually. *Skip* denotes the empty program. The program $\mathit{Update}\ f\ p$ denotes the program that first transforms the state with f and then proceeds with program p. Program $\mathit{If}\ c\ p\ q\ r$ checks whether the current state satisfies predicate c, and proceeds with p if it does and with q if it does not; in either case, it subsequently continues with r. Finally, the program $\mathit{While}\ c\ p\ q$ checks whether the current state satisfies the predicate c; if it does, it executes p and then repeats the while loop, and if it does not, it continues with q.

We now continue by using the abstract syntax to define several basic combinators that will allow us to write programs more compactly:

$$
\begin{array}{l}
\mathit{skip} :: \mathit{CL}\ s \\
\mathit{skip} = \mathit{Skip} \\
\\
\mathit{update} :: (s \to s) \to \mathit{CL}\ s \\
\mathit{update}\ f = \mathit{Update}\ f\ \mathit{skip} \\
\\
\mathit{cond} :: (s \to \mathit{Bool}) \to \mathit{CL}\ s \to \mathit{CL}\ s \to \mathit{CL}\ s \\
\mathit{cond}\ c\ p\ q = \mathit{If}\ c\ p\ q\ \mathit{skip} \\
\\
\mathit{while} :: (s \to \mathit{Bool}) \to \mathit{CL}\ s \to \mathit{CL}\ s \\
\mathit{while}\ c\ p = \mathit{While}\ c\ p\ \mathit{skip}
\end{array}
$$

And we can define sequential composition:

$$
\begin{array}{l}
(⨟) :: \mathit{CL}\ s \to \mathit{CL}\ s \to \mathit{CL}\ s \\
\mathit{Skip} \qquad\quad ⨟ k = k \\
\mathit{Update}\ f\ p\ ⨟ k = \mathit{Update}\ f\ (p ⨟ k) \\
\mathit{If}\ c\ p\ q\ r \quad ⨟ k = \mathit{If}\ c\ p\ q\ (r ⨟ k) \\
\mathit{While}\ c\ p\ q ⨟ k = \mathit{While}\ c\ p\ (q ⨟ k)
\end{array}
$$

Using the combinators and $(⨟)$, the examples can be written equivalently as

$$
\begin{array}{l}
\mathit{example}_1 = \\
\quad \mathit{update}\ (\lambda s \to s.y := 0) ⨟ \\
\quad \mathit{while}\ (\lambda s \to s.x > 0)
\end{array}
$$

$$(update\ (\lambda s \to s.y := (s.y + s.x)) \fatsemi$$
$$\quad update\ (\lambda s \to s.x := (s.x - 1)))$$

Note that $\langle CL\ s, skip, (\fatsemi) \rangle$ forms a monoid.

Semantics We can define a compositional semantics for CL programs over a state of type S as the carrier of a $(CL_{\mathrm{F}}\ S)$-algebra for the functor $CL_{\mathrm{F}}\ S$:

$$\begin{aligned}
&\textbf{\textit{data}}\ CL_{\mathrm{F}}\ s\ a \\
&\quad = Skip_{\mathrm{F}} \\
&\quad \mid Update_{\mathrm{F}}\ (s \to s)\ a \\
&\quad \mid If_{\mathrm{F}}\ (s \to Bool)\ a\ a\ a \\
&\quad \mid While_{\mathrm{F}}\ (s \to Bool)\ a\ a
\end{aligned}$$

Note that such a semantic definition, i.e. as a carrier of a syntactic algebra, is compositional by construction. A $(CL_{\mathrm{F}}\ S)$-algebra is a pair (A, alg) consisting of a type A and a function $alg :: CL_{\mathrm{F}}\ S\ A \to A$. In particular, $(CL\ S, c)$ is a $(CL_{\mathrm{F}}\ S)$-algebra, where

$$\begin{aligned}
&c :: CL_{\mathrm{F}}\ s\ (CL\ s) \to CL\ s \\
&c\ Skip_{\mathrm{F}} &&= Skip \\
&c\ (Update_{\mathrm{F}}\ f\ p) &&= Update\ f\ p \\
&c\ (If_{\mathrm{F}}\ c\ p\ q\ r) &&= If\ c\ p\ q\ r \\
&c\ (While_{\mathrm{F}}\ c\ p\ q) &&= While\ c\ p\ q
\end{aligned}$$

Indeed, $(CL\ S, c)$ is the initial $(CL_{\mathrm{F}}\ S)$-algebra; which is to say, for any other $(CL_{\mathrm{F}}\ S)$-algebra (A, alg) there is a unique morphism of algebras $CL\ S \to A$; informally, this unique morphism 'propagates alg through the abstract syntax tree'. Since this unique morphism is determined by the algebra alg, we introduce a notation $(\!|alg|\!)$ for it. Concretely, $(\!|alg|\!)$ is defined as follows:

$$\begin{aligned}
&(\!|-|\!) :: (CL_{\mathrm{F}}\ s\ a \to a) \to (CL\ s \to a) \\
&(\!|alg|\!)\ Skip &&= alg\ Skip_{\mathrm{F}} \\
&(\!|alg|\!)\ (Update\ f\ p) &&= alg\ (Update_{\mathrm{F}}\ f\ ((\!|alg|\!)\ p)) \\
&(\!|alg|\!)\ (If\ c\ p\ q\ r) &&= alg\ (If_{\mathrm{F}}\ c\ ((\!|alg|\!)\ p)\ ((\!|alg|\!)\ q)\ ((\!|alg|\!)\ r)) \\
&(\!|alg|\!)\ (While\ c\ p\ q) &&= alg\ (While_{\mathrm{F}}\ c\ ((\!|alg|\!)\ p)\ ((\!|alg|\!)\ q))
\end{aligned}$$

which propagates alg through the abstract syntax of a given program; $(\!|alg|\!)$ is known as the fold for algebra (A, alg) (Hutton, 1999).

Any compositional semantics of $CL\ s$ programs can be formalised as a fold with an appropriate algebra. For example, one straightforward semantics

is as an algebra on the carrier $s \to s$, i.e., interpreting programs as state-transformation functions.

$$
\begin{array}{l}
sem :: CL\ s \to (s \to s) \\
sem = (\!|alg|\!)\ \textbf{\textit{where}} \\
\quad alg :: CL_{\mathrm{F}}\ s\ (s \to s) \to (s \to s) \\
\quad alg\ Skip_{\mathrm{F}} \qquad\quad = id \\
\quad alg\ (Update_{\mathrm{F}}\ f\ p) \ = f \ggg p \\
\quad alg\ (If_{\mathrm{F}}\ c\ p\ q\ r) \ \ = conditional\ c\ p\ q \ggg r \\
\quad alg\ (While_{\mathrm{F}}\ c\ p\ q) = \textbf{\textit{let}}\ while = conditional\ c\ (p \ggg while)\ q \\
\qquad\qquad\qquad\qquad\quad \textbf{\textit{in}}\ \ while \\
\quad conditional :: (s \to Bool) \to (s \to s) \to (s \to s) \to (s \to s) \\
\quad conditional\ c\ t\ e = (\lambda s \to (c\ s, s)) \ggg \\
\qquad\qquad\qquad\qquad\quad (\lambda(b, s) \to \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ t\ s\ \textbf{\textit{else}}\ e\ s)
\end{array}
$$

Here, $(\ggg)$ is forward function composition: $f \ggg g = g \circ f$.

Note that *sem* is not only a fold over the abstract syntax, but also a monoid morphism from the *CL S* monoid to $\langle S \to S, id, (\ggg) \rangle$.

$$
\begin{array}{l}
sem\ skip \quad = id \\
sem\ (p \fatsemi q) = sem\ p \ggg sem\ q
\end{array}
$$

The above has set out our general approach to defining a language and its semantics, illustrating it on the simplest (useful) language possible. We now add probability.

4.2 Adding probability: CL → pGCL

The probabilistic version of CL is called PCL, following Haskell's (upper-case) convention for type constructors. It is effectively the *pGCL* of (McIver and Morgan, 2005), in turn derived from the seminal work of (Kozen, 1983). Its difference from CL is that state updates in PCL are probabilistic, i.e., an update does not assign a single new state, but rather a probability distribution over (new) states. For instance, we will be able to express that the variable `x` in our running example is decremented probabilistically by either `1` (probability $^2/_3$) or `2` (probability $^1/_3$), as here:

```
y := 0;
while (x > 0) {
  y := y + x;
  x := x - (1 2/3⊕ 2);
}
```
(8)

Similarly, we will make the conditionals in our program probabilistic. For instance, the following program's `while` loop chooses with probability $1/2$ on each iteration whether to apply the test $x > 0$ or $x > 1$ for termination:

```
y := 0;
while (x > 0 1/2⊕ x > 1) {
  y := y + x;
  x := x - 1;
}
```

We can also model the probabilistic choice between entire statements rather than merely expressions. For instance,

```
(y := y + 1) 5/6⊕ (x := x + 1)
```

is short-hand for

```
if (true 5/6⊕ false) {
  y := y + 1
} else {
  x := x + 1
}
```

The probabilistic nature of state updates is reflected as follows in the abstract syntax:

$$
\begin{aligned}
&\textbf{\textit{data}}\ \mathit{PCL}\ s \\
&\quad = \mathit{Skip} \\
&\quad \mid \mathit{Update}\ (s \to_{\mathrm{D}} s)\ (\mathit{PCL}\ s) \\
&\quad \mid \mathit{If}\ (s \to_{\mathrm{D}} \mathit{Bool})\ (\mathit{PCL}\ s)\ (\mathit{PCL}\ s)\ (\mathit{PCL}\ s) \\
&\quad \mid \mathit{While}\ (s \to_{\mathrm{D}} \mathit{Bool})\ (\mathit{PCL}\ s)\ (\mathit{PCL}\ s)
\end{aligned}
$$

where $\to$ in (6) has been replaced by the Kleisli arrow $\to_{\mathrm{D}}$ for the probability monad $\mathbb{D}$, so that $A \to_{\mathrm{D}} B$ is the type of probabilistic programs from A to B:

$$\textbf{\textit{type}}\ a \to_{\mathrm{D}} b = a \to \mathit{Dist}\ b$$

The abstract syntax of our first probabilistic example (8) becomes:

$$
\begin{aligned}
&\mathit{example}_2 :: \mathit{PCL}\ S \\
&\mathit{example}_2 = \\
&\quad \mathit{update}\ (\lambda s \to \mathit{return}\ (s.y := 0))\ ; \\
&\quad \mathit{while}\ (\lambda s \to \mathit{return}\ (s.x > 0)) \\
&\qquad (\mathit{update}\ (\lambda s \to \mathit{return}\ (s.y := (s.y + s.x)))\ ; \\
&\qquad\quad \mathit{update}\ (\lambda s \to (s.x := (s.x - 1))\ {}_{2/3}\!\oplus\ (s.x := (s.x - 2))))
\end{aligned}
$$

Observe that what was written earlier in the example as an assignment of a probabilistically chosen value is represented here as abstract syntax in the form of a probabilistic choice between assignments of pure values.

We obtain an interpreter for PCL by reusing the constructions of §4.1, simply adapting the target of the monoid morphism *sem* from the monoid $\langle s \to s, id, (\ggg) \rangle$ of endofunctions with function composition to the monoid $\langle s \to_{\mathrm{D}} s, return, (>=>) \rangle$ of Kleisli arrows with forward Kleisli composition.

$$
\begin{array}{l}
sem_{\mathrm{D}} :: Ord\ s \Rightarrow PCL\ s \to (s \to_{\mathrm{D}} s) \\
sem_{\mathrm{D}} = (\!|alg|\!)\ \textbf{\textit{where}} \\
\quad alg :: Ord\ s \Rightarrow PCL_{\mathrm{F}}\ s\ (s \to_{\mathrm{D}} s) \to (s \to_{\mathrm{D}} s) \\
\quad alg\ Skip_{\mathrm{F}} = return \\
\quad alg\ (Update_{\mathrm{F}}\ f\ p) = f >=> p \\
\quad alg\ (If_{\mathrm{F}}\ c\ p\ q\ r) = conditional\ c\ p\ q >=> r \\
\quad alg\ (While_{\mathrm{F}}\ c\ p\ q) = \textbf{\textit{let}}\ while = conditional\ c\ (p >=> while)\ q \\
\qquad\qquad\qquad\qquad\ \ \textbf{\textit{in}}\ while \\
\quad conditional :: (Ord\ s) \Rightarrow s \to_{\mathrm{D}} Bool \to (s \to_{\mathrm{D}} s) \to (s \to_{\mathrm{D}} s) \to (s \to_{\mathrm{D}} s) \\
\quad conditional\ c\ p\ q = (\lambda s \to fmap\ (\lambda b \to (b, s))\ (c\ s)) >=> \\
\qquad\qquad\qquad\qquad (\lambda(b, s) \to \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ p\ s\ \textbf{\textit{else}}\ q\ s)
\end{array}
$$

4.3 Adding observations: pGCL $\to$ **Kuifje**

Finally, we extend PCL with 'observations' to yield our *QIF* language "Kuifje". Observations are probabilistically chosen values that the computation outputs, or 'leaks', as a side-effect. For instance, in our running example we express that we can observe `x` at the end of each iteration by adding *observe x* there:

```
y := 0;
while (x > 0) {
  y := y + x;
  x := x - (1 2/3⊕ 2);
  observe x
}
```
(9)

To express that we will observe either x or y, with equal probability, we would write

```
y := 0;
while (x > 0) {
  y := y + x;
  x := x - (1 2/3⊕ 2);
  observe x 1/2⊕ observe y
}
```
(10)

And finally, we can express that we either observe x or observe y *but we don't know which* with

```
y := 0;
while (x > 0) {
  y := y + x;
  x := x - (1 2/3⊕ 2);
  observe (x 1/2⊕ y)
}
```
(11)

As an example of that important distinction, suppose there were a secret number x uniformly distributed with $0 \leq x < 3$, and that with probability $1/2$ either $x \bmod 2$ or $x \div 2$ were revealed in an observation, i.e. either its low- or high-order bit. If the observer knew whether mod or $\div$ had been used, then afterwards

knowing that mod *was used:*
with probability $1/3$ she would be able to conclude that x was either 0 or 2, assigning equal probability to each;
with probability $1/6$ she would be able to conclude that x was certainly 1.
knowing that $\div$ *was used:*
with probability $1/3$ she would be able to conclude that x was either 0 or 1, assigning equal probability to each; and
with probability $1/6$ she would be able to conclude that x was certainly 2.

On the other hand, if she did *not* know which of mod or $\div$ had been used, then afterwards

not knowing which of mod *or* $\div$ *was used:*
with probability $2/3$ she would be able to conclude that x was either 0,1 or 2, with probabilities $1/2$, $1/4$ and $1/4$ respectively; and
with probability $1/3$ she would be able to conclude that x was equally likely to be 1 or 2.

There's no denying that this is a subtle distinction — but it is a real one, and our programming language expresses it easily. [18]

[18] An explanation of the precise probabilities above is given in App. A.

To support observations –and to obtain our third and final language *Kuifje*– we add a new constructor *Observe* to the type *PCL*, based on a datatype *Bits* that allows the observation to be effectively of any type; in fact "*Bits*" is a temporary expedient that, below, will allow us to construct in Haskell the lists of heterogeneous element-type that accumulate the observations made. In our final presentation, these lists will disappear — taking "*Bits*" with them.

```
data Kuifje s
  = Skip
  | Update (s →D s) (Kuifje s)
  | If (s →D Bool) (Kuifje s) (Kuifje s) (Kuifje s)
  | While (s →D Bool) (Kuifje s) (Kuifje s)
  | Observe (s →D Bits) (Kuifje s)                    — added
```

The new form *Observe f p* uses *f* to probabilistically determine a sequence of bits from the current state, observe them and then proceed with program *p*. Here a sequence of bits is simply a list of booleans.

```
type Bit  = Bool
type Bits = [Bit]
```

The new basic combinator

```
observe :: ToBits a ⇒ (s →D a) → Kuifje s
observe f = Observe (fmap toBits ∘ f) skip
```

allows us to observe values of any type *a* whose conversion to *Bits* has been defined via the *ToBits* type class

```
class ToBits a where
  toBits :: a → Bits
```

For instance, *Int* values can be observed as bits through a binary encoding (here, *quot* is division rounding towards zero, so the encoding consists of a sign bit followed by a list of binary digits, least significant first; thus, -6 is encoded as $[True, False, True, True]$):

```
instance ToBits Int where
  toBits n = (n < 0) : unfoldr (λm → if m ≡ 0
                                     then Nothing
                                     else Just (odd m, quot m 2)) n
```

For syntax, the new constructor requires only a straightforward extension of the composition function (⨟):

$$
\begin{aligned}
&(\fatsemi) :: \mathit{Kuifje}\ s \to \mathit{Kuifje}\ s \to \mathit{Kuifje}\ s\\
&\mathit{Skip} \fatsemi k = k\\
&\mathit{Update}\ f\ p \fatsemi k = \mathit{Update}\ f\ (p \fatsemi k)\\
&\mathit{While}\ c\ p\ q \fatsemi k = \mathit{While}\ c\ p\ (q \fatsemi k)\\
&\mathit{If}\ c\ p\ q\ r \fatsemi k = \mathit{If}\ c\ p\ q\ (r \fatsemi k)\\
&\mathit{Observe}\ f\ p \fatsemi k = \mathit{Observe}\ f\ (p \fatsemi k) \quad \text{— added}
\end{aligned}
$$

The abstract syntax tree of the first example (9) with observations is:

$$
\begin{aligned}
&\mathit{example}_{3a} :: \mathit{Kuifje}\ S\\
&\mathit{example}_{3a} =\\
&\quad \mathit{update}\ (\lambda s \to \mathit{return}\ (s.y := 0)) \fatsemi\\
&\quad \mathit{while}\ (\lambda s \to \mathit{return}\ (s.x > 0))\\
&\qquad (\mathit{update}\ \ (\lambda s \to \mathit{return}\ (s.y := (s.y + s.x))) \fatsemi\\
&\qquad\ \mathit{update}\ \ (\lambda s \to (s.x := (s.x - 1)) \ {}_{2/3}\!\oplus\ (s.x := (s.x - 2))) \fatsemi\\
&\qquad\ \mathit{observe}\ (\lambda s \to \mathit{return}\ (s.x)))
\end{aligned}
$$

For the second (10), it is

$$
\begin{aligned}
&\mathit{example}_{3b} :: \mathit{Kuifje}\ S\\
&\mathit{example}_{3b} =\\
&\quad \mathit{update}\ (\lambda s \to \mathit{return}\ (s.y := 0)) \fatsemi\\
&\quad \mathit{while}\ (\lambda s \to \mathit{return}\ (s.x > 0))\\
&\qquad (\mathit{update}\ (\lambda s \to \mathit{return}\ (s.y := (s.y + s.x))) \fatsemi\\
&\qquad\ \mathit{update}\ (\lambda s \to (s.x := (s.x - 1)) \ {}_{2/3}\!\oplus\ (s.x := (s.x - 2))) \fatsemi\\
&\qquad\ \mathit{cond}\ (\lambda s \to \mathit{True} \ {}_{1/2}\!\oplus\ \mathit{False})\\
&\qquad\qquad (\mathit{observe}\ (\lambda s \to \mathit{return}\ (s.x)))\\
&\qquad\qquad (\mathit{observe}\ (\lambda s \to \mathit{return}\ (s.y))))
\end{aligned}
$$

And for the third (11) it is

$$
\begin{aligned}
&\mathit{example}_{3c} :: \mathit{Kuifje}\ S\\
&\mathit{example}_{3c} =\\
&\quad \mathit{update}\ (\lambda s \to \mathit{return}\ (s.y := 0)) \fatsemi\\
&\quad \mathit{while}\ (\lambda s \to \mathit{return}\ (s.x > 0))\\
&\qquad (\mathit{update}\ \ (\lambda s \to \mathit{return}\ (s.y := (s.y + s.x))) \fatsemi\\
&\qquad\ \mathit{update}\ \ (\lambda s \to (s.x := (s.x - 1)) \ {}_{2/3}\!\oplus\ (s.x := (s.x - 2))) \fatsemi\\
&\qquad\ \mathit{observe}\ (\lambda s \to (s.x) \ {}_{1/2}\!\oplus\ (s.y)))
\end{aligned}
$$

For semantics, however, the domain of interpretation requires a more significant change. Indeed, we augment the distribution monad *Dist* with the capabilities of a *Bits*-writer monad to accommodate the list of accumulated

observations: [19] Thus we interpret programs as Kleisli arrows $s \to_{\mathrm{DB}} s$ of this augmented monad.

$$\textbf{\textit{type}}\ a \to_{\mathrm{DB}} b = a \to \mathit{Dist}\ (\mathit{Bits}, b)$$

Again because of our general approach, the structure of the interpreter remains largely the same, because most cases are parametric in the underlying monad: first we had $s \to s$; then we had $s \to_{\mathrm{D}} s$ and now we have $s \to_{\mathrm{DB}} s$. The two cases that require attention are *Update* and *Observe*.

$$\begin{aligned}
&\mathit{sem}_{\mathrm{DB}} :: \mathit{Kuifje}\ s \to (s \to_{\mathrm{DB}} s)\\
&\mathit{sem}_{\mathrm{DB}} = (\!|\mathit{alg}_P|\!)\ \textbf{\textit{where}}\\
&\quad \mathit{alg}_P :: (\mathit{Ord}\ s) \Rightarrow \mathit{Kuifje}_{\mathrm{F}}\ s\ (s \to_{\mathrm{DB}} s) \to (s \to_{\mathrm{DB}} s)\\
&\quad \mathit{alg}_P\ \mathit{Skip}_{\mathrm{F}} = \lambda x \to \mathit{return}\ ([\,], x)\\
&\quad \mathit{alg}_P\ (\mathit{Update}_{\mathrm{F}}\ f\ p) = \mathit{uplift}\ f >\!\!=\!\!> p\\
&\quad \mathit{alg}_P\ (\mathit{If}_{\mathrm{F}}\ c\ p\ q\ r) = \mathit{conditional}\ c\ p\ q >\!\!=\!\!> r\\
&\quad \mathit{alg}_P\ (\mathit{While}_{\mathrm{F}}\ c\ p\ q) = \textbf{\textit{let}}\ \mathit{while} = \mathit{conditional}\ c\ (p >\!\!=\!\!> \mathit{while})\ q\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{\textit{in}}\ \ \mathit{while}\\
&\quad \mathit{alg}_P\ (\mathit{Observe}_{\mathrm{F}}\ f\ p) = \mathit{obsem}\ f >\!\!=\!\!> p
\end{aligned}$$

In the case of *Update* we lift the update function f from the distribution monad to the augmented distribution monad, by adding an empty sequence of observations.

$$\begin{aligned}
&\mathit{uplift} :: (a \to_{\mathrm{D}} b) \to (a \to_{\mathrm{DB}} b)\\
&\mathit{uplift}\ f = \mathit{fmap}\ (\lambda b \to ([\,], b)) \circ f
\end{aligned}$$

In the case of *Observe* we extract the observation and return it alongside the current state.

$$\begin{aligned}
&\mathit{obsem} :: (s \to_{\mathrm{D}} \mathit{Bits}) \to (s \to_{\mathrm{DB}} s)\\
&\mathit{obsem}\ f = \lambda s \to \mathit{fmap}\ (\lambda w \to (w, s))\ (f\ s)
\end{aligned}$$

Finally, while we preserve the definition of *If* and *While* in terms of the more primitive *conditional*, we do modify the conditional to leak its argument.

$$\begin{aligned}
&\mathit{conditional} :: (\mathit{Ord}\ s) \Rightarrow (s \to_{\mathrm{D}} \mathit{Bool}) \to (s \to_{\mathrm{DB}} s) \to (s \to_{\mathrm{DB}} s) \to (s \to_{\mathrm{DB}} s)\\
&\mathit{conditional}\ c\ p\ q = \mathit{obsem}\ (\mathit{fmap}\ \mathit{toBits} \circ c) >\!\!=\!\!>\\
&\qquad\qquad (\lambda([b], s) \to \mathit{return}\ ([b], (b, s))) >\!\!=\!\!>\\
&\qquad\qquad (\lambda(b, s) \to \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ p\ s\ \textbf{\textit{else}}\ q\ s)
\end{aligned}$$

[19] This list of observations becomes ever longer as the program executes; but in §5.3 the whole list, and *Bits*, will be 'quotiented away'.

The consequence is that the programs *skip* and *cond* $(\lambda s \to \mathit{True}\ {}_{1/2}\!\oplus \mathit{False})$ *skip skip* now have different semantics:

$> \mathit{sem}_{\mathrm{DB}}\ \mathit{skip}\ ()$
$1 \div 1 \quad ([\,], ())$
$> \mathit{sem}_{\mathrm{DB}}\ (\mathit{cond}\ (\lambda s \to \mathit{True}\ {}_{1/2}\!\oplus \mathit{False})\ \mathit{skip}\ \mathit{skip})\ ()$
$1 \div 2 \quad ([\mathit{False}], ())$
$1 \div 2 \quad ([\mathit{True}], ())$

There are two good reasons for having leaking conditionals. It is a common (though not universal) assumption in *QIF*, a pragmatic principle that one should not 'branch on high'. Much more important, however, is that it enables a compositional hyper-semantics, as we explain in Section 5.3.

5 Hyper-distributions: from leaked values to leaked information

In §4 we defined the syntax and semantics of our language in three stages, of which Kuifje –with probabilistic choice and observations– was the final outcome. We now undertake a major abstraction, removing the sequence of observed values but leaving behind the information-flow effect they induce.

5.1 Abstracting from observed values

If an observer is interested only in information flow about the value of a program's final state, then –we will argue– the values of the observations themselves are irrelevant. An example of this is spies who speak different languages. If they are to report the value of a secret Booolean, it makes no difference whether they say "True/False" or "Waar/Onwaar" or "Vrai/Faux" as long as their controller knows the correspondence between the observed utterance and the hidden value that caused it. Thus in our *Bit*-sequence semantics it should not matter whether the observations are say

```
    ["true","true","false"]
or  ["waar","waar","onwaar"]
or  ["vrai","vrai","faux"] .
```

They all three result in exactly the same information flow.

The abstraction we are about to implement, motivated by that, allows a drastic simplification of the $\mathit{sem}_{\mathrm{DB}}$ semantics: we throw away the sequences of *Bits*, but we retain the distinctions they made. Further, if we propagate

the abstraction to the leaves of the abstract syntax tree, we never have to construct the sequences in the first place — and then we can throw away the *Bits* type itself. Here is how it is done.

In the above interpretation, the type $S \to Dist\ (Bits, S)$ denotes a map from initial states of type S to probability distributions of sequences of observed *Bits* together with a final state of type S.

The resulting distributions dp of type $Dist\ (Bits, S)$ are isomorphic to pairs (d, f) of type $(Dist\ Bits, Bits \to Dist\ S)$, where the domain of f is restricted to bit sequences that occur with non-zero probability in d. This isomorphism is witnessed by the functions *toPair* and *fromPair*.

```
toPair :: (Ord s) ⇒ Dist (Bits, s) → (Dist Bits, Bits → Dist s)
toPair dp = (d, f)
  where
    d    = fmap fst dp
    f ws = let dpws = D [(s, p) | ((ws', s), p) ← runD dp, ws' ≡ ws]
           in  D [(s, p / weight dpws) | (s, p) ← runD dpws]
fromPair :: (Dist Bits, Bits → Dist s) → Dist (Bits, s)
fromPair (d, f) = join (fmap (λws → fmap (λs → (ws, s)) (f ws)) d)
```

The function *toPair* allows us to determine the likelihood of each possible trace of observations, together with the conditional distribution of possible states that the trace induces.

For instance, for the program over two Booleans

```
example4 :: Kuifje (Bool, Bool)
example4 = observe (λ(b1, b2) → b1 1/2⊕ b2)
```

and the uniform distribution of Boolean pairs as input distribution

```
boolPairs :: Dist (Bool, Bool)
boolPairs = uniform [(b1, b2) | b1 ← bools, b2 ← bools]
  where bools = [True, False]
```

we can see the distribution of observations

```
> fst (toPair (boolPairs >>= sem_DB example4))
1 ÷ 2   [False]
1 ÷ 2   [True]
```

that is the sequence $[False]$ with probability $1/2$ and the sequence $[True]$ the same; and their respective conditional distributions of final states

$$
\begin{array}{ll}
> \mathit{snd}\ (\mathit{toPair}\ (\mathit{boolPairs} \gg\!= \mathit{sem}_{\mathrm{DB}}\ \mathit{example}_4))\ [\mathit{False}] & \\
1 \div 2 & (\mathit{False}, \mathit{False}) \\
1 \div 4 & (\mathit{False}, \mathit{True}) \\
1 \div 4 & (\mathit{True}, \mathit{False})
\end{array}
\tag{12}
$$

that is that if *False* is observed, the posterior distribution of states is as above, and similarly

$$
\begin{array}{ll}
> \mathit{snd}\ (\mathit{toPair}\ (\mathit{boolPairs} \gg\!= \mathit{sem}_{\mathrm{DB}}\ \mathit{example}_4))\ [\mathit{True}] & \\
1 \div 4 & (\mathit{False}, \mathit{True}) \\
1 \div 4 & (\mathit{True}, \mathit{False}) \\
1 \div 2 & (\mathit{True}, \mathit{True})
\end{array}
\tag{13}
$$

If *True* is observed, the posterior distribution of states is as here instead.

If we do not care about the particular trace of observations –our postulate– but are only interested in the variation of distributions of final states, we can eliminate the sequences of observations altogether while retaining however the conditional distributions they determine. This yields so-called hyper-distributions, i.e., distributions of distributions, of type $\mathit{Dist}\ (\mathit{Dist}\ s)$. Each of the 'inner' distributions of the hyper has as its own 'outer' probability the probability that was assigned to the no-longer-present *Bits*-sequence that gave rise to it. We do that with this function:

$$
\begin{array}{l}
\mathit{multiply} :: (\mathit{Dist}\ \mathit{Bits}, \mathit{Bits} \to \mathit{Dist}\ s) \to \mathit{Dist}\ (\mathit{Dist}\ s) \\
\mathit{multiply}\ (d, f) = \mathit{fmap}\ f\ d
\end{array}
$$

Thus, putting everything together, we can compute the hyper-distribution of final states as follows from a given distribution of initial states.

$$
\begin{array}{l}
\mathit{hyper} :: (\mathit{Ord}\ s) \Rightarrow \mathit{Kuifje}\ s \to (\mathit{Dist}\ s \to \mathit{Dist}\ (\mathit{Dist}\ s)) \\
\mathit{hyper}\ p\ d_0 = \mathit{multiply}\ (\mathit{toPair}\ (d_0 \gg\!= \mathit{sem}_{\mathrm{DB}}\ p))
\end{array}
\tag{14}
$$

That yields the following result on our example program, where the outers are on the left and the inners are on the right:

$$
\begin{array}{lll}
> \mathit{hyper}\ \mathit{example}_4\ \mathit{boolPairs} & & \\
1 \div 2 & 1 \div 4 & (\mathit{False}, \mathit{True}) \\
 & 1 \div 4 & (\mathit{True}, \mathit{False}) \\
 & 1 \div 2 & (\mathit{True}, \mathit{True}) \\
1 \div 2 & 1 \div 2 & (\mathit{False}, \mathit{False}) \\
 & 1 \div 4 & (\mathit{False}, \mathit{True}) \\
 & 1 \div 4 & (\mathit{True}, \mathit{False})
\end{array}
$$

5.2 Hyper-distributions in theory: why?

Although the definition of *hyper* calculates the sequences of observations, which we can then remove, we will see in §5.3 below that these two steps can be fused into one so that only the abstractions are used, and the *Bits*-sequences are never constructed.

First however we look at the theoretical reasons for doing this.

Our model of a *QIF*-aware program is as an initial-state to final-state mechanism that chooses its final state probabilistically, depending on the initial state, and might release information about the state at the same time. The fundamental insight *in theory* is that the actual value of the leak is unimportant: the leak's only role is in allowing an adversary to make deductions about what the state must have been at that point in order for that leak-value to have been observed. That is, we have decided as a design principle that the two programs

$$\mathit{observe}\ b \quad \text{and} \quad \mathit{observe}\ (\mathit{not}\ b)$$

are the same: each one, if executed, will allow the adversary to deduce the (current) value of b because, knowing the source-code, she also knows (in this example) whether she must negate the leak or not; and –beyond leaking b– they are operationally the same, since neither changes the state. This being so, we use hyper-distributions in the (denotational) theory to abstract from those output values; and the result is that two programs are behaviourally equal just when their denotations are equal — i.e. this way we achieve full abstraction.

Further, this 'tidiness' in the semantic structures allows us in a more extensive presentation to discuss the domain structure of the semantic space: its refinement order; whether it is complete; how fixed-points are found etc. This is extensively discussed in other work (McIver et al., 2014b).

5.3 Hyper-distributions in practice: why?

If we accept the arguments in §5.2 just above, that we are interested only in the hyper-distribution of final states for a given distribution of initial states then, as we have suggested, *hyper* is not the most efficient way to compute it: *hyper* first computes the full distribution $\mathit{Dist}\ (\mathit{Bits}, s)$ before condensing it to the usually much more compact hyper-distribution $\mathit{Dist}\ (\mathit{Dist}\ s)$. In this section we explain how that can be implemented. We work directly with the more compact hyper-distributions throughout: programs are interpreted as hyper-arrows, and the writer-monad is no longer used; and this increased efficiency is the 'why' of hyper-distributions in practice. Thus we define

$$\textbf{type}\ a \to_{\mathrm{DD}} b = Dist\ a \to Dist\ (Dist\ b)$$

which happens to form a monoid as well: $\langle s \to_{\mathrm{DD}} s, return, >=> \rangle$. Indeed, we can derive the *hyper* semantics directly as a fold sem_{DD},

$$\begin{aligned} &sem_{\mathrm{DD}} :: (Ord\ s) \Rightarrow Kuifje\ s \to (s \to_{\mathrm{DD}} s) \\ &sem_{\mathrm{DD}} = (\!|alg_H|\!) \end{aligned} \tag{15}$$

by solving the "fold fusion" equation for the algebra alg_H

$$post \circ alg_P \quad = \quad alg_H \circ fmap\ hyper \tag{16}$$

where

$$\begin{aligned} &post :: Ord\ s \Rightarrow (s \to_{\mathrm{DB}} s) \to (s \to_{\mathrm{DD}} s) \\ &post\ t = \lambda d \to multiply\ (toPair\ (d \gg\!= t)) \end{aligned}$$

Using the following three properties of *post*

$$post\ return = return \tag{17}$$

$$post\ (f >=> g) = post\ f >=> post\ g \tag{18}$$

$$post\ (\lambda s \to (f\ s)\ {}_w\!\oplus\ (g\ s)) = \lambda d \to (post\ f\ d)\ {}_w\!\oplus\ (post\ g\ d) \tag{19}$$

it is possible to verify[20] that the definition of alg_H below satisfies (16).

$$\begin{aligned} &alg_H :: (Ord\ s) \Rightarrow Kuifje_{\mathrm{F}}\ s\ (s \to_{\mathrm{DD}} s) \to (s \to_{\mathrm{DD}} s) \\ &alg_H\ Skip_{\mathrm{F}} &&= return \\ &alg_H\ (Update_{\mathrm{F}}\ f\ p) &&= huplift\ f >=> p \\ &alg_H\ (If_{\mathrm{F}}\ c\ p\ q\ r) &&= conditional\ c\ p\ q >=> r \\ &alg_H\ (While_{\mathrm{F}}\ c\ p\ q) &&= \textbf{let}\ while = conditional\ c\ (p >=> while)\ q \\ &&&\quad \textbf{in}\ \ while \\ &alg_H\ (Observe_{\mathrm{F}}\ f\ p) &&= hobsem\ f >=> p \end{aligned}$$

$$\begin{aligned} &conditional :: Ord\ s \Rightarrow (s \to_{\mathrm{D}} Bool) \to (s \to_{\mathrm{DD}} s) \to (s \to_{\mathrm{DD}} s) \to (s \to_{\mathrm{DD}} s) \\ &conditional\ c\ t\ e = \lambda d \to \\ &\quad \textbf{let}\ d' = d \gg\!= \lambda s \to c\ s \gg\!= \lambda b \to return\ (b, s) \\ &\qquad w_1 = sum\ [p \mid ((b, s), p) \leftarrow runD\ d', b] \\ &\qquad w_2 = 1 - w_1 \\ &\qquad d_1 = D\ [(s, p\ /\ w_1) \mid ((b, s), p) \leftarrow runD\ d', b] \\ &\qquad d_2 = D\ [(s, p\ /\ w_2) \mid ((b, s), p) \leftarrow runD\ d', not\ b] \\ &\qquad h_1 = t\ d_1 \\ &\qquad h_2 = e\ d_2 \end{aligned}$$

[20] Note that Equation 19 only holds when there are no two inputs x and y such that $f\ x$ and $g\ y$ yield the same observations. This is the case for two branches of a conditional, which, because they leak their condition, always yield disjoint observations.

```
  in if      null (runD d2) then h1
     else if null (runD d1) then h2
                            else join (h1 w1⊕ h2)
```

```
huplift :: Ord s ⇒ (s →D s) → (s →DD s)
huplift f = return ∘ (>>= f)
```

Because sem_{DD} immediately abstracts over the observations and never collects them in a (homogeneously-typed) list, we do not first have to convert them to *Bits*. This means that we can generalize our syntax of programs to observations of arbitrary type o:

```
data Kuifje s
  = Skip
  | Update (s →D s) (Kuifje s)
  | If (s →D Bool) (Kuifje s) (Kuifje s)
  | While (s →D Bool) (Kuifje s) (Kuifje s)
  | ∀o . (Ord o, ToBits o) ⇒ Observe (s →D o) (Kuifje s)
```

and interpret them without requiring a *ToBits* instance.

```
hobsem :: (Ord s, Ord o) ⇒ (s →D o) → (s →DD s)
hobsem f = multiply ∘ toPair ∘ (>>= obsem f)
  where
    obsem :: Ord o ⇒ (a →D o) → (a →D (o, a))
    obsem f = λx → fmap (λw → (w, x)) (f x)
    toPair :: (Ord s, Ord o) ⇒ Dist (o, s) → (Dist o, o → Dist s)
    toPair dp = (d, f)
      where
        d    = fmap fst dp
        f ws = let dpws = D [(s, p) | ((ws', s), p) ← runD dp, ws' ≡ ws]
               in  D [(s, p / weight dpws) | (s, p) ← runD dpws]
    multiply :: (Dist o, o → Dist s) → Dist (Dist s)
    multiply (d, f) = fmap f d
```

In summary, with sem_{DD} from (15) we obtain the same results as with sem_{DB} (12,13) — but without the need for post-processing:

```
> semDD example4 boolPairs
1 ÷ 2   1 ÷ 4   (False, True)
        1 ÷ 4   (True, False)
        1 ÷ 2   (True, True)
```

$1 \div 2$ $1 \div 2$ $(False, False)$
$1 \div 4$ $(False, True)$
$1 \div 4$ $(True, False)$

6 Case Studies

6.1 The Monty-Hall problem

The (in)famous Monty-Hall problem (Rosenhouse, 2009) concerns a quiz-show where a car is hidden behind one of three curtains. The other two curtains conceal goats. The show's host is Monty Hall, and a contestant (Monty's adversary) is trying to guess which curtain conceals the car.

Initially, the contestant believes the car is equally likely to be behind each curtain. She chooses one of the three, reasoning (correctly) that her chance of having chosen the car is $^1/_3$; but the host does not open the curtain. Instead Monty opens one of the two other curtains, making sure that a goat is there. (Thus if the contestant has chosen the car –though she does not know that– he will open either of the other two curtains; but if she has not chosen the car he opens the (unique) other curtain that hides a goat. Either way, from her point of view, he has opened a curtain where there is a goat.)

Monty Hall then says "Originally you had a one-in-three chance of getting the car. But now there are only two possible positions for the car: the curtain you chose, and the other closed curtain. Would you like to change your mind?" The notorious puzzle is then "Should she change?"

A qualitative (i.e. non-quantitative) approach to this, relying on intuition, suggests that

(i) Since Monty Hall could have opened a goat-curtain no matter where the car is, his doing so conveys nothing; and
(ii) Since the contestant still does not know where the car is, nothing has been leaked.

But a quantitative approach enables more sophisticated reasoning.[21] Even though the contestant does not know for sure where the car is, after Monty's action, is it really true that she knows *no* more than before? Or has she perhaps learned something, but not everything? Has *some* information flowed?

[21] ... but also sometimes unsophisticated too: "Since there are now only two doors, the chance of the car's being behind the door already chosen has risen to $^1/_2$."

There are many compelling informal arguments for that.[22] But here we give one based on the information-flow semantics of Kuifje.

We declare the three-element type *Door*

$$\textbf{\textit{data}}\ \mathit{Door} = \mathit{DoorA} \mid \mathit{DoorB} \mid \mathit{DoorC}\ \textbf{\textit{deriving}}\ (\mathit{Eq}, \mathit{Show}, \mathit{Ord})$$

and describe Monty's action with a single Kuifje statement: choose a door that is neither the door already chosen by the contestant nor the one with the car. This is the *hall* program just below, with argument *ch* for the contestant's choice; its initial state *d* is where the car is:

$$\begin{array}{l} \mathit{hall} :: \mathit{Door} \to \mathit{Kuifje}\ \mathit{Door} \\ \mathit{hall}\ \mathit{ch} = \mathit{observe}\ (\lambda d \to \mathit{uniform}\ ([\mathit{DoorA}, \mathit{DoorB}, \mathit{DoorC}] \setminus\setminus [d, \mathit{ch}])) \end{array}$$

The list $[\mathit{DoorA}, \mathit{DoorB}, \mathit{DoorC}] \setminus\setminus [d, \mathit{ch}]$ is those doors that were not chosen by the contestant *and* don't conceal the car: there can be one or two of them, depending on whether the contestant (so far, unknowingly) chose the car.[23]

If the contestant initially chooses *DoorA*, then we obtain the following hyper-distribution of the car's door after observing the goat revealed by Monty:

$$\begin{array}{l} \mathit{doors} = \mathit{uniform}\ [\mathit{DoorA}, \mathit{DoorB}, \mathit{DoorC}] \\ \mathit{monty} = \mathit{sem}_{\mathrm{DD}}\ (\mathit{hall}\ \mathit{DoorA})\ \mathit{doors} \end{array}$$

$$\begin{array}{lll} > \mathit{monty} & & \\ 1 \div 2 & 1 \div 3 & \mathit{DoorA} \\ & 2 \div 3 & \mathit{DoorB} \\ 1 \div 2 & 1 \div 3 & \mathit{DoorA} \\ & 2 \div 3 & \mathit{DoorC} \end{array} \tag{20}$$

It expresses that the contestant will know (or *should* realise) that the car is with probability $^2/_3$ behind the still-closed curtain. The two $^1/_2$ 'outer' probabilities reflects (given she chose *DoorA*) that the remaining closed door is equally likely to be *DoorB* or *DoorC*.[24] Since the program treats all doors in the same way, the same argument holds even if another initial door was chosen: in every case, it is better to change.

Since we have captured the result hyper in the variable *monty*, we can

[22] For those who still doubt: Suppose there were 100 doors, 99 goats and one car. The contestant chooses one door, and Monty opens 98 others with goats behind every one...

[23] The operator ($\setminus\setminus$) removes all elements of the right list from the left list.

[24] That can happen in two ways. If she chose the car (unknowingly) at *DoorA*, then Monty is equallty likely to open *DoorB* or *DoorC*; if she did not choose the car, it is equally likely to be behind *DoorB* or *DoorC*, and thus equally likely that Monty will open *DoorC* or *DoorB* respectively.

carry this analysis a bit further. (Note that we do not have to re-run the program: the output hyper *monty* contains all we need for the analysis.) Recall that the *Bayes Vulnerability bv* of a distribution (for which we wrote $V()$ in Fig. 4) is precisely the maximum probability of any element:

```
bv :: Ord a ⇒ Dist a → Prob
bv = maximum ∘ map snd ∘ runD ∘ reduction
```

and that this represents a rational adversary's strategy in guessing a secret whose distribution is known: guess the secret of (possibly equal) greatest probability. To use *bv* in the situation above, where there are *two* possible distributions the contestant might face, we simply average her best-chance over each of the two distributions' likelihood of occurring. For the first, with probability $1/2$, she will be able to guess correctly with probability $2/3$, giving $1/2 \times 2/3 = 1/3$ for the overall probability that Monty will reveal *DoorC* and the car will be behind *DoorB*. We get $1/3$ for the other alternative (it is effectively the same, with the doors changed), and so her overall probability of finding the car is $1/3 + 1/3 = 2/3$.

That process can be automated by defining

```
condEntropy :: (Dist a → Rational) → Dist (Dist a) → Rational
condEntropy e h = average (fmap e h) where
  average :: Dist Rational → Rational   — Average a distr. of Rational's.
  average d = sum [r × p | (r, p) ← runD d]
```

which for any entropy, that is *bv* in the case just below, gives its average when applied to all the inners of a hyper — yielding the conditional entropy. Thus we get

```
> condEntropy bv monty
2 ÷ 3
```

for the (smart) contestant's chance of getting her new car.

6.2 A defence against side-channels

Our second example here concerns a side-channel attack on the well known fast-exponentiation algorithm used for public-key encryptions.

The algorithm is given in pseudo-code at Fig. 7. As usual we assume that the program code is public, so that any side channels caused e.g. by 'branching on high' might allow an adversary to make deductions about the inputs: in this case, a careful analysis might be used to distinguish between

```
VAR B ← Base.                                        Global variables.
    E ← Exponent.
    p ← To be set to B^E.

BEGIN VAR b,e:= B,E                                  Local variables.
  p:= 1
  WHILE e≠0 DO
    VAR r:= e MOD 2
    IF r≠0 THEN p:= p*b FI                           ← Side channel.
    b,e:= b^2,e÷2
  END
END
{ p = B^E }
```

Here we are assuming that the 'branch on high' is the undesired side-channel: by detecting whether or not the branch is taken, the adversary can learn the bits of exponent `E` –which is the secret key– one by one. When the loop ends, she will have learned them all.

Figure 7 Insecure implementation of public/private key encryption.

```
Global variables.
VAR B ← Base.                                        Global variables.
    D ← Set of possible divisors.

    p ← To be set to B^E.
    E:= uniform(0..N-1)  Choose exponent uniformly at random.

BEGIN VAR b,e:= B,E                                  Local variables.
  p:= 1
  WHILE e≠0 DO
    VAR d:= uniform(D) ← Choose divisor uniformly from set D.
    VAR r:= e MOD d
    IF r≠0 THEN p:= p*b^r FI                         ← Side channel.
    b,e:= b^d,e÷d
  END
END
{ p = B^E }  What does the adversary know about E at this point?
```

Here the side channel is much less effective: although the adversary learns whether `r`=0, she knows nothing about `d` except that it was chosen uniformly from `D`, and thus learns little about `e`, and hence `E` at that point. A typical choice for `D` would be $[2, 3, 5]$. When the loop ends, she will have learned something about `E`, but not all of it. (In order to be able to analyse the program's treatment of `E` as a secret, we have initialised it uniformly from `N` possible values.)

Figure 8 Obfuscated implementation of public/private key encryption.

the two branches of the `IF`, [25] as indicated in Fig. 7, which is equivalent to determining whether the current value of `e` is divisible by 2. That occurs each time the loop iterates, and an adversary who has access to this (e.g. by analysing timing) would therefore be able to figure out exactly the initial value of `E` one bit at a time — and `E` is in fact the encryption key.

A defence against this side channel attack was proposed by (Walter, 2002) and is implemented at Fig. 8. His idea is that rather than attempting to close the side channel, instead one can reduce its effectiveness. The problem with the implementation at Fig. 7 is that 2 is always used as a divisor, which is why the i^{th} branching at the `IF`-statement is correlated exactly with the i'th bit of the original secret `E`. In Fig. 8, that correlation is attenuated by adding an extra variable `d`, used as divisor in place of 2 — and it is selected independently at random on each iteration. That obfuscates the relationship between `e` and the branching at the `IF`-statement, because the adversary does not know which value of `d` is being used. The information transmitted by the channel is therefore no longer exactly correlated with the i'th bit of the secret.

To compare the two programs we used the semantics of Kuifje to compute the final hyper-distribution resulting from Fig. 8 for an example range of `E`, determined by `N`. We assume that all the variables are hidden, as we are only interested in the information flowing through the side channel and what it tells us about `e`'s divisibility by `d`. [26]

Below is a translation of our obfuscated algorithm Fig. 8 into Kuifje. All variables are global, so the state space is:

$$\textbf{\textit{data}}\ SE = SE\ \{\_base, \_exp, \_e, \_d, \_p :: Integer\}$$

And we initialise the state as follows:

$$\begin{aligned}&initSE :: Integer \to Integer \to SE\\ &initSE\ base\ exp = SE\ \{\_base = base, \_exp = exp, \_e = 0, \_d = 0, \_p = 0\}\end{aligned}$$

And here is the body of the algorithm, based on Fig. 8, taking a list ds of divisors:

$$\begin{aligned}&exponentiation :: [Integer] \to Kuifje\ SE\\ &exponentiation\ ds =\\ &\quad update\ (\lambda s \to return\ (s.e := (s.exp)))\ ;\\ &\quad update\ (\lambda s \to return\ (s.p := 1))\ ;\\ &\quad while\ (\lambda s \to return\ (s.e \neq 0))\end{aligned}$$

[25] For example it could be a timing leak.

[26] We are not, in this analysis, considering a brute force attack that could invert the exponentiation.

$$
\begin{array}{l}
(\mathit{update}\ (\lambda s \to \mathit{uniform}\ [s.d := d' \mid d' \leftarrow ds])\fatsemi \\
\ \mathit{cond}\ (\lambda s \to \mathit{return}\ (s.e\ `\mathit{mod}`\ s.d \neq 0)) \\
\quad (\mathit{update}\ (\lambda s \to \mathit{return}\ (s.p := ((s.p) \times ((s.\mathit{base}) \uparrow (s.e\ `\mathit{mod}`\ s.d)))))\fatsemi \\
\quad\ \mathit{update}\ (\lambda s \to \mathit{return}\ (s.e := (s.e - (s.e\ `\mathit{mod}`\ s.d))))) \quad \text{— Then} \\
\quad \mathit{skip}\fatsemi \quad \text{— Else} \\
\ \mathit{update}\ (\lambda s \to \mathit{return}\ (s.\mathit{base} := ((s.\mathit{base}) \uparrow (s.d))))\fatsemi \\
\ \mathit{update}\ (\lambda s \to \mathit{return}\ (s.e := (s.e\ `\mathit{div}`\ s.d))) \\
) \qquad (21)
\end{array}
$$

Finally, we project the program's output onto a hyper retaining only the variable E (that is $\_exp$), using the following function:

$$
\begin{array}{l}
\mathit{project} :: \mathit{Dist}\ (\mathit{Dist}\ \mathit{SE}) \to \mathit{Dist}\ (\mathit{Dist}\ \mathit{Integer}) \\
\mathit{project} = \mathit{fmap}\ (\mathit{fmap}\ (\lambda s \to s.\mathit{exp}))
\end{array}
$$

In the two runs below we choose E uniformly from $[0\,..\,15]$, that is a 4-bit exponent (secret key). The first case *hyper2* is effectively the conventional algorithm of Fig. 7, because we restrict the divisor d to being 2 every time:

$$
\begin{array}{l}
\mathit{hyper2} = \mathit{project}\ (\mathit{sem}_{\mathrm{DD}}\ (\mathit{exponentiation}\ [2]) \\
\qquad\qquad\qquad\qquad (\mathit{uniform}\ [\mathit{initSE}\ 6\ \mathit{exp} \mid \mathit{exp} \leftarrow [0\,..\,15]]))
\end{array}
$$

The value of *hyper2*, that is what is known about E after calculating the power p, is shown in Fig. 9. The first column (all $1 \div 16$, that is $1/16$) shows that there are sixteen possible outcomes distributed just as the hidden input E was, that is uniformly. The second- and third columns show that in each of those outcomes, the adversary will know for certain $(1 \div 1)$ what the secret ket E was. That is, with probability $1/16$ she will know for certain (i.e. with probability $1/1$) that it was 0, with probability $1/16$ that it was 1, with $1/16$ it was 2 etc. If the prior distribution were different, then the outer of the hyper would be correspondingly different: but in each case the second column, the inners, would be "probability 1" throughout. Compare that with the "perfect channel" of Fig. 5 — it has the same effect, making a hyper all of whose inners are point distributions.

The second case *hyper235*, again with uniform choice of E over 4 bits, is what we are more interested in: it is not a perfect channel for E. In that case we can see what happens with divisor d's being chosen uniformly from $[2, 3, 5]$:

$$
\begin{array}{l}
\mathit{hyper235} = \mathit{project}\ (\mathit{sem}_{\mathrm{DD}}\ (\mathit{exponentiation}\ [2, 3, 5]) \\
\qquad\qquad\qquad\qquad (\mathit{uniform}\ [\mathit{initSE}\ 6\ \mathit{exp} \mid \mathit{exp} \leftarrow [0\,..\,15]]))
\end{array}
$$

The value of *hyper235* is shown in Fig. 10. Surprisingly, there are still cases where E is learned exactly by the adversary: for example, with probability

```
> hyper2
1 ÷ 16   1 ÷ 1   0
1 ÷ 16   1 ÷ 1   1
1 ÷ 16   1 ÷ 1   2
1 ÷ 16   1 ÷ 1   3
1 ÷ 16   1 ÷ 1   4
1 ÷ 16   1 ÷ 1   5
1 ÷ 16   1 ÷ 1   6
1 ÷ 16   1 ÷ 1   7
1 ÷ 16   1 ÷ 1   8
1 ÷ 16   1 ÷ 1   9
1 ÷ 16   1 ÷ 1   10
1 ÷ 16   1 ÷ 1   11
1 ÷ 16   1 ÷ 1   12
1 ÷ 16   1 ÷ 1   13
1 ÷ 16   1 ÷ 1   14
1 ÷ 16   1 ÷ 1   15
```

Figure 9 Hyper *hyper2* produced by running the program of Fig. 8 when d=[2].

$^1/_{432}$ she will learn that E=12 is certain (and similarly 13, 14, 15). But her probability $^1/_{432}$ of learning that is very low. On the other hand, with a higher probability $^{41}/_{144}$, i.e. about $^1/_3$, the adversary will learn only that E is in the set {3..14} with certain probabilities. Thus in the second case the hyper shows that with low probability the adversary learns a lot, but with high probability the adversary learns only a little.

We now discuss further the significance of *hyper235* resulting from running the exponentiation program when d can be 2,3 or 5. A rational adversary will guess that the value of *exp* is the one of highest probability: thus in the case $^{41}/_{144}$ mentioned above, she will guess that *exp* is 7. To find out her overall probability of guessing correctly, we take the average of those maxima.

As in the previous example (§6.1) we use the Bayes Vulnerability *bv* of a distribution, the maximum probability of any element:

```
bv :: Ord a ⇒ Dist a → Prob
bv = maximum ∘ map snd ∘ runD ∘ reduction
```

and we average that value over *hyper235* using *condEntropy*:

```
condEntropy :: (Dist a → Rational) → Dist (Dist a) → Rational
condEntropy e h = average (fmap e h) where
  average :: Dist Rational → Rational   — Average a distr. of Rational's.
  average d = sum [r × p | (r, p) ← runD d]
```

$>$ *hyper235*

$1 \div 16$	$1 \div 1$	0	$7 \div 72$	$1 \div 14$	5
$7 \div 48$	$3 \div 7$	1		$1 \div 7$	7
	$2 \div 7$	2		$1 \div 14$	8
	$1 \div 7$	3		$1 \div 21$	9
	$1 \div 7$	4		$1 \div 14$	10
$5 \div 36$	$3 \div 20$	2		$3 \div 14$	11
	$3 \div 20$	3		$1 \div 14$	12
	$1 \div 10$	4		$4 \div 21$	13
	$3 \div 20$	5		$5 \div 42$	14
	$3 \div 20$	6	$17 \div 216$	$3 \div 34$	6
	$1 \div 20$	8		$3 \div 34$	8
	$1 \div 20$	9		$3 \div 34$	9
	$1 \div 10$	10		$5 \div 34$	10
	$1 \div 20$	12		$3 \div 17$	12
	$1 \div 20$	15		$3 \div 17$	14
$41 \div 144$	$3 \div 41$	3		$4 \div 17$	15
	$3 \div 41$	4	$2 \div 27$	$3 \div 32$	7
	$5 \div 41$	5		$3 \div 32$	9
	$3 \div 41$	6		$3 \div 32$	10
	$6 \div 41$	7		$1 \div 4$	11
	$5 \div 41$	8		$3 \div 16$	13
	$4 \div 41$	9		$3 \div 32$	14
	$1 \div 41$	10		$3 \div 16$	15
	$3 \div 41$	11	$1 \div 108$	$1 \div 4$	8
	$2 \div 41$	12		$3 \div 4$	12
	$3 \div 41$	13	$5 \div 432$	$1 \div 5$	9
	$3 \div 41$	14		$3 \div 5$	13
$31 \div 432$	$3 \div 31$	4		$1 \div 5$	14
	$6 \div 31$	6	$1 \div 108$	$1 \div 4$	10
	$2 \div 31$	8		$1 \div 2$	14
	$3 \div 31$	9		$1 \div 4$	15
	$6 \div 31$	10	$1 \div 144$	$1 \div 3$	11
	$5 \div 31$	12		$2 \div 3$	15
	$6 \div 31$	15	$1 \div 432$	$1 \div 1$	12
			$1 \div 432$	$1 \div 1$	13
			$1 \div 432$	$1 \div 1$	14
			$1 \div 432$	$1 \div 1$	15

Figure 10 Hyper *hyper234* produced by running the program of Fig. 8 when $\mathbb{d}$=[2, 3, 5].

yielding the conditional entropy:

$>$ *condEntropy bv hyper235*
$7 \div 24$ (22)

We see that her chance of guessing *exp* is now less than $1/3$, significantly less than the 'can guess with certainty' of *hyper2*:

```
> condEntropy bv hyper2
1 ÷ 1
```

A more interesting situation however is one where we look at the adversary in more abstract terms: it is not the secret key E that she wants, but rather the money she can get by using it. If that were say \$1, then her expected profit from an attack on *hyper235* is from (22) of course $\$^{161}/_{1296}$, i.e. about 12 cents. But now –even more abstractly– we imagine that if she guesses incorrectly, she is caught in the act and is punished: the extra abstraction is then that we assign a notional cost to her of say \$5 for being punished. In this setting then, one might imagine that she would never bother to guess the key: as we saw, her probability of guessing correctly is only about $^1/_8$, and thus of guessing incorrectly is $^7/_8$, giving an expected profit of $^1/_8 \times 1 - {}^7/_8 \times 5$, i.e. a *loss* of about \$4.25.

If that *were* so then, since she is rational, she would not guess at all: it is too risky because she will lose on average \$4.25 every time she does. But it is *not* so: that is the wrong conclusion. Recall for example that with probability $^1/_{432}$ she will learn that *exp* was 12 (and similarly for 13, 14, 15) [27] — and in those cases, she *will* guess. With a bit of arithmetic, we capture the true scenario of gaining \$1 if the guess is correct and losing \$5 if guess is incorrect as follows:

```
jail :: Ord a ⇒ Dist a → Rational
jail d = let m = maximum (map snd (runD (reduction d))) in
           (1 × m − 5 × (1 − m)) `max` 0
```

where the term $1 \times m - 5 \times (1-m)$ represents her expected (abstract) profit, and bounding below by 0 encodes her strategy that if that profit is negative, she won't risk a guess at all. Now we find

```
> condEntropy jail hyper235
31 ÷ 432
> condEntropy jail hyper2
1 ÷ 1
```

— that is, that by choosing rationally to guess the password only when her expected gain is non-negative, the adversary gains 7 cents on average. We see also just above that in the 2-only case she gets the full \$1 on average, because she has no risk of guessing incorrectly: the value of E is completely revealed.

[27] There is also the case where *exp* is 0, in which case she learns that for sure — and guesses 0. In practical circumstances that choice of *exp* would be forbidden; but to keep things simple in the presentation, we have left it in.

Note that the calculations and experiments we just carried out were on the hyper-distributions *hyper2* and *hyper235* and did *not* require the program *exponentiation* at (21) to be re-run on each experiment. Just one run captures in the resulting hyper all the information we need to evaluate various attacker strategies (like *bv* and *jail*).

Finally, we note that Kuifje is not able currently to deal with the large inputs required for realistic cryptographic computations. As our examples show however, it is a useful experimental tool to increase the understanding of the underlying risks associated with those computations, for example side channels in implementations of cryptography, and what can be done about them.

7 Conclusion

7.1 Related work

This paper brings together two ideas for the first time: quantitative information flow for modelling security in programs, and functional programming's generic use of monads to incorporate computational effects in programming languages. The result is a clean implementation of a security-based semantics for programs written in Haskell.

What makes this synthesis possible is that information flows can be modelled in program semantics using the Giry Monad for probabilistic semantics (Giry, 1981) which, as explained above, is applied to the type $\mathbb{D}\mathcal{X}$ rather than the more familiar $\mathcal{X}$ for some state space.

Quantitative information flow for modelling confidentiality was described by (Gray, 1990) and in even earlier work by (Millen, 1987), establishing a relationship between information channels and non-interference of (Goguen and Meseguer, 1984) and the strong dependence of (Cohen, 1977). This last treatment of non-interference turned out however to be unable to impose the weaker security guarantees which are required for practical implementations — it does not allow for partial information flows, for example, which are very difficult and perhaps even impossible to eradicate.

The *channel model* for information flow was mentioned by (Chatzikokolakis et al., 2008) for studying anonymity protocols and was further developed by (Alvim et al., 2012) to include the ideas of gain functions to generalise entropies and secure refinement to enable robust comparisons of information flows between channels. Both of these ideas were already present in earlier work (McIver et al., 2010) which described a model for quantitative information flow in a sequential programming context. A special case of that model

are programs which only leak information without updating variables. Such programs correspond exactly to channels.

(McIver et al., 2015) demonstrated that information flow in programs (and therefore channels too) can be expressed in terms of the Giry monad, unifying the sequential program operator for programs and 'parallel composition' for channels. The idea of refinement's merging posterior behaviour is a generalisation of the way ignorance is handled in qualitative model for information flow (Morgan, 2006, 2009) which is similarly based on monads (for sets rather than distributions).

The abstraction for information flow and state updates that is required for this monadic program semantics is inspired by Hidden Markov Models (Baum and Petrie, 1966), but does not assume that all Markov updates and channel leaks are the same — this generalisation was not present in the original concrete model. Others have also used a concrete version of Hidden Markov Models for analysing information flow in programs (Clark et al., 2005a,b) and do not consider refinement.

Probability in sequential program semantics to express randomisation (but not information flow) was originally due to (Kozen, 1981) although it was not presented in the monadic form used here; that seems to be due to (Lawvere, 1962) and then later brought to a wider audience by (Giry, 1981), as mentioned above.

Haskell's use of monads goes back to (Moggi, 1991), who famously showed that monads provide a semantic model for computational effects, and (Jones and Plotkin, 1989) used this to present a monadic model of a probabilistic lambda calculus. (Wadler, 1992) promoted Moggi's insight within functional programming, with the consequence that monads now form a mainstream programming abstraction, especially in the Haskell programming language. In particular, several people (Ramsey and Pfeffer, 2002; Erwig and Kollmansberger, 2006; Gibbons and Hinze, 2011) have explored the representation of probabilistic programs as Kleisli arrows for the probability monad.

7.2 Discussion

The approach we have taken to providing an executable model of *QIF* is as an *embedded domain-specific language* (Hudak, 1996; Gibbons, 2013) called Kuifje, hosted within an existing general-purpose language. That is, we have not taken the traditional approach of designing a standalone language for *QIF*, and building a compiler that translates from *QIF* concrete syntax to some more established language. Instead, we have defined a datatype *Kuifje* to represent *QIF* abstract syntax trees as values, a semantic domain $\rightarrow_{\text{DD}}$

to represent the behaviour of *QIF* programs, and a translation function $\mathit{hyper} :: \mathit{Ord}\ s \Rightarrow \mathit{Kuifje}\ s \to (s \to_{\mathrm{DD}} s)$ from abstract syntax to semantic domain — all within an existing host language. In our case, that host language is Haskell, although that fact is not too important — we could have chosen OCaml, or Scala, or F#, or any one of a large number of alternatives instead.

Embedded DSLs offer a number of benefits over standalone languages, especially when it comes to early exploratory studies. For example, one can reuse existing tools such as type-checkers, compilers, and editing modes, rather than having to build one's own; moreover, programs in the DSL may exploit features of the host language such as definition mechanisms, modules, and basic datatypes, so these do not have to be added explicitly to the language. These benefits make it quick and easy to build a prototype for the purposes of studying a new language concept. On the other hand, programs in the embedded DSL have to be encoded as abstract syntax trees, and written using the syntactic conventions of the host language, rather than enjoying a free choice of notation best suited to the task; this can be a bit awkward. Once DSL design decisions have been explored and the language design is stable, and the number of users and uses starts to grow, it is easier to justify the additional effort of developing a standalone implementation, perhaps taking the embedded DSL implementation as a starting point.

Our Haskell implementation is inspired by work on *algebraic effects and handlers* (Pretnar, 2015), a language concept that assigns meaning to a syntax tree built out of effectful operations by folding over it with an appropriate algebra known as a handler. While the conventional approach of algebraic effects and handlers applies to trees that have a free-monad structure, our approach is an instance of the generalized framework of (Pieters et al., 2017) that admits handlers for trees with a generalized monoid structure (Rivas and Jaskelioff, 2017) like our plain monoids.

The monoidal representation of programs as state transformers, while relatively simple, has one big limitation: it requires that the type of the initial state is the same as that of intermediate states and of the final state. This means that the program cannot introduce local and result variables, or drop the initial variables and local variables. This leads to awkward models where the initial state contains dummy values for local variables that are initialized in the program. We can overcome this limitation and model heterogenous state by moving –within the framework of generalized monoids– from plain monoids over set to Hughes' *arrows* (Hughes, 2000), which are monoids in a category of so-called *profunctors*.

Acknowledgements

We are grateful for the helpful comments of Nicolas Wu and other members of IFIP Working Group 2.1 on *Algorithmic Languages and Calculi*. Annabelle McIver and Carroll Morgan thank the Australian Research Council for its support via grants DP120101413 and DP141001119; Morgan thanks the Australian Government for its support via CSIRO and Data61. Tom Schrijvers thanks the Research Foundation – Flanders (FWO). Gibbons and Schrijvers thank the Leibniz Center for Informatics: part of this work was carried out during Dagstuhl Seminar 18172 on *Algebraic Effect Handlers go Mainstream*.

Appendix A *A-priori-* and *a-posteriori* distributions

In §4.3 an example compared two programs that released information about a variable x, initially distributed uniformly so that $0 \leq x < 3$. Here we show how those numbers are calculated. Note that this is not an innovation of this paper: we are merely filling in the background of the conventional treatment of priors (*a priori* distributions) and posteriors (*a posteriori* distributions), for those who might not be familiar with them.

In our example x is initially either 0, 1 or 2 with probability $1/3$ for each, i.e. the uniform distribution: this is called *a priori* because it is what the observer believes *before* any leaks have occurred. *Prior* is short for *a-priori* distribution.

If $x \bmod 2$ is leaked, then the observer will see either 0 –when x is 0 or 2–, or 1 when x is 1. The 0 observation occurs with probability $2/3$, because that is the probability that x is 0 or 2; observation 1 occurs with remaining probability $1/3$. In the 0 case, the observer reasons that x cannot be 1; and that its probability of being 0 or 2 is $1/2$ each, because their initial probabilities were equal. This is the *a posteriori* distribution, conditioned on the observation's being 0, so that x is equally likely to be 0 or 2, and cannot be 1. In the 1 case, she reasons that x must be 1 (and can't be 0 or 2).

The corresponding calculations if $x \div 2$ is leaked are that when 0 is observed, she reasons that x is equally likely to be 0 or 1; and if she sees 1, she knows for sure that x is 2.

Now if the choice is made between $x \bmod 2$ and $x \div 2$ with probability $1/2$, and the observer knows whether mod or $\div$ is used then, whichever it turns out to have been used, she will be able to carry out the corresponding

reasoning above. And so overall her conclusions are the weighted sum of the separate outcomes, i.e. their average in this case. Thus:

- with probability $1/2 \times 2/3 = 1/3$ she will know that x is either 0 or 2 (with equal probability for each, as explained above);
- with probability $1/2 \times 1/3 = 1/6$ she will know that x is 1;
- with probability $1/2 \times 2/3 = 1/3$ she will know that x is either 0 or 1; and
- with probability $1/2 \times 1/3 = 1/6$ she will know that x is 2.

The other situation is when the observer does *not* know whether mod or $(\div)$ was used. In that case she will see 0 with probability $1/2 \times 2/3 + 1/2 \times 2/3 = 2/3$, either because mod was used and x was 0 or 2, or because $(\div)$ was used and x was 0 or 1. And in that case 0 is twice as likely as each of the other two, so that the posterior is that x is 0 with probability $1/2$ and is 1 or 2 with probability $1/4$ each. When she sees 1, with probability $1/2 \times 1/3 + 1/2 \times 1/3 = 1/3$, she will reason *a posteriori* that x is equally likely to be 1 or 2.

These two cases are handled automatically by the semantics we have defined. The first is modelled by a conditional, leaking which branch is taken:

```
modOrDiv1 :: Kuifje Int
modOrDiv1 =
  cond (λs → uniform [True, False])
    (observe (λx → return (x `mod` 2)))
    (observe (λx → return (x `div` 2)))
```

Its effect is as follows:

```
> sem_DD modOrDiv1 (uniform [0..2])
1 ÷ 3   1 ÷ 2   0
        1 ÷ 2   1
1 ÷ 3   1 ÷ 2   0
        1 ÷ 2   2
1 ÷ 6   1 ÷ 1   1
1 ÷ 6   1 ÷ 1   2
```

The second case is modelled by a probabilistic observation

```
modOrDiv2 :: Kuifje Int
modOrDiv2 = observe (λx → (x `mod` 2) 1/2⊕ (x `div` 2))
```

whose effect is

```
> sem_DD modOrDiv_2 (uniform [0..2])
2 ÷ 3   1 ÷ 2   0
        1 ÷ 4   1
        1 ÷ 4   2
1 ÷ 3   1 ÷ 2   1
        1 ÷ 2   2
```

Appendix B A password checker

The state is a record of five fields: a password *_pw* and a guess *_gs*, each a list of characters; a loop counter *_i*; a list *_l* of indices still to check; and a Boolean result *_ans*. Each of the programs uses either *_i* or *_left* to control the loop, but not both; for simplicity, we use a common state record for them all the programs.

```
data SP = SP {_pw :: [Char], _gs :: [Char], _l :: [Int], _i :: Int, _ans :: Bool}
  deriving (Show, Eq, Ord)
```

Here is some boilerplate that invokes Template Haskell to generate a lens (a particular higher-order function) for each of the state variables: each acts as a getter and setter for its associated variable.

```
makeLenses '' SP
```

Function *makeState* takes a value for the password *pw* and for the guess *gs* and produces a state containing those values (setting the other variables to appropriate defaults):

```
makeState :: [Char] → [Char] → SP
makeState pw gs = SP {_pw = pw, _gs = gs, _l = [], _i = 0, _ans = True}
```

At the end of the run, we will project the five-variable hyper onto a hyper for *pw* alone, since that is the secret the adversary is trying to discover:

```
projectPw :: Dist (Dist SP) → Dist (Dist [Char])
projectPw = fmap (fmap (λs → s.pw))
```

A number of versions of the program now follow. Each starts not from an initial *state*, but rather from an initial distribution over states. We will make that a uniform distribution over all permutations of a password, and a single fixed guess.

```
initialDist pw gs = uniform [makeState pw' gs | pw' ← permutations pw]
```

```
basicI :: Int → Kuifje SP
basicI n =
  update (λs → return (s.i := 0)) ⨟                          — i := 0;
  update (λs → return (s.ans := True)) ⨟                     — ans := true
  while (λs → return (s.ans ∧ s.i < n))                      — while (ans ∧ i < N) do
    (                                                        — begin
    cond (λs → return ((s.pw !! s.i) ≠ (s.gs !! s.i)))       —   if (pw [i] ≠ gs [i])
      (update (λs → return (s.ans := False)))                —     then ans := false
      skip ⨟                                                 —     else skip
    (update (λs → return (s.i := (s.i + 1))))                —   i++
    )                                                        — end
```

Figure B.1 Basic password checker, with early exit

The first program, shown in Figure B.1, checks the guess against the password character-by-character, and exits the loop immediately if a mismatch is found.

Now we prepare to run the program and use *projectPw* to discover the hyper over *pw* that results.

$$hyperI\ pw\ gs = projectPw\ (sem_{\mathrm{DD}}\ (basicI\ (length\ pw))\ (initialDist\ pw\ gs))$$

Here we choose as possible passwords all permutations of `"abc"` and actual guess `"abc"`. It yields the following output, showing that the early exit does indeed leak information about the password: how long a prefix of it agrees with the guess:

```
> hyperI "abc" "abc"
1 ÷ 6     1 ÷ 1     "abc"
1 ÷ 6     1 ÷ 1     "acb"
2 ÷ 3     1 ÷ 4     "bac"
          1 ÷ 4     "bca"
          1 ÷ 4     "cab"
          1 ÷ 4     "cba"
```

The first inner, with probability $^1/_6$, is the case where the password is correctly guessed: only then will the loop run to completion, because of our choice of passwords and guess — if the guess is correct for the first two characters, it must be correct for the third also.

The second inner corresponds to the loop's exiting after the second iteration: here, again because of the particular values we have chosen, the only possibility is that the first letter of the guess is correct but the second and third are swapped.

The third inner is the case where the loop is exited after one iteration:

```
basicL :: Int → Kuifje SP
basicL n =
  update (λs → return (s.i := 0)) ⨟                    — i := 0;
  update (λs → return (s.ans := True)) ⨟               — ans := true
  while (λs → return (s.i < n))                        — while i < N do
    (                                                  — begin
    cond (λs → return ((s.pw !! s.i) ≠ (s.gs !! s.i))) —   if (pw [i] ≠ gs [i])
      (update (λs → return (s.ans := False)))          —     then ans := false
      skip ⨟                                           —     else skip
    (update (λs → return (s.i := (s.i + 1))))          —   i++
    )                                                  — end
```

Figure B.2 Basic password checker, without early exit

then the first letter must be incorrect (2 possibilities), and the second and third can be in either order (2 more possibilities), giving $^1/_{2\times2}$ for the inner probabilities.

For our second example of this program, we use a guess `"axc"` that is not one of the possible passwords: here, as just above, the $^2/_3$ inner corresponds to an exit after the first iteration. Unlike the above, there is a $^1/_3$ inner representing exit after the second iteration — guaranteed because the second character `"x"` of the guess is certainly wrong. In this case however, the adversary learns nothing about whether the password ends with `"bc"` or with `"cb"`.

```
> hyperI "abc" "axc"
1 ÷ 3     1 ÷ 2     "abc"
          1 ÷ 2     "acb"
2 ÷ 3     1 ÷ 4     "bac"
          1 ÷ 4     "bca"
          1 ÷ 4     "cab"
          1 ÷ 4     "cba"
```

In our second program *basicL* we try to plug the leak that *basicI* contains simply by removing the loop's early exit. It is shown in Fig. B.2.

We run it with

$$hyperL\ pw\ gs = projectPw\ (sem_{\mathrm{DD}}\ (basicL\ (length\ pw))\ (initialDist\ pw\ gs))$$

and obtain this surprising result:

```
> hyperL "abc" "abc"
1 ÷ 6     1 ÷ 1     "abc"
1 ÷ 6     1 ÷ 1     "acb"
```

```
basicM :: Int → Kuifje SP
basicM n =
  update (λs → return (s.i := 0)) ⨟                  — i := 0;
  update (λs → return (s.ans := True)) ⨟             — ans := true
  while (λs → return (s.i < n))                      — while i < N do
    (                                                — begin
    (update (λs → return (s.ans :=                   —    ans:=
      (s.ans ∧ (s.pw !! s.i) ≡ (s.gs !! s.i))))) ⨟   —       ans ∧ (pw [i] = gs [i]);
    (update (λs → return (s.i := (s.i + 1))))        —    i++
    )                                                — end
```

Figure B.3 Basic password checker, without early exit and without leaking conditional

```
1 ÷ 6    1 ÷ 1    "bac"
1 ÷ 3    1 ÷ 2    "bca"
         1 ÷ 2    "cab"
1 ÷ 6    1 ÷ 1    "cba"
```

Still the program is leaking information about the password, even though the loop runs to completion every time — and this, we now realise, is because the condition statement within the loop is leaking its condition. We knew that, but had perhaps forgotten it: remember "Don't branch on high."

Our next attempt therefore is to replace the leaking conditional with an assignment of a conditional expression, which is how we make the Boolean $pw\,[i] \neq gs\,[i]$ unobservable. That is shown in Fig. B.3, and we find

$$hyperM\ pw\ gs = projectPw\ (sem_{\mathrm{DD}}\ (basicM\ (length\ pw))\ (initialDist\ pw\ gs))$$

```
> hyperM "abc" "abc"
1 ÷ 1    1 ÷ 6    "abc"
         1 ÷ 6    "acb"
         1 ÷ 6    "bac"
         1 ÷ 6    "bca"
         1 ÷ 6    "cab"
         1 ÷ 6    "cba"
```

indicating that in this case the adversary discovers nothing about the password at all: the resulting hyper, projected onto pw, is a singleton over an inner whose probabilities are simply those we knew before running the program in the first place.

But at this point we should wonder why the adversary does not discover

```
basicN :: Int → Kuifje SP
basicN n =
  update (λs → return (s.i := 0)) ⨟                  — i := 0;
  update (λs → return (s.ans := True)) ⨟             — ans := true
  while (λs → return (s.i < n))                      — while i < N do
    (                                                — begin
    (update (λs → return (s.ans :=                   —    ans:=
      (s.ans ∧ (s.pw !! s.i) ≡ (s.gs !! s.i))))) ⨟   —       ans ∧ (pw [i] = gs [i]);
    (update (λs → return (s.i := (s.i + 1))))        —    i++
    ) ⨟                                              — end;
  observe (λs → return (s.ans))                      — observe ans
```

Figure B.4 Basic password checker, success observed

the password when she guesses correctly; and we should wonder as well why we haven't noticed that issue before...

The reason is that in our earlier examples the adversary was learning whether she had guessed correctly merely by observing the side channel! That is, the leak was so severe she did not even have to look to see whether the password checker had accepted her guess or not. Only now, with the side channel closed, do we discover that we have accidentally left off the final **observe** *ans* that models the adversary's learning the result of her guess. We remedy that in Fig. B.4, and find

$$hyperN\ pw\ gs = projectPw\ (sem_{\mathrm{DD}}\ (basicN\ (length\ pw))\ (initialDist\ pw\ gs))$$

```
> hyperN "abc" "abc"
1 ÷ 6     1 ÷ 1     "abc"
5 ÷ 6     1 ÷ 5     "acb"
          1 ÷ 5     "bac"
          1 ÷ 5     "bca"
          1 ÷ 5     "cab"
          1 ÷ 5     "cba"
```

that is that with probability $^1/_6$ the adversary learns the password exactly, because she guessed it correctly; but when she guesses incorrectly, she finds none of the passwords she didn't guess to be any more likely than any other.

And now –finally– we come to our obfuscating password checker that compares the characters of the password and the guess in a randomly chosen order. It is in Fig. B.5.

Running *basicR* we discover that in the $^1/_6$ case the adversary guesses the password correctly, she is of course still certain what it is. But now, when

```
basicR :: Int → Kuifje SP
basicR n =
  update (λs → return (s.l := [0 .. n − 1])) ⨟          — l := [0, ..., n − 1];
  update (λs → return (s.ans := True)) ⨟                — ans := true;
  while (λs → return (s.ans ∧ not (null (s.l))))        — while (ans ∧ l ≠ []) do
    (                                                   — begin
    update (λs → uniform [s.i := j | j ← s.l]) ⨟        —   i := uniform (l);
    (update (λs → return (s.ans :=                      —   ans:=
      (s.ans ∧ (s.pw !! s.i) ≡ (s.gs !! s.i))))) ⨟      —     ans ∧ (pw [i] = gs [i]);
    (update (λs → return (s.l := (s.l \\ [s.i]))))      —     l := l − {i}
    ) ⨟                                                 — end;
  observe (λs → return (s.ans))                         — observe ans
```

Figure B.5 Randomized password checker

she does *not* guess correctly, she knows much less than she did in the case *hyperI*, where we began, where early exit leaked the length of the longest matching prefix.

```
hyperR pw gs = projectPw (sem_DD (basicR (length pw)) (initialDist pw gs))
```

```
> hyperR "abc" "abc"
1 ÷ 6     1 ÷ 1     "abc"
2 ÷ 3     1 ÷ 6     "acb"
          1 ÷ 6     "bac"
          1 ÷ 4     "bca"
          1 ÷ 4     "cab"
          1 ÷ 6     "cba"
1 ÷ 6     1 ÷ 3     "acb"
          1 ÷ 3     "bac"
          1 ÷ 3     "cba"
```

The difference in security between *basicI* and *basicR* is clearly revealed by taking the conditional Bayes entropy of each, i.e. (as we saw in the *exponential* example) the probability that an adversary will be able to guess the password after running the checker. We find

```
> condEntropy bv (hyperR "abc" "abc")
7 ÷ 18
> condEntropy bv (hyperI "abc" "abc")
1 ÷ 2
```

that is that the chance is $^1/_2$ for the "check in ascending order" version *basicI*, but it is indeed slightly less, at $^7/_{18}$, in the case that the order is random.

For longer passwords, printing the hyper is not so informative; but still we can give the conditional Bayes vulnerability (and other entropies too). We find for example that the obfuscated algorithm, even with its early exit, reduces the probability of guessing the password by half:

```
> condEntropy bv (hyperI "abcde" "abcde")
1 ÷ 24
> condEntropy bv (hyperR "abcde" "abcde")
13 ÷ 600
```

There are other entropies, of course: one of them is "guessing entropy" which is the average number of tries required to guess the secret: the adversary's strategy is to guess possible secret values one-by-one in decreasing order of their probability. [28]

We define

```
ge :: Ord a ⇒ Dist a → Prob
ge = sum ∘ zipWith (∗) [1 . .] ∘ sortBy (flip compare) ∘ map snd ∘
     runD ∘ reduction
```

and find

```
> condEntropy ge (hyperR "abc" "abc")
7 ÷ 3
> condEntropy ge (hyperI "abc" "abc")
2 ÷ 1
```

that is that the average number of guesses for a three-character password is just more than in the sequential case, where the average number of guesses is exactly 2.

For five-character passwords (of which there are $5! = 120$) we find

```
> condEntropy ge (hyperR "abcde" "abcde")
5729 ÷ 120
> condEntropy ge (hyperI "abcde" "abcde")
1613 ÷ 40
```

that is about 47 guesses for the obfuscated version, on average, versus about 40 guesses for the sequential version. For six-character passwords we find

[28] This does not mean that the adversary runs the password checker many times: rather it means that she runs it once (only) and, on the basis of what she learns, makes successive guesses "on paper" as to what the password actually is.

```
> condEntropy bv (hyperI "abcdef" "abcdef")
1 ÷ 120
> condEntropy bv (hyperR "abcdef" "abcdef")
3 ÷ 800
> condEntropy ge (hyperI "abcdef" "abcdef")
20571 ÷ 80
> condEntropy ge (hyperR "abcdef" "abcdef")
214171 ÷ 720
```

which is about probability 0.008 vs. 0.004 for Bayes vulnerability, and expected guesses 257 vs. 297 for guessing entropy of the sequential vs. randomised versions respectively. Those results suggest that the extra security might not be worth the effort of the obfuscation, at least in these examples.

Finally, we might wonder that –since now we are again allowing an early (though obfuscated) exit– whether there is any longer a reason to replace our original conditional in *basicI* and *basicL* with the "atomic" assignment to *ans* in *basicM* and its successors. After all, now that the loop's exit is (once again) observable, the adversary knows what the "answers" *ans* must have been: a succession of *true*'s and then perhaps a *false*. The randomisation of i ensures however that, so to speak, she does not know the questions. Thus (one last time) we re-define our program:

```
hyperS pw gs = projectPw (sem_DD (basicS (length pw)) (initialDist pw gs))
```

```
> hyperS "abc" "abc"
1 ÷ 6    1 ÷ 1    "abc"
2 ÷ 3    1 ÷ 6    "acb"
         1 ÷ 6    "bac"
         1 ÷ 4    "bca"
         1 ÷ 4    "cab"
         1 ÷ 6    "cba"
1 ÷ 6    1 ÷ 3    "acb"
         1 ÷ 3    "bac"
         1 ÷ 3    "cba"
```

And indeed we find replacing the conditional seems to offer no extra security.

```
basicS :: Int → Kuifje SP
basicS n =
  update (λs → return (s.l := [0 .. n − 1])) ⨟          — l := [0, ..., n − 1];
  update (λs → return (s.ans := True)) ⨟                — ans := true;
  while (λs → return (s.ans ∧ not (null (s.l))))        — while (ans ∧ l ≠ [ ]) do
    (                                                   — begin
    update (λs → uniform [s.i := j | j ← s.l]) ⨟        —   i := uniform (l);
    cond (λs → return ((s.pw !! s.i) ≠ (s.gs !! s.i)))  —   if (pw [i] ≠ gs [i])
      (update (λs → return (s.ans := False)))           —     then ans := false
      skip ⨟                                            —     else skip
    (update (λs → return (s.l := (s.l \\ [s.i]))))      —     l := l − {i}
    ) ⨟                                                 — end;
  observe (λs → return (s.ans))                         — observe ans
```

Figure B.6 Randomized password checker, but with conditional reinstated.

Bibliography

Alvim, Mário S., Chatzikokolakis, Kostas, Palamidessi, Catuscia, and Smith, Geoffrey. 2012 (June). Measuring Information Leakage using Generalized Gain Functions. Pages 265–279 of: *Proc. 25th IEEE Computer Security Foundations Symposium (CSF 2012)*.

Baum, L. E., and Petrie, T. 1966. Statistical Inference for Probabilistic Functions of Finite State Markov Chains. *The Annals of Mathematical Statistics*, **37**(6), 1554–1563.

Chatzikokolakis, Konstantinos, Palamidessi, Catuscia, and Panangaden, Prakash. 2008. Anonymity protocols as noisy channels. *Information and Computation*, **206**(2), 378–401. Joint Workshop on Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis (FCS-ARSPA 06).

Clark, D., Hunt, S., and Malacaria, P. 2005a. Quantitative Information Flow, Relations and Polymorphic Types. *J. Logic and Computation*, **15**(2), 181–199.

Clark, David, Hunt, Sebastian, and Malacaria, Pasquale. 2005b. Quantified Interference for a While Language. *Electr. Notes Theor. Comput. Sci.*, **112**, 149–166.

Cohen, E.S. 1977. Information Transmission in Sequential Programs. *ACM SIGOPS Operatings Systems Review*, **11**(5), 133–9.

Denning, Dorothy. 1982. *Cryptography and Data Security*. Reading: Addison-Wesley.

Erwig, Martin, and Kollmansberger, Steve. 2006. Probabilistic Functional Programming in Haskell. *Journal of Functional Programming*, **16**(1), 21–34.

Feller, W. 1971. *An Introduction to Probability Theory and its Applications*. second edn. Vol. 2. Wiley.

Gibbons, Jeremy. 2013. Functional Programming for Domain-Specific Languages. Pages 1–28 of: Zsók, Viktória, Horváth, Zoltán, and Csató, Lehel (eds), *Central European Functional Programming School*. Lecture Notes in Computer Science, vol. 8606. Springer.

Gibbons, Jeremy, and Hinze, Ralf. 2011 (Sept.). Just `do` It: Simple Monadic Equational Reasoning. Pages 2–14 of: *International Conference on Functional Programming*.

Giry, Michèle. 1981. A Categorical Approach to Probability Theory. In: Banachewski, B. (ed), *Categorical Aspects of Topology and Analysis*. Lecture Notes in Mathematics, vol. 915. Springer.

Goguen, J.A., and Meseguer, J. 1984. Unwinding and Inference Control. Pages 75–86 of: *Proc. IEEE Symp on Security and Privacy*. IEEE Computer Society.

Gray, J.W. 1990. Probabilistic interference. Pages 170–179 of: *IEEE Symposium on Security and Privacy*.

Hudak, Paul. 1996. Building Domain-Specific Embedded Languages. *ACM Computing Surveys*, **28**(4es), 196.

Hughes, John. 2000. Generalising Monads to Arrows. *Science of Computer Programming*, **37**(1-3), 67–111.

Hutton, Graham. 1999. A Tutorial on the Universality and Expressiveness of Fold. *Journal of Functional Programming*, **9**(4), 355–372.

Jones, Claire, and Plotkin, Gordon D. 1989. A Probabilistic Powerdomain of Evaluations. Pages 186–195 of: *Logic in Computer Science*. IEEE Computer Society.

Kiselyov, Oleg, and Shan, Chung-chieh. 2009. Embedded Probabilistic Programming. Pages 360–384 of: Taha, Walid Mohamed (ed), *IFIP TC2 Working Conference on Domain-Specific Languages*. Lecture Notes in Computer Science, vol. 5658. Springer.

Kozen, D. 1983. A Probabilistic PDL. Pages 291–7 of: *Proceedings of the 15th ACM Symposium on Theory of Computing*. New York: ACM.

Kozen, Dexter. 1981. Semantics of Probabilistic Programs. *Journal of Computer and System Sciences*, **22**(3), 328–350.

Lawvere, F. William. 1962. *The Category of Probabilistic Mappings*. Manuscript.

Malcolm, Grant. 1990. Data Structures and Program Transformation. *Science of Computer Programming*, **14**(2-3), 255–279.

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

McIver, Annabelle, Meinicke, Larissa, and Morgan, Carroll. 2010. Compositional Closure for Bayes Risk in Probabilistic Noninterference. Pages 223–235 of: *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part II*.

McIver, Annabelle, Morgan, Carroll, Smith, Geoffrey, Espinoza, Barbara, and Meinicke, Larissa. 2014a. Abstract Channels and Their Robust Information-Leakage Ordering. Pages 83–102 of: Abadi, Martín, and Kremer, Steve (eds), *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Lecture Notes in Computer Science, vol. 8414. Springer.

McIver, Annabelle, Meinicke, Larissa, and Morgan, Carroll. 2014b. Hidden-Markov Program Algebra with Iteration. *Mathematical Structures in Computer Science*.

McIver, Annabelle, Morgan, Carroll, and Rabehaja, Tahiry. 2015. Abstract Hidden Markov Models: a monadic account of quantitative information flow. In: *Proc. LiCS 2015*.

Millen, J. K. 1987 (April). Covert Channel Capacity. Pages 60–60 of: *1987 IEEE Symposium on Security and Privacy*.

Moggi, Eugenio. 1991. Notions of Computation and Monads. *Information and Computation*, **93**(1).

Morgan, C.C. 2006. *The Shadow Knows:* Refinement of Ignorance in Sequential Programs. Pages 359–78 of: *Math Prog Construction*.

Morgan, C.C. 2009. *The Shadow Knows:* Refinement of Ignorance in Sequential Programs. *Science of Computer Programming*, **74**(8), 629–653.

Peyton Jones, Simon L. 2003. The Haskell 98 Language. *Journal of Functional Programming*, **13**.

Pieters, Ruben, Schrijvers, Tom, and Rivas, Exequiel. 2017. Handlers for Non-Monadic Computations. In: *Implementation and Application of Functional Programming Languages*.

Pretnar, Matija. 2015. An Introduction to Algebraic Effects and Handlers. *Electronic Notes in Theoretical Computer Science*, **319**, 19–35.

Ramsey, Norman, and Pfeffer, Avi. 2002. Stochastic Lambda Calculus and Monads of Probability Distributions. Pages 154–165 of: Launchbury, John, and Mitchell, John C. (eds), *Principles of Programming Languages*. ACM.

Rivas, Exequiel, and Jaskelioff, Mauro. 2017. Notions of Computation as Monoids. *Journal of Functional Programming*, **27**, e21.

Rosenhouse, Jason. 2009. *The Monty Hall Problem: The Remarkable Story of Math's Most Contentious Brain Teaser*. Oxford University Press.

Shannon, C.E. 1948. A mathematical theory of communication. *Bell System Technical Journal*, **27**, 379–423, 623–656.

Smith, Geoffrey. 2009. On the Foundations of Quantitative Information Flow. Pages 288–302 of: de Alfaro, Luca (ed), *Proc. 12th International Conference on Foundations of Software Science and Computational Structures (FoSSaCS '09)*. Lecture Notes in Computer Science, vol. 5504.

Wadler, Philip. 1992. Monads for Functional Programming. In: Broy, Manfred (ed), *Program Design Calculi: Proceedings of the Marktoberdorf Summer School*.

Walter, Colin D. 2002. MIST: An Efficient, Randomized Exponentiation Algorithm for Resisting Power Analysis. Pages 53–66 of: *Topics in Cryptology - CT-RSA 2002, The Cryptographer's Track at the RSA Conference, 2002, San Jose, CA, USA, February 18-22, 2002, Proceedings*.