Realizability models for sequential computation

Talk given at the 1998 APPSEM workshop,
Pisa, 16 September 1998
(Unfinished Draft!)

John Longley

Abstract

We give an overview of some recently discovered realizability models that embody notions of sequential computation, due mainly to Abramsky, Nickau, Ong, Streicher, van Oosten and the author. Some of these models give rise to fully abstract models of PCF; others give rise to the type structure of *sequentially realizable* functionals, also known as the *strongly stable* functionals of Bucciarelli and Ehrhard. Our purpose is to give an accessible introduction to this area of research, and to collect together in one place the definitions of these new models. We give some precise definitions, examples and statements of results, but no full proofs.

Preface

Over the last two years, researchers in various places (principally Abramsky, Nickau, Ong, Streicher, van Oosten and the present author) have come up with a number of new *realizability models* that embody some notion of "sequential" computation. Many of these give rise to fully abstract and universal models for PCF and related languages. Although the constructions of these various models have quite similar motivations, and the models themselves share many similar properties, some of the developments are so recent that they have not yet become generally known even among the handful of people working in the area. The purpose of this note is to bring together all these new models in one place, and to give a broad overview of the subject area and what it is trying to achieve.

In comparison to most of the other work presented at the Pisa workshop, the material discussed here is fairly theoretical in flavour, but I will suggest a few ways in which this work might have more practical repercussions. If any other workers whose background and interests are more practical than my own feel inclined to pursue any of these suggestions, I would be delighted to hear from them.

I would like to thank all the people mentioned above for explaining their ideas to me, and also Thomas Streicher for suggesting that I give the talk on which this note is based.

1 Introduction to realizability models

I will start with a crash course in realizability models for programming languages and some of the motivations for studying them. Fuller details may be found in my thesis [19] or that of Wesley Phoa [25].

The *vague idea* behind realizability semantics for programming languages is as follows. We start by choosing some model of data and computations on it which we think of as *primitive*. If we like, we can think of this primitive layer as a model for low-level or "machine level" computation. In some cases, this might look quite close to what actually goes on inside a machine; in other cases, it might correspond to a machine only in some quite abstract sense.

In any case, we then build a category of *datatypes* derived from this primitive model of computation. We can think of these as high-level or "programming language level" datatypes, and they will include types for natural numbers, functions, lists, streams and many other kinds of data familiar to functional programmers. In our model, these high-level datatypes will be related to the world of low-level computation as follows: each element x of a high-level datatype will have a non-empty set $\|x\|$ of low-level representations or *realizers*, and moreover every function between datatypes in the model will be *realized* by some low-level computation acting on these representations. This corresponds to the idea that, in any high-level programming language, all data values must somehow have representations at a lower machine level, and all run-time computations really happen at this machine level. We might have in mind the representation of data as sequences of bytes, or some intermediate level of description such as the representation of a functional program by closures.

Even in terms of the vague description just given, we may note two typical features of realizability models. Firstly, an element x of a datatype may have more than one realizer. This corresponds to the fact that the "same" value (for instance, the same function) may have several possible machine representations, the differences between them being irrelevant from the point of view of the high-level programmer. (Of course, there might be important differences between representations from the point of view of *efficiency*, but here we are considering only the relation between inputs and outputs.)

Secondly, the fact that we require every function in the high-level model to be realized by a low-level computation means that we obtain a category in which *all morphisms are computable* in the sense determined by the underlying machine level. We should clarify what we mean here by *computable*. In some instances, our low-level model may have some notion of effectivity built into it, and in this case, all morphisms in the high-level model may indeed be computable in some realistic sense. In other instances, the low-level model might admit many non-effective computations, in which case morphisms need not be computable in any genuinely effective sense. Nevertheless, we can still regard the morphisms as computable in an *abstract* sense: we think of the combinatory algebra as *defining* what we mean by computability, and proceed from there. Both the effective and non-effective kinds of realizability models turn out to be of interest for the study of programming languages.

1.1 Combinatory algebras

We now consider one particular class of models that embody the vague idea outlined above. Our primitive models of computation will be provided by the notion of a *combinatory algebra*.

Definition 1.1 *(i) A* combinatory algebra *consists of a set A together with a binary operation $\cdot : A \times A \to A$ (to be thought of as* application *and taken by convention to be left-associative), in which there exist elements $k, s, \bot$ such that for all $x, y, z \in A$ we have*

$$k \cdot x \cdot y \;=\; x, \qquad\qquad s \cdot x \cdot y \cdot z \;=\; (x \cdot z) \cdot (y \cdot z).$$

(ii) A combinatory algebra with bottom *(henceforth CA$\bot$) is a combinatory algebra A together with a distinguished element $\bot \in A$ such that for all $x \in A$ we have $\bot \cdot x = \bot$.*

First let us briefly recall some standard examples (we will not require their precise definitions). The Scott *graph model* $\mathcal{P}\omega$, consisting of the powerset of N with a certain continuous application operation, is a combinatory algebra, as is its *effective submodel* $\mathcal{P}\omega_{re}$ consisting of just the r.e. subsets of N. Both of these may be made into CA$\bot$s by taking as $\bot$ the empty subset of N. Another kind of example is given by term models for the untyped lambda calculus. Let Λ^0 be the set of closed untyped lambda terms and let T be any *sensible* lambda theory (i.e. one that equates all unsolvable terms). Then the quotient Λ^0/T has the structure of a combinatory algebra, which may be made into a CA$\bot$ by taking as $\bot$ the equivalence class of unsolvable terms. (All the above examples, and many others, are discussed in detail in the book by Barendregt [4].)

Experts will recognize that our notion of a CA$\bot$ is somewhat less general than the notion more often considered, namely that of a *partial combinatory algebra* equipped with an arbitrary *divergence* (see e.g. [19, 20]). However, the definition given above is slightly simpler to present, and will suffice for all the examples I will consider in this paper.

Any combinatory algebra can be seen as a model of *untyped* computation, in which *application* is taken to be the primitive operation. One can think of a combinatory algebra as a very fluid structure, consisting of a sea of elements which may be freely applied to one another and even to themselves. We will view a combinatory algebra as a kind of "abstract machine" on top of which we will implement various datatypes (this point of view has also been advocated by John Mitchell).

It is by no means inherent in the idea of realizability that the primitive model of computation should be untyped, and one can perfectly well construct realizability models over typed structures (such models have been considered e.g. in [1, 29]). Indeed, in some ways it would seem ideologically preferable to work with typed structures. However, there are good reasons why untyped notions of realizability are also worthy of study. One reason is that in the untyped setting there is a good "axiomatic" development of the theory of realizability models—the definition of a combinatory algebra is very simple, but it gives rise to models

with an amazingly rich mathematical structure. In the typed setting, it appears that one would have to work with much heavier axioms in order to obtain all the structure one would like. (It is hard, for instance, to see how to use typed realizability to model *impredicative* type systems without excessive complications in the definitions.) To summarize this point, in the untyped setting we get more out for what we put in.

There is another possible reason for interest in untyped structures, of a more philosophical nature. This rests on the idea that run-time computation really happens at an untyped level—the types are merely a kind of scaffolding which is thrown away at compile-time. I do not know whether a present-day compiler writer would find this idea credible, but it is an appealing picture. In any case, our main aim is not to model the real-world situation precisely, but to find natural and mathematically compelling structures with some associated computational intuition.

The simple if slightly mysterious definition of a combinatory algebra gives rise to a surprising amount of computational machinery. By the artifices of combinatory logic, one can construct, within any combinatory algebra A, representations of the booleans and natural numbers, pairing and projection operators, recursors and fixed-point operators. (Details may be found in Chapter 1 of [19].) In the case of the natural numbers, this means that any number n can be *coded* by an element $\bar{n}$ of A (in fact, there is a definition of $\bar{n}$ from k and s which works uniformly for any combinatory algebra). Moreover, any total recursive function $\varphi : \mathrm{N} \to \mathrm{N}$ is representable in A, in the following sense: there is an element $a \in A$ such that $a \cdot \overline{n} = \overline{\varphi(n)}$ for all $n \in \mathrm{N}$. Properties such as this give substance to the idea that any combinatory algebra provides a rich universe of untyped computation.

1.2 Partial equivalence relations

Given any combinatory algebra A one may take as a world of high-level datatypes the well-known category of partial equivalence relations on A:

Definition 1.2 *(i) A* partial equivalence relation *(or PER) on A is just an equivalence relation R on a subset of A.*

(ii) A morphism $f : R \to S$ *of PERs is a function f from R-equivalence classes to S-equivalence classes which is* realized *by some element $a \in A$: that is, for all R-equivalence classes r and all $b \in r$ we have $a \cdot b \in f(r)$.*

(iii) We write $\mathbf{PER}(A)$ *for the category of PERs on A and morphisms between them.*

The intuition here is that a PER R represents some high-level datatype; the equivalence classes of R correspond to values of the datatype, and R relates realizers at the primitive level that represent the same value. Furthermore, all morphisms between datatypes are "computable" in the sense that they are realized by elements of the combinatory algebra. It turns out that the category $\mathbf{PER}(A)$ (or, equivalently, the category of *modest sets* on A) has very pleasing properties—for instance, it is cartesian closed and regular—and it sits as a full

subcategory in an even better category, the *realizability topos* $\mathbf{RT}(A)$. (Again, more details of the construction and properties of these categories can be found in Chapter 1 of [19].) The idea behind the construction of $\mathbf{PER}(A)$ can be traced back to Kreisel [15], but appears more explicitly in Girard's thesis [12]. The toposes $\mathbf{RT}(A)$ were constructed by Hyland *et al.* in [13]. In this paper I will use the phrase "realizability model" as a general term covering categories such as $\mathbf{PER}(A)$ and $\mathbf{RT}(A)$.

The rich categorical structure of realizability models has been exploited to give models for call-by-name and call-by-value languages [19], dependent types [21], higher-order polymorphism, recursive types, and subtyping (see e.g. [6]).

1.3 Type structures in realizability models

In this note, I will concentrate on a very restricted class of datatypes, the *simple types* over the natural numbers with bottom. We will be looking at PERs in various categories that correspond to objects such as $N_\perp^{N_\perp^{N_\perp}}$ in domain theory; one is led to consider these objects when giving interpretations of simply-typed languages for partial computable functions, such as Plotkin's PCF [27]. Our language of types σ is given by the grammar

$$\sigma \quad ::= \quad 0 \mid \sigma_1 \to \sigma_2,$$

where 0 is a "ground type", to be thought of as the type of natural numbers. Of particular importance are the *pure types* $0, 1, 2, \ldots$, defined inductively by $n+1 = n \to 0$.

The following definition shows how we can interpret these types as PERs over any CA$\perp$:

Definition 1.3 *Let A be any CA$\perp$.*

(i) For each type σ, define a PER $\sim_\sigma$ on A as follows.

- *At ground type: $\overline{n} \sim_0 \overline{n}$, $\perp \sim_0 \perp$, and that's all.*
- *At function types: $f \sim_{\sigma\to\tau} g$ iff for all $x, y \in A$, $x \sim_\sigma y \;\Rightarrow\; f \cdot x \sim_\tau g \cdot y$.*

(ii) For each σ, let E_σ be the set of equivalence classes of $\sim_\sigma$, and for each σ, τ, let $\cdot_{\sigma\tau}$ be the induced application *operation $\cdot_{\sigma\tau} : E_{\sigma\to\tau} \times E_\sigma \to E_\tau$. We write $E(A)$ for the* type structure *consisting of the sets E_σ equipped with the operations $\cdot_{\sigma\tau}$.*

It is not hard to show that the PERs $\sim_\sigma$ are exactly the objects obtained from $\sim_0$ by exponentiation in $\mathbf{PER}(A)$. Note that E_0 may be canonically identified with $\mathrm{N}_\perp$, and $E_{\sigma\to\tau}$ may be identified with a set of functions from E_σ to E_τ. Such structures are sometimes called *partial* type structures, to distinguish them from *total* type structures in which E_0 is canonically identified with N.

For convenience, we have chosen to consider the hierarchy of PERs such as $N_\perp^{N_\perp^{N_\perp}}$—these are the objects that one would use in interpreting the *call-by-name* version of PCF. One might also consider PERs such as $N_\perp^{N^N}$, which would correspond to types of *call-by-value* PCF. For our purposes, however, the differences

between these variants are of minor importance, since the call-by-value objects (apart from N itself) can be recovered as syntactically definable *retracts* of the call-by-name objects and *vice versa*. This means that results of the kind we will be interested in, such as full abstraction and universality, transfer immediately from the call-by-name setting to the call-by-value setting, and conversely. (These ideas are worked out in detail in [19, Chapter 6].) For similar reasons, we could have included products or sums in our language of types, with no significant effect on the results we are concerned with.

The line of research we here describe starts from the observation that *different CA$\perp$s often give rise to non-isomorphic type structures.* This can happen for a number of reasons. For instance, a CA$\perp$ with no effectiveness built in, such as the graph model $\mathcal{P}\omega$, will not be expected to yield the same type structure as the effective analogue $\mathcal{P}\omega_{re}$. But there are other more interesting ways in which CA$\perp$s may yield different type structures. For example, the type structure $E(\mathcal{P}\omega_{re})$ is *not* isomorphic to $E(\Lambda^0/\mathcal{B})$ (where $\mathcal{B}$ is the theory that equates lambda terms iff they have the same Böhm tree), even though both CA$\perp$s are in some sense effective. In fact, these type structures do contain exactly the same functions of type $0 \to 0$ (essentially the partial recursive functions); but $E(\mathcal{P}\omega_{re})$ contains a function of type $0 \to 0 \to 0$ corresponding to "parallel-or", while $E(\Lambda^0/\mathcal{B})$ does not. The general idea that different CA$\perp$s could give different type structures in this way was first made explicit by Wesley Phoa [25, 26], to whom the above example is due.

Phoa's example calls to mind the difference between "sequential" and "parallel" computability in the context of PCF (see [27]). This connection was explored in [19, Chapter 7], where I argued that one should try to match up particular realizability models with particular programming languages by means of full abstraction and universality results. For instance, it follows easily from known results that the programming language PCF^{++} (= PCF extended with "parallel-or" and "exists" operators) has an interpretation in $\mathbf{PER}(\mathcal{P}\omega_{re})$ which is not only fully abstract but *universal*—that is, every element of each E_σ is the denotation of some closed term of PCF^{++}. (Note that universality implies full abstraction, since our models $\mathbf{PER}(A)$ are always well-pointed.) Put another way, the term model for PCF^{++}, consisting of closed terms modulo observational equivalence, is isomorphic to the type structure $E(\mathcal{P}\omega_{re})$. Thus, the model $\mathbf{PER}(\mathcal{P}\omega_{re})$ and the programming language PCF^{++} in some sense embody the same notion of (parallel) higher-type computability.

2 Realizability models for sequential computation

In view of results such as the above, it is natural to ask whether one can find realizability models that likewise embody a notion of *sequential* computability—for instance, that provide universal models for sequential PCF. The lambda term model mentioned above seems a promising candidate, and so we might hazard the following:

Conjecture 2.1 (Longley-Phoa) *For any sensible theory T, the interpretation of PCF in $\mathbf{PER}(\Lambda^0/T)$ is universal (hence fully abstract).*

This was implicitly conjectured by Phoa (for the case $T = \mathcal{B}$) in [26]. The general statement was formulated explicitly in [19, Chapter 7], where evidence for the conjecture was discussed and some partial results obtained. It is believed (by myself and a few other people) that the conjecture is very difficult, and that a proof would require some deep new insight into the untyped lambda calculus.

The importance of the Longley-Phoa Conjecture is hard to gauge. On the one hand, it appears to stand as rather an isolated problem and probably one of limited practical interest, given the apparent difficulty of proving anything at all about the model. I have therefore come to suspect that it may not be worth the effort it would take to prove it. On the other hand, the simplicity of the conjecture's statement and the fact that it links two well-studied languages—the untyped lambda calculus and PCF—give it a certain conceptual appeal, and the very difficulty of the problem is alluring. Besides, Thomas Ehrhard has recently pointed out to me that the conjecture is strongly reminiscent of questions arising in the work of Krivine on abstract machine implementations of System F, though this connection has not yet been explored.

Much of the difficulty of the Longley-Phoa Conjecture seems to stem from the "syntactic" nature of the model $\mathbf{PER}(\Lambda^0/T)$, and indeed, it is more than likely that this very feature would limit the usefulness of this model even if the conjecture were true. It therefore seems natural to ask whether we can construct other combinatory algebras—perhaps more semantic in nature than Λ^0/T—that embody some notion of sequentiality. This will be the subject of the rest of this paper.

2.1 Why more fully abstract models?

Of course, we already have other kinds of models for sequential computation. Around 1993, several constructions of fully abstract models for PCF were independently announced: three using game-theoretic ideas [2, 14, 22], and one using Kripke logical relations [23]. So one should at least pause to ask whether the world really needs any more fully abstract models of PCF. Of course, it is hard to answer questions of this kind definitively, but I would like to suggest some reasons why good realizability models of PCF may still be of particular interest. (Some of these are also reasons for studying realizability models in general.)

1. Firstly, as we have seen, realizability models offer a very rich categorical structure, and there are known techniques for modelling phenomena such as polymorphism and recursive types. If one found a good (e.g. universal) realizability model for PCF, one would know in advance, as it were, that all this machinery would be available. For other kinds of model, one might have to work harder in order to model all these additional language features.

2. Another distinctive feature of realizability models is that they have their own *internal logic*. In other words, realizability toposes can be seen as

"universes" within which one can perform various kinds of constructive "set-theoretic" reasoning. Although the precise practical significance of this fact is perhaps not yet clear, the ongoing work of Reus and Streicher [28] explores the possibility of using this logic as a basis for program verification.

3. Many of the combinatory algebras we shall describe are in some way inspired by the fully abstract models mentioned above; however, it sometimes turns out that the realizability models are technically simpler to construct than the models that inspired them. A good example is Abramsky's model of well-bracketed strategies described in Section 5. The corresponding realizability model closely resembles the games model of [2], but many of the rather technical conditions appearing in [2] become unnecessary in the realizability model because they actually come for free as *consequences* of quite simple definitions. This suggests that realizability models may sometimes be pedagogically useful as ways of simplifying the presentation of other known models.[1]

4. Even though we already have fully abstract models of PCF, it remains possible that a new *construction* of such a model, if sufficiently different from known constructions, would provide a new kind of handle on PCF and would offer fresh theoretical insight into the nature of sequentiality. For instance, although we understand the intensional games models for PCF, we still understand their extensional *quotient* rather poorly, and there is a chance that another model construction would shed new light on this.

5. Finally (and most speculatively), the idea of viewing combinatory algebras as abstract machines suggests that realizability models could perhaps be useful in providing inspiration to language implementors and compiler writers. The task of implementing a high-level language can be thought of in a highly idealized way as follows: one implements some "abstract machine" for run-time computation, plus a way of compiling high-level programs down to abstract machine programs. It was already pointed out in [25] that realizability based on lambda term models is at least vaguely reminiscent of the fact that functional languages are often implemented in terms of the lambda calculus or combinatory logic. But it seems at least conceivable that other kinds of abstract machine would serve just as well, and it is possible that new combinatory algebras for sequential computation could suggest alternative ways of implementing high-level languages.

When I wrote [19], I did not have any examples of good combinatory algebras for sequential computation (apart from the lambda term models mentioned above). Over the last couple of years, however, several interesting combinatory algebras have been found that do indeed have a "sequential" flavour. These

[1] We have already hinted at the idea that with untyped realizability models one frequently gets a lot out for what one puts in. It is worth mentioning another example of this phenomenon: The partial combinatory algebra K_1 gives rise to the same type structure as the effective Scott model (see e.g. [19]). The construction of the Scott model is based on *continuity*, to which effectivity is then added on, while the realizability model is constructed using only *effectivity*, and continuity follows as a consequence.

have been discovered mostly independently by workers in various places, and one purpose of the present paper is to collect together the definitions of these new combinatory algebras.

These recent models have also yielded a welcome surprise. Whilst some of them do indeed give rise to exactly the PCF-computable functionals, others give rise to a quite different—and larger—class of "sequentially computable" functionals. This class is closely related to the *strongly stable* functionals of Bucciarelli and Ehrhard [8]. It was already implicit in the work of Ehrhard [10] that these functionals were in some sense computable by sequential algorithms, but the realizability models have shed new light on this class of functionals and have provided fresh evidence that it really is a mathematically natural class of higher-type computable functions. This seems to me to be a very exciting development, and I will touch briefly on this "alternative" to the PCF notion of sequentiality in Section 4 below.

3 A decision tree model (van Oosten)

We start by describing a combinatory algebra $\mathcal{B}$ based on decision trees, discovered by van Oosten late in 1996, and independently by the present author early in 1997. The first written account of this model appeared in [24]; the description we give here is taken from our draft paper [17], where further details may be found.

Let N be the set of natural numbers (including 0), and let $\mathrm{N}_\perp = \mathrm{N} \sqcup \{\perp\}$. We will identify partial functions $\mathrm{N} \rightharpoonup \mathrm{N}$ with total functions $\mathrm{N} \to \mathrm{N}_\perp$; we write $\mathrm{N}_\perp^\mathrm{N}$ for the set of such functions.

As a first step, let us consider informally the possible behaviours of an algorithm or strategy σ for computing a partial function $F : \mathrm{N}_\perp^\mathrm{N} \rightharpoonup \mathrm{N}$ in a "sequential" way. Suppose σ is presented with a function $g : \mathrm{N} \to \mathrm{N}_\perp$ as an argument. There are three possibilities: Firstly, σ might simply diverge, in which case F is the everywhere undefined function. Secondly, σ might return a natural number straightaway, in which case F is a constant function. Thirdly, σ might ask for the value of g on some argument n, say. If $g(n)$ is undefined, then σ can do nothing; however, if $g(n)$ returns an answer m, then the subsequent behaviour of σ may be conditional on m. Indeed, for each value of m we again have the above three possibilities for the behaviour of σ, and so on.

We may represent the behaviour of such a strategy σ as a (finite or infinite) *decision tree*. This consists of a set of nodes, each carrying a label which may be either a *question* $?n$ or an *answer* $!n$ ($n \in \mathrm{N}$). At each node, the children are indexed by (a subset of the) natural numbers. The way in which such a decision tree represents a strategy should be clear from Figure 1, which shows part of a decision tree together with pseudocode for the corresponding part of the strategy.

Formally, we may identify the nodes of a decision tree with finite sequences of natural numbers: the root node is the empty sequence ϵ, and the child m of the node $[m_1, \ldots, m_k]$ is the node $[m_1, \ldots, m_k, m]$. (This is the notation we will use for displaying finite sequences.) We write Seq(N) for the set of finite sequences of natural numbers, and use $\alpha, \beta, \ldots$ to range over Seq(N). (Note that Roman

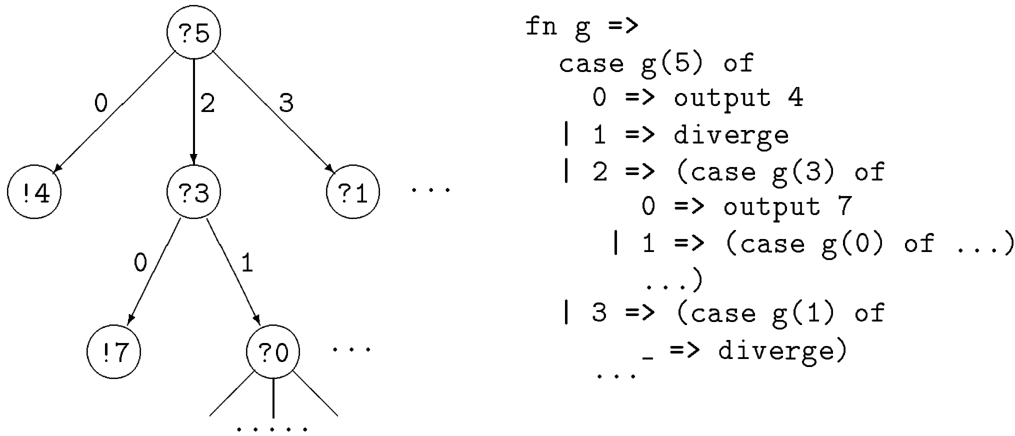

Figure 1: Part of a decision tree for a sequential strategy

letters will stand for natural numbers and Greek letters for sequences.) We write $m;\alpha$ or $\alpha;m$ for the result of adjoining a new element at the beginning or end of a sequence, and $\alpha;\beta$ for concatenation of sequences.

Likewise, we may identify the labels in a decision tree with elements of $\mathrm{N}+\mathrm{N}$, where the left and right summands correspond to the tags ? and ! respectively. This leads us to the following definition:

Definition 3.1 *A* decision tree *is formally a partial function* $\sigma : \mathrm{Seq}(\mathrm{N}) \to \mathrm{N}+\mathrm{N}$*. The domain of* σ *is the set of* nodes *in the tree.*

One might ask whether one ought to impose hygiene conditions on σ: for example, that the set of nodes of the tree is prefix-closed, or that labels $!n$ may appear only on leaves of the tree. In fact such conditions will not be necessary, although trees that do not satisfy them will contain “dead” nodes that will never be reached in any play of the strategy. Note also that the same question may be asked twice on the same path through the tree, and this too may give rise to inaccessible nodes.

The key observation in the construction of $\mathcal{B}$ is that nodes and labels can themselves be coded as natural numbers—that is, both $\mathrm{Seq}(\mathrm{N})$ and $N+N$ admit injections into N. Let $\langle\cdots\rangle : \mathrm{Seq}(\mathrm{N}) \rightarrowtail \mathrm{N}$ be some effective coding for nodes: for example, take

$$\langle\epsilon\rangle = 0, \qquad \langle m_1,\ldots,m_k\rangle = 2^{m_1}3^{m_2}\ldots p_k^{m_k+1} - 1,$$

where $\langle m_1,\ldots,m_k\rangle$ abbreviates $\langle[m_1,\ldots,m_k]\rangle$. Likewise, let $[?,!] : \mathrm{N}+\mathrm{N} \rightarrowtail \mathrm{N}$ be some effective coding for labels, for example:

$$?n = 2n, \qquad !n = 2n+1.$$

In general, we will require both that the encoding operations themselves are effective and that their images are decidable subsets of N. The particular codings given above are of course bijective, but this is not a necessary property and we will not assume it.

Using these codings, we can represent a decision tree $\sigma : \mathrm{Seq}(\mathrm{N}) \rightharpoonup \mathrm{N} + \mathrm{N}$ simply by a partial function $f : \mathrm{N} \rightharpoonup \mathrm{N}$ (i.e. by an element $f \in \mathrm{N}_\bot^\mathrm{N}$). Explicitly, we say f *represents* σ if for all $\alpha \in \mathrm{Seq}(\mathrm{N})$ and $l \in \mathrm{N} + \mathrm{N}$ we have

$$f\langle\alpha\rangle = [?, !](l) \quad \text{iff} \quad \sigma\alpha = l.$$

Clearly, every partial function $f \in \mathrm{N}_\bot^\mathrm{N}$ represents a unique decision tree σ_f, and for every decision tree σ there is a least partial function f_σ that represents it.

If f represents σ, a play of σ against a function g may be viewed as a "dialogue" between f and g. Thus, the procedure for playing a strategy against an argument gives rise to an operation $| : \mathrm{N}_\bot^\mathrm{N} \times \mathrm{N}_\bot^\mathrm{N} \to \mathrm{N}_\bot$. This operation is itself sequentially computable; the following informal ML-style definition gives the idea:

$$\begin{array}{l} \text{fun } \mathit{Play}\ f\ g\ \alpha\ = \\ \quad \text{case } f\langle\alpha\rangle \text{ of} \\ \qquad !n \Rightarrow n \\ \quad\ \ |\ ?n \Rightarrow (\text{case } g(n) \text{ of } m \Rightarrow \mathit{Play}\ f\ g\ (\alpha; m)) \\ \text{fun } \mid\ f\ g\ = \mathit{Play}\ f\ g\ \epsilon \end{array}$$

A minor modification of this gives us an operation $\bullet : \mathrm{N}_\bot^\mathrm{N} \times \mathrm{N}_\bot^\mathrm{N} \to \mathrm{N}_\bot^\mathrm{N}$: in effect, we use f to represent an infinite forest of decision trees rather than just a single one.

$$\text{fun } \bullet\ f\ g\ = (\text{fn } n \Rightarrow \mathit{Play}\ f\ g\ [n])$$

These ideas may be expressed more formally as follows:

Definition 3.2 *Let Play* $: \mathrm{N}_\bot^\mathrm{N} \times \mathrm{N}_\bot^\mathrm{N} \times \mathrm{Seq}(\mathrm{N}) \to \mathrm{N}_\bot$ *be the smallest partial function such that, for all* f, g, α, n, m*,*

- *if* $f\langle\alpha\rangle = !n$ *then* $\mathit{Play}(f, g, \alpha) = n$*,*
- *if* $f\langle\alpha\rangle = ?n$ *and* $g(n) = m$ *then* $\mathit{Play}(f, g, \alpha) = \mathit{Play}(f, g, (\alpha; m))$*.*

Now define $|, \bullet$ *by* $f \mid g = \mathit{Play}(f, g, \epsilon)$ *and* $f \bullet g = \lambda n.\mathit{Play}(f, g, [n])$*.*

We write $\mathcal{B}$ for the applicative structure $(\mathrm{N}_\bot^\mathrm{N}, \bullet)$.

It is easy to see that if $f, g \in \mathrm{N}_\bot^\mathrm{N}$ are *partial recursive* then so is $f \bullet g$. We write $\mathcal{B}_{\mathit{eff}}$ for the applicative substructure of $\mathcal{B}$ consisting of the partial recursive functions.

The following fact is stated in [24] and proved in [17].

Theorem 3.3 *$\mathcal{B}$ and $\mathcal{B}_{\mathit{eff}}$ are combinatory algebras.*

Indeed, both $\mathcal{B}$ and $\mathcal{B}_{\mathit{eff}}$ can be made into CA$\bot$s by taking as $\bot$ the everywhere undefined partial function $\lambda n.\bot$.

What type structure does $\mathcal{B}$ give rise to? Given the obviously sequential character of $\mathcal{B}$, one's first guess might be that $E(\mathcal{B})$ would be a fully abstract model of PCF, while $E(\mathcal{B}_{\mathit{eff}})$ would be a universal model of PCF. At least that was my reaction on encountering $\mathcal{B}$, and it came as a considerable surprise to me to discover that $E(\mathcal{B})$ and $E(\mathcal{B}_{\mathit{eff}})$ contain functions that are *not* PCF-definable but are still "sequentially computable" in some sense. Indeed, these functions

can be implemented in existing programming languages such as Standard ML using non-functional features such as exceptions or references.

Thus, we have a class of "sequential" higher-type functionals, strictly wider than the familiar PCF-sequential functionals. We call the functionals in $E(\mathcal{B})$ the *sequentially realizable (SR)* functionals; likewise, the elements of $E(\mathcal{B}_{\mathit{eff}})$ are called the *effective SR* functionals.

A deeper understanding of this "alternative" notion of higher-type sequentiality is one of the most interesting things to have emerged so far from the study of sequential realizability models, so it is worth devoting a separate section to these type structures.

4 The sequentially realizable functionals

4.1 Theory

We start by sketching the picture of the SR functionals from a theoretical point of view. The material described here is covered extensively in [17].

The type structure $E(\mathcal{B})$ turns out to be a very well-behaved structure, and has several apparently quite different characterizations, including some which were known previously. Indeed, the following constructions all give rise to type structures isomorphic to $E(\mathcal{B})$:

- The *strongly stable* model due to Bucciarelli and Ehrhard [7].
- The extensional collapse of the Berry-Curien *sequential algorithms* model [5].
- The category of presheaves over the monoid of sequential endofunctions on $\mathrm{N}_\perp^\mathrm{N}$.
- The *modified realizability* model over $\mathcal{B}$.

The equivalences between these constructions are due to Ehrhard [10, 11], van Oosten [24] and the author. The diversity of these descriptions contribute to the impression that the class of SR functionals is somehow a mathematically natural structure. Moreover, the above descriptions can be used to show many good theoretical properties of $E(\mathcal{B})$.

Not surprisingly, the effective analogue $E(\mathcal{B}_{\mathit{eff}})$ is isomorphic to a substructure of $E(\mathcal{B})$ (though this is not immediate from the definitions we have given). One can also characterize $E(\mathcal{B}_{\mathit{eff}})$ in various ways—for instance, as the extensional collapse of the *effective* sequential algorithms model.

It can be shown that $E(\mathcal{B}_{\mathit{eff}})$ contains a particular functional H, of type level 3, such that every effective SR functional is definable from H in PCF. That is, the effective SR functionals are precisely the functionals computable in the language PCF+H. Furthermore, H can in fact be implemented in any of a number of non-functional languages extending PCF—for example, using references, exceptions or continuations. Whilst the implementations of H make internal use of non-functional features, the observable behaviour of H is purely functional, or *extensional*, in the sense that it will produce the same result if

applied to two different implementations of the same function. This means that programmers could use functions such as H and still believe they were doing pure functional programming.

In retrospect, the existence of a class of sequentially computable functionals that were implementable in existing languages but contained more than the PCF functionals was already clear from the fact that the strongly stable functionals are the extensional collapse of the sequential algorithms model, combined with the fact that all effective sequential algorithms are definable in the language PCF+**catch** [9]. However, the realizability models based on $\mathcal{B}$ and $\mathcal{B}_{eff}$ have brought this class of functionals more clearly into focus. Moreover, the existence of a language that defines the effective SR functionals and *only* them is a new result, and its proof was directly inspired by the realizability model $E(\mathcal{B}_{eff})$.

4.2 An example

We give here just one example of an effective SR functional that is not PCF-definable, to illustrate the kind of extra power that they provide. The functional we describe, though simple by comparison with H, seems to be sufficiently powerful for most of the plausible programming applications of the SR functionals. Further examples, and some possible applications, are given in an ML source file available from my home page [18].

The functional in question computes the *modulus* of a sequential type 2 function F at a type 1 argument g. Let us suppose that $g : 0 \to 0$ is a strict function, and that the application Fg converges—that is, Fg terminates yielding some natural number n. Since F is sequential and therefore a *stable* function, there is a unique smallest finite subfunction $g_0 \sqsubseteq g$ such that $Fg_0 = n$. (Intuitively, by the time the computation of Fg finishes, F can only have interrogated finitely many values of g.) Since g_0 is also strict, its *graph* can be uniquely represented as a finite list of ordered pairs $[(a_0, b_0), \ldots, (a_r, b_r)]$, where $a_0 < \ldots < a_r$.

We may now consider the following specification for a function $Mod : 2 \to 1 \to 0$: given any F, g satisfying the conditions above, $Mod\ F\ g$ returns the graph of the corresponding finite function g_0 (suitably coded up as a natural number). Clearly no such function Mod can be defined in pure PCF, since it is not monotone with respect to the pointwise order. However, there *does* exist an effective SR functional Mod with the above properties. Indeed it is easy to see how to implement a function of this kind in SML, e.g. with type

```
((int->int)->'a) -> (int->int) -> (int*int)list.
```

The point of Mod is that it provides information about *how much of g is actually used by F*. Intuitively, $Mod\ F\ g$ tells us what calls to g are made in the course of computing Fg. It is crucial here that the graph is represented as a sorted list, so that $Mod\ F\ g$ does not give away any information about the *order* in which the calls to g were made. This ensures that Mod indeed acts extensionally—it will produce the same result when given two different implementations of the same function F that use different evaluation orders (such as $\lambda g.\, g0 + g1$ and $\lambda g.\, g1 + g0$).

In [18] we give some examples of programming tasks for which a function like *Mod* might be useful: one involving the construction of general-purpose search algorithms, and one involving exact real-number computation. It would be interesting to know if there were other natural applications.

4.3 Possible applications

The existence of a language that defines precisely the effective SR functionals (PCF+H) opens up the possibility of a practical programming language that incorporated them into its "purely functional" fragment. One could do this by building in H, or something equally powerful, as a primitive in the language. Even without going to these lengths, by implementing functions like H or *Mod* in SML we can experiment with *styles* of functional programming that make use of SR functionals.

There are various plausible reasons why either of the above possibilities might be of interest from a practical point of view. For example, there are natural examples of programs in this extended functional setting which would be (variously) impossible, inefficient or just inelegant to implement in pure functional ML, for instance. For such programs, one would gain the transparency and ease of reasoning offered by pure functional programming. If our SR primitives were hard-wired into a language implementation, one can imagine that one might also gain increased scope for compiler optimization and maybe even garbage collection.

From the point of view of formal program development and verification, there are also more theoretically based reasons why one might prefer a language whose functional core corresponded to SR rather than PCF. In many ways the SR type structure is mathematically better behaved than the PCF type structure. For example, the finitary fragment of the SR type structure is decidable, unlike the finitary PCF type structure [16]; and every simple SR type is a retract of type 2, whereas in PCF there is no such "universal type". Facts such as this can be expected to make it easier to find a good axiomatization for a program logic for an SR-functional language than for a PCF-functional one. Furthermore, having *designed* the logic, one would expect it to be easier to carry out proofs within it, since larger fragments of the logic will be decidable.

At present, the main obstacle to the implementation of our proposal is that our functional H is computationally somewhat intractable: it involves a factorial-size search in the worst case, which unfortunately does arise in many natural instances. Another obstacle (related to this) is the conceptual difficulty of the definition of H. One might get round these obstacles in two ways. Firstly, one might be content with implementing simpler functionals (such as *Mod*) that covered most of the natural applications. Although intellectually somewhat unsatisfying, this might in the end be the most practical course. Secondly, one might hope to discover a better functional or primitive construct than H but with the same universality property. Although I suspect that any universal SR functional must have at least exponential complexity in the worst case, it seems possible that that there may be one whose behaviour is tractable in almost all naturally arising cases.

5 Some history-free models (Abramsky)

Next we describe some combinatory algebras due to Abramsky based on history-free strategies. The definitions are inspired by Girard's Geometry of Interaction, and they have much the same flavour as the games models of [2].

5.1 A linear combinatory algebra

Let T be some infinite set of *tokens* or *moves*, and suppose we have an injective coding function $[L, R] : T + T \rightarrowtail T$ We will construct a combinatory algebra $\mathcal{A}$ whose underlying set is the set of all partial functions $T \rightharpoonup T$. We may think of an element $f \in \mathcal{A}$ as a *strategy*, telling us to respond to a move t with a move ft (if this is defined). It is convenient to visualize f as a box with one input wire and one output wire, each capable of carrying an element of T.

Using the coding function, we can regard an element $f \in \mathcal{A}$ as representing the partial function

$$[L, R]^{-1} \circ f \circ [L, R] : \; T + T \rightharpoonup T + T.$$

The idea is that inputs and outputs are now elements of T tagged with some "address" information—tokens are marked as belonging to the left or the right copy of T.

We now define an *application* operation $\cdot$ on $\mathcal{A}$ as follows. Informally, given $f, g \in \mathcal{A}$, we view f as a function $T + T \rightharpoonup T$, and hook up its right summand to the element g as shown in Figure 2. The unconnected left-hand wires of f then form the input and output wires of the resulting element $f \cdot g$. Intuitively, a token t comes in along the left-hand input to f. If $f(Lt)$ belongs to the left copy of T, the token $L^{-1} f(Lt)$ is immediately returned as the result. If $f(Lt)$ belongs to the right copy, the token $R^{-1} f(Lt)$ is fed to g, and the result (tagged as a right token) is fed back to f. Thus we may go round the loop one or more times and then (perhaps) eventually emerge on the left-hand output of f. We can thus see f and g as the two participants in a dialogue, the outcome of which defines the function $f \cdot g$.

This definition of application may be formalized as follows:

$$f \cdot g \;=\; L^{-1} \circ \bigcup_{r \geq 0} h^r \circ f \circ L, \qquad \text{where} \;\; h = f \circ R \circ g \circ R^{-1}.$$

This is essentially Girard's *execution formula*. Here $\circ$ can be understood as composition of relations, and $\bigcup$ as union of graphs of relations. (It is easy to verify that if the relations f, g are both partial functions then so is $f \cdot g$.)

The structure $(\mathcal{A}, \cdot)$ that we have defined is not yet a combinatory algebra, but it is an *(affine) linear combinatory algebra*—that is, there are elements $i, b, c, k \in \mathcal{A}$ satisfying

$$i \cdot x \;=\; x, \qquad b \cdot x \cdot y \cdot z \;=\; (x \cdot z) \cdot y, \qquad c \cdot x \cdot y \cdot z \;=\; x \cdot (y \cdot z), \qquad k \cdot x \cdot y$$

for all $x, y, z \in \mathcal{A}$. In fact it is very easy to construct these elements: for instance, i is induced by the "twist" map on $T + T$ (more formally, $i = [L, R]^{-1} \circ [R, L]$). It is a pleasant geometrical exercise to construct the elements b, c, k and to check that they have the required properties.

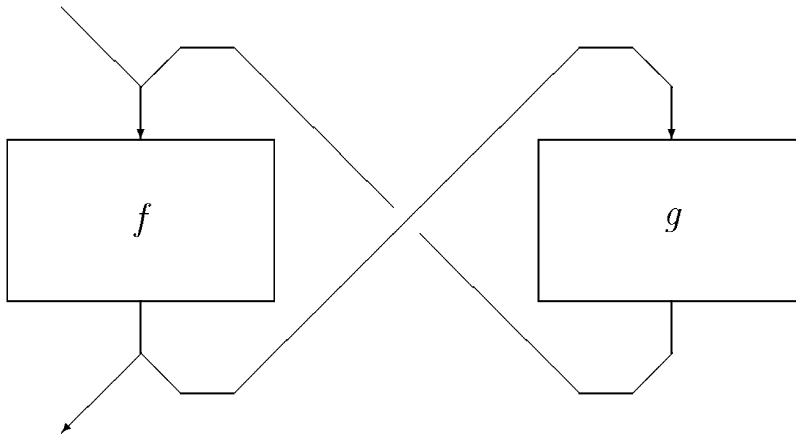

Figure 2: The linear application $f \cdot g$

5.2 An ordinary combinatory algebra

The algebra $(\mathcal{A}, \cdot)$ is interesting in its own right, but in order to obtain a realizability model of PCF we need to start from an *intuitionistic* combinatory algebra—that is, a combinatory algebra in the usual sense. In fact, an affine linear combinatory algebra $(A, \cdot)$ is an ordinary iff it contains an element w (corresponding to contraction in linear logic) such that

$$w \cdot x \cdot y \;=\; x \cdot y \cdot y$$

for all $x, y \in A$. One can then define the s combinator using the magic formula

$$s \;=\; c \cdot (c \cdot (c \cdot w) \cdot b) \cdot (c \cdot c),$$

and verify that $s \cdot x \cdot y \cdot z = (x \cdot z) \cdot (y \cdot z)$.

Intuitively, the reason why $(\mathcal{A}, \cdot)$ contains no w combinator is that in order to compute $x \cdot y \cdot y$ one requires two separate "copies" of y. To overcome this, we will introduce an operator $! : \mathcal{A} \to \mathcal{A}$ that takes an element and in some sense produces infinitely many copies of it. Formally, we assume we are given an injective coding function $\langle - . - \rangle : \omega \times T \rightarrowtail T$ (where ω is the set of natural numbers); the element $\langle i, t \rangle$ then represents the element t from the ith copy of T. Given $f \in \mathcal{A}$, we then define

$$!f \;=\; \langle -, - \rangle \circ (\mathrm{id}_\omega \times f) \circ \langle -, - \rangle^{-1},$$

so that whenever ft is defined we have $!f\langle i, t\rangle = \langle i, ft \rangle$ for all $i \in \omega$. Thus, $!f$ represents an ω-indexed disjoint union of copies of f.

We can now define a new application operation $\bullet$ on $\mathcal{A}$ via $f \bullet g = f \cdot !g$. Intuitively, instead of using just a single copy of g, we now hook up f to infinitely many copies of g (see Figure 3). Using the same ideas as before it is easy to construct elements i, b, c, k of $(\mathcal{A}, \bullet)$ satisfying the appropriate equations (but note that these are not the same elements as before!) Furthermore, it is now straightforward to construct an element w satisfying $w \cdot x \cdot y = x \cdot y \cdot y$, and so

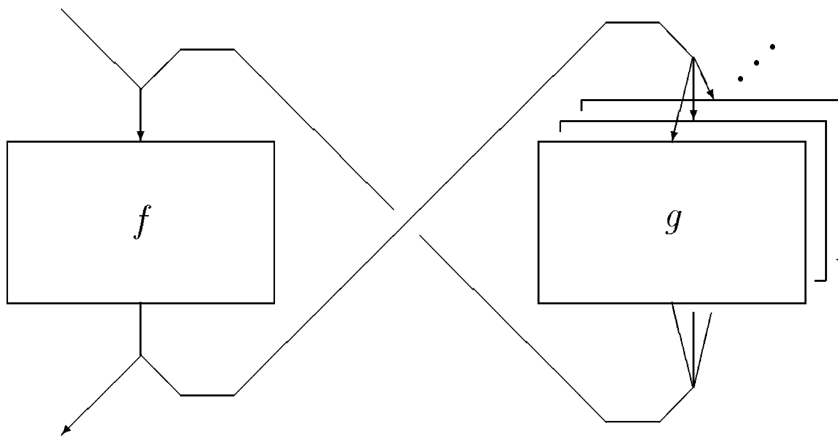

Figure 3: The intuitionistic application $f \bullet g$

the structure $(\mathcal{A}, \bullet)$ is an ordinary combinatory algebra. Since the everywhere undefined function $T \rightharpoonup T$ is an obvious choice for a bottom element, we have a CA⊥.

5.3 Relationship to $\mathcal{B}$

What is the type structure realized by $(\mathcal{A}, \bullet)$? For definiteness, let us suppose that $T = \mathrm{N}$ and that $[L, R], \langle -, - \rangle$ are standard recursive codings. It turns out that in this case $\mathcal{A}$ is actually equivalent to the van Oosten algebra $\mathcal{B}$, in the following sense:

Proposition 5.1 *The combinatory algebras $\mathcal{A}$ and $\mathcal{B}$ have the same underlying set (the set of partial functions $N \rightharpoonup N$), and there are elements α, β such that, for all f, g,*

$$\alpha \bullet_{\mathcal{B}} f \bullet_{\mathcal{B}} g \;=\; f \bullet_{\mathcal{A}} g, \qquad \beta \bullet_{\mathcal{A}} f \bullet_{\mathcal{A}} g \;=\; f \bullet_{\mathcal{B}} g.$$

That is, application in each combinatory algebra is representable in the other. This is in fact a special case of the notion of equivalence of combinatory algebras studied in [19, Chapter 2]. It follows from general results there (and is easy enough to see directly in this case) that $\mathcal{A}, \mathcal{B}$ give rise to exactly the same category of PERs, and exactly the same type structure. So $E(\mathcal{A})$ again consists of precisely the SR functionals.

In the concrete case in question, one may also consider the subalgebra $\mathcal{A}_{\mathit{eff}}$ consisting of partial recursive functions. Not surprisingly, $\mathcal{A}_{\mathit{eff}}$ is equivalent to $\mathcal{B}_{\mathit{eff}}$, and so $E(\mathcal{A}_{\mathit{eff}})$ consists of precisely the effective SR functionals.

From the point of view of realizability, one could therefore regard $\mathcal{A}$ just as different ways of presenting the same model. However, there are several reasons why the presentation via $\mathcal{A}$ may be of particular interest:

1. It seems that $\mathcal{A}$ is a more “fine-grained” model of computation than $\mathcal{B}$, as is evidenced by the fact that its construction naturally decomposes into a

linear structure plus the ! operator. A related point is that the proof that we get a combinatory algebra seems significantly simpler for $\mathcal{A}$ than for $\mathcal{B}$.

2. The construction of $\mathcal{A}$ relies only on very simple facts about the category of partial functions, and so exactly the same construction yields analogous combinatory algebras in many other categories. Thus, one can equally well define a combinatory algebra consisting of arbitrary (binary) *relations* on T, or partial *injective* functions, or *symmetric* partial functions $f : T \to T$ (i.e. those satisfying $ft = u \Rightarrow fu = t$). This last case is particularly interesting as it leads to a highly constrained model in which all computations are "reversible". The corresponding realizability model promises to be useful for providing good interpretations of System F polymorphism.

3. Lastly, the construction of $\mathcal{A}$ admits the identification of a particularly interesting subalgebra of *well-bracketed* strategies, which seems not to arise with $\mathcal{B}$. It is this subalgebra and its associated type structure that we consider next.

5.4 A well-bracketed subalgebra

In order to define a notion of well-bracketing, we need to assume some additional properties of the set T and the coding functions. For convenience, we will assume henceforth that the set T of tokens t is freely generated by the following grammar:

$$t \ ::= \ * \mid n \mid Lt \mid Rt \mid \langle i, t \rangle \qquad (n \in \mathrm{N}, i \in \omega).$$

We regard T as equipped with the intrinsic coding functions $[L, R] : T + T \rightarrowtail T$ and $\langle -, - \rangle : \omega \times T \rightarrowtail T$, and construct the combinatory algebra $\mathcal{A}$ from these.

We think of the token $*$ as a single basic question, and the tokens n as corresponding answers. Note that every token t has the form $C[*]$ or $C[n]$ for some single-holed context C—we call t a *question* or an *answer* according to which of these forms it has.

We say that an answer $C'[n]$ *matches* a question $C[*]$ just when $C' \equiv C$. A sequence of tokens is said to be *well-bracketed* if, intuitively, every answer in the sequence matches the most recent unanswered question. We formalize this notion as follows:

Definition 5.2 *(i) A finite or infinite sequence $t_0 t_1 \ldots$ of tokens is* well-bracketed *if there is an injective function $m : \{i \in \mathrm{N} \mid t_i \text{ is an answer}\} \to \mathrm{N}$ such that, for all i where t_i is an answer,*

- *$m(i) < i$,*
- *$t_{m(i)}$ is a question and t_i matches $t_{m(i)}$,*
- *if $m(i) < j < i$ and t_j is a question, there is $i' < i$ such that $m(i') = j$.*

(ii) A finite or infinite sequence $t_0 t_1 \ldots$ has a bracketing violation at position *r if $t_0 \ldots t_{r-1}$ is well-bracketed but $t_0 \ldots t_r$ is not.*

We can now say what it means for a strategy $f \in \mathcal{A}$ to be well-bracketed. The informal idea is this: whenever we play f against an argument g, as long as g does not introduce a bracketing violation, neither will f (this is a kind of "rely-guarantee" condition). To formalize this, we need to make explicit the sequence of moves generated by a play of f against g.

Definition 5.3 *(i) Given $f, g \in \mathcal{A}$ and $t \in T$, the* sequence generated by f, g at *t is the finite or infinite sequence $t_0 t_1 \ldots$ constructed inductively as follows:*

- *$t_0 = Lt$.*
- *if $f(t_{2i})$ is defined then $t_{2i+1} = f(t_{2i})$.*
- *if $t_{2i+1} = R\langle j, u\rangle$ and $g(u)$ is defined then $t_{2i+2} = R\langle j, g(u)\rangle$.*

(ii) An element $f \in \mathcal{A}$ is well-bracketed *if, for all $g \in \mathcal{A}$ and $t \in T$, the sequence $t_0 t_1 \ldots$ generated by f, g at t does not contain a bracketing violation at an odd position.*

Note that $(f \bullet g)(t) = u$ iff the sequence generated by f, g at t finishes with Lu at an odd position.

It is a pretty exercise to show that the well-bracketed elements of $\mathcal{A}$ are closed under application, and that the basic combinators i, b, c, k, w are all well-bracketed. The well-bracketed elements thus constitute a combinatory subalgebra $\mathcal{A}_{wb}$ of $\mathcal{A}$. We also have the subalgebra $\mathcal{A}_{wb,\mathit{eff}}$ of *effective* well-bracketed strategies.

The main interest of $\mathcal{A}_{wb}$ lies in the following result.

Theorem 5.4 *(i) The type structure $E(\mathcal{A}_{wb})$ coincides with that given by the (extensional) fully abstract games models for PCF [2, 14].*

(ii) The type structure $E(\mathcal{A}_{wb,\mathit{eff}})$ consists of precisely the PCF-definable functionals.

The proof of this theorem is non-trivial. It proceeds by showing, at each type σ, that the only possible realizers for elements of E_σ are sufficiently well-behaved that they are equivalent to strategies living in the games model of [2]. A paper including the detailed proof is planned [3].

[Sorry, that's as far as I've got! Rest to follow soon, I hope.]

6 A 'concurrent process' model (Abramsky)

7 Some innocent games models (Nickau,Ong)

8 Domain models and a term model (Streicher)

9 Conclusion

Most papers end with a conclusion.

References

[1] S. Abramsky. Typed realizability. Talk given at CTCS, Cambridge, 1995.

[2] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. Submitted for publication, 1996.

[3] S. Abramsky and J.R. Longley. Some combinatory algebras for sequential computation. In preparation.

[4] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, revised edition, 1984.

[5] G. Berry and P.-L. Curien. Sequential algorithms on concrete data structures. *Theoretical Computer Science*, 20(3), 1982.

[6] K. Bruce and J.C. Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *Proc. ACM Symposium on Principles of Programming Languages*, 1992.

[7] A. Bucciarelli and T. Ehrhard. Sequentiality and strong stability. In *Proc. 6th Annual Symposium on Logic in Computer Science*, 1991.

[8] A. Bucciarelli and T. Ehrhard. A theory of sequentiality. *Theoretical Computer Science*, 113:273–291, 1993.

[9] R. Cartwright, P.-L. Curien, and M. Felleisen. Fully abstract semantics for observably sequential languages. *Information and Computation*, 111(2):297–401, 1994.

[10] T. Ehrhard. Projecting sequential algorithms on strongly stable functions. *Ann. Pure Appl. Logic*, 77, 1996.

[11] T. Ehrhard. A relative PCF-definability result for strongly stable functions and some corollaries. Preprint, 1997.

[12] J.-Y. Girard. *Interprétation functionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Paris, 1972.

[13] J.M.E. Hyland, P.T. Johnstone, and A.M. Pitts. Tripos theory. *Math. Proc. Camb. Phil. Soc.*, 88, 1980.

[14] J.M.E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II and III. Submitted for publication, 1996.

[15] G. Kreisel. Interpretation of analysis by means of functionals of finite type. In A. Heyting, editor, *Constructivity in Mathematics.* North-Holland, 1959.

[16] R. Loader. Finitary PCF is not decidable. To appear, 1996.

[17] J.R. Longley. The sequentially realizable functionals. In preparation.

[18] J.R. Longley. When is a functional program not a functional program?: a walk-through introduction to the sequentially realizable functionals. Standard ML source file, available from `http://www.dcs.ed.ac.uk/home/jrl/`.

[19] J.R. Longley. *Realizability Toposes and Language Semantics.* PhD thesis, University of Edinburgh, 1995. Available as ECS-LFCS-95-332.

[20] J.R. Longley and A.K. Simpson. A uniform approach to domain theory in realizability models. *Math. Struct. in Comp. Sci.*, 7:469–505, 1997.

[21] G. Longo and E. Moggi. Constructive natural deduction and its 'ω-set' interpretation. *Math. Structures in Computer Science*, 1, 1991.

[22] H. Nickau. Hereditarily sequential functionals. In *Proc. Symp. Logical Foundations of Computer Science*. Springer, 1994.

[23] P.W. O'Hearn and J.G. Riecke. Kripke logical relations and PCF. *Information and Computation*, 120(1):107–116, 1995.

[24] J. van Oosten. A combinatory algebra for sequential functionals of finite type. Technical Report 996, University of Utrecht, 1997.

[25] W.K.-S. Phoa. *Domain Theory in Realizability Toposes*. PhD thesis, University of Cambridge, 1990. Available as CST-82-91, Department of Computer Science, University of Edinburgh.

[26] W.K.-S. Phoa. From term models to domains. In *Proc. of Theoretical Aspects of Computer Software, Sendai*. Springer LNCS 526, 1991.

[27] G.D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5, 1977.

[28] B. Reus and T. Streicher. General synthetic domain theory—a logical approach. In *Category Theory in Computer Science '97*, pages 293–313. Springer LNCS 1290, 1997.

[29] A.K. Simpson. The convex powerdomain in a category of posets realized by cpos. In *Proc. Category Theory and Computer Science*, pages 117–145. Springer LNCS 953, 1995.