Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 249 (2009) 117–134
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Some Programming Languages Suggested by Game Models (Extended Abstract)

John Longley[1]

LFCS, School of Informatics
University of Edinburgh
10 Crichton Street, Edinburgh, UK

Abstract

We consider a simple and well-known category of alternating games (also known as sequential data structures) and several categories derived from it. In each case, we present an extension of Plotkin's language FPC (or a suitable linearization thereof) which defines all computable strategies of appropriate types. The quest for such languages results in a novel selection of language primitives for state encapsulation, coroutining and backtracking.

Keywords: Game semantics, sequential data structures, definability, coroutines, backtracking, linear types, FPC

1 Introduction

Ever since Plotkin's classic paper [24], one of the principal concerns of denotational semantics has been to establish close connections between particular programming languages and particular mathematical models of computation — preferably models whose construction is independent of the syntax of the language in question. For example, one might prove that an interpretation $[\![-]\!]$ of a language $\mathcal{L}$ in a model $\mathcal{M}$ is *fully abstract*, or that all computable elements of $\mathcal{M}$ are *definable* by programs of $\mathcal{L}$. In general, the tighter the connection, the better the prospects for using $\mathcal{M}$ to prove facts about programs in $\mathcal{L}$ (see *e.g.* [19]).

As explained *e.g.* in [8], one can broadly distinguish two approaches to this enterprise: either we take a language as given and look for a mathematical model to fit it, or we take the model as given and look for a corresponding programming language. A possible motivation for the latter approach is that it offers the prospect

[1] Email: jrl@staffmail.ed.ac.uk

doi:10.1016/j.entcs.2009.07.087

of more mathematically based programming languages or language features: given a model $\mathcal{M}$ with a rich and pleasing mathematical structure, a language $\mathcal{L}$ that closely matches $\mathcal{M}$ can be expected to be particularly amenable to reasoning. Moreover, a definability result will assure us that we are getting "value for money" from $\mathcal{M}$, in the sense that *all* behaviours which $\mathcal{M}$ gives us the capacity to reason about are expressible in the language. (Already in [24] it was suggested that denotational semantics could be used in this way to identify language constructs "unintentionally left out by the language designer".)

Of course, in some cases this approach may lead to languages that look decidedly unnatural from a programming language point of view, even when the model $\mathcal{M}$ is mathematically a natural one — witness the language PCF+parallel-or+exists corresponding to the Scott/Ershov model of partial continuous functions [24], or (even worse) the language PCF+H corresponding to the sequentially realizable functionals [16]. However, the approach seems more promising in the case of models that represent a range of computational phenomena more in accord with realistic programming practice. Among the denotational models currently on the market, we suggest that *game models* offer a particularly attractive combination of rich mathematical structure and computational expressiveness. For a survey of the main ideas and results in game semantics, see *e.g.* [7].

In much of the game semantics literature to date, the language-driven (or feature-driven) perspective has been dominant: by and large, the goal has been to provide fully abstract models for a range of toy languages embodying familiar programming constructs. This programme has been very successful, as witnessed by the wide range of language features (exceptions, higher-order store, non-determinism, polymorphism, name generation *etc.*) that have been modelled using games. However, this success has often come at a price: in order to obtain a precise fit with a given language, some specialized machinery typically needs to be incorporated into the model construction (*e.g.* justification pointers, innocence, visibility, well-bracketing, permutation invariance), with the consequence that many game model constructions in the literature are technically quite elaborate. In the present paper, we offer some contributions to game semantics from a more model-driven perspective: starting from some models that appear simple and mathematically compelling in their own right, we ask what programming languages might correspond to them (in the sense of full abstraction and definability). These results are of some intrinsic interest in that they shed light on the precise level of computational power inherent in the models; but we also venture to suggest that some of the language primitives arising in this way may hold interest as a basis for some novel and useful programming language constructs, in particular for various forms of *state encapsulation*, *coroutining* and *backtracking*.

In fact, in this paper we shall concentrate entirely on categories derived from one of the simplest and most natural of all game models — the category of (negative) alternating games or *sequential data structures* as studied in [1,6]. This category, itself symmetric monoidal closed, can be endowed with a linear exponential (or '!' operator) in at least three natural ways, making it in each case a model of intuitionistic linear logic. Each of these in turn gives rise to a cartesian closed category in the usual way, so including the underlying category itself we have a total of seven models of interest. We summarize the construction and basic properties of these models in Section 2.

For each of the seven models, we will present a programming language with the corresponding computational power, formulated as an extension of the recursively typed functional language FPC (or an appropriate linearization thereof). In each case, we obtain definability and full abstraction results using the method explained in [17], exploiting the fact that our model possesses a *universal object* U with a particularly simple structure, denotable by a certain language type υ. Since definability and full abstraction at type υ are clear, it suffices to show that a small handful of types (such as $\upsilon{*}\upsilon$, $\upsilon \multimap \upsilon$, $!\upsilon$) are definable retracts of υ. In most cases, it was the requirement that these retractions be programmable that led us naturally to the identification of suitable language primitives. In Section 3 we introduce a suitable collection of primitives at a semantic level, and in Section 4 we use these to give complete languages for our models. In this extended abstract, we content ourselves with statements of the results and a high-level discussion of the proofs — further details will appear elsewhere.

The models studied in this paper form the semantic basis for our "Eriskay project" [20], in which we are seeking to take seriously the challenge of basing the design of a usable programming language on clean mathematical foundations. To date, we have completed a formal definition for a substantial sublanguage of Eriskay (called Lingay [18]), and provided an implementation based on a direct animation of the underlying game semantics. The language primitives described here do indeed feature in Lingay, and part of our purpose here is to explain where they came from. Note that we are concentrating here on the theory underlying the "quasi-functional" aspects of Lingay (broadly speaking, the fragment consisting of FPC-like types, albeit with non-functional operators at these types) — the semantic foundations for the specifically "object-oriented" aspects of the language (class implementations, inheritance, dynamic binding) are not treated in this paper.

I am grateful to the coordinators of the Mathematically Structured Functional Programming special session for the invitation to present this material, and to Jim Laird, Guy McCusker and Nicholas Wolverson for helpful and enjoyable discussions. Much of the research reported in this paper was carried out under the EPSRC project GR/T08791: "A programming language based on game semantics."

2 The game model and its exponentials

We give the definition of our game model in a form close to that of Curien [6], who was inspired by earlier unpublished work of Lamarche [15]. The model corresponds to the category of negative games and strategies as described in [1].

For any sets X, Y we write $\mathrm{Alt}(X, Y)$ for the set of finite sequences $z_0 \ldots z_n$ where $z_i \in X$ for i even, $z_i \in Y$ for i odd. If $L \subseteq \mathrm{Alt}(X, Y)$, we write L^{odd}, L^{even} for the sets of odd- and even-length sequences in L respectively.

Definition 2.1 (i) A *game* G consists of disjoint countable sets O_G, P_G of *opponent* and *player* moves, together with a non-empty prefix-closed set $L_G \subseteq \mathrm{Alt}(O_G, P_G)$ of *legal positions.*

(ii) A *(player) strategy* for G is a partial function $f : L_G^{odd} \rightharpoonup P_G$ such that $f(s) = y$ implies $sy \in L_G$, and $syx \in \mathrm{dom}\, f$ implies $f(s) = y$.

We write $X+Y$ for the set $\{(x,0) \mid x \in X\} \cup \{(y,1) \mid y \in Y\}$. Given any sequence s whose elements are pairs, we write s_i for the inverse image under $z \mapsto (z,i)$ of the subsequence of s consisting of elements (z,i).

Given games G, H, we may form games $G \otimes H$, $G \multimap H$, $G \& H$, $G \oslash H$ as follows:

$$O_{G\otimes H} = O_G + O_H \qquad P_{G\otimes H} = P_G + P_H$$
$$L_{G\otimes H} = \{s \in \mathrm{Alt}(O_{G\otimes H}, P_{G\otimes H}) \mid s_0 \in L_G, s_1 \in L_H\}$$

$$O_{G\multimap H} = P_G + O_H \qquad P_{G\multimap H} = O_G + P_H$$
$$L_{G\multimap H} = \{s \in \mathrm{Alt}(O_{G\multimap H}, P_{G\multimap H}) \mid s_0 \in L_G, s_1 \in L_H\}$$

$$O_{G\&H} = O_G + O_H \qquad P_{G\&H} = P_G + P_H$$
$$L_{G\&H} = \{s \in \mathrm{Alt}(O_{G\&H}, P_{G\&H}) \mid (s_0 \in L_G \wedge s_1 = \epsilon) \vee (s_1 \in L_G \wedge s_0 = \epsilon)\}$$

$$O_{G\oslash H} = O_G + O_H \qquad P_{G\oslash H} = P_G + P_H$$
$$L_{G\oslash H} = \{s \in L_{G\otimes H} \mid s = \epsilon \vee s \text{ starts with a move } (0,x)\}$$

We now obtain a category $\mathcal{G}$ as follows: objects are games, and morphisms $G \to H$ are strategies for $G \multimap H$. Identities and composition are defined as usual in game semantics (see *e.g.* [6] for definitions and further intuition). It is routine to check that $\mathcal{G}$ has the structure of a symmetric monoidal (in fact affine) closed category; note however that $\mathcal{G}$ does not contain diagonals $G \to G \otimes G$. In addition, & is actually a cartesian product operation on $\mathcal{G}$. The operator $\oslash$ is the *sequoidal product* introduced by Laird in [12], except that our $G \oslash H$ is his $H \oslash G$. Note that $G \oslash H$ is a retract of $G \otimes H$. Detailed proofs of all these facts are given in [26].

The strategies for any game G form a CPO under the inclusion ordering, and games themselves form a (large) CPO under the ordering

$$G \sqsubseteq H \ \text{ iff } \ O_G \subseteq O_H,\ P_G \subseteq P_H,\ L_G \subseteq L_H$$

These observations enable us to interpret recursion at the level of terms and types respectively in a familiar way (note in particular that $\multimap$ is covariantly monotone in both arguments).

We denote the empty game by 1. The following games N, N^ω and U will also play an important role:

$$\begin{array}{lll} O_N = \{?\} & P_N = \mathbb{N} & L_N = \{\epsilon, ?, ?0, ?1, \ldots\} \\ O_{N^\omega} = \mathbb{N} & P_{N^\omega} = \mathbb{N} & L_{N^\omega} = \{s \in \mathrm{Alt}(\mathbb{N},\mathbb{N}) \mid \mathrm{length}(s) \le 2\} \\ O_U = \mathbb{N} & P_U = \mathbb{N} & L_U = \mathrm{Alt}(\mathbb{N},\mathbb{N}) \end{array}$$

The game N will serve to interpret the type `nat` of natural numbers, and N^ω

the call-by-value type `nat -o nat` of functions on natural numbers that may be invoked just once. A trivial but crucial fact is that U is a *universal object* in $\mathcal{G}$, in the sense that every game $G \in \mathcal{G}$ is a retract of U. Later, we will also be exploiting the fact that U is isomorphic to the denotation of the (!-free) affine FPC type `rectype t => nat -o nat * t` (see Section 4).

The relatively simple definition of $\mathcal{G}$ thus gives rise to a rich and pleasing mathematical structure. Further (and deeper) evidence for the mathematical credentials of this model is provided by a striking result of Laird [13]: if the definition of strategy is modified to allow for a single non-recoverable error value $\top$, our model becomes equivalent to the (well-pointed) category of *locally Boolean domains* and affine bistable maps.

Next, we describe three ways in which $\mathcal{G}$ may be endowed with an operator '!', each embodying a different conception of what it means for a strategy to be "reusable". Each of these has appeared previously in the literature, though not all in the same place as far as we know. We point out in advance that each of our operators carries the categorical structure of a *linear exponential comonad* on $\mathcal{G}$ (see [25]), and moreover satisfies $!(G\&H) \cong !G \otimes !H$. As is now well known, this is sufficient to yield an interpretation of intuitionistic linear logic, and also implies that the co-Kleisli category $\mathcal{G}_!$ is cartesian closed. (Recall that $\mathcal{G}_!$-morphisms $G \to H$ are $\mathcal{G}$-morphisms $!G \to H$.) For our present purposes, a description of the action of each '!' on objects will be all that is required; we shall refer to the literature for definitions of the remaining structure and verifications of the requisite properties.

Our first '!' is the *non-repetitive backtracking* exponential, outlined in [1] and studied in detail in [6].

Definition 2.2 For any game G, we define $!_1 G$ as follows. Moves are given by

$$O_{!_1 G} \;=\; L_G^{even} \times O_G \qquad P_{!_1 G} \;=\; P_G$$

Legal positions in $!_1 G$ are sequences $s \in \mathrm{Alt}(O_{!_1 G}, P_{!_1 G})$ such that

(i) if an O-move (t, x) appears in s then $tx \in L_G^{odd}$, and if (t, x) is immediately followed in s by a P-move y then $txy \in L_G^{even}$;

(ii) if an O-move (txy, z) appears in s, then the adjacent move pair $(t, x)y$ must appear at some earlier point in s ;

(iii) no O-move (t, x) appears more than once in s.

The intuition is that the game G is here made reusable in the sense that at any point, Opponent may "backtrack" to a previously encountered position in G and play a new move so as to explore a fresh part of the game tree for G (repetitions of earlier moves are not admitted). In principle, one could get away with tagging an O-move x with the list of O-moves in t rather than the whole of t, but the above formulation simplifies the definition of $L_{!_1 G}$. For the associated structure and verification of the required properties, see [6], where it is also shown that the co-Kleisli category $\mathcal{G}_{!_1}$ is a full sub-CCC of the *sequential algorithms* model of [4].

Our second '!' may be described as a *repetitive* exponential without backtracking. It was introduced in [11], and is perhaps the most easily grasped of our three exponentials.

Definition 2.3 For any game G, we define $!_2G$ as follows. Moves are given by

$$O_{!_2G} \;=\; \mathbb{N} \times O_G \qquad P_{!_2G} \;=\; \mathbb{N} \times P_G$$

Legal positions in $!_2G$ are sequences $s \in \mathrm{Alt}(O_{!_2G}, P_{!_2G})$ such that

(i) for each $i \in \mathbb{N}$, $s_i \in L_G$;

(ii) if a move $(i+1, z)$ appears in s, then a move (i, x) appears at some earlier point in s.

The definition implies that any P-move carries the same index i as the preceding O-move (thus, we could in principle dispense with the P-indices). Intuitively, Opponent may at any point either play in an existing "copy" of G or choose to start a new play in a fresh copy (where copies must be activated in the order $0, 1, 2, \ldots$). By contrast with $!_1$, Opponent cannot backtrack to an arbitrary previous position — only to the start of the game G. However, a strategy for $!_2G$ may behave quite differently in different copies of G, suggesting that this '!' corresponds to some kind of stateful behaviour.

An economical but rigorous presentation of the structure and relevant properties of $!_2$ is given in [26]. A useful observation is that $!_2G \cong G \oslash !_2G$ (in fact, $!_2G$ is the minimal solution to this equation). As noted in [14], $!_2$ also enjoys a certain distinguished status as the *cofree commutative comonoid* on $\mathcal{G}$.

Our final '!' may be regarded as a *repetitive backtracking* exponential, combining the power of the previous two. It is tempting to think that such an exponential could be obtained simply by dropping condition 3 from the definition of $!_1$, but unfortunately the operator thus obtained is not even functorial. The correct definition is given by making each O-move in $!_3G$ refer not just to a previously arising play in G, but to a particular *occurrence* of such a play. This exponential was first explicitly presented in [10], along with its associated structure; an informal description of the corresponding co-Kleisli category also appeared in [17]. We here write $\mathbb{N}_1$ for the natural numbers excluding 0.

Definition 2.4 For any game G, we define $!_3G$ as follows. Moves are given by

$$O_{!_3G} \;=\; \mathbb{N} \times O_G \qquad P_{!_3G} \;=\; \mathbb{N}_1 \times P_G$$

Legal positions in $!_3G$ are finite sequences

$$s = (a_1, x_1)(b_1, y_1)(a_2, x_2)(b_2, y_2)\ldots \;\in \mathrm{Alt}(O_{!_3G}, P_{!_3G})$$

such that

(i) each $a_i < i$ and each $b_i = i$;

(ii) for each prefix s' of s, the *thread* $\theta(s')$ extracted from s' is a position in L_G. Here the mapping θ is defined by:
- $\theta(\epsilon) = \epsilon$;
- if s is even then $\theta(s(0, x)) = x$;
- if $s = (a_1, x_1)\ldots(b_r, y_r)$ and $0 < a \leq r$ then $\theta(s(a, x)) = \theta((a_1, x_1)\ldots(b_a, y_a))x$;

- if s is odd then $\theta(s(b,y)) = \theta(s)y$.

Thus, each O-move in a play of $!_3 G$ is either *initial* (annotated with 0) or points to an earlier P-move occurrence.

The mere existence of all these structures on $\mathcal{G}$ is in itself an indication of the mathematical fecundity of this model. However, the above definitions may appear rather mysterious, and the precise computational power embodied by the respective exponentials may not be very evident at this stage. The programming language characterizations to be given below will shed more light on this.

We will also employ one further categorical construction. It will be mildly more convenient to formulate our programming languages in a call-by-value style, since (for example) the mathematically natural universal types are denotable in such a framework, and several of our language primitives also look more natural in a call-by-value setting. We therefore invoke a general construction described in [3], yielding a category $\mathrm{Fam}(\mathcal{G})$ suitable for modelling call-by-value computation. We see the passage from $\mathcal{G}$ to $\mathrm{Fam}(\mathcal{G})$ as a mild convenience that expands the category to accommodate the full range of FPC types; however, from the point of view of computational power, it adds nothing essential and everything of interest can in principle be expressed in terms of $\mathcal{G}$ alone.

We summarize here what we will need to know about $\mathrm{Fam}(\mathcal{G})$. Objects are set-indexed families $(G_i \mid i \in I)$ of objects of $\mathcal{G}$, and morphisms $(G_i \mid i \in I) \to (H_j \mid j \in J)$ are functions mapping elements $i \in I$ to pairs $(j \in J, f \in \mathcal{G}(G_i, H_j))$. Both $\mathcal{G}$ and the category of sets embed fully in $\mathrm{Fam}(\mathcal{G})$ via the inclusions $G \mapsto (G \mid i \in \{*\})$ and $I \mapsto (1_i \mid i \in I)$ respectively; we shall often tacitly identify a game G or set I with its image in $\mathrm{Fam}(\mathcal{G})$, and also refer to sets as *ground objects* within $\mathrm{Fam}(\mathcal{G})$. Affine products $\otimes$ lift readily to $\mathrm{Fam}(\mathcal{G})$, and $\multimap$ on $\mathcal{G}$ yields a bifunctor $\multimap$: $\mathrm{Fam}(\mathcal{G})^{op} \times \mathcal{G} \to \mathcal{G}$. Moreover, $\mathrm{Fam}(\mathcal{G})$ has sums (denoted using $+$). We also have a *lift* functor $-_\perp : \mathrm{Fam}(\mathcal{G}) \to \mathcal{G}$, taking an object $(G_i \mid i \in I)$ to a game whose legal positions are all sequences $*is$ and prefixes thereof, where $*$ is a distinguished initial move, $i \in I$, and $s \in L_{G_i}$. In this context, we will call the O-move $*$ the *request* associated with the application of $-_\perp$, and the P-move i the corresponding *reply*. Clearly, each of our comonads on $\mathcal{G}$ lifts to one on $\mathrm{Fam}(\mathcal{G})$ via $!(G_i \mid i \in I) = (!G_i \mid i \in I)$; note that under this correspondence, the category $\mathrm{Fam}(\mathcal{G})_!$ coincides with $\mathrm{Fam}(\mathcal{G}_!)$.

Finally, we note that all the categories $\mathcal{C}$ we have introduced are *algebraically simple*, in the sense that they admit no proper algebraic quotients that do not collapse (say) the homset $\mathcal{C}(1, N)$. This is essentially because if f, f' are distinct strategies in a game G, it is easy to define a strategy for $G \multimap N$ which plays in G up to a point at which f and f' differ, and then exposes this difference in N. Thus, definability results for $\mathcal{C}$ entail full abstraction results in the following sense: given any language $\mathcal{L}$ with a compositional interpretation $[\![-]\!]$ in $\mathcal{C}$ such that every morphism in $\mathcal{C}([\![\sigma]\!], N)$ is definable by an $\mathcal{L}$-context $K[- : \sigma] : \mathtt{nat}$, then we have $[\![e]\!] = [\![e']\!]$ for any closed terms $e, e' : \sigma$ where $[\![K[e]]\!] = [\![K[e']]\!]$ for all such K. This condition captures the purely denotational aspect of full abstraction without reference to an operational semantics of $\mathcal{L}$. It follows that if $\mathcal{L}$ is endowed with *any* operational semantics for which $[\![-]\!]$ is adequate, then $[\![-]\!]$ will be fully abstract in the usual sense. Thus, we are justified in restricting attention henceforth to definability results.

3 State, coroutining and backtracking operators

In this section we now point out some strategies of particular interest that live in our various models, intuitively embodying various kinds of stateful, backtracking or coroutining behaviour. These will give us what we need in order to interpret the languages to be introduced in Section 4.

3.1 State operators

Even in $\mathcal{G}$ at !-free types, some mildly stateful behaviour may be observed. Consider for instance the following strategy $\mathit{share} : U \multimap U \otimes U$, which transforms a strategy for U into a pair of such strategies via a simple coding $\mathbb{N}+\mathbb{N} \cong \mathbb{N}$. We write inl, inr_1 and inr_2 for the injections on move sets corresponding to the three appearances of U in the above type.

$$\begin{array}{lcl} \mathit{share}(s.\mathrm{inr}_1(n)) & = & \mathrm{inl}(2n) \\ \mathit{share}(s.\mathrm{inr}_2(n)) & = & \mathrm{inl}(2n+1) \\ \mathit{share}(s.\mathrm{inr}_1(n).\mathrm{inl}(2n).\mathrm{inl}(m)) & = & \mathrm{inr}_1(m) \\ \mathit{share}(s.\mathrm{inr}_2(n).\mathrm{inl}(2n+1).\mathrm{inl}(m)) & = & \mathrm{inr}_2(m) \end{array}$$

One may think of this operation as "stateful" insofar as it would be naturally implemented by storing the argument of type U where it could be accessed by both sides of the result (bearing in mind that this argument is "non-copyable").

More interesting stateful behaviour arises in connection with $!_i$ where $i = 2, 3$. An easy but powerful observation is that for these exponentials (though not for $i = 1$) we have a canonical isomorphism $\mathit{flatten}_i : U \rightarrow !_i N^\omega$ (whence $!_2 N^\omega \cong !_3 N^\omega$). Thus, $\mathit{flatten}_i$ uses a non-reusable lazy forest to create a reusable function $\mathbb{N} \rightharpoonup \mathbb{N}$, which is stateful insofar as repetitions of the same function call may yield different results. This small piece of magic is a significant part of what makes these models mathematically pleasant to work with (we shall see this when we obtain definability results for the models involving $!_2$ and $!_3$).

In fact, *flatten* can be seen as arising from a more general state operator, which is most easily presented in the call-by-value setting. We first consider the situation for $!_2$. If S, X, Y are objects in $\mathrm{Fam}(\mathcal{G})$, we may ask whether we have a morphism

$$\mathit{encaps}_{2,S,X,Y} : \; S \;\otimes\; !_2((S \otimes X) \multimap (S \otimes Y)_\perp) \;\longrightarrow\; !_2(X \multimap Y_\perp)$$

which can be seen as creating a single-method "object" with encapsulated internal state of type S, given an initial state and a concrete method implementation. It turns out that such a morphism does indeed exist when X is a ground object in $\mathrm{Fam}(\mathcal{G})$ — a straightforward recursive definition may be given by exploiting the isomorphisms $!_2 G \cong G \oslash !_2 G$ in $\mathcal{G}$. (For full details and an axiomatization of the key properties of this morphism, see [26, Chapter 3], where the morphism is called *linthread*.) It is easy to see how $\mathit{flatten}_2$ may be obtained from encaps_2 (with $S = U$), and also how encaps_2 may be used to construct an integer store cell; thus, encaps_2 embodies a level of stateful behaviour intermediate between ground-type and full higher-order store.

A similar operator exists for $!_3$ (again for X a ground object), except that here it seems best to modify the type slightly:

$$\mathit{encaps}_{3,S,X,Y} : \; S \;\otimes\; !_3((S \otimes X) \multimap (S \otimes !_3 Y)_\perp) \;\longrightarrow\; !_3(X \multimap Y_\perp)$$

The extra $!_3$ on Y here allows for the possibility that Opponent might invoke the resulting function $!_3(X \multimap Y_\perp)$ and then exploit backtracking to re-use the resulting value of type Y (even if the type Y is not itself reusable). A construction analogous to that of encaps_2 may be used to define such an operator.

We digress briefly to mention a further generalization of *encaps*, which is not required for the main results of this paper but gives some additional insight into the relative computational power of our various models (and also furnishes another example of a semantically inspired language construct). One may ask whether operators $\mathit{encaps}_{i,S,X,Y}$ exist where X is a non-ground object (for $i = 2, 3$). In such a case, we had better insist that S is reusable (say $S = !_i S'$), since interaction with X might trigger "re-entrant" method calls resulting in multiple concurrent interactions with the state. Subject to this proviso, an encapsulation operator does exist, but it only works well in conjunction with strategies g for $!_i((S \otimes X) \multimap (S \otimes Y)_\perp)$ satisfying the following *argument-safety* condition: after the reply to the initial '$\perp$' request, g never responds to a move in the right-hand copy of S with a move in X. The argument-safety condition permits certain kinds of interaction with higher-order arguments, but prevents the construction of a general higher-order store cell. The notion of argument-safety and the corresponding encapsulation operators are studied in detail in [26], where it is shown that such a condition can be statically enforced in a programming language via a suitable syntactic restriction.

3.2 Backtracking

Next, we observe some structure that is available for our two backtracking exponentials $!_1$ and $!_3$, but not for $!_2$. Given a game G and moves $x \in O_G$, $y \in P_G$, let us define G_{xy} to be the game with the same moves as G, and with $L_{G_{xy}} = \{s \mid xys \in L_G\}$. We may now note that, for $i = 1, 3$, there is a canonical isomorphism $(!_i G)_{xy} \cong !_i(G_{xy})$, where we identify xy with an initial move pair in $!_i G$ in the obvious way. Intuitively, in the presence of backtracking, the tree of possible plays in $!_i(G_{xy})$ arises as the subtree rooted at xy of the tree of possible plays in $!_i G$. (This does not hold for $!_2$, since the tree for $!_2 G$ branches only at its root.) Using this idea, for any object X of $\mathrm{Fam}(\mathcal{G})$ one may define a morphism

$$\mathit{force}_{i,X} : \; !(X_\perp) \;\longrightarrow (!X)_\perp$$

The operational intuition here is that an interaction with $X_\perp$ consists of two stages: forcing the "thunk" to yield a value of type X, and further interaction with this value. The above operator forces the thunk in a way which allows the result to be re-used, giving us a point in the computation to which we may later "backtrack".

3.3 Coroutining

We now describe an operator with a coroutining flavour, living in the basic model $\mathcal{G}$ without '!'. Operationally, this may be viewed as a linearly typed version of a *resumable exception* operator as found *e.g.* in Common Lisp — it allows us not only to jump out of a computation, but to jump back in at a later time and continue from where we left off.

Once again, it is easiest to work in Fam($\mathcal{G}$). Consider a strategy f of type $(X \multimap Y_\perp) \multimap (Z_0 \otimes Z_1)_\perp$, where X and Z_0 are ground objects, and (for simplicity) $Z_1 \in \mathcal{G}$. The initial O-move x in this game will be the unique request corresponding to the right-hand '$\perp$', and f will respond to this (if at all) in one of two ways:

- f may respond immediately with a reply y for the right-hand '$\perp$', carrying an element of Z_0. In this case, the subsequent behaviour of f in the subgame rooted at xy may be canonically identified with a strategy in $(X \multimap Y_\perp) \multimap Z_1$.
- f may choose to invoke the argument of type $X \multimap Y_\perp$. Specifically, f may play a request for the left-hand '$\perp$', carrying an element of X. In this case, the ongoing behaviour of f canonically yields a strategy in $Y \multimap (Z_0 \otimes Z_1)_\perp$.

Putting this together, we obtain a canonical morphism

$$\begin{aligned} \mathit{lincatchcont} : ((X \multimap Y_\perp) \multimap (Z_0 \otimes Z_1)_\perp) \;\longrightarrow \\ (Z_0 \otimes ((X \multimap Y_\perp) \multimap Z_1)) \;+\; (X \otimes (Y \multimap (Z_0 \otimes Z_1)_\perp) \end{aligned}$$

The name here means "linear catch-with-continue", suggesting its kinship with the Cartwright/Felleisen `catch` operator [5]. "Linear" means the $X \multimap Y_\perp$ argument is not reusable (*cf.* the `catchcont` operator in the current version of Lingay [18]).

Our operator is essentially equivalent in computational power to the morphism lfe^{-1} identified by Laird in [14]. Laird's categorical analysis of $\mathcal{G}$ is in a sense more fundamental than what we are attempting here, and his operator captures more succinctly than ours the quintessence of this kind of coroutining behaviour. However, Laird's operator does not live at an (affine) FPC-denotable type, whereas our concern here is to show how this level of computational power may be expressed within the realm of such types, and also perhaps to make a stronger connection with programming intuition.

Let us also comment briefly on the relationship with *callcc*, of which *lincatchcont* may seem reminiscent. It is possible to define strategies within $\mathcal{G}$ for higher order (linear versions of) *callcc*. However, it seems that these strategies do not suffice for defining *lincatchcont* without the further addition of some kind of higher-order store operator. It thus appears that it is *lincatchcont* rather than some version of *callcc* that naturally captures the computational strength of $\mathcal{G}$. We note also that *lincatchcont* enforces a "delimited" form of non-local control which pre-empts the kinds of runtime errors that sometimes arise from expired continuations in typical implementations of *callcc*. Finally, we have found that many people will admit that they find *lincatchcont* easier to grasp as a programming primitive than *callcc*.

More powerful operators of this kind are possible in the models $(\mathcal{G}, !_i)$. For instance, if $i = 1, 3$, a straightforward variation on the above construction leads to the definition of a morphism

$$\begin{aligned} \mathit{catchcont}_i : (!_i(X \multimap Y_\perp) \multimap (Z_0 \otimes Z_1)_\perp) \;\longrightarrow \\ (Z_0 \otimes (!_i(X \multimap Y_\perp) \multimap Z_1)) \;+ \\ (X \otimes (!_i Y \multimap !_i(X \multimap Y_\perp) \multimap (Z_0 \otimes Z_1)_\perp) \end{aligned}$$

This operator acts on a strategy f which may invoke its "throw operation" of type $X \multimap Y_\perp$ several times. This means that in the second summand of the result type, an additional argument of type $!_i(X \multimap Y_\perp)$ is needed so that after resuming interaction with f, we know what to do when f subsequently re-invokes the throw operation. The appearance of $!_i Y$ rather than Y in the second summand is explained by the fact that if the exponential in $!_i(X \multimap Y_\perp)$ allows backtracking, then f is also allowed to perform multiple explorations the result returned from the first throw invocation. (We also naturally obtain a morphism $\mathit{catchcont}_2$, but its type has Y in place of $!_i Y$ at this point.)

There is an important operational difference between $\mathit{catchcont}_1$ and $\mathit{catchcont}_3$. In the case of the non-repetitive $!_1$, if the given strategy f happens to arise as the denotation of a program which (operationally) invokes its throw twice with the same value v of type X, only the first of these invocations will show up in the strategy f itself. When we invoke the resume operation provided by $\mathit{catchcont}_1(f)$, it is therefore as if the supplied argument of type $!Y$ is "memoized" so that it can be re-used whenever the value v is again thrown; only for thrown values $v' \neq v$ will we refer to the supplied argument of type $!(X \multimap Y_\perp)$. By contrast, in the case of $!_3$, repeated throw invocations do show up in f, so that after resumption, later throw invocations are *always* referred to the $!(X \multimap Y_\perp)$ argument. To emphasize this distinction, we shall later refer to $\mathit{catchcont}_1$ by the name *memocatchcont*, and to $\mathit{catchcont}_3$ by the name *fullcatchcont*.

3.4 Dynamic selection of strategies

One final jigsaw piece is needed. We have seen how the *encaps* operators provide means to construct non-trivial strategies at types $!_2 G$, $!_3 G$ (that is, strategies that are not merely promotions of strategies for G), but we do not yet possess a way to do this for $!_1 G$. The following operator fills this gap.

The idea is that for any $X \in \mathrm{Fam}(\mathcal{G})$, we can give a strategy in $!_1(\mathbb{N} \multimap X_\perp)$ by first giving an $\mathbb{N}$-indexed family of possible such strategies, and then deciding which of these strategies to use in response to the first Opponent move played (which will be a request carrying a natural number). This mildly stateful behaviour may be embodied by a morphism

$$\mathit{switch}_X : \;(\mathbb{N} \multimap !_1(\mathbb{N} \multimap X_\perp)) \;\longrightarrow\; !_1(\mathbb{N} \multimap X_\perp)$$

which may readily be seen to exist in $(\mathcal{G}, !_1)$. Note that the number n supplied by the first Opponent move will initially be used for both of the $\mathbb{N}$ arguments in the type $\mathbb{N} \multimap !_1(\mathbb{N} \multimap X_\perp)$.

4 Languages for game models

4.1 The base languages

We now work towards defining programming languages that match each of our seven models, in the sense that they define all computable morphisms of appropriate type. As a base to work from, we use Plotkin's recursively typed language FPC and linearizations thereof: specifically, we use an *affine* variant AFPC for the underlying model $\mathcal{G}$; an *affine-exponential* variant AEFPC for the models $(\mathcal{G}, !)$; and the original (intuitionistic) language FPC for the co-Kleisli categories $\mathcal{G}_!$. The definitions of these languages and their semantic interpretation proceed along mostly standard lines, so we content ourselves here with a summary of the essential points. Since our results consider these languages from a purely denotational perspective, operational semantics will not be required.

We start with FPC itself, as it is familiar from the literature. Types τ of FPC may be generated by the following grammar, where α ranges over type variables:

$$\tau \;::=\; \alpha \mid \tau\texttt{x}\tau \mid \tau\texttt{+}\tau \mid \tau\texttt{->}\tau \mid \texttt{rectype}\ \alpha\texttt{=>}\tau$$

The syntax of terms e may be defined by the following grammar, where x ranges over variables and ρ over rectypes:

$$\begin{aligned} e \;::=\; & x \mid \texttt{(}e\texttt{,}e\texttt{)} \mid \texttt{let (}x\texttt{,}x'\texttt{)=}e\ \texttt{in}\ e \mid \texttt{inl}_{\tau,\tau'}\ e \mid \texttt{inr}_{\tau,\tau'}\ e \\ & \mid \texttt{case}\ e\ \texttt{of inl}\ x\texttt{=>}e\,\texttt{|}\,\texttt{inr}\ x'\texttt{=>}e \\ & \mid \texttt{fn}\ x\texttt{:}\tau\texttt{=>}e \mid e\,e \mid \texttt{fold}_\rho\ e \mid \texttt{unfold}\ e \mid \texttt{rec}\ x\texttt{:}\tau\texttt{=>}e \end{aligned}$$

The choice of unpairing construct here, and the inclusion of `rec` as a primitive, are intended to facilitate a uniform presentation of the ordinary and affine languages. Typing judgements are of the form $\Gamma \vdash e : \tau$, where contexts Γ are of the form $x_1 : \tau_1, \ldots, x_n : \tau_n$. Typing rules are given as in the literature (see *e.g.* [23]). Note in particular that *contraction* is freely available — that is, there is no restriction on the re-use of variables.

Next we describe the affine variant AFPC, intended for interpretation in $\mathcal{G}$. Here the type constructors `x`, `->` are replaced by their linear counterparts:

$$\tau \;::=\; \alpha \mid \tau\texttt{*}\tau \mid \tau\texttt{+}\tau \mid \tau\texttt{-o}\tau \mid \texttt{rectype}\ \alpha\texttt{=>}\tau$$

The grammar for terms may be taken to be identical to that of FPC, but the typing rules differ in three ways. First, the type constructors `x`, `->` are replaced by `*`, `-o` respectively throughout the rules for pairing, unpairing, abstraction and application. (The overloading of term constructs here should cause little confusion given that we are keeping the type constructors syntactically distinct.) Secondly, as usual in linear type systems, the treatment of contexts in multi-premise rules is set up to prevent re-use of variables, for example:

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma', x : \tau_1 \vdash e_1 : \tau' \qquad \Gamma', x : \tau_2 \vdash e_2 : \tau'}{\Gamma, \Gamma' \vdash \texttt{case}\ e\ \texttt{of inl}\ x\texttt{=>}e\ \ : \tau'}$$

where Γ, Γ' are required to be disjoint (contraction is not admitted). Thirdly, a `rec` expression is permitted only if it has no free variables.

The language AEFPC is obtained by extending AFPC with a type constructor $\texttt{!}\tau$ and term constructors `der` e and `prom` e. A type is *reusable* if it is of the form $\texttt{!}\tau$; a context $x_1 : \tau_1, \ldots, x_n : \tau_n$ is reusable if all the τ_i are reusable. The typing rules for `der` and `prom` are as follows, where v ranges over *values* (*i.e.* terms constructed using variables, pairing, injection, abstraction, folding and promotion).

$$\frac{\Gamma \vdash e : \texttt{!}\tau}{\Gamma \vdash \texttt{der}\ e : \tau} \qquad\qquad \frac{\Gamma \vdash v : \tau}{\Gamma \vdash \texttt{prom}\ v : \texttt{!}\tau}\ \Gamma \text{ reusable}$$

Contraction is admitted for variables of reusable type only; likewise, the restriction on `rec` expressions is that their free variables must be reusable.

We now turn to the interpretation of these languages in our models. The call-by-value interpretation of AFPC in $\mathcal{G}$ is straightforward. Each type τ denotes an object $[\![\,\tau\,]\!]_\Xi$ of Fam($\mathcal{G}$) relative to a valuation Ξ assigning objects of Fam($\mathcal{G}$) to type variables:

$$\begin{array}{rclcrcl} [\![\,\alpha\,]\!]_\Xi & = & \Xi(\alpha) & \qquad & [\![\,\tau\texttt{*}\tau'\,]\!]_\Xi & = & [\![\,\tau\,]\!]_\Xi \otimes [\![\,\tau'\,]\!]_\Xi \\ [\![\,\tau\texttt{+}\tau'\,]\!]_\Xi & = & [\![\,\tau\,]\!]_\Xi + [\![\,\tau'\,]\!]_\Xi & & [\![\,\tau\texttt{-o}\tau'\,]\!]_\Xi & = & [\![\,\tau\,]\!]_\Xi \multimap ([\![\,\tau'\,]\!]_\Xi)_\perp \end{array}$$

$$[\![\,\texttt{rectype}\ \alpha\texttt{=>}\tau\,]\!]_\Xi \ = \ \mathsf{Fix}\,(X \mapsto [\![\,\tau\,]\!]_{\Xi,\alpha\mapsto X})$$

(In the last clause we appeal to the CPO structure on the class of games mentioned in Section 2.) Typing judgements $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau$ are then interpreted by morphisms $[\![\,\tau_1\,]\!] \otimes \cdots \otimes [\![\,\tau_n\,]\!] \to [\![\,\tau\,]\!]_\perp$ in Fam($\mathcal{G}$) along standard lines (*cf.* [2]), paying due attention to issues of linearity. Furthermore, for each of our models $(\mathcal{G}, !_i)$, the above interpretation may be readily extended to AEFPC, by defining $[\![\,\texttt{!}\tau\,]\!]_\Xi = !_i\,[\![\,\tau\,]\!]_\Xi$ and using the comonad structure to interpret `der` and `prom`.

We would also like to define a call-by-value interpretation of FPC in the categories Fam($\mathcal{G}_{!_i}$). However, there is a snag: in general, the lifting monad on $\mathcal{G}$ need not itself "lift" to a monad on $\mathcal{G}_!$. What is needed in order for it to do so is a distributivity law $!\perp \to \perp!$ (*cf.* [10]) — in fact, none other than our *force* operator! Thus, a call-by-value interpretation of FPC in Fam($\mathcal{G}_{!_i}$) works for $i = 1, 3$ but not for $i = 2$. We will sidestep this obstacle by contenting ourselves with a *call-by-name* interpretation when we come to consider the language for $\mathcal{G}_{!_2}$ (see Theorem 4.6) — such an interpretation may be straightforwardly given along the lines of [23]. In what follows, the notation $[\![\,-\,]\!]$ will refer to the call-by-value interpretation unless otherwise stated.

4.2 Definability results

We now come to our main results. We say a language $\mathcal{L}$ is *complete for* a game model $\mathcal{C}$ if all computable strategies in $\mathcal{C}$ at $\mathcal{L}$-denotable types are definable in $\mathcal{L}$.

We shall present extensions of the above languages that are complete for each of our models (whence, by the discussion at the end of Section 2, the models are also fully abstract for these languages). We give here only a high-level route map of the proofs, omitting many interesting details. A source file containing Lingay implementations for most of the programs here claimed to exist is available online [21].

In AFPC and AEFPC, we define the following useful types:

$$\texttt{empty} = \texttt{rectype}\ \alpha\texttt{=>}\alpha \qquad \texttt{unit} = \texttt{void -o void}$$

$$\texttt{nat} = \texttt{rectype}\ \alpha\texttt{=>unit} + \alpha \qquad \texttt{univ} = \texttt{rectype}\ \alpha\texttt{=>nat-o}(\texttt{nat*}\alpha)$$

We use γ, δ to range over ground types, *i.e.* those that may be constructed from `unit` without use of `-o`.

Consider first the interpretation of AFPC in $\mathrm{Fam}(\mathcal{G})$. It is easy to check that $[\![\, \texttt{univ} \,]\!] \cong U$, and that all partial computable functions on $\mathbb{N}$, and hence all computable strategies in $[\![\, \texttt{univ} \,]\!]$, are definable in AFPC. For general reasons (see [17]), we will have definability at all types as soon as each of $[\![\, \texttt{univ + univ} \,]\!]$, $[\![\, \texttt{univ * univ} \,]\!]$, $[\![\, \texttt{univ -o univ} \,]\!]$ is a programmable retract of $[\![\, \texttt{univ} \,]\!]$. For '`+`' this is already true in AFPC, while for '`*`' it clearly becomes true if we add a constant `share : univ -o univ*univ` to the language. (We adopt the evident convention that a named language primitive is to be interpreted using the semantic operation of the same name introduced in Section 3.) For `-o`, there is a semantically evident retraction $[\![\, \texttt{univ -o univ} \,]\!] \lhd [\![\, \texttt{univ} \,]\!]$, of which the "application" half is definable in AFPC, while the "abstraction" half is non-trivial and naturally calls for the operator

$$\texttt{lincatchcont}_{\gamma,\sigma,\delta,\tau} : \ ((\gamma\texttt{-o}\sigma)\texttt{-o}(\delta\texttt{*}\tau))\ \texttt{-o}\ (\delta\texttt{*}((\gamma\texttt{-o}\sigma)\texttt{-o}\tau)\ \texttt{+}\ (\gamma\texttt{*}(\sigma\texttt{-o}(\delta\texttt{*}\tau))))$$

We thus have:

Theorem 4.1 *The language* AFPC + `lincatchcont` + `share` *is complete for* $\mathcal{G}$.

One might wonder whether the "stateful" strategy *share* could itself be defined from a more primitive operator in $\mathcal{G}$ with more of the feel of a natural programming construct, but this seems to us to be more trouble than it is worth. We note that `share` is not needed for definability at types not involving `*`.

Next, consider the interpretations of AEFPC in our models $(\mathcal{G}, !_i)$. In each case, all that we require in addition to the above is that some retraction $[\![\, \texttt{!univ} \,]\!] \lhd [\![\, \texttt{univ} \,]\!]$ is programmable. The case of $!_1$ is perhaps the most delicate. A suitable section $[\![\, \texttt{!univ} \,]\!] \to [\![\, \texttt{univ} \,]\!]$ may be coded up using the operator

$$\texttt{force}_\tau : \ \texttt{!}(\texttt{unit -o}\ \tau)\ \texttt{-o !}\tau$$

(or equivalently by admitting terms `prom` e where e need not be a value), while the corresponding retraction may be implemented using the primitive

$$\texttt{switch} : \ (\texttt{nat -o !}(\texttt{nat -o nat*univ}))\ \texttt{-o !}(\texttt{nat -o nat*univ})$$

together with another use of `share`. We thus have:

Theorem 4.2 *The language* AEFPC + `lincatchcont` + `force` + `share` + `switch` *is complete for* $(\mathcal{G}, !_1)$.

For both $!_2$ and $!_3$, we can exploit the isomorphism $[\![\,\texttt{univ}\,]\!] \cong [\![\,\texttt{!(nat -o nat)}\,]\!]$, which becomes programmable once we add a language primitive

$$\texttt{encaps}_{\sigma,\gamma,\delta} : \ \sigma \ \texttt{*} \ \texttt{!}(\sigma\texttt{*}\gamma \ \texttt{-o} \ \sigma\texttt{*}\delta) \ \texttt{-o} \ \texttt{!}(\gamma\texttt{-o}\delta)$$

(The distinction between *encaps*$_2$ and *encaps*$_3$ is immaterial at such types, although of course we can add more general forms of $\texttt{encaps}_2$, $\texttt{encaps}_3$ to our languages if we wish.) In the case of $!_2$, this isomorphism, along with some uses of ground type store cells which may also be coded using `encaps`, is by itself sufficient to define a retraction $[\![\,\texttt{!univ}\,]\!] \lhd [\![\,\texttt{univ}\,]\!]$. Since *share* may be defined using `encaps`, we have

Theorem 4.3 *The language* AEFPC + `lincatchcont` + `encaps` *is complete for* $(\mathcal{G}, !_2)$.

Finally, in the case of $!_3$, the retraction $[\![\,\texttt{!univ}\,]\!] \lhd [\![\,\texttt{univ}\,]\!]$ is definable using `force` and `encaps` (an analogue of `switch` is not necessary here), so we have

Theorem 4.4 *The language* AEFPC+`lincatchcont`+`force`+`encaps` *is complete for* $(\mathcal{G}, !_3)$.

We next turn to FPC and the co-Kleisli models. Of course, the primitives identified above for $(\mathcal{G}, !_i)$ already suffice to define everything in $\mathcal{G}_i$, but the point is to find primitives of *intuitionistic* type that precisely represent the computational power available at such types.

Define `empty`, `unit`, `nat` as before, replacing `-o` by `->`. For our methodology to apply, we need to find a universal type `univ'` within the world of FPC types. Here we can simply replace `-o` by `->` in the definition of `univ`, or else use the following ingenious choice due to Laird [14]:

$$\texttt{univ'} \ = \ \texttt{nat -> (nat->nat) -> empty}$$

Remarkably, this gives $[\![\,\texttt{univ'}\,]\!] \cong U$ (canonically in the case of $!_2$ and $!_3$, or less canonically in the case of $!_1$). In either case, our principal task is to ensure that some retraction $[\![\,\texttt{univ -> univ}\,]\!] \lhd [\![\,\texttt{univ}\,]\!]$ is programmable, the retractions for `x` and `+` being straightforwardly definable in FPC.

In the case of $!_1$, the co-Kleisli model coincides with the world of sequential algorithms, and it is already known that the language FPC+`catch` is complete for this model [5]; however, an alternative solution, which fits better into the overall scheme we are presenting, is to introduce the language primitive

$$\begin{aligned}\texttt{memocatchcont}_{\gamma,\sigma,\delta,\tau} : \ & ((\gamma\texttt{->}\sigma)\texttt{->}(\delta\texttt{x}\tau)) \ \texttt{->} \\ & (\delta\,\texttt{x}\,((\gamma\texttt{->}\sigma)\texttt{->}\tau) \ \texttt{+} \ (\gamma\,\texttt{x}\,(\sigma\texttt{->}(\gamma\texttt{->}\sigma)\texttt{->}(\delta\texttt{x}\tau)))\end{aligned}$$

We then have

Theorem 4.5 *The language* FPC + `memocatchcont` *is complete for* $\mathcal{G}_{!_1}$.

A potential advantage of `memocatchcont` over `catch` here is that it promises a tighter, more "intensional" correspondence between strategies and operational behaviour: with `catch`, there are operations that cannot be computed without repeated function invocations at an operational level, whereas these can be avoided using `memocatchcont`. We leave this as a topic for future research.

For $!_2$, we may define an operator

$$\texttt{coroutine}_{\gamma,\delta,\gamma',\delta'} : \ ((\gamma \texttt{->} \delta) \texttt{ -> } \gamma') \texttt{ -> } (\gamma \texttt{ -> } (\delta \texttt{->} \gamma) \texttt{ -> } \delta') \texttt{ -> } \gamma' \texttt{+} \delta'$$

This pleasantly captures what is perhaps the most familiar conception of coroutining within a simply-typed framework. Informally, `coroutine` interleaves the execution of two processes, starting with the first, and transferring control between them whenever the "yield" operations of types γ`->`δ and δ`->`γ are invoked. The output of whichever process completes first is returned as the final result. One may easily define a game semantics for $\texttt{coroutine}_{\gamma,\delta,\gamma',\delta'}$ and $\texttt{coroutine}_{\delta,\gamma,\delta',\gamma'}$ by simultaneous recursion, using a suitable *catchcont* operator to pass between them; the differences between a call-by-value and call-by-name interpretation here are merely bureaucratic.

Theorem 4.6 *The language* FPC+`coroutine` *(with a call-by-name interpretation) is complete for* $\mathcal{G}_{!_2}$.

An essentially equivalent result appears in [14], formulated in terms of a lambda calculus with coroutine composition.

Finally, for $!_3$, we may introduce the primitive `fullcatchcont` with the same type as `memocatchcont`, and we then have:

Theorem 4.7 *The language* FPC + `fullcatchcont` *is complete for* $\mathcal{G}_{!_3}$.

5 Conclusion

The methodology of obtaining definability results via universal types has led us to a selection of primitive operators which shed light on the computational power of some natural mathematical models. As we have seen, this approach leads us to a choice of programming primitives related to, but not the same as, those usually encountered in existing languages: we are led to favour data encapsulation rather than store cells as the basic mechanism for stateful behaviour, and coroutining rather than first-class continuations as the basis for a flexible approach to control. Moreover, our operators exploit the potentialities of a linear type system to advantage, *e.g.* offering the programmer valuable runtime security guarantees in the case of `lincatchcont`. Finally, our semantic approach highlights certain *combinations* of these operators which may coexist safely (for instance, we can combine powerful control operators with certain higher order store constructs without losing runtime safety). We suggest that the primitives we have discussed might serve as a basis for the design of practically useful language constructs (we are pursuing this possibility in our ongoing work on Eriskay).

Our primary motivation for a model-driven approach to language design is the prospect of obtaining languages amenable to program verification. However, it would also be interesting to explore the possible programming applications of our primitives more extensively than we have done so far. One promising area of application is the construction of efficient "generic search" algorithms. For instance, consider the problem of counting the vectors $x \in \{0,1\}^n$ (for fixed n) satisfying a certain property P (which might arise, for instance, from a formula of propositional logic). In the interests of modularity and reusability, one would like to do this uniformly in P — that is, by means of a higher order program taking P as an input. One may improve on the naive approach by representing vectors x using the type `nat -> bool`, and properties P using `(nat -> bool) -> bool`, and using `catchcont` and `force` to implement a traversal of the search space which uses backtracking to avoid repeated computation, and which moreover takes care of 2^{n-r} vectors at a stroke if only r components of x are requested. Of course, this kind of algorithm could also be naturally implemented in a `callcc` style with reusable continuations, but we think our operators lead to more perspicuous implementations. For other algorithms of a similar flavour but involving *infinite* search spaces, see [9].

An obvious omission from this paper is the lack of any precise connection with operational semantics (although the executability of the game semantics means that this is already in some sense "operational"). Adequacy for a language with higher-order state encapsulation is proved in [26]. A traditional big-step semantics for `catchcont` and `force` is included in [18], although in these cases detailed adequacy proofs have yet to be carried out.

More also remains to be done on the mathematical side. For instance, the mathematical relationships between our various exponentials require further elucidation (see [22] for some relevant results). It would also be interesting to carry out a similar investigation of the computational power of the various exponentials in other natural game models; the selection in [22] offers a good starting point.

References

[1] Abramsky, S., and R. Jagadeesan, *Games and full completeness for multiplicative linear logic*. Technical report: DoC 92/94, Imperial College (1992). Journal version: J. Symb. Logic **59** (1994) 543–574.

[2] Abramsky, S., K. Honda, and G. McCusker, *A fully abstract game semantics for general references*, Proc. Logic in Computer Science 1998, 334–344,

[3] Abramsky, S., and G. McCusker, *Call-by-value games*, Lect. Notes in Comp. Sci. **1414** (1998), Springer, 1–17.

[4] Berry, G., and P.-L. Curien, *Sequential algorithms on concrete data structures,* Theor. Comp. Sci. **20** (1982) 265-321.

[5] Cartwright, R., P.-L. Curien and M. Felleisen, *Observable sequentiality and full abstraction*, Proc. POPL, 1992.

[6] Curien, P.-L., *On the symmetry of sequentiality*, Proc. 9th MFPS, Lect. Notes in Comp. Sci. **802** (1993), Springer, 29–71.

[7] Curien, P.-L., *Notes on game semantics*, course notes, 2006.

[8] Curien, P.-L., *Definability and full abstraction,* Electr. Notes in Theor. Comp. Sci. **172** (2007) 301–310.

[9] Escardó, M.H., *Infinite sets that admit fast exhaustive search*, Proc. Logic in Computer Science 2007, 443–452.

[10] Harmer, R., J.M.E. Hyland and P.-A. Melliès, *Categorical combinatorics for innocent strategies*, Proc. Logic in Computer Science 2007, 379–388.

[11] Hyland, J.M.E., *Game semantics*, in Semantics and Logics of Computation, CUP, 1997, 131–194.

[12] Laird, J., *A categorical semantics of higher-order store*, Proc. CTCS '02, Electr. Notes in Theor. Comp. Sci. **69** (2002).

[13] Laird, J., *Locally Boolean domains*, Theor. Comp. Sci. **342** (2005) 132–148.

[14] Laird, J., *Higher-order programs as coroutines: a semantic analysis*, to appear in Logical Methods in Computer Science.

[15] Lamarche, F., *Sequentiality, games and linear logic*, Proc. CLiCS workshop, Aarhus University, DAIMI-397-II, 1992.

[16] Longley, J., *The sequentially realizable functionals*, Ann. Pure Appl. Logic **117** (2002) 1–93.

[17] Longley, J., *Universal types and what they are good for*, Proc. 2nd Int. Symp. Domain Theory, Kluwer, 2003, 25–63.

[18] Longley, J., *Definition of the Lingay programming language (Version 0.2)*, Research report EDI-INF-RR-1283, University of Edinburgh, 2008.

[19] Longley, J., and G. Plotkin, *Logical full abstraction and PCF*, Tbilisi Symp. Logic, Language and Comp., CSLI, 1997, 333–352.

[20] Longley, J.R., and N. Wolverson, "Eriskay: a programming language based on game semantics", presented at GaLoP III, Budapest, 2008. For further information, see the Eriskay project website at `http://homepages.inf.ed.ac.uk/jrl/Eriskay`

[21] Longley, J.R., "Universal languages for certain game models". Lingay source fi le available at `http://homepages.inf.ed.ac.uk/jrl/Eriskay/gamelangs.er`

[22] Melliès, P.-A., *Comparing hierarchies of types in models of linear logic*, Inf. Comp. **189** (2004) 202–234.

[23] McCusker, G., *Games and full abstraction for FPC*, Inf. Comp. **160** (2000) 1–61.

[24] Plotkin, G.D., *LCF considered as a programming language*, Theor. Comp. Sci. **5** (1977) 223–255.

[25] Schalk, A., *What is a categorical model for Linear Logic?*, lecture notes, University of Manchester, 2004.

[26] Wolverson, N., "Game semantics for an object-oriented language," PhD thesis, University of Edinburgh, approved 2008.