Math. Struct. in Comp. Science (1992), *vol.* 2, *pp.* 249–256

An imperative language based on distributive categories

R. F. C. WALTERS†

School of Mathematics and Statistics, University of Sydney, NSW 2006, Australia.

Received 9 December 1989; revised 11 November 1991

It is the contention of the author that there is a preferred categorical structure appropriate for the analysis of imperative programming languages, namely the existence of finite sums and products and a distributive law of products over sums. An imperative language based on these operations is described.

1. Introduction

This paper contains a definition of imperative program in terms of categories with finite sums and products, and a distributive law of products over sums. A category with this structure has been called in Walters (1989) a *distributive category*. In an early note, Lawvere (1967) pointed out the significance of the distributive law in such constructions as '*if ... then ... else*'. Others who have studied distributive categories include Arbib and Manes (1986) and Cockett (1989). There is an abundance of examples of such categories, since any cartesian-closed category with sums, and hence any topos, is a distributive category.

Briefly, imperative programming is the construction of dynamical systems from a given set of built-in data types and functions using the operations available in a distributive category. The behaviours of a program are the behaviours (orbits) of the dynamical system.

To make the paper as accessible as possible, after a first section of preliminary material on distributive categories, the second section contains a rather informal definition of imperative program with simple examples illustrating various features of our point of view. In the third section the notion of distributive graph is introduced and a precise definition of imperative program is given. The main notion required is that of free distributive category on a distributive graph.

Sufficient details are given to make clear that we are describing a family of programming languages, each of which would be rather straightforward to implement. The family is parameterized by the particular built-in functions of each language.

I would like to thank members of the Sydney Categories in Computer Science Seminar,

† This research was supported in part by an Australian Research Council Program Grant

in particular Michael Johnson, Wafaa Khalil, Gordon Monro, and Bill Unger, for helpful discussion.

2. Distributive categories

For the basic notions of category theory including product and sum (=coproduct) of objects, and initial and terminal objects, see Mac Lane (1971).

In a category with (assigned) finite products, the following operations exist. Given an object X and a terminal object I, there is a unique arrow $!_X : X \to I$. Associated with a product $X \times Y$, there are the projection arrows $p_1 : X \times Y \to X$, $p_2 : X \times Y \to Y$. Given an object X, there is a diagonal arrow $\Delta_X : X \to X \times X$. Given arrows $f_1 : X_1 \to Y_1$, $f_2 : X_2 \to Y_2$, there is the product of the two arrows $f_1 \times f_2 : X_1 \times X_2 \to Y_1 \times Y_2$. Given two objects X, Y, there is an arrow $twist_{X,Y} : X \times Y \to Y \times X$.

The dual operations in a category with (assigned) finite sums are denoted, respectively, $!_X : O \to X$ (O the initial object); the injections $i_1 : X \to X + Y$, $i_2 : Y \to X + Y$; the codiagonal $\nabla : X + X \to X$; the sum of two arrows $f_1 + f_2 : X_1 + X_2 \to Y_1 + Y_2$, and $twist_{X,Y} : X + Y \to Y + X$.

Definition 1. A category is *distributive* if it has assigned finite sums and products, and the following arrows are isomorphisms:

$$!_{X \times O} : O \longrightarrow X \times O \qquad \text{and}$$

$$\nabla_{X \times (Y+Z)}((1_X \times i_1) + (1_X \times i_2)) : X \times Y + X \times Z \longrightarrow X \times (Y + Z).$$

Example 2.1. The category **Sets** of sets and functions is a distributive category.

The one-point set $\{*\}$ is terminal. The empty set is initial. The product of two sets is the cartesian product. The sum is the disjoint union; we denote the elements of $X + Y + Z$ by $(x, 0)$, $(y, 1)$ or $(z, 2)$ depending on which component of the sum the element lies in. The projections of the product, the diagonal arrow, and the injections of the sum are the obvious functions suggested by their names. The codiagonal function ∇_X is given by

$$\begin{array}{rccc} \nabla_X : & X + X & \longrightarrow & X \\ & (x, 0) & \longmapsto & x \\ & (x, 1) & \longmapsto & x. \end{array}$$

The function $twist_{X,Y} : X \times Y \to Y \times X$ takes (x, y) to (y, x); the function $twist_{X,Y} : X + Y \to Y + X$ takes $(x, 0)$ to $(x, 1)$ and $(y, 1)$ to $(y, 0)$.

Finally the isomorphism of the distributive law is given by:

$$\begin{array}{ccc} X \times Y + X \times Z & \longrightarrow & X \times (Y + Z) \\ ((x, y), 0) & \longmapsto & (x, (y, 0)) \\ ((x, z), 1) & \longmapsto & (x, (z, 1)). \end{array}$$

Remark 2.1. We denote $I + I$ as B to indicate that $I + I$ is the Boolean algebra of truth values. We will often denote the truth value *false* by 0, and *true* by 1.

3. An informal definition of imperative program

We begin by describing a simple special case – *isolated* imperative programs; that is, imperative programs with no input.

The notion of imperative program is always relative to some given class of *built-in functions*.

Definition 2. An *isolated imperative program* $\mathscr{P}$ in a language with a given class of built-in functions is a set X, called the state space of the program, and a function $f : X \to X$, constructed from the built-in functions using only the operations of a distributive category – that is, identities, composition, product, sum, and the distributive law. A behaviour of the imperative program is an initial state $x_0 \in X$ and the sequence of states obtainable from that state by repeated application of f; that is the sequence:

$$x_0 \mapsto f(x_0) \mapsto f^2(x_0) \mapsto f^3(x_0) \mapsto \ldots$$

The first example should make clear the intention of this definition.

Example 3.1. A program to calculate $n!$ in terms of built-in functions *multiply* : $\mathbf{Z}\times\mathbf{Z} \to \mathbf{Z}$, *subtract_1* : $\mathbf{Z} \to \mathbf{Z}$, and $test_{x>0} : \mathbf{Z} \to B \quad (= I + I)$.

Take the state space (= the space of global variables) to be $X = \mathbf{Z} \times \mathbf{Z}$ ($\mathbf{Z}$ the integers). We will denote an element of X by (p,k); our intention is that p stand for the partial product and k for the decreasing factor.

Take the initial state to be $x_0 = (1,n)$, n a positive integer. Then the function $f : X \to X$ of the imperative program is given by

$$f(p,k) = \begin{cases} (p \cdot k, k-1), & (k > 0) \\ (p,k), & (k \leq 0) \end{cases}$$

The behavior of the program beginning with initial state $(1,n)$ is the sequence

$$(1,n) \mapsto (1 \cdot n, n-1) \mapsto (1 \cdot n \cdot (n-1), n-2) \mapsto \cdots (n!,0) \to (n!,0) \mapsto \cdots$$

That is, for all sufficiently large m, we have $f^m(1,n) = (n!,0)$, which is what is meant by saying that the program calculates $n!$.

We have not completed the argument. The function $f : X \to X$ must be expressed in terms of the built-in functions using the operations of a distributive category – in fact, the expression of the function in this form *is* the program. But f can be expressed as the following composite:

$$\begin{array}{ccccccc} \mathbf{Z}^2 & \xrightarrow{1_Z \times \Delta_Z} & \mathbf{Z}^3 & \xrightarrow{1_{Z^2} \times test_{x>0}} & \mathbf{Z}^2 \times (I+I) & \xrightarrow{\cong} & \mathbf{Z}^2 + \mathbf{Z}^2 \\ (p,k) & \longmapsto & (p,k,k) & \longmapsto & \begin{cases} (p,k,0) & (k\leq 0) \\ (p,k,1) & (k>0) \end{cases} & \longmapsto & \end{array}$$

$$\begin{array}{ccccc} & \xrightarrow{1_{Z^2}+g} & \mathbf{Z}^2 + \mathbf{Z}^2 & \xrightarrow{\nabla} & \mathbf{Z}^2 \\ & \longmapsto & \begin{cases} (p,k,0) & (k\leq 0) \\ (p\cdot k, k-1, 1) & (k>0) \end{cases} & \longmapsto & \begin{cases} (p,k) & (k\leq 0) \\ (p\cdot k, k-1) & (k>0) \end{cases} \end{array}$$

where $g : \mathbf{Z}^2 \to \mathbf{Z}^2$ is the composite

$$\mathbf{Z}^2 \xrightarrow{1_Z \times \Delta_Z} \mathbf{Z}^3 \xrightarrow{mult \times 1_Z} \mathbf{Z}^2 \xrightarrow{1_Z \times sub\_1} \mathbf{Z}^2.$$

The next two examples are not programs, but functions, which may be used in the construction of programs.

Example 3.2. Any function g between finite sets may be constructed using the operations available in distributive categories, without using any given functions, as follows:

$$m \cdot I = I + I + \cdots + I \xrightarrow{i_{g(1)} + i_{g(2)} + \cdots + i_{g(m)}} n \cdot I + n \cdot I + \cdots + n \cdot I \xrightarrow{\nabla} n \cdot I$$

This means, in particular, that any Boolean function $B^k \to B^l$ is available, and hence tests may be constructed from given tests by Boolean operations.

Example 3.3. Given two functions $f, g : X \to X$ and a test $t : X \to B$, we can construct the function which takes x to $f(x)$ if the test $t(x)$ is satisfied, else it takes x to $g(x)$, as follows:

$$X \xrightarrow{\Delta} X \times X \xrightarrow{1_X \times t} X \times B \xrightarrow{\cong} X + X \xrightarrow{g+f} X + X \xrightarrow{\nabla} X.$$

Of course, this was used in the program in example 3.1.

Using examples 3.2 and 3.3 and appropriate data types, it is clearly possible to construct the usual control structures used in programming.

Example 3.4. This is a further example of an isolated program to illustrate the role of data types. In Walters (1989) we showed how common data types could be expressed in a distributive category. For example, a data type stack of X is an object S with two operation $pop : S \to I + X \times S$ and $push : I + X \times S \to S$ which are mutually inverse.

Now let's construct a program, in terms of the functions *pop* and *push* together with $add\_1 : \mathbf{Z} \to \mathbf{Z}$, to compute the depth of a stack.

Take the state space to be $\mathbf{Z} \times S$. Then the function we would like to construct is:

$$f(k, s) = \begin{cases} (k+1, x_2 x_3 x_4 \cdots x_n), & (s = x_1 x_2 x_3 \cdots x_n) \\ (k, 0), & (s \text{ the empty stack } 0). \end{cases}$$

But this function is the composite of the following three arrows:

$$\mathbf{Z}S \xrightarrow{1_Z + pop} \mathbf{Z} \times (I + XS) \cong \mathbf{Z} + \mathbf{Z}XS$$

$$\mathbf{Z} + \mathbf{Z}XS \xrightarrow{1_Z \times empty + projection} \mathbf{Z}S + \mathbf{Z}S$$

$$\mathbf{Z}S + \mathbf{Z}S \xrightarrow{1_{ZS} + add\_1 \times 1_S} \mathbf{Z}S + \mathbf{Z}S \xrightarrow{\nabla} \mathbf{Z}S.$$

Clearly, for large enough m we have that $f^m(0, s) = (depth(s), 0)$.

In the next example we indicate how programs may be 'composed'.

Example 3.5. (Wafaa Khalil) Suppose $f : X + Y \to X + Y$ is a program. We say that f *idles on* Y if $f i_Y = i_Y$i; that is, $f(y) = y$ if $y \in Y$. Now suppose another program $g : Y + Z \to Y + Z$ idles on Z. Then we can construct a third program with state space $X + Y + Z$ which, with initial state in X, does f until f idles, and then does g until g idles. The program is

$$(1_X + g)(f + 1_Z) : X + Y + Z \to X + Y + Z.$$

Next we define general imperative programs with input. The definition is relative to some built-in functions and an alphabet Σ of inputs.

Definition 3. An *imperative program* with input alphabet Σ is a set X, the state space, and to each element $a \in \Sigma$ a function $f_a : X \to X$ constructed out of the built-in functions using the operations available in a distributive category.

As a first example, let us construct an imperative program out of the same built-in functions as in example 3.1, which allows a positive natural number to be input and which calculates the factorial function.

Example 3.6. Take the state space to be $X = \mathbf{Z}^2 \times (I + I) \cong \mathbf{Z}^2 + \mathbf{Z}^2$. We will call the two components $\mathbf{Z}^2$ of the state space the 0-*mode* and the 1-*mode* of the program. Take the alphabet to be $\Sigma = \{clock\} \cup \mathbf{N}$, ($\mathbf{N}$ the natural numbers).

By the universal property of sums, a function from $h : X \to X$ corresponds to two functions $h_0, h_1 : \mathbf{Z}^2 \to X$. So we need to construct functions $f_{clock,0}$, $f_{clock,1}$, $f_{n,0}$, $f_{n,1}$ ($n = 1, 2, 3, \cdots$). (The index indicates the mode.) The required functions are

$$\begin{aligned} f_{clock,0}(p,k) &= \begin{cases} (p \cdot k, k-1, 0), & (k > 0) \\ (p, k, 1), & (k \leq 0) \end{cases} \\ f_{clock,1}(p,k) &= (p,k,1) \\ f_{n,0}(p,k) &= (p,k,0) \\ f_{n,1}(p,k) &= (1,n,0). \end{aligned}$$

The intention is that in 0-mode, only the clock input is enabled, and the program computes $n!$. When the computation is finished control is passed to the numerical input. If a number is input, the control is passed immediately back to the clock input.
It is clear from our experience with example 3.1 that these functions can be constructed from the built-in functions using the constructions available in a distributive category.

Example 3.7. Example 3.2 shows that if the state space X is finite, all possible functions are available to construct a program. Hence any finite state machine can be modelled by an imperative program. No built-in functions are necessary – only the operations of a distributive category.

The remaining examples of this section indicate how the demands of different agents can be managed in an imperative program.

Example 3.8. Consider two agents, the first of which acts on state space X and produces action $g : X \to X$; the second acts on state space Y and produces action $h : Y \to Y$. We will construct a program which allows both agents to act independently, without conflict for resources.

Take state space $X \times Y$. Take alphabet $\{a, a * b, b\}$. The input a will be the first agent acting alone; the input b will be the second agent acting alone; the input $a * b$ will be the two agents acting simultaneously.
The imperative program consists of three functions on the state space:

$$\begin{aligned} f_a &= g \times 1_Y : X \times Y \to X \times Y \\ f_b &= 1_X \times h : X \times Y \to X \times Y. \end{aligned}$$

$$f_{a*b} = g \times h : X \times Y \to X \times Y.$$

Example 3.9. Consider two agents as in the previous example. We will now construct a program in which the agents are in complete conflict for resources.

Take state space $X + Y$. Take alphabet $\{a, a * b, b\}$. The input a will be the first agent acting alone; the input b will be the second agent acting alone; the input $a * b$ will be the two agents acting simultaneously.

The imperative program consists of three functions on the state space:

$$\begin{aligned} f_a &= g + 1_Y : X + Y \to X + Y \\ f_b &= 1_X + h : X + Y \to X + Y \\ f_{a*b} &= g + h : X + Y \to X + Y. \end{aligned}$$

With this program, whoever is initially in control remains forever in control.

Example 3.10. Again consider the situation of two agents acting, but now on the same state space X. Finally we consider the situation in which the control alternates between the two agents.

Take state space to be $X + X \cong X \times B$. Take alphabet $\{a, a * b, b\}$. The input a will be the first agent acting alone; the input b will be the second agent acting alone; the input $a * b$ will be the two agents acting simultaneously.

The imperative program consists of three functions on the state space:

$$\begin{aligned} f_a &= (twist_{X,X})(g + 1_Y) : X + X \to X + X \\ f_b &= (twist_{X,X})(1_X + h) : X + X \to X + X \\ f_{a*b} &= (twist_{X,X})(g + h) : X + X \to X + X. \end{aligned}$$

4. A precise definition of imperative program

First we need to be more precise about the notion of 'built-in functions'. For this we need the notion of distributive graph (see Sun and Walters (in preparation) for more details).

Definition 4. A *distributive graph* **G** consists of a set of *objects*, denoted A, B, $C, \cdots$ say, together with a set of arrows whose domains and codomains are expressions formed out of the objects using the binary operations '+' and '×' and the nullary operations 'O' and 'I'. A morphism of distributive graphs is a function on arrows and objects which preserves the domains and codomains.

Example 4.1. The following set of (purely formal) arrows is a distributive graph:

$$\{\alpha : A \to I + (B \times A), \quad \beta : I + (B \times A) \to A\}.$$

Example 4.2. Given any distributive category **C** there is an evident underlying distributive graph $\mathscr{U}\mathbf{C}$ whose objects are the objects of **C** and whose arrows are arrows of **C** between expressions made up out of products and sums of objects in **C**.

Let's now consider what a morphism ϕ of distributive graphs from a distributive graph $\mathbf{G}$ to $\mathscr{U}(\mathbf{Sets})$ is. The morphism ϕ assign to each object of $\mathbf{G}$ a set, and to each arrow of $\mathbf{G}$ a function with appropriate domain and codomain. For example, if $\mathbf{G}$ is the graph of example 3.1, then a morphism ϕ from $\mathbf{G}$ to $\mathscr{U}(\mathbf{Sets})$ amounts to two sets $S = \phi A$, $X = \phi B$, and two functions $pop = \phi\alpha : S \to I + XS$, $push = \phi\beta : I + XS \to S$ – the data, but not the equations of a type stack of X.

It is clear that each of the sets of built-in functions of the examples in §3 may be thought of as distributive graph morphisms from a distributive graph $\mathbf{G}$ to $\mathscr{U}(\mathbf{Sets})$. Each imperative language has a particular class of built-in functions; that is, a particular graph $\mathbf{G}$ and a particular graph morphism $\phi : \mathbf{G} \to \mathscr{U}(\mathbf{Sets})$.

The next thing that needs to be made precise is what it means for a function to be constructed from the built-in functions using the operations available in distributive categories. For this we need to know about the free distributive category $\mathscr{F}\mathbf{G}$ on a distributive graph $\mathbf{G}$. This is described in detail in Sun and Walters (in preparation). Briefly, the objects and arrows of $\mathscr{F}\mathbf{G}$ are congruence classes of expressions built up out of the objects and arrows of $\mathbf{G}$ using the operations of a distributive category. The congruence is generated by the equational axioms of a distributive category.
The universal property of $\mathscr{F}\mathbf{G}$ implies that a graph morphism $\phi : \mathbf{G} \to \mathscr{U}(\mathbf{Sets})$ induces a product and coproduct preserving functor $\mathscr{F}\mathbf{G} \to \mathbf{Sets}$.

Given a class of built-in functions $\phi : \mathbf{G} \to \mathscr{U}(\mathbf{Sets})$ we can now define imperative program.

Definition 5. An imperative program consists of an alphabet Σ of *inputs*, and a functor

$$\Gamma : \Sigma^* \to \mathscr{F}\mathbf{G},$$

(Σ^* the free monoid on Σ), which assigns to the inputs the expression of the actions to be performed as a result of the inputs.

Given an imperative program the morphism ϕ induces a functor $\mathscr{F}\mathbf{G} \to \mathbf{Sets}$ and hence we have the composite functor

$$\Sigma^* \to \mathscr{F}\mathbf{G} \to \mathbf{Sets}.$$

This functor assigns to the single object of the free monoid Σ^* a set X, the state space of the program, and to each $a \in \Sigma$ a function $f_a : X \to X$. Hence we arrive at the data of the description of imperative program in §3. It is clear that the functions f_a are expressed in terms of the built-in functions using the operations of a distributive category, since they are images of expressions in $\mathscr{F}\mathbf{G}$.

REFERENCES

Arbib, M. A. and Manes, E. G. (1986) *Algebraic approaches to program semantics*, Springer Verlag.

Cockett, R. (1989) *Distributive logic.* Research Report CS-89-01, Department of Computer Science, University of Tennessee.

Lawvere, F. W. (1967) Theories as categories and the completeness theorem. *J. of Symbolic Logic* **32** 562.

Mac Lane, S. (1971) *Categories for the working mathematician*. Graduate Texts in Mathematics **5**, Springer Verlag.

Sun Shu-Hao and Walters, R. F. C. *Free distributive categories* (in preparation).

Walters, R. F. C. (1989) Data types in distributive categories. *Bull. Austral. Math. Soc.* **40** 79–82.