J. Symbolic Computation (1989) **8**, 629–650

Codatatypes in ML

Tatsuya HAGINO
Data Processing Center, Kyoto University, Kyoto, Japan.

(Received 25 March 1988)

A new data type declaration mechanism of defining codatatypes is introduced to a functional programming language ML. Codatatypes are dual to datatypes for which ML already has a mechanism of defining. Sums and finite lists are defined as datatypes, but their duals, products and infinite lists, could not be defined in ML. This new facility gives ML the missing half of data types and makes ML symmetric. Categorical and domain-theoretic characterization of codatatypes are also given.

1 Introduction

ML is a strongly typed mostly functional programming language, which was first developed at Edinburgh University in conjunction with the famous proof system, Edinburgh LCF [9]. Even though the principles have not been changed, it has been modified and extended by Edinburgh researchers as well as others over a decade, and we now have several versions of ML.

◇ **DEC-10 ML**: This is the original version which was developed with Edinburgh LCF. It ran on DEC-10 but is no longer in use.
◇ **Cardelli's ML**: Luca Cardelli rewrote DEC-10 ML and introduced several new features. It is written in Pascal and runs on UNIX. ML programs are compiled into an intermediate language called FAM (Functional Abstract Machine). It is optional to assemble FAM codes into native machine codes [7].
◇ **CAM-ML**: Researchers at INRIA have improved the compiler and changed it to generate CAM codes (Categorical Abstract Machine codes) [19, 20]. CAM is an abstract machine language based on the categorical work done by Curien [8].
◇ **Standard ML**: It was designed to consolidate various versions of ML. This was mainly designed by Edinburgh researchers [14]. It is designed for easy transplantation, so it runs on various UNIX machines. This ML also generates FAM codes.

0747–7171/89/120629+22 $03.00/0

ML has mainly been used to write proof systems because it enables elegant implementation (refer [9] for Edinburgh LCF and [22] for Cambridge LCF). However, it has also been used for education because it has various important features computer science students must learn. These features are:

1. Functions are treated as first class objects.
2. It employs *strong typing principle.*
3. It allows *polymorphic typing* for relaxing the restriction of strong typing and retaining the flexibility of untyped languages.
4. Users can define their own data types as *abstract data types.* They can be defined as solutions of recursively defined data type equations and their representation is protected from outside.
5. *Datatypes* are similar to abstract data types except that their representation is not protected and that they can be defined by listing their constructors which can be used for pattern matching.
6. ML programs may cause *exceptions* to exist from deep recursive calls, and exceptions may be caught by exception handlers.
7. ML functions and types may be combined as *modules.*

Among various features of ML, we are interested in one particular feature, the feature of datatypes.

As a related work, the author has developed a functional programming language called CPL (Categorical Programming Language) which is based on category theory [12]. Its principal features are:

1. It has no base types.
2. Types are defined categorically as left and right adjoints. Products, sums and function spaces can be defined in this way.
3. Recursive types like natural numbers and lists can also be defined categorically as initial F-algebras.
4. Dually, final F-coalgebras can be defined.
5. Types are defined with their control structure. For example, boolean is defined with `if` statement control structure.
6. Programs have no variables.
7. Evaluation rules are simple because types are defined uniformly and control structures are associated with types.

In set theory, dualities of mathematical objects are not so apparent. For example, the duality between surjections and injections can not easily be detected from their set theoretic definitions. Neither is the duality between products and sums. However, in category theory, dualities are very important. Sums are products in the dual category (or *opposite category*) which is the mirror image of the original category.

In CPL, the definition of sums is exactly the same as that of products except that arrows point to the other direction. CPL classifies types into two classes, *right objects* and *left objects.* Right objects are types which can be characterized as right adjoints or as final coalgebras, whereas left objects are those which can be characterized as left adjoints or as initial algebras. The following table shows some right objects and some left ones.

right objects	left objects
terminal object	initial object
(= one point set)	(= empty set)
products	sums (= coproducts)
function spaces	natural numbers
(= exponentials)	
infinite lists	finite lists
	binary trees

In some sense, right objects are infinite, whereas left objects are finite. Right objects are often pre-defined in programming languages, whereas left objects are open to users for defining their own data types (natural numbers and lists are usually pre-defined for efficiency, but it is often the case that they can be re-defined as user-defined types).

ML data types correspond to left objects, but ML does not have the mechanism of defining their duals, right objects. In this paper, we introduce a new data type declaration mechanism which enables users to define duals of data types, which we call *co-data types.*

In section 2, we will look back the type system of DEC-10 ML and that of Standard ML and find out why we need co-data types. In section 3, we will introduce the codatatype declaration mechanism with its meaning. In section 4, we will show that codatatypes can be characterized as final coalgebras in category theory. In section 5, a domain-theoretic account of codatatypes will be given, and a set-theoretic one will be given in section 6. In section 7, we will draw some conclusions.

2 Types in ML

In this section, we will investigate the type system of DEC-10 ML and that of Standard ML.

2.1 Types in DEC-10 ML

DEC-10 ML had the following built-in base types.

type	meaning	element
.	unit type	()
`int`	integers	$\ldots, -2, -1, 0, 1, 2, \ldots$
`bool`	booleans	`true` and `false`
`token`	strings	abc, aaaaaa, ...

It also had some other base types for PPLAMBDA (Polymorphic Predicate λ-calculus), but we omit them here. These are base types, and users can create more complex types by combining them together using the following type constructors:

type constructor	meaning	element
`* list`	lists	`[7,19] : int list`
`* # **`	binary products	`(3,'abc') : int # token`
`* + **`	binary sums	`inl(5) : int # **`
`* -> **`	function spaces	`\x.x+1 : int -> int`

where `*` and `**` are type variables. Binary products are associated with the following two polymorphic functions:

```
fst : * # ** -> *
snd : * # ** -> **
```

whereas binary sums are associated with the following five polymorphic functions:

```
 inl : * -> * + **
 inr : ** -> * + **
outl : * + ** -> *
outr : * + ** -> **
 isl : * + ** -> bool
```

Two projections, `outl` and `outr`, may cause exceptions, that is, they are partial functions.

By looking at those functions, we notice asymmetry between binary products and binary sums. Categorically they are dual, but this fact is not reflected in DEC-10 ML. Although elements of binary sums can be constructed by `inl` and `inr`, there are no explicit functions to construct elements of binary products. When two elements, say x and y, of type S and type T are given, we can construct an element of type S `#` T by pairing them as (x, y), but $(\_,\_)$ is not a function in an ordinary sense. Pairing is implicitly embedded into the language.

The ML type system is largely influenced by domain theory. In domain theory, types are domains, and domains are often given as solutions of recursive domain equations. For example, the domain of binary trees whose leaves are integers can be given as the solution of the following domain equation.

$$T \cong Z + T \times T$$

where Z is the domain of integers. Any recursive domain equation involving constant domains, binary products $\times$, binary sums $+$ and function spaces $\rightsquigarrow$ can be solved [24]. This is a very powerful way of defining new domains from already-existing ones.

DEC-10 ML has a mechanism of defining new types in this fashion. For example, an ML type of binary trees can be defined as follows:

```
absrectype btree = int + (btree # btree)
    with leaf n = absbtree(inl n)
     and node(t1,t2) = absbtree(inr(t1,t2))
```

```
and isleaf t = isl(repbtree t)
and leafvalue t = outl(repbtree t)
and left t = fst(outr(repbtree t))
and right t = snd(outr(repbtree t));;
```

The first line is the recursive equation for the type of binary trees and the other lines define primitive functions over binary trees. The isomorphism from `btree` to `int + (btree # btree)` is given as `repbtree` which decomposes binary trees, and the opposite isomorphism from `int + (btree # btree)` to `btree` is given as `absbtree` which creates binary trees. These isomorphisms are available only inside the `absrectype` declaration in order to hide the representation of binary trees.

2.2 Types in Standard ML

In DEC-10 ML, the binary sum type, `* # **`, is given as a built-in primitive. There is no way to express the binary sum type in terms of other primitives. Although the type of integers, `int`, is given as a primitive, it can be defined as an abstract data type (or the solution of the domain equation $Z \cong 1 + Z + Z$).

In Standard ML, however, this is not the case. The binary sum type can be defined by the `datatype` declaration mechanism as follows:

```
datatype 'a + 'b = inl of 'a | inr of 'b;
```

where `'a` and `'b` are type variables. The right-hand side of the declaration lists the constructors of this type with their argument types. The declaration says that an element of `'a + 'b` can be obtained by applying `inl` to an element of `'a` or applying `inr` to an element of `'b`. The declaration also says that these are the only ways to get an element of `'a + 'b`.

Constructors can also be used for pattern matching for doing case analysis. For example, DEC-10 ML's primitive function `isl` can now be defined as follows:

```
fun isl(x) = case x of
               inl(y) => true
             | inr(z) => false;
```

When an element of `'a + 'b` is one which is given by `inl`, then the first case is selected. Otherwise, the second case is selected. This function can also be given as follows:

```
fun isl(inl(y)) = true
  | isl(inr(z)) = false;
```

This looks very much like a Prolog program.

Datatype declarations may be recursive, that is, constructors may refer itself. In Standard ML, the old idea of abstract data types is separated into two: solving recursive equations and hiding representation. The former is captured as the `datatype` declaration mechanism and the latter by `abstype` declaration mechanism. In this paper, we are only interested in the former. The type of binary trees can be defined in Standard ML as follows:

```
datatype btree = leaf of int | node of btree * btree;

exception btree;

fun isleaf(leaf _) = true
  | isleaf(node _) = false;

fun leafvalue(leaf n) = n
  | leafvalue(node _) = raise btree;

fun left(leaf _) = raise btree
  | left(node(t1,t2)) = t1;

fun right(leaf _) = raise btree
  | right(node(t1,t2)) = t2;
```

If we ignore constructor names and treat '|' as the binary sum type '+', datatype declarations are just recursive type equations. Roughly speaking, standard ML has shifted object level '+' into meta level '|'. Although we got rid of the binary sum type from primitives, we still need the product type. From a categorical point of view, this is asymmetric. We will look at the dual of datatypes in the next section.

3 Codatatypes

In this section, we introduce the dual notion of datatypes and get rid of the binary product type from ML. We call this kind of types *codatatypes*. Remember that the general form of datatype declarations is

$$\mathtt{datatype}\ T = c_1\ \mathtt{of}\ A_1 \mid \cdots \mid c_n\ \mathtt{of}\ A_n$$

where T is the type which is going to be defined, $c_1, \ldots, c_n$ are constructors and $A_1, \ldots, A_n$ are domain types of these constructors. T may have some type variables and A_i may refer T recursively.

Each constructor c_i gives a function of the following type.

$$A_i \to T$$

When we think this in the dual category where all the arrows go to the opposite direction, it gives a function of the following type.

$$T \to A_i$$

This can be regarded as a *destructor* of T. It is natural to think that the dual concept of 'constructor' is 'destructor'. Therefore, a codatatype can be declared by listing destructors and their codomains (= dual of domains). The general form is

$$\texttt{codatatype}\ S = d_1\ \texttt{is}\ B_1 \ \&\ \cdots\ \&\ d_n\ \texttt{is}\ B_n.$$

We use the keyword 'is' instead of 'of' and '&' instead of '|'.

It may seem that S is just a record type with n components whose types are $B_1, \ldots, B_n$, but, as we will see shortly, it is not the case when the declaration is recursive.

Let us see some examples of codatatypes. First of all, as we claimed, we can define the binary product type. In category theory, the binary product functor and the binary sum functor are the right and left adjoints of the diagonal functor. Therefore, it is very natural that the binary product type is defined exactly in the same way as the binary sum type except the difference of keywords. Remember that the binary sum type was defined as follows:

```
datatype 'a + 'b = inl of 'a | inr of 'b;
```

Now, the binary product type can be defined as follows:

```
codatatype 'a * 'b = fst is 'a & snd is 'b;
```

We can appreciate the symmetry between '+' and '*'.

The binary product type is associated with two destructors:

```
fst: 'a * 'b -> 'a
```

gives the projection function of the first component, and

```
snd: 'a * 'b -> 'b
```

gives the projection function of the second component.

For datatypes, we had `case` statements as their control structure. They decompose datatypes. As their dual, for codatatypes, we will have `merge` statements. They are the control structure for codatatypes. They create elements of codatatypes. For a codatatype declared by

$$\texttt{codatatype}\ S = d_1\ \texttt{is}\ B_1 \ \&\ \cdots\ \&\ d_n\ \texttt{is}\ B_n$$

`merge` statements have the following form.

$$\texttt{merge}\ d_1 \Leftarrow e_1 \ \&\ \cdots\ \&\ d_n \Leftarrow e_n$$

where $d_1, \ldots, d_n$ are destructors and $e_1, \ldots, e_n$ are expressions of type $B_1, \ldots, B_n$, respectively.

If we write $e\colon A$ for e having type A, we have the following typing rules for the codatatype declared above

$$d_i\colon S \to B_i$$

$$\frac{e_1 : B_1 \quad \ldots \quad e_n : B_n}{(\texttt{merge}\ d_1 \Leftarrow e_1\ \&\ \cdots\ \&\ d_n \Leftarrow e_n) : S}$$

For the binary product type '*', merge statements just pair elements.

$$\texttt{merge fst} \Leftarrow x\ \&\ \texttt{snd} \Leftarrow y$$

can be regarded as (x, y). If we got rid of the binary product type from ML, (x, y) would no longer be in the ML syntax. Therefore, we might take this to be the definition of (x, y).

As an another example of codatatypes, let us define the type of infinite lists. Remember that the type for finite lists is defined by the following datatype declaration.

```
datatype 'a list = nil | cons of 'a * 'a list;
```

Nil and cons are the list constructors. List destructors are head and tail functions defined as follows:

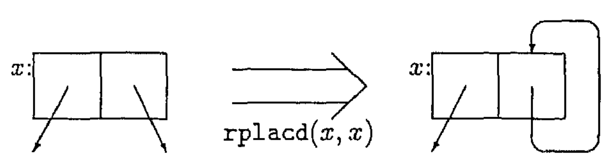
```
fun head(nil) = raise head
  | head(cons(x,l)) = x;

fun tail(nil) = raise tail
  | tail(cons(x,l)) = l;
```

For codatatypes, constructors and destructors play the opposite roles. We declare the type of infinite lists by listing the two destructors.

```
      codatatype 'a inflist = head is 'a & tail is 'a
inflist;
```

This tells us that an element of 'a inflist has two components, its head is an element of 'a and its tail is again an element of 'a inflist. This definition seems to be recursive without bottom. Unless we obtain an element of 'a inflist out of somewhere, we cannot make any elements of 'a inflist. In LISP, infinite lists can be obtained by destructive pointer manipulation using rplaca and rplacd.



This is possible because infinite lists and finite lists are the same and destructive operations are allowed. Form a theoretical point of view, it is very difficult to handle destructive operations[18]. Therefore, we do not regard this as a proper way of making infinite lists.

Here, we use merge statements to create infinite lists. For example, the infinite sequence of 1 can be given as the result of evaluating the following function.

```
fun iseq1() = merge head <= 1
                  & tail <= iseq1();
```

Note that merge statements cannot simply make records after evaluating components because the above function would never terminate if they did. Therefore, the evaluation mechanism of merge statements needs to be lazy. Here, we take the following evaluation strategy: when

$$\mathtt{merge}\ d_1 \Leftarrow e_1 \,\&\, \cdots \,\&\, d_n \Leftarrow e_n$$

is evaluated, it immediately returns a record which has n empty components; when one of the components, say ith one, is accessed by a destructor d_i first time, e_i is evaluated and the result is stored in the ith component as well as returned as the value of d_i; next time when the same component is accessed by d_i, it simply returns the value stored in the ith component.

Therefore, when iseq1 is evaluated, it returns an infinite list as a record of two components. Let this record be s. We cannot see the components of s unless we access them by head and tail. If we try to see them, head(s) will be 1 and tail(s) will be another infinite list which is identical to s.

In the above, iseq1 is given as a function, but one may want to write it as follows:

```
val rec iseq1 = merge head <= 1 & tail <= iseq1;
```

Usually we cannot make recursive definitions unless they define functions. Here, we make recursive definitions for infinite lists. This is allowed because the evaluation mechanism for merge statements is as lazy as it is for function closures. In fact, function spaces and codatatypes are closely related. Categorically, they are characterized as right adjoints (whereas natural numbers and lists are characterized as left objects, and we cannot make recursive definitions over them).

Let us write some functions over infinite lists.

```
fun comb(l1,l2) = merge head <= head l1
                      & tail <= comb(l2,tail l1);
```

This function combines two infinite lists together and makes one infinite list by picking up elements alternately from two infinite lists.

$$\begin{tikzcd}[column sep=small] (a_1\ a_2\ a_3\ \ldots) \arrow[dr] & \\ & \mathtt{comb} \rightarrow (a_1\ b_1\ a_2\ b_2\ a_3\ b_3\ \ldots) \\ (b_1\ b_2\ b_3\ \ldots) \arrow[ur] & \end{tikzcd}$$

```
fun nth(l,n) = if n = 0 then head l
               else nth(tail l,n-1);
```

nth(l,n) returns the nth element of l.

```
fun sum(l) =
    let fun sum1(l,s) = merge head <= s
                            & tail <= sum1(tail l,s+(head l))
    in sum1(l,0)
    end;
```

sum(l) gives the partial sum sequence of l, that is, the nth element of sum(l) is the sum of 1st $\sim$ $(n-1)$th elements of l.

$$(a_1\ a_2\ a_3\ \ldots a_n \ldots) \to \mathtt{sum} \to (0\ a_1\ a_1 + a_2 \ldots \sum_{i=1}^{n-1} a_i \ldots)$$

We can summerize the symmetry between datatypes and codatatypes by the following table.

	datatypes	codatatypes
declaration	datatype $T =$ c_1 of $A_1 \mid \cdots \mid c_n$ of A_n	codatatype $S =$ d_1 is B_1 & $\cdots$ & d_n is B_n
primitives	constructors c_i	destructors d_i
control	case statements	merge statements
typing	$c_i: A_i \to T$	$d_i: S \to B_i$
	$\dfrac{e: T \quad e_i: A_i \to B}{(\mathtt{case}\ e\ \mathtt{of}\ c_1 \Rightarrow e_1 \mid \cdots \mid c_n \Rightarrow e_n): B}$	$\dfrac{e_i: B_i}{(\mathtt{merge}\ d_1 \Leftarrow e_1\ \&\ \cdots\ \&\ d_n \Leftarrow e_n): S}$

If we look at the typing rules, case statements and merge statements are not exactly dual. If we want to make them exactly dual, we may use a different form of merge statements.

$$\frac{e: C \qquad e_i: C \to B_i}{(\mathtt{merge}'\ d_1 \Leftarrow e_1\ \&\ \cdots\ \&\ d_n \Leftarrow e_n\ \ \mathtt{of}\ \ e): S}$$

However, merge′ can be expressed in terms of merge as follows:

$$\begin{aligned} &\mathtt{merge}'\ d_1 \Leftarrow e_1\ \&\ \cdots\ \&\ d_n \Leftarrow e_n\ \ \mathtt{of}\ \ e \\ &\quad \equiv \mathtt{let}\ \mathtt{val}\ x = e \\ &\qquad \mathtt{in}\ \mathtt{merge}\ d_1 \Leftarrow e_1(x)\ \&\ \cdots\ \&\ d_n \Leftarrow e_n(x) \\ &\qquad \mathbf{end} \end{aligned}$$

Therefore, we chose the simpler form. This fact that merge statements are simpler than case statements resembles the fact that the natural deduction rule for logical 'and' is simpler that that of logical 'or'. Categorically, this is just the same thing happening in different categories.

4 Categorical View of Codatatypes

In [12], the author developed a functional programming language called CPL which is based on category theory. The concept of codatatypes arose from this work. Category theory has developed a powerful concept called *adjunctions*. It is very simple, yet it unifies a lot of seemingly unrelated concepts together and reveals true nature of mathematical objects. The binary product functor and the binary sum functor (or the binary coproduct functor) are exactly dual. The product functor is the right adjoint of the diagonal functor whereas the sum functor is the left adjoint of the same functor.

CPL has two kinds of declarations of types (or functors), right objects and left objects. They are based on right adjoints and left adjoints, respectively, and characterized by the following concept.

Definition 4.1: Let $\mathcal{C}$ and $\mathcal{D}$ be categories and both F and G be functors from $\mathcal{C}$ to $\mathcal{D}$.

$$\begin{tikzcd} \mathcal{C} \arrow[r, shift left, "F"] \arrow[r, shift right, "G"'] & \mathcal{D} \end{tikzcd}$$

We define an F, G-dialgebras as follows:

1. its objects are pairs $\langle A, f \rangle$ where A is a $\mathcal{C}$ object and f is a $\mathcal{D}$ morphism of $F(A) \to G(A)$, and
2. its morphisms $h: \langle A, f \rangle \to \langle B, g \rangle$ are $\mathcal{C}$ morphisms $h: A \to B$ such that the following diagram commutes.

$$\begin{tikzcd} F(A) \arrow[r, "f"] \arrow[d, "F(h)"'] \arrow[dr, phantom, "\circlearrowleft"] & G(A) \arrow[d, "G(h)"] \\ F(B) \arrow[r, "g"'] & G(B) \end{tikzcd}$$

In the case where F or G is contravariant, we have to modify the direction of some arrows.

It is easy to show that it is a category; let us write $\mathbf{DAlg}(F, G)$ for it. $\square$

This extends the notion of F-algebras which are often used in domain theory [25].

Proposition 4.2: For an endo-functor $F: \mathcal{C} \to \mathcal{C}$, $\mathbf{DAlg}(F, \mathbf{I})$ is the category of F-algebras. Dually, $\mathbf{DAlg}(\mathbf{I}, F)$ is the category of F-coalgebras. $\square$

If we parametrize the definition, it also extends the notion of right and left adjoints (see [12]).

Let us use F, G-dialgebras to explain datatypes and codatatypes. In the following, let $\mathcal{C}$ be the category of ML types. If T is a datatype given by

$$\texttt{datatype } T = c_1 \texttt{ of } A_1 \mid \cdots \mid c_n \texttt{ of } A_n,$$

T is a type which is an object of $\mathcal{C}$ and A_i is a type expression which can be regarded as an endo-functor of $\mathcal{C}$ with respect to T (T may appear in A_i). Let F and G be functors from $\mathcal{C}$ to $\mathcal{C}^n$ ($= \underbrace{\mathcal{C} \times \cdots \times \mathcal{C}}_{n}$) defined as follows:

$$F(X) \equiv \langle A_1(X), \ldots, A_n(X) \rangle \qquad G(X) \equiv \langle \underbrace{X, \ldots, X}_{n} \rangle$$

$$\mathcal{C} \underset{\langle X, \ldots, X \rangle}{\overset{\langle A_1(X), \ldots, A_n(X) \rangle}{\rightrightarrows}} \underbrace{\mathcal{C} \times \cdots \times \mathcal{C}}_{n}$$

Then, T and $c_1, \ldots, c_n$ can be characterized as the initial object $\langle T, \langle c_1, \ldots, c_n \rangle \rangle$ of $\mathbf{DAlg}(F, G)$. From definition 4.1, $\langle c_1, \ldots, c_n \rangle$ is a morphism

$$\langle c_1, \ldots, c_n \rangle : F(T) \equiv \langle A_1(T), \ldots, A_n(T) \rangle \rightarrow G(T) \equiv \langle T, \ldots, T \rangle,$$

that is, each c_i is the following morphism.

$$c_i : A_i(T) \equiv A_i \rightarrow T$$

This coincide with the type of c_i in ML.

Next, let us give the meaning to the following case statement.

$$\texttt{case}\ \ e\ \ \texttt{of}\ \ c_1 \Rightarrow e_1 \mid \cdots \mid c_n \Rightarrow e_n \tag{+}$$

e is an expression of type T, so it can be regarded as a morphism from 1 to T (where 1 is the terminal object of $\mathcal{C}$), and e_i is an expression of type $A_i \rightarrow B$ which can be regarded as a morphism of the same type.

$$e : 1 \rightarrow T \qquad e_i : A_i \rightarrow B$$

Then, $\langle c_i, e_i \rangle \circ A_i(\pi_1)$ gives a morphism of

$$A_i(T \times B) \longrightarrow T \times B$$

where π_1 is the first projection associated with the product $T \times B$.

$$\begin{tikzcd} & & T \\ A_i(T \times B) \arrow[r, "A_i(\pi_1)"] & A_i(T) \arrow[ur, "c_i"] \arrow[r, "\langle c_i, e_i \rangle"] \arrow[dr, "e_i"'] & T \times B \\ & & B \end{tikzcd}$$

Therefore,

$$\langle T \times B, \langle \langle c_1, e_1 \rangle \circ A_1(\pi_1), \ldots, \langle c_n, e_n \rangle \circ A_n(\pi_1) \rangle \rangle$$

is an object of $\mathbf{DAlg}(F, G)$. Since $\langle T, \langle c_1, \ldots, c_n \rangle \rangle$ is the initial object of $\mathbf{DAlg}(F, G)$, there is a unique $\mathbf{DAlg}(F, G)$ morphism

$$\langle T, \langle c_1, \ldots, c_n \rangle \rangle \rightarrow \langle T \times B, \langle \langle c_1, e_1 \rangle \circ A_1(\pi_1), \ldots, \langle c_n, e_n \rangle \circ A_n(\pi_1) \rangle \rangle.$$

From definition 4.1, this is a $\mathcal{C}$ morphism from T to $T \times B$. Let us write it as follows:

$$\psi(\langle c_1, e_1\rangle \circ A_1(\pi_1), \ldots, \langle c_n, e_n\rangle \circ A_n(\pi_1)).$$

Then, the meaning of the case statement (+) can be given by the following morphism.

$$\pi_2 \circ \psi(\langle c_1, e_1\rangle \circ A_1(\pi_1), \ldots, \langle c_n, e_n\rangle \circ A_n(\pi_1)) \circ e \qquad (++)$$

where π_2 is the second projection of $T \times B$. This morphism goes from 1 to B, which coincides with the type of the case statement, B.

Let us check whether this morphism really satisfies the desired properties.

Proposition 4.3: Because $\psi(f_1, \ldots, f_n)$ gives unique morphisms in the category of F, G-dialgebras, it satisfies the following equations.

$$\begin{cases} \psi(f_1, \ldots, f_n) \circ c_i = f_i \circ A_i(\psi(f_1, \ldots, f_n)) \\ \psi(c_1, \ldots, c_n) = id \\ \forall i (h \circ c_i = f_i \circ A_i(h)) \Rightarrow h = \psi(f_1, \ldots, f_n) \end{cases} \qquad \square$$

Proposition 4.4: $\pi_1 \circ \psi(\langle c_1, e_1\rangle \circ A_1(\pi_1), \ldots, \langle c_n, e_n\rangle \circ A_n(\pi_1)) = id$
Proof: Let h be $\psi(\langle c_1, e_1\rangle \circ A_1(\pi_1), \ldots, \langle c_n, e_n\rangle \circ A_n(\pi_1))$. Then, from proposition 4.3,

$$\begin{aligned} \pi_1 \circ h \circ c_i &= \pi_1 \circ \langle c_i, e_i\rangle \circ A_i(\pi_1) \circ A_i(h) \\ &= c_i \circ A_i(\pi_1 \circ h). \end{aligned}$$

From proposition 4.3 (the uniqueness of ψ),

$$\pi_1 \circ h = \psi(c_1, \ldots, c_n) = id. \qquad \square$$

Now, we have the following proposition.

Proposition 4.5: $\pi_2 \circ \psi(\langle c_1, e_1\rangle \circ A_1(\pi_1), \ldots, \langle c_n, e_n\rangle \circ A_n(\pi_1)) \circ c_i = e_i$.
Proof:

$$\begin{aligned} &\pi_2 \circ \psi(\langle c_1, e_1\rangle \circ A_1(\pi_1), \ldots, \langle c_n, e_n\rangle \circ A_n(\pi_1)) \circ c_i \\ &= \pi_2 \circ \langle c_i, e_i\rangle \circ A_i(\pi_1) \circ A_i(\psi(\langle c_1, e_1\rangle \circ A_1(\pi_1), \ldots, \langle c_n, e_n\rangle \circ A_n(\pi_1))) \\ &= e_i \circ A_i(\pi_1 \circ \psi(\langle c_1, e_1\rangle \circ A_1(\pi_1), \ldots, \langle c_n, e_n\rangle \circ A_n(\pi_1))) \\ &= e_i \end{aligned} \qquad \square$$

Therefore, if e is an element of T which is created by the constructor c_i, e is $c_i \circ e'$, and, from this proposition, (++) is equal to $e_i \circ e'$. Hence,

Proposition 4.6: If we use $[\![\ ]\!]$ for denoting the meaning of programs,

$$[\![\texttt{case } c_i(e) \texttt{ of } c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n]\!] = [\![e_i]\!]([\![e]\!]). \qquad \square$$

Next, let us consider codatatypes. Let S be a codatatype given as follows:

$$\texttt{codatatype } S = d_1 \texttt{ is } B_1 \mathbin{\&} \cdots \mathbin{\&} d_n \texttt{ is } B_n$$

B_i can be regarded an endo-functor of $\mathcal{C}$. We put F and G as follows:

$$F(X) \equiv \langle \underbrace{X, \ldots, X}_{n} \rangle \qquad G(X) \equiv \langle B_1(X), \ldots, B_n(X) \rangle$$

Then, $\langle S, \langle d_1, \ldots, d_n \rangle \rangle$ can be characterized as the final F,G-dialgebra (or the final object in $\mathbf{DAlg}(F,G)$). The meaning of merge statements can be given as follows.

Definition 4.7:

$$\begin{aligned} &[\![\texttt{merge } d_1 \Leftarrow e_1 \mathbin{\&} \cdots \mathbin{\&} d_n \Leftarrow e_n]\!] \\ \equiv\ &\phi(B_1(\nu_1) \circ [d_1, e_1], \ldots, B_n(\nu_1) \circ [d_n, e_n]) \circ \nu_2 \end{aligned}$$

where ϕ gives unique arrows, ν_1 and ν_2 are injections of $S+1$, and $[d_i, e_i]$ is the unique arrow from $S+1$ to A_i. []

For codatatypes, we have the following propositions.

Proposition 4.8:

$$\begin{cases} d_i \circ \phi(f_1, \ldots, f_n) = B_i(\phi(f_1, \ldots, f_n)) \circ f_i \\ \phi(d_1, \ldots, d_n) = id \\ \forall i (d_i \circ h = B_i(h) \circ f_i) \Rightarrow h = \phi(f_1, \ldots, f_n) \end{cases} \qquad []$$

Proposition 4.9: $\phi(B_1(\nu_1) \circ [d_1, e_1], \ldots, B_n(\nu_1) \circ [d_n, e_n]) \circ \nu_1 = id$. []

Proposition 4.10: $d_i \circ \phi(B_1(\nu_1) \circ [d_1, e_1], \ldots, B_n(\nu_1) \circ [d_n, e_n]) \circ \nu_2 = e_i$. []

Therefore,

Proposition 4.11: $[\![d_i(\texttt{merge } d_1 \Leftarrow e_1 \mathbin{\&} \cdots \mathbin{\&} d_n \Leftarrow e_n)]\!] = [\![e_i]\!]$. []

We have shown that datatypes can be characterized as initial F,G-algebras and that codatatypes can be characterized as final F,G-dialgebras.

Note that case and merge statements do not straightforwardly correspond to categorical morphisms compared with constructors and destructors. In CPL, we took the categorical versions of case and merge, so we could see much more natural correspondence between datatypes/codatatypes and initial/final F,G-dialgebras.

Note also that we did not take recursive definitions of functions into consideration. ML allows general recursion, but categorically it is rather complicated and CPL does not allow general recursion but only primitive recursion.

We also assumed that A_i and B_i are covariant functors. If they involve function spaces, they might not be covariant functors. We cannot handle non-covariant functors by F,G-dialgebras.

At the beginning of this section, we fixed a category $\mathcal{C}$. One may wonder what it is. We assumed that it is the category of ML types, but the number of ML types increases as we declare new datatypes and codatatypes. Therefore, one may wonder whether $\mathcal{C}$ is fixed or not. For example, if we define product first and then coproduct, is this the same as to define coproduct first and then product? Are we using the same $\mathcal{C}$?

There may be two approaches for this problem. In the first approach, we regard that $\mathcal{C}$ is given by the God and never changes. It is the universe of the ML data types. It contains all the data types and co-data types we can define in ML. For example, $\mathbf{CPO}_\perp$ we will present in the next section is such a category. F, G-dialgebras do not add new objects but pick up existing objects. As we prove in domain theory the existence of solutions of recursive domain equations, we have to prove that initial and final objects of F, G-dialgebras exist. We have the following result.

Proposition 4.12: If F is continuous and G has the left adjoint, there exists the initial F, G-dialgebra. Dually, if G is co-continuous and F has the right adjoint, there exists the final F, G-dialgebra.
proof: See [12]. []

This proposition puts some restrictions to F and G, but it is easy to see that F and G for datatypes and codatatypes satisfy these conditions. In this apporach, because $\mathcal{C}$ is fixed, the order of declarations does not matter very much. Declaring products before coproducts and declaring coproducts before products are the same. They just pick up products and coproducts of $\mathcal{C}$ (e.g. $\mathbf{CPO}_\perp$). Of course, we have to declare products before lists because the declaration of lists depends on that of products.

We may also take the other approach. In this approach, $\mathcal{C}$ changes as we declare new datatypes and codatatypes. We regard initial and final F, G-dialgebra characterization as specification of categories. Since this characterization is given by equations (see proposition 4.3 and 4.8), we can device a specification language similar to an algebraic one (see [12] about this specification language). We can translate initial and final F, G-dialgebra conditions into equational specifications. In this specification language, a model is a category associated with appropriate functors and satisfying equations. Therefore, the category of models is a category of categories. When nothing is declared, the model category is the category of (small) categories. When the terminal object, the binary product and the exponential are declared, the model category is the category of cartesian closed categories. We can prove that every model category has the initial model and we can exactly follow the initial algebra approach to give meaning to datatypes and codatatypes (see [12] for more details). In this approach, declaring products before coproducts may differ from declaring coproducts before products. We have to prove that two models are isomorphic. This should follow from the fact that two declarations are independent.

5 Domain Theoretic View of Codatatypes

The abstract data types in DEC-10 ML were derived from the domain theoretic idea of recursive domain equations. In domain theory, domains other than primitive ones are defined by solving recursive domain equations.

$$D \cong F(D)$$

For example, the domain of natural numbers can be given as the solution of the following equation.

$$N \cong 1 + N$$

In this paper, we use the category $\mathbf{CPO}_\perp$ of complete partial orders with the least element and strict continuous functions. In this category, the initial object 0 is $\{ \perp \}$, the final object 1 is the same, the product $A \times B$ is given by $\{ (a, b) \mid a \in A, b \in B \}$ with

$$(a, b) \sqsubseteq (a', b') \iff a \sqsubseteq a' \wedge b \sqsubseteq b',$$

and the sum $A + B$ is given by

$$(\{ 0 \} \times (A \setminus \{ \perp \}) \cup \{ 1 \} \times (B \setminus \{ \perp \})) \cup \{ \perp \}$$

with

$$\begin{cases} (0, a) \sqsubseteq (0, a') & (a, a' \in A \text{ and } a \sqsubseteq a') \\ (1, b) \sqsubseteq (1, b') & (b, b' \in B \text{ and } b \sqsubseteq b') \\ \perp \sqsubseteq c & (c \in A + B). \end{cases}$$

However, it does no have the categorical function space. Instead, it has the strict function space $A \to_\perp B$ of strict continuous functions which is the right adjoint to the following smash product $A \otimes B$.

$$A \otimes B \equiv \{ (a, b) \in A \times B \mid a = \perp \iff b = \perp \}$$

It also has the lifting functor $A_\perp$ which adds the new least element to A.

In general, when $F: \mathbf{CPO}_\perp \to \mathbf{CPO}_\perp$ is continuous, the initial fixed point of F can be given as the colimit of the following diagram.

$$0 \to F(0) \to F^2(0) \to F^3(0) \to \cdots \to F^n(0) \to \cdots$$

This diagram may not be in $\mathbf{CPO}_\perp$ but in $\mathbf{CPO}_\perp^E$ which is the associated category of embeddings. The datatype T declared by

$$\texttt{datatype } T = c_1 \texttt{ of } A_1(T) \mid \cdots \mid c_n \texttt{ of } A_n(T)$$

can be regarded as the initial fixed point of the following functor.

$$F(X) \equiv A_1(X) + \cdots + A_n(X)$$

The meaning of c_i and case statements can easily be given by using the isomorphisms $T \cong F(T)$.

Example 5.1: The type of natural numbers can be declared by

$$\texttt{datatype}\ N = \texttt{zero} \mid \texttt{succ of}\ N.$$

Zero is a constant of N and its domain is regarded as 1. Therefore, this defines the domain which is the initial fixed point of

$$F(X) = 1 + X. \qquad \square$$

Since codatatypes are dual to datatypes, we might expect that the codatatype declared by

$$\texttt{codatatype}\ S = d_1\ \texttt{is}\ B_1(S)\ \&\ \cdots\ \&\ d_n\ \texttt{is}\ B_n(S)$$

can be regarded as the final fixed point of the following functor.

$$G(X) \equiv B_1(X) \otimes \cdots \otimes B_n(X) \qquad (\dagger)$$

Here, we use smash products instead of categorical ones because categorical ones contain undesirable elements. For the codatatype

$$\texttt{codatatype}\ S = \texttt{fst is}\ A\ \&\ \texttt{snd is}\ B,$$

we would like it to denote $A \otimes B$ rather than $A \times B$.

However, ($\dagger$) does not work for recursive definitions. For example, the codatatype of infinite lists is defined by

$$\texttt{codatatype}\ I = \texttt{head is}\ A\ \&\ \texttt{tail is}\ I,$$

and $G(X)$ is $A \otimes X$. The final fixed point of G can be given as the limit of the following diagram.

$$1 \leftarrow G(1) \leftarrow G^2(1) \leftarrow \cdots \leftarrow G^n(1) \leftarrow \cdots$$

Because $A \otimes 1 \cong 1$, the final fixed point is $1 \equiv \{\ \perp\ \}$. This is not what we want. In fact, initial fixed points and final fixed points always coincide in $\mathbf{CPO}_\perp$. Then, it may seem that there is no difference between datatypes and codatatypes. We need some tricks to make difference. We use the lifting functor.

Definition 5.2: For the codatatype declared by

$$\texttt{codatatype}\ S = d_1\ \texttt{is}\ B_1(S)\ \&\ \cdots\ \&\ d_n\ \texttt{is}\ B_n(S),$$

it is characterized as the initial fixed point (= the final fixed point) of the following functor.

$$G(X) \equiv B_1(X_\perp) \otimes \cdots \otimes B_n(X_\perp) \qquad \square$$

We can give the semantics of destructors and merge statements as follows.

Definition 5.3: Let us write ψ for the isomorphism $S \to G(S)$ and ϕ for its inverse.

$$\begin{cases} [\![d_i]\!](x) \equiv \pi_i(\psi(x)) \\ [\![\texttt{merge}\ d_1 \Leftarrow e_1\ \&\ \cdots\ \&\ d_n \Leftarrow e_n]\!] \equiv \phi(\langle([\![e_1]\!])_\perp, \ldots, ([\![e_n]\!])_\perp\rangle) \end{cases}$$

where π_i is the ith project of $B_1(S_\perp)\otimes\cdots\otimes B_n(S_\perp)$ and $([\![e_i]\!])_\perp$ is the result of injecting $[\![e_i]\!]$ into $B_i(S_\perp)$. []

Example 5.4: The product of A and B is defined by

$$\texttt{codatatype}\ P = \texttt{fst is}\ A\ \&\ \texttt{snd is}\ B.$$

Semantically, it denotes the initial fixed point of

$$G(X) = B_1(X_\perp) \otimes B_2(X_\perp) = A \otimes B.$$

Since $G(X)$ is a constant functor, the initial fixed point is $A \otimes B$, which we expected. Fst and snd are projections and $[\![\texttt{merge fst} \Leftarrow e_1\ \&\ \texttt{snd} \Leftarrow e_2]\!]$ is just $\langle[\![e_1]\!], [\![e_2]\!]\rangle$. []

Example 5.5: Let us consider the codatatype of infinite lists defined by

$$\texttt{codatatype}\ I = \texttt{head is}\ A\ \&\ \texttt{tail is}\ I.$$

Semantically, I is the initial fixed point of

$$G(X) \equiv A \otimes X_\perp.$$

By calculating the limit of

$$0 \to A \otimes 0_\perp \to A \otimes (A \otimes 0_\perp)_\perp \to A \otimes (A \otimes (A \otimes 0_\perp)_\perp)_\perp \to \cdots,$$

we obtain

$$I = \{\ (a_1\ a_2\ \ldots a_n) \mid 0 \le n \le \infty, a_i \in A, a_i \ne \perp\ \}$$

with its ordering given by

$$(a_1\ a_2\ a_3\ \ldots a_n) \sqsubseteq (b_1\ b_2\ b_3\ \ldots b_m) \iff n \le m \land \forall i = 1, \ldots, n(a_i \sqsubseteq b_i).$$

Note that n may be ∞ and, therefore, I includes infinite lists of A elements. Merge statements concatenate an A element to an infinite list. Therefore,

```
val rec iseq1 = merge head <= 1 & tail <= iseq1;
```

denotes the lest upper bound of the following sequence.

$$() \sqsubseteq (1) \sqsubseteq (1\ \ 1) \sqsubseteq (1\ \ 1\ \ 1) \sqsubseteq \cdots \sqsubseteq (1\ \ 1 \ldots 1) \sqsubseteq \cdots$$

The least upper bound is the infinite list of 1. []

6 Codatatypes as Greatest Fixed Points

In $\mathbf{CPO}_{\perp}$, final fixed points and initial fixed points are the same, and we have to use lifting to get the proper meaning of codatatypes. If we do not use function spaces, we can give the meaning in the category of sets, **Set**. In [2], Arbib and Manes emphasized the usefulness of greatest fixed points (= final fixed points). In **Set**, the initial object is the empty set, the final object is the one-point set, products are given by cartesian products, and sums are given by disjoint unions.

The meaning of the datatype defined by

$$\texttt{datatype}\ T = c_1\ \texttt{of}\ A_1(T) \mid \cdots \mid c_n\ \texttt{of}\ A_n(T)$$

is given by the initial fixed point of

$$F(X) \equiv A_1(X) + \cdots + A_n(X),$$

in the same way as in $\mathbf{CPO}_{\perp}$, whereas the meaning of the codatatype defined by

$$\texttt{codatatype}\ S = d_1\ \texttt{is}\ B_1(S)\ \&\ \cdots\ \&\ d_n\ \texttt{is}\ B_n(S).$$

is given simply by the final fixed point of

$$G(X) = B_1(X) \times \cdots \times B_n(X).$$

In the approach, the duality of datatypes and codatatypes is more apparent.

Example 6.1: Let us once more consider the codatatype of infinite lists given in example 5.5. $G(X)$ is $A \times X$. Its initial fixed point is the empty set, but the final fixed point is given as the limit of the following diagram.

$$1 \leftarrow A \times 1 \leftarrow A \times (A \times 1) \leftarrow \cdots$$

It is easy to see that the limit is the following set.

$$\{\ (a_1\ a_2\ \ldots\ a_n\ \ldots) \mid a_i \in A\ \}$$ ▯

7 Conclusions

In this paper, we proposed a new data type declaration mechanism of codatatypes for ML. The product type no longer needs to be primitive, and lazy data types, like infinite lists, can be declared by this mechanism. We obtained symmetry in ML. Let us call this new ML SymML.

We can regard codatatypes as mirror images of datatypes. In category theory, datatypes are characterized as left adjoints, whereas codatatypes are characterized as right adjoints. Traditionally, datatypes have been open to users for defining their own data types, but codatatypes have been fixed and given as primitive data types (e.g. products). Users have not been allowed to define them. However, in SymML, both datatypes and codatatypes are treated equally and are open to users to define new data types.

Codatatypes have totally opposite properties of datatypes. A datatype is declared by listing its constructors, but a codatatype is declared by listing its destructors. The control structure for datatypes is `case` statements which decompose elements of datatypes into their components, but the control structure of codatatypes is `merge` statements which combine components and create elements of codatatypes. In **Set**, datatypes are characterized as initial fixed points, but codatatypes are characterized as finial fixed points. Algebraically, datatypes are ordinary algebras, but codatatypes are co-algebras.

Because everything (except the function space type) can be defined either as a datatype or a codatatype, the semantics of SymML can be given in a uniform manner. We do not need to treat the product type specially.

In SymML, infinite lists are defined as codatatypes. Usually, infinite lists are obtained by changing evaluation mechanism from full evaluation to lazy evaluation. However, in SymML, lazy evaluation is embedded into codatatypes. The data type of finite lists and the data type of infinite lists are two distinct data types. We have an advantage of having both data types in the same framework. Users can choose which data type to use according to their need.

The author believes that lazy evaluation should be treated in the framework of codatatypes. Lazy evaluation is often discussed for lists. The reason why lists can be treated lazily is that lists are made of pairs. Pairs are declared as codatatypes in SymML. Therefore, the laziness of lists comes out of that of codatatypes. In addition, when lazy evaluation is simulated in a programming language which employs full evaluation, function closures are often used. A function space is a kind of codatatype. In SymML, it cannot be declared as a codatatype, but, in CPL from which SymML is derived, it can be declared as a codatatype. Therefore, the laziness comes out again from codatatypes.

Although we have not shown here, one can define the lazy data type of natural numbers as a codatatype. The data type of ordinary natural numbers can be defined by listing 0 and the successor function as a datatype, but the lazy one is defined by listing one destructor, the predecessor function.

```
codatatype CoNat = pred is unit+CoNat;
```

`Pred` decreases the given number by one or fails. In the latter case, it returns the element of `unit`. It turns out that `CoNat` has one extra element, the infinity. Furthermore, in CPL, the data type of ordinary natural numbers is associated with primitive recursion, whereas the lazy one is associated with general recursion. We cannot define μ operator for the ordinary one, but we can define it for the lazy one.

SymML is derived from a categorical programming language CPL. Refer [12] for CPL. For a lambda calculus version of SymML, refer [13].

References

[1] Arbib, M. A. and Manes, E. G.: *Arrows, Structures, and Functors — The Categorical Imperative —*. Academic Press, 1975.

[2] Arbib, M. A. and Manes, E. G.: The Greatest Fixed Points Approach to Data Types. In *proceedings of Third Workshop Meeting on Categorical and Algebraic Methods in Computer Science and System Theory*, Dortmund, West Germany, 1980.

[3] Burstall, R. M. and Goguen, J. A.: The Semantics of Clear: A Specification Language. Internal Report CSR-65-80, Department of Computer Science, University of Edinburgh, 1980.

[4] Burstall, R. M. and Goguen, J.A.: An Informal Introduction to Specifications using Clear. In *The Correctness Problem in Computer Sciences*, Academic Press, pp. 185–213, 1981.

[5] Burstall, R. M. and Goguen, J.A.: *Algebras, Theories and Freeness: an Introduction for Computer Scientists.* Internal Report CSR-65-80, Department of Computer Science, University of Edinburgh, 1981.

[6] Burstall, R. M., MacQueen, D. and Sannella, D.: *HOPE: An Experimental Applicative Language.* Internal Report CSR-62-80, Department of Computer Science, University of Edinburgh, 1980.

[7] Cardelli, L.: ML under UNIX. Bell Laboratories, Murray Hill, New Jersey, 1982.

[8] Curien, P-L.: *Categorical Combinators, Sequential Algorithms and Functional Programming.* Research Notes in Theoretical Computer Science, Pitman, 1986.

[9] Gordon, M. J., Milner, A. J. and Wordsworth, C. P.: *Edinburgh LCF. Lecture Notes in Computer Science*, Volume 78, 1979.

[10] Goguen, J. A., Thatcher, J. W. and Wagner, E. G.: An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. In *Current Trends in Programming Methodology*, prentice-Hall, pp. 80–149, 1978.

[11] Goldblatt, R.: *Topoi: The Categorical Analysis of Logic.* Studies in Logic and Foundation of Mathematics, Volume 98, North-Holand, 1979.

[12] Hagino, T.: *Category Theoretic Approach to Data Types.* Ph. D. thesis, University of Edinburgh, 1987.

[13] Hagino, T.: A Typed Lambda Calculus with Categorical Constructions. *Category Theory and Computer Science*, Lecture Notes in Computer Science, Vol. 283, Springer-Verlag, pp. 140–157, 1987.

[14] Harper, R., MacQueen, D. and Milner, R.: *Standard ML.* LFCS Report Series, ECS-LFSC-86-2. Department of Computer Science, University of Edinburgh, 1986.

[15] Lambek, J.: From Lambda-calculus to Cartesian Closed Categories. In *To H. B. Curry; Essays on Combinatory Logic, Lambda-calculus and Formalism*, edited by J. P. Seldin and J. R. Hindley, Academic Press, 1980.

[16] Lambek, J. and Scott, P. J.: *Introduction on Higher-Order Categorical Logic. Cambridge Studies in Advanced Mathematics*, Volume 7, 1986.

[17] Lehmann, D. and Smyth, M.: Algebraic Specification of Data Types – A Synthetic Approach –. *Mathematical System Theory*, Volume 14, pp. 97–139, 1981.

[18] Mason, I.: *The Semantics of Destructive Lisp.* CSLI Lecture Notes No. 5, 1986.

[19] Mauny, M.: *Compilation des Languages Functionnels dnas les Combinateurs Categoriques, Application au language ML.* These de 3eme cycle, UniversiteParis 7, 1985.

[20] Mauny, M. and Saurez, A.: Implementing Functional Languages in the Categorical Abstract Machine. A. C. M. Conference on Lisp and Functional Programming, Cambridge, pp. 266–278, 1986.

[21] Mac Lane, S.: *Categories for the Working Mathematician. Graduate Texts in Mathematics 5*, Springer-Verlag, 1971.

[22] Paulson, L. C.: *Logic and Computation — Interactive Proof with Cambridge LCF* —. Cambridge Tracts in Theoretical Computer Science, Combridge University Press, 1987.

[23] Pitt, D., Abramsky, S., Poigné, A. and Rydeheard, D. (edited): *Category Theory and Computer Programming*, Lecture Notes in Computer Science, Volume 240, Springer-Verlag, 1986.

[24] Scott, D.: Data Types as Lattices. *SIAM Journal of Computing*, Volume 5, pp. 552–587, 1976.

[25] Smyth, M. B. and Plotkin, G. D.: The Category-Theoretic Solution of Recursive Domain Equations. *SIAM Journal of Computing*, Volume 11, pp. 761–783, 1982.