ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

M P A Sellink (Ed)

2nd International Workshop on the Theory and Practice of Algebraic Specifications, Amsterdam 1997

Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications, Amsterdam, 25-26 September 1997

Towards an Efficient Implementation of Distributive Programs

S. Vigna

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

ISBN: 3-540-76228-0

Towards an Efficient Implementation of Distributive Programs

Sebastiano Vigna

Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano
Milano, Italy

Abstract

Programs and data in distributive categories must be coded in some way in order to be executed on standard architectures. In the original mathematical description of an **IMP**(G) interpreter [KW93] distributive data were coded as words on a suitable set. We present a new efficient coding for such data. This coding allows to exploit the known techniques used in run-time environments for object-oriented systems in order to optimize the execution of **IMP**(G) programs on RAM machines. Moreover, the coding is at the basis of a compiler and bytecode interpreter, which have been described equationally using ASF+SDF.

Keywords: category theory, distributive categories, algebraic specifications, computability.

1 Introduction

Distributive computability is a categorically based notion of computability stemming from the definition of the language **IMP**(G), which was introduced by R.F.C. Walters in [Wal92, KW93]. For any class of “basic functions” (described by a distributive graph), the functions obtained from them using the operators of a distributive category are iterated in order to compute new functions, which form the class of the functions distributively computable over the given class of basic functions (i.e., over the given distributive graph). Iteration here is a very general concept which is expressible in any countably extensive category (for instance, sets, topological spaces, or sheaves). The first equivalence results (with the class of recursive functions) were given in [SVW96], while it was proven in [Vig96] that a particular instance of the distributive model yields the functions computable in the BSS model [BSS89]. These results suggest that the distributive approach encompasses and generalizes many different particular notions of computability.

The definition of the syntax of an **IMP**(G) program is given in an abstract form using free distributive categories. However, any implementation mapping a distributive program (say, over **Set**) onto a standard architecture must face a series of problems in the representation and manipulation of distributive data (i.e., elements belonging to polynomials of sets).

In [KW93] a coding for the data of **IMP**(G) programs has been proposed which represents an element of a set X derived from the basic sets $A, B, \dots$ as a word, using the fact that X is a “subset” of $(A + B + \dots + I)^*$. Correspondingly, a specific coding has been given for the terms built using a certain set of operators which come from the structure of a distributive category. The coding transforms a term in a series of rewriting rules, whose application emulates the behaviour of the function described by the term on a given word (representing an element of the domain of the function).

In this paper, we propose a new coding for distributive data based on labelled forest. The coding has been used in order to implement the first **IMP**(G) compiler and interpreter [Vig95a], and has both space and manipulability advantages over the original coding.

The paper is structured as follows: Section 2 introduces a particular instantiation of the ideas of distributive computability in the category of sets; all definitions are given here in an elementary form. Then we show how to represent elements belonging to a polynomial of sets by means of certain labelled forests, and how to compile arrows into functions between forests.

This paper does not describe in full detail the compiler and the bytecode interpreter; in fact, it is essentially an extended abstract highlighting the most important points of the implementation; the interested reader can find a

complete (and executable) description in [Vig95a]. The ASF+SDF tool [Kli93] has been used in order to specify a context-free grammar and a set of equational rewrite rules to this purpose; a thorough discussion of the equations, unfortunately, goes beyond the space available here. However, in the last section we glimpse through the equations in order to get a feeling of the implementation.

2 An Elementary Definition

In this section we shall give an elementary definition of the operators we shall use in order to build functions in the category of sets.

For any pair of sets X, Y , the *sum* $X + Y$ is the disjoint union of X and Y , i.e., the set $X \times \{0\} \cup Y \times \{1\}$. The canonical injections $\text{inj}_1 : X \rightarrow X + Y$, $\text{inj}_2 : Y \rightarrow X + Y$ 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 , i.e., the set $\{\langle x, y \rangle \mid x \in X \wedge y \in Y\}$. The canonical projections $\text{proj}_1 : X \times Y \rightarrow X$, $\text{proj}_2 : X \times Y \rightarrow Y$ 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 $\emptyset$ and the singleton $I = \{*\}$ are the units of $+$ and $\times$ (up to isomorphism, i.e., up to bijections of sets; we shall not insist in making such isomorphisms explicit). For any set X , there are unique functions

$$\begin{aligned}!_X &: \emptyset \rightarrow X \\ i_X &: X \rightarrow I,\end{aligned}$$

called the *initial* and *terminal* maps.

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

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

Of course, the composition of two functions is still a function. If $f : X \rightarrow Y$, $g : X' \rightarrow Y'$ we write also

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

and

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

Sums and products of sets are related by the *distributivity isomorphism*

$$\delta : X \times Y + X \times Z \xrightarrow{\cong} X \times (Y + Z)$$

which maps $\langle \langle x, w \rangle, k \rangle$ to $\langle x, \langle w, k \rangle \rangle$, with $k = 0$ or 1 , and w an element of Y or Z , respectively. Moreover, any set X has an associated identity map $\mathbf{1}_X : X \rightarrow X$. We shall write Δ_X for the map $(\mathbf{1}_X, \mathbf{1}_X) : X \rightarrow X \times X$.

Now, if we have a family of sets (the *basic sets*), consider all the sets that can be obtained from them and $\emptyset$, I by repeated application of sums and products (these are the *derived sets*). Given 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, initial maps, terminal maps and δ^{-1} is again a function between derived sets: all functions defined in this way are called *derived functions*.

Finally, for each triple of sets X , U , Y , and any derived function

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

such that for each $x \in X$ there is an n_x such that¹ $f^{n_x}(x) \in Y$ (i.e., the function *terminates* for each input) we define the *function which f computes by iteration*, denoted by **call**[X, U, Y, f] (or **call**[f], if X , U and Y are clear from the context), as

$$\mathbf{call}[X, U, Y, f](x) = f^{n_x}(x).$$

Note that if such an n_x exists, it is unique, because we cannot iterate f over an element of Y . The class of *distributively computable functions* is exactly the class of functions of the form **call**[f], with f a derived function.

A fundamental property, called the *normal form theorem* [SVW96], is that the functions computed by iteration, when added to the family of basic functions, do *not* change its computational power. In other words, we can equivalently define the class of distributively computable functions as the class of functions obtained by repeated use of $(- \mid -)$, $(-, -)$, composition and **call**[$-$] on the basic functions, injections, projections, identities, initial maps, terminal maps and δ^{-1} .

Note that without loss of generality we can restrict our attention to derived sets built without using $\emptyset$, and to derived functions whose definition does not use initial maps. This happens because of the *strictness* of $\emptyset$, i.e., because if a function $X \rightarrow \emptyset$ exists then $X = \emptyset$. It is easy to show that if a derived function f contains an initial map then, once its domain and codomain have been normalized using the rules $X + \emptyset = X$ and $X \times \emptyset = \emptyset$, f can be rewritten without using initial maps (unless, of course, f itself is such a map). Since initial maps themselves do not compute anything, we can forget about the empty set and all maps which have it as domain or codomain (by strictness these maps are *all* initial).

3 A New Efficient Coding for $\mathbf{IMP}(G)$ Data

In this section we shall introduce a new coding for $\mathbf{IMP}(G)$ data and programs which is particularly suited to implementation. We shall show how the elements of a derived set can be described as a suitable *forest*, and that it is possible to associate inductively to a derived arrow a function on forests. The basic idea behind the coding is that, in analogy with standard programming, *products represent structures and sums represent union types*. Thus, each time we shall need to represent an element of a sum, we shall *tag* it with the index of its summand.

When discussing such strictly syntactical issues, two possibilities arise: a wordy description, which highlights the main ideas and skips the syntactical details, or a very formal development, which usually turns the reader away. Since the coding has been used in the formal specification of a series of $\mathbf{IMP}(G)$ tools, we shall discuss the main mathematical and algorithmical ideas informally; a fully formal (even executable!) description of the same ideas is given in [Vig95a].

3.1 The Old Coding

Let D be the sum of all the basic sets plus $\{*, e\}$. The *length* $|X|$ of a derived set X is inductively given by $|\emptyset| = |I| = 1$ and $|X + Y| = |X \times Y| = |X| + |Y|$ (if X is seen as a tree, it is just its number of leaves). An element of X is represented by a word over D :

- (i). the only element of I is represented by $*$ ($\emptyset$ has no elements, but we assign it the word e);

¹Note that we shall omit the natural number tagging the element of a sum whenever no confusion is possible, thus writing just x instead of (x, k) .

- (ii). the elements of a basic set are represented by themselves (as one-letter words);
- (iii). the elements of $X \times Y$ are represented by words uv , and the elements of $X + Y$ by words $ue^{|Y|}$ or $e^{|X|}v$, where u and v represent elements of X and Y , respectively.

The distinguished element e is used in order to tell which components of a derived set “do hold” a value. For instance, if $X = A + B$ then the elements of X can be elements of A or elements of B ; correspondingly, a representation of either of the form ae , or of the form eb . It is immediate to see that the length of the word representing an element of X is $|X|$.

3.2 The New Coding

The new coding we are going to propose is based on ordered forests whose leaves are labelled by elements of the basic sets and whose nodes are labelled by natural numbers. We shall identify the empty forest with the only element of I , the one-node forests with the elements of the basic sets, denote by juxtaposition the ordered concatenation of forests and by $\langle n, f \rangle$ the creation of a tree with a root labelled by $n \in \mathbf{N}$ and having as first-level children the trees of the forest f . The precise definition and properties of such a structure, the *free N-motor on the sum of the basic sets*, have been studied in [PQ68, Vig91, KV91]. We shall describe the manipulation of forests informally; the discussion can be recasted in term of algebraic operations and initiality in the category of $\mathbf{N}$ -motors.

We shall define our coding on the basis of a normal form for the derived sets, and consider $+$ and $\times$ as *operators of variable arity* (something which is allowed by associativity). A *flat term (or object)* is defined inductively as:

- (i). a basic set;
- (ii). a list of flat terms surrounded by “+” and “”, or “ $\times$ ” and “” (a $+$ -list and a $\times$ -list, respectively).

We confuse again a flat term and its obvious interpretation as a set, calling both of them *flat objects*.

Now we define a normal form (obtained by associative merge) on the flat objects which optimizes the storage requirements. A *flat object in normal form* is inductively defined as:

- (i). a basic set
- (ii). a $+$ -list of length different from 1, whose items are basic sets and $\times$ -lists in normal form;
- (iii). a $\times$ -list of length different from 1, whose items are basic sets and $+$ -lists in normal form.

In other words, flat objects in normal form do not contain lists with just one element, and it never happens that a sublist can be merged in the containing one by associativity.

For each derived set (or flat object) X there is exactly one corresponding flat object in normal form, obtained by list merging after having applied the mapping ν inductively defined by

$$\begin{aligned}
 \nu(I) &= \times() \\
 \nu(\emptyset) &= +() \\
 \nu(A) &= A \\
 \nu(X + Y) &= +(\nu(X) \nu(Y)) \\
 \nu(X \times Y) &= \times(\nu(X) \nu(Y)).
 \end{aligned}$$

A simple example of flat normal form is given in the following picture:

We shall denote the normal form of a flat object X by $\bar{X}$. By abuse of notation, we shall write $\bar{X}$ for $\overline{\nu(X)}$ when X is a derived set. Of course, many derived sets (or flat objects) correspond to a normal form. Note also that there is a canonical isomorphism between a derived set X , $\nu(X)$ and $\overline{\nu(X)}$.

Now the representation of an element $x \in X$, where X is a flat object, is defined as follows:

- (i). if X is a basic set, x represents itself;
- (ii). if $X = \times(X_1 X_2 \cdots X_p)$, then x is a tuple $\langle x_1, x_2, \dots, x_p \rangle$, and if u_k represents x_k as an element of X_k the representation of x as an element of X is $u_1 u_2 \cdots u_p$;
- (iii). if $X = +(X_1 X_2 \cdots X_p)$, then x is a pair $\langle u, k \rangle$, where $0 \leq k < p$ and u represents x as an element of X_k , and the representation of x as an element of X is $\langle k, u \rangle$.

Given an element x of X , we shall write $\Phi[x]$ for its representation as a forest (following the notation of [KW93]); when X is a derived set, we shall write $\Phi[x]$ for the representation of x as an element of $\nu(X)$. It is immediate that

Proposition 1 For all $x, x' \in X$, $\Phi[x] = \Phi[x']$ implies $x = x'$.

Note that for the sake of notational simplicity we shall write $+(X)$ (or $\times(X)$) for the simple object or $\times$ -list in normal form X (for the simple object or $+$ -list in normal form X , respectively). The notation is not ambiguous, since there are no $+/ \times$ -lists in normal form with just one element, and it allows us to describe in one shot as $+(X_1 \cdots X_p)$ a $+$ -list, when $p > 1$, a $\times$ -list or simple object, when $p = 1$, or the empty set, when $p = 0$ (analogously for $\times(X_1 \cdots X_p)$).

Once we reduce to flat objects in normal form, it is immediate, for instance, to note that the space necessary for storing $n \cdot X^n$ is $O(n^2)$ with the old coding and $O(n)$ with the new one. In general, the storage required for the sum of two flat objects is the sum of the storage for X and Y in the old case, and the maximum (plus one) of the storage for X and Y in the new case.

3.3 Normalization and Associativity

The new coding we just described has a disadvantage: while the associativity isomorphisms related to the product, i.e., the maps

$$X \times (Y \times Z) \xrightarrow{a} (X \times Y) \times Z$$

derived from the projections, satisfy $\Phi[x] = \Phi[a(x)]$ for every element x of X , and moreover $\Phi[x]$ is exactly the representation of x as an element of $\times(X Y Z)$, we have that if

$$X + (Y + Z) \xrightarrow{b} (X + Y) + Z$$

is the analogous isomorphism for the sum and z is an element of Z , $\Phi[z] = \langle 1, \langle 1, z \rangle \rangle$ but $\Phi[b(z)] = \langle 1, z \rangle$. Moreover, the representation of z as an element of $+(X Y Z)$ is $\langle 2, z \rangle$. Thus, the associativity isomorphisms of product induce the identity at the representation level, but this does not happen for the sum.

However, there is a simple way of overcoming this problem. For a forest u , let $\bar{u}$ denote its unique normal form with respect to the (confluent, strongly normalizing) rewrite rule

$$\langle j, \langle k, u \rangle \rangle = \langle j + k, u \rangle.$$

It is straightforward to prove the following

Proposition 2 Let X, Y and Z be flat objects such that $X = +(Y Z)$; let $\bar{Y} = +(Y_1 \cdots Y_q)$ and $\bar{Z} = +(Z_1 \cdots Z_r)$, and suppose $q, r > 0$. Then if x_1 is an element of Y , with representation u_1 in $\bar{Y}$, the representation of $\text{inj}_1(x_1)$ in $\bar{X}$ is $\langle 0, u_1 \rangle$; analogously, if x_2 is an element of Z , with representation u_2 in $\bar{Z}$, the representation of $\text{inj}_2(x_2)$ in $\bar{X}$ is $\langle q, u_2 \rangle$. If $q = 0$ (or $r = 0$) the representation of $\text{inj}_2(x_2)$ (or $\text{inj}_1(x_1)$) is u_2 (u_1 , respectively).

This proposition has several consequences: first of all, it explains how to obtain the representation of an element of X as an element of $\bar{X}$; second, it defines the effect of injections on representations; third, it allows to reverse the effect of the injections:

Proposition 3 With the same notation of Proposition 2, suppose $q, r > 1$; then an element of Y (or Z) injected in $\bar{X}$ is coded as $\langle n, u \rangle$, with $0 \leq n < q$ ($q \leq n < q + r$, respectively). If $q = 1$ ($r = 1$, respectively) its coding as an element of $\bar{Y}$ is u , otherwise it is $\langle n, u \rangle$ ($\langle n - q, u \rangle$, respectively).

The categorical result underlying the previous discussion is the *coherence theorem* for monoidal categories (see [Mac71]). The corollary of interest here is that if we choose a normal form with respect to associativity of $+$ and $\times$, there is a canonical way of making the interpretation of the original term and of the normal form isomorphic.

Thus, we shall suppose that all objects are given in flat normal form (some preprocessing can take care of this fact) and that all arrows are precomposed and postcomposed with the canonical isomorphisms from and to the normal forms (in practice, we shall work in an equivalent category where $+$ and $\times$ happen to be strict). This has the major advantage that all the monoidal isomorphisms (associativity and unit axioms) will induce identities at the representation level.

4 Coding IMP(G) Programs

Recall that an **IMP**(G) program is an arrow in a free distributive category [Vig95a]. For our purposes, it can be thought of as a term built out of the operations described in Section 2. Our goal is to transform it into an operator on forests, which can be actually computed by a (Turing) machine.

4.1 Coding Simple Functions

We now introduce the coding of the simple functions, which is given by stating the effect of the coding on normal form representations. For each basic function a we denote with $\Phi[a]$ the function that a induces on the representations (and defined in the obvious way: if $u = \Phi[x]$ then $\Phi[a](u) = \Phi[a(x)]$). Moreover,

$$\begin{aligned} \Phi[\text{inj}_1^{+(X_1 \cdots X_p), (Y_1 \cdots Y_q)}] : u &\mapsto \overline{\langle 0, u \rangle} \\ \Phi[\text{inj}_2^{+(X_1 \cdots X_p), (Y_1 \cdots Y_q)}] : u &\mapsto \overline{\langle p, u \rangle} \\ \Phi[\text{proj}_1^{\times(X_1 \cdots X_p), \times(Y_1 \cdots Y_q)}] : u_1 \cdots u_p v_1 \cdots v_q &\mapsto u_1 \cdots u_p \\ \Phi[\text{proj}_2^{\times(X_1 \cdots X_p), \times(Y_1 \cdots Y_q)}] : u_1 \cdots u_p v_1 \cdots v_q &\mapsto v_1 \cdots v_q \\ \Phi[i_X] : u &\mapsto \varepsilon \\ \Phi[1_X] : u &\mapsto u \end{aligned}$$

Note that all operations on the right side produce automatically forests in normal form, except for the injections, for which the required normalization is shallow: one just need to check that the rewrite rule $\langle j, \langle k, u \rangle \rangle = \langle j + k, u \rangle$ is applicable at the outermost level. We supposed $p, q > 0$, for otherwise the injections trivialize to identities or initial maps.

The distributivity isomorphism has a more complex description; Proposition 3 tell us how to “unpack” the representation of an element of a sum, and, depending on q or r being equal to one, we have the following four cases:

$$\begin{aligned} \Phi[\delta_{\times(X_1 \cdots X_p), (Y_1 \cdots Y_q), (Z_1 \cdots Z_r)}^{-1}] : u \langle 0, v \rangle &\mapsto \langle 0, uv \rangle && \text{if } q = 1 \\ u \langle k, v \rangle &\mapsto \langle 0, u \langle k, v \rangle \rangle && \text{if } q > 1, k < q \\ u \langle q, v \rangle &\mapsto \langle n, uv \rangle && \text{if } r = 1 \\ u \langle k, v \rangle &\mapsto \langle n, u \langle k - q, v \rangle \rangle && \text{if } r > 1, k \geq q \end{aligned}$$

where we assumed $p, q, r > 0$ and

$$\overline{\times(X_1 \cdots X_p + (Y_1 \cdots Y_q))} = +(S_1 \cdots S_n);$$

note that $n > 0$, and that $n > 1$ implies $p, q = 1$, $Y_1 = *()$ and $X_1 = +(S_1 \cdots S_n)$. If p, q or r is equal to zero, δ^{-1} is the identity or an injection.

4.2 Coding Derived Functions

To each operator used in the construction of derived arrows we associate inductively an operator on forests:

$$\begin{aligned} \Phi\llbracket (f, g) \rrbracket : u &\mapsto \Phi\llbracket f \rrbracket(u) \Phi\llbracket g \rrbracket(u) \\ \Phi\llbracket g \circ f \rrbracket : u &\mapsto (\Phi\llbracket g \rrbracket \circ \Phi\llbracket f \rrbracket)(u). \end{aligned}$$

In the case of the $(- \mid -)$ operator, as for the distributivity isomorphism, we must take some more care. Suppose $\overline{\mathbf{dom}(f)} = +(X_1 \cdots X_p)$, $\overline{\mathbf{dom}(g)} = +(Y_1 \cdots Y_q)$ and $p, q > 0$. Then, following Proposition 3 we have

$$\begin{aligned} \Phi\llbracket (f \mid g) \rrbracket : \langle 0, u \rangle &\mapsto \Phi\llbracket f \rrbracket(u) && \text{if } p = 1 \\ \langle k, u \rangle &\mapsto \Phi\llbracket f \rrbracket(\langle k, u \rangle) && \text{if } p > 1, k < p \\ \langle p, u \rangle &\mapsto \Phi\llbracket f \rrbracket(u) && \text{if } q = 1 \\ \langle k, u \rangle &\mapsto \Phi\llbracket f \rrbracket(\langle k - p, u \rangle) && \text{if } q > 1, k \geq p. \end{aligned}$$

The relevant result here is the following

Theorem 1 Let f be a derived function. Then $\Phi\llbracket f \rrbracket(\Phi\llbracket x \rrbracket) = \Phi\llbracket f(x) \rrbracket$.

Proof. A straightforward but tedious exercise in structural induction and case-by-case analysis. ■

The reader should notice that $\Phi\llbracket - \rrbracket$ sends arrows to (simple) operations of forests. This allows to precompile an arrow into a bytecode which can be easily executed.

5 Some Remarks on the Implementation

We would like to remark that the new coding proposed here can be easily and efficiently implemented on a standard computing machine as follows: each tree is represented by a pointer (either to some basic data representation, or to a forest tagged with a natural number), and a forest is represented by an array of such pointers. It is not difficult to see that this is the standard way of storing variant records, for instance in object-oriented languages with dynamic typing. Thus, the coding presented here is an essential step towards an efficient implementation of distributive programs on standard machines.

It could be argued that the old coding could be “compressed” by representing sequences of the special e symbol as tagged natural numbers. Thus, for instance, the representation of an element of $n \cdot X^n$ would be of the form $e^{j^n} x_1 x_2 \cdots x_n e^{k^n}$, which requires $O(n)$ space. However, this would impose the application of a very complicated “packing/unpacking mechanism” every time a derived arrow has to be computed. Moreover, we would always need, for instance, $O(n)$ storage for I^n (against $O(1)$ in the new coding).

In other words, in real implementations the new coding has always a good advantage (although in some cases it is asymptotically equivalent).

6 A Glimpse at the Implementation

The description given in [Vig95a] of the tools for the manipulation of programs written in **IMP**(G) has been written using ASF+SDF, a system for developing specifications and manipulating rewriting systems created at the University

of Amsterdam [Kli93]. The description, which alternates formal and natural language, is an example of *literate programming* [Knu84], and has been realized using the tool ToL^AT_EX [VK94]. The type-checker has been presented previously in [Vig95b].

The choice of ASF+SDF has the great advantage of making extensions or changes to the syntax of the language very easy, so that one can try, for instance, many different mathematical formalisms (e.g., $f \circ g$, fg or $f; g$ for functional composition). The main disadvantage is the need of large resources, even for running relatively small programs. However, this is not a big issue, since the main point is the implementation of the algorithmical ideas described in the previous sections. The description given in the ASF+SDF syntax, while being very near to the standard mathematical notation, has the advantage of being executable. A direct implementation in a more efficient language would be straightforward.

The reader interested in understanding every detail of the implementation should have some knowledge of the ASF+SDF system, but for a general understanding of the following examples no particular background is necessary. We just remark that in the following the formal syntax is context-free (the rules are written “from right to left”, i.e., they are seen as functions rather than productions), while the equational part specifies rewrite rules with equational conditions. “Execution” in the context of ASF+SDF means that a term is reduced using the given equations, thought of as reduction rules directed from left to right. A rule is applied only if its conditions are all satisfied.

We now highlight two pieces of the system, which show how the operations of forests are defined, and how their composition is optimized locally. The code has been slightly simplified with respect to [Vig95a].

The compilation function $\Phi[-]$ we are going to specify will transform an arrow into a form which can be used by the interpreter. We shall construct inductively a CODE which mimicks the structure of an arrow f , but uses as building blocks simple operations on forests called *instructions*.

The *basic codes* are those instructions which can be executed independently of the basic arrows chosen. Distinguishing them greatly simplifies the rules for compilation and execution. In order to reduce ambiguity, we surround all instructions with braces.

The basic codes described here correspond to the operations on forests used in the inductive definitions of Section 4. The TREE instruction corresponds to the $\langle n, - \rangle$ operator, while RIGHTDEL and LEFTDEL delete the rightmost or leftmost n trees of a forest. Note that the code for the distributivity isomorphism requires four integer arguments, which are the p, q, r and n of Section 4.1.

context-free syntax

BASIC-ARROW	→ BASIC-CODE
“NOP”	→ BASIC-CODE
“INIT”	→ BASIC-CODE
“TERM”	→ BASIC-CODE
“DIST” “[” INT “,” INT “,” INT “,” INT “]”	→ BASIC-CODE
“TREE” “[” INT “]”	→ BASIC-CODE
“RIGHTDEL” “[” INT “]”	→ BASIC-CODE
“LEFTDEL” “[” INT “]”	→ BASIC-CODE
“{” BASIC-CODE “}”	→ CODE
“{” “ITER” “,” CODE “,” INT “,” INT “,” INT “}”	→ CODE
CODE “ ” “[” INT “,” INT “]” CODE	→ CODE
CODE “,” CODE	→ CODE
CODE “;” CODE	→ CODE

After compilation, the term representing an arrow of a distributive category is turned into a term of the grammar we just described, which specifies concrete operations on forests. The ITER code is used in order to iterate an arrow $f : X + U \rightarrow U + Y$, and it requires the number of summands in the flat normal form of X , U and Y . Just to give an example, if $f : X \rightarrow Y$ and $g : Z \rightarrow W + W'$ the arrow $\delta^{-1} \circ (f, g)$ is compiled to

$$\{f\}, \{g\} \quad ; \quad \{\text{DIST}\}[1, 1, 1, 2].$$

The general strategy of the compiler is to create redundant information in the code, which however avoids special cases during compilation. Then, a simple sequence of *peephole optimizations* (see [McK65]) handles special cases and reduces redundancy. Such optimizations are obtained by examining small fragments of code, in order to make them more efficient. The rules we apply, which are based on equations true in every distributive category, can in turn activate other rules, until no other rule is applicable. This technique particularly useful in our case because it allows to eliminate a series of trivial maps which would be very cumbersome to manage during compilation on a case-by-case basis. For instance, composition with `NOP` (i.e., with the identity) can be eliminated:

$$[\text{CONOP1}] \{ \text{NOP} \} ; \gamma = \gamma$$

$$[\text{CONOP2}] \gamma ; \{ \text{NOP} \} = \gamma$$

Iteration starting from the empty set can only produce the initial map, and if the local state space is empty we can apply the normal form theorem [SVW96].

$$[\text{COITER1}] \{ \text{ITER}, \gamma, 0, n, p \} = \{ \text{INIT} \}$$

$$[\text{COITER2}] m \neq 0 \Rightarrow \{ \text{ITER}, \gamma, m, 0, p \} = \gamma$$

Initial and terminal maps enjoy the following properties:

$$[\text{COINIT}] \{ \text{INIT} \} ; \gamma = \{ \text{INIT} \}$$

$$[\text{COTERM}] \gamma ; \{ \text{TERM} \} = \{ \text{TERM} \}$$

Now we have a series of rules which eliminate trivial cases of the $(- \mid -)$ operator in which the arrow really acting is just one. The two parameters correspond to the integers p and q of Section 4.2.

$$[\text{COB1}] \gamma \mid [0, q] \gamma' = \gamma'$$

$$[\text{COB2}] \gamma \mid [p, 0] \gamma' = \gamma$$

The most trivialized map is the distributivity isomorphism. If one of its parameters is zero, it reduces to an identity or to an injection. For instance, the first rule correspond to the case $I \times (X + Y) \xrightarrow{\delta^{-1}} I \times X + I \times Y$, where the flat normal form for both the domain and the codomain is $+(X \ Y)$, and δ^{-1} is the identity, while the second rule correspond to the case $X \times (\emptyset + Y) \xrightarrow{\delta^{-1}} X \times \emptyset + X \times Y$ (with $X \neq I$), where the normal form of the domain is $\times(X \ Y)$, the normal form of the codomain is $+(\times(X \ \emptyset) \times (X \ Y))$ and δ^{-1} is the second injection.

$$[\text{CODIST1}] \{ \text{DIST}[0, q, q', n] \} = \{ \text{NOP} \}$$

$$[\text{CODIST2}] p \neq 0 \Rightarrow \{ \text{DIST}[p, 0, q', n] \} = \{ \text{TREE}[1] \}$$

$$[\text{CODIST3}] p \neq 0 \Rightarrow \{ \text{DIST}[p, q, 0, n] \} = \{ \text{TREE}[0] \}$$

7 Conclusions

We have presented a new efficient coding for distributive data. Moreover, this coding allows to exploit the known techniques used in run-time environments for object-oriented systems in order to optimize the execution of **IMP**(G) programs on RAM machines, due to the strict similarities with the machine representation of variant records (or polymorphic objects). Finally, the coding is at the basis of a compiler and bytecode interpreter, which have been specified equationally. We feel that these results are important steps towards an efficient implementation (mapping) of distributive programs on standard architectures.

References

- [BSS89] Lenore Blum, Mike Shub, and Steve Smale. On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines. *Bull. Amer. Math. Soc.*, 21:1–46, 1989.
- [Kli93] Paul Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.
- [Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984. Reprinted as Chapter 4 of [Knu92].
- [Knu92] Donald E. Knuth. *Literate Programming*. Number 27 in CSLI Lecture Notes. Center for the Study of Language and Information, 1992.
- [KV91] Stefano Kasangian and Sebastiano Vigna. Introducing a calculus of trees. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT/CAAP '91)*, number 493 in LNCS, pages 215–240. Springer-Verlag, 1991.
- [KW93] Wafaa Khalil and Robert F.C. Walters. An imperative language based on distributive categories II. *RAIRO Informatique Théorique et Applications*, 27(6):503–522, 1993.
- [Mac71] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.
- [McK65] W.M. McKeeman. Peephole optimization. *Communications of the ACM*, 8(7):443–444, 1965.
- [PQ68] C. Pair and A. Quère. Définition et étude des bilangages réguliers. *Information and Control*, 13:565–593, 1968.
- [SVW96] Nicoletta Sabadini, Sebastiano Vigna, and Robert F.C. Walters. A note on recursive functions. *Mathematical Structures in Computer Science*, 6, 1996.
- [Vig91] Sebastiano Vigna. Una teoria categoriale degli alberi. Tesi di Laurea. Università degli Studi di Milano, Facoltà di Scienze Matematiche, Fisiche e Naturali, Dipartimento di Matematica, 1991.
- [Vig95a] Sebastiano Vigna. *Distributive Computability*. PhD thesis, Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, 1995.
- [Vig95b] Sebastiano Vigna. Specifying **IMP**(G) using ASF+SDF: A case study. In *Proc. of the ASF+SDF95 workshop on generating tools from algebraic specifications*. Technical Report P9504 of the University of Amsterdam, 1995.
- [Vig96] Sebastiano Vigna. On the relations between distributive computability and the BSS model. *Theoretical Computer Science*, 162:5–21, 1996.
- [VK94] Eelco Visser and Paul Klint. ToL^AT_EX: The program. Programming Research Group, University of Amsterdam, 1994.
- [Wal92] Robert F.C. Walters. An imperative language based on distributive categories. *Mathematical Structures in Computer Science*, 2:249–256, 1992.