Theory and practice of sequential algorithms : the Kernel of the applicative Language CDS

Gérard Berry, P.L. Curien

INRIA

CENTRE
SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(3) 954 90 20

Rapports de Recherche

N° 225

THEORY AND PRACTICE OF SEQUENTIAL ALGORITHMS: THE KERNEL OF THE APPLICATIVE LANGUAGE CDS

Gérard BERRY
Pierre-Louis CURIEN

Juillet 1983

THEORY AND PRACTICE OF SEQUENTIAL ALGORITHMS: THE KERNEL OF THE APPLICATIVE LANGUAGE CDS

G. Berry, Ecole des Mines, 06560 Valbonne, France
P-L. Curien, LITP, Université Paris VII, Paris, France

RESUME

Nous présentons les principales caractéristiques du langage applicatif CDS0, qui est fondé sur la théorie des algorithmes séquentiels sur structures de données concrètes. Nous montrons comment déclarer les structures de données en CDS0 à l'aide de quatre primitives simples, et comment les manipuler par des programmes écrits sous forme de définitions directes d'algorithmes composées à l'aide de six combinateurs. Nous étudions en détail la principale originalité de CDS0: les structures de données d'ordre supérieur sont traitées exactement comme les structures de données de base. Nous décrivons les sémantiques opérationnelles et dénotationnelles de CDS0, et montrons leur identité. Enfin nous montrons comment utiliser les primitives de CDS0 pour implémenter des primitives plus classiques et de plus haut niveau.

ABSTRACT

We present the main characteristics of the applicative language CDS0, which is based on the theory of sequential algorithms on concrete data structures. We show how to declare concrete data structures with only four simple primitives, and how to write programs using basic sequential algorithms and six combinators. We investigate in detail the structure of higher order spaces, which are treated exactly as ground ones (this is the main originality of CDS0). We study the denotational and operational semantics of CDS0, and show their identity (the full abstraction theorem). We show how to use CDS0 as a kernel language for implementing high level primitives.

Recherche financée par le GRECO PROGRAMMATION du CNRS. Cet article doit paraître dans les actes du Séminaire Franco Américain de Sémantique, Fontainebleau, 1982, et doit être cité comme tel.

Research financed by the GRECO PROGRAMMATION of the CNRS. This paper will appear in the Acts of the French-American Seminar on Semantics, Fontainebleau, 1982, and should be cited as such.

THEORY AND PRACTICE OF SEQUENTIAL ALGORITHMS: THE KERNEL OF THE APPLICATIVE LANGUAGE CDS

G. Berry, Ecole des Mines, 06560 Valbonne, France
P-L. Curien, LITP, Université Paris VII, Paris, France

1. Introduction.

We have presented in [1] a preliminary version of a new applicative language called CDS, which is directly issued from the authors theory of sequential algorithms on Kahn-Plotkin's concrete data structures [3,6,10]. The implementation of the language lead us to take an intermediate step and to define a rather low-level kernel language CDS0, which can be understood as defining an "abstract machine" or "assembly language" suited for any further development of CDS. Future versions of CDS should have a ML-like syntax [8,5] and have their expressions translated into CDS0 at parse-time for execution.

In the present paper we put the emphasis on three originalities of CDS0: its power w.r.t. data structure definitions, the fact that there is no distinction between ground and higher-order data structures, and its full abstraction property. That last property expresses the identity of the operational and denotational semantics, which are both presented in some detail. We indicate briefly how higher-level constructs can be translated into CDS0.

The CDS0 (or CDS) data structures are simply the original Kahn-Plotkin (distributive) concrete data structures, or dcds for short. They form an unusually large world, and may be constructed from unusually few primitives: enumeration of values, graft of data structures onto other

(*) The theory of sequential algorithms was actually developed for trying to solve the full abstraction problem for classical sequential languages[16,13], which is studied in detail elsewhere in this volume [4] and is still unsolved in its original form.

ones, recursion, exponentiation, together with a simple calculus on cell and value names. Constructs which are classically primitive are easily derived here: records, sums, lists, arrays etc. Recursion (the key to domain equations) has a particularly simple theory: the dcds themselves form a complete partial order, and the data structure constructors are just ordinary continuous functions. Hence domain equations are just common fixpoint equations; moreover the solution to any domain equation is an exact domain equality, not just an isomorphism. One may also define and manipulate infinite data structures (streams, infinite arrays...). But the main originality of CDS0 concerns the exponentiation (M->M') of two data structures: that space is itself a standard dcds, not a space of "functions" or "closures". Hence it may be handled in the same way as any other dcds.

The CDS0 basic computing agents are called sequential algorithms, and may be viewed as simple output-driven programs written with only two basic instructions. The set of algorithms between two dcds M and M' may itself be organized into a dcds, which is the exponentiation dcds (M->M') just mentioned. In that sense algorithms need not to be primitive: they are just ordinary values - states in our terminology - of ordinary dcds. According to the results of [3], an algorithm may also be viewed as a pair of two functions: a classical input-output function, and a control function which gives information about the way the algorithm evaluates its arguments. Two algorithms may perfectly well define the same input-output function and be distinct in CDS0 because their differ in their control part. For example a left-sequential and a right-sequential (strict) boolean conjunction are distinct objects in CDS0, and they may actually be distinguished at the terminal (this does in no case prevent CDS0 to have the "referential transparency" property necessary to any decent language).

The CDS0 expressions are build from states (or algorithms) by using six simple combinators: application, composition, fixpoint, pairing, currying, uncurrying. These combinators are directly suggested by the theory of cartesian closed categories, see [12,2,18]), and they permit easy implementation of higher-level constructs such as lambda-binders.

The denotational semantics of CDS0 is directly given by the theory of [3]. The semantics of any term is simply a state of the dcds corresponding to its type, hence a standard CDS0 object. This has far-reaching consequences: a higher-order type expression may be evaluated on its own, without giving it arguments; moreover CDS0 programs may take the semantics of other CDS0 programs as argument. In some sense CDS0 is a tool for manipulating the semantics of CDS0 expressions exactly as LISP is a tool for manipulating the syntax of LISP programs. We give several examples of such manipulations.

CDS0 admits two quite different operational semantics called CDS01 and CDS02, which are both implemented. We study only CDS01 here. The semantics is given by a set of conditional rewrite rules in the style of [17]. The rules suggest two possible evaluation strategies: lazy evaluation, which is natural on sequential machines (and is the actually implemented strategy), and eager evaluation where algorithms may be viewed as coroutines exchanging pieces of information of arbitrarily complex data structures, and not just streams of tokens as in [9]. This style of implementation would be suited to parallel machines, but has not yet been investigated in detail.

The full abstraction property expresses the identity of the denotational and operational semantics (CDS01 or CDS02), this for any expression of any type.

Section 1 treats the dcds definition facilities besides exponentiation. Section 2 is devoted to algorithms: we introduce many examples of these objects and introduce their denotational semantics together with the dcds exponentiation construct. We then study the CDS01 combinators, give examples of program semantics manipulation and finish with the translation of the lambda-calculus into CDS0. The operational semantics and the full abstraction theorem are presented in section 4.

The implementation of CDS01 and CDS02 was realized together with M. Devin, F. Montagnac and A. Ressouche. A prototype system is available from the authors (it runs under Multics-maclisp and VAX-franzlisp).

2. Concrete data structures.

We first introduce the concrete syntax used to declare finite concrete data structures in CDSO. Then we define the notion of state (amount of information) in a concrete data structure. We show how a state may be declared as a constant, and have a first look to the interpreter. Then we formalize concrete data structures as mathematical objects, giving their abstract order-theoretical properties (the work of Kahn-Plotkin). Finally we indicate how to declare infinite structures in CDSO, using macrogeneration techniques, how to build new concrete data structures from old ones, using a very simple grafting constructor, and how to define concrete data structures recursively. We show that the mathematical framework of concrete data structures gives a semantics for the CDSO declarations.

2.1. Cells,values,events,enablings,states.

Declaring a deterministic concrete data structure (dcds for short) in CDSO looks like specifying the rules of a very simple game, the goal of which is to fill some specified places or cells with some specified values, with at most one value per cell. Here are some elementary examples of dcds declarations:

```
let OO =
  dcds
    cell O values T
  end
```

The unique cell O may receive the unique value T. Putting the value T in the cell O produces the event O=T, and the game has two possible states: O empty, written {}, and O filled, written {O=T}.

```
let BOOL_PAIR =
  dcds
    cell B.1 values T,F
    cell B.2 values T,F
  end
```

The two cells B.1 and B.2 may receive either T or F. This is very much like declaring a two fields record of booleans. Here {}, {B.1=T} and {B.1=T,B.2=F} are three possible states of the game.

```
let TT_BOOL_PAIR =
  dcds
    cell B.1 values T
    cell B.2 values T,F
  end
```

In this more selective dcds B.2 may still receive either T or F, but B.1 may receive only T, thus restricting the range of values of the first field of our "record". Notice that the states of TT_BOOL_PAIR are states of BOOL_PAIR. Later on we shall see that TT_BOOL_PAIR is included in BOOL_PAIR.

The game has one more rule: one may specify access conditions on cells. Here is a declaration including such conditions:

```
let VARIANT =
  dcds
    cell CHOICE values L,R
    cell B.1 values T access CHOICE=L
    cell B.2 values T,F access CHOICE=R
  end
```

The cell CHOICE may receive either L or R, and may be filled by the player at the beginning of the game since it has no access condition. Suppose the player has filled CHOICE with L, provoking the event CHOICE=L. From that point, the player can only fill B.1, since the access condition is valid for B.1 and not for B.2. If symmetrically the event CHOICE=R had occurred (i.e if the player had filled CHOICE with R), that would have enabled B.2 and disabled B.1 (*). Hence VARIANT is a "record with variant".

Once enabled, a cell may hold any of the values listed after the keyword "values" in its declaration. Hence either B.2=T or B.2=F may occur after CHOICE=R.

An access condition to a cell, also called an enabling, may consist of several events. Then all these events must have occurred in order to enable the cell. Furthermore a cell may have several access conditions,

(*) The reader familiar with Petri nets will feel that there are some connections between concrete data structures and Petri nets. Those were thoroughly studied in [15,21] in the more general framework of event structures.

separated by the "or" keyword. It is then enabled if at least one condition is valid (i.e if all the events listed in one condition have occurred). It is essential to impose also that at most one access condition is valid at a time, see [3]. This condition is called the determinism or stability condition, and justifies the letter "d" in the keyword "dcds". Here is an example of a dcds with multiple enablings:

```
let MULT_ENAB =
  dcds
    cell C0 values 0,1
    cell C1 values 0
    cell C2 values 0 access C0=0,C1=0 or C0=1
  end
```

All its possible states are:

```
{}
{C0=0}
{C0=1}
{C1=0}
{C0=0,C1=0}
{C0=1,C1=0}
{C0=0,C1=0,C2=0}
{C0=1,C2=0}
{C0=1,C1=0,C2=0}
```

Notice that {C0=0,C2=0} is not a state, since C2 is not enabled, and that {C0=0,C0=1} is neither a state, since C0 can only hold one value in a state. Notice also that the two enablings of C2 cannot occur together, since it is impossible to have C0=0 and C0=1 in the same state.

To summarize, once a dcds is declared, the player is allowed to start filling its cells, provided that each cell receives at most one value and that the enablings are always satisfied. The player may leave the play at any time, and any configuration of values in cells (events) that it may reach is called a state.

2.2. Evaluating in CDSO.

We have already enough to introduce the notion of interpretation session. The interpreter receives expressions of the language, which all denote states. After some static verifications, it enters a loop of requests: it prompts the user for the name of a cell, checks that the

cell name is correct, and checks if the cell is filled in the expression's denotation. If yes, it returns its value, and prompts the user again. If it finds out that the cell is not filled, it returns no value and prompts the user for another cell. It may also be involved in an infinite computation and never answer.

The simplest CDSO expressions are the state constants that we just defined, so that we may show a sample session using one of the states listed at the end of 2.1:

```
# {C0=1,C2=0};
request? C0;
--> 1
request? C1;
-->
request? C2;
--> 0
request? ;
```

The strings "#" and "request?" are respectively the prompts at the top level and request level of CDSO, and the arrow "-->" precedes the evaluator response. There is no response for C1 since it is empty. The request loop is terminated by a single ";", returning to the expression evaluator.

Of course we should have typed our state constant, in order to allow static verifications (correctness of the state and of the cell names). In the present paper we shall freely forget about all static aspects of this kind.

We shall also use abbreviations (macros) defined by a simple "let" construct, as in

```
let TRUE_FALSE = {B.1=T,B.2=F};
```

Then TRUE_FALSE is macroexpanded at parse-time whenever it appears in an expression, and is therefore not a dynamic variable.

Before defining more complex cds and expressions in CDSO, we show that the objects defined so far may be formalized as mathematical objects, and possess nice order-theoretical properties.

2.3. Concrete data structures as mathematical objects.

2.3.1. Definition: A concrete data structure or cds $M=(C,V,E,\vdash)$ is given by

- a denumerable set C of cells;
- a denumerable set V of values;
- a set E of events included in CxV. An event e is written indifferently (c,v) or c=v;
- an enabling relation $\vdash$ between finite subsets of E and cells. An element of $\vdash$ is called a rule and is written $e_1,e_2,..,e_n \vdash c$. The set $\{e_1,e_2,..,e_n\}$ is called an enabling of c. A cell c such that $\vdash c$ is called an initial cell.

A state x of M is a set of events such that

(i) $(c,v_1),(c,v_2) \in x \Rightarrow v_1=v_2$

(ii) $(c,v) \in x \Rightarrow e_1,e_2,..,e_n \in x$ for some rule $e_1,e_2,..,e_n \vdash c$.

The set of states of M is written D(M), and is ordered by set inclusion, written $\leq$. Notice that two states x_1,x_2 have an upperbound in D(M) iff $(c,v_1) \in x_1$, $(c,v_2) \in x_2$ always imply $v_1=v_2$. The set union of x_1 and x_2 is then their least upper bound (lub).

A cell c is filled in x if $(c,v) \in x$ for some v. A cell c is enabled by x if x contains an enabling of c, and is accessible from x if it is not filled in x and enabled in x. We denote by F(x) and A(x) the sets of cells filled in x and accessible from x. We write $x <_c y$ if $x<y$ and if $c \in A(x)$, $c \in F(y)$. We say that y covers x and write $x -< y$ iff $x<y$ and $x<z\leq y$ implies $z=y$. Hence x and y differ by exactly one event. We write $x -<_c y$ iff $x <_c y$ and $x -< y$.

We always assume the following well-foundedness axiom (which ensures the equivalence of our definition of state and of that of Kahn-Plotkin [3,10]):

(WF) set $c \leq_E c_1$ if (c,v) belongs to an enabling of c_1 for some v. The reflexive and transitive closure of $\leq_E$ must be well founded.

We impose also the determinism or stability axiom:

(D) For any state x, any cell c filled in x has exactly one enabling in x.

This axiom is essential in our construction. Without it, functional expressions in CDSO would not denote states. A cds satisfying (D) is called a stable or deterministic concrete data structure, or dcds for short. In a dcds the set intersection of an upperbounded set of states is a state.

The Kahn-Plotkin representation theory [10] relates the concrete data structures to a subclass of complete partial orders. First it is easily seen that the set D(M) of states of a cds M, ordered by inclusion, is an ω-algebraic coherent cpo, with the empty state as least element, and the finite states as isolated elements. Moreover it satisfies the following finiteness axiom:

(F) Any isolated element dominates finitely elements.

Reciprocally, Kahn and Plotkin [10] have shown that a complete partial order satisfying those properties and three more axioms (which concern the covering relation $-<$) is isomorphic to the set of states of a cds. The domains satisfying the Kahn-Plotkin axioms are called concrete domains. Intuitively, the events are reconstructed from the covering relation: if $x-<y$, then y has only one event more than x. One of the three axioms guarantees that covering chains between two points have the same length, which is classical in lattice theory. The two other axioms recover cells, and involve suitable equivalence classes of pairs (x,y), where $x-<y$. Adding a distributivity axiom yields the correspondence with dcds.

Winskel [21] has simplified and generalized the Kahn-Plotin construction by weakening one of the axioms of concrete domains, yielding the correspondence with event structures, which are concrete structures more general than the cds.

We shall not need these powerful representation theorems here. However they provide a solid foundation to the language CDSO.

The structures declared in CDSO are easily related with the mathematical objects of this section, which provide a mathematical semantics for the syntax of 2.1. The translation of a dcds declaration as in 2.1 into an object (C,V,E,|-) may be described formally by straightforward denotational semantic equations, as will be shown in 2.5.

2.4. The language of declarations in CDSO.

We have seen basic dcds declarations so far, allowing only to declare finite dcds in extension, i.e. by listing cells, values and enablings. Now we show more complex dcds declarations.

First we briefly indicate how some infinite structures may be declared, using macrogeneration of names. Further details may be found in [1]. The following declarations successively define the ten first natural numbers, all the integers, and the data structure with states $\{N.1=m_1\}$ for all natural m_1 and $\{N.1=m_1, N.2=m_2\}$ for all m_1 such that $0 \leq m_1 \leq 10$ and all integer m_2:

```
let FIRST_TEN =
  dcds
    cell N values [0..10]
  end

let INTEGER =
  dcds
    cell N values [..]
  end

let FOO =
  dcds
    cell N.1 values [0..]
    cell N.2 values [..] access N.1=[0..10]
  end
```

Values may themselves be composite:

```
let FRACTIONS =
  dcds
    cell R values [..].[1..]
  end
```

Here a value is a pair m.n, $n \geq 1$ and the numerator and denominator are defined or undefined together, unlike in a classical record.

We next define a very useful grafting constructor, which is used for building new dcds from old ones. Let us take an example. The dcds FOO above may clearly be decomposed into the dcds of natural numbers and into a copy or "graft" of the dcds INTEGER, labeled with ".2", and accessible from any state {N.1=x} with $0 \leq x \leq 10$. One may alternatively define FOO in the following way:

```
let FOO =
  dcds
    cell N.1 values [0..]
    graft INTEGER.2 access N.1=[0..10]
  end
```

This of course becomes more useful if the grafted dcds is more intricate. What would for instance happen if one replaced INTEGER by VARIANT in the above declaration? The obtained dcds would be by definition the following one:

```
let FOO_BAR =
  dcds
    cell N.1 values [0..]
    cell CHOICE.2 values L,R access N.1=[0..10]
    cell B.1.2 values T access N.1=[0..10], CHOICE.2=L
    cell B.2.2 values T,F access N.1=[0..10], CHOICE.2=R
  end
```

The cell declarations of the grafted dcds are taken one by one, and the following operations are (conceptually) performed for each of them: the tag ".2" is appended to the cell names as indicated in the graft declaration, the original value list is kept, the access conditions of the graft declaration are added to each access condition of the new cell (where any cell of the grafted data structure receives also the ".2" tag).

We are not bound to having finitely many cells: here is a way of defining powerseries over fractions, where the nth coefficient is held

in the cell (R.n) - see 3.8 for examples of powerseries manipulations:

```
let POWERSERIES =
  dcds
    graft FRACTIONS.[0..]
  end
```

The graft construct allows to derive the product and sum constructors on dcds. Suppose M1,M2 are two dcds identifiers:

```
let PRODUCT =
  dcds
    graft M1.1
    graft M2.2
  end

let SUM =
  dcds
    cell CHOICE values L,R
    graft M1.L access CHOICE=L
    graft M2.R access CHOICE=R
  end
```

It is easily seen that the set of states of PRODUCT is isomorphic to the cartesian product of the sets of states of M1 and M2. The point is to decorate the names of cells of M1 and M2 in order to ensure that they are distinct, and to put the structures aside. Similarly, it is easily seen that SUM generates indeed the usual separated sum of M1 and M2 in the theory of cpo's: the cell CHOICE acts as a switch, allowing to access either a (tagged) copy of M1 or a (tagged) copy of M2.

The declarations above may of course be parametrized. In this case we call the formal dcds parameters *,**,*1,... as in ML [8]: we define for example the product operator by

```
let *1#*2 =
  dcds
    graft *1.1
    graft *2.2
  end
```

We shall freely use parametrized products and sums in the sequel.

Grafting may finally be recursive. Here is a parametrized recursive definition of lists (as in ML, . denotes the empty dcds which could be simply declared by .=dcds end):

```
letrec LIST(*) =.+(*#LIST(*))
```

Replacing ., # and + by their definition, one could equivalently write

```
letrec LIST(*) =
  dcds
    cell CHOICE values L,R
    graft *.1.R access CHOICE=R
    graft LIST(*).2.R access CHOICE=R
  end
```

We may define streams of integers in the following way:

```
letrec STREAMS =
  dcds
    cell N.1 values [..]
    graft STREAMS.2 access N.1=[..]
  end
```

The access condition is crucial in that definition of streams. Notice that STREAMS has infinitely many cells holding infinitely many values. Such an object is not definable in ML without using functional types.

Generally speaking, complex data structures such as arrays etc. are obtained by implementing explicit descriptors, as we did for SUM. See [1] for details. (Notice that in most languages descriptors exist but are more or less hidden from the user. This leads to awkward syntactic constructs such as the T'FIRST, T'NEXT and T'LAST constructs which give access to array descriptors in ADA.) However the use of explicit descriptors has its drawbacks. For example, when defining a list, the user has to know about its denotation, which he would probably prefer to be an "internal form" and not to have to cope with. If he means, for example the list consisting of integers 1,5,3, he has to type the following horrendous state of LIST(INTEGERS):

```
{CHOICE=R, N.1.R=1, CHOICE.2.R=R, N.1.R.2.R=5, CHOICE.2.R.2.R=R,
 N.1.R.2.R.2.R=3, CHOICE.2.R.2.R.2.R=L}
```

Of course the right solution is to hide the actual implementation of lists and to use an abstract data type definition for LIST, with the classical NIL, CONS, NULL, HD, and TL constructors and destructors. This will be possible in future versions of CDS based on a ML-like syntax for declaring abstract types.

2.5. The semantics of dcds declarations.

The classical techniques for giving the semantics of recursive domain definitions involve fixpoint theorems in categories or in universal domains, see [19]. The idea is the same as for fixpoints in cpo's, and the additional difficulties come from the fact that a domain may usually approximate another domain in several ways, the classical notion of approximation being that of embedding.

One advantage of the theory of concrete data structures is that it leads to an extremely simple notion of approximation between domains, based on the fact that the complete definition of a dcds contains a very precise information about the way things are named, which disappears in usual theories of domains where objects are defined up to isomorphism. The notion of approximation we use is simply that of inclusion:

2.5.1. Definition: We call DCDS the set of concrete data structures we can build by resticting cell and value names to be integers, identifiers, or lists of names separated by ".", as in our present syntax. Given M, M' in DCDS, we say that M is included in M' and write McM' if the following set inclusion holds: CcC', VcV', EcE', |-c|-'. Then D(M) is included in D(M').

2.5.2. Proposition: The set DCDS ordered by inclusion is a coherent ω-algebraic complete partial order. The lubs are just obtained by unions of the sets of cells, values, events and rules. The isolated points are the dcds with finite sets of cells and values.

The same result holds replacing dcds by cds. It is of course not bound to our particular choice of a set of names. Inclusions are indeed related to the classical notion of embedding, see [3,19].

It remains to be seen that dcds declarations define continuous functions on the complete partial order DCDS. The syntax used so far for dcds declarations may be formalized as follows in a BNF form:

```
<DCDS>        ::= dcds {<COMPONENT>}* end
                | <ID>
<COMPONENT>   ::= cell <NAME> values <VALUE_LIST> <ACCESS>
                | graft <DCDS>.<NAME> <ACCESS>
<VALUE_LIST>  ::= <NAME> , {<NAME>}*
<ACCESS>      ::=                          (no access condition given)
                | access <ENABLING>
                | <ACCESS> or <ENABLING>
<ENABLING>    ::= <EVENT>
                | <ENABLING> , <EVENT>
<EVENT>       ::= <NAME>=<NAME>
```

The <ID>'s represent formal arguments to the definition. Hence the semantics of a dcds declaration is a function from DCDS environments (mapping <ID>'s to DCDS) to DCDS. It may be defined rigorously by syntax-directed semantic equations; this is however rather tedious, and we just indicate what is going on. Call M the resulting dcds. Any cell declaration "cell c values $v_1,\ldots,v_n$ access ACCESS" produces trivially a cell of M and some values and rules for that cell. The only delicate situation appears in a "graft M'.n access ACCESS" declaration. Let then $M'=(C',V',E',|-')$ as given by the environment. The graft declaration generates:

- a cell c'.n of M for each cell c' of M;
- a value v' of M for each value v' of M';
- an event c'.n=v' of M for each event c'=v' of M';
- rules of M in the following way: if $c'_1=v'_1,\ldots,c'_n=v'_n|-c'$ is a rule of M' and if $c_1=v_1,\ldots,c_n=v_n$ is an enabling of ACCESS (with $n\geq 0$), then the following rule of M is generated:

$$c'_1.n=v'_1,\ldots,c'_n.n=v'_n,c_1=v_1,\ldots,c_n=v_n|-c$$

Hence the enablings listed in the ACCESS part of the declaration are added to the original enablings of M', where each cell receives the mark n.

It is trivial that any dcds declaration defines a continuous function w.r.t. inclusion. Solving recursive domain equations is therefore no

more than solving usual fixpoint equations in DCDS. Moreover parametrized domain equations are just usual "recursive programs" in the space DCDS. Notice also that the solutions are always exact, i.e. are equalities and not just isomorphisms.

For example let us compute the solution to the equation

```
letrec INTLIST = .+INTEGER#INTLIST
```

which expands into (replacing CHOICE by S)

```
letrec INTLIST =
  dcds
    cell S values L,R
    graft INTEGER.1.R access S=R
    graft INTLIST.2.R access S=R
  end
```

The solution is obtained by iteration, starting from $INTLIST_0$=. One obtains:

- $C = \{S.(2.R)^n,\ n\geq 0\} \cup \{N.1.R.(2.R)^n,\ n\geq 0\}$
- $E = \{S.(2.R)^n=L,\ S.(2.R)^n=R \mid n\geq 0\}$
  $\cup \{N.1.R.(2.R)^n=p \mid n\geq 0\}$
- $\vdash S$
- $S=R,\ldots,S.(2.R)^n=R \vdash S.(2.R)^{n+1}$ for $n\geq 0$
- $S=R,\ldots,S.(2.R)^n=R \vdash N.1.R.(2.R)^{n+1}$ for $n\geq 0$

Then INTLIST in indeed equal to .+INTEGER#INTLIST.

The semantics of arbitrarily complex declarations may of course be treated in the same way.

3. Sequential algorithms.

Once we have defined our data structures, we must write programs for manipulating them. According to the theory of [3] our basic programs will be sequential algorithms which output values in cells of an output dcds according to values read in cells of an input dcds. We start by presenting algorithms as program texts. We show how to apply an algorithm to an input and how to compose algorithms. By an easy coding, we transform the algorithms from M to M′ into states of a new dcds (M->M′) called the exponentiation of M and M′. This helps us in formalizing sequential algorithms as mathematical objects. We introduce the remaining CDSO combinators (pairing, currying, uncurrying, fixpoint). We show how to translate lambda-calculus based languages into CDSO. Finally we give two examples which illustrate the unusual expressive power of CDSO.

3.1. Sequential algorithms as programs.

Sequential algorithms may be thought of as working in an output-directed way: they receive requests for filling output cells, and execute instructions for outputting values or inspecting their input. Assume first that the output dcds M′ has only one cell c′. Then an algorithm from M to M′ has the form:

```
algo
  request c′ do
    <instruction>
  end
end
```

where the <instruction> has two possible forms:

(i) the output instruction outputs a value into the cell of M′.It is written

```
output v′
```

(ii) the input and test instruction branches to other instructions according to the value read in a given input cell c. It is written

```
valof c is
  v1: <instruction>
  .
  vn: <instruction>
end
```

where v1 , ..., vn are possible values of c.

Here are three very simple programs, defining respectively the identity, an approximation of the identity and the negation on booleans:

```
let ID_BOOL: BOOL->BOOL =
  algo
    request B do
      valof B is
        T: output T
        F: output F
      end
    end
  end

let APPROX_ID_BOOL: BOOL->BOOL =
  algo
    request B do
      valof B is
        T: output T
      end
    end
  end

let NOT: BOOL->BOOL =
  algo
    request B do
      valof B is
        T: output F
        F: output T
      end
    end
  end
```

And here are four sequential algorithms for computing the boolean conjunction:

```
let LEFT_AND: BOOL#BOOL->BOOL =
  algo
    request B do
      valof B.1 is
        T: valof B.2 is
             T: output T
             F: output F
           end
        F: output F
      end
    end
  end


let RIGHT_AND =
  algo
    request B do
      valof B.2 is
        T: valof B.1 is
             T: output T
             F: output F
           end
        F: output F
      end
    end
  end


let LEFT_STRICT_AND =
  algo
    request B do
      valof B.1 is
        T: valof B.2 is
             T: output T
             F: output F
           end
        F: valof B.2 is
             T: output F
             F: output F
           end
      end
    end
  end
```

```
let RIGHT_STRICT_AND =
  algo
    request B do
      valof B.2 is
        T: valof B.1 is
             T: output T
             F: output F
           end
        F: valof B.1 is
             T: output F
             F: output F
           end
      end
    end
  end
```

These four algorithms differ in the way they test their arguments, or in their behaviour on partially defined arguments. We shall see later on that they are all distinct in CDS, and that we may actually distinguish them at the terminal.

What happens now if the output dcds has more than one cell? If the cells are all initial, an algorithm consists of different request constructs, one for each different output cell; independent parts of an algorithm compute independent parts of the output. If an output cell is not initial, then it may become accessible in different ways, i.e from different output states which have of course been obtained from different input states. Independent computations may start from these input states. Each of these computations is an instruction enclosed in a construct

```
from <input state> do
  <instruction>
end
```

The from constructs are themselves enclosed in a request construct. Here is an example, taking BOOL_PAIR as input and MULT_ENAB as output:

```
let EXAMPLE =
  algo
    request C0 do
      valof B.1 is
        T: output 1
        F: valof B.2 is
             F: output 0
           end
      end
    end
    request C1 do
      valof B.2 is
        T: output 0
        F: output 0
      end
    end
    request  C2
      from {B.1=T} do
        valof B.2 is
          T: output 0
        end
      end
      from {B.1=F,B.2=F} do
        output 0
      end
    end
  end
```

(be aware that input states appear in the from's, not output states).

Macro-generation may also be used for algorithms. For example

```
valof c is
  &v: output &v
end
```

macro-generates the instruction which tests c and outputs the value read in c, the name variable &v ranging over the possible values of c. That instruction being very common, we abbreviate it into

```
transmit c
```

Polymorphic identity or projection algorithms can then be declared as follows:

```
let IDENTITY: *->* =
  algo
    request &c do
      transmit &c
    end
  end

let fst: *1#*2->*1 =
  algo
    request &C do
      transmit &C.1
    end
  end
```

(We skip however a little difficulty here: some "from" parts are missing; one may safely assume that a "request" without "from" parts macro-generates "request c from X do ..." for each minimal state X enabling c).

One can also perform some arithmetic on names:

```
let PLUS: INT#INT->INT =
  algo
    request N do
      valof N.1 is
        &V1: valof N.2 is
               &V2: output &V1+&V2
             end
      end
    end
  end
```

The addition is conceptually done at macro-generation time: the above text is just an abbreviation for an infinite addition table.

There are some syntactic restrictions on algorithms, which will be detailed in a moment. Basically one is not allowed to test an input cell twice in the text of an instruction, and one must test only accessible cells. These conditions will allow us to give the semantics of algorithms by using only trivial textual translations.

3.2. Applying algorithms to input states. Composing algorithms.

The application of an algorithm to an input state is a CDSO expression, the semantics of which is a state, according to the principle stated in 2.2. We use the symbol "." for denoting application

in CDSO. The intuitive semantics is obvious, and we illustrate it by some interpreter sessions:

```
# NOT . {B=T};
request? B;
--> F
request? ;

# NOT . (NOT . {B=T});
request? B;
--> T
request? ;

# LEFT_AND . {B.1=T,B.2=T};
request? B;
--> T
request? ;

# LEFT_AND . {B.1=F};
request? B;
--> F
request? ;

# LEFT_STRICT_AND . {B.1=F};
request? B;
-->
request? ;

# RIGHT_AND . {B.1=F};
request? B;
-->
request? ;
```

We see at once how to distinguish between LEFT_AND, LEFT_STRICT_AND, and RIGHT_AND. But we must wait for the next section for distinguishing between LEFT_STRICT_AND and RIGHT_STRICT_AND, which always give the same results when applied to the same arguments.

Let us investigate in detail what the above algorithm EXAMPLE computes, when given an input state:

```
# EXAMPLE . {B.1=T};
request? C0;
--> 1
request? C1;
-->
request? C2;
-->
request? ;
```

```
# EXAMPLE . {B.1=F};
request? C0;
-->
request? C1;
-->
request? C2;
-->
request? ;

# EXAMPLE . {B.1=T,B.2=T};
request? C0;
--> 1
request? C1;
--> 0
request? C2;
--> 0
request? ;

# EXAMPLE . {B.1=F,B.2=F};
request? C0;
--> 0
request? C1;
--> 0
request? C2;
--> 0
request? ;
```

The states {B.1=T},{B.1=F,B.2=F} respectively produce answer 1 in cell C0, and answers 0 in cell C0 and 0 in cell C1, which both enable C2.

The composition of algorithms is also intuitively simple, and we use the symbol "|" for it:

```
# (NOT | RIGHT_AND) . {B.1=T,B.2=T};
request? B;
--> F
request? ;
```

Clearly (NOT | RIGHT_AND) behaves like the following algorithm:

```
let NOT_RIGHT_AND =
  algo
    request B do
      valof B.2 is
        T: valof B.1 is
             T: output F
             F: output T
           end
        F: output T
       end
    end
  end
```

We shall see that (NOT | RIGHT_AND) and NOT_RIGHT_AND have indeed the same semantics. Algorithm composition may be seen as coroutine composition [9]; in a composition f|g, any global request is given to g. The "output" instructions of g answer such requests, while the "valof" instructions of g generate requests for f. We shall be more precise in section 4.

3.3. Sequential algorithms as states.

One of the originalities of CDS is that there is actually no difference between states and algorithms, or between ground and higher-order data structures. As an example we show how to tramsform the algorithm EXAMPLE into a state.

An algorithm has a tree shape, while a state is a simple set. The first operation is to get rid of the tree structure of the text of each instruction by coding its nodes unambiguously. This is done at each line by recording the input state read so far, starting with the empty state at the first line if the instruction is directly enclosed in a request construct; if the instruction is enclosed in a from construct, then the starting point is the state listed after "from". After that transformation, EXAMPLE looks like this:

```
algo
  request C0 do
    {}              : valof B.1
    {B.1=T}         : output 1
    {B.1=F}         : valof B.2
    {B.1=F,B.2=F}   : output 0
  end
  request C1 do
    {}              : valof B.2
    {B.2=T}         : output 0
    {B.2=F}         : output 0
  end
  request C2
    {B.1=T}         : valof B.2
    {B.1=T,B.2=T}   : output 0
    {B.1=F,B.2=F}   : output 0
  end
end
```

Next we distribute the output cells in the lines:

```
algo
    {}C0              : valof B.1
    {B.1=T}C0         : output 1
    {B.1=F}C0         : valof B.2
    {B.1=F,B.2=F}C0   : output 0
    {}C1              : valof B.2
    {B.2=T}C1         : output 0
    {B.2=F}C1         : output 0
    {B.1=T}C2         : valof B.2
    {B.1=T,B.2=T}C2   : output 0
    {B.1=F,B.2=F}C2   : output 0
end
```

Finally we replace ":" by "=", we add a "," at the end of lines, and replace "algo" by "{" and "end" by "}":

```
# {
    {}C0              = valof B.1,
    {B.1=T}C0         = output 1,
    {B.1=F}C0         = valof B.2,
    {B.1=F,B.2=F}C0   = output 0,
    {}C1              = valof B.2,
    {B.2=T}C1         = output 0,
    {B.2=F}C1         = output 0,
    {B.1=T}C2         = valof B.2,
    {B.1=T,B.2=T}C2   = output 0,
    {B.1=F,B.2=F}C2   = output 0
  };
request? {}C0;
--> valof B.1
request? ;
```

As indicated by the "request?" prompt, we have indeed written a valid state of a new dcds called the exponentiation dcds (BOOL_PAIR->BOOL) of BOOL_PAIR and BOOL: its cells are pairs xc′ where x is a state of M and c′ is a cell of M′. Its values have the form "valof c" or "output v′" where c and v′ are respectively an input cell and an output value. An event "xc′=output v′" means "with input x and request c′, output the value v′", while an event "xc′=valof c" means "with input x and request c′, ask for the value of c". What we have done is essentially to decompose the algorithm into quanta of algorithmic information, in a way that fits into the formalism of concrete data structures. Notice that control instruction (valof′s) are as important as value instructions (output′s) in that decomposition. See [21] for an analogous treatment of functions in event structures. Some rules must be respected when writing the text of an algorithm to ensure that the obtained set of events is indeed a state. We shall give them in 3.4 together with the exact definition of the exponentiation dcds.

The CDSO interpreter actually only knows about states. Every algorithm is translated at parse time into a state as we just did. This state may hence be called the internal form of the algorithm. In other words the syntax of sequential algorithms is nothing but "syntactic sugar". Now it makes perfect sense to evaluate directly algorithms as normal states, without applying them to arguments:

```
# LEFT_AND;
request? {}B;
--> valof B.1
request? {B.1=T}B;
--> valof B.2;
request? {B.1=F}B;
--> output F
request? {B.1=T.B.2=F}B:
--> output F
request? {B.1=T,B.2=T}B;
--> output T
request? :
```

As a consequence, we are now able to distinguish between the two STRICT_AND algorithms, which we could not do using applications:

```
# LEFT_STRICT_AND;
request? {}B;
--> valof B.1
request? ;

# RIGHT_STRICT_AND;
request? {}B;
--> valof B.2
request? ;
```

We get two distinct answers for the same question, hence the two AND's are different in CDSO. The point is that they differ by their control, which is part of their semantics as well as their input-output behaviour.

Let us show how a composition behaves:

```
# NOT | RIGHT_AND;
request? {}B;
--> valof B.2
request? {B.2=F}B;
--> output T
request? {B.2=T}B;
--> valof B.1
request? {B.2=T,B.1=F}B;
--> output T
request? {B.2=T,B.1=T}B;
--> output F
request? ;
```

We have explored the whole state denoted by NOT | RIGHT_AND. The reader may check that the intepreter answers are the same as those one would obtain by evaluating NOT_RIGHT_AND. Hence NOT | RIGHT_AND and NOT_RIGHT_AND have indeed the same semantics.

That sequential algorithms are ordinary states has another important consequence: it is not necessary to define new primitives such as lambda-binders for defining higher-order algorithms, which are nothing but ordinary algorithms taking as arguments ordinary states, hence are also ordinary states. Let us give an example of that unusual fact: we may write a second order algorithm AND_TASTER which is able to recognize all boolean AND algorithms. Its (rather long) text is given in annex. Its input dcds is (BOOL_PAIR->BOOL) and its output dcds is the following one, just introduced for that "tasting" purpose:

```
let TASTE =
  dcds
    cell WHICH_AND values
                        IS_LEFT_AND,
                        IS_RIGHT_AND,
                        IS_LEFT_STRICT_AND,
                        IS_RIGHT_STRICT_AND,
                        IS_NOT_AN_AND
  end
```

Let us experiment with AND_TASTER:

```
# AND_TASTER . LEFT_AND;
request? WHICH_AND;
--> IS_LEFT_AND
request? ;
```

We get similar answers for the other three logical and algorithms. But AND_TASTER does not "taste" only explicit algorithms:

```
# AND_TASTER . (NOT | LEFT_AND);
request? WHICH_AND;
--> IS_NOT_AN_AND
request? ;

# AND_TASTER . ((NOT | NOT) | LEFT_AND);
request? WHICH_AND;
--> IS_LEFT_AND
request? ;
```

The two last experiments show that what is taken as input by AND_TASTER is not an expression as a piece of syntax, but the state it denotes, i.e. its semantics. In other words, CDS0 is a tool for exploring the semantics of expressions, rather like LISP is a tool for exploring their syntax (which is actually also feasible in CDS, by delaring suitable types for abstract syntax trees). We shall see later on funnier examples of the same kind.

3.4. Sequential algorithms as mathematical objects.

Before presenting the other CDS0 primitives, we define algorithms mathematically. Most of this section recalls constructions and results of [3,6], to which the reader is refered for details. The formalism of [3] is slightly rephrased, in order to match the concrete syntax of CDS0.

We first define formally the exponentiation of two dcds M, M', i.e. the dcds of sequential algorithms between M and M'.

3.4.1. Definition: Let $M=(C,V,E,\vdash)$, $M'=(C',V',E',\vdash')$ be two dcds. The exponentiation $(M\to M') = (C_{\to},V_{\to},E_{\to},\vdash_{\to})$ is defined as follows:

(i) $C_{\to} = \{xc' \mid x \text{ finite state of } M,\ c'\in C'\}$

(ii) $V_{\to} = \{\text{valof } c \mid c\in C\} \cup \{\text{output } v' \mid v'\in V'\}$

(iii) $E_{\to} = \{(xc', \text{valof } c) \mid c\in A(x)\} \cup \{(xc', \text{output } v') \mid (c',v')\in E'\}$

(iv) $(xc', \text{valof } c)\vdash_{\to} yc'$ iff $x\prec_c y$

$(x_1c'_1, \text{output } v'_1),..,(x_nc'_n, \text{output } v'_n) \vdash_{\to} xc'$
iff $x=x_1\cup\ldots\cup x_n$
and $(c'_1,v'_1),..,(c'_n,v'_n)\vdash' c'$

That $(M\to M')$ satisfies (WF) is easily verified. That it is deterministic is not obvious. It is a corollary of the lemma 3.4.2. A state of $(M\to M')$ is called a sequential algorithm.

In terms of sequential algorithms as programs, the conditions on events and enablings may be formulated as follows: any cell may be tested only once in an instruction. Only accessible cells (w.r.t. to the input state read so far) may be tested. A state listed after "from" in a from construct must be a minimal state producing all the events of an enabling of the output cell specified in the enclosing "request" statement. The two conditions (iii) and (iv) forbid any "hole" while reading input: if a cell is read, cells enabling it must have been read before. Here are typically forbidden texts:

```
valof c is
  3: valof c is
       4:...
```

```
valof C0 is
  0: valof C2 is
       0: ...
(supposing that MULT_ENAB is the input dcds)
```

the text of EXAMPLE where

```
from {B.1=T} do
 ...
end
```

is replaced by

```
from {B.1=T,B.2=F} do
  output 0
end
```

The reader may check that the set of events produced from an algorithm text satisfying the restrictions mentioned so far is a state. Reciprocally, in order to recover the text of an algorithm from a state of M->M', we have to recover the tree structure of an algorithm. The following lemma is the key of that construction:

3.4.2. Lemma: Let M, M' be two dcds, let X be a set of events of (M->M'). Then X is a sequential algorithm iff it satisfies for every output cell c':

(i) if xc', yc' are filled in X and if x, y are compatible, then $x \leq y$ or $y \leq x$. If $x \leq y$ then there exists a sequence (called covering chain in [3]) $(z_0,c_0),..,(z_n,c_n)$ s.t.

$$x = z_0 -< c_0\, z_1 -< c_1 \ldots z_n -< c_n\, y$$

$(z_i c', \text{valof } c_i) \in X$ for all i s.t. $0 \leq i \leq n$

(ii) if x is minimal s.t. xc' is filled in X, then there exist $x_1,..,x_n$, $c'_1,..,c'_n$, $v'_1,..,v'_n$, $n \geq 0$, s.t
s.t. $x = x_1 \cup \ldots \cup x_n$ and
$(x_i c'_i, \text{output } v'_i) \in X$ for all i s.t. $1 \leq i \leq n$
Property (i) is the indexed forest property of [3], and is nothing but an abstract order-theoretical definition of forests.

The input-output function defined by a sequential algorithm is very easily recovered from its state form: we set

$$a.x = \{(c',v') \mid (yc', \text{output } v') \in a \text{ for some } y \leq x\}$$

which is easily checked to be a state. We denote by $\bar{a}$ the function which associates a.x with x, and call it the input-output function of a. The above equality of course defines also the mathematical semantics of application in CDSO.

The function $\bar{a}$ is sequential in the sense of Kahn-Plotkin [3]: at any point x, for any cell $c' \in A(\bar{a}(x))$, either c' is not filled in $\bar{a}(y)$ for all $y \geq x$ or $\bar{a}$ has at least one sequentiality index, i.e. a cell c s.t.

$$x \leq y \text{ and } \bar{a}(x) <_c, \bar{a}(y) \text{ imply } x <_c y.$$

It is shown in [3] that any sequential function is the input-output function of at least one algorithm. Conversely, what structure must be added to a sequential function in order to specify exactly one sequential algorithm? Consider the different logical conjunction algorithms: they differ in the way they choose sequentiality indexes, in other words in their control strategy. That strategy may be easily defined from the state form of an algorithm:

3.4.3. Definition: the index choice function of a is a partial function i_a from $D(M) \times C'$ to C defined by

$$i_a(x,c') = c \text{ iff } (yc', \text{valof } c) \in a \text{ for some } y \leq x \text{ and } c \in A(x)$$

The pair $(\bar{a}, i_a)$ is called the abstract form of a.

Reciprocally, it is shown in [3] that a pair of a sequential function and an index choice function satisfying some consistency and minimality axioms defines a unique sequential algorithm. This correspondence is actually an order-isomorphism, if sequential algorithms are ordered by the stable ordering [3] and index choice functions by the natural ordering of partial functions ($f \leq g$ if $f(x) = g(x)$

whenever f(x) is defined).

The abstract form is well suited for defining the composition a'|a of two algorithms, and hence the mathematical semantics of the composition operation |.

3.4.4. Definition: The composition a'|a of a:M->M' and a':M'->M'' is defined by

(i) $(a'|a).x=a'.(a.x)$ for all x

(ii) $i_{a'|a}(x,c'')=c$ iff $i_{a'}(a.x,c'')=c'$ and $i_a(x,c')=c$

It is shown in [3] that these equations indeed define an algorithm, and that the concrete data structures and sequential algorithms form a cartesian closed category.

3.5. The CDSO combinators.

So far we have seen all 0-ary CDSO expressions (the states), and two binary combinators: application "." and composition "|". We finish the presentation of CDSO by studying the remaining combinators: fixpoint, pairing, currying, uncurrying.

3.5.1. Fixpoint.

As for recursive functions, any algorithm a:M->M has a least fixpoint determined by $fix(a)=lub\{a^n.\{\}\}$. The fix operator is primitive in CDSO.

3.5.2. Pairing.

The product of dcds was defined in 2.5, and the product projections programmed in 3.1. Given two states x_1 and x_2 of M_1 and M_2, we call (x_1,x_2) the state of $M_1 \times M_2$ defined by

$$(x_1,x_2) = \{(c_1.1,v_1) \mid (c_1,v_1) \in x_1\} \cup \{(c_2.2,v_2) \mid (c_2,v_2) \in x_2\}$$

Given two algorithms $a_1:M->M'_1$ and $a_2:M->M'_2$, we define the pair $<a_1,a_2>:M->M'_1xM'_2$ in the following (abstract) way:

(i) $<a_1,a_2>.x=(a_1.x,a_2.x)$

(ii) $i_{<a_1,a_2>}(x,c'_1.1) = i_{a_1}(x,c'_1)$

(iii) $i_{<a_1,a_2>}(x,c'_2.2) = i_{a_2}(x,c'_2)$

Describing the concrete form of $<a_1,a_2>$ is easy and left to the reader. The pairing operation <-,-> is primitive in CDSO.

3.5.3. Currying and uncurrying.

These operations are classical with functions: curry associates with any function f:M#M'->M'' the function curry(f):M->(M'->M'') defined by curry(f)(x)(v)=f(x,v), and uncurry is the converse operation. The definition is a bit more complicated with algorithms, but still easy on their concrete form. It is given by the following correspondence between cells and values of (M#M'->M'') and (M->(M'->M'')), according to the definition of an exponentiation dcds.

	(M#M'->M'')	(M->(M'->M''))
cells:	(x,x')c''	x(x'c'')
values:	valof c.1 valof c'.2 output v''	valof c output valof c' output output v''

The primitive CDSO operators curry and uncurry realize respectively the left-to-right and right-to-left transformations. Consider for example an event (x,x')c''=valof c'.2 of an algorithm a:(M#M'->M''). That event indicates that a tests the value of c' in M' whenever its input is (x,x'). Consider now curry(a) with input x and request x'c''. The algorithm curry(a)(x) tests c' if it is given input x' and request c'', hence x'c''=valof c' is an event of curry(a)(x). This is precisely the information output by curry(a) with input x and request x'c'', as

recorded in the event x(x'c'')=output valof c' of curry(a).

· Let us do some experiments with the interpreter:

```
# curry(RIGHT_AND);        -- type BOOL->(BOOL->BOOL)
request? {}{}B;
--> output valof B
request? {}{B=T}B;
--> valof B
request? {B=F}{B=T}B;
--> output output F
request? ;
```

Call ID_BOOL_BOOL the identity algorithm on (BOOL->BOOL)

```
# let app=uncurry(ID_BOOL_BOOL);
# app:          -- type ((BOOL->BOOL)#BOOL)->BOOL
request? {}B;
--> valof ({}B).1
request? {({}B).1=output T} B;
--> output T
request? {({}B).1=valof B} B;
--> valof B.2
request? {({}B).1=valof B, B.2=T} B;
--> valof ({B=T}B).1
request? {({B=T}B).1=output F, ({}B).1=valof B, B.2=T} B;
--> output F
request? ;
```

Although it may look rather complicated, the last example is particularly instructive: uncurry(ID_BOOL_BOOL) is nothing but the application algorithm, which takes as argument (f,x) with f:BOOL->BOOL and x:BOOL, and which applies f to x. With empty argument and request B as in the first request, app asks f what it does with argument {} and request B. This is the meaning of the first interpreter response ("valof ({}B).1" for (f,x) is nothing but "valof {}B" for f). If f directly outputs T as in the second listed request, then app also outputs T. If f asks for its argument as in the third request, then app asks for the value of its second argument (the argument of f). If that value is T as in the fourth request, app asks f what it does with argument {B=T} and request B. If f then outputs F as in the last request, then app also outputs F. In short, app asks f what to do and then does it.

3.6. Extending the language.

The choice of the CDS0 combinators is not arbitrary. It is suggested by the mathematical theory of cartesian closed categories [12,3,2,18]. The dcds and sequential algorithms form such a category [3]. We shall not give details here. But we shall indicate how our combinators allow to "implement" easily more classical languages based for example on the lambda calculus, and we shall study completely an example: Plotkin's language PCF (Programming Computable Functions) [16], also considered elsewhere in this volume [4]. We shall pay no attention to the efficiency of the translation here (improving it is a subject in itself).

PCF has two ground types i and o (integers and booleans), and derived types $(\sigma\to\tau)$ for any types σ and τ. The primitive functions are well-known: the integer constants 0,1,-1,..,. the booleans "tt" and "ff", successor and predecessor functions "succ" and "pred", the test to zero "iszero", the conditional "cond". The terms are build as usual by application, lambda-abstraction (written with the symbol \, and fixpoint (written Y). Here is a way to define integer addition in PCF:

```
Y(\f.\x.\y. cond (iszero x) y (f(pred x)(succ y)))
```

The basic types INTEGER and BOOLEAN have already been defined in CDS0. Integer and boolean constants are simply interpreted as the corresponding states. It is easy to define algorithms for the other basic function symbols. For example:

```
let SUCC =
  algo
    request N do
      valof N is
        &V: output &V+1
      end
    end
  end
```

```
let IF_I = curry(curry(
  algo
    request N do
      valof B.1 is
        T: transmit N.2
        F: transmit N.3
      end
    end
  end))
```

The main problem is the elimination of the PCF variables. We shall not use the S and K combinators as in [20], but rather code the usual manipulation of environments into CDSO, by defining appropriate dcds. For simplicity we shall forget about types. Let @ID be a new name generator which generates all identifier names:

```
let VAR =  -- data structure for variables
  dcds
    cell NAME values @ID
  end

let ENV(*) =  -- data structure for environments over *
  dcds
    graft *.@ID
  end
```

Hence ENV(*) is an infinite product of the parameter * indexed by variable names. We need two (classical) algorithms for manipulating environments. The first one fetches the value of the variable x in an environment, the second one changes that value; we shall mainly use their curried form.

```
let FETCH: VAR#ENV->* =
  algo
    request &C do
      valof NAME.1 is          -- fetch the variable name
        &X: transmit &C.&X.2   -- fecth the value and give it back
      end
    end
  end

let CFETCH=curry(FETCH)
```

```
let SUBST: VAR#(ENV#*)->ENV =
  algo
    request &C.&X do                    -- &C cell, &X variable name
      valof NAME.1 is                   -- fetch substituted var.
        &X: transmit &C.2.2             -- the same: value in *
        &Y with  not &X=&Y:             -- different:
            transmit &C.&X.1.2          --    fetch old value
      end
    end
  end

let CSUBST=curry(SUBST)     -- type VAR->(ENV#*->ENV)
```

We also need the usual K operator:

```
let K: *1->(*2->*1) = curry(
  algo
    request &C do
      transmit &C.1
    end
  end)
```

and a PFIX constant for interpreting Y (take for example fix(curry(app|<snd,app>))), where snd is the second product projection.

Using the above operators, one could define for any PCF term M an interpretation function ⟦M⟧ from environments to value in the following way:

⟦f⟧(p) = F

for any function symbol f with associated algorithm F

⟦Y⟧(p) = PFIX

⟦x⟧(p) = CFETCH.{NAME=x}.p

for any variable x

⟦MN⟧(p) = (⟦M⟧(p)).(⟦N⟧(p))

⟦\x.M⟧(p)(d) = ⟦M⟧(CSUBST.{NAME=x}.p.d)

But these equations are not enough for defining algorithms, which are not simply determined by their values on arguments. The next step is to get rid of the meta-variables p and d, using our combinators for abstracting the above formulae into true CDS01 expressions. One may

write:

⟦f⟧ = K.F
for any PCF function symbol f with associated algorithm F.

⟦Y⟧ = K . PFIX

⟦x⟧ = CFETCH . {NAME=x}

⟦MN⟧ = uncurry(⟦M⟧) | <ID, ⟦N⟧>
(ID is the identity algorithm)

⟦\x.M⟧ = curry(⟦M⟧|(CSUBST.{NAME=x}))

This translation is extensively studied in [2]. We leave to the reader to check that the first formulae are indeed derived from the second ones by applying suitable arguments ρ and d. For any M, the object ⟦M⟧ is now an algorithm from environments to values, the control part of which telling how M uses its free variables. (One may even go further, as suggested in [3]: one may organize the PCF terms themselves into a dcds, and turn the mere function ⟦ ⟧ into an algorithm, which really acts as a PCF interpreter written in CDSO!)

In the actual CDSO systems, we have several parsers: we indicate to switch to PCF by the command "pcf;". We then type in PCF expressions, which are immediately expanded into their CDSO form, and evaluated as any CDSO expression. Let us evaluate the above addition expression:

```
# pcf;      -- the prompt changes to $ in PCF
$ let lplus = Y(\f.\x.\y. cond (iszero x) y (f(pred x)(succ y)))
$ lplus 3 5;
request? N;
-->8
request? ;
```

Of course we can evaluate lplus by itself, as any PCF term of any type. We can therefore distinguish it from the right addition rplus which recurses on y (remember that rplus and lplus have type i->(i->i)):

```
$ lplus;
request? {}{}N;
--> valof N
request? {N=3}{}N;
--> output valof N
request? {N=3}{N=5}N;
--> output 8
request? ;

$ Y(\f.\x.\y. cond (iszero y) x (f(pred y)(succ x)));
request? {}{}N;
--> output valof N
request? {}{N=5}N;
--> valof N
request? {N=3}{N=5}N;
--> output 8
request? ;
```

Hence CDS0 may also serve as a tool for analyzing the control of PCF programs.

Finally let us notice that lambda-definition may not be used for defining non-functionnally monotonic objects such as AND_TASTER. In that sense PCF-like calculi are strictly less powerful than CDS0.

3.7. Manipulating infinite objects.

In the translation of PCF ito CDS01, we have already manipulated purely infinite data structures such as ENV. However only a finite part of that structure was useful at a time. We come to a fully infinite example (due to Kahn-Mac Queen [9]). Remember the dcds FRACTIONS and POWERSERIES of 2.4. We shall compute the exponential powerseries by the following equation:

```
letrec EXP=INTEGRATE(1,EXP)
```

INTEGRATE(c,s) being the integral of s with constant c. It is easy to remove the variable EXP on the right-hand-side, writing an actual CDS0 expression:

```
let EXP=fix(curry(INTEGRATE).{N=1})
```

INTEGRATE is easily defined:

```
let INTEGRATE: FRACTIONS#POWERSERIES->POWERSERIES =
  algo
    request R.0 do
      transmit R.1                    -- the constant
    end
    request R.&N with &N>=0 do
      valof R.(&N-1).2 is
        &P.&Q: output &P.(&Q*&N)      -- integration step
      end
    end
  end
```

The state EXP is really infinite, all the cells of POWERSERIES being filled in it; But its "computation" is still possible, since the interpreter computes only on request:

```
# EXP:
request? R.0;
--> 1
request? R.6;
--> 720
request? ;
```

Notice that we used a direct-access representation of powerseries, not a stream representation as in [9]. Indeed there is no reason to impose access conditions to coefficients. One could solve the following equations:

```
letrec SINE = INTEGRATE(0,COSINE)
   and COSINE = INTEGRATE(1,MINUS(SINE))
```

Then the 4th coefficient of SINE needs not to be computed for computing the 5th one. With streams, it would have to be computed anyway, just to access the 5th one (this is the case in the Kahn-Mac Queen system). The extra access structure of streams make them less lazy.

3.8. Editing program semantics.

We finish this section with an amazing example due to F. Montagnac and A. Ressouche. The problem is to define the fibonnacci function in CDSO, without using variables at all (that would of course be easy in PCF). The fibonnacci functional is defined by

Ffibo(f)(n)=if n=1 or n=2 then 1 else f(n-1)+f(n-2)

and the function fibo is just the least fixpoint of that functional. The technique we use for defining directly Ffibo is close to the technique used in LISP for tracing functions: one uses a LISP program for inserting tracing instructions in the text of the lisp function f to be traced, in other words one edits the text of f. In CDSO, we shall similarly edit the semantics of Ffibo's argument f, and do some surgery on it. We give the text of Ffibo and explain it.

```
let Ffibo: (INTEGER->INTEGER)->(INTEGER->INTEGER) =
  algo
    request {}N do
      output valof N
    end
    request {N=1}N do
      output output 1
    end
    request {N=2}N do
      output output 1
    end
    request {N=&I with &I>2}N do
      valof {}N is
        output &X: output output &X+&X
        valof N:
          valof {N=&I-1}N is
            output &X:
              valof {N=&I-2}N is
                output &Y: output output &X+&Y
              end
          end
      end
    end
  end
```

The idea is as follows: In any case Ffibo.f must test its argument, hence Ffibo outputs "valof N" on the request {}N. If that argument is 1 or 2, Ffibo.f always outputs 1, hence Ffibo outputs the statement "output 1" for the requests {N=1}N and {N=2}N. Given the request {N=&I}N for &I>2, Ffibo must know what f does, hence execute a "valof {}N" instruction. If f is a constant, i.e. outputs &X on {}, Ffibo outputs the instruction "output &X+&X". If f does "valof N", call &X and &Y the values f(&I-1) and f(&I-2). Then Ffibo outputs the instruction of ouputting &X+&Y. Let us try:

```
# fix(Ffibo);
request? {}N
--> valof N
request? {N=1}N;
--> output 1
request? {N=20}N;
--> output 6765
request? ;
```

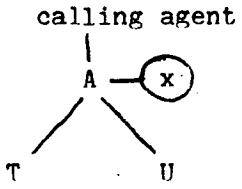
4. The CDS01 operational semantics.

At the time being we have two different interpreters of CDS0, named CDS01 and CDS02. They are based on two quite different operational semantics. Both interpreters use question-answer mechanisms: a question to a an expression is understood as a request for the value of some cell it denotes, and an answer is simply a value of that cell. In higher-order types (types containing the -> operator), cell names contain states. In CDS01, questions are always extensional, i.e. are such that the states they involve are pure CDS0 states as sets of events. On the contrary, CDS02 questions are intentional: the states are themselves CDS02 expressions, and the cell associated with a question is obtained by replacing each expression by its denotation, which is a state. We shall only study CDS01 here. We shall not treat name variables, which are handled by straightforward pattern-matching techniques.

4.1. Intuitive presentation.

The CDS01 evaluation mechanism raises non-standard problems which we study in a rather long discussion based on examples. The mechanism is best understood in terms of communicating agents or coroutines [14,9]. Consider for example an application A=T.U, where T has type (M->M') and U has type M. Then three communicating agents are generated: a main agent A corresponding to the application node, and two subagents computing respectively T and U, also called T and U for simplicity (they are of course generated by induction on the structure of the terms T and U). The agent A communicates with T and U, which ignore eachother: moreover A maintains a local memory containing a state x of M, which is the currently computed approximation of the denotation of U. The agent U

produces events in x, and x itself is used by A for generating questions to T.



As an example, let us analyze a possible evaluation of T.U=LEFT_AND.{B.1=T,B.2=F}, when A receives the request B from its caller. The state x is initially empty, and A starts by asking T the value of xB={}B. The answer being "valof B.1", A asks U the value of B.1. The answer "T" is recorded in x, which becomes {B.1=T}. Then A asks T again the value of xB={B.1=T}B. The anwser being "valof B.2", A asks U the value of B.2, and after U's response "F", x becomes {B.1=T,B.2=F}. Now the question xB={B.1=T,B.2=F}B to T generates the answer "output F", and the result "F" is output to the calling agent. The strategy used here is call by need: ask T what needs to be done and do it. We shall see that other strategies are possible.

The question now arises of the status of x when the computation is finished: should its current value be kept, or should x be reinitialized to {}? That question is actually delicate: keeping the current value of x uses much storage, but has a "memo function" effect: if later on A is called again with the request B, the first question to T will be simply xB={B.1=T,B.2=F}B: it will generate at once the response F, with no further request to U. On the contrary resetting x to {} would use less storage but more time, since x would have to be completely recomputed. This is a fairly classical time-storage exchange problem, which in usual languages is solved by not using memo-functions: one observes that most of the time memo-functions keep too much unnecessary information, as in the classical recursive fibonnacci example (when computing fib(n), memo-functions keep all the values fib(0) to fib(n-1), while keeping only two values would suffice). However the situation is less clear in CDSO. Consider the following example:

```
let M =
  dcds
    cell C1 values 0
    cell C2 values 0 access C1=0
  end
```

and let ID be the identity on M:

```
let ID:M->M =
  algo
    request C1 do transmit C1 end
    request C2 from {C1=0} do transmit C2 end
  end
```

or in state form:

```
{{}C1           = valof C1,
 {C1=0}C1       = output 0,
 {C1=0}C2       = valof C2,
 {C1=0,C2=0}C2 = output 0}
```

Consider the term T.U=ID.{C1=0,C2=0}. The request C1 to A leads to the answer 0 with x={C1=0}. Now if x is kept to that value and if A receives the request C2, A asks T=ID the value of xC2={C1=0}C2. This is a perfectly valid request to ID, which returns "valof C2", and the computation continues normally until 0 is output with x={C1=0,C2=0}. On the contrary assume that x is reset to {} when A receives the request C2: now {}C2 is indeed empty and even not enabled in ID. Therefore A should first reconstruct the state x={C1=0}. In that case this step is feasible, since one sees in the definition of M that C2 depends upon C1. However that reconstruction process is not feasible in arbitrary dcds, as shown by the next example:

```
let STABLE =
  dcds
    cell C1 values T,F
    cell C2 values T,F
    cell C3 values T,F
    cell C4 values 1,2,3
              access C1=T,C2=F or C2=T,C3=F or C3=T,C1=F
  end
```

```
let DISCRIMINATE: STABLE->STABLE =
  algo
    request C1 do transmit C1 end
    request C2 do transmit C2 end
    request C3 do transmit C3 end
    request C4
      from {C1=T,C2=F} do
        valof C4 is
          1: output 1
        end
      end
      from {C2=T,C3=F} do
        valof C4 is
          1: output 2
        end
      end
      from {C3=T,C1=F} do
        valof C4 is
          1: output 3
        end
      end
  end
```

The dcds STABLE is actually stable (deterministic), but has the annoying property that the predicate which tests whether C4 is enabled in a given state is a stable but not sequential function, see [3]. The DISCRIMINATE algorithm deserves its name, since in discriminates which "from" part has enabled C4 in the output. Consider then a term DISCRIMINATE.U where U is some state of STABLE where C4 is enabled; If C4 is given as a question to that term with x reset to {}, one does not know how to reconstruct the approximation of U which enabled C4: the reconstruction process would involve at least one of the questions C1, C2, C3 to U, and there is no way to choose which one a priori; if one chooses C1, then U may be such that C2=T,C3=F,C4=1 while C1 is empty and the question C1 to U actually leads to an endless computation (the actual construction of such an U is given in [7]). The interpreter would then loop instead of giving the correct answer 2. The situation is similar for C2 and C3.

There are now two solutions: to forbid STABLE, or to keep x. The first solution involves the definition of sequential concrete data structures and will not be treated here (it is however necessary for CDSO2). We choose the solution of keeping x.

We are then faced with another problem, which is however simpler. Condider another example:

```
let T_PAIR =
  dcds
    cell 0.1 values T
    cell 0.2 values T
  end
```

Let ID be the identity on T_PAIR. It is the following state:

```
{{}(0.1)       = valof 0.1
 {0.1=T}(0.1) = output T
 {}(0.2)       = valof 0.2
 {0.2=T}(0.2) = output T}
```

Consider ID.U with U={0.1=T,0.2=T}. A first request 0.2 generates the answer T with x={0.2=T}. Now a request 0.1 generates a request x(0.1)={0.2=T}(0.1) to ID, and again that cell is not enabled in ID. But here the situation is simpler than before: instead of having not enough information in x, we have too much information: the information 0.2=T is simply irrelevant w.r.t. 0.1, and definitely not cumbersome. Hence the normally incorrect question {0.2=T}(0.1) to ID may safely generate the same answer "valof 0.1" as would the correct question {}(0.1). Notice that the role of the computation stategy i_a of an algorithm is precisely to take care of that fact: here one has

$$i_{ID}(\{\},0.1) = i_{ID}(\{0.2=T\},0.1) = 0.1$$

To treat formally that problem, we shall use the fact that {} is the biggest state z below {0.2=T} such that z(0.1) is filled in ID. For this purpose it is convenient to introduce an ordering of cells, induced by the ordering of states, saying {}(0.1) $\leq$ {0.2=T}(0.1) A very last difficulty appears when ID is curried: the questions {}(0.2) and {0.2=T}(0.1) are then transformed into {}{}(0.2) and {}{0=T}(0.1), and the useless information 0=T appears in the second state of the question. The ordering on cells must therefore be defined inductively.

4.1.1. Definition: The cell names in DCDS may have the following form:

(i) a token or number

(ii) a pair xc′ where x is a state anf c′ a cell name

(iii) a list of cell names, where only the first name may contain states For example xyc′.1.2 is correct, but xc′.yc′.2 is not

Then we define an ordering $\leq$ on cells as follows:

(i) $\leq$ is equality for tokens or numbers

(ii) $\leq$ is defined componentwise for lists

(iii) $x_1c'_1 \leq x_2c'_2$ holds iff $x_1 \leq x_2$ and $c'_1 \leq c'_2$

4.1.2. Definition: Given any state x and any cell name c, one easily checks that the set $\{c_1 | c_1 \leq c$ and $c_1=v$ is in x, $c=v$ is in $E\}$ has a least upper bound when it is not empty. This least upper bound is called the projection of c on x: it corresponds to an event $c_1=v$ of x, and we set $\bar{x}(c)=v$. The function $\bar{x}$ is called the extended evaluation function of x.

In the above example, setting x=ID and c={0.2=tt}(0.1), one sees that the projection of c on x is {}(0.1), so that $\bar{x}$(c)=valof 0.1 as required. Setting x=curry(ID) and c={}{0=T}(0.1), the projection of c on x is {}{}(0.1) with $\bar{x}$(c)=valof 0.

4.2. The CDS01 rewrite rules. The full abstraction theorem.

The behaviour of an application was discussed rather informally in the previous section. We shall be much more formal here, and we shall define the CDS01 semantics by conditional rewrite rules as in [17].

We first define the CDS01 expressions. They resemble the CDS0 ones, except that nodes may have local storage according to their form. The set of CDS01 expressions is defined inductively:

(i) Any CDS0 state of type M is a CDS01 expression of the same type.

(ii) If T is a CDS01 expression, then curry(T) and uncurry(T) are CDS01 expressions (same types as in CDS0).

(iii) If T_1,T_2 are CDS01 expressions, then $\langle T_1,T_2\rangle$ is a CDS01 expression (same types as in CDS0).

(iv) If T,U are CDS01 expressions with T of type (M->M′) and U of type M, if x is a state of M, then [T.U,x] is a CDS01 expression of type M′.

(v) If T is a CDS01 expression of type (M->M), if x is a state of M, then [fix(T),x] is a CDS01 expression of type M.

(vi) If T,T′ are CDS01 expressions of type (M->M′) and (M′->M″), and if F is a set of pairs x,x′ of states of M,M′, then [T|T′,F] is a CDS01 expression of type (M->M″).

Hence local storage is used only for application, fixpoint and composition. The rather complex form of the composition local storage will be explained later on.

The objects involved in the rewrite rules have two possible forms:

(i) <u>questions</u> T?c where T is a CDS01 term anc c is a cell name (of the dcds corresponding to the type of T)

(ii) <u>answers</u> T!v where T is a CDS01 term and v is a value (CDS01 terms have to ke kept in answers to take care of local storage modifications)

Computations will of course lead from questions to answers, through intermediate questions. A question T?c will receive an answer v whenever $\bar{T}(c)=v$, in the sense of 4.1.

We start by giving the simplest rules, which simply mimic denotational semantics

State:

$$\frac{\bar{x}(c)=v}{x?c \rightarrow x!v}$$

Pairing:

$$\frac{T?c \xrightarrow{*} T_1!v_1}{\langle T,U\rangle?c.1 \rightarrow \langle T_1,U\rangle!v_1}$$

$$\frac{U?c \xrightarrow{*} U_1!v}{\langle T,U\rangle?c.2 \rightarrow \langle T,U_1\rangle!v}$$

A question is transmitted either to T or to U according to its suffix.

Currying:

$$\frac{T?(x,x')c'' \xrightarrow{*} T_1!\text{valof } c.1}{\text{curry}(T)?x(x'c'') \rightarrow \text{curry}(T_1)!\text{valof } c}$$

$$\frac{T?(x,x')c'' \xrightarrow{*} T_1!\text{valof } c'.2}{\text{curry}(T)?x(x'c'') \rightarrow \text{curry}(T_1)!\text{output valof } c'}$$

$$\frac{T?(x,x')c'' \xrightarrow{*} T_1!\text{output } v''}{\text{curry}(T)?x(x'c'') \rightarrow \text{curry}(T_1)!\text{output output } v''}$$

Uncurrying:

$$\frac{T?x(x'c'') \xrightarrow{*} T_1!\text{valof } c}{\text{uncurry}(T)?(x,x')c'' \rightarrow \text{uncurry}(T_1)!\text{valof } c.1}$$

$$\frac{T?x(x'c'') \xrightarrow{*} T_1!\text{output valof } c'}{\text{uncurry}(T)?(x,x')c'' \rightarrow \text{uncurry}(T_1)!\text{valof } c'.2}$$

$$\frac{T?x(x'c'') \xrightarrow{*} T_1!\text{output output } v''}{\text{uncurry}(T)?(x,x')c'' \rightarrow \text{uncurry}(T_1)!\text{output } v''}$$

Next we study application and fixpoint, which involve simple local storage.

Application:

The local storage retains the information computed so far from the application argument. The rules are as follows:

$$\frac{T?xc' \xrightarrow{*} T_1!\text{output } v'}{[T.U,x]?c' \to [T_1.U,x]!v'}$$

$$\frac{T?xc' \xrightarrow{*} T_1!\text{valof } c \qquad U?c \xrightarrow{*} U_1!v}{[T.U,x]?c \to [T_1.U_1, xU\{c=v\}]?c}$$

In the first rule, x contains already enough information for T to output a value, and that value is simply returned. In the second rule, this is not the case and T asks for the value of a cell c in U. If that cell contains v in U, the event c=v is added to x and the computation may resume from that new partial information on the value of U.

Fixpoint:

Again the local storage contains an approximation of the argument's value.

$$\frac{T?xc \xrightarrow{*} T_1!\text{output } v}{[fix(T),x]?c \to [fix(T_1),x]!v}$$

$$\frac{T?xc \xrightarrow{*} T_1!\text{valof } c_1 \qquad [fix(T_1),x]?c_1 \xrightarrow{*} [fix(T_2),y]!v}{[fix(T),x]?c_1 \to [fix(T_2),yU\{c_1=v\}]?c}$$

Composition:

That last combinator is slightly more complicated. To understand why, let us explain intuitively how a composition $T'|T$ is computed. Any request has the form xc'', where x is an explicit state. It is clear

that x acts as an input for T while c″ acts as a request for T′. One rule for composition could be:

$$\frac{T'?T(x)c'' \xrightarrow{*} T'_1!\text{valof } c' \qquad T?xc' \xrightarrow{*} T_1!\text{valof } c}{T'|T?xc'' \to T'_1|T_1!\text{valof } c}$$

But such a rule is not permitted, since T′?T(x)c″ is not a valid CDS01 question, T(x) not being an explicit state. The local storage will contain enough information to reconstruct an adequate approximation x′ of T(x). It will consist of sets of pairs (x,x′). Given such a set F and a state x of M, we write

$$(y,y')=\bigcup\{(z,z')\in F \mid z\leq x\}$$

Of course (y,y′)=({},{}) if the set on the right-hand-side is empty. Here are the rewriting rules:

$$\frac{T'?y'c'' \xrightarrow{*} T'_1!\text{output } v''}{[T'|T,F]?xc'' \to [T'_1|T,F]!\text{output } v''}$$

$$\frac{T'?y'c'' \xrightarrow{*} T'_1!\text{valof } c' \qquad T?yc' \xrightarrow{*} T_1!\text{valof } c \qquad c\notin F(x)}{[T'|T,F]?xc'' \to [T_1|T'_1,F]!\text{valof } c}$$

$$\frac{T'?y'c'' \xrightarrow{*} T'_1!\text{valof } c' \qquad T?yc' \xrightarrow{*} T_1!\text{valof } c \qquad c=v\in x}{[T'|T,F]?xc'' \to [T_1|T'_1,FU\{(xU\{c=v\},y')\}]?xc''}$$

$$\frac{T'?y'c'' \xrightarrow{*} T'_1!\text{valof } c' \qquad T?yc' \xrightarrow{*} T_1!\text{output } v'}{[T'|T,F]?xc'' \to [T'_1|T_1,FU\{(x,y'U\{c'=v'\})\}]?xc''}$$

A computation by the above rule is simply a deduction in the formal system they define. Set for example X=LEFT_AND and Y={B.1=T,B.2=F}; here is a computation of X.Y

```
X?{}B -> X!valof B.1
Y?B.1 -> Y!T
[X.Y,{}]?B -> [X.Y,{B.1=T}]?B
X?{B.1=T}B -> X!valof B.2
Y?B.2 -> Y!F
[X.Y,{B.1=T}]?B ->[X.Y,{B.1=T,B.2=F}]?B
X?{B.1=T,B.2=F}B -> X!output F
[X.Y,{B.1=T,B.2=F}]?B -> [X.Y,{B.1=T,B.2=F}]!F
```

Hence one has $[X.Y,\{\}]?B \xrightarrow{*} [X.Y,\{B.1=T,B.2=F\}]!F$ by transitivity, meaning that the actual anwer is F. Up to trivial interversion of some steps, the given deduction is the only possible one.

4.3. Sessions and the full abstraction theorem.

We have just seen what a computation by the rules is, and we have now to define what an interpretation session is. Intuitively a session is given by a starting CDS0 term and a sequence of question-answers, mimicing the terminal sessions considered before. We shall first consider total sessions where the interpreter always yields back a value (as usual, we omit types).

4.3.1. Definition: A session is a sequence

```
T;
request? c1:
--> v1
request? c2:
--> v2
...
request? cn:
--> vn
```

where the following conditions are satisfied, writing $X_i=\{c_j=v_j \mid j\leq i\}$:

(i) For each $i\geq 0$, c_i is enabled by X_i

(ii) Call T_0 the CDS01 term obtained from T by adding empty storage units. There must exist CDS01 terms T_i such that for each i one has $T_i?c_i \xrightarrow{*} T_{i+1}!v_i$

Hence we require that each question is enabled by the state got so far. (Notice that we forget here about the ordering of cells, which is meaningful only inside the interpreter's computations.)

The essential full abstraction result shows the identity of the denotational semantics of CDS0 and of its operational semantics CDS01. We just state it here, see [7] for the proof.

4.3.2. The full abstraction theorem: Let T be a CDS0 term, let ⟦T⟧ be the state denoted by T as given by the denotational semantics. Then $(c,v) \in ⟦T⟧$ if and only if there exists a session

```
T;
request? c0
--> v0
request? c1
--> v1
...
request?c
--> v
```

We still have to treat the case of empty cells. By the full abstraction theorem, $T?c \xrightarrow{*} U!v$ is impossible for any U and v if c is not filled in ⟦T⟧. Then there are three cases:

(i) The cell c is not enabled in ⟦T⟧. This is detected by the type checker (not implemented presently).

(ii) All reductions from T?c are finite: the interpreter detects that fact and responds with "-->". This happens typically in X.Y where Y is an explicit state: if X asks for the value of c_1 in Y and if c_1 is empty in Y, one may deduce that c is also empty in X.Y.

(iii) All reductions from T?c are infinite: then the interpreter loops. This happens typically in fix(ID).

The CDS02 semantics has exactly the same properties. But is is simpler in some respect: there is no need to ask only for enabled cells in a session: any cell is a valid question at any time.

4.4. Lazy versus eager evaluation. Evaluation versus reduction.

The CDS01 rewrite rules that we have given force lazy evaluation. Consider for example a question [T.U,x]?c'. By the second rule of application evaluation of U?c is possible only if T?xc' outputs a "valof

c" statement, and it becomes then necessary. The situation is similar for composition and fixpoint. The rules are indeed sequential in the sense that there is always one critical operation to perform. Hence only one of our computing agents needs to be active at a time. Our implementation on a sequential machine uses that fact.

However one may imagine totally different <u>eager evaluation</u> mechanisms in which the agents really work in parallel, each at his own pace. Replace the application rules by the following ones:

$$\frac{T?xc' \overset{*}{\rightarrow} T_1!\text{output } v'}{[T.U,x]?c' \rightarrow [T_1.U,x]!v'}$$

$$\frac{U?c \overset{*}{\rightarrow} U_1!v}{[T.U,x]?c' \rightarrow [T.U_1,x\cup\{c=v\}]?c'}$$

Then T and U become really independent, and U may produce values in advance, even before the application T.U is called at all. Here is for example an evaluation of T.U=LEFT_AND.{B.1=T,B.2=F}:

U?B.2 -> U!F
[T.U,{}]?B -> [T.U,{B.2=F}]?B
U?B.1 -> U!T
[T.U,{B.2=F}]?B -> [T.U,{B.2=F,B.1=T}]?B
T?{B.2=F,B.1=T}B -> T!output F
[T.U,{B.2=F,B.1=T}]?B -> [T.U,{B.2=F,B.1=T}]!F

Here U starts computing B.2, which is not the first cell that T would ask.

The problem of what U should compute has not been investigated yet, but one could use the techniques of [11]. One could also profitably intermix both evaluation mechanisms. From time to time, T could consult the intermediate value x. If $T?xc'\rightarrow T_1!v$ of c, one may send a signal to U that it should compute c in priority. Any computation strategy would then be correct.

Our evaluation strategy leaves CDS0 terms basically unchanged, except in the evolution of their local storage units. On the contrary, in classical combinatory calculi one uses term reduction, with formulae such as Kab->a and Sabc->ac(bc). All these formulae are perfectly valid in CDS0, and it is generally useful to apply some of them at "compile time" in order to simplify the CDS0 terms to evaluate (especially when they are obtained by translation of higher-level constructs). Actually evaluation and term reduction may be freely intermixed at any time. There is much work to do in order to understand what are the best possible strategies.

5. Conclusion.

We put the emphasis on two aspects of CDS0: the manipulation of higher-order objects and the mathematical semantics. In the full abstraction theorem, we showed the coincidence of the denotational and operational semantics. To our knowledge, CDS0 is the only sequential language for which such a result exists, at least if we require the denotational semantics to be somewhat natural.

With respect to implementation aspects, we are working on making CDS0 relatively fast. The CDS02 interpreter is quite promising there, and we are planning to build a LISP-like CDS02 compiler. The efficiency should become comparable to that of more classical languages. One should also try distributed implementations, as suggested in 4.4.

The CDS0 language is low-level at least as far as computable terms are concerned, and its syntax is heavy. We are currently working on a high-level version called simply CDS and based on a ML-like external syntax. We have shown how to translate lambda-calculus based languages into CDS01, but the translation we gave is not very efficient. Progress can certainly be done there.

Acknowledgements

We thank M.Devin, F. Montagnac and A. Ressouche who played an essential role in the definition and implementation of CDS01 and CDS02.

References

1. G. Berry, "Programming with Concrete Data Structures and Sequential Algorithms," Proc. ACM Conf. on Functional Programming Languages and Computer Architecture (1981).

2. G. Berry, "Some Syntactic and Categorical Constructions of Lambda-calculus Models," Rapport INRIA 80 (Juin 1981).

3. G. Berry and P.L. Curien, "Sequential Algorithms on Concrete Data Structures," Theoretical Computer Science Vol. 20, pp.265-321 (1982).

4. G. Berry and P.L. Curien, "Full Abstraction for Sequential Languages: the State of the Art," in This volume (1983).

5. L. Cardelli, "ML," VAX Implementation, Edinburgh University and Bell Laboratories.

6. P.L. Curien, "Algorithmes Séquentiels sur structures de données concrètes," Thèse de Troisième cycle, Université Paris VII (Mars 1979).

7. P.L. Curien, "Full Abstraction for a Calculus of Sequential Algorithms," LITP report 82-5, University Paris VII (1982).

8. M. Gordon, R. Milner, and C. Wadsworth, Edinburgh LCF, Lecture Notes in Computer Science 78, Springer Verlag (1980).

9. G. Kahn and D. MacQueen, "Coroutines and Networks of Parallel Processes," pp. 993-998 in Proc. Information Processing 1977, North-Holland Pub. Co. (1977).

10. G. Kahn and G. Plotkin, "Structures de données concrètes," Rapport IRIA-LABORIA 336 (1978).

11. R. M. Keller and R. Sleep, "Applicative Caching: Programmer Control of Object Sharing and Lifetime in Distributed Implementations of Applicative Languages," Proc. ACM Conf. on Functional Programming

Languages and Machine Architecture (Oct 18-22, 1981).

12. S. MacLane, Categories for the Working Mathematician, Springer Verlag Graduate Texts in Mathematics (1971).

13. R. Milner, "Fully Abstract Models of Typed Lambda-Calculi," Theoretical Computer Science Vol. 4(1), pp.1-23 (1977).

14. R. Milner, "A Calculus of Communicating Systems," Springer-Verlag LNCS 92 (1980).

15. M. Nielsen, G. Plotkin, and G. Winskel, "Petri Nets, Event Structures and Domains," Theoretical Computer Science Vol. 13(1), p.85,108 (1981).

16. G. Plotkin, "LCF Considered as a Programming Language," Theoretical Computer Science Vol. 5(3), pp.223-256 (dec. 1977).

17. G.D. Plotkin, "A Structural Approach to Operational Semantics," DAIMI FN-19, University of Aarhus (1981).

18. D.S. Scott, "Relating Theories of the Lambda-Calculus," pp. 403-450 in To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism, ed. J.P Seldin and J.R. Hindley, Academic Press (1980).

19. M.B. Smyth and G.D. Plotkin, "The Category-theoretic Solution of Recursive Domain Equations," Proc. 18th. Symp. on Foundations of Computer Science, Providence, R.I. (1977).

20. D.A. Turner, "A New Implementation Technique for Applicative Languages," Software Practice and Experience Vol. 9, pp.31-49 (1979).

21. G. Winskel, "Events in Computations," Ph.D. Thesis, Univ of Edinburgh (1980).

ANNEX: the text of AND_TASTER

```
let AND_TASTER: (BOOL#BOOL->BOOL)->TASTE = algo request WHICH_AND do
  valof {}B is
    output T: output IS_NOT_AN_AND
    output F: output IS_NOT_AN_AND
    valof B.1:
      valof {B.1=T}B is
        output T: output IS_NOT_AN_AND
        output F: output IS_NOT_AN_AND
        valof B.2:
          valof {B.1=T,B.2=T}B is
            output F: output IS_NOT_AN_AND
            output T:
              valof {B.1=T,B.2=F}B is
                output T: output IS_NOT_AN_AND
                output F:
                  valof {B.1=F}B is
                    output T: output IS_NOT_AN_AND
                    output F: output IS_LEFT_AND
                    valof B.2:
                      valof {B.1=F,B.2=T}B is
                        output T: output IS_NOT_AN_AND
                        output F:
                          valof {B.1=F,B.2=F} is
                            output T: output IS_NOT_AN_AND
                            output F: output IS_LEFT_STRICT_AND
                          end
                      end
                  end
              end
          end
      end
    valof B.2: ...   -- symmetrical
  end
```

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique