Semi-decidability of may, must and probabilistic testing in a higher-type setting

Martín Escardó

School of Computer Science, University of Birmingham, UK

Abstract

We show that, in a fairly general setting including higher-types, may, must and probabilistic testing are semi-decidable. The case of must testing is perhaps surprising, as its mathematical definition involves universal quantification over the infinity of possible outcomes of a non-deterministic program. The other two involve existential quantification and integration. We also perform first steps towards the semi-decidability of similar tests under the simultaneous presence of non-deterministic and probabilistic choice.

Keywords: Non-deterministic and probabilistic computation, higher-type computability theory and exhaustible sets, may and must testing, operational and denotational semantics, powerdomains.

1 Introduction

We consider a non-deterministic higher-type language, in the style of PCF [38,32,17], which includes angelic, demonic and probabilistic choice. The types are closed under finite products and function spaces, and certain powertype constructors, interpreted as powerdomain monads, which capture various kinds of non-determinism. Choices can only be performed at powertypes, and the different powertypes have different operational interpretations of choice.

We show that (i) may, (ii) must and (iii) probabilistic testing are semi-decidable for this language. The idea is that, given a semi-decidable property u, one can semi-decide whether a given non-deterministic program (i) has some outcome satisfying u, (ii) has all outcomes satisfying u, and (iii) has all outcomes satisfying u with probability bigger than a given number. The proofs exploit recent results on exhaustible sets in higher-type computation [13,8], and older results on exact computability and definability of integrals in PCF-like languages [6,41,39].

Even at ground types, the claim for must testing may seem suspicious: for example, it implies that for any non-deterministic program of natural number type, possibly including subterms of arbitrarily high types, it is semi-decidable whether the outcome of the program must be a prime number. The semi-decision procedure

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

is expected to answer *yes* if all possible outcomes are prime numbers, and to diverge if the outcomes include composite numbers or divergent computations. Of course, it is also semi-decidable whether all outcomes are not prime, as primality is decidable and hence non-primality is semi-decidable. However, it doesn't follow that it would be decidable whether all outcomes are prime, and in fact this is not the case because some outcomes can be divergent computations.

From the point of view of computability theory, we have an extensional procedure that operates on programs, but that is not definable directly on the observable input-output behaviour of programs, and hence the Rice-Shapiro theorem [34] fails for non-deterministic programming systems. This leads us to extend the language with may, must and probabilistic testing primitives. It is interesting that these tests define basic open sets of the Scott topology of powerdomains that are not definable in the original language. The resulting language has an operational semantics and is regarded as an executable program logic for semi-decidable properties, which plays the role of a sort of "Rice-Shapiro completion" of the programming language (although we don't have at present a precise formulation of such a concept).

Our semi-decision procedures are defined using operational semantics, and their correctness is proved using domain-theoretic denotational semantics. Perhaps surprisingly, may testing is harder than must testing in a sense: inclusion of the former to the language leads to definability of parallel-convergence (even on programs of deterministic type), but the latter can be defined without parallel features. Probabilistic testing also requires parallel features.

Our results on may, must and probabilistic testing are very general, but have restrictions, discussed in the body of the paper, due to open problems in domain theory involving the probabilistic powerdomain [27].

We also perform first steps towards the semi-decidability of similar tests under the simultaneous presence of probabilistic choice and non-deterministic choice [29,46,47]. We develop semi-decision procedures for this, but their correctness is a conjecture and their scope is open (Section 7.4).

This work is related to Abramsky's work on logic of observable properties [1], going back to Smyth [44], but we take a different approach and also account for probabilistic computation. The relationship between the two approaches certainly deserves more scrutiny.

Organization. 2. A programming language for non-determinism and probability. 3. Logical types. 4. An executable program logic. 5. Operational semantics of the executable logic. 6. Denotational semantics of the executable logic. 7. Discussion and questions.

Acknowledgements. A preliminary version of this work, with more conjectures than results, was presented in 2005 at a workshop in McGill University Bellairs research institute, organized by Prakash Panangaden. I thank him and the other participants for discussions, in particular Vincent Danos, Achim Jung, Klaus Keimel, Jimmie Lawson, Gordon Plotkin, Steve Vickers. I also had useful discussions with Alex Simpson, Dan Ghica and Paul Levy, and detailed comments by one of the referees, which are addressed in this revised version.

2 A programming language for non-determinism and probability

We consider a language with a type system that makes an explicit distinction between deterministic and various kinds of non-deterministic types. Any term of deterministic type has only one outcome, including the possibility of divergence. A term of non-deterministic type has one or more runs, in general continuum many, each of which either produces a single outcome, again including the possibility of divergence, 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.

2.1 An extension of PCF

We consider an extension of the programming language PCF [32] so that PCF 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.

Types. The ground types, ranged over by γ, are those of PCF:

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

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 type constructors defined by

$$F ::= \mathtt{H} \mid \mathtt{S} \mid \mathtt{P} \mid \mathtt{V}.$$

The three constructors $\mathtt{H}, \mathtt{S}, \mathtt{P}$ (Hoare, Smyth and Plotkin powertypes) are for non-deterministic computation. They respectively allow may, must and both testings. The constructor $\mathtt{V}$ (probabilistic powertype) is for probabilistic computation.

Example 2.1 The type $\sigma \times \tau \to \mathtt{V}\,\tau$ can be used to code labeled Markov processes with label space $A = \sigma$, state space $S = \tau$ and transition function $t : A \times S \to \mathtt{V}\,S$.

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

Choice rules. The Hoare, Smyth and Plotkin powertypes have a binary choice operator $\mathbin{\mathbb{O}}$. The idea is that the runs of a term $M \mathbin{\mathbb{O}} N$ are those of the term M together with those of the term N. The probabilistic powertype has 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 $M \mathbin{\mathbb{O}} N$ is angelic or demonic, whereas the choice $M \oplus N$ is probabilistic, with equal probability for both branches.

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

$$(\diamondsuit^\sigma)\colon F\sigma \times F\sigma \to F\sigma,$$

written in infix notation.

Probabilistic choice rule. For each type σ, we have an infix constant

$$(\oplus^\sigma)\colon \mathtt{V}\,\sigma \times \mathtt{V}\,\sigma \to \mathtt{V}\,\sigma.$$

Monad rules. Let $F \in \{\mathtt{H}, \mathtt{S}, \mathtt{P}, \mathtt{V}\}$ be a unary type constructor.

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

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

This amounts to the fact that any deterministic function can be considered as a non-deterministic function. To compute $Ff\colon F\sigma \to F\tau$ at a given input $x\colon F\sigma$, we first compute an outcome of x and then feed it to f, which in turn gives one of the possible outcomes of Ffx.

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). This amounts to the fact that any deterministic computation can be regarded as a possibly non-deterministic computation which just happens to be able to produce precisely one outcome.

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

$$\mu^\sigma_F\colon FF\sigma \to F\sigma.$$

This amounts to the fact that a non-deterministic computation, each of whose possible outcomes is another non-deterministic computation of an element of σ, can be seen simply as a non-deterministic computation of an element of σ. To compute an outcome of the term μX, we first compute an outcome $x\colon F\sigma$ of $X\colon FF\sigma$, and then compute an outcome of x in σ.

Strength rule. For all types σ, we have a (for the moment nameless) constant 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.

Example 2.2 In this language, the terms $\eta(\lambda x.0) \diamondsuit \eta(\lambda x.1)$ and $\lambda x.\eta(0) \diamondsuit \eta(1)$ are distinguishable from their types $F(\sigma \to \mathtt{Nat})$ and $\sigma \to F\mathtt{Nat}$ respectively, for any type constructor $F \in \{\mathtt{H}, \mathtt{S}, \mathtt{P}\}$. The first defines, non-deterministically, a function, whereas the second defines a single function with non-deterministic output. In languages that omit the type distinction we are making, and hence omit the coercion

η, the terms $(\lambda x.0) \mathbin{\mathbb{V}} (\lambda x.1)$ and $\lambda x.(0 \mathbin{\mathbb{V}} 1)$ are indistinguishable under call by name, as discussed e.g. by Sieber [40]. Here the two terms are distinguishable in terms of their behaviour too. In the first case, once we see a zero in the output for some given input, we'll always see zeros afterwards, no matter what the input is, whereas in the second we'll be able to get zeros and ones even for the same input.

Example 2.3 The idea behind the above example can be illustrated in imperative style as follows. The term $\eta(\lambda x.0) \mathbin{\mathbb{V}} \eta(\lambda x.1)$ corresponds to the non-deterministic algorithm

```
    (repeat for ever (print 0))
 or
    (repeat for ever (print 1))
```

If a run of this algorithm prints 0 first, then it must print 0 the second time as well. The term $\lambda x.\eta(0) \mathbin{\mathbb{V}} \eta(1)$ corresponds to the algorithm

```
    repeat for ever ((print 0) or (print 1))
```

If this prints 0 first, it may print 1 the second time, and hence the two algorithms can be distinguished by may and must testing.

Example 2.4 Think of the elements of the type $\mathtt{Cantor} = (\mathtt{Nat} \to \mathtt{Bool})$ as sequences of booleans. Then $\mathrm{cons} \colon \mathtt{Bool} \to \mathtt{Cantor} \to \mathtt{Cantor}$ defined by

$$\mathrm{cons}\, p\, s = \lambda i.\, \mathrm{if}\ i == 0\ \mathrm{then}\ p\ \mathrm{else}\ s(i-1),$$

adds p as a first new element of the sequence s, shifting the original elements to the right. Define, using functoriality, a term $\mathrm{prefix} \colon \mathtt{Bool} \to \mathtt{V}\,\mathtt{Cantor} \to \mathtt{V}\,\mathtt{Cantor}$ by

$$\mathrm{prefix}\, p = \mathtt{V}(\mathrm{cons}\, p).$$

Then the following term $\mathrm{random} \colon \mathtt{V}\,\mathtt{Cantor}$ is intended to randomly choose a total element of $\mathtt{Cantor}$ with uniform distribution:

$$\mathrm{random} = (\mathrm{prefix}\,\mathrm{False}\,\mathrm{random}) \oplus (\mathrm{prefix}\,\mathrm{True}\,\mathrm{random}).$$

Example 2.5 For any type σ, recursively define a term

$$(s, v, w) \mapsto (v \oplus_s w) \colon \mathtt{Cantor} \times \mathtt{V}\,\sigma \times \mathtt{V}\,\sigma \to \mathtt{V}\,\sigma$$

by

$$(v \oplus_s w) = \mathrm{if}\ s(0)\ \mathrm{then}\ v \oplus (v \oplus_{\mathrm{tl}\, s} w)\ \mathrm{else}\ (v \oplus_{\mathrm{tl}\, s} w) \oplus w$$

where $\mathrm{tl}\, s = \lambda i.s(i+1)$ is the tail map. If $s \colon \mathtt{Cantor}$ is a term encoding the binary expansion of a real number $p \in [0,1]$, where False encodes zero and True encodes 1, then it is intended that $v \oplus_s w$ chooses v with probability p, and w with probability $1-p$.

More generally, using the same idea, it is possible to define an n-ary term weighted-choice that chooses among n branches with given probabilities $p_1, \ldots, p_n$ that add up to 1.

2.2 Operational semantics

We extend the big-step operational semantics of PCF with the following rules. Firstly, we stipulate that if $v\colon\sigma$ is a value (or weak head normal form) then so is $\eta(v)\colon F\sigma$. Then, omitting types and contexts, we add the rules:

$$\frac{M \Downarrow v}{M \mathbin{\textcircled{\scriptsize V}} N \Downarrow v} \qquad \frac{N \Downarrow v}{M \mathbin{\textcircled{\scriptsize V}} N \Downarrow v} \qquad \frac{M \Downarrow v}{M \oplus N \Downarrow v} \qquad \frac{N \Downarrow v}{M \oplus N \Downarrow v}$$

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

Thus, all choice operators have the same meaning under this semantics: this semantics only says what the possible outcomes of a term are, if any, and hence doesn't fully capture the intended meaning (cf. [21]). The following is easy to establish (see e.g. [40]):

Proposition 2.6 *There is a computable partial function*

$$M \Downarrow^s v,$$

with inputs M and s, and output v, where s ranges over the Cantor space of infinite binary sequences, such that

$M \Downarrow v$ 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 term term M, we can say that

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

Despite the universal quantification over an uncountable set, we can prove that must convergence is semi-decidable for closed terms. The reason is that the Cantor space is compact and computable functions are continuous [13,8]. However, we don't know how to define probabilistic testing and a more general must testing in the big-step style in an elegant and algorithmic way. Hence we instead translate our language into a deterministic language, using Proposition 2.6 above as the guiding idea, in Section 5. Moreover, rather than performing the tests externally to the language, we incorporate the tests into the language, obtaining an executable logic.

3 Logical types

The may, must and probabilistic testing operators of the executable program logic defined in Section 4 below will have values in the types `S` (Sierpinski space) and `I` (vertical unit interval), and in this section we extend PCF with such base types as a preparation for that section. The Sierpinski type `S` is for results of observations or semi-decisions, with an element $\top$ (observable true) and divergence (unobservable false), and hence is interpreted as the Sierpinski domain $\{\bot, \top\}$. The type `I` is

for observations of probabilities, and is interpreted as the set $[0,1] \subseteq \mathbb{R}$ under the natural order (hence zero is bottom and one is top).

3.1 The Sierpinski type

We have the following term formation rules for the Sierpinski type $\mathtt{S}$:

(i) $\top : \mathtt{S}$ is a term.

(ii) If $M : \mathtt{S}$ and $N : \sigma$ are terms then $(\text{if } M \text{ then } N) : \sigma$ is a term.

(iii) If $M, N : \mathtt{S}$ are terms then so is $M \vee N : \mathtt{S}$.

The only value (or canonical form) of type $\mathtt{S}$ is $\top$. Notice that there is no "else" clause in the above construction, and $(\vee)$ is intended to be parallel convergence (or weak parallel or). For future use, we define

$$p \wedge q = \text{if } p \text{ then } q.$$

The big-step operational semantics for these constructs is given by the following evaluation rules:

$$\frac{M \Downarrow \top \quad N \Downarrow V}{\text{if } M \text{ then } N \Downarrow V} \qquad \frac{M \Downarrow \top}{M \vee N \Downarrow \top} \qquad \frac{N \Downarrow \top}{M \vee N \Downarrow \top}.$$

Recall that computational adequacy of the Scott model of PCF amounts to the statement that:

If M is a closed term of ground type and v is a value then $[\![M]\!] = v$ iff $M \Downarrow v$.

Standard proofs of adequacy, e.g. Streicher [45], easily apply to this extension of PCF with the type $\mathtt{S}$.

3.2 The vertical unit-interval type

Because we are concerned with semi-decision procedures, our computations of terms M of type $\mathtt{I}$ are set-up so that, for any rational number $p \in [0,1]$, it is possible to semi-decide the condition $p < M$, uniformly in M and p, but not the conditions $M = p$ or $M < p$ in general. Hence the intended interpretation of the type $\mathtt{I}$ is the set $[0,1] \subseteq \mathbb{R}$ under its usual order. This is naturally regarded as a sub-dcpo of the unit-interval domain [11], by thinking of $x \in \mathtt{I}$ as the interval $[x,1]$. Thus, if M denotes x, then all we know about M from an operational (and constructive) point of view is the set of rational numbers p with $p < x$ (and so x is a lower real number from a constructive point of view).

We take the primitive operations for $\mathtt{I}$ as those of the interval type of Real PCF [11] (or the alternative version [7]), restricted to intervals of the form discussed above, with the same operational rules.

These primitive operations include simple unary arithmetic functions, a partial inequality test $p < (-)$ with p rational, and a parallel conditional. Because the value False doesn't arise in our semi-decisions, we replace the boolean type of terms of the form $p < M$ by the Sierpinski type $\mathtt{S}$. Similarly, we use the Sierpinski type for the parallel conditional, obtaining a *weak parallel conditional* $\mathrm{wif} : \mathtt{S} \times \mathtt{I} \times \mathtt{I} \to \mathtt{I}$ so that

$\mathrm{wif}\ \top\ \mathrm{then}\ x\ \mathrm{else}\ y = x$

$\mathrm{wif}\ \bot\ \mathrm{then}\ x\ \mathrm{else}\ y = x \sqcap y = \min(x, y)$.

We omit the discussion of computational adequacy for closed terms M of type $\mathtt{I}$, referring the interested reader to the references [11,7]. For our purposes, it is enough to know that this can be reduced to the computational adequacy of terms of type $\mathtt{S}$, by considering the term $(p < M) : \mathtt{S}$, as follows:

$[\![M]\!] = x$ iff for every rational number p, we have that $p < x \iff (p < M) \Downarrow \top$.

Basic definability results. Using the programming techniques developed in the references [11,15], in fact with essentially the same programs, it is easy to see that the average (or midpoint or mediation) operation, defined by

$$x \oplus y = (x + y)/2,$$

and the multiplication and binary minimum and maximum operations are definable in this language.

Quantification and integration over the Cantor space. For the operational semantics defined in Section 5 below, we need quantification and integration over the Cantor type

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

As in Example 2.4, we think of this as a type of sequences of booleans, where we are mostly concerned with total sequences, which will play the role of schedulers.

For the operational semantics of may and must testing we need two terms

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

For definability of the existential quantifier one needs parallel-convergence, but the universal quantifier is sequentially definable [13]:

$\exists(p) = p(\bot) \vee (\exists(\lambda s.p(\mathrm{cons\,False}\ s)) \vee \exists(\lambda s.p(\mathrm{cons\,True}\ s)))$,

$\forall(p) = p(\mathrm{if}\ \forall(\lambda s.p(\mathrm{cons\,False}\ s)) \wedge \forall(\lambda s.p(\mathrm{cons\,True}\ s))\ \mathrm{then}\ c)$,

where c is an arbitrary total term of type $(\mathtt{Nat} \to \mathtt{Bool})$, e.g. $\lambda i.\,\mathrm{True}$, and where cons is defined in Example 2.4.

For probabilistic testing, we need a term

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

where we take the uniform distribution on the total elements of $\mathtt{Cantor}$. This can be defined as

$$\int f = \max\left(f(\bot), \int \lambda s.f(\mathrm{cons\,False}\ s) \oplus \int \lambda s.f(\mathrm{cons\,True}\ s)\right).$$

The idea is the same as that applied for integration in Real PCF for the unit interval domain [6].

In Section 7.4, we also need supremum and infimum operators

$$\inf, \sup \colon (\mathtt{Cantor} \to \mathtt{I}) \to \mathtt{I},$$

which can be defined along the same lines [6].

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

$$\exists s.p[s] = \exists(\lambda s.p[s]), \quad \forall s.p[s] = \forall(\lambda s.p[s]), \quad \int f[s]\,\mathrm{d}s = \int(\lambda s.f[s]),$$

and likewise for sup and inf.

4 An executable program logic

None of the powertypes can be distinguished on the basis of the possible outcomes that their terms have (cf. [21]). The powertypes become observably different when one considers may, must and probabilistic testing, which describe whether choice is interpreted as angelic, demonic or probabilistic. Rather than having these tests external to the language, we extend the language with them, obtaining an executable program logic, which we refer to as MMP. 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 probabilistic testing constructs to the language.

The Hoare powertype admits only may testing, the Smyth powertype admits only must testing, and the Plotkin powertype admits both. And, of course, the probabilistic powertype admits probabilistic testing.

4.1 May and must testing

The $\mathtt{S}$-valued terms are characteristic functions of open sets and hence we define a type of opens

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

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 constants $\diamondsuit$ (may) and $\Box$ (must):

$$\diamondsuit^{\sigma}_{\mathtt{H}} : \mathcal{O}\,\sigma \to \mathcal{O}\,\mathtt{H}\,\sigma,$$
$$\Box^{\sigma}_{\mathtt{S}} : \mathcal{O}\,\sigma \to \mathcal{O}\,\mathtt{S}\,\sigma,$$
$$\diamondsuit^{\sigma}_{\mathtt{P}} : \mathcal{O}\,\sigma \to \mathcal{O}\,\mathtt{P}\,\sigma,$$
$$\Box^{\sigma}_{\mathtt{P}} : \mathcal{O}\,\sigma \to \mathcal{O}\,\mathtt{P}\,\sigma.$$

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, when we consider the (operational and denotational) semantics of the language. The idea for the Hoare and Smyth powertypes is the same, but only one kind of test is made available for each of them, as discussed above.

Example 4.1 Suppose one wants to semi-decide whether the outcome of a term $n\colon F\mathtt{Nat}$ must be prime. Then one first writes a semi-decision term $\mathtt{prime}\colon \mathtt{Nat} \to \mathtt{S}$ and then runs, in the executable logic, the ground term $\Box\,\mathtt{prime}\,n$ of type $\mathtt{S}$. Of course, considering a term $\mathtt{Nat} \to \mathtt{S}$ for semi-deciding non-primeness, one can semi-decide whether n must be non-prime. However, it doesn't follow that primeness of all outcomes of n is decidable: if n has at least one non-divergent run, then both must tests diverge. (Cf. the discussion in Section 7.3, which considers the possibility of replacing the Sierpinski space by the type of booleans.)

Example 4.2 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 $\mathtt{converge}\colon \mathtt{Nat} \to \mathtt{S}$ be a term such that $\mathtt{converge}(n) = \top$ iff $n \neq \bot$. Then we intend that

$\Diamond\,\mathtt{converge}(f(0)) = \top$ ("$f(0)$ may converge")

and that

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

but

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

Example 4.3 Parallel-convergence is definable from may testing. In fact, taking $\mathtt{converge}\colon \mathtt{S} \to \mathtt{S}$ as the identity function, the function $(\vee)\colon \mathtt{S} \times \mathtt{S} \to \mathtt{S}$ is characterized by the equation

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

However, it cannot be defined from must testing, because must testing can be defined without parallel features (as we do). Notice that $(p \wedge q) = \Box\,\mathtt{converge}(\eta(p) \mathbin{\textcircled{\scriptsize V}} \eta(q))$.

4.2 Probabilistic testing

This time, from an open set of σ we get an expectation 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) by composition with a coercion function $\mathtt{S} \to \mathtt{I}$ that maps $\bot$ to 0 and $\top$ to 1 (definable as $\lambda p.\,\mathrm{if}\,p\,\mathrm{then}\,1$). Hence expectations generalize open sets. For probabilistic testing, we include a constant

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

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

Example 4.4 Cf. Example 4.2. 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

$\bigcirc\,\mathtt{converge}(g(0)) = 1$ ("the probability that $g(0)$ converges is 1"),

and

$\bigcirc\,\mathtt{converge}_n(g(0)) = 2^{-n-1}$ ("the probability that $g(0)$ converges to n is 2^{-n-1}"),

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

Example 4.5 Parallel-convergence is definable from probabilistic testing (cf. Example 4.3 and the references [10,6,30]):

$$(p \vee q) = 0 < \bigcirc\,\mathtt{converge}(\eta(p) \oplus \eta(q)).$$

Example 4.6 Cf. Example 2.4. Define a term $\mathrm{prefix}\colon \mathtt{I} \to \mathtt{V}\,\mathtt{I} \to \mathtt{V}\,\mathtt{I}$ by

$$\mathrm{prefix}\, x = \mathtt{V}(\lambda y.x \oplus y),$$

where here $(\oplus)\colon \mathtt{I} \times \mathtt{I} \to \mathtt{I}$ is the average operation. Then the term $\mathrm{random}\colon \mathtt{V}\,\mathtt{I}$ defined below is intended to randomly choose a real number with uniform distribution:

$$\mathrm{random} = (\mathrm{prefix}\, 0\, \mathrm{random}) \oplus (\mathrm{prefix}\, 1\, \mathrm{random}).$$

Hence for example $\bigcirc(\lambda x.p < x)\,\mathrm{random} = 1 - p$ for any $p \in \mathtt{I}$. That is, the probability that a uniformly chosen random number x satisfies $p < x$ is $1 - p$. This is generalized to invariant measures of iterated function systems in Example 5.1 below.

5 Operational semantics of the executable logic

We define the operational semantics of the executable logic MMP by compositional compilation into its deterministic sub-language $\mathrm{PCF} + \mathtt{S} + \mathtt{I}$ introduced in Section 3, where the translation is the identity on this sub-language. The compilation map

$$\phi\colon \mathrm{MMP} \to \mathrm{PCF} + \mathtt{S} + \mathtt{I}$$

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 $\phi(\sigma)$.

The idea is to reduce may, must and probabilistic testing in MMP to quantification and integration in $\mathrm{PCF} + \mathtt{S} + \mathtt{I}$. In principle, as discussed in Section 5.1, the quantifications and integrations are over sets of possible outcomes. For the translation, we further reduce them to quantifications and uniform integrations over the Cantor space in Section 5.3, and Section 5.1 is intended as a motivation for this, which anticipates the denotational semantics given in Section 6 below.

5.1 Quantification and integration over sets of possible outcomes

Consider the may testing operator

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

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

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

That is,

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

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

$$\exists x \in C.u[x] = \exists(C)(\lambda x.u[x]).$$

With this notation, we have

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

Similarly, from the must testing operator $\Box\colon \mathcal{O}\,\sigma \to \mathcal{O}\,\mathtt{S}\,\sigma$, we get a term

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

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 probabilistic testing operator

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

we get a term

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

defined by

$$\int_\nu u = \bigcirc(u)(\nu).$$

where $\nu\colon \mathtt{V}\,\sigma$ and $u\colon \sigma \to \mathtt{I}$.

Example 5.1 Generalizing Example 4.6, if $(\sigma, f_1, \ldots, f_n, p_1, \ldots, p_n)$ is an iterated function system with probabilities [25,24], then its invariant measure $\nu\colon \mathtt{V}\,\sigma$ can be defined as

$$\nu = \text{weighted-choice}(p_1, \ldots, p_n)(\mathtt{V}(f_1)(\nu), \ldots, \mathtt{V}(f_n)(\nu)),$$

where weighted-choice is the term discussed in Example 2.5. Scriven developed a PCF program for computing integrals of functions $u\colon \sigma \to \mathtt{I}$ with respect to the invariant measure [39]. Here we get the alternative algorithm $\int_\nu u = \bigcirc(u)(\nu)$ in the program logic MMP instead. Notice that the underlying space can be a function space (as in Example 2.4).

5.2 Translation of types

This is defined by induction:

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

Recall from Section 3 that $\mathtt{Cantor}$ is the type $(\mathtt{Nat} \to \mathtt{Bool})$. As in Proposition 2.6, 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 false as left and of true 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.

5.3 Translation of terms

This is also defined by induction. The translation of a syntactic variable is a variable with the same name but its type modified appropriately for powertypes. In order to be precise here, one needs some more-or-less evident syntactic bureaucracy which can be safely omitted for our purposes. For PCF 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 that the translation is a congruence:

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

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

For $\star \in \{\mathbin{\mathbb{O}}, \oplus\}$, we define, where k_0 and k_1 range over $\phi(F\sigma) = \mathtt{Cantor} \to \phi(\sigma)$,

$$\phi(\star) = \lambda(k_0, k_1).\lambda s.\,\text{if } \mathrm{hd}(s) \text{ then } k_0(\mathrm{tl}(s)) \text{ else } k_1(\mathrm{tl}(s)).$$

Here hd and tl are the head and tail maps on sequences, defined by $\mathrm{hd}(s) = s(0)$ and $\mathrm{tl}(s)(i) = s(i+1)$. As discussed above, the idea is that the first element of the scheduler s dictates which branch is chosen, and the remainder of the scheduler then acts on the corresponding branch.

For the translation of the testing operators, recall that the type of may and must testing is

$$(\sigma \to \mathtt{S}) \to (F\sigma \to \mathtt{S}),$$

and hence their translations are to have type

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

Similarly, the translation of probabilistic testing is to have type

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

All quantifications and integrals in the following definitions are over the Cantor type (as introduced in Section 3.2), and we let the variable k range over the type $\phi(F\sigma) = \mathtt{Cantor} \to \phi(\sigma)$:

$$\phi(\diamondsuit) = \lambda u.\lambda k.\exists s.u(k(s)),$$
$$\phi(\Box) = \lambda u.\lambda k.\forall s.u(k(s)),$$
$$\phi(\bigcirc) = \lambda u.\lambda k.\int u(k(s))\,\mathrm{d}s.$$

For the functor and units of the monads, we define

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

For the multiplication, we consider PCF terms

$$\mathtt{evens}, \mathtt{odds} \colon \mathtt{Cantor} \to \mathtt{Cantor}$$

that take subsequences at even and odd indices, and define:

$$\phi(\mu_F) = \lambda k.\lambda s.k(\mathtt{evens}(s))(\mathtt{odds}(s)).$$

That is, we split the scheduler into two schedulers and pass each one to a different subcomputation. In the target language, the monad laws fail for the translations (both denotationally and operationally), but they will hold modulo the appropriate notion of testing. The strength is translated in a similar, mechanical, manner.

5.4 Semi-decision procedures for may, must and probabilistic testing

Given a semi-decidable property coded as a term $u \colon \sigma \to \mathtt{S}$ and a non-deterministic program $n \colon \mathtt{P}\,\sigma$, in order to semi-decide may and must testing we evaluate the terms

$$\phi(\Box(u)(n)), \qquad \phi(\diamondsuit(u)(n))$$

of ground type $\mathtt{S}$ in the deterministic language PCF $+ \mathtt{S} + \mathtt{I}$. Similarly, to compute the probability that outcomes of n satisfy u one evaluates the term

$$\phi(\bigcirc(c \circ u)(n))$$

of type $\mathtt{I}$ where $c \colon \mathtt{S} \to \mathtt{I}$ is the coercion defined by $c(p) = \text{if } p \text{ then } 1$. To semi-decide whether the probability that outcomes of n satisfy u is bigger than a definable real number $r \colon \mathtt{I}$, one evaluates the term

$$\phi(r < \bigcirc(c \circ u)(n)).$$

5.5 Ground evaluation

For MMP terms $M \colon \sigma$ with $\gamma \neq \mathtt{I}$ ground, it is convenient to define

$$M \Downarrow v \iff \phi(M) \Downarrow v.$$

6 Denotational semantics of the executable logic

In this section we apply domain theory to prove the correctness of the semi-decision procedures for may, must and probabilistic testing developed in Section 5.

We work in the category of dcpos and continuous maps to give the semantics, but eventually need to consider continuous dcpos to prove the correctness of the semi-decision procedures. As discussed in Section 3, for the sub-language PCF, we consider its Scott model, and we interpret the type $\mathtt{S}$ as the Sierpinski domain $\{\bot, \top\}$ and $\mathtt{I}$ as the unit interval $[0, 1] \subseteq \mathbb{R}$ with its natural order, getting an interpretation of the sub-language PCF+ $\mathtt{S}$ + $\mathtt{I}$ as in Section 3. We then interpret the powertypes as the Hoare, Smyth, Plotkin and probabilistic powerdomains [2,26].

6.1 Computational adequacy

To establish semi-decidability of may, must and probabilistic testing, we first prove *computational adequacy* of the model:

Lemma 6.1 *For any closed term M in the executable logic* MMP *of ground type other than $\mathtt{I}$, and all syntactical values v,*

$$[\![M]\!] = [\![v]\!] \iff M \Downarrow v.$$

In particular, this will imply, for $M \colon \mathtt{I}$ closed and $r \in \mathbb{Q}$, that

$$r < [\![M]\!] \iff r < M \Downarrow \top.$$

Because the model is already known to be computationally adequate for the deterministic sub-language PCF + $\mathtt{S}$ + $\mathtt{I}$, we have the following purely denotational formulation of computational adequacy for the full language MMP:

Lemma 6.2 *Computational adequacy holds if and only if $[\![M]\!] = [\![\phi(M)]\!]$ for every closed term M of ground type.*

To prove computational adequacy using this, we rely on the description of the powerdomains as free algebras for the (interpretations of) the choice operators $\mathbin{⩒}$ and $\oplus$. The axioms for the operations can be found in [26,35,2]. Then the semantics of the test operators $\diamondsuit, \Box, \bigcirc$ are uniquely determined by the conditions that $\diamondsuit(u)$, $\Box(u)$ and $\bigcirc(u)$ are algebra homomorphisms:

$$\begin{aligned}
\diamondsuit(u)(\eta(x)) &= u(x), \quad \diamondsuit(u)(X_0 \mathbin{⩒} X_1) = \diamondsuit(u)(X_0) \vee \diamondsuit(u)(X_1),\\
\Box(u)(\eta(x)) &= u(x), \quad \Box(u)(X_0 \mathbin{⩒} X_1) = \Box(u)(X_0) \wedge \Box(u)(X_1),\\
\bigcirc(u)(\eta(x)) &= u(x), \quad \bigcirc(u)(\nu_0 \oplus \nu_1) = \bigcirc(u)(\nu_0) \oplus \mathtt{V}(u)(\nu_1),
\end{aligned}$$

where we are using the fact that $\mathtt{S}$ and $\mathtt{I}$ are algebras when endowed with the operations$(\vee), (\wedge) \colon \mathtt{S} \times \mathtt{S} \to \mathtt{S}$ and $(\oplus) \colon \mathtt{I} \times \mathtt{I} \to \mathtt{I}$. This completes our proof sketch for computational adequacy.

6.2 Correctness proofs of the semi-decision procedures

An alternative, also well-known [26,35,2], definition of the powerdomains is in terms of non-empty subsets and of valuations. The elements of the Hoare power domain are the closed subsets, that of the Smyth powerdomain are the compact upper sets, that of the Plotkin powerdomain are the lenses, and that of the probabilistic powerdomain are the continuous valuations with total mass 1. The non-deterministic choice constants are interpreted as the union operation, and the probabilistic choice constant is interpreted as the convex-combination operation $(\nu_0, \nu_1) \mapsto \nu_0 \oplus \nu_1$. The logical constants are interpreted so that, as expected,

(i) $\diamondsuit(u)(X) = \top$ iff $u(x) = \top$ for some $x \in X$,

(ii) $\square(u)(X) = \top$ iff $u(x) = \top$ for all $x \in X$,

(iii) $\bigcirc(u)(\nu) = \int_\nu u$.

More usually, $\diamondsuit$ and $\square$ are seen as open-set constructors: The Scott topologies of the Hoare, Smyth, and Plotkin powerdomains have the following bases of open sets:

(i) Hoare: $\diamondsuit U$, where U ranges over open sets, and $X \in \diamondsuit U \iff U \cap X \neq \emptyset$.

(ii) Smyth: $\square U$, where U ranges over open sets, and $X \in \square U \iff X \subseteq U$.

(iii) Plotkin: Both $\diamondsuit U$ and $\square U$, but restricted to lenses.

Hence, writing $\mathcal{O}\, D$ for the lattice of open sets of a domain D, we have that $\diamondsuit$ and $\square$ are functions:

(i) $\diamondsuit\colon \mathcal{O}\, D \to \mathcal{O}\, \mathtt{H}\, D$.

(ii) $\square\colon \mathcal{O}\, D \to \mathcal{O}\, \mathtt{S}\, D$.

(iii) (a) $\diamondsuit\colon \mathcal{O}\, D \to \mathcal{O}\, \mathtt{P}\, D$.
(b) $\square\colon \mathcal{O}\, D \to \mathcal{O}\, \mathtt{P}\, D$.

Now $\mathcal{O}\, D \cong (D \to \mathtt{S})$ via characteristic functions, and hence $\diamondsuit$ and $\square$ can be seen as the functions discussed earlier. Moreover, it is clear that the condition $U \cap X \neq \emptyset$ amounts to existential quantification over X and the condition $X \subseteq U$ amounts to universal quantification. We have to show that these functions are continuous, but this is straightforward.

6.3 Obstacles

It may seem that the above observations, together with computational adequacy, would conclude the proof of semi-decidability of may, must and probabilistic testing. Unfortunately, this is not the case, because the abstract descriptions of the powerdomains given in Section 6.1 (in terms of universal properties) coincide with the concrete descriptions given in Section 6.2 (in terms of subsets and valuations) for restricted classes of domains only.

For the Hoare powerdomain, the abstract description given in Section 6.1 always agrees with the concrete one given in Section 6.2 (see [35]). But for the Smyth, Plotkin and probabilistic powerdomains, the two descriptions agree only for continuous domains [35,26,2]. However, not all continuous domains are closed under function spaces and the Plotkin powerdomain, as is also well known, and there is

no known cartesian closed category of continuous domains closed under the probabilistic powerdomain [27]. Hence we cannot expect all the types of our language to have continuous interpretations, unless further progress is made in domain theory.

Remark 6.3 These and similar issues led the authors of [4] to propose an alternative form of domain theory, called topological domain theory, that would overcome these and other kinds of obstacles in semantics. We plan to investigate its use to the resolution of the problems explained here, but, for the moment, we establish very general, albeit partial, results using classical domain theory.

6.4 Partial results

The following theorem summarizes the above discussion, where the adjectives to types refer to their domain interpretations defined above:

Theorem 6.4

(i) *For any type σ, may testing on terms of type $\mathtt{H}\,\sigma$ is semi-decidable.*

(ii) *For any continuous type σ, must testing on terms of type $\mathtt{S}\,\sigma$ is semi-decidable.*

(iii) *For any RSFP type σ, may and must testing on terms of type $\mathtt{P}\,\sigma$ are semi-decidable.*

(iv) *For any continuous type σ, probabilistic testing on terms of type $\mathtt{V}\,\sigma$ is semi-decidable.*

Remark 6.5 The smallest collection of types containing the ground types and closed under finite products, function spaces, and the Hoare, Smyth and Plotkin powertypes consists entirely of RSFP types [2]. Hence if we hadn't included the probabilistic powertype in our language, we wouldn't have had any of the above difficulties, and may and must testing would be semi-decidable for all types. What causes the restrictions is the presence of the probabilistic powertype.

But still the restrictions are not severe in practice: for example, probabilistic computations on any PCF type of any order have semi-decidable probabilistic testing. More generally, we can syntactically capture a large class of types for which the above theorem applies, as follows. Inductively define collections of types S, R, C as follows, where γ ranges over ground types:

$$S ::= \gamma \mid S \times S \mid (C \to S) \mid \mathtt{H}\,C \mid \mathtt{S}\,C,$$
$$R ::= S \mid R \times R \mid (R \to R) \mid \mathtt{P}\,R,$$
$$C ::= R \mid C \times C \mid \mathtt{V}\,C.$$

By a *continuous Scott domain* we mean a bounded complete continuous dcpo.

Proposition 6.6

(i) *The interpretation of an S type is a continuous Scott domain.*

(ii) *The interpretation of an R type is an RSFP domain.*

(iii) *The interpretation of a C type is a continuous dcpo.*

Proof The ground types are continuous Scott domains. Continuous Scott domains are RSFP domains, and RSFP domains are continuous domains [2]. Plotkin showed that the category RSFP is cartesian closed, and closed under the Plotkin powerdomain [2]. Scott showed that the continuous Scott domains are densely injective spaces, and the densely injective spaces are an exponential ideal [16]. Schalk showed that the continuous dcpos are closed under the Hoare and Smyth powerdomains, and it is immediate that they are bounded complete [35]. Jones showed that the continuous domains are closed under the probabilistic powerdomain [26]. □

7 Discussion and questions

Of course, the purpose of the algorithms developed here is to show that may, must and probabilistic testing are semi-decidable in principle, and not to attempt to provide usable algorithms for that purpose.

7.1 Programming with closed sets, compact sets and distributions

We have considered the full language as an executable logic for semi-decidable properties, with a sub-language singled out as a programming language for non-deterministic and probabilistic computation. From a different point of view, compatible with the denotational model, the full language can also be regarded as a deterministic programming language for computation with closed sets, compact sets, lenses, and probability distributions.

7.2 Turing universality

Can (a suitable extension of) our language define all computable elements of all types? For the case of the Smyth powertype, this is not unlikely because every compact upper set is a continuous image of the Cantor space, at least in the case of continuous Scott domains. What we need is to show that every computable element of the Smyth powerdomain is a computable image of the Cantor space. For the probabilistic powerdomain, one would have to show that every computable probabilistic continuous valuation is a computable image of the uniform distribution on the Cantor space, in the sense that for every $\nu \in \mathtt{V}\,\sigma$ there is a term $f\colon \mathtt{Cantor} \to \sigma$ such that $\nu = \mathtt{V}(f)(u)$ where $u\colon \mathtt{V}\,\mathtt{Cantor}$ is the uniform distribution (cf. Example 2.4). Of course, one first needs to define suitable effective presentations of the domains of interpretation of the language [42]. This in itself stumbles on the fact that it is not known whether, in the presence of the probabilistic powertype, all types are interpreted as continuous domains, as discussed in Section 6.3, and that the notion of effective presentation for domains is not well understood beyond the continuous case. Again, it may be worth looking at the ideas of the reference [4] (cf. Remark 6.3). Moreover, to achieve Turing universality, one has to take the discussion of Section 7.3 into account, and hence perhaps extend the language. Notice also that, denotationally, $\mathtt{I} \cong \mathtt{V}\,\mathtt{S}$. Is this isomorphism definable? The techniques discussed in [12,15] applied to establish Turing-universality of PCF extended with real numbers [11] may prove useful here.

7.3 Categorical universality revisited

For simplicity, in this discussion we restrict our attention to the RSFP types that exclude probabilistic powertypes. As discussed above, from the point of view of our denotational semantics, the may testing operator $\Diamond\colon \mathcal{O}\,\sigma \to \mathcal{O}\,\mathtt{H}\,\sigma$ constructs, from any given $u\colon \sigma \to \mathtt{S}$, the unique $\mathtt{H}$-algebra homomorphism $\bar{u}\colon \mathtt{H}\,\sigma \to \mathtt{S}$ extending u along $\eta\colon \sigma \to \mathtt{H}\,\sigma$, namely $\bar{u} = \Diamond\, u$. The structure map of an algebra is uniquely determined by its underlying domain, because $\mathtt{H}$ is a Köck-Zöberlein monad [28,48,14], and amounts to the non-deterministic choice operator $\mathbin{\mathbb{V}}$ for $\mathtt{H}\,\sigma$ and parallel convergence for $\mathtt{S}$. 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 [35].

A similar observation applies to the Smyth power domain, but things get more interesting. The monad is again of the Köck-Zöberlein type, and, additionally, all types are (interpreted as) underlying objects of algebras (with domain-theoretic meet as the structure map). Moreover, if one postulates generalized must testing terms $\Box\colon (\sigma \to \tau) \to (\mathtt{S}\,\sigma \to \tau)$, for τ ground, then one gets, by structural induction on types, generalized must testing programs for all PCF types τ, which articulates the universal property of the Smyth powerdomain construction within the language. In connection with the discussion of Section 5.1, 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 $Q \mapsto (f \mapsto \inf f(Q))$, cf. [35]. Applying this to the ground case $\tau = \mathtt{Bool}$, one gets a semi-decision procedure for must testing that answers True or False, and diverges if and only if some outcome of the given non-deterministic computation is divergent (cf. Example 4.1).

The Plotkin powerdomain is not of the Köck-Zöberlein type, and domains admit zero, one or more structure maps, and Hoare and Smyth structures are always Plotkin structures. Hence domains in the interpretation of PCF types have at least one Plotkin structure, namely the Smyth structure. For example, the may operator on the Plotkin powerdomain gives the universal property for the Hoare structure on $\mathtt{S}$, and the must operator for the Smyth structure.

7.4 Combination of probability with non-determinism

A number of authors have considered powerdomains that simultaneously account for probability and non-determinism [29,46,47]. Here we discuss first steps in this direction, closely following the ideas of [46]. A major difference is that this reference considers non-negative real-valued valuations rather than unit-interval valued valuations with total mass 1. To prove the correctness of the semi-decision procedures discussed here, we would need a theory based on the latter.

Recall that the Hoare powerdomain can be defined as the set of non-empty closed sets, and hence $\mathtt{H}\,\mathtt{V}\,D$ is the collection of closed sets of valuations. Tix, Keimel and Plotkin [46] consider a powerdomain $\mathtt{V}_{\mathtt{H}}\,D$ consisting of the geometrically convex, closed sets of valuations. Similarly, they consider powerdomains $\mathtt{V}_{\mathtt{S}}\,D$, consisting of geometrically convex, compact upper sets of valuations, and $\mathtt{V}_{\mathtt{P}}\,D$, consisting of geometrically convex lenses of valuations. Using our notations $\mathbin{\mathbb{V}}$ and $\oplus$ for non-deterministic and probabilistic choice, the equational theories for these

powerdomains are given by the following distributive law

$$x \oplus (y \mathbin{\textcircled{\vee}} z) = (x \oplus y) \mathbin{\textcircled{\vee}} (x \oplus z),$$

for $\mathtt{V_P}$, and where $\textcircled{\vee}$ has to additionally satisfy the semi-lattice equations plus:

(i) $x \mathbin{\textcircled{\vee}} y \sqsupseteq x$ for $\mathtt{V_H}$ ($\textcircled{\vee}$ is the binary join operation w.r.t. the information order),

(ii) $x \mathbin{\textcircled{\vee}} y \sqsubseteq x$ for $\mathtt{V_S}$ ($\textcircled{\vee}$ is the binary meet operation w.r.t. the information order).

Non-deterministic/probabilistic powertypes. We now consider the extension of the above programming language and logic with powertypes corresponding to these powerdomains, using the same notation for them, where now each of them has both constructs ($\textcircled{\vee}$) and ($\oplus$), as discussed above. Moreover, the functor, unit, multiplication, and strength rules are defined in the same way as for the other powertype constructors.

May-probabilistic and must-probabilistic rules. We need four constants

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

Continuing the discussion of Section 5.1, for the Hoare probabilistic powertype, from the may-probabilistic operator $\diamondsuit\colon\ \mathcal{E}\,\sigma \to \mathcal{E}\,\mathtt{V_H}\,\sigma$ we get a term of type $\mathtt{V_H}\,\sigma \to ((\sigma \to \mathtt{I}) \to \mathtt{I})$ written $\sup_{\nu\in\mathcal{C}} \int_\nu u = \diamondsuit(u)(\mathcal{C})$. The fictitious bound variable ν is included to make the notation suggestive of the denotational semantics discussed above. What we mean by this notation is that the application of the nameless term $\mathtt{V_H}\,\sigma \to ((\sigma \to \mathtt{I}) \to \mathtt{I})$ to a term $\mathcal{C}\colon\ \mathtt{V_H}\,\sigma$ followed by an application to a term $u\colon \sigma \to I$ is written $\sup_{\nu\in\mathcal{C}} \int_\nu u$. For the Smyth probabilistic powertype, we get a similar term, but we instead write $\inf_{\nu\in\mathcal{Q}} \int_\nu u$. For the Plotkin probabilistic powertype we get both.

Translation of types. For non-determinism combined with probability, we propose that two schedulers are needed: one to perform the non-deterministic choices and the other to perform the probabilistic choices. The first one is chosen in an angelic or demonic way, but the second one with uniform distribution. We thus extend the definition of the translation ϕ defined in Section 5 by stipulating that

$$\phi(G\sigma) = \mathtt{Cantor} \times \mathtt{Cantor} \to \phi(\sigma), \qquad \text{for } G \in \{\mathtt{V_H}, \mathtt{V_S}, \mathtt{V_P}\}.$$

Operational semantics. The translations of the non-deterministic choice operators consume tokens from the first scheduler:

$$\phi(\textcircled{\vee}) = \lambda(k_0, k_1).\lambda(s,t).\,\text{if } \mathrm{hd}(s) \text{ then } k_0(\mathrm{tl}(s), t) \text{ else } k_1(\mathrm{tl}(s), t).$$

Those of probabilistic choice operators consume tokens from the second scheduler:

$$\phi(\oplus) = \lambda(k_0, k_1).\lambda(s,t).\,\text{if } \mathrm{hd}(t) \text{ then } k_0(s, \mathrm{tl}(t)) \text{ else } k_1(s, \mathrm{tl}(t)).$$

The translations of may and must probabilistic testing have type

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

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

$$\phi(\diamondsuit) = \lambda u.\lambda k.\sup_s(\int u(k(s,t))\,\mathrm{d}t),$$
$$\phi(\Box) = \lambda u.\lambda k.\inf_s(\int u(k(s,t))\,\mathrm{d}t).$$

For the monad structure, we define:

$$\phi(Gf) = \lambda k.\lambda(s,t).f(k(s,t)),$$
$$\phi(\eta_G) = \lambda x.\lambda(s,t).x,$$
$$\phi(\mu_G) = \lambda k.\lambda(s,t).k(\mathtt{evens}(s),\mathtt{evens}(t))(\mathtt{odds}(s),\mathtt{odds}(t)).$$

For the technical reasons discussed above, we leave the correctness of this proposed translation open.

References

[1] S. Abramsky. Domain theory in logical form. *Ann. Pure Appl. Logic*, 51(1-2):1–77, 1991.

[2] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Oxford science publications, 1994.

[3] T. Anberrée. On the non-sequential nature of domain models of real-number computation. *Elec. Notes in Theoret. Comp. Sc.*, 173:41–46, 2007.

[4] I. Battenfeld, M. Schröder, and A. Simpson. A convenient category of domains. *Electron. Notes Theor. Comput. Sci.*, 172:69–99, 2007.

[5] P. Di-Gianantonio. *A Functional Approach to Computability on Real Numbers*. PhD thesis, Università Degli Studi di Pisa, Dipartamento di Informatica, 1993.

[6] A. Edalat and M.H. Escardó. Integration in Real PCF. *Inform. and Comput.*, 160:128–166, 2000.

[7] M. Escardó. Mathematical foundations of functional programming with real numbers. Course notes for a course delivered at the Midlands Graduate School in the Foundations of Computer Science, Leicester, http://www.cs.bham.ac.uk/~mhe/papers/mgs.pdf, April 2003.

[8] M. Escardó. Exhaustible sets in higher-type computation. *Log. Methods Comput. Sci.*, 4(3):3:3, 37, 2008.

[9] M. Escardó and W. K. Ho. Operational domain theory and topology of sequential programming languages. *Information and Computation*, 207:411–437, 2009.

[10] M. Escardó, M. Hofmann, and T. Streicher. On the non-sequential nature of the interval-domain model of real-number computation. *Math. Structures Comput. Sci.*, 14(6):803–814, 2004.

[11] M.H. Escardó. PCF extended with real numbers. *Theoret. Comput. Sci.*, 162(1):79–115, 1996.

[12] M.H. Escardó. Real PCF extended with ∃ is universal. In A. Edalat, S. Jourdan, and G. McCusker, editors, *Advances in Theory and Formal Methods of Computing: Proceedings of the Third Imperial College Workshop, April 1996*, pages 13–24, Christ Church, Oxford, 1996. IC Press.

[13] M.H. Escardó. Synthetic topology of data types and classical spaces. *Electron. Notes Theor. Comput. Sci.*, 87:21–156, 2004.

[14] M.H. Escardó and R.C. Flagg. Semantic domains, injective spaces and monads. *Electron. Notes Theor. Comput. Sci.*, 20, 1999.

[15] M.H. Escardó and Th. Streicher. Induction and recursion on the partial real line with applications to Real PCF. *Theoret. Comput. Sci.*, 210(1):121–157, 1999.

[16] G. Gierz, K.H. Hofmann, K. Keimel, J.D. Lawson, M. Mislove, and D.S. Scott. *Continuous Lattices and Domains*. Cambridge University Press, 2003.

[17] C.A. Gunter. *Semantics of Programming Languages—Structures and Techniques*. The MIT Press, 1992.

[18] R. Heckmann. An upper power domain construction in terms of strongly compact sets. *Lecture Notes in Comput. Sci.*, 598:272–293, 1992.

[19] R. Heckmann. Power domains and second order predicates. *Theoret. Comput. Sci.*, 111:59–88, 1993.

[20] R. Heckmann. Spaces of valuations. In *Papers on general topology and applications (Gorham, ME, 1995)*, volume 806 of *Ann. New York Acad. Sci.*, pages 174–200. New York Acad. Sci., New York, 1996.

[21] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. *Lecture Notes in Comput. Sci.*, 85:299–309, 1980.

[22] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *J. Assoc. Comput. Mach.*, 32(1):137–161, 1985.

[23] M. C. B. Hennessy and E. A. Ashcroft. A mathematical semantics for a nondeterministic typed λ-calculus. *Theoret. Comput. Sci.*, 11(3):227–245, 1980.

[24] S.G. Hoggar. *Mathematics for Computer Graphics*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[25] J.E. Hutchinson. Fractals and self-similarity. *Indiana University Mathematics Journal*, 30:713–747, 1981.

[26] C. Jones. *Probabilistic Non-determinism*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, January 1990. .

[27] A. Jung and R. Tix. The troublesome probabilistic powerdomain. *Electron. Notes Theor. Comput. Sci.*, 1997.

[28] A. Kock. Monads for which structures are adjoint to units (version 3). *Journal of Pure and Applied Algebra*, 104:41–59, 1995.

[29] A. K. McIver and C. Morgan. Partial correctness for probabilistic demonic programs. *Theoret. Comput. Sci.*, 266(1-2):513–541, 2001.

[30] D. Normann. Exact real number computations relative to hereditarily total functionals. *Theoret. Comput. Sci.*, 284(2):437–453, 2002.

[31] C.-H. L. Ong. Non-determinism in a functional setting (extended abstract. In *In Proceedings 8th LICS*, pages 275–286, 1993.

[32] G.D. Plotkin. LCF considered as a programming language. *Theoret. Comput. Sci.*, 5(1):223–255, 1977.

[33] P.J. Potts, A. Edalat, and M.H. Escardó. Semantics of exact real arithmetic. In *Proceedings of the Twelveth Annual IEEE Symposium on Logic In Computer Science*, Warsaw, Polland, Jun 1997.

[34] H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1967.

[35] A. Schalk. *Algebras for Generalized Power Constructions*. PhD thesis, Technische Hochschule Darmstadt, July 1993. ftp://ftp.cl.cam.ac.uk/papers/as213/diss.dvi.gz.

[36] M. Schröder and A. Simpson. Probabilistic observations and valuations (extended abstract). *Electron. Notes Theor. Comput. Sci.*, 155:605–615, 2006.

[37] M. Schröder and A. Simpson. Representing probability measures using probabilistic processes. *J. Complexity*, 22(6):768–782, 2006.

[38] D.S. Scott. A type-theoretical alternative to CUCH, ISWIM and OWHY. *Theoret. Comput. Sci.*, 121:411–440, 1993. Reprint of a 1969 manuscript.

[39] A. Scriven. A functional algorithm for exact real integration with invariant measures. *Electron. Notes Theor. Comput. Sci.*, 218:337–353, 2008.

[40] K. Sieber. Call-by-value and nondeterminism. *Lecture Notes in Comput. Sci.*, 664:376–390, 1993.

[41] A. Simpson. Lazy functional algorithms for exact real functionals. *Lec. Not. Comput. Sci.*, 1450:323–342, 1998.

[42] M.B. Smyth. Effectively given domains. *Theoret. Comput. Sci.*, 5(1):256–274, 1977.

[43] M.B. Smyth. Power domains. *Journal of Computer and Systems Sciences*, 16:23–36, 1977.

[44] M.B. Smyth. Power domains and predicate transformers: a topological view. *Lec. Not. Comput. Sci.*, 154:662–675, 1983.

[45] T. Streicher. *Domain-theoretic Foundations of Functional Programming*. World Scientific, 2006.

[46] R. Tix, K. Keimel, and G. Plotkin. Semantic domains for combining probability and non-determinism. *Electron Notes in Theor. Comput. Sci.*, 129, 2005.

[47] D. Varacca and G. Winskel. Distributing probability over non-determinism. *Math. Structures Comput. Sci.*, 16(1):87–113, 2006.

[48] S. Vickers. Locales are not pointless. In C. Hankin, I. Mackie, and R. Nagarajan, editors, *Theory and Formal Methods 1994: Proceedings of the Second Imperial College Department of Computing Workshop on Theory and Formal Methods*, Møller Centre, Cambridge, 11–14 September 1994. IC Press. 1995.