Math. Struct. in Comp. Science (1996), *vol.* 6, *pp.* 127–139

A note on recursive functions†

NICOLETTA SABADINI‡, SEBASTIANO VIGNA‡ and ROBERT F. C. WALTERS§

‡*Dipartimento di Scienze dell'Informazione, Università di Milano, Italy.*
§*School of Mathematics and Statistics, University of Sydney, Australia.*

Received 3 March 1994; revised 13 January 1995

In this paper, we propose a new and elegant definition of the class of recursive functions, which is analogous to Kleene's definition but differs in the primitives taken, thus demonstrating the computational power of the concurrent programming language introduced in Walters (1991), Walters (1992) and Khalil and Walters (1993).
The definition can be immediately rephrased for any distributive graph in a countably extensive category with products, thus allowing a wide, natural generalization of computable functions.

1. Introduction

This paper has two objectives. We propose a new and elegant definition of the class of recursive functions, which is analogous to Kleene's definition but differs in the primitives taken, and we demonstrate the computational power of the concurrent imperative programming language introduced in Walters (1991), Walters (1992) and Khalil and Walters (1993).

In Kleene's definition the class of recursive functions is defined to be the smallest containing the projections, the successor function and the zero function, and closed under substitution, primitive recursion and the μ operator. Notice that the only sets involved in this definition are the cartesian powers of **N**. The definition proposed in this paper involves not only the cartesian product of sets but also the sum (or disjoint union of sets). The class is defined to be the smallest containing the predecessor and successor functions, the injections and projections, the diagonal and codiagonal functions, and the distributive law, and that is closed under the operations of composition, product, sum and iteration.

The definition is mathematically elegant because, apart from iteration, the operations allowed on sets and functions are exactly those of a *distributive category* (Walters 1991), that is, a category with finite products and sums and a distributive law between them – the 'arithmetic of sets'. Furthermore, the iteration operation satisfies nice properties relating it to composition, product and sum.

† This work has been supported by the Australian Research Council, Esprit BRA ASMICS, Italian MURST 40% and the Italian CNR.

These same operations are the basis of a concurrent imperative programming language. The sums and products allow the description of complex data types, rather than just the natural numbers, the product of functions expresses parallel execution, the sum of functions expresses execution by cases, and the distributive law allows the 'if then else' construction. In Sabadini *et al.* (1993) this language was shown to be asynchronous. In fact, Sabadini *et al.* (1993) together with this paper demonstrates how to implement recursive functions with delay-independent asynchronous circuits. More general asynchronous circuits are modelled in Katis *et al.* (1994).

This programming language has a very clear mathematical semantics, contrasting with the common view (Backus 1978) that imperative languages do not have a clear mathematical meaning.

Moreover, the definition can be immediately rephrased in any countably extensive category with products, thus allowing a wide, natural generalization of computable functions.

2. Kleene's definition of recursive function

In this section $\underline{m}$ will be used to denote the generic vector of integers $m_1, m_2, \ldots, m_k$. Moreover, in order to compare the new definition with the classical one, we will use 'recursive functions' to mean the functions $\mathbf{N}^k \longrightarrow \mathbf{N}$ of classic recursion theory, and 'δ-recursive functions' to mean the set maps generated by our operations.

Definition 1. The set of recursive functions from $\mathbf{N}^k$ to $\mathbf{N}$ is the smallest set of functions containing the 0 constant function, the successor function $\sigma : \mathbf{N} \longrightarrow \mathbf{N}$ and the projections $\pi_j : \mathbf{N}^k \longrightarrow \mathbf{N}$, and that is closed under the following constructions:

(i) **Substitution** Given $f : \mathbf{N}^k \longrightarrow \mathbf{N}$ and $g_i : \mathbf{N}^j \longrightarrow \mathbf{N}$ $(1 \leq i \leq k)$, form $f(g_1, g_2, \ldots, g_k) : \mathbf{N}^j \longrightarrow \mathbf{N}$.

(ii) **Primitive recursion** Given $g : \mathbf{N}^k \longrightarrow \mathbf{N}$, $h : \mathbf{N}^{k+2} \longrightarrow \mathbf{N}$, form $f : \mathbf{N}^{k+1} \longrightarrow \mathbf{N}$ by setting

$$f(\underline{m}, 0) = g(\underline{m})$$

and

$$f(\underline{m}, n+1) = h(\underline{m}, n, f(\underline{m}, n)).$$

(iii) **μ-operator** Given a function $f : \mathbf{N}^{k+1} \longrightarrow \mathbf{N}$ such that for all $\underline{m}$ there is an n such that $f(\underline{m}, n) = 0$, form $g : \mathbf{N}^k \longrightarrow \mathbf{N}$ by setting

$$g(\underline{m}) = \mu_n f(\underline{m}, n) = \min\{n \in \mathbf{N} \mid f(\underline{m}, n) = 0\}.$$

If we drop the condition of existence of a solution in n for $f(\underline{m}, n) = 0$, we have the definition of a partial recursive function.

Definition 2. The set of partial recursive functions from $\mathbf{N}^k$ to $\mathbf{N}$ is the smallest set of partial functions containing the 0 constant function, the successor function $s : \mathbf{N} \longrightarrow \mathbf{N}$ and the projections $\pi_j : \mathbf{N}^k \longrightarrow \mathbf{N}$, and that is closed under substitution, primitive recursion, and under the following construction (partial μ-operator): given $f : \mathbf{N}^{k+1} \longrightarrow \mathbf{N}$, form $\mu_n f(\underline{m}, n)$ by setting

$$\min\{n \in \mathbf{N} \mid f(\underline{m}, n) = 0 \text{ and } f(\underline{m}, i) \text{ is defined for all } i \text{ less than } n\};$$

$\mu_n f(\underline{m}, n)$ is undefined when this set is empty.

It is a well-known but non-trivial fact that the class of recursive functions coincides with the class of total partial recursive functions. We will give a very simple proof of this fact in Section 9, by using a property of **call**.

3. Products and sums

We now describe the operations on sets and functions that will be used in Section 7 to give our new definition.

For any pair of sets X, Y, the *sum* $X + Y$ is the disjoint union of X and Y, that is, the set $X \times \{0\} \cup Y \times \{1\}$. The canonical injections $\text{inj}_1 : X \longrightarrow X + Y$, $\text{inj}_2 : Y \longrightarrow X + Y$ are defined by

$$\begin{aligned} \text{inj}_1(x) &= \langle x, 0\rangle \\ \text{inj}_2(y) &= \langle y, 1\rangle. \end{aligned}$$

The *product* $X \times Y$ is the cartesian product of X and Y, that is, the set $\{\langle x, y\rangle \mid x \in X \wedge y \in Y\}$. The canonical projections $\text{proj}_1 : X \times Y \longrightarrow X$, $\text{proj}_2 : X \times Y \longrightarrow Y$ are defined by

$$\begin{aligned} \text{proj}_1(\langle x, y\rangle) &= x \\ \text{proj}_2(\langle x, y\rangle) &= y. \end{aligned}$$

The empty set $O = \emptyset$ and the singleton $I = \{*\}$ are the units of $+$ and $\times$.

We also have corresponding constructions on functions: for any pair of functions $f : X \longrightarrow A$, $g : Y \longrightarrow A$, $(f \mid g) : X + Y \longrightarrow A$ is defined by

$$(f \mid g)(z) = \begin{cases} f(x) & \text{if } z = \langle x, 0\rangle \\ g(y) & \text{if } z = \langle y, 1\rangle, \end{cases}$$

and for any pair of functions $f : A \longrightarrow X$, $g : A \longrightarrow Y$, $(f, g) : A \longrightarrow X \times Y$ is defined by

$$(f, g)(a) = \langle f(a), g(a)\rangle.$$

Of course, the composition of two functions is still a function.

Note that the sums and products of sets are related by the *distributivity isomorphism*:

$$\delta : X \times Y + X \times Z \cong X \times (Y + Z).$$

Since any set X has an associated identity map $\mathbf{1}_X : X \longrightarrow X$, we can build some new derived maps, and define some new operation based on the old ones:

— $\Delta_X = (\mathbf{1}_X, \mathbf{1}_X) : X \longrightarrow X \times X$;
— $\nabla_X = (\mathbf{1}_X \mid \mathbf{1}_X) : X + X \longrightarrow X$;
— for any pair of functions $f : X \longrightarrow X'$, $g : Y \longrightarrow Y'$, we have

$$f \times g = (f \circ \text{proj}_1, g \circ \text{proj}_2) : X \times Y \longrightarrow X' \times Y'$$

and

$$f + g = (\mathrm{inj}_1 \circ f \mid \mathrm{inj}_2 \circ g) : X + Y \longrightarrow X' + Y'.$$

Notice that $(f \mid g) = \nabla \circ (f + g)$ and $(f, g) = (f \times g) \circ \Delta$, so anything constructable using $(-,-)$ and $(- \mid -)$ is also constructable using $\times$, $+$, Δ and ∇. For instance, the canonical function $\delta : X \times Y + X \times Z \longrightarrow X \times (Y + Z)$ described above is exactly

$$\delta = (\mathbf{1}_X \times \mathrm{inj}_1 \mid \mathbf{1}_X \times \mathrm{inj}_2),$$

and the sum and product commutativity isomorphisms, both of which by abuse of notation we call **twist**, can be expressed as

$$\mathbf{twist} : X \times Y \overset{(\mathrm{proj}_2, \mathrm{proj}_1)}{\longrightarrow} Y \times X$$

and

$$\mathbf{twist} : X + Y \overset{(\mathrm{inj}_2 \mid \mathrm{inj}_1)}{\longrightarrow} Y + X.$$

4. Iteration

Finally, we describe a construction that allows us to iterate a function.

Definition 3. For each triple of sets X, U, Y, and any function

$$f : X + U + Y \longrightarrow X + U + Y$$

such that

(i) for each $y \in Y$, $f(y) = y$ (the function is *idle* in Y);

(ii) for each $x \in X$ there is an integer n_x such that $f^{n_x}(x) \in Y$ (the function terminates);

we define

$$\mathbf{call}\,[X, U, Y, f] : X \longrightarrow Y, \quad x \mapsto f^{n_x}(x).$$

We will omit the triple X, U, Y whenever it can be determined from the context, and simply write **call** $[f]$.

5. The new definition

In this section we give our new definition of total recursive functions. We leave to Section 9 the minor modifications needed to handle the partial recursive functions.

What we have described is a family of operations for producing new sets out of old ones. If we have a family of sets (the *basic sets*), consider all the sets that can be obtained from them and O, I by repeated application of sums and products (these are the *derived sets*). Now suppose we have a family of (*basic*) functions between derived sets. Any function obtained by repeated use of $(- \mid -)$, $(-,-)$ and composition on the basic functions, injections, projections, identities and δ^{-1} is again a function between derived sets: all functions defined in this way are called *derived* functions.

In what follows the words 'function' and 'set' will mean a derived function or set, unless otherwise specified.

Given a certain class of basic sets and basic functions between derived sets, we can

consider the smallest class of derived functions closed with respect to **call**, that is, the smallest class containing projections, injections, identities and δ^{-1}, and that is closed under composition, $(-,-)$, $(-\mid -)$ and **call**.

The basic functions on which the definition is based are a slight modification of the usual predecessor and successor functions, and are defined as follows:

$$\begin{array}{rcll} p : \mathbf{N} & \longrightarrow & I + \mathbf{N} & (= \{*\} + \mathbf{N}) \\ n & \mapsto & n-1 & \text{if } n \neq 0 \\ 0 & \mapsto & * & \end{array}$$

$$\begin{array}{rcl} s : I + \mathbf{N} & \longrightarrow & \mathbf{N} \\ * & \mapsto & 0 \\ n & \mapsto & n+1. \end{array}$$

Note that with these definitions, s and p are inverse.

Definition 4. The set of δ-recursive functions is the smallest set of functions containing the successor $s : I + \mathbf{N} \longrightarrow \mathbf{N}$ and predecessor $p : \mathbf{N} \longrightarrow I + \mathbf{N}$ maps, projections, injections, identities and δ^{-1} and that is closed under composition, $(-,-)$, $(-\mid -)$ and **call**.

6. Properties of iteration

In this section we state the basic properties of **call**.

Proposition 1. The following properties hold for **call** (we assume that **call** is defined whenever it is mentioned):

(i) For any function $f : X \longrightarrow Y$,

$$f = \mathbf{call}\,[(f \mid \mathrm{inj}_2) : X + Y \longrightarrow X + Y].$$

(ii) If $f : X + U + Y \longrightarrow X + U + Y$ and $g : Y + V + Z \longrightarrow Y + V + Z$, then

$$\mathbf{call}\,[f] \circ \mathbf{call}\,[g] = \mathbf{call}\,[(f + \mathbf{1}_{V+Z}) \circ (\mathbf{1}_{X+U} + g) : X + W + Z \longrightarrow X + W + Z],$$

where $W = U + Y + V$.

(iii) If $f : X + U + Y \longrightarrow X + U + Y$ and $g : X' + U' + Y' \longrightarrow X' + U' + Y'$, then

$$\mathbf{call}\,[f] + \mathbf{call}\,[g] = \mathbf{call}\,[a^{-1} \circ (f + g) \circ a]$$

where a is the obvious commutativity isomorphism

$$(X + X') + (U + U') + (Y + Y') \xrightarrow{\cong} (X + U + Y) + (X' + U' + Y'),$$

and

$$\mathbf{call}\,[f] \times \mathbf{call}\,[g] = \mathbf{call}\,[b^{-1} \circ (f \times g) \circ b],$$

where b is an isomorphism (given by the distributive law)

$$XX' + (XU' + XY' + UU' + UY') + YY' \xrightarrow{\cong} (X + U + Y) \times (X' + U' + Y').$$

(iv) If $f : (X + U + Y) + W + (X + U + Y) \longrightarrow (X + U + Y) + W + (X + U + Y)$, then

$$\textbf{call}\,[\textbf{call}\,[f]] = \textbf{call}\,[(i \mid j \mid k) \circ f] : X \longrightarrow Y,$$

where i is the injection $X + U + Y + W \longrightarrow X + U + Y + W + X + U + Y$, j is the injection of $X + U$ as the first component of $X + U + Y + W + X + U + Y$ and k is the injection of Y as the last component of $X + U + Y + W + X + U + Y$.

Proof. We outline the proof of (ii), leaving the other proofs to the interested reader. Since f is idle in Y, only one of $f + \mathbf{1}_{V+Z}$ and $\mathbf{1}_{X+U} + g$ can be non-idle on a given summand of $X + U + Y + V + Z$. The action of $f + \mathbf{1}_{V+Z}$ will move the state successively from X to Y (through U). At that point, the action of $\mathbf{1}_{X+U} + g$ will move the state from Y to Z (passing through V). □

Note that since $(f \mid g) = \nabla \circ (f + g)$ and $(f, g) = (f \times g) \circ \Delta$, we also have that (**call** $[f]$ | **call** $[g]$) and (**call** $[f]$, **call** $[g]$) can be expressed by a single **call**.

The properties of **call** make it possible to prove the following proposition.

Proposition 2. The set of δ-recursive functions is the set of functions of the form **call** $[f]$, where f is a function derived from the successor s and the predecessor p maps.

Indeed, Proposition 1 shows that it is possible to move each use of **call** in the definition of a δ-recursive function to the outermost level. Then, Part (iv) of Proposition 1 reduces all the outermost applications to a single one.

7. The main theorem

The purpose of this section is to compare the class of classic recursive functions from $\mathbf{N}^k$ to $\mathbf{N}$, and the class of functions generated by the set operation we described in the previous section starting from $s : I + \mathbf{N} \longrightarrow \mathbf{N}$, the successor function, and $p : \mathbf{N} \longrightarrow I + \mathbf{N}$, the predecessor function.

It is, of course, impossible to prove that the two classes coincide, since the domains and codomains of the second class are much more varied (for instance, they contain sums of sets). However, it is clear that any polynomial P built from $\mathbf{N}$, O and I can be mapped injectively to $\mathbf{N}$, and that $\mathbf{N}$ itself can be mapped surjectively to any such polynomial P (except O) using standard arithmetic operations (based upon primitive recursion only; see, for instance, Manin (1977)) – such morphisms can be considered computable from any point of view. Once we add these morphisms to the standard class of recursive functions, we do indeed get the maps generated by s and p. This claim will be made formally clear in what follows.

Note that the predecessor function allows one to test a number against zero: by composing p with a function of the form $f + g$, the computation will continue with f or g, depending on the input of p being null or not.

Consider now the (obvious) isomorphisms $i_+ : \mathbf{N} + \mathbf{N} \longrightarrow \mathbf{N}$, $i_\times : \mathbf{N} \times \mathbf{N} \longrightarrow \mathbf{N}$, the injection $0 : I \longrightarrow \mathbf{N}$, the projection $p : \mathbf{N} \longrightarrow I$ and the identity $\mathbf{1}_N : \mathbf{N} \longrightarrow \mathbf{N}$. It is easy to define them in such a way that both they and their inverses are computable (in fact, up to some trivial encodings, they are primitive recursive; for an explicit definition of $i_\times$,

the well-known *pairing function*, and its inverse, see Manin (1977); the function i_+ can be defined as $(2n \mid 2n+1)$).

By structural induction, we can define for any polynomial P built up from $\mathbf{N}$ and I a specified *computable domain morphism* $P \longrightarrow \mathbf{N}$ using i_+, $i_\times$, $\mathbf{1}_N$ and 0, and a *computable codomain morphism* $\mathbf{N} \longrightarrow P$ using i_+^{-1}, $i_\times^{-1}$, $\mathbf{1}_N$ and p. The composition of a codomain and of a domain morphism is a function $\mathbf{N} \longrightarrow \mathbf{N}$, which is always primitive recursive. (We should also discuss the presence of O, but it can always be eliminated from the polynomial without interfering with the construction of the computable morphisms, unless the polynomial is isomorphic to O; functions having O as domain are obviously computable – there is nothing to compute.)

Note that, for a given polynomial P, the domain and codomain morphisms, which will be denoted d_P and c_P, respectively, enjoy the following obvious property.

Proposition 3. $c_P \circ d_P = \mathbf{1}_P$, $d_P \circ c_P$ is primitive recursive.

Using computable morphisms, we can give a precise definition of the extension of the domain of recursive functions.

Definition 5. The class of *extended recursive function* is the class of functions of the form

$$c_Q \circ f \circ d_P,$$

where $f : \mathbf{N} \longrightarrow \mathbf{N}$ is a recursive function, $c_Q : \mathbf{N} \longrightarrow Q$ is a computable codomain morphism, and $d_P : P \longrightarrow \mathbf{N}$ is a computable domain morphism (for any polynomial of sets P, Q containing $\mathbf{N}$ and I).

Note that in the last definition we restricted the domains of recursive functions to $\mathbf{N}$. This can be done without any loss of generality.

We can now state our main theorem precisely.

Theorem 1. The class of δ-recursive functions coincides with with the class of extended recursive functions.

Proof. We begin by showing that all extended recursive functions are δ-recursive. Clearly, it suffices to show that all recursive functions are δ-recursive.

First of all, let us note that the basic functions mentioned in the definition of recursive function are δ-recursive functions ($\sigma = s \circ \mathrm{inj}_2$, and $0 = s \circ \mathrm{inj}_1$). Substitution is easily obtained using pairing and composition, so the only thing left to prove is primitive recursion and application of the μ operator.

Primitive recursion. We are given functions $g : \mathbf{N}^k \longrightarrow \mathbf{N}$ and $h : \mathbf{N}^{k+2} \longrightarrow \mathbf{N}$, which define by primitive recursion a function $f : \mathbf{N}^{k+1} \longrightarrow \mathbf{N}$, and we want to build a 'for' loop α such that **call** $[\alpha] = f$.

Our state space will be

$$\mathbf{N}^k \times \mathbf{N} + (\mathbf{N}^k \times \mathbf{N} \times \mathbf{N} \times (I + \mathbf{N})) + \mathbf{N},$$

where intuitively the first summand is the input, the second summand is given by

$$\text{input} \times \text{last output} \times \text{recursion index} \times \text{counter}$$

(the recursion index is n in Definition 1; the counter is necessary because it has to run backwards with respect to n), and the last summand is the output.

We will now describe the loop by pointing out how it works on specific pieces of the state space. The first part consists in moving the data out of the initial state space, decrementing the counter, initializing the recursion index to 0 and applying at the same time the 'initial data' function g

$$\mathbf{N}^k \times \mathbf{N} \cong \mathbf{N}^k \times I \times \mathbf{N} \xrightarrow{(1,g)\times 0\times p} \mathbf{N}^k \times \mathbf{N} \times \mathbf{N} \times (I + \mathbf{N})$$

so that we land in the second summand of the state space.
Before defining the action on the second summand, note that

$$\mathbf{N}^k \times \mathbf{N} \times \mathbf{N} \times (I + \mathbf{N}) \cong (\mathbf{N}^k \times \mathbf{N} \times \mathbf{N}) + (\mathbf{N}^k \times \mathbf{N} \times \mathbf{N} \times \mathbf{N}),$$

so we can define the action separately on the summands. Clearly,

$$\mathbf{N}^k \times \mathbf{N} \times \mathbf{N} \xrightarrow{\mathrm{proj}_2} \mathbf{N}$$

projects the result on the final states. Indeed, when the counter lands in the first part of $(I + \mathbf{N})$ we iterated the loop exactly the number of times required.
The following map defines the action on the second summand:

$$\begin{array}{lll} \mathbf{N}^k \times \mathbf{N} \times \mathbf{N} \times \mathbf{N} & \xrightarrow{\Delta\times 1\times\Delta\times 1} & \mathbf{N}^k \times \mathbf{N}^k \times \mathbf{N} \times \mathbf{N} \times \mathbf{N} \times \mathbf{N} \\ & \xrightarrow{1\times 1\times \mathbf{twist}\times 1\times 1} & \mathbf{N}^k \times \mathbf{N}^k \times \mathbf{N} \times \mathbf{N} \times \mathbf{N} \times \mathbf{N} \\ & \xrightarrow{1\times h\times 1\times 1} & \mathbf{N}^k \times \mathbf{N} \times \mathbf{N} \times \mathbf{N} \\ & \xrightarrow{1\times 1\times(s\circ \mathrm{inj}_2)\times p} & \mathbf{N}^k \times \mathbf{N} \times \mathbf{N} \times (I + \mathbf{N}). \end{array}$$

Note that in the end we again land in the local state space.
In order to prove that **call** $[\alpha] = f$, we will explicitly compute the loop α. For an input $(\underline{m}, n)$ we have

$$(\underline{m}, n) \mapsto (\underline{m}, g(\underline{m}), 0, n - 1).$$

If $n = 0$, the next iteration will be

$$(\underline{m}, g(\underline{m}), 0) \mapsto g(\underline{m}) = f(\underline{m}, 0),$$

which correctly computes f. If $n > 0$,

$$\begin{array}{rcl} (\underline{m}, g(\underline{m}), 0, n - 1) & \mapsto & (\underline{m}, \underline{m}, g(\underline{m}), 0, 0, n - 1) \\ & \mapsto & (\underline{m}, \underline{m}, 0, g(\underline{m}), 0, n - 1) \\ & \mapsto & (\underline{m}, h(\underline{m}, 0, g(\underline{m})), 0, n - 1) = (\underline{m}, f(\underline{m}, 1), 0, n - 1) \\ & \mapsto & (\underline{m}, f(\underline{m}, 1), 1, n - 2) \end{array}$$

and, by induction, when α computes $f(\underline{m}, k)$ the counter is decremented to $n - k - 1$, thus moving the state to the final summand.
The μ operator. The function β, which we will use to emulate the μ operator, has as state space

$$\mathbf{N}^k + (\mathbf{N}^k \times \mathbf{N}) + \mathbf{N}$$

and the action on $\mathbf{N}^k$ is defined as $\mathbf{N}^k \xrightarrow{1\times 0} \mathbf{N}^k \times \mathbf{N}$, while on $\mathbf{N}^k \times \mathbf{N}$ it acts as

$$\mathbf{N}^k \times \mathbf{N} \xrightarrow{\Delta} \mathbf{N}^k \times \mathbf{N} \times \mathbf{N}^k \times \mathbf{N}$$

$$\begin{array}{ll}
\xrightarrow{1\times 1\times f} & \mathbf{N}^k \times \mathbf{N} \times \mathbf{N} \\
\xrightarrow{1\times 1\times p} & \mathbf{N}^k \times \mathbf{N} \times (I + \mathbf{N}) \\
\xrightarrow{\delta^{-1}} & (\mathbf{N}^k \times \mathbf{N}) + (\mathbf{N}^k \times \mathbf{N} \times \mathbf{N}) \\
\xrightarrow{\mathrm{proj}_2+(\mathrm{proj}_1,\mathrm{proj}_2)} & \mathbf{N} + (\mathbf{N}^k \times \mathbf{N}) \\
\xrightarrow{\mathbf{twist}} & (\mathbf{N}^k \times \mathbf{N}) + \mathbf{N} \\
\xrightarrow{(1\times(s\circ \mathrm{inj}_2))+1} & (\mathbf{N}^k \times \mathbf{N}) + \mathbf{N}.
\end{array}$$

Thus, if the result of f is 0, the function lands in $\mathbf{N}$, returning the number of iterations; otherwise, it remains in $\mathbf{N}^k \times \mathbf{N}$, incrementing the iteration counter (the second factor of the product).

In more detail, for a given input $\underline{m}$ the first application of β gives

$$\underline{m} \mapsto (\underline{m}, 0),$$

and then we have

$$\begin{array}{rcl}
(\underline{m}, 0) & \mapsto & (\underline{m}, 0, \underline{m}, 0) \\
& \mapsto & (\underline{m}, 0, f(\underline{m}, 0)) \\
& \mapsto & (\underline{m}, 0, f(\underline{m}, 0) - 1).
\end{array}$$

Now, if $f(\underline{m}, 0) = 0$, we simply have

$$(\underline{m}, 0, *) \mapsto 0,$$

otherwise

$$(\underline{m}, 0, f(\underline{m}, 0) - 1) \mapsto (\underline{m}, 0) \mapsto (\underline{m}, 1),$$

so β keeps incrementing the loop counter until f is evaluated to 0. Clearly, $\mathbf{call}\,[\beta] = \mu_n f(\underline{m}, n)$.

Now we will show that all δ-recursive functions are extended recursive functions. The identity function $\mathbf{N} \longrightarrow \mathbf{N}$ can be easily defined by primitive recursion, and Proposition 3 guarantees that the identity on $\mathbf{N}$ can be used for defining the identity on every polynomial P.

The successor function $s : I + \mathbf{N} \longrightarrow \mathbf{N}$ can be analogously defined once the definition of i_+ is made explicit. We will develop this case fully, leaving the remaining basic functions to the interested reader.

Recall that $i_+ = (2n \mid 2n + 1)$. Then, $d_{I+\mathbf{N}} = i_+ \circ (0 + \mathbf{1}_N)$. The recursive function sending even numbers to 1 and odd numbers to the integer part of half of the number, composed with $d_{I+\mathbf{N}}$, is exactly s.

Now, given extended functions $c_A \circ f \circ d_X$ and $c_A \circ g \circ d_Y$, we have

$$\begin{array}{rcl}
(f \mid g) & = & (c_A \circ f \circ d_X \mid c_A \circ g \circ d_Y) \\
& = & (c_A \mid c_A) \circ (f + g) \circ (d_X + d_Y) \\
& = & c_A \circ \nabla_N \circ (f + g) \circ i_+^{-1} \circ i_+ \circ (d_X + d_Y),
\end{array}$$

where $i_+ \circ (d_X + d_Y) = d_{X+Y}$. The recursive function $\nabla_N \circ (f + g) \circ i_+^{-1} : \mathbf{N} \longrightarrow \mathbf{N}$ sends even numbers to $f(n/2)$, and odd numbers to $g((n-1)/2)$. An analogous analysis can be done on $\times$.

The most interesting part of the proof is related to **call** $[f]$. As is typical when proving the computability of recursive functions through an imperative language, the idea behind the proof is that you first build by primitive recursion a function that iterates a loop f any number of times. Then you use μ in order to 'discover' the first iteration landing in the third part of the state space. Finally, you use primitive recursion again in order to iterate the function exactly that number of times.

Take a loop $c_{X+U+Y} \circ f \circ d_{X+U+Y}$ (note that we are ambiguously forgetting the associativity isomorphisms; in this case, complete rigor would only lead to tedious details). As noted previously, $d_{X+U+Y} \circ c_{X+U+Y}$ is a (primitive) recursive function. Thus, we can use the following recursion scheme:

$$\begin{cases} g(n,0) = n \\ g(n,k+1) = (d_{X+U+Y} \circ c_{X+U+Y} \circ f \circ g)(n,k). \end{cases}$$

Obviously the test function $t : \mathbf{N} \longrightarrow \mathbf{N}$ defined by

$$t(n) = \begin{cases} 0 & \text{if } c_{X+U+Y}(n) \in Y \\ 1 & \text{otherwise} \end{cases}$$

is recursive. Then, $\mu_k(t \circ g(n,k))$ is the least number of iterations after which f lands in Y for a given input n, and $h(n) = g(n, \mu_k(t \circ g(n,k)))$ iterates f exactly that number of times. But then,

$$c_Y \circ (\nabla \circ (d_X + d_U + d_Y) \circ c_{X+U+Y}) \circ h \circ (d_{X+U+Y} \circ \mathrm{inj}_1 \circ c_X) \circ d_X$$

is exactly **call** $[f]$ (it is trivial to show that the functions $\nabla \circ (d_X + d_U + d_Y) \circ c_{X+U+Y}$ and $d_{X+U+Y} \circ \mathrm{inj}_1 \circ c_X$ are recursive). □

8. The completeness theorem

It is clear that δ-recursive functions are intuitively computable, from the constructivity of their definition. We have still to show that the class of δ-recursive functions contains *all* intuitively computable functions between polynomials. Church's thesis, that intuitively computable functions from **N** to **N** coincide with recursive functions, allows us to deduce the following

Theorem 2. All intuitively computable functions between polynomials in **N**, O and I are δ-recursive functions.

Proof. Suppose $f : P \longrightarrow Q$ is an intuitively computable function between polynomials. Then, $c_Q^{-1} \circ f \circ d_P^{-1} : \mathbf{N} \longrightarrow \mathbf{N}$ is an intuitively computable function, and, by Church's thesis, a recursive function. But then $c_Q \circ (c_Q^{-1} \circ f \circ d_P^{-1}) \circ d_P = f$ is an extended recursive function and, by Theorem 1, a δ-recursive function. □

9. Partial recursive functions

We will now briefly outline the (single) modification necessary to handle partiality, and state the equivalence theorem between recursive and total partial recursive functions, which admits a very simple proof in this context.

Definition 6. For each triple of sets X, U, Y, and any function

$$f : X + U + Y \longrightarrow X + U + Y$$

such that for each $y \in Y$, $f(y) = y$, we define the *partial* function

$$\mathbf{pcall}\,[X, U, Y, f] : X \longrightarrow Y = \begin{cases} f^{n_x}(x) & \text{if there is an integer } n_x \\ & \text{such that } f^{n_x}(x) \in Y \\ \perp(\text{undefined}) & \text{otherwise} \end{cases}$$

Substituting **pcall** for **call** in the proof of Theorem 1 immediately yields a result analogous to Theorem 1 for partial functions. This fact, coupled with Proposition 2, yields a simple proof of the following well-known but non-trivial theorem.

Theorem 3. The class of total partial recursive functions coincides with the class of recursive functions.

Proof. Obvious, because the use of **call** or **pcall** in the definition of a recursive map f can be reduced to a single, outermost application. □

10. Countably extensive categories

In this section, we want to outline a general setting of which sets and functions are a particular case. The interest of such a discussion lies in the fact that in this general setting it is still possible to define a *function computed by iteration*, that is, it is possible to state the condition under which **call** can be applied, and its effect.

The theoretical basis for such a generalization is given by Category Theory (an alternative categorical generalization has been proposed in Heller (1990)). The categorically informed reader will have already noticed that the notion of *derived function* is best expressed in term of the notion of *arrow in a free distributive category over a given distributive graph.* However, the crucial point for the applicability of **call** is the generalization of the termination condition of Definition 3.

In order to make this condition explicit, we will use the notion of *extensive category* (Carboni *et al.* 1993). An extensive category is one for which the canonical functor

$$\mathbf{E}/A \times \mathbf{E}/B \longrightarrow \mathbf{E}/(A + B)$$

is an equivalence, for each pair of objects $A, B \in \mathbf{E}$. In such categories 'sums behave well', in the sense that whenever we have a map

$$C \xrightarrow{f} A + B,$$

we can pull f back along the injections of A and B into $A + B$ and obtain maps $f_{\lceil A} : f^{-1}(A) \longrightarrow A$ and $f_{B\rceil} : f^{-1}(B) \longrightarrow B$ such that $f_{\lceil A} + f_{B\rceil} = f$. In other words, whenever we have a map into a sum, we can break the source of the map into the part

that maps into the first summand, and the part that maps into the second summand. We will use the notation $\cap$ for the pullback of two summands (complemented subobjects).

Now consider a loop $f : X + U + Y \longrightarrow X + U + Y$ in a countably extensive category with products (*i.e.*, a category that has countable sums, finite products, and enjoys the extensivity property for countable sums). Such a category is necessarily countably distributive (Carboni *et al.* 1993), so the calculus of functions we have described can be reformulated without any modification whatsoever.

Suppose f satisfies $f \circ i_Y = i_Y$, where $i_Y : Y \longrightarrow X + U + Y$ is the canonical injection (this condition is the obvious generalization of the idleness condition). Then

$$X \cap f^{-n}(Y) \cap f^{-(n-1)}(X + U)$$

is the part of X that lands in Y after exactly n iterations. The loop f terminates only if

$$\sum_{n>0} X \cap f^{-n}(Y) \cap f^{-(n-1)}(X + U) \cong X,$$

in which case we can define **call** $[f]$ by describing its restrictions to each summand. In order to do so, we just need to define functions on $f^{-n}(Y)$, and it is natural to choose $f^n{}_{Y]} : f^{-n}(Y) \longrightarrow Y$.

We mention, without developing the theory, that products and countable extensivity are necessary in order to prove Proposition 1. Under these conditions, the calculus enjoys the same properties we have outlined for the category of sets and functions. Thus, a wealth of categories where iteration can be studied is available. Examples include the category of topological spaces and any category of set-valued sheaves. Even in the category of sets, if we vary the distributive graph of basic sets and functions, we obtain new classes of computable functions in a uniform way. For example, if we take as basic sets stacks of a given alphabet A with the operations of push and pop (Walters 1991), the resulting δ-recursive functions will be the recursive functions from A^* to A^*.

In fact, the programming language introduced in Walters (1991), Walters (1992) and Khalil and Walters (1993) computes exactly the δ-recursive functions of its base distributive graph. Hence, the results of this paper provide a proof of the strength of this language.

References

Backus, J. (1978) Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM* **21** (8) 613.

Carboni, A., Lack, S. and Walters, R. F. C. (1993) Introduction to extensive and distributive categories. *Journal of Pure and Applied Algebra* **84** 145–158.

Heller, A. (1990) An existence theorem for recursion categories. *Journal of Symbolic Logic* **55** (3) 1252–1268.

Katis, P., Sabadini, N. and Walters, R. F. C. (1994) The bicategory of circuits. School of Mathematics and Statistics Report 94-22, Sydney University.

Khalil, W. and Walters, R. F. C. (1993) An imperative language based on distributive categories II. *Informatique Théorique et Applications* **27** (6) 503–522.

Manin, Yu. I. (1977) *A Course in Mathematical Logic*, Springer-Verlag.

Sabadini, N., Walters, R. F. C. and Weld, H. (1993) Distributive automata and asynchronous circuits. CTCS '93, Amsterdam. Available by anonymous `ftp` at `maths.su.oz.au` in the directory `sydcat/papers/walters`.

Walters, R. F. C. (1991) *Categories and Computer Science*, Carslaw Publications (also Cambridge University Press (1992)).

Walters, R. F. C. (1992) An imperative language based on distributive categories. *Mathematical Structures in Computer Science* **2** 249–256.