

Chapter 1

UNIVERSAL TYPES AND WHAT THEY ARE GOOD FOR

John R. Longley
School of Informatics
University of Edinburgh
King's Buildings, Edinburgh EH9 3JZ, U.K.
jrl@informatics.ed.ac.uk

Abstract We discuss the standard notions of *universal object* and *universal type*, and illustrate the usefulness of these concepts via several examples from denotational semantics. The purpose of the paper is to provide a gentle introduction to these notions, and to advocate a particular point of view which makes significant use of them. The main ideas here are not new, though our expository slant is somewhat novel, and some of our examples lead to seemingly new results.

Keywords: category, retract, Karoubi envelope, λ -algebra, universal object, universal type, denotational semantics, extensions of PCF

1. Universal objects

The purpose of this article is to revisit some standard concepts from category theory and denotational semantics: the notions of *universal object* and *universal type*. These ideas are quite simple and are already widely known, but in this article we hope to achieve three things. Firstly, we will try to give a gentle introduction to these ideas for the reader unfamiliar with them. Secondly, although the above concepts themselves are well known, it is perhaps not so widely appreciated how powerful they are — here we wish to advertise these ideas, and to argue that we should perhaps make greater use of them. Finally, we will present some specific applications of the ideas in denotational semantics which may be less familiar, leading to some interesting new results.

Our starting point is the following definition.

Definition 1.1 (Retract). In any category \mathcal{C} , an object X is a *retract* of an object Y ($X \triangleleft Y$) if there are morphisms $f : X \rightarrow Y$, $g : Y \rightarrow X$ such that $g \circ f = id_X$. We then say the pair (f, g) is a *retraction*.

A helpful intuition is that if X is a retract of Y then the object X can be “coded up” or represented by the object Y . For instance, in concrete categories, we can think of f as coding up elements of X as elements of Y , with g as a corresponding decoding function. More generally, $X \triangleleft Y$ means that morphisms into X can be coded as morphisms into Y , and likewise morphisms out of X can be coded as morphisms out of Y . Indeed, we have the following easy proposition, which perhaps helps to motivate the definition of a retraction:

Proposition 1.1. *X is a retract of Y iff for any objects A, B there are mappings*

$$\begin{aligned} a &\mapsto \bar{a} : Hom(A, X) \rightarrow Hom(A, Y), \\ b &\mapsto \tilde{b} : Hom(X, B) \rightarrow Hom(Y, B) \end{aligned}$$

such that for all a, b we have $\tilde{b} \circ \bar{a} = b \circ a$.

The central notion that we shall consider in this paper is the following:

Definition 1.2 (Universal object). In a category \mathcal{C} , an object U is *universal* if every object of \mathcal{C} is a retract of U .

If a category \mathcal{C} has a universal object U , the whole of \mathcal{C} can in some sense be coded up in terms of U and its endomorphisms. Firstly, given any object X and a retraction $(f, g) : X \triangleleft U$, the object X is clearly determined up to isomorphism by the composite $f \circ g : U \rightarrow U$. (Notice that this morphism is always an *idempotent*—that is, $(f \circ g)^2 = f \circ g$.) Secondly, given any morphism $h : X \rightarrow Y$ and retractions $(f_X, g_X) : X \triangleleft U$ and $(f_Y, g_Y) : Y \triangleleft U$, the morphism h is determined by the composite $f_Y \circ h \circ g_X : U \rightarrow U$.

Indeed, the whole of \mathcal{C} can essentially be reconstituted from just the monoid M of endomorphisms of U , using the following standard construction. The idea is that the endomorphisms used above to represent objects and morphisms of \mathcal{C} can themselves be taken to be the objects and morphisms of a certain category:

Definition 1.3 (Karoubi envelope). For any monoid (\mathcal{M}, \cdot, id) , we may define a category $\mathcal{K}(\mathcal{M})$ (called the *Karoubi envelope* of \mathcal{M}) as follows:

- objects are elements $a \in \mathcal{M}$ such that $a \cdot a = a$;

- morphisms $a \rightarrow b$ are elements $m \in \mathcal{M}$ such that $b.m.a = m$;
- the identity id_a is simply a ;
- given $m : a \rightarrow b$ and $n : b \rightarrow c$, their composition $n \circ m$ is $n.b.m$.

One can more generally define the Karoubi envelope $\mathcal{K}(\mathcal{D})$ for any category \mathcal{D} , but we shall concentrate here on the case of monoids (which can be regarded as one-object categories). Clearly $id \in \mathcal{M}$ is always a universal object of $\mathcal{K}(\mathcal{M})$, and the endomorphisms $id \rightarrow id$ are all the elements of \mathcal{M} . Moreover, the following fact is easily verified:

Proposition 1.2. *Let \mathcal{C} be a category with a universal object U , and let \mathcal{M} be the monoid of endomorphisms of U . Then there is a full and faithful functor $E : \mathcal{C} \rightarrow \mathcal{K}(\mathcal{M})$.*

Thus, the original category \mathcal{C} is equivalent to a full subcategory of the reconstituted category $\mathcal{K}(\mathcal{M})$. Note that $\mathcal{K}(\mathcal{M})$ may be bigger than \mathcal{C} , since there may be idempotents $a : U \rightarrow U$ which do not arise from retractions in \mathcal{C} , but the important thing for our purposes is that we have at least reconstructed all of \mathcal{C} — we will not bother too much in this article about the difference between \mathcal{C} and $\mathcal{K}(\mathcal{M})$. (A common situation in denotational semantics is that one has a bunch of categories with some full subcategory in common, and the objects we are really interested in — those corresponding to the types of some programming language — all live in this common subcategory, so we do not care too much about the precise extent of the categories.)

The point of view we want to advocate in this paper is that if \mathcal{C} has a universal object U , and we are able to get a good grasp on the monoid of endomorphisms of U , then we have somehow “cracked” the whole structure of \mathcal{C} , since in some sense all the complexity of \mathcal{C} is already present in this monoid. This idea has two main aspects. Firstly, if the relevant monoid \mathcal{M} is itself easy to construct, the definition of $\mathcal{K}(\mathcal{M})$ can offer a simple way of *constructing* the category of interest in the first place—which in at least some cases can be simpler than a more concrete construction of the objects and morphisms of \mathcal{C} . Secondly, whether or not we choose to construct our category in this way, once we know it has a universal object, we can often use this to prove *properties* of the category, and in particular to obtain results of interest for denotational semantics.

Of course, this strategy of trying to understand categories of interest via universal objects is not always applicable: the category in question might not have a universal object, or even if it does, its monoid of endomorphisms might not be easy to understand. (We will see examples

of both these situations later on.) The point we wish to emphasize, however, is that if there happens to be a readily intelligible universal object, it is certainly worth knowing about it. As we shall see, it turns out that an approach which emphasizes universal objects works well in a large number of interesting cases, and indeed offers a pleasantly “uniform” way of treating a wide variety of situations.

2. λ -algebras

It is natural to ask how particular properties of monoids \mathcal{M} are correlated with properties of the categories $\mathcal{K}(\mathcal{M})$. Here we will consider just one example of such a correlation, namely, the conditions under which $\mathcal{K}(\mathcal{M})$ is a cartesian closed category. The material in this section is fairly standard, and is due mainly to Scott [Sc80] and Koymans [K082]; further details may be found in Barendregt’s book [Ba84].

We will assume some basic familiarity with the untyped λ -calculus. To fix notation, for any set A we may consider a set $C(A) = \{c_a \mid a \in A\}$ of *constant symbols*, one for each element of A . We will write $\Lambda^0(A)$ for the set of closed λ -terms over $C(A)$ (that is, untyped λ -terms with no free variables, possibly involving constants drawn from $C(A)$). We use M, N, \dots to range over λ -terms, and write $=_\beta$ for the relation of β -equivalence.

The following definition gives one possible notion of a “model” for the untyped λ -calculus:

Definition 2.1 (λ -algebra). A λ -algebra is a set A equipped with an “interpretation” function $\llbracket - \rrbracket : \Lambda^0(A) \rightarrow A$ with the following properties:

$$\begin{aligned} \llbracket c_a \rrbracket &= a, \\ \text{if } M =_\beta N &\text{ then } \llbracket M \rrbracket = \llbracket N \rrbracket. \end{aligned}$$

Alternatively (and equivalently), one can define the notion of λ -algebra by means of an interpretation of *open* λ -terms over $C(A)$ (see e.g. [Ba84, Chapter 5]), but this entails a little extra machinery which the above definition avoids. Note that any λ -algebra A can be endowed with a binary operation $\cdot : A \times A \rightarrow A$ (called *application*) defined by

$$a \cdot b = \llbracket c_a c_b \rrbracket.$$

As an aside, note that every $M \in \Lambda^0(A)$ is β -equivalent to some term built up from constants c_a and the terms $K = \lambda xy.x$, $S = \lambda xyz.xz(yz)$ using just application. It follows that the function $\llbracket - \rrbracket$ is completely determined by the above conditions once we have fixed on an application operation and a suitable choice of elements $k = \llbracket K \rrbracket$ and $s = \llbracket S \rrbracket$.

We will give several examples of λ -algebras in Section 6. In practice, the concrete construction of particular λ -algebras usually conforms to the following general pattern. First, for each finite set F of variables we define a set A_F , elements of which will serve as denotations for λ -terms with free variables drawn from F ; we will take A to be A_\emptyset . (We may regard elements of A_F as “operations” of type $A^F \rightarrow A$, where A^F is the set of all functions from F to A — that is, the set of all valuations for the variables in F . The operations here may be functions of type $A^F \rightarrow A$, or they may be algorithms of some intensional kind.) Next we define an interpretation $\llbracket - \rrbracket$ on *open* λ -terms by induction on the term structure. It is then usually straightforward to verify that if $M =_\beta N$ then $\llbracket M \rrbracket = \llbracket N \rrbracket$.

Let us now see how λ -algebras are related to universal objects. Let \mathcal{C} be a cartesian closed category containing an object U such that $U^U \triangleleft U$. (Such a U is called a *reflexive object*. In a cartesian closed category, any universal object is reflexive.) Then U gives rise to a λ -algebra in the following way. First choose some retraction $U^U \triangleleft U$; call its components $lambda : U^U \rightarrow U$ and $apply : U \rightarrow U^U$. Next, let A be the set of all morphisms $1 \rightarrow U$ in \mathcal{C} , and let $\cdot : A \times A \rightarrow A$ be the function induced by $uncurry(apply) : U \times U \rightarrow U$. Finally, for any term M with free variables drawn from x_1, \dots, x_n , we may define a morphism $\llbracket M \rrbracket_\Gamma : U^n \rightarrow U$ as follows:

$$\begin{aligned} \llbracket c_a \rrbracket_\Gamma &= a \circ !_{U^n} \\ \llbracket x_i \rrbracket_\Gamma &= \pi_i \\ \llbracket MN \rrbracket_\Gamma &= uncurry(apply) \circ \langle M_\Gamma, N_\Gamma \rangle \\ \llbracket \lambda x.M \rrbracket_\Gamma &= \lambda \circ curry(\llbracket M \rrbracket_\Gamma, x) \end{aligned}$$

(Here, Γ abbreviates x_1, \dots, x_n ; $!_X$ is the unique morphism $X \rightarrow 1$; π_i is the i th projection $X_1 \times \dots \times X_n \rightarrow X_i$; and $curry(f) : X_1 \times \dots \times X_i \rightarrow Y^{X_{i+1}}$ is the exponential transpose of $f : X_1 \times \dots \times X_{i+1} \rightarrow Y$.) Restricting our attention to closed terms, this yields a function $\llbracket - \rrbracket : \Lambda^0(A) \rightarrow A$.

Proposition 2.1. *The structure $(A, \cdot, \llbracket - \rrbracket)$ obtained in this way is a λ -algebra.*

Proof. The conditions for constants and application are immediate. For the condition regarding β -equivalence, clearly it suffices to check that

$$\llbracket (\lambda x.M)N \rrbracket_\Gamma = \llbracket M[N/x] \rrbracket_\Gamma$$

for all suitably typed M, N . But this follows easily from the fact that $apply \circ lambda = id_{U^U}$ (together with the evident connection between syntactic substitution and categorical composition). \square

Conversely, given any λ -algebra $(A, \cdot, \llbracket - \rrbracket)$, we may build a category in which it lives as a reflexive object. Let $(\mathcal{M}_A, \cdot, id)$ be the monoid defined as follows:

- $\mathcal{M}_A = \{a \in A \mid a = \llbracket \lambda x.c_a x \rrbracket\} = \{\llbracket \lambda x.c_a x \rrbracket \mid a \in A\}$,
- $b.a = \llbracket \lambda x.c_b(c_a x) \rrbracket$,
- $id = \llbracket \lambda x.x \rrbracket$.

It is easy to check that this is indeed a monoid. Moreover:

Proposition 2.2. *For any λ -algebra A , $\mathcal{K}(\mathcal{M}_A)$ is cartesian closed. If $U = id$, considered as the canonical universal object of $\mathcal{K}(\mathcal{M})$, then U^U is the object given by $\llbracket \lambda xy.xy \rrbracket \in \mathcal{M}$. Finally, the λ -algebra arising from the canonical retraction $U^U \triangleleft U$ is isomorphic to A .*

Proof. For a terminal object in $\mathcal{K}(\mathcal{M})$ we may take $\llbracket \lambda x.e \rrbracket$ for any $e \in A$. Given idempotents $a, b \in \mathcal{M}$, the product $a \times b$ is the idempotent

$$\llbracket \lambda x.pair(c_a(fst x))(c_b(snd x)) \rrbracket,$$

where $pair = \lambda uvz.zuv$, $fst = \lambda uv.u$ and $snd = \lambda uv.v$; and the exponential b^a is $\llbracket \lambda xy.c_b(x(c_a y)) \rrbracket$. As a special case of the latter, we have $U^U = \llbracket \lambda xy.xy \rrbracket$. Finally, morphisms $1 \rightarrow U$ in $\mathcal{K}(\mathcal{M})$ are exactly elements of \mathcal{M} of the form $\llbracket \lambda xy.c_a y \rrbracket$ for $a \in A$; clearly these correspond precisely to elements of A itself. It is routine to check that the λ -algebra structure arising from U coincides with the original one on A . \square

It is also easy to show that if \mathcal{M} is any monoid such that $\mathcal{K}(\mathcal{M})$ is cartesian closed, the resulting λ -algebra gives rise by the above construction to a monoid isomorphic to \mathcal{M} . The above results may therefore be summarized as follows:

Theorem 2.3. *Let \mathcal{M} be any monoid. Then $\mathcal{K}(\mathcal{M})$ is cartesian closed iff $\mathcal{M} \cong \mathcal{M}_A$ for some λ -algebra A .*

One might ask whether it is possible to state the condition for $\mathcal{K}(\mathcal{M})$ being cartesian closed purely algebraically in terms of the monoid structure of \mathcal{M} . This can indeed be done, but the algebraic conditions are rather more complicated than one might hope for. The details are not especially illuminating and we will not give them here.

The above theorem is just one instance of a correlation between a property of \mathcal{M} and a property of $\mathcal{K}(\mathcal{M})$, and will serve in this paper to illustrate our viewpoint. For the record, we mention a couple of other similar correlations.

First, there is a “linear” analogue of the above correlation: $\mathcal{K}(\mathcal{M})$ is a symmetric monoidal closed category iff \mathcal{M} arises from a linear λ -algebra. (In the linear λ -calculus, we may form the abstraction $\lambda x.M$ only if M contains exactly one free occurrence of x . In fact, we need to work here with a version of the linear λ -calculus which also includes a *pairing* operation $\langle M, N \rangle$ and a *pattern-matching* construct $\lambda \langle x, y \rangle.M$.)

Secondly, if \mathcal{M} arises from a CPO-enriched λ -algebra A (with least element) then $\mathcal{K}(\mathcal{M})$ has initial solutions to many recursive domain equations: essentially, one can solve such an equation by taking the fixed point of a continuous operator on A . This idea was first exploited by Scott in the case of the λ -algebra $\mathcal{P}\omega$ [Sc76]. Correlations of this kind are very useful if we wish to approach the study of particular categories via their universal objects.

3. Denotational semantics

Next we recall some basic concepts from denotational semantics, and explain how universal types may be used to advantage in this setting.

The general situation we consider will consist of the following data. We think of items 1 and 2 below as constituting a programming language, and 3–5 as constituting a denotational model for such a language. The framework given here may look rather cumbersome, but it allows us to state our results in a very general form.

1 A typed language \mathcal{L} , consisting of

- a set of types σ ,
- an infinite supply of variables $x : \sigma$ for each type σ ,
- a set of terms M , such that all variables are terms,
- a *typing relation* $\Gamma \vdash M : \sigma$, where Γ ranges over *environments* $(x_1 : \sigma_1, \dots, x_n : \sigma_n)$, such that $\Gamma \vdash x : \sigma$ whenever $x : \sigma \in \Gamma$ (we will write $\mathcal{L}_\Gamma^\sigma$ for the set of terms M such that $\Gamma \vdash M : \sigma$),
- an operation of type-respecting *substitution*, yielding for any $\Delta \vdash M : \tau$ (where $\Delta = (x_1 : \sigma_1, \dots, x_n : \sigma_n)$) and $\Gamma \vdash N_i : \sigma_i$ (for $i = 1, \dots, n$) a term $\Gamma \vdash M[\vec{N}_i/\vec{x}_i] : \tau$.
- for each Γ, σ and Γ', σ' , a set of functions $C(-)$ (which we call *contexts*) mapping terms M such that $\Gamma \vdash M : \sigma$ to terms $C(M)$ such that $\Gamma' \vdash C(M) : \sigma'$. We require, moreover, that contexts constitute the morphisms of a category (whose objects are pairs (Γ, σ)), and that for any well-typed term

$(x : \sigma) \vdash P : \sigma'$ and any environment Γ , the mapping

$$M \mapsto P[M/x] : \mathcal{L}_\Gamma^\sigma \longrightarrow \mathcal{L}_\Gamma^{\sigma'}$$

is a context.

We call a term M *closed* if $\emptyset \vdash M : \sigma$ for some σ (we usually omit mention of the empty environment).

- 2 For one or more types σ (called *program types*), a set of special closed terms $V : \sigma$ called *values*, an *evaluation* relation $M \Rightarrow V$ between closed terms and values, and a function Obs_σ mapping values $V : \sigma$ to some set O_σ of *observations*. (We do not require evaluation to be deterministic: we may have $M \Rightarrow V$ and $M \Rightarrow V'$ where V, V' are different values.)
- 3 A category \mathcal{C} with finite products.
- 4 An interpretation $\llbracket - \rrbracket$ of \mathcal{L} in \mathcal{C} , assigning to each type σ an object $\llbracket \sigma \rrbracket$; to each environment $\Gamma = (x_1 : \sigma_1, \dots, x_n : \sigma_n)$ the object $\llbracket \Gamma \rrbracket = \llbracket \sigma_1 \rrbracket \times \dots \times \llbracket \sigma_n \rrbracket$; and to each typing judgement $\Gamma \vdash M : \sigma$ a morphism $\llbracket M \rrbracket_\Gamma : \llbracket \Gamma \rrbracket \rightarrow \llbracket \sigma \rrbracket$. The interpretation is required to satisfy the following conditions:

Variable condition: if $\Gamma = (x_1 : \sigma_1, \dots, x_n : \sigma_n)$ then $\llbracket x_i \rrbracket_\Gamma$ is the i th projection $\llbracket \Gamma \rrbracket \rightarrow \llbracket \sigma_i \rrbracket$.

Compositionality: if $\Gamma \vdash N_i : \sigma_i$ for each i , and $\Delta \vdash M : \tau$ where $\Delta = (x_1 : \sigma_1, \dots, x_n : \sigma_n)$, then

$$\llbracket M[\vec{N}_i/\vec{x}_i] \rrbracket_\Gamma = \llbracket M \rrbracket_\Delta \circ \langle \llbracket N_1 \rrbracket_\Gamma, \dots, \llbracket N_n \rrbracket_\Gamma \rangle : \llbracket \Gamma \rrbracket \rightarrow \llbracket \tau \rrbracket.$$

Context condition: if $\llbracket M \rrbracket_\Gamma = \llbracket M' \rrbracket_\Gamma$ and $C(-)$ is a context of appropriate type, then $\llbracket C(M) \rrbracket_{\Gamma'} = \llbracket C(M') \rrbracket_{\Gamma'}$.

- 5 For each program type σ , a relation \sqsubseteq between morphisms $\llbracket V \rrbracket$ (V a value of type σ) and morphisms $\llbracket M \rrbracket$ (M a closed term of type σ), such that the following requirement is satisfied:

Adequacy: $v \sqsubseteq \llbracket M \rrbracket$ iff there is some V such that $\llbracket V \rrbracket = v$ and $M \Rightarrow V$.

(In the case of deterministic programming languages, \sqsubseteq will typically be just the equality relation. In non-deterministic settings, $\llbracket M \rrbracket$ will be something like a set of possible values for M , and \sqsubseteq will be something like subset inclusion.)

In this situation, we may obtain (from items 1 and 2 alone) a notion of observational equivalence for \mathcal{L} :

Definition 3.1 (Observational equivalence). Given $M, N \in \mathcal{L}_\Gamma^\sigma$, we say M, N are *observationally equivalent* at (Γ, σ) ($M \approx_\Gamma^\sigma N$) if for all contexts $C(-) : (\Gamma, \sigma) \rightarrow (\emptyset, \tau)$ and values $V : \tau$, we have $C(M) \Rightarrow V$ iff $C(N) \Rightarrow V'$ for some V' with $Obs_\tau(V) = Obs_\tau(V')$.

Remark 3.1. In the case of non-deterministic languages, this notion of observational equivalence is rather a weak one, in at least two respects. Firstly, it does not allow us to distinguish a program that *always* returns 0 from one which may return 0 or may diverge. Secondly, it does not capture anything like *bisimilarity* of terms: for instance, if M, N are of some type such as $unit \rightarrow unit \rightarrow nat$, then $M \approx N$ will typically tell us only that the possible values of $M(*)$ and $N(*)$ are the same — it does not tell us that for any potential value of $M(*)$ there is a potential value of $N(*)$ with the same possible behaviours. We will not worry too much about these limitations, since non-deterministic languages will play a rather peripheral role in this paper. Nevertheless, it would be interesting to know whether the ideas presented here can be made to work for stronger notions of observational equivalence.

We now introduce two conditions which express “goodness of fit” criteria for a language and a model. Both of these have played major roles in the development of denotational semantics.

Definition 3.2 (Full abstraction, completeness). Let $\llbracket - \rrbracket$ be an interpretation for a language \mathcal{L} as above.

(i) $\llbracket - \rrbracket$ is *fully abstract* if for all terms $M, N \in \mathcal{L}_\Gamma^\sigma$,

$$M \approx_\Gamma^\sigma N \Rightarrow \llbracket M \rrbracket_\Gamma = \llbracket N \rrbracket_\Gamma.$$

(The converse implication holds automatically in our setting.) (ii) $\llbracket - \rrbracket$ is *complete* if for every morphism $f : \llbracket \Gamma \rrbracket \rightarrow \llbracket \sigma \rrbracket$ there is a term $M \in \mathcal{L}_\Gamma^\sigma$ with $\llbracket M \rrbracket_\Gamma = f$.

The word “universal” is sometimes used here in place of “complete”, but in the context of the present paper we prefer not to overload this term too heavily.

It is easy to see that there is essentially only one model of \mathcal{L} that is both fully abstract and complete. More precisely:

Proposition 3.1. *There is exactly one model $(\mathcal{C}, \llbracket - \rrbracket)$ of \mathcal{L} (up to equivalence of categories) that is fully abstract and complete, and in which every object of \mathcal{C} is a finite product of objects $\llbracket \sigma \rrbracket$. If $(\mathcal{D}, \llbracket - \rrbracket')$ is*

any other fully abstract and complete model of \mathcal{L} , then \mathcal{C} is equivalent to a full subcategory of \mathcal{D} .

Proof. We may construct such a model as follows:

- objects of \mathcal{C} are tuples of types $(\sigma_1, \dots, \sigma_n)$;
- morphisms $(\sigma_1, \dots, \sigma_n) \rightarrow (\tau_1, \dots, \tau_m)$ in \mathcal{C} are m -tuples

$$([M_1], \dots, [M_m]),$$

where $(x_1 : \sigma_1, \dots, x_n : \sigma_n) \vdash M_i : \tau_i$, and $[M_i]$ is the observational equivalence class of M_i ;

- the interpretation $\llbracket - \rrbracket$ of \mathcal{L} in \mathcal{C} is the canonical one;
- the relation \sqsubseteq is given by $v \sqsubseteq f$ iff $v = \llbracket V \rrbracket$ and $f = \llbracket M \rrbracket$ for some $V, M : \sigma$ where $M \Rightarrow V$.

Given any other such model \mathcal{C}' , we may assume the objects of \mathcal{C}' are tuples $(\sigma_1, \dots, \sigma_n)$, and by completeness, the morphisms $(\sigma_1, \dots, \sigma_n) \rightarrow (\tau_1, \dots, \tau_m)$ must be tuples of terms M_i as above modulo some equivalence relation \sim . By full abstraction, \sim must include componentwise observational equivalence; but by the context condition and adequacy, \sim cannot be any larger than this. Thus \mathcal{C}' is equivalent to \mathcal{C} . The second statement of the Proposition follows immediately from the first. \square

The model \mathcal{C} in this proposition is in some sense the canonical or “best possible” model of \mathcal{L} . The above proof shows how \mathcal{C} can be obtained as the syntactic model (or *term model*) of \mathcal{L} ; however, this way of constructing a model is not very enlightening, since proving things about this model is no easier than proving things directly about \mathcal{L} itself. In general, we would prefer a more “semantic” characterization of \mathcal{C} which gave us a better mathematical handle on its structure — in other words, which offered us some understanding of \mathcal{C} that was independent of the syntax of \mathcal{L} . Indeed, one possible working philosophy for denotational semantics (which the present author finds appealing) is that its goal should be to give good mathematical characterizations of the fully abstract and complete model for various languages \mathcal{L} .

Note in passing that since most programming languages of interest are executable in an “effective” fashion, a universal model will inevitably need some notion of effectivity built in, to ensure that all morphisms are in some sense computable. Usually we do this simply by making use of the set of partial recursive functions at some point in the construction.

4. Universal types

We now show how the idea of universal objects can be of use in denotational semantics. Given a model $(\mathcal{C}, \llbracket - \rrbracket)$ of a language \mathcal{L} as in items 1–5 above, we may introduce the following notion:

Definition 4.1 (Universal type). (i) A type ρ is a *definable retract* of σ with respect to $\llbracket - \rrbracket$ if there are well-typed terms

$$x : \rho \vdash F : \sigma, \quad y : \sigma \vdash G : \rho$$

such that $\llbracket G \rrbracket_{(x:\rho)} \circ \llbracket F \rrbracket_{(y:\sigma)} = id_{\llbracket \rho \rrbracket}$.

(ii) A type σ is a *universal type* for $\llbracket - \rrbracket$ if every type of \mathcal{L} is a definable retract of σ w.r.t. $\llbracket - \rrbracket$.

Clearly, if σ is a universal type and σ is a definable retract of τ then τ is also a universal type. Thus, if there exists a universal type there are likely to be several. Usually, of course, we are interested in identifying the simplest possible universal type.

Of course, it frequently happens that σ is a universal type for $\llbracket - \rrbracket$ and $\llbracket \sigma \rrbracket$ is a universal object in \mathcal{C} , although neither fact implies the other in general: there may be non-denotable objects of \mathcal{C} that are not retracts of $\llbracket \sigma \rrbracket$, and conversely the components of a retraction $\llbracket \rho \rrbracket \triangleleft \llbracket \sigma \rrbracket$ need not be definable in \mathcal{L} . It can also happen that \mathcal{C} has a universal object, but none of the form $\llbracket \sigma \rrbracket$.

If the types of \mathcal{L} are inductively generated in some way, one can often prove by induction on the types that some type σ is universal. For example:

Proposition 4.1. *Suppose \mathcal{L} is a simply typed λ -calculus over some set of ground types γ — that is, the types of \mathcal{L} are given by the grammar $\sigma ::= \gamma \mid \sigma_1 \rightarrow \sigma_2$. Suppose also that $M =_{\beta\eta} M'$ implies $\llbracket M \rrbracket_{\Gamma} = \llbracket M' \rrbracket_{\Gamma}$ for any suitable Γ . Then σ is universal for $\llbracket - \rrbracket$ iff all the ground types plus the type $\sigma \rightarrow \sigma$ are definable retracts of σ .*

Proof. The left-to-right implication is immediate. For the converse, we show by induction that all types ρ are definable retracts of σ . If ρ is a ground type this is true by hypothesis. If $\rho = \rho_1 \rightarrow \rho_2$ where the ρ_i are definable retracts of σ via terms

$$x_i : \rho_i \vdash F_i : \sigma, \quad y_i : \sigma \vdash G_i : \rho_i$$

($i = 1, 2$), then ρ is a definable retract of $\sigma \rightarrow \sigma$ via the terms

$$\begin{aligned} f : \rho &\vdash (\lambda y_1 : \sigma. F_2[f(G_1)/x_2]) : \sigma \rightarrow \sigma, \\ g : \sigma \rightarrow \sigma &\vdash (\lambda x_1 : \rho_1. G_2[g(F_1)/y_2]) : \rho \end{aligned}$$

But $\sigma \rightarrow \sigma$ is itself a definable retract of σ , so the result follows easily. \square

For some languages we wish to consider (particularly call-by-value languages), full $\beta\eta$ -equality does not hold in the natural models. For such languages, a slightly more refined (but more cumbersome) version of the above proposition is useful:

Proposition 4.2. *Suppose \mathcal{L} is a simply typed λ -calculus over a set of ground types, in which certain closed terms (including all of the form $\lambda x.M$) are designated as values. Suppose also that $\llbracket - \rrbracket$ is an interpretation such that*

- *if V is a value then $\llbracket (\lambda x.M)V \rrbracket_\Gamma = \llbracket M[V/x] \rrbracket_\Gamma$ for any suitable Γ ;*
- *for any type $\sigma = \sigma_1 \rightarrow \sigma_2$, $\llbracket (\lambda f x.f x)f \rrbracket_{(f:\sigma)} = \llbracket f \rrbracket_{(f:\sigma)}$.*

Let α be any type for which there exists some value $U : \alpha$. Then σ is universal for $\llbracket - \rrbracket$ iff all ground types plus the type $(\alpha \rightarrow \sigma) \rightarrow \sigma$ are definable retracts of σ .

Proof. As in the previous proposition, except that if $\rho = \rho_1 \rightarrow \rho_2$ we use the retraction $\llbracket \rho \rrbracket \triangleleft \llbracket (\alpha \rightarrow \sigma) \rightarrow \sigma \rrbracket$ defined by the terms

$$\begin{aligned} f : \rho_1 \rightarrow \rho_2 &\vdash (\lambda y_1 : \alpha \rightarrow \sigma. F_2[f(G_1[y_1 U])]) : (\alpha \rightarrow \sigma) \rightarrow \sigma, \\ g : (\alpha \rightarrow \sigma) \rightarrow \sigma &\vdash (\lambda x_1 : \rho_1. G_2[g(\lambda z : \alpha. F_1[x_1])]) : \rho. \end{aligned}$$

\square

The examples we shall focus on in this paper will all be of the above kind. However, the same principle clearly applies to many languages with other type constructors.

We come at last to the connection between universal types and the key ideas of denotational semantics. The following observation, though trivial to prove, turns out to be very powerful:

Proposition 4.3. *Suppose σ is a universal type for $\llbracket - \rrbracket$. Then*

- (i) *$\llbracket - \rrbracket$ is fully abstract iff it is fully abstract for terms*

$$M \in \mathcal{L}_{(x_1:\sigma, \dots, x_n:\sigma)}^\sigma$$

in the obvious sense.

- (ii) *$\llbracket - \rrbracket$ is complete iff it is complete for morphisms*

$$f : \llbracket \sigma \rrbracket \times \cdots \times \llbracket \sigma \rrbracket \rightarrow \llbracket \sigma \rrbracket.$$

In the case of typed λ -calculi, we can simplify this further:

Proposition 4.4. *Suppose \mathcal{L} is a typed λ -calculus, $\llbracket - \rrbracket$ respects β -equality as in Proposition 4.1, and σ is a universal type for $\llbracket - \rrbracket$. Then*

- (i) $\llbracket - \rrbracket$ is fully abstract iff it is fully abstract for closed terms $M : \sigma$.
- (ii) $\llbracket - \rrbracket$ is complete iff it is complete for morphisms $1 \rightarrow \llbracket \sigma \rrbracket$.

These results provide a useful tool for showing that models are fully abstract or complete. As we shall see in the next section, this often leads to simpler proofs than those obtained by more traditional methods.

Definition 4.1 above gives a notion of universal type relative to a particular interpretation $\llbracket - \rrbracket$. However, we may also define the following notion, which depends only on \mathcal{L} and its evaluation relation:

Definition 4.2 (Universal type for a language). A type σ is a *universal type for \mathcal{L}* iff it is a universal type for the fully abstract and complete model of \mathcal{L} — that is, if for all ρ there are terms $x : \rho \vdash F : \sigma$, $y : \sigma \vdash G : \rho$ such that $G[F/y] \approx x$.

The following proposition is clear:

Proposition 4.5. *Suppose $(\mathcal{C}, \llbracket - \rrbracket)$ is the term model of \mathcal{L} . Then $\llbracket \sigma \rrbracket = (\sigma)$ is a universal object in \mathcal{C} iff σ is a universal type for \mathcal{L} and $(\sigma, \sigma) \triangleleft (\sigma)$.*

(Note that the condition $(\sigma, \sigma) \triangleleft (\sigma)$ holds automatically under various mild conditions: for instance, if \mathcal{L} has well-behaved product types, or if \mathcal{L} is a typed λ -calculus admitting a representation of the booleans.)

In the next section we will see some examples of universal types for particular languages. In a sense, a universal type is a rabbit pulled from a hat — there is no obvious general method for finding a universal type for a language if there is one — but once we have pulled this rabbit, we can often use it to good effect. For example, in the case of simply typed λ -calculi, the following general line of attack suggests itself. First, give a semantic construction of a category \mathcal{C} , either by constructing an appropriate λ -algebra and taking its Karoubi envelope, or otherwise. Next, give a simply typed λ -calculus \mathcal{L} , an interpretation $\llbracket - \rrbracket$ of \mathcal{L} in \mathcal{C} , and a type σ of \mathcal{L} , such that:

- 1 $\llbracket - \rrbracket$ validates $\beta\eta$ -equality [resp. the conditions of Proposition 4.2],
- 2 all ground types are definable retracts of σ with respect to $\llbracket - \rrbracket$,
- 3 $\sigma \rightarrow \sigma$ [resp. $(\alpha \rightarrow \sigma) \rightarrow \sigma$] is a definable retract of σ w.r.t. $\llbracket - \rrbracket$,
- 4 $\llbracket - \rrbracket$ is fully abstract for closed terms $M : \sigma$,
- 5 $\llbracket - \rrbracket$ is complete for morphisms $1 \rightarrow U$.

In practice these are usually trivial to verify except for fact 3, which typically just requires a little programming in \mathcal{L} to define a suitable retraction. It then follows from the above results that $(\mathcal{C}, \llbracket - \rrbracket)$ is a fully abstract and complete model for \mathcal{L} .

Of course, this does not give us much of a clue how to construct a good semantic model for a given language, though once we have found one (together with a universal type) it often offers (perhaps with hindsight) a smooth route to the relevant results.

5. Syntax and semantics of PCF

Next, we wish to consider how the above ideas work in practice for a variety of particular languages. For convenience, we will formulate all our languages as extensions of PCF, the prototype functional programming language first studied by Scott [Sc93] and Plotkin [P177]. PCF embodies an appealing notion of sequential, functional computation at higher types, and in essence forms the basis of programming languages such as Standard ML and Haskell. From our point of view, PCF provides a convenient “core language” to which we may add various other computational features in order to investigate the expressive power and the semantics of the resulting languages. In this section we will review the information we will need concerning the syntax and semantics of PCF.

In fact, it will be convenient to define two slightly different versions of PCF — the original *call-by-name* version PCF_N as in [P177], and a *call-by-value* version PCF_V as in e.g. [Si90]. From our point of view, the difference between the two versions is not very significant, and all the results we mention will transfer easily from one to the other. However, for several of the extensions of PCF that we consider, one version of the language turns out to be more convenient. (This is because, in the categories we consider, the simplest and most natural universal object corresponds sometimes to a call-by-name type, and sometimes to a call-by-value type.)

The syntax of the two versions is almost identical. We take PCF_N to be the simply typed λ -calculus with ground types *nat*, *bool* and *unit*, together with constants

$$\begin{array}{ll} * & : \textit{unit}, & \text{true, false} & : \textit{bool}, \\ 0, 1, 2, \dots & : \textit{nat}, & \text{succ, pred} & : \textit{nat} \rightarrow \textit{nat}, \\ \text{iszero} & : \textit{nat} \rightarrow \textit{bool}, & \text{cond}_\gamma & : \textit{bool} \rightarrow \gamma \rightarrow \gamma \rightarrow \gamma, \\ Y_\sigma & : (\sigma \rightarrow \sigma) \rightarrow \sigma \end{array}$$

where γ ranges over ground types and σ over all types. The definition for PCF_V is the same except that σ ranges only over *arrow* types.

The operational semantics for PCF_N is given as follows. We declare the ground types (only) to be program types, and designate as *values* V the terms $*$, true , false , $0, 1, 2, \dots$. For each ground type γ , we take O_γ to be the set of values of type γ and Obs_γ to be the identity. The evaluation relation $M \Rightarrow V$ is defined via the following derivation rules. (Note that \smile denotes truncated subtraction: $m \smile n = \max(m - n, 0)$. We also define $\text{iszero}(0) = \text{true}$ and $\text{iszero}(n + 1) = \text{false}$.)

$$\begin{array}{c} \frac{}{\overline{V \Rightarrow V}} \qquad \frac{M \Rightarrow n}{\text{succ } M \Rightarrow n + 1} \\ \\ \frac{M \Rightarrow n}{\text{pred } M \Rightarrow n \smile 1} \qquad \frac{M \Rightarrow n}{\text{iszero } M \Rightarrow \text{iszero}(n)} \\ \\ \frac{M \Rightarrow \text{true} \quad N \Rightarrow V}{\text{cond}_\gamma MNP \Rightarrow V} \qquad \frac{M \Rightarrow \text{false} \quad P \Rightarrow V}{\text{cond}_\gamma MNP \Rightarrow V} \\ \\ \frac{M[N/x]P_1 \dots P_r \Rightarrow V}{(\lambda x.M)NP_1 \dots P_r \Rightarrow V} \qquad \frac{M(Y_\sigma M)P_1 \dots P_r \Rightarrow V}{Y_\sigma MP_1 \dots P_r \Rightarrow V} \end{array}$$

The operational semantics of PCF_V is somewhat more complicated. Here we declare *all* types to be program types. We designate as values V all the ground type values as before, together with all closed terms of the form $\lambda x.M$. For ground types γ , we take O_γ and Obs_γ as before; for arrow types σ , we take O_σ to be the one-element set and Obs_σ the unique mapping into this set. (Intuitively, ground type values are observable, but at higher types only the fact of termination is observable.) The evaluation relation is defined by the same rules as above, except that in place of the last two rules we have

$$\frac{M \Rightarrow \lambda x.M' \quad N \Rightarrow V \quad M'[V/x] \Rightarrow V'}{MN \Rightarrow V'} \qquad \frac{}{Y_\sigma M \Rightarrow \lambda x.M(Y_\sigma M)x}$$

The first of these rules encapsulates the fundamental idea of call-by-value: a term N must be *evaluated* before it can be passed as a parameter to a function. The absence of the additional arguments $P_1 \dots P_r$ in both these rules corresponds to the fact that we have values at all types, not just ground types.

Several minor variations in these definitions are possible — for instance, we could dispense with the types *bool* and *unit*, or add product

types — but these variations are not very interesting from the point of view of denotational semantics since all the variants are easily inter-translatable. In fact, even PCF_N and PCF_V admit good translations into each other and are in some sense just different presentations of the same thing (see e.g. [Lo95, Chapter 6]). Similar remarks apply to the extensions we consider. (We will return to this point in Section 7.) In the examples to follow, the choice between the two versions is thus more a matter of convenience and taste than of mathematical substance.

We will view the above languages, and the extensions to be introduced, as instances of the scenario described in Section 3. In all cases, we will take the notion of substitution to be the usual operation of simultaneous capture-avoiding substitution, and the notion of context to be given by the usual notion of term context (admitting textual substitutions which may result in variable capture).

We will also consider interpretations of a certain standard kind in cartesian closed categories. Often we can use the same cartesian closed category \mathcal{C} for both the call-by-name and call-by-value versions of a language (in many cases, \mathcal{C} will arise as the Karoubi envelope of some λ -algebra). Let us consider how to model a call-by-name language \mathcal{L}_N extending PCF_N . First, for each ground type γ of \mathcal{L}_N we choose an object X_γ of \mathcal{C} . Unless otherwise stated, the objects X_{nat} , X_{bool} , X_{unit} will be the evident objects N_\perp , 2_\perp , 1_\perp respectively. We can then interpret the types of \mathcal{L}_N as follows:

$$\llbracket \gamma \rrbracket = X_\gamma, \quad \llbracket \sigma_1 \rightarrow \sigma_2 \rrbracket = \llbracket \sigma_2 \rrbracket^{\llbracket \sigma_1 \rrbracket}$$

Next, for each constant $c : \sigma$ of \mathcal{L}_N we specify a suitable morphism $\llbracket c \rrbracket : 1 \rightarrow \llbracket \sigma \rrbracket$. The intended interpretation of the constants is usually obvious. For example, we generally take $\llbracket Y_\sigma \rrbracket$ to be the exponential transpose of some fixed point operator $Fix_\sigma : \llbracket \sigma \rrbracket^{\llbracket \sigma \rrbracket} \rightarrow \llbracket \sigma \rrbracket$; in the case that $\mathcal{C} \equiv \mathcal{K}(A)$, such operators arise in a standard way from a single fixed point operator on A itself. We may then extend the interpretation to all terms by means of the clauses:

$$\begin{aligned} \llbracket x_i \rrbracket^\Gamma &= \pi_i \quad (\text{where } \Gamma = (x_1 : \sigma_1, \dots, x_n : \sigma_n)), \\ \llbracket \lambda x : \sigma. M \rrbracket^\Gamma &= \text{curry}(\llbracket M \rrbracket^{\Gamma, x : \sigma}), \\ \llbracket MN \rrbracket^\Gamma &= \text{eval} \circ \langle \llbracket M \rrbracket^\Gamma, \llbracket N \rrbracket^\Gamma \rangle. \end{aligned}$$

where *curry* and *eval* are given by the cartesian closed structure of \mathcal{C} . Finally, unless otherwise stated, we take the relation \sqsubseteq to be simply the identity relation on morphisms of the form $\llbracket V \rrbracket$.

The interpretation of call-by-value languages is slightly more involved. Here we require additionally that \mathcal{C} has ω -fold products, and is equipped with a “lifting” monad (\perp, η, μ) . Suppose \mathcal{L}_V is an extension of PCF_V .

As before, we take an object X_γ for each ground type γ ; in all the examples we shall consider, the choice of X_γ will be standard. We now define for each type σ an object $\llbracket \sigma \rrbracket$, and for each *arrow* type σ an object $\llbracket \sigma \rrbracket$, by mutual recursion as follows:

$$\begin{aligned} \llbracket \gamma \rrbracket &= X_\gamma, & \llbracket \sigma_1 \rightarrow \sigma_2 \rrbracket &= \llbracket \sigma_1 \rightarrow \sigma_2 \rrbracket_\perp, \\ \llbracket \gamma \rightarrow \sigma_2 \rrbracket &= \llbracket \sigma_2 \rrbracket^{|\gamma|}, & \llbracket \sigma_1 \rightarrow \sigma_2 \rrbracket &= \llbracket \sigma_2 \rrbracket^{\llbracket \sigma_1 \rrbracket} \quad (\sigma_1 = \tau \rightarrow \tau'). \end{aligned}$$

Here, by definition, $|unit| = 1$, $|bool| = 2$, and $|nat| = \omega$; the notation $X^{|\gamma|}$ simply means the categorical product in \mathcal{C} of $|\gamma|$ copies of X . The only reason for the distinction between ground types and arrow types in the above definition is that \mathcal{C} may lack suitable objects to play the role of $\llbracket \gamma \rrbracket$.

Next, one again specifies a morphism $\llbracket c \rrbracket : 1 \rightarrow \llbracket \sigma \rrbracket$ for each constant c . One then extends the interpretation to all terms by means of clauses for variables, abstraction and application. The following works for all the call-by-value examples treated below:

$$\begin{aligned} \llbracket x_i \rrbracket^\Gamma &= \pi_i, \\ \llbracket \lambda x : \sigma. M \rrbracket^\Gamma &= \eta \circ \llbracket \tau \rrbracket^\eta \circ \mathit{curry}(\llbracket M \rrbracket^{\Gamma, x:\sigma}), \\ \llbracket MN \rrbracket^\Gamma &= \mathit{eval} \circ \langle \mathit{strict} \circ \alpha_{\sigma \rightarrow \tau} \circ \llbracket M \rrbracket^\Gamma, \llbracket N \rrbracket^\Gamma \rangle. \end{aligned}$$

(In the second clause, we assume M has type τ , and if σ is a ground type γ then we understand $\llbracket \tau \rrbracket^\eta$ to mean the evident map $\llbracket \tau \rrbracket^{|\gamma|} \rightarrow \llbracket \tau \rrbracket^{|\gamma|}$. In the third clause, $\alpha_{\sigma \rightarrow \tau}$ is the evident map $\llbracket \sigma \rightarrow \tau \rrbracket_\perp \rightarrow \llbracket \sigma \rightarrow \tau \rrbracket$, and $\mathit{strict} : B_\perp^A \rightarrow B_\perp^{A_\perp}$ is the morphism that extends a map $A \rightarrow B_\perp$ to a strict map $A_\perp \rightarrow B_\perp$.) As before, the relation \sqsubseteq is the identity on morphisms $\llbracket V \rrbracket$ unless otherwise stated.

For all the languages $\mathcal{L}_N, \mathcal{L}_V$ and all the interpretations we shall consider, the proof of adequacy is completely routine (the standard method of proof as in [Pl77] works without any problems).

6. Examples of universal types

We will now investigate how the ideas of Sections 1–4 work out for a variety of extensions of PCF. Since our main purpose is to survey a large number of instances of the generic situation outlined above, we will not present all the examples in full detail but will refer freely to other papers, concentrating here on those points that are not covered elsewhere in the literature.

6.1. PCF itself

We start with a negative result: for the language PCF itself (either PCF_N or PCF_V) there is no universal type. This can be proved fairly

easily using the fully abstract and complete game models described e.g. in [HO00, AM99]. (The result is folklore among the experts in game semantics, but as far as we know a proof has not yet appeared in the literature.) It would seem, intuitively, that the term model of PCF increases indefinitely in its complexity as one goes up the types.

Of course, one can conservatively extend PCF by adding recursive types (the resulting language is known as FPC) — it is then fairly easy to show that a recursively defined type such as $\mu\sigma.nat + (\sigma \rightarrow \sigma)$ is universal. However, there would seem to be little gain in doing this, since we do not have any particularly good handle on the structure of this type and its monoid of endomorphisms. Indeed, it would seem that in the known fully abstract and complete models for FPC, it is really no easier to understand this type than to understand the finite approximations from which it is constructed, viz.

$$\sigma_0 = \emptyset, \quad \sigma_{n+1} = nat + (\sigma_n \rightarrow \sigma_n),$$

which is tantamount to understanding all the PCF types in the first place. It would therefore seem that our approach via universal types has little to offer in the case of pure PCF.

6.2. PCF with parallel functions

We now consider the well known extension of PCF with “parallel” computable functionals discovered by Plotkin [Pl77] and Sazonov [Sa76]. Here it suits our purposes to work with the call-by-value version. Specifically, let PCF_V^{++} be the language obtained by extending the definition of PCF_V with constants

$$\begin{aligned} \text{por} & : (unit \rightarrow bool) \rightarrow (unit \rightarrow bool) \rightarrow bool, \\ \text{exists} & : ((unit \rightarrow nat) \rightarrow bool) \rightarrow bool \end{aligned}$$

(where *por* abbreviates “parallel or”), and evaluation rules

$$\frac{M \Rightarrow \text{true}}{\text{por } MN \Rightarrow \text{true}} \quad \frac{N \Rightarrow \text{true}}{\text{por } MN \Rightarrow \text{true}} \quad \frac{M \Rightarrow \text{false} \quad N \Rightarrow \text{false}}{\text{por } MN \Rightarrow \text{false}}$$

$$\frac{MP_n \Rightarrow \text{true}}{\text{exists } M \Rightarrow \text{true}} \quad \frac{MP_{\perp} \Rightarrow \text{false}}{\text{exists } M \Rightarrow \text{false}}$$

where $P_n = \lambda x.n$ for $n = 0, 1, \dots$, and $P_{\perp} = Y(\lambda t.t)$.

It is well known that a fully abstract and complete model for PCF^{++} is provided by the category of effective Scott domains and computable maps (see [Pl77]), or indeed by its full subcategory $\mathbf{Coh}_{\text{eff}}$ of *effective*

coherent domains. It is also known that the object $\mathbb{T}^\omega = 2_{\perp}^{\mathbb{N}}$ is a universal object in this category, and indeed that \mathbf{Coh}_{eff} is equivalent to the Karoubi envelope of the corresponding λ -algebra \mathbb{T}_{eff}^ω (see [P178]). Since $\mathbb{T}^\omega = \llbracket nat \rightarrow bool \rrbracket$, it follows that $nat \rightarrow bool$ is a universal type for \mathbf{PCF}_V^{++} . (Note that $nat \rightarrow bool$ is also a universal type for the call-by-name analogue \mathbf{PCF}_N^{++} , but that in this case the corresponding object of \mathbf{Coh}_{eff} is the more complicated object $2_{\perp}^{\mathbb{N}^+}$.)

Let us now use this universal type to reconstruct the above results. It is possible to define the λ -algebra \mathbb{T}_{eff}^ω concretely, and then obtain the category \mathbf{Coh}_{eff} as its Karoubi envelope (the details are more or less covered in [P178]), though this seems slightly perverse as it is probably easier to construct \mathbf{Coh}_{eff} directly. In any case, we may then give an adequate interpretation $\llbracket - \rrbracket$ of \mathbf{PCF}^{++} in \mathbf{Coh}_{eff} as usual. Let σ be the type $nat \rightarrow bool$. Regarding the five conditions given at the end of Section 4, it is clear that $\llbracket - \rrbracket$ validates the conditions mentioned in Proposition 4.2, and that nat , $bool$ and $unit$ are definable retracts of σ . It is also trivial that $\llbracket - \rrbracket$ is fully abstract and complete for the type σ : just note that terms denoting different functions are observationally distinguishable, and that \mathbf{PCF}^{++} is complete for first-order partial recursive functions. To show full abstraction and completeness at all types, It therefore only remains to supply terms

$$\begin{aligned} f &: (unit \rightarrow \sigma) \rightarrow \sigma \vdash \mathit{Lambda}[f] : \sigma, \\ x &: \sigma \vdash \mathit{Apply}[x] : (unit \rightarrow \sigma) \rightarrow \sigma \end{aligned}$$

that define a retraction $\llbracket (unit \rightarrow \sigma) \rightarrow \sigma \rrbracket \triangleleft \llbracket \sigma \rrbracket$. It is an interesting (and somewhat non-trivial) exercise in programming in \mathbf{PCF}^{++} to find suitable terms which correspond to the retraction $\llbracket \mathbb{T}^\omega \rightarrow \mathbb{T}^\omega \rrbracket \triangleleft \llbracket \mathbb{T}^\omega \rrbracket$ described semantically in [P178]. However, the terms themselves are slightly grungy and we omit them here.

Finally, we have to check that the terms *do* define a retraction. If \mathbf{Coh}_{eff} has been defined concretely, this amounts to showing that the interpretation of

$$f : (unit \rightarrow \sigma) \rightarrow \sigma \vdash \mathit{Apply}[\mathit{Lambda}[f]]$$

is the identity map on $\llbracket (unit \rightarrow \sigma) \rightarrow \sigma \rrbracket$. If \mathbf{Coh}_{eff} has been obtained as the Karoubi envelope of \mathbb{T}_{eff}^ω , it amounts to showing that the interpretation of

$$x : \sigma \vdash \mathit{Lambda}[\mathit{Apply}[x]]$$

coincides with the idempotent on \mathbb{T}_{eff}^ω corresponding to the untyped λ -term $\lambda fx. fx$. In either case, the verification is straightforward but tedious, owing to the complexity of the terms *Lambda* and *Apply*.

It is a moot point whether this is any simpler than the usual proof of full abstraction and completeness (given in [P177]), but it at least offers an interesting alternative. In particular, it is distinguished from the usual proof by the fact that only relatively low types are involved.

Another advantage of our proof is that it is easily *portable*: once the terms *Lambda* and *Apply* have been constructed, they can be re-used for other models of PCF^{++} . For example, we can now show cheaply that the interpretation of PCF^{++} in the category of PERs over the natural numbers (see e.g. [Lo95]) is fully abstract and complete. Here again, all the other conditions are trivial to verify, so we just need to show that the above terms do define a retraction in this model. But this follows immediately from the classical Myhill-Shepherdson theorem, which tells us that the elements of $\llbracket (\text{unit} \rightarrow \sigma) \rightarrow \sigma \rrbracket$ in this category are the same as in $\mathbf{Coh}_{\text{eff}}$.

6.3. PCF with non-determinism

Next, let us consider an extension of PCF with non-deterministic choice. In this case, the call-by-name version is definitely simpler. Specifically, we add to the definition of PCF_N a single new constant

$$\text{choose} : \text{nat}$$

together with the evaluation rule

$$\frac{}{\text{choose} \Rightarrow n}$$

where n ranges over all numerals. To give an adequate semantics for this language, the interpretation of the type *nat* cannot be just the usual set \mathbb{N}_\perp , but will need to incorporate all r.e. subsets of *nat*; the denotation of a closed term $M : \text{nat}$ will then be the set of all n such that $M \Rightarrow n$. This suggests that we work with Scott's λ -algebra $\mathcal{P}\omega$, or rather its effective submodel $\mathcal{P}\omega_{\text{re}}$, whose underlying set is the set of r.e. subsets of \mathbb{N} (see [Sc76]).

The concrete construction of the λ -algebra $\mathcal{P}\omega_{\text{re}}$ is relatively familiar and straightforward (see e.g. [Sc76, Ba84]). As is shown in [Sc76], it yields as its Karoubi envelope the category $\mathbf{CLat}_{\text{eff}}$ of *effective continuous lattices* and computable maps.

One can now give an interpretation of $\text{PCF} + \text{choose}$. in this category. The interpretation of types is given as above but using non-standard interpretations for the ground types:

$$X_{\text{nat}} = \mathcal{P}\omega, \quad X_{\text{bool}} = \mathcal{P}2, \quad X_{\text{unit}} = \mathcal{P}1.$$

The interpretation of the constants is reasonably evident: for example, $\llbracket n \rrbracket : 1 \rightarrow \mathcal{P}\omega$ corresponds to the singleton $\{n\} \in \mathcal{P}\omega$; $\llbracket \text{choose} \rrbracket$ corresponds to the set $\omega \in \mathcal{P}\omega$; and $\llbracket \text{succ} \rrbracket$ corresponds to the continuous mapping $A \mapsto \{n+1 \mid n \in A\} : \mathcal{P}\omega \rightarrow \mathcal{P}\omega$. The interpretation of variables, abstraction and application is then completely standard. Finally, the relation \sqsubseteq for ground types corresponds to set-theoretic inclusion for singleton subsets: if $\llbracket V \rrbracket$ corresponds to $\{v\} \in \mathcal{P}\omega$ and $\llbracket M \rrbracket$ corresponds to $A \in \mathcal{P}\omega$, then $\llbracket V \rrbracket \sqsubseteq \llbracket M \rrbracket$ iff $v \in A$. The proof of adequacy for this interpretation is straightforward.

We now claim that *nat* is a universal type for the above interpretation. Clearly all the ground types are definable retracts of *nat*. To define a retraction $\llbracket \text{nat} \rightarrow \text{nat} \rrbracket \triangleleft \llbracket \text{nat} \rrbracket$, we have to code up the familiar retraction $[\mathcal{P}\omega \rightarrow \mathcal{P}\omega] \triangleleft \mathcal{P}\omega$ in $\text{PCF} + \text{choose}$. We give here only a bare outline of how to do this, making free use of pseudocode. Suppose $\langle -, - \rangle$ is an effective pairing operation on \mathbb{N} , and $[-]$ is an effective encoding of finite subsets of \mathbb{N} as elements of \mathbb{N} . Let *check* be the term $\lambda b n. \text{cond } b \text{ } n$ (diverge). Now consider the terms

$$x : \text{nat} \vdash \lambda y. \text{let } \langle [A], r \rangle = x \text{ in} \\ \text{check (for each } a \in A, y = a) \text{ } r$$

$$f : \text{nat} \rightarrow \text{nat} \vdash \text{let } \langle [A], r \rangle = \text{choose in} \\ \text{check } (f(\text{let } n = \text{choose in check } (n \in A) \text{ } n) = r) \\ (\langle [A], r \rangle)$$

For the first term, the intuition is that we regard x as representing a term whose evaluation will give us some non-deterministic choice of an element of the graph of some function $f : \mathcal{P}\omega \rightarrow \mathcal{P}\omega$, and our task is to code the (non-deterministic) operation of type $\text{nat} \rightarrow \text{nat}$ whose behaviour on (non-deterministic) arguments y is precisely as given by f . For the second term, given just such an operation, our task is to construct a term of type *nat* that will non-deterministically choose an element of the graph of this operation.

It can be shown that the above terms do define a retraction as required. Since full abstraction and completeness are trivial at type *nat*, we have the following pleasing result, which as far as we are aware has not appeared in the literature before:

Theorem 6.1. *The above interpretation of $\text{PCF}_N + \text{choose}$ in $\mathbf{CLat}_{\text{eff}}$ is fully abstract and complete.*

6.4. The sequentially realizable functionals

Next, we briefly mention a more complex example in which the existence of a universal type turns out to be extremely useful. We refer the reader to [Lo02] for all the details.

The language PCF^{++} above shows how we can extend PCF with “parallel” operators whilst retaining the purely functional character of the language. (“Functional” here means that the meaning of a closed term of arrow type can be taken to be simply a function between the appropriate sets. More precisely, it means that the term model for the language is a *well-pointed* category: that is, $f = g : X \rightarrow Y$ iff $f \circ x = g \circ x$ for all $x : 1 \rightarrow X$.) Somewhat surprisingly, it is also possible to extend PCF by adding “sequentially computable” operators not already definable in PCF, and still remain functional in this sense. In fact, it turns out that there is a mathematically natural class of sequentially computable functionals with many attractive properties. These are known in [Lo02] as the (*effective*) *sequentially realizable* (or SR) functionals, and are closely akin to the *strongly stable* functionals of Bucciarelli and Ehrhard [BE91].

One of the main results of [Lo02] is that in the category constituted by the SR functionals, the object $[N_{\perp}^N \rightarrow N_{\perp}]$ is universal. Since this object is the denotation of the call-by-value type $(\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat}$ (call this type $\bar{2}$), we may state this fact as follows: for any call-by-value type σ , its denotation $\llbracket \sigma \rrbracket$ in the category of SR functionals is a retract of $\llbracket \bar{2} \rrbracket$. This follows easily once we have shown that $\llbracket \bar{3} \rrbracket \triangleleft \llbracket \bar{2} \rrbracket$ (where $\bar{3} \equiv \bar{2} \rightarrow \text{nat}$), which we show by giving a semantic construction of the required retraction. In fact, one half of this retraction is already definable in PCF_V ; the other half (the morphism $\llbracket \bar{2} \rrbracket \rightarrow \llbracket \bar{3} \rrbracket$) we may call H .

We may now consider the language obtained by extending PCF_V with a constant $H : \bar{2} \rightarrow \bar{3}$, and stipulating that $\llbracket H \rrbracket = H$. (One can give an operational semantics by means of a compilation to an abstract machine, but we will not give details here.) We now have an instance of the general situation described in this paper: the retraction $\llbracket (\text{unit} \rightarrow \bar{2}) \rightarrow \bar{2} \rrbracket \triangleleft \llbracket \bar{2} \rrbracket$ arising from $\llbracket \bar{3} \rrbracket \triangleleft \llbracket \bar{2} \rrbracket$ is easily seen to be definable in $\text{PCF}_V + H$, and the model is fully abstract and complete at type $\bar{2}$ even for PCF_V . Thus, the category of SR functionals coincides with the term model for $\text{PCF}_V + H$, and this gives us another mathematical handle on this class of functionals.

In this case, the approach via universal types offers the only known reasonable route to the full abstraction and completeness results. The universality of type $\bar{2}$ is also used in [Lo02] to prove other results: in particular, that various semantic characterizations of the SR functionals

do indeed coincide. The idea is that it suffices to show that the monoid of endomorphisms of the object $\bar{2}$ looks the same in various categories; from this it follows for general reasons that the categories coincide, at least as far as the PCF-denotable objects are concerned.

6.5. PCF with control

All the examples considered so far have involved functional languages in the sense that the corresponding category is well-pointed (albeit with a non-standard interpretation of the natural numbers in the case of PCF + choose). For these languages, this boils down to the fact that the *equational context lemma* holds: that is, two closed terms $M, M' : \sigma_1 \rightarrow \dots \rightarrow \sigma_r \rightarrow \gamma$ are observationally equivalent iff for all closed terms $P_i : \sigma_i$ and values $V : \gamma$ we have

$$MP_1 \dots P_r \Rightarrow V \text{ iff } M'P_1 \dots P_r \Rightarrow V.$$

In this subsection and the next, we consider two examples of languages that are not functional in this sense.

The first example is essentially the language SPCF of Cartwright and Felleisen [CF92]; it is equivalent (in the sense of having an equivalent term model) to the language μ PCF introduced in [OS97]. It will be convenient to consider a call-by-value version of this language. Let us add to PCF_V a constant

$$\text{catch} : ((\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat}) \rightarrow \text{nat}$$

whose operational behaviour may be described informally as follows. To compute $\text{catch } F$, we apply F to a dummy argument $g : \text{nat} \rightarrow \text{nat}$ and observe whether F ever interrogates g with an argument of type nat . If F first calls g with argument n , we “catch” this value and jump out of the computation of $F(g)$, returning $2n$ as the final result of $\text{catch } F$. If the computation of $F(g)$ completes successfully without any calls to g , returning the result m , we return $2m + 1$ as the result of $\text{catch } F$. It is easy to see how the catch operator may be implemented in a programming language with a control mechanism such as exceptions or continuations; indeed, it turns out that $\text{PCF}_V + \text{catch}$ serves well as a prototypical language for “functional programming plus control”.

We may formalize the operational semantics of this language in a manner inspired by an implementation of catch using exceptions. For each type σ and each $i \in \mathbb{N}$, let us add an auxiliary constant $e_\sigma^i : \text{nat} \rightarrow \sigma$; the idea is that $e_\sigma^i n$ plays the role of an exception carrying the value n . (We can imagine, if we like, that the same exception $e^i n$ can manifest itself as a term of any type σ as required; we write $e_\sigma^i n$ for $e^i n$ construed

as a term of type σ . The role of the superscript i is simply to ensure that we have an infinite supply of distinguishable exceptions.) We declare the constants e_σ^i to be values, so that they can be passed as parameters to functions. Terms of the form $e_\sigma^i n$ are not themselves declared to be values, but we modify our definition of the evaluation relation $M \Rightarrow U$ to allow U to be either a value V or a term $e_\sigma^i n$. We may now augment the rules for PCF_V with the following rules for catch:

$$\frac{F e_{nat}^i \Rightarrow e_{nat}^i n}{\text{catch } F \Rightarrow 2n} \quad (e^i \text{ not in } F) \quad \frac{F e_{nat}^i \Rightarrow m}{\text{catch } F \Rightarrow 2m + 1}$$

plus a host of other rules to handle the propagation of exceptions, for example:

$$\frac{M \Rightarrow e_{nat}^i n}{\text{iszero } M \Rightarrow e_{bool}^i n} \quad \frac{M \Rightarrow \lambda x.M' \quad N \Rightarrow e_\sigma^i n}{MN \Rightarrow e_\tau^i n} \quad \frac{M \Rightarrow e_{\sigma \rightarrow \tau}^i n}{MN \Rightarrow e_\tau^i n}$$

(With these examples in mind, the reader will easily be able to supply the remaining propagation rules.) In the first rule for catch, the side-condition that e^i does not appear in the term F ensures that the occurrence of e_{nat}^i on the right hand side of $F e_{nat}^i \Rightarrow e_{nat}^i n$ “arises from” the given occurrence on the left hand side in an obvious sense.

We now declare that we will only really be interested in terms of $\text{PCF}_V + \text{catch}$ — the exceptions are merely auxiliary machinery used to define the operational semantics of this language. Note that no term of $\text{PCF}_V + \text{catch}$ can ever evaluate to an exception.

To get a feel for this language, the reader might like to consider the terms

$$F_0 \equiv \lambda f.f 0 + f 1, \quad F_1 \equiv \lambda f.f 1 + f 0$$

(where we assume $+$ has already been defined), and to verify that $\text{catch } F_0 \Rightarrow 0$ but $\text{catch } F_1 \Rightarrow 2$. Thus, F_0 and F_1 are observably different, even though they define the same function $[\mathbb{N}_\perp \rightarrow \mathbb{N}_\perp] \rightarrow \mathbb{N}_\perp$. It follows that in any adequate model for $\text{PCF} + \text{catch}$, the denotational elements will need to be something more fine-grained than functions (at least if we wish to work with a standard interpretation of *nat*). It was shown by Cartwright, Curien and Felleisen [CCF94] that Berry and Curien’s category of *concrete data structures* (CDSs) and *sequential algorithms* provides an adequate model for $\text{PCF} + \text{catch}$. Indeed, it is shown in [CCF94] that the sequential algorithms model of $\text{PCF} + \text{catch}$ is fully abstract, and in [KCF93] that the *effective* sequential algorithms model is also complete. (These authors work with the call-by-name version, but the results hold equally for the call-by-value version. We will not

give a precise definition of sequential algorithms here, though the astute reader will be able to glean more or less what they must be from the facts mentioned below.)

The proofs of the above results in [CCF94, KCF93] work on traditional lines and proceed by showing definability of the finite elements of the model, by induction on types. This technique is rather onerous to apply in the case of $\text{PCF} + \text{catch}$, and the proof of full abstraction in [CCF94] runs to around 50 pages. It turns out, however, that these results can be proved much more cheaply by exploiting the fact that the type $\text{nat} \rightarrow \text{nat}$ is universal for $\text{PCF}_V + \text{catch}$.

Specifically, one first shows that the CDS $\llbracket (\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat} \rrbracket$ is a retract of $\llbracket \text{nat} \rightarrow \text{nat} \rrbracket$ in the (effective) sequential algorithms model. The key observation is that a sequential algorithm of type $(\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat}$ can be represented by a *decision tree*, which can itself be encoded by a function of type $\text{nat} \rightarrow \text{nat}$. The idea behind decision trees should be evident from the example shown in Figure 1. (In fact, sequential algorithms correspond precisely to *irredundant* decision trees — those in which the same question is never asked twice along any path through the tree.) For the encoding in $\text{nat} \rightarrow \text{nat}$, we note that a tree is essentially a partial function from *nodes* in the tree to *labels* on these nodes, which may be questions or answers. A node may be specified by a finite list of natural numbers giving the path to this node; a label is specified by a natural number together with a question/answer tag. It is easy to see that both nodes and labels can be coded up by natural numbers, and hence that a decision tree can be represented by a function $\mathbb{N} \rightarrow \mathbb{N}_\perp$.

Likewise, a sequential algorithm of type $(\text{nat} \rightarrow \text{nat}) \rightarrow (\text{nat} \rightarrow \text{nat})$ can be given by an \mathbb{N} -indexed forest of such decision trees. By a minor tweak to the above encoding, such forests can also be represented by functions $\mathbb{N} \rightarrow \mathbb{N}_\perp$.

Secondly, we show that both halves of this retraction are definable by terms of $\text{PCF}_V + \text{catch}$; this is just a little exercise in programming in this language. One half of the retraction is simply the algorithm that plays a decision tree (represented by a function $f : \text{nat} \rightarrow \text{nat}$) against a function $g : \text{nat} \rightarrow \text{nat}$ — this can be done in pure PCF. The other half is an algorithm that extracts the decision tree for a given algorithm $F : (\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat}$; this makes essential use of *catch*. As in the example of Section 6.4, it then follows easily that all PCF types are definable retracts of $\text{nat} \rightarrow \text{nat}$.

Thirdly, it is a triviality that at type $\text{nat} \rightarrow \text{nat}$ the sequential algorithms model is fully abstract, and the effective sequential algorithms model is also complete. We are therefore in the situation of Section 4,

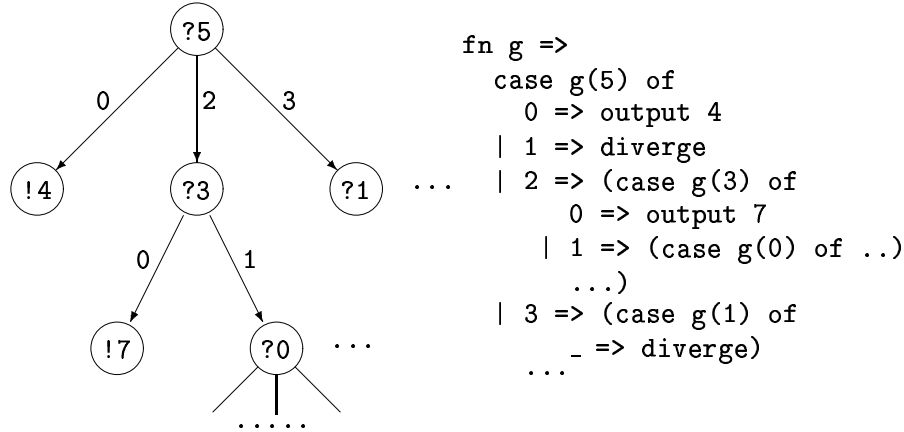


Figure 1.1. Part of a decision tree for a sequential strategy

and so the full abstraction and completeness results follow immediately. (More details of this proof can be found in [Lo02].)

Thus, in the case of $\text{PCF}_V + \text{catch}$, the existence of a universal type can be used to great advantage. Indeed, we are inclined to suggest that in this case it even offers a simpler route to the construction of the model in question. The usual definition of CDSs and sequential algorithms is rather cumbersome and requires a fair amount of setting up. Although this approach has its advantages — it is of independent interest to have a concrete description of the objects concerned — we would like to commend the following as an interesting alternative route. Let \mathcal{B} be the λ -algebra corresponding to the object $\llbracket \text{nat} \rightarrow \text{nat} \rrbracket$ above, and let \mathcal{B}_{eff} be its effective analogue. It is straightforward enough to construct these λ -algebras concretely from first principles: the underlying set is just the set of partial (recursive) functions $\mathbb{N} \rightarrow \mathbb{N}$, and λ -terms are interpreted via encodings of decision trees for the corresponding algorithms as indicated above. (This was first done in [vO99]; for further details see [Lo02].) By taking the Karoubi envelopes of these λ -algebras, we now obtain categories that contain at least the (effectively) sequential CDSs as a full subcategory. This construction gives us all we need in order to define the interpretation of $\text{PCF}_V + \text{catch}$ and prove adequacy; and as we have seen, full abstraction and completeness are then almost immediate.

6.6. PCF with control and state

In the previous example we saw how universal objects offered a simpler approach to some already known results. We now turn to an example

in which our approach leads to some genuinely new (and quite interesting) results. We will concentrate here very much on giving the relevant intuitions — a more formal account of this material will appear in a subsequent paper [Lo03].

We start by giving some background motivation. In the world of sequential algorithms that we have just considered, one takes account of the order of function calls (so that $\llbracket f\ 0 + f\ 1 \rrbracket \neq \llbracket f\ 1 + f\ 0 \rrbracket$) but not of the number of repetitions (so that $\llbracket f\ 0 + f\ 0 \rrbracket = \llbracket f\ 0 * 2 \rrbracket$). This is because sequential algorithms correspond to *irredundant* decision trees — we can skip all repetitions of questions since in this setting we know they will always receive the same answer as before. This corresponds to what one can observe in a language like PCF+catch, with control features (allowing us to distinguish different evaluation orders) but no state (which would be needed to detect a repeated function call). We would now like to tell an analogous story, but in a more fine-grained setting where we take account of both the order and number of function calls, in order to provide a suitable setting for modelling a language with both control features and state. (This way of thinking about control and state as separate “ingredients” of a programming language owes much to Abramsky’s work on game semantics, see e.g. [AM99].)

The idea, then, is to give a more refined version of the λ -algebra \mathcal{B} , embodying a more intensional notion of computation strategy. The Karoubi envelope of this algebra will then give something more fine-grained than sequential algorithms — let us call the resulting strategies *sequential processes*. We will then be able to give a programming language to match this model, and obtain full abstraction and completeness results.

As a preliminary intuition, let us consider the following ridiculously simple game G. Two participants, Opponent and Player, take turns to name a natural number, with Opponent starting. For example:

Opponent:	5	2	5	...
Player:	3	4	1	

At any stage, either participant may “give up”, in which case the game can proceed no further. The play may continue indefinitely. There is no notion of winning or losing.

It might be felt that this game is so boring that nothing of interest can conceivably be said about it. However, what is important about this game is that many much more interesting games can be coded up in it. For example:

- 1 Chess (Chinese or European). It is easy enough to code up possible moves in chess as natural numbers, and having fixed on a

coding convention, you and I could perfectly well play chess just by swapping natural numbers.

- 2 Any reasonable kind of interaction between two computational agents or coroutines, such as a function and its argument. Indeed, any situation at all in which two participants take turns to perform finitely describable actions can be regarded as a play of the game G .

We are therefore thinking of G as a kind of “generic” or “universal” game. The idea, then, is that by understanding this game really well, we will be able to get a handle on a very large and rich category of games.

A λ -algebra of strategies. We will now show how to build a λ -algebra inspired by the game G . The first step is to understand the nature of *strategies* for G . A strategy for Opponent may be given in an obvious way as a decision tree in which the nodes are labelled simply by natural numbers (with no distinction between questions and answers): the node labels correspond to Opponent moves, the edge labels to Player moves. A strategy for Player is similar (interchanging the roles of Player and Opponent), except that there is no root node, since the game starts with an Opponent move. (Alternatively, we may think of Player strategies as N -indexed forests of decision trees.)

Whereas in the construction of \mathcal{B} we coded up decision trees as functions $N \rightarrow N_{\perp}$, here we will take the trees themselves to be the underlying objects. Let \mathcal{S} be the set of Player strategies as described above; more formally, we may define \mathcal{S} coinductively as the final coalgebra for the domain equation

$$\mathcal{S} \cong N \rightarrow (N \times \mathcal{S})_{\perp}$$

in the category of sets. Given $S \in \mathcal{S}$ and α a finite list of natural numbers, we write $S|_{\alpha} \in \mathcal{S}$ for the subforest of S below node α .

We will take \mathcal{S} to be the underlying set of our λ -algebra: the idea is that Player strategies for G can also serve as strategies for all other games of interest. Note in particular that there is no irredundancy condition on strategies; indeed, if Opponent repeats a move, then Player may well decide to respond differently the second time (as in the little dialogue above).

In a programming language with suitable recursive datatypes, clearly we can represent strategies using a type σ of lazy trees. In Standard ML, for example, we may define:

```
datatype strategy = strategy of nat -> (nat * strategy)
```

where `nat` is a type for natural numbers. We may now ask: what does a strategy for an operation (or sequential process) of type $\sigma \rightarrow \sigma$ look like (say, in a language like ML)? We will first answer this informally by outlining the way in which a dialogue with such a strategy can proceed. Suppose $P : \sigma \rightarrow \sigma$ is some operation, $S : \sigma$ is some strategy, and we wish to obtain the resulting strategy $T = P(S)$. We will let Player play the role of P , and let Opponent play the role of T and also that of the opponent to T . The dialogue may then proceed as follows:

- Opponent starts by playing a move q_0 in the game G against T . That is, Opponent asks about the subtree on branch q_0 in the strategy T .
- Player *either* responds immediately with a move p_0 in G (that is, declares that the label on the node $[q_0]$ in T is p_0), *or* asks about the subtree on some branch m_0 in S .
- Assuming the latter, Opponent must reply by giving the label n_0 on the node $[m_0]$ in S .
- Player now has three options:
 - He may respond with a move p_0 in G .
 - He may ask about the subtree of S on some branch m_1 (possibly $m_1 = m_0$, in which case he is asking the “same” question as before).
 - He may ask about the subtree of S $|_{[m_0]}$ on some branch m_1 . Here the subforest S $|_{[m_0]}$ was “opened up” by the previous move and is now available for exploration by Player.

and so on.

In the case of the last possibility, the subforest S $|_{[m_0, m_1]}$ will also be opened up and made available for exploration by subsequent moves. Indeed, at any stage in the game, there will be some prefix-closed set of nodes α of S that we have visited so far, and a corresponding collection of subforests S $|_{\alpha}$ that are currently visible. (Also the forest S $|_{\epsilon} = S$ is already visible at the start of the game, and remains so.) At any stage in the game, Player must either play a move in G or else ask about some branch in one of the visible subforests. The idea should be clear if one bears in mind the possible ways in which an ML program of type `strategy -> strategy` can behave.

Actually, the above description is in need of a small correction. Suppose Player’s first move is to ask about branch m_0 of S , and his second

move is again to ask about branch m_0 of S . We then have, strictly speaking, two *copies* of $S|_{[m_0]}$ opened up, and for subsequent questions about this subforest, we can in principle ask which copy is being explored. Of course, if S is a “pure forest” (*i.e.*, an element of \mathcal{S}), it will make no difference, since we will always get the same answer from either copy. However, if the role of S is played by a process with some kind of internal state, we may well find we get different answers from different copies. So strictly speaking, we have at each stage a set of copies of subforests, or *subforest instances*, available for exploration. Note that each time Player asks a question about one of the visible copies, he opens up a new subforest instance (whether or not it has asked the same question before), and so adds exactly one to the stock of visible subforest instances. We may therefore identify these copies by *timestamps*: 0 for the given instance of \mathcal{S} itself, and $1, 2, 3, \dots$ for the new subforest instances in the order in which they become visible.

Typically, Player will continue exploring subforest instances of S until he has enough information to be able to respond to Opponent’s move q_0 in \mathbf{G} with a move p_0 , signifying that the node $[q_0]$ in T carries the label p_0 . Next, Opponent will play a move q_1 , asking about branch q_1 of the subforest $T|_{[q_0]}$. Again, Player may continue his exploration of subforest instances of S (all previously visited instances remaining visible), until he is able to respond with a move p_1 in \mathbf{G} , and so on. Of course, at any stage in the game, either participant may fail to supply a move (this corresponds to entering a non-terminating computation), in which case the game cannot proceed.

It is now clear how such a strategy can itself be represented by a decision tree. As before, we take labels on nodes to represent Player moves, and labels on edges to represent Opponent moves. Again, there is no root node, since Opponent starts. Each Player move is either an *answer* (*i.e.*, a natural number played in \mathbf{G}), or a *question* posed to some subforest instance of S (such a question may be specified by a timestamp identifying the subforest instance interrogated, together with a number giving the branch of this subforest that we are asking for). Thus, node labels (Player moves) may be taken to be elements of $\mathbf{N} + (\mathbf{N} \times \mathbf{N})$. Opponent moves, on the other hand, are either moves in \mathbf{G} (natural numbers) or answers to Player questions about node labels in S (again natural numbers). Which of these kinds any Opponent move is may be determined from the context: if the move follows a Player question, it must be a node label in S , otherwise it must be a move in \mathbf{G} . This means that no coding is necessary for Opponent moves, and we may take edge labels to be just elements of \mathbf{N} .

Using some bijective coding $\mathbb{N} \cong \mathbb{N} + (\mathbb{N} \times \mathbb{N})$, such a decision tree can easily be represented by an element of \mathcal{S} itself. Thus, strategies of type $\mathcal{S} \rightarrow \mathcal{S}$ may themselves be represented by elements of \mathcal{S} . In the other direction, given strategies R and S we may construe R as representing a strategy of type $\mathcal{S} \rightarrow \mathcal{S}$ and so “apply” R to S . (The only subtlety here is that we may at some point encounter an out-of-range timestamp, in which case we should diverge). Armed with the above ideas, it is easy to show along standard lines that \mathcal{S} is indeed a λ -algebra (the proof is very similar to that for \mathcal{B}). We also of course have the subalgebra \mathcal{S}_{eff} consisting of the recursive strategies. (A more formal definition of \mathcal{S} and its λ -algebra structure will be given in [Lo03].)

Categories of sequential processes. The Karoubi envelopes $\mathcal{K}(\mathcal{S})$ and $\mathcal{K}(\mathcal{S}_{eff})$ now give us good cartesian closed categories; we may call the morphisms of these categories (*effective*) *sequential processes*. It is easy to see that these categories provide adequate models for (both versions of) PCF; in fact, we shall consider them as models for PCF_N extended with an additional ground type corresponding to \mathcal{S} itself. Let PCF_N^* be the language obtained by adding to the definition of PCF_N a new ground type ζ representing \mathcal{S} (*not* designated as a program type), together with suitable constants and evaluation rules (the precise details are rendered slightly complicated by the absence of product types in our framework). Then we have an adequate interpretation $\llbracket - \rrbracket$ of PCF_N^* in $\mathcal{K}(\mathcal{S})$ such that $\llbracket \zeta \rrbracket = \mathcal{S}$ (more accurately, $\llbracket \zeta \rrbracket = id \in \mathcal{S}$); similarly for \mathcal{S}_{eff} . It is easy to see that both these interpretations are fully abstract at type ζ , and that the interpretation in $\mathcal{K}(\mathcal{S}_{eff})$ is complete at type ζ ; also that *nat*, *bool*, *unit* are PCF_N^* -definable retracts of ζ . The λ -algebra structure of $\mathcal{S}, \mathcal{S}_{eff}$ gives us a retraction $\llbracket \zeta \rightarrow \zeta \rrbracket \triangleleft \llbracket \zeta \rrbracket$, of which one half (the morphism *Apply*) is already definable in PCF_N^* .

Now let us add to PCF_N^* a constant

$$\text{lambda} : (\zeta \rightarrow \zeta) \rightarrow \zeta$$

and stipulate that $\llbracket \text{lambda} \rrbracket$ is the other half of the above retraction. It is then immediate that $\mathcal{K}(\mathcal{S}_{eff})$ is a complete model for $\text{PCF}_N^* + \text{lambda}$, and that both models are fully abstract in the sense that $\llbracket M \rrbracket = \llbracket N \rrbracket$ iff $\llbracket C[M] \rrbracket = \llbracket C[N] \rrbracket$ for all contexts $C[-]$ of program type. That is, the models would be fully abstract if we could give an operational semantics for $\text{PCF}_N^* + \text{lambda}$ with respect to which our interpretations are adequate.

This naturally raises the question: is there any reasonable operational semantics for $\text{PCF}_N^* + \text{lambda}$? As we have seen, *lambda* is an operation which extracts the computation strategy behind a given program of type

$\zeta \rightarrow \zeta$ — but is this operation “computable” in any natural sense? Fortunately, the answer is yes: in fact, lambda is programmable in a language with continuations and local state (such as New Jersey ML), so one can at least give an operational semantics for $\text{PCF}_N^* + \text{lambda}$ indirectly via a translation into such a language. We can then think of lambda as a primitive, and show that the above denotational semantics for $\text{PCF}_N^* + \text{lambda}$ is indeed adequate.

We have, in fact, implemented lambda in New Jersey ML as part of a program called STRATAGEM [STR], which may be downloaded from the author’s web page. This program allows the user to extract the computation strategy behind a program of any type, and to interactively explore and manipulate this strategy in a variety of interesting ways. The whole system is based on the universality of the type ζ , and serves as a showcase for many of the above ideas.

We therefore have, in principle, a programming language whose fully abstract and complete model is essentially $\mathcal{K}(\mathcal{S}_{eff})$. Still, lambda is hardly a natural language primitive from a programming point of view: it would be nice to have a more programmer-friendly characterization of a language that was equivalent in expressive power to $\text{PCF}_N^* + \text{lambda}$. In fact, it turns out that the language $\mathcal{L} = \text{“PCF} + \text{continuations} + \text{local state”}$, if appropriately defined, can both implement lambda and be adequately translated into $\text{PCF}_N^* + \text{lambda}$, so that the term models for these languages are equivalent (as far as the types of PCF_N^* are concerned). We therefore have that $\mathcal{K}(\mathcal{S}_{eff})$ is essentially the fully abstract, complete model of \mathcal{L} . The precise definition of \mathcal{L} , and the details of the syntactic translations between \mathcal{L} and $\text{PCF}_N^* + \text{lambda}$, will be presented in [Lo03].

To summarize, starting from an intuitive concept of computation strategy we have constructed a λ -algebra \mathcal{S}_{eff} , and hence a category $\mathcal{K}(\mathcal{S}_{eff})$, and then found a programming language \mathcal{L} to match it, guided by the existence of a universal type. Since \mathcal{S}_{eff} is mathematically a very tractable object, we therefore have a very good grasp of the term model for \mathcal{L} . In our view, this way of using models to inspire our choice of languages is a very significant part of the usefulness of denotational semantics: it helps us to identify languages for which observational equivalence and definability are simple to characterize, and which therefore lend themselves to the design of clean and principled program logics.

We emphasize that in this situation we are able to reap all the benefits of having a denotational semantics without ever giving a concrete description of the objects of $\mathcal{K}(\mathcal{S}_{eff})$. In fact, it turns out that this category is essentially the same as (*i.e.*, has a large full subcategory in common with) one of the categories of games described in [AM99],

though we only discovered this after obtaining the above results. Specifically, our category agrees with the intensional category of games and effective strategies without well-bracketing or innocence constraints; we therefore have that \mathcal{L} is a complete language for this category of games. Although the idea that this category corresponds to “PCF + control + state” is explicit in [AM99], no precise language matching this model was given there, and so this appears to be a new result.

Finally, we remark on a curious feature of this example: there is a universal type for $\text{PCF}_N^* + \text{lambda}$, but there is no universal *simple* type (that is, a type built up from *unit*, *bool*, *nat* using \rightarrow). So the addition of a type ζ to our language is quite essential for our purposes. This is perhaps surprising, because in the setting of weaker languages such as PCF (or even PCF + catch), we are accustomed to the idea that recursive datatypes such as ζ can be expressed as retracts of simple types. The explanation, of course, is that a pair of terms (F, G) defining a retraction $\zeta \triangleleft \sigma$ in PCF + catch need not yield a retraction in PCF + lambda: the observational equivalence in the latter language may be sufficiently fine-grained that we no longer have $G \circ F \approx id$.

7. Conclusions and further directions

We end by drawing together some conclusions from the discussion in this paper, and pointing out a few further ramifications of the ideas.

7.1. Full abstraction and completeness results

In this paper we have discussed an approach to denotational semantics which exploits the existence of universal types in many computational settings, and shown how this leads to full abstraction and completeness results for various programming languages. The whole approach is somewhat opportunistic, in that it takes advantage of universal types when these happen to exist; it is quite inapplicable in situations where there is no universal type (e.g. the language PCF). It could be objected that the existence of a universal type is a rather *ad hoc* property of a programming language; however, as our examples show, a remarkably large number of interesting programming languages do possess universal types, so our approach works well in a large number of cases. As we have seen, an approach via universal types is sometimes simpler than other approaches (as with PCF + catch), and sometimes harder (as with PCF⁺⁺) — though even in the latter cases it offers an interesting alternative route to the results. Moreover, we have seen that our approach can inspire interesting new results (as with PCF + choose, PCF + H and $\text{PCF}_N^* + \text{lambda}$).

Besides giving a method for proving full abstraction and completeness results, we have seen that universal types often give an attractive way of constructing the categories of interest in the first place. Our most dramatic example was the category \mathcal{S} , for which this was the *only* construction we gave — we were able to obtain all the results we wanted without ever giving a concrete description of the category. One might feel that this is cheating in some way — certainly, for a well-rounded understanding of the category, an explicit description of the objects and morphisms seems desirable — but it is nonetheless interesting to note that technically we can get away without it, and that it may be simpler to do so.

It is also perhaps of interest that our approach achieves some uniformity in its treatment of a diverse collection of instances: in each case we simply have to check the five conditions listed at the end of Section 4. From an expository point of view, our approach thus offers an efficient way of covering a wide range of scenarios in denotational semantics.

We have concentrated in this paper on cartesian closed categories and their connection with λ -algebras. However, as mentioned in Section 2, there are other correlations of this kind that one might consider. For instance, λ -algebras with a certain *least fixed point* property give rise to categories with initial algebras for “representable endofunctors”, and hence to models for inductive datatypes (this is a straightforward abstraction of the ideas in [Sc76]). All the examples of λ -algebras A that we have considered carry a well-behaved partial ordering, and moreover have the property that every representable function $A \rightarrow A$ has a least fixed point, so it would be straightforward to extend all our results to versions of our languages with inductive types. Another example is the correlation between *linear* λ -algebras and symmetric monoidal closed categories. It is easy enough to give a “linear” version of many of the ideas in this paper, and this is likely to play a useful role in the study of programming languages with a linear flavour (e.g. languages involving objects with state).

7.2. A notion of expressive equivalence

On a more conceptual level, the point of view we have advocated suggests one possible notion of when two programming languages (or models) are essentially *equivalent* in terms of their expressive power. The idea behind our whole approach is that two categories match up well iff they have the same Karoubi envelope. (The Karoubi envelope $\mathcal{K}(\mathcal{C})$ of a general category \mathcal{C} has as objects all morphisms $a : X \rightarrow X$ in \mathcal{C} such that $a^2 = a$, and as morphisms $a \rightarrow b$ all morphisms f in \mathcal{C}

such that $b \circ f \circ a = f$.) In this paper we have emphasized the case where one category is a term model for a programming language, and the other is the monoid given by a λ -algebra, but one can propose the same condition more generally as a notion of “expressive equivalence”.

Let us say that a general category \mathcal{C} is a λ -algebra iff $\mathcal{K}(\mathcal{C})$ is cartesian closed; it is then natural to say that \mathcal{C} and \mathcal{D} are *equivalent* as λ -algebras iff $\mathcal{K}(\mathcal{C})$ and $\mathcal{K}(\mathcal{D})$ are equivalent as categories. If both categories are syntactically constructed from programming languages, this gives us a reasonable notion of expressive equivalence. (For instance, the call-by-name and call-by-value variants of a programming language will typically be equivalent in this sense: see [Lo95, Chapter 6].) If one of the categories is semantically constructed, the equivalence amounts to a full abstraction and completeness result. As an example, we have that the following λ -algebras are equivalent:

- (The term model for) the language $\text{PCF}_V + \text{catch}$.
- (The term model for) its call-by-name analogue $\text{PCF}_N + \text{catch}$.
- (The monoid arising from) the untyped λ -algebra \mathcal{B}_{eff} .
- The category of effectively sequential CDSs and effective sequential algorithms.

(Note that this notion of “equivalence as λ -algebras” is stronger than the notion of “equivalence as partial combinatory algebras” introduced in [Lo99]. For example, our untyped structures \mathcal{B} and \mathcal{S} are equivalent as combinatory algebras, but not as λ -algebras.)

7.3. Program logics

The usefulness of universal types does not end with the proofs of full abstraction and completeness results. Indeed, in the author’s view, one of the chief reasons why these results are themselves interesting is that they offer a stepping-stone to the design of clean logics for reasoning about programs, and here again universal types have a valuable part to play.

Let us recall briefly the approach to program logics advocated e.g. in [LP97]. Let \mathcal{L} be a programming language as in Section 3 above, and consider a many-sorted first-order predicate logic $\mathbf{J}_{\mathcal{L}}$ given as follows:

- Sorts of $\mathbf{J}_{\mathcal{L}}$ are just types of \mathcal{L} .
- Terms of sort σ in $\mathbf{J}_{\mathcal{L}}$ are just terms of type σ in \mathcal{L} .
- For each type σ we have a binary relation symbol $=_{\sigma}$, and for each program type σ we have a unary relation symbol \Downarrow_{σ} .

- Formulae are built up as usual from atomic formulae using the connectives $\wedge, \vee, \Rightarrow, \forall, \exists$.

We would like an interpretation of this logic in terms of purely operational concepts, so that a programmer familiar with \mathcal{L} can understand what sentences of $\mathbf{J}_{\mathcal{L}}$ mean without knowing anything about denotational semantics. One such interpretation (for closed formulae of $\mathbf{J}_{\mathcal{L}}$) is the following:

- The relations $=$ and \Downarrow are interpreted as *observational equivalence* and *termination* respectively.
- The propositional connectives have their usual truth-table interpretation.
- Bound variables are understood as ranging over closed terms of \mathcal{L} : thus, a formula $\forall x : \sigma. \phi$ is interpreted as saying that $\phi[M/x]$ holds for all closed terms $M : \sigma$.

With this interpretation, the logic $\mathbf{J}_{\mathcal{L}}$ provides a powerful specification language for expressing correctness properties of programs, much in the spirit of Extended ML [KST97].

We would now like to design a sound and reasonably complete proof system for the logic with this interpretation. Clearly, the standard rules of classical logic will suffice as inference rules, but the problem is to find a good set of axioms which suffice for proving interesting properties of programs. In view of Gödel's Incompleteness Theorem, one cannot hope to provide a complete axiomatization, but one might at least hope for a *relative completeness* result, saying for instance that our axiom system was "as complete as Peano arithmetic".

Denotationally, our interpretation of $\mathbf{J}_{\mathcal{L}}$ obviously coincides with the usual Tarskian interpretation of first order logic over the naive set-theoretic model in which we take $\llbracket \sigma \rrbracket$ to be just the set of closed terms $M : \sigma$ modulo observational equivalence. Our goal is to provide a set of axioms that suffice to "pin down" what this structure is up to isomorphism (as well as the interpretation of particular terms within this structure). The hope is that a good fully abstract and complete model for \mathcal{L} should give us a more semantic characterization of this structure, giving us a better mathematical handle on it and hence yielding a good choice of axioms for achieving our goal.

We may now go a stage beyond the ideas in [LP97]. In view of the above caveat about incompleteness, we had better limit our ambitions to pinning down the structure under the assumption that the set $\llbracket nat \rrbracket$ (say) is *standard* — that is, canonically isomorphic to the usual set

N_{\perp} . Here, the existence of a universal type σ can be very useful: once we have given enough axioms to pin down the structure of $\llbracket \sigma \rrbracket$ (under the above assumption), and also to pin down the idempotents on $\llbracket \sigma \rrbracket$ corresponding to the retractions $\llbracket \tau \rrbracket \triangleleft \llbracket \sigma \rrbracket$, we have said enough to determine the model up to isomorphism, and our axiomatization will be relatively complete. Since the universal type is typically a relatively low type, it is generally not too hard to come up with axioms that suffice to pin down its structure.

The above approach to program logics forms the basis of a current project in which we are developing a sound and relatively complete proof system for some large fragments of ML [Prop]. In this project, we identify three sublanguages of New Jersey ML of increasing expressive power: a language \mathcal{L}_1 equivalent to PCF + H; a language \mathcal{L}_2 equivalent to PCF + catch; and a language \mathcal{L}_3 equivalent to PCF* + lambda. (The notion of equivalence here is the one described in Section 7.2; in fact, one can regard the languages \mathcal{L}_i simply as sugared versions of the corresponding “toy language”.) Our purpose is to provide program logics along the above lines for these languages, to encode these in the Isabelle theorem prover [Pa94], and to develop some support for verifying properties of programs in these logics. Although the languages \mathcal{L}_i are much more complex in their detailed description than the corresponding toy languages, the above equivalences mean that their term models are well understood, and so our method scales up successfully. (Indeed, a crucial ingredient in our methodology is the idea that these considerations should be used to help *determine* the choice of fragments of \mathcal{L}_i — we then know in advance that we are dealing with languages for which we have a good semantic understanding.) Moreover, in each of these cases there is a universal type, which enables us to come up with a simple but relatively complete axiom system in the manner outlined above.

Finally, we are hopeful that universal types may be of further assistance when we come to design proof tactics and decision algorithms for restricted fragments of the logics, since we can concentrate our attention on designing tactics and algorithms that work for the universal type. It will be interesting to see whether this is indeed the case.

References

- [AM99] Abramsky, S. and McCusker, G., Game semantics, In: H. Schwichtenberg and U. Berger (editors), *Computational Logic: Proceedings of the 1997 Marktoberdorf summer school*, Springer-Verlag, 1999, 1–56.

- [Ba84] Barendregt, H. P., *The Lambda Calculus: Its Syntax and Semantics*, North-Holland 1984.
- [BE91] Bucciarelli, A. and Ehrhard, T., Sequentiality and strong stability, In: *Proc. 6th Annual Symposium on Logic in Computer Science*, IEEE Computer Soc. Press, 1991, 138–145.
- [CCF94] Cartwright, R., Curien, P.-L. and Felleisen, M., Fully abstract semantics for observably sequential languages, *Information and Computation* **111**(2) (1994), 297–401.
- [CF92] Cartwright, R. and Felleisen, M., Observable sequentiality and full abstraction, In: *Proc. 19th Symp. Principles of Programming Languages*, ACM Press, 1992, 328–342.
- [HO00] Hyland, J. M. E. and Ong, C.-H. L., On full abstraction for PCF: I, II and III, *Information and Computation* **163** (2000), 285–408.
- [KCF93] Kanneganti, R., Cartwright, R. and Felleisen, M., SPCF: its model, calculus, and computational power, In: *Proc. REX Workshop on Semantics and Concurrency, Lecture Notes in Computer Science 666*, Springer-Verlag, 1993, 318–347.
- [KST97] Kahrs, S. and Sannella, D. and Tarlecki, A., The definition of Extended ML: a gentle introduction, *Theor. Comp. Sci.* **173** (1997).
- [K082] Koymans, K., Models of the lambda calculus, *Information and Control* **52** (1982), 306–332.
- [Lo95] Longley, J. R., *Realizability Toposes and Language Semantics*, PhD thesis, University of Edinburgh, 1995.
- [Lo99] —, Matching typed and untyped realizability, In: Proc. Workshop on Realizability, Trento, published as *Electronic Notes in Theoretical Computer Science* **23**(1), Elsevier, 1999.
- [Lo02] —, The sequentially realizable functionals, *Annals of Pure and Applied Logic* **117**(1) (2002), 1–93.
- [Lo03] —, A category of sequential processes, In preparation, 2003.
- [LP97] Longley, J. R. and Plotkin, G. D., Logical full abstraction and PCF, In: J. Ginzburg (editor), *Tbilisi Symposium on Language, Logic and Computation*, SiLLI/CSLI, 1997, 333–352.
- [OS97] Ong, C.-H. L. and Stewart, C. A., A Curry-Howard foundation for functional computation with control, In: *Proc. 24th Symp. on Principles of Programming Languages*, ACM Press, 1997, 215–227.
- [Pa94] Paulson, L. C., *Isabelle: A generic theorem prover*, Vol. 828 of *Lecture Notes in Computer Science*, Springer-Verlag, 1994.
- [P177] Plotkin, G. D., LCF considered as a programming language, *Theoretical Computer Science* **5** (1977), 223–255.

- [P178] —, T^ω as a universal domain, *Journal of Computer and System Sciences* **17** (1978), 209–236.
- [Prop] Fourman, M. P., Fleuriot, J. D. and Longley, J. R., A proof system for correct program development. Case for support for EPSRC grant GR/N64571, available from the author's home page, 2000.
- [Sa76] Sazonov, V. Yu., Degrees of parallelism in computations, In: *Mathematical Foundations of Computer Science 1976*, Vol. 45 of *Lecture Notes in Computer Science*, Springer-Verlag, 1976, 517–523.
- [Sc76] Scott, D. S., Data types as lattices, *SIAM Journal of Computing* **5** (1976), 522–587.
- [Sc80] —, Lambda calculus: some models, some philosophy, In: J. Barwise, H.J. Keisler and K. Kunen (editors), *The Kleene Symposium*, North-Holland, 1980, 223–266.
- [Sc93] —, A type-theoretical alternative to ISWIM, CUCH, OWHY, *Theoretical Computer Science* **121** (1993), 411–440. First written in 1969 and widely circulated in unpublished form since then.
- [Si90] Sieber, K., Relating full abstraction results for different programming languages, In: *Proc. 10th Conference on Foundations of Software Technology and Theoretical Computer Science, Bangalore*, Vol. 472 of *Lecture Notes in Computer Science*, Springer-Verlag, 1990.
- [STR] STRATAGEM, for SML of New Jersey, Source code and user documentation available from the author's home page, 2001.
- [vO99] van Oosten, J., A Combinatory Algebra for Sequential Functionals of Finite Type, In: S.B. Cooper and J.K. Truss (editors), *Models and Computability*, Cambridge University Press, 1999, 389–406.