On may, must and average testing

A non-deterministic and probabilistic higher-order programming metalanguage that incorporates its own program logic

(evolving research notes)

Martín Escardó

6th January 2004, version of 14th January 2005

Must- and average-testing for bounded non-deterministic and probabilistic computations are extensionally semi-decidable by inspecting the source code of programs, but not observable from finitely many runs. We can say that the Rice–Shapiro Theorem fails for this kind of computation.

We consider a higher-type functional programming (meta)language with recursion and non-deterministic and probabilistic features. Instead of considering a program logic for may-, must- and average-testing, we directly incorporate them in the programming language. The full language can be regarded as a sort of Rice–Shapiro completion of its test-free fragment.

We give a conventional big-step evaluation semantics for the test-free fragment, and an operational semantics to the full language by compositional compilation into a deterministic sublanguage.

For may- and average-testing, the target sublanguage needs to include the parallel-convergence test (also known as weak parallel-or). Moreover, this construct is already definable in the source language from may-testing, and also from average-testing. Interestingly, must-testing doesn't require any parallel feature.

We consider a domain-theoretic semantics based on powerdomains. For various kinds of non-deterministic computations we use the Hoare, Smyth, and Plotkin powerdomains, and for probabilistic computations we use the probabilistic powerdomain. For computations combining non-determinism and probability, we use three powerdomains introduced by Tix in her thesis. However, in the case of probability combined with non-determinism, it is not clear how to proceed at higher types due to (seemingly) unresolved technical problems in the model.

(At the moment we don't consider recursive types, but this will be important for the practical and theoretical applications we have in mind. In any case, their inclusion doesn't seem to pose any fundamental obstacle.)

Contents

1 A programming metalanguage for non-determinism and probability

We study a language with deterministic, non-deterministic and probabilistic types, which we refer to as MMA (for May, Must, and Average-testing).

Any term of any deterministic type has only one run, which either produces a single outcome or else diverges. A term of non-deterministic and/or probabilistic type has one or more runs (in general continuum many), each of which either produces a single outcome or else diverges, but different convergent runs may produce different outcomes.

We take the ground types to be deterministic, and the product- and function-type constructions to preserve determinism. We can say that PCF is fully and faithfully embedded in this language by a cartesian-closed functor that preserves the operational semantics.

In fact, a slightly larger deterministic language is fully and faithfully embedded in our language. This deterministic language is introduced in Section 1.1. In Section 2.2, we give an operational semantics of the full language by compositional compilation into the deterministic fragment, where the translation is the identity on this fragment.

1.1 A deterministic language

This section will be properly written later — at the moment I am concentrating my efforts in the others. We consider an extension of (call-by-name) PCF with product types and base types Σ and $\mathtt{I}$. Thus, our ground types are

$$\gamma ::= \mathtt{Nat} \mid \mathtt{Bool} \mid \Sigma \mid \mathtt{I}.$$

Here `Nat` and `Bool` are types of natural numbers and booleans, including a divergent element but no multi-valued elements; Σ is a Sierpinski type for results of observations or semi-decisions, with an element $\top$ (observable true) and divergence (unobservable false); and `I` is a "vertical unit-interval type" with elements in $[0, 1]$ (contextually) ordered by magnitude (hence zero is bottom and one is top). This will be made precise in two ways, one using operational semantics and another using denotational semantics.

For the type `I`, we will decide later what is the minimum one ought to have, and how to formulate the operational semantics (this real-number type is not quite the same as the one considered in Real PCF, but rather similar considerations apply).

Whatever we decide to include, we'll eventually need three terms

$$\sup, \inf, \int : (\mathtt{Cantor} \to \mathtt{I}) \to \mathtt{I}.$$

Here the type

$$\mathtt{Cantor} = (\mathtt{Nat} \to \mathtt{Bool})$$

is thought of as a type of sequences of booleans, where we are mostly concerned with total sequences (for totality defined with respect to a data language, as in my Barbados notes and in my paper with Ho Weng Kin, not with respect to the programming language).

For integration we take the uniform distribution on the total elements of the Cantor type (we'll define this precisely later). Notice that we are not requiring the definability of any functional of type $(\mathtt{I} \to \mathtt{I}) \to \mathtt{I}$.

Additionally, we'll need two terms

$$\exists, \forall : (\mathtt{Cantor} \to \Sigma) \to \Sigma,$$

but these are already definable: for the existential quantifier one needs parallel-convergence, but the universal quantifier is sequentially definable in PCF+Σ (see my Barbados notes).

For the sake of clarity, we use the following notation for writing terms, where the letter s ranges over the Cantor type:

$$\begin{aligned}
\sup_s f[s] &= \sup(\lambda s.f[s]),\\
\inf_s f[s] &= \inf(\lambda s.f[s]),\\
\int f[s]\,\mathrm{d}s &= \int(\lambda s.f[s]),\\
\exists s.p[s] &= \exists(\lambda s.p[s]),\\
\forall s.p[s] &= \forall(\lambda s.p[s]),
\end{aligned}$$

Actually, I have made up my mind regarding the choice of primitive constructions and the formulation of the operational semantics. I'll write this down later.

1.2 A language for non-determinism and probability

We consider a "full and faithful" extension of the language of the previous section, in the following sense: there are new types and new terms, but no new terms for the old types. Moreover, for the old terms, the operational semantics remains the same. Thus, PCF + Σ + I remains deterministic when it is embedded into the extension. Non-deterministic and probabilistic terms have to be explicitly typed as such. Types that admit non-deterministic and probabilistic terms are obtained via "powertype" constructors.

Interestingly, and perhaps counter-intuitively, we shall have terms defined on non-deterministic or probabilistic types with values on deterministic types, which hence will produce deterministic outputs from non-deterministic or probabilistic inputs. These arise from the introduction of may-, must- and average-testing constructs.

Types. The ground types, ranged over by γ, are those PCF + Σ + I. General types are ranged over by σ and τ and are given by

$$\sigma, \tau ::= \gamma \mid \sigma \times \tau \mid \sigma \to \tau \mid F\sigma,$$

where F ranges over (unary) type constructors defined by

$$F ::= \mathtt{H} \mid \mathtt{S} \mid \mathtt{P} \mid \mathtt{V} \mid \mathtt{V_H} \mid \mathtt{V_S} \mid \mathtt{V_P}\,.$$

The three constructors $\mathtt{H}, \mathtt{S}, \mathtt{P}$ (Hoare, Smyth and Plotkin powertypes) are for non-deterministic computation, the constructor $\mathtt{V}$ (probabilistic powertype) is for probabilistic computation, and the constructors $\mathtt{V_H}, \mathtt{V_S}, \mathtt{V_P}$ (Hoare, Smyth and Plotkin probabilistic powertypes) are for computation combining probability and non-determinism. Although one can certainly consider, for any type σ, the types $\mathtt{H}\,\mathtt{V}\,\sigma$, $\mathtt{S}\,\mathtt{V}\,\sigma$ and $\mathtt{P}\,\mathtt{V}\,\sigma$, they are not quite the same as $\mathtt{V_H}\,\sigma$, $\mathtt{V_S}\,\sigma$ and $\mathtt{V_P}\,\sigma$, as will become apparent in due course.

Notational conventions. We use the letters C (suggesting closed), Q (compact), L (lens), ν (valuation), $\mathcal{C}$, $\mathcal{Q}$ and $\mathcal{L}$ to range over terms of the powertypes, in the order they have been given above:

$$C\colon \mathtt{H}\,\sigma, \quad Q\colon \mathtt{S}\,\sigma, \quad L\colon \mathtt{P}\,\sigma, \quad \nu\colon \mathtt{V}\,\sigma, \quad \mathcal{C}\colon \mathtt{V_H}\,\sigma, \quad \mathcal{Q}\colon \mathtt{V_S}\,\sigma, \quad \mathcal{L}\colon \mathtt{V_P}\,\sigma.$$

Terms. We extend the inductive definition of PCF + Σ + I terms with the following rules.

Monad rules. Let F be a unary type constructor as above.

Functor rule. If $f\colon \sigma \to \tau$ is a term, then so is

$$Ff\colon F\sigma \to F\tau.$$

Unit rule. For each type σ, we have a term

$$\eta^\sigma_F\colon \sigma \to F\sigma,$$

where in practice we often omit one or both super- and subscripts from η (and from other terms that have similar decorations).

Multiplication rule. For each type σ, we have a term

$$\mu_F^\sigma \colon FF\sigma \to F\sigma.$$

Strength rule. For all types σ, we have a (for the moment nameless) term of type $\sigma \times F\tau \to F(\sigma \times \tau)$. This is needed to get terms $F\sigma_1 \times \cdots \times F\sigma_n \to F\tau$ from terms $\sigma_1 \times \cdots \times \sigma_n \to \tau$ and functoriality. An important example is the construction of various non-deterministic and probabilistic conditionals from the deterministic one.

Choice rules. The Hoare, Smyth and Plotkin (probabilistic or not) powertypes have a binary choice operator $\mathbin{\mathbb{V}}$. The idea is that the runs of a term $M \mathbin{\mathbb{V}} N$ are those of the term M together with those of the term N. The probabilistic powertype and the Hoare, Smyth and Plotkin probabilistic powertypes have a binary choice operator $\oplus$. Again the runs of a term $M \oplus N$ are those of the term M together with those of the term N. However, the choice $M \mathbin{\mathbb{V}} N$ is non-deterministic, whereas the choice $M \oplus N$ is probabilistic. This is made precise below. Notice that the Hoare, Smyth and Plotkin probabilistic powertypes include both choice operators, and hence they combine non-determinism and probability.

Non-deterministic choice rule. For $F \in \{\mathtt{H}, \mathtt{S}, \mathtt{P}, \mathtt{V_H}, \mathtt{V_S}, \mathtt{V_P}\}$, and for each type σ, we have a term

$$(\mathbin{\mathbb{V}}_F^\sigma) \colon F\sigma \times F\sigma \to F\sigma,$$

written in infix notation.

Probabilistic choice rule. For $F \in \{\mathtt{V}, \mathtt{V_H}, \mathtt{V_S}, \mathtt{V_P}\}$, and for each type σ, we have an infix term

$$(\oplus_F^\sigma) \colon F\sigma \times F\sigma \to F\sigma.$$

May, must and average testing rules. None of the seven powertypes types can be distinguished on the base of the runs that their terms have. In order to make the powertypes (look) genuinely different, one considers may-, must- and average-testing. For may-testing, one existentially quantifies over runs, for must-testing one universally quantifies over runs, and for average-testing one integrates over runs, where we assume "uniform distribution over the space of schedulers". Hence the interpretation of the probabilistic choice rule amounts to choosing each branch with probability half. For powertypes combining non-determinism and probability, the situation is more complicated and its discussion is postponed (see Section 1.4).

The Hoare powertype will admit only may-testing, the Smyth powertype will admit only must-testing, and the Plotkin powertype will admit both. The probabilistic powertype will admit average testing, and the powertypes types combining non-determinism and probability, will admit may-average-testing (Hoare

probabilistic powertype), must-average testing (Smyth probabilistic powertype), or both (Plotkin probabilistic powertype).

May and must rules. We think of Σ-valued terms as open sets (we shall discuss later whether they should be regarded as semidecidable or observable properties), and we define a type of opens

$$\mathcal{O}\,\sigma = (\sigma \to \Sigma).$$

May- and must-testing can be seen as ways of obtaining open sets of $F\sigma$ from open sets of σ, for certain non-deterministic type constructors, and hence we postulate corresponding terms $\diamondsuit$ (may) and $\Box$ (must):

$$\begin{array}{rcl} \diamondsuit^{\sigma}_{\mathtt{H}} & : & \mathcal{O}\,\sigma \to \mathcal{O}\,\mathtt{H}\,\sigma, \\ \Box^{\sigma}_{\mathtt{S}} & : & \mathcal{O}\,\sigma \to \mathcal{O}\,\mathtt{S}\,\sigma, \\ \diamondsuit^{\sigma}_{\mathtt{P}} & : & \mathcal{O}\,\sigma \to \mathcal{O}\,\mathtt{P}\,\sigma, \\ \Box^{\sigma}_{\mathtt{P}} & : & \mathcal{O}\,\sigma \to \mathcal{O}\,\mathtt{P}\,\sigma. \end{array}$$

The idea in the case of the Plotkin powertype is that if $u\colon \mathcal{O}\,\sigma$ and $N\colon \mathtt{P}\,\sigma$, then $\diamondsuit(u)(N) = \top$ if and only $u(x) = \top$ for some outcome x of a run of N, and $\Box(u)(N) = \top$ if and only $u(x) = \top$ for all outcomes x of runs of N. This is made precise later. The idea for the Hoare and Smyth powertypes is the same, but only one kind of test is made available for each of them.

Average rule. This time, from an open set of σ we get an expectaction on $\mathtt{V}\,\sigma$, where an expectation on σ is an $\mathtt{I}$-valued function:

$$\mathcal{E}\,\sigma = (\sigma \to \mathtt{I}).$$

By virtue of the vertical nature of the unit-interval type $\mathtt{I}$, any Sierpinski-valued term amounts to an $\mathtt{I}$-valued term with values 0 (bottom) and 1 (top). Hence expectations generalize open sets. We postulate an averaging term

$$\bigcirc^{\sigma}_{\mathtt{V}}\colon\ \mathcal{E}\,\sigma \to \mathcal{E}\,\mathtt{V}\,\sigma.$$

For a term $u\colon \mathcal{O}\,\sigma$ seen as a term of type $\mathcal{E}\,\sigma$, as discussed above, and $N\colon \mathtt{V}\,\sigma$, the idea is that $\bigcirc(u)(N) \in \mathtt{I}$ is the probability that u holds for outcomes x of runs of N.

May-average and must-average rules. We postulate four terms

$$\begin{array}{rcl} \diamondsuit^{\sigma}_{\mathtt{V_H}} & : & \mathcal{E}\,\sigma \to \mathcal{E}\,\mathtt{V_H}\,\sigma, \\ \Box^{\sigma}_{\mathtt{V_S}} & : & \mathcal{E}\,\sigma \to \mathcal{E}\,\mathtt{V_S}\,\sigma, \\ \diamondsuit^{\sigma}_{\mathtt{V_P}} & : & \mathcal{E}\,\sigma \to \mathcal{E}\,\mathtt{V_P}\,\sigma, \\ \Box^{\sigma}_{\mathtt{V_P}} & : & \mathcal{E}\,\sigma \to \mathcal{E}\,\mathtt{V_P}\,\sigma. \end{array}$$

The idea here is subtler and will be explained later.

1.3 Some examples illustrating the intended meaning

Recursively define a term $f\colon \mathtt{Nat} \to \mathtt{P}\,\mathtt{Nat}$ by

$$f(n) = \eta(n) \mathbin{\textcircled{\scriptsize V}} f(n+1),$$

and let $\mathtt{converge}\colon \mathtt{Nat} \to \Sigma$ be a term such that $\mathtt{converge}(n) = \top$ iff $n \neq \bot$. Then we intend that

$$\Diamond\, \mathtt{converge}(f(0)) = \top \quad (\text{“}f(0) \text{ may converge”})$$

and that

$$\Box\, \mathtt{converge}(f(0)) = \bot \quad (\text{“it is not the case that } f(0) \text{ must converge”}),$$

but

$$\Box\, \mathtt{converge}(\eta(0) \mathbin{\textcircled{\scriptsize V}} \eta(1)) = \top.$$

Now recursively define a term $g\colon \mathtt{Nat} \to \mathtt{V}\,\mathtt{Nat}$ by

$$g(n) = \eta(n) \oplus g(n+1),$$

Then we intend that

$$\bigcirc\, \mathtt{converge}(g(0)) = 1 \quad (\text{“the probability that } g(0) \text{ converges is 1”}),$$

and

$$\bigcirc\, \mathtt{converge}_n(g(0)) = 2^{-n-1} \quad (\text{“probability}[g(0) \text{ converges to } n] = 2^{-n-1}\text{”}),$$

where $\mathtt{converge}_n\colon \mathtt{Nat} \to \Sigma$ is a term such that $\mathtt{converge}_n(x) = \top$ iff $x = n$.

We omit, in this version of the notes, examples combining probability and non-determinism.

Parallel-convergence defined from may-testing. Parallel convergence

$$(\vee)\colon \Sigma \times \Sigma \to \Sigma,$$

written in infix notation, can be defined by

$$(p \vee q) = \Diamond\, \mathtt{converge}(\eta(p) \mathbin{\textcircled{\scriptsize V}} \eta(q)),$$

where $\mathtt{converge}\colon \Sigma \to \Sigma$ is the identity function. Of course, to be convinced of this, one needs to know a precisely defined semantics of the language.

We omit, in this version of the notes, a definition of parallel convergence from average-testing.

1.4 Another view of may, must and average testing

This section motivates the operational semantics developed in Section 2.2 and mirrors the denotational semantics given in Sections 2.3 and 3.2. Consider the may-testing term

$$\diamondsuit\colon \mathcal{O}\,\sigma \to \mathcal{O}\,\mathtt{H}\,\sigma$$

and recall that $\mathcal{O}\,\sigma = (\sigma \to \Sigma)$. By uncurrying this term, then twisting the product, and currying again, we get a term that will be natural to denote by

$$\exists\colon \mathtt{H}\,\sigma \to ((\sigma \to \Sigma) \to \Sigma).$$

That is,

$$\exists(C)(u) = \diamondsuit(u)(C).$$

For $C\colon \mathtt{H}\,\sigma$, we write $\exists_C$ rather than $\exists(C)$. Moreover, for a term $u[x]\colon \sigma \to \Sigma$ possibly including a free syntactic variable x, we write $\exists x \in C.u[x]$ rather than $\exists(C)(\lambda x.u[x])$. With this notation, we have

$$\diamondsuit(u)(C) = \exists x \in C.u(x).$$

This tautology motivates the operational semantics for $\diamondsuit$ given in Section 2.2.

Similarly, from the must-testing term

$$\Box\colon \mathcal{O}\,\sigma \to \mathcal{O}\,\mathtt{S}\,\sigma,$$

we get a term

$$\forall\colon \mathtt{S}\,\sigma \to ((\sigma \to \Sigma) \to \Sigma),$$

for which analogous notational conventions are adopted.

For the Plotkin powertype, we get both quantifiers.

For the probabilistic powertype, recalling that $\mathcal{E}\,\sigma = (\sigma \to \mathtt{I})$, from the average-testing term

$$\bigcirc\colon \mathcal{E}\,\sigma \to \mathcal{E}\,\mathtt{V}\,\sigma$$

we get a term

$$\int\colon \mathtt{V}\,\sigma \to ((\sigma \to \mathtt{I}) \to \mathtt{I}).$$

For $\nu\colon \mathtt{V}\,\sigma$, we write $\int_\nu$ rather than $\int \nu$, and, for $f\colon \sigma \to \mathtt{I}$ we write $\int_\nu f$.

For the Hoare probabilistic powertype, from the may-average term

$$\diamondsuit\colon \mathcal{E}\,\sigma \to \mathcal{E}\,\mathtt{V}_{\mathtt{H}}\,\sigma$$

we get a term

$$\begin{array}{rcl} \mathtt{V}_{\mathtt{H}}\,\sigma & \to & ((\sigma \to \mathtt{I}) \to \mathtt{I}) \\ \mathcal{C} & \mapsto & (f \mapsto \sup\limits_{\nu\in\mathcal{C}} \int_\nu f). \end{array}$$

What we mean by this notation is that the application of the nameless term $\mathtt{V}_{\mathtt{H}}\,\sigma \to ((\sigma \to \mathtt{I}) \to \mathtt{I})$ to a term $\mathcal{C}\colon \mathtt{V}_{\mathtt{H}}\,\sigma$ followed by an application to a term $f\colon \sigma \to I$ is written $\sup_{\nu\in\mathcal{C}} \int_\nu f$.

For the Smyth probabilistic powertype, we get a similar term, but we instead write $\inf_{\nu\in\mathcal{Q}} \int_\nu f$. For the Plotkin probabilistic powertype we get both.

2 Non-determinism and probability

In this section we consider the language restricted to the types

$$\sigma, \tau ::= \gamma \mid \sigma \times \tau \mid \sigma \to \tau \mid F\sigma \quad \text{for} \quad F ::= \mathtt{H} \mid \mathtt{S} \mid \mathtt{P} \mid \mathtt{V}\,.$$

Thus, we discuss non-deterministic types and probabilistic types, but not types that embody non-determinism and probability simultaneously, which are postponed to Section 3.

2.1 Big-step semantics of the test-free fragment

For the fragment of the language without the may-, must- and average-testing operators, we extend the big-step operational semantics of PCF + Σ + I with the following rules (where some of the terms defined above should be instead regarded as term constructors for the usual reasons).

Firstly, we stipulate that if $v\colon \sigma$ is a value then so is $\eta(v)\colon F\sigma$. Then, omitting types and contexts, we add the rules:

$$\frac{M \Downarrow v}{M \mathbin{\mathbin{⦶}} N \Downarrow v} \qquad \frac{N \Downarrow v}{M \mathbin{\mathbin{⦶}} N \Downarrow v} \qquad \frac{M \Downarrow v}{M \oplus N \Downarrow v} \qquad \frac{N \Downarrow v}{M \oplus N \Downarrow v}$$

$$\frac{M \Downarrow \eta(v) \quad f(v) \Downarrow w}{Ff(M) \Downarrow \eta(w)} \qquad \frac{M \Downarrow v}{\eta(M) \Downarrow \eta(v)} \qquad \frac{\mathcal{M} \Downarrow \eta(V)}{\mu(\mathcal{M}) \Downarrow V}$$

Thus, all choice operators have the same meaning: this semantics only says what values a term *may* evaluate to, if any. The following is easy to establish:

2.1.1 *For the test-free fragment of the language, there is a ternary relation*

$$M \Downarrow^s v,$$

where s ranges over the Cantor space of infinite binary sequences, such that

1. *for any M and any s there is at most one v with $M \Downarrow^s v$, and*
2. *$M \Downarrow v$ iff there is some s with $M \Downarrow^s v$.*

The idea is that s is a scheduler that dictates which branches the choice operators have to take during evaluation. Once the scheduler is chosen, the evaluation is completely deterministic. For a test-free term M, we can say that

M must converge iff for every s there is v with $M \Downarrow^s v$.

Despite the universal quantification over an uncountable set, we can prove:

2.1.2 *Must-convergence is semidecidable for closed, test-free terms.*

The reason is that the Cantor space is compact (cf. my Barbados notes). However, we don't know how to define this and more general must- and average-testings in the big-step style. Hence we instead translate our language into a deterministic language, using §2.1.1 above as the guiding idea.

2.2 Operational semantics of the full language

We define an operational semantics for our language MMA by compositional compilation into its deterministic sublanguage PCF + Σ + I, where the translation is the identity on this sublanguage. The compilation map

$$\phi\colon \text{MMA} \to \text{PCF} + \Sigma + \mathtt{I}$$

acts on both terms and types (like a functor): for every source term M of type σ, the translation produces a target term $\phi(M)$ of type $\phi(\sigma)$.

Translation of types. This is defined by induction:

$$\begin{array}{rcll} \phi(\gamma) & = & \gamma, & \\ \phi(\sigma \times \tau) & = & \phi(\sigma) \times \phi(\tau), & \\ \phi(\sigma \to \tau) & = & \phi(\sigma) \to \phi(\tau), & \\ \phi(F\sigma) & = & \mathtt{Cantor} \to \phi(\sigma), & \text{for } F \in \{\mathtt{H}, \mathtt{S}, \mathtt{P}, \mathtt{V}\}. \end{array}$$

Recall from Section 1.1 that `Cantor` is the type (`Nat` → `Bool`). As in the previous section, the idea here is that the Cantor type plays the role of a type of schedulers. To run a non-deterministic program, one non-deterministically comes up with a scheduler, and then deterministically runs the program with respect to that scheduler, where the scheduler is used in order to decide which branches of the choice constructs are taken (think e.g. of true as left and of false as right). To run a probabilistic program, one first comes up with a scheduler, where the choice of scheduler is performed with uniform distribution over the Cantor space.

Translation of terms. This is also defined by induction. The translation of a syntactic variable is itself, with its type modified appropriately for powertypes. (In order to be precise here, one needs some more-or-less evident syntactic bureaucracy which I don't want to go into.) For PCF + Σ + I constants, the translation is the identity. It is also the identity on all fixed-point combinators, including those of the non-deterministic and probabilistic types, with a suitable change of types for the latter. Moreover, we stipulate

$$\begin{aligned} \phi(MN) &= \phi(M)\phi(N), \\ \phi(\lambda x.M) &= \lambda\phi(x).\phi(M). \end{aligned}$$

This takes care of the deterministic fragment of the language, for which the translation is then the identity.

We now consider the translation of the testing operators. Recall that the type of may- and must-testing is

$$(\sigma \to \Sigma) \to (F\sigma \to \Sigma),$$

and hence their translations are to have type

$$(\phi(\sigma) \to \Sigma) \to ((\mathtt{Cantor} \to \phi(\sigma)) \to \Sigma).$$

Similarly, the translation of average-testing is to have type

$$(\phi(\sigma) \to \mathtt{I}) \to ((\mathtt{Cantor} \to \phi(\sigma)) \to \mathtt{I}).$$

With this in mind, together with the discussion of Section 1.4, it should be possible to decode the following definition, where all quantifications and integrals are over the Cantor type (as introduced in Section 1.1):

$$\begin{aligned}
\phi(\diamondsuit_{\mathtt{H}}) &= \lambda u.\lambda C.\exists s.u(C(s)),\\
\phi(\Box_{\mathtt{S}}) &= \lambda u.\lambda Q.\forall s.u(Q(s)),\\
\phi(\diamondsuit_{\mathtt{P}}) &= \lambda u.\lambda L.\exists s.u(L(s)),\\
\phi(\Box_{\mathtt{P}}) &= \lambda u.\lambda L.\forall s.u(L(s)),\\
\phi(\bigcirc_{\mathtt{V}}) &= \lambda f.\lambda \nu.\int f(\nu(s))\,\mathrm{d}s.
\end{aligned}$$

Next we consider the compilation of non-deterministic and probabilistic choice operators:

$$\begin{aligned}
\phi(\mathbin{\mathbb{V}}_{\mathtt{H}}) &= \lambda(C_0, C_1).\lambda s.\,\text{if } \mathrm{hd}(s) \text{ then } C_0(\mathrm{tl}(s)) \text{ else } C_1(\mathrm{tl}(s)),\\
\phi(\mathbin{\mathbb{V}}_{\mathtt{S}}) &= \lambda(Q_0, Q_1).\lambda s.\,\text{if } \mathrm{hd}(s) \text{ then } Q_0(\mathrm{tl}(s)) \text{ else } Q_1(\mathrm{tl}(s)),\\
\phi(\mathbin{\mathbb{V}}_{\mathtt{P}}) &= \lambda(L_0, L_1).\lambda s.\,\text{if } \mathrm{hd}(s) \text{ then } L_0(\mathrm{tl}(s)) \text{ else } L_1(\mathrm{tl}(s)),\\
\phi(\oplus_{\mathtt{V}}) &= \lambda(\nu_0, \nu_1).\lambda s.\,\text{if } \mathrm{hd}(s) \text{ then } \nu_0(\mathrm{tl}(s)) \text{ else } \nu_1(\mathrm{tl}(s)).
\end{aligned}$$

Notice that these are all essentially the same definition.

We now consider the translation of the monad structure. For the functor, we define

$$\phi(Ff) = \lambda k.\lambda s.f(k(s)),$$

where k is an element of the translation of the powertype F and s is a scheduler. For the unit, we define

$$\phi(\eta_F) = \lambda x.\lambda s.x.$$

For the multiplication, we assume PCF terms

$$\mathtt{onepart}, \mathtt{theotherpart} \colon \mathtt{Cantor} \to \mathtt{Cantor}$$

such that the term

$$\langle \mathtt{onepart}, \mathtt{theotherpart} \rangle \colon \mathtt{Cantor} \to \mathtt{Cantor} \times \mathtt{Cantor}$$

is a "bijection on total elements" and moreover "evenly splits the uniform distribution" (precise definition in future versions), and we define

$$\phi(\mu_F) = \lambda K.\lambda s.K(\mathtt{onepart}(s))(\mathtt{theotherpart}(s)).$$

In the target language, the monad laws fail (both denotationally and operationally), but we intend them hold modulo the appropriate notion of testing.

It should be now clear how to translate the strength (I'll include this in a future version and it is already included in the Haskell implementation given below).

Some properties of the translation. Here we let s range over closed terms of type $\mathtt{Cantor}$ in the data language of PCF$+\Sigma+\mathtt{I}$ in the sense of my Barbados notes (easier to check my paper with Ho Weng Kin). Otherwise Kleene trees cause trouble: some claims made below would be false.

2.2.1 *If $M\colon F\gamma$ is closed and test-free, then*

$$M \Downarrow \eta(v) \text{ iff there is } s \in \mathtt{Cantor} \text{ such that } \phi(M)(s) \Downarrow v.$$

It is easy to see that, for every ground type γ other than $\mathtt{I}$, and for every value $v\colon \gamma$, there is a PCF$+\Sigma$ program

$$\mathtt{converge}_v\colon \gamma \to \Sigma$$

such that, for all closed terms $M\colon \gamma$,

$$\mathtt{converge}_v M \Downarrow \top \iff M \Downarrow v.$$

2.2.2 *For a closed term $M\colon F\gamma$ with $\gamma \neq \mathtt{I}$,*

1. $\phi(\Diamond\, \mathtt{converge}_v M) \Downarrow \top$ *iff* $\phi(M)(s) \Downarrow v$ *for some* s.
2. $\phi(\Box\, \mathtt{converge}_v M) \Downarrow \top$ *iff* $\phi(M)(s) \Downarrow v$ *for all* s.

The ground type $\mathtt{I}$ and the average-testing operator are a bit subtler and will be discussed in a future version of these notes.

2.3 Denotational semantics

A domain-theoretic denotational semantics is easily obtained — we have temporarily avoided trouble by postponing the combination of probability with non-determinism to Section 3.

Category. Dcpos with bottom and continuous maps. (We cannot take a category of continuous domains because of the troublesome probabilistic powerdomain. But I don't think this is going to cause difficulties.)

Ground types. $\mathtt{Nat}$, $\mathtt{Bool}$, Σ are interpreted as the usual flat domains. $\mathtt{I}$ is intepreted as the unit interval $[0, 1]$ under its natural order.

Constants. Constants other than the testing operators, including fixed-points combinators, are interpreted in the usual way (we still have to properly discuss the constants for the type $\mathtt{I}$).

Products- and function-types, and simply-typed λ-calculus machinery. Cartesian closed structure of the category.

Powertypes. Powerdomains with the same names. This includes the monad structures and the testing operators.

Adequacy. We take this to be the statement that, for M a closed term of ground type other than $\mathtt{I}$,

$$[\![M]\!] = [\![v]\!] \iff \phi(M) \Downarrow v.$$

We'll discuss the type $\mathtt{I}$ in a future version, but we remark that the above statement is sufficient, because adequacy at type $\mathtt{I}$ can be reduced to adequacy at type Σ, very much in the same way as for $\mathtt{Nat}$ and $\mathtt{Bool}$.

Because the model is already known to be computationally adequate for the codomain of the translation, we have the following purely denotational formulation:

2.3.1 *Adequacy holds if and only if* $[\![M]\!] = [\![\phi(M)]\!]$ *for every closed term* M *of ground type other than* $\mathtt{I}$.

Because we have included the testing operators in the language, there is no need to formulate an adequacy property for terms of type $F\gamma$ with γ ground. (But a discussion is in order, which is postponed to a future version of these notes.)

Something to think about. I think we need the powerdomain of probabilistic continuous valuations (i.e. those with total mass 1) to get computational adequacy. Notice that the bottom element of the subprobabilistic powerdomain is strictly smaller than that of the probabilistic one. Hence, I am almost convinced, the domain-theoretic recursions for the subprobabilistic powerdomain won't match the operational recursions. This is due to the fact that our target language is call-by-name. (By the way, it is difficult to say whether the source language is call-by-name or not. I'd rather just say that it is cartesian closed, with product-type and function-types giving the categorical products and exponentials.)

2.4 Side remark: universal properties

Consider again the may-testing term

$$\Diamond\colon \mathcal{O}\,\sigma \to \mathcal{O}\,\mathtt{H}\,\sigma$$

and again recall that $\mathcal{O}\,\sigma = (\sigma \to \Sigma)$, i.e.,

$$\Diamond\colon (\sigma \to \Sigma) \to (\mathtt{H}\,\sigma \to \Sigma).$$

From the point of view of the denotational semantics given in Section 2.3, this constructs, from any given $u\colon \sigma \to \Sigma$, the unique $\mathtt{H}$-algebra homomorphism $\bar{u}\colon \mathtt{H}\,\sigma \to \Sigma$ extending u along $\eta\colon \sigma \to \mathtt{H}\,\sigma$, namely $\bar{u} = \Diamond\, u$. The structure maps of the algebras are uniquely determined by the underlying types, because $\mathtt{H}$ is a KZ-monad, and amounts to the non-deterministic choice operator $\mathbin{\mathbb{V}}$ for $\mathtt{H}\,\sigma$ and parallel convergence for Σ. In general, not every type is the underlying object of a Hoare algebra, because the Hoare structure, when it exists, amounts to binary join in the domain-theoretic order.

A similar observation applies to the Smyth powertype, but things get more interesting. The monad is again KZ, and, additionally, all types are underlying objects of algebras (with domain-theoretic meet as the structure map). Moreover, if one postulates generalized must-testing terms for all ground types γ,

$$\Box\colon (\sigma \to \gamma) \to (\mathtt{S}\,\sigma \to \gamma),$$

then one gets, by structural induction on types, using the strength, generalized must-testing programs for all types τ,

$$\Box\colon (\sigma \to \tau) \to (\mathtt{S}\,\sigma \to \tau),$$

which articulate the universal property of the Smyth powerconstruction. In connection with the discussion of the previous section, by currying, twisting and currying we get a term of type $\mathtt{S}\,\sigma \to ((\sigma \to \tau) \to \tau)$, whose denotation is the functional $Q \mapsto (f \mapsto \inf f(Q))$.

The Plotkin powertype is not a KZ-monad, and types admit at least one structure map and in general more than one (Hoare and Smyth structures are Plotkin structures), as is the case for the Sierpinski type. The may-operator on the Plotkin powertype gives the universal property for the Hoare structure on Σ, and the must-operator for the Smyth structure.

A similar discussion can be easily carried out for the probabilistic powertype, and for the combined versions, taking into account the denotational semantics given in Section 3.2.

2.5 Must-testing implemented in Haskell

Haskell is a call-by-name language (actually, call-by-need, but because it is deterministic and doesn't have any effect other than non-termination, this doesn't make any difference). Notice that a product type $\sigma \times \tau$ is written (σ, τ) in Haskell, i.e. the pairing notation is used for products(!). Notice also that a lambda expression $\lambda x.e$ is written `\x -> e`.

```
-- Martin Escardo, 6th January 2005.
-- A deterministic encoding of bounded non-determinism
-- with must-testing.

-- Sierpinski space:
data S = T

bot :: a
bot = bot

(/\) :: S -> S -> S
T /\ T = T

sif :: S -> a -> a
sif T x = x
```

```
-- Open sets, ranged over by u,v,w:
type O a = (a -> S)

-- Schedulers, ranged over by s,t:
data Scheduler = Left Scheduler | Right Scheduler

biased :: Scheduler
biased = Left biased

-- Any function Scheduler -> (Scheduler,Scheduler) which
-- is "an isomorphism on total elements" will do:
split :: Scheduler -> (Scheduler,Scheduler)
split (Left  (Left  s)) = (Left  s1, Left  s2) where (s1,s2) = split s
split (Left  (Right s)) = (Left  s1, Right s2) where (s1,s2) = split s
split (Right (Left  s)) = (Right s1, Left  s2) where (s1,s2) = split s
split (Right (Right s)) = (Right s1, Right s2) where (s1,s2) = split s

-- A similar remark applies here:
split3 :: Scheduler -> (Scheduler,Scheduler,Scheduler)
split3 s = let (s1,t) = split s in let (s2,s3) = split t in (s1,s2,s3)

-- Universal quantification over total schedulers (see Barbados notes).
-- The naive recursive definition
--   forall u = forall(\s -> u(Left s))) /\ (forall(\s -> u(Right s)))
-- doesn't work. But the following modification does:
forall :: (Scheduler -> S) -> S
forall u = probe u ((forall(\s -> u(Left s))) /\ (forall(\s -> u(Right s))))
         where probe u = \x -> u(sif x biased)

-- A powertype, ranged over by q,r.
type Smyth a = (Scheduler -> a)

smyth :: (a -> b) -> (Smyth a -> Smyth b)
smyth f = \q -> f.q

eta :: a -> Smyth a
eta x = \s -> x

mu :: Smyth (Smyth a) -> Smyth a
mu qq = \s -> let (s1,s2) = split s in qq s1 s2

strength :: (p, Smyth a) -> Smyth (p,a)
strength (x,q) = \s -> (x, q s)

-- The following can be defined using mu, strength, curry and uncurry,
-- but I prefer to define them directly here.
```

```
smyth2 :: (a->b->c) -> (Smyth a) -> (Smyth b) -> (Smyth c)
smyth2 f q1 q2 = \s -> let (s1,s2) = split s in f (q1 s1) (q2 s2)

smyth3 :: (a->b->c->d) -> (Smyth a) -> (Smyth b) -> (Smyth c) -> (Smyth d)
smyth3 f q1 q2 q3 = \s -> let (s1,s2,s3) = split3 s in f (q1 s1) (q2 s2) (q3 s3)

-- Application of smyth3: Non-determinstic, sequential conditional.
nif :: Smyth Bool -> Smyth a -> Smyth a -> Smyth a
nif = smyth3 dif where dif p x y = if p then x else y

-- Erratic choice.
eor :: Smyth a -> Smyth a -> Smyth a
(q `eor` r)(Left  s) =q(s)
(q `eor` r)(Right s) =r(s)

-- Must testing.
must :: O a -> O (Smyth a)
must u = \q -> forall(\s ->u(q s))

-- Material for the discussion following the program.
type Baire = [Int]

-- Intuitively, the following definition is
--   cantor = 0:cantor `eor` 1:cantor
-- But this doesn't typecheck. We have to apply the
-- Smyth functor to the maps (0:) and (1:).
cantor :: Smyth Baire
cantor = (smyth (0:) cantor) `eor` (smyth (1:) cantor)

converge :: Int -> S
converge x = if x==x then T else bot

-- END OF PROGRAM -------------------------------------------------
```

Then, for example, the Haskell interpreter evaluates the term

```
must converge ((eta 2) `eor` (eta 3))
```

to `T`. Similarly, for any total program `f :: Baire->Int`, the term

```
must converge (smyth f cantor)
```

evaluates to `T`. Topologically speaking, what is going on is that f is continuous and hence maps compact sets to compact sets, the Cantor set is compact, and a compact set of total elements of `Int` is finite. For a denotional-free articulation of these ideas, see my paper with Ho Weng Kin.

It is also possible to implement boolean-valued must-testing. In my Barbados notes, and in my paper with Ho Weng Kin, we can find an implementation of a quantifier (due to Simpson, going back to Berger)

```
boolean_forall :: (Scheduler -> Bool) -> Bool
```

that universally quantifies over total predicates defined on schedulers. We omit the Haskell code. With this, one can define

```
type Clopen a = (a -> Bool)

boolean_must :: Clopen a -> Clopen (Smyth a)
boolean_must u = \q -> boolean_forall(\s ->u(q s))
```

Then, for example, the term

```
boolean_must odd ((eta 2) `eor` (eta 3)))
```

evaluates to `False`. In a future version of these notes I plan to include more sophisticated examples (suggestions are very welcome). Have we implemented model-free model checking?

3 Non-determinism combined with probability

In this section we briefly consider the remaining powertype constructors $\mathtt{V}_\mathtt{H}$, $\mathtt{V}_\mathtt{S}$, $\mathtt{V}_\mathtt{P}$, here ranged over by G.

3.1 Operational semantics

We extend that given in Section 2.2.

Translation of types. For non-determinism combined with probability, one needs two schedulers: one to perform the non-deterministic choices and the other to perform the probabilistic choices. The first is chosen in an erratic way, but the second with uniform distribution. The translation as if we were working with a probabilistic powertype followed by a non-deterministic powertype, and to some extent this is the case. However, the translations of the may- and average-testing operators introduces a subtle interaction between the two.

$$\phi(G\sigma) = \mathtt{Cantor} \times \mathtt{Cantor} \to \phi(\sigma), \qquad \text{for } G \in \{\mathtt{V}_\mathtt{H}, \mathtt{V}_\mathtt{S}, \mathtt{V}_\mathtt{P}\}.$$

The translations of must- and may-average-testing then have type

$$(\phi(\sigma) \to \mathtt{I}) \to ((\mathtt{Cantor} \to \mathtt{Cantor} \to \phi(\sigma)) \to \mathtt{I}),$$

and are given as follows, where we emphasize that the infima, suprema and integrals are over the Cantor type:

$$\begin{aligned}
\phi(\Diamond_\mathtt{H}) &= \lambda f.\lambda\mathcal{C}.\sup_s\Big(\int f(\mathcal{C}(s,t))\,\mathrm{d}t\Big),\\
\phi(\Box_\mathtt{S}) &= \lambda f.\lambda\mathcal{Q}.\inf_s\Big(\int f(\mathcal{Q}(s,t))\,\mathrm{d}t\Big),\\
\phi(\Diamond_\mathtt{P}) &= \lambda f.\lambda\mathcal{L}.\sup_s\Big(\int f(\mathcal{L}(s,t))\,\mathrm{d}t\Big),\\
\phi(\Box_\mathtt{P}) &= \lambda f.\lambda\mathcal{L}.\inf_s\Big(\int f(\mathcal{L}(s,t))\,\mathrm{d}t\Big).
\end{aligned}$$

The translations of the non-deterministic choice operators consume tokens from the first scheduler and behave in the same way:

$$\begin{aligned}
\phi(\mathbin{\mathbb{V}}_{\mathtt{V_H}}) &= \lambda(\mathcal{C}_0, \mathcal{C}_1).\lambda(s,t).\,\text{if hd}(s) \text{ then } \mathcal{C}_0(\text{tl}(s), t) \text{ else } \mathcal{C}_1(\text{tl}(s), t),\\
\phi(\mathbin{\mathbb{V}}_{\mathtt{V_S}}) &= \lambda(\mathcal{Q}_0, \mathcal{Q}_1).\lambda(s,t).\,\text{if hd}(s) \text{ then } \mathcal{Q}_0(\text{tl}(s), t) \text{ else } \mathcal{Q}_1(\text{tl}(s), t),\\
\phi(\mathbin{\mathbb{V}}_{\mathtt{V_P}}) &= \lambda(\mathcal{L}_0, \mathcal{L}_1).\lambda(s,t).\,\text{if hd}(s) \text{ then } \mathcal{L}_0(\text{tl}(s), t) \text{ else } \mathcal{L}_1(\text{tl}(s), t).
\end{aligned}$$

Those of probabilistic choice operators consume tokens from the second scheduler and again behave in the same way:

$$\begin{aligned}
\phi(\oplus_{\mathtt{V_H}}) &= \lambda(\mathcal{C}_0, \mathcal{C}_1).\lambda(s,t).\,\text{if hd}(t) \text{ then } \mathcal{C}_0(s, \text{tl}(t)) \text{ else } \mathcal{C}_1(s, \text{tl}(t)),\\
\phi(\oplus_{\mathtt{V_S}}) &= \lambda(\mathcal{Q}_0, \mathcal{Q}_1).\lambda(s,t).\,\text{if hd}(t) \text{ then } \mathcal{Q}_0(s, \text{tl}(t)) \text{ else } \mathcal{Q}_1(s, \text{tl}(t)),\\
\phi(\oplus_{\mathtt{V_P}}) &= \lambda(\mathcal{L}_0, \mathcal{L}_1).\lambda(s,t).\,\text{if hd}(t) \text{ then } \mathcal{L}_0(s, \text{tl}(t)) \text{ else } \mathcal{L}_1(s, \text{tl}(t)).
\end{aligned}$$

Finally, we consider the translation of the monad structure:

$$\begin{aligned}
\phi(Gf) &= \lambda k.\lambda(s,t).f(k(s,t)),\\
\phi(\eta_G) &= \lambda x.\lambda(s,t).x,\\
\phi(\mu_G) &= \lambda K.\lambda(s,t).K(\mathtt{onepart}(s), \mathtt{onepart}(t))(\mathtt{theotherpart}(s), \mathtt{theotherpart}(t)).
\end{aligned}$$

3.2 Denotational semantics

If I undertand Tix's thesis and the monograph by Tix, Keimel and Plotkin correctly, the contructions there apply to Lawson compact continuous domains only, which would cause trouble in the presence of function spaces in much the same way as the probabilistic powerdomain does. Or have I missed something and one can define suitable constructions with the required universal properties in the category of continuous maps of dcpos with bottom? If the construction with the required properties can be performed in this category, then there is no problem in defining the semantics. Otherwise, we can only apply a type constructor G only once. From the point of view of the applications, this is not necessarily bad. Does adequacy, as formulated above, hold for this fragment?