UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT

Sort Theory

M.A. Nait Abdallah

Research Report
CS-82-19

September 1982

UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT

Sort Theory

by

M.A. Nait Abdallah

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

Research Report CS-82-19
September, 1982

Sort Theory

Introduction

We give in this paper an intuitive and tutorial presentation of the basic ideas of sort theory. The originator of sort theory is L. Nolin, who describes his work [7,8,9,10] as "une théorie des algorithmes spécialement concue pour les besoins de l'informatique" (e.g. "an algorithm theory especially conceived for computer science purposes"). Although a careful study shows that this notion of an algorithm has operational connotations as well, as we illustrate in the present paper, it does differ, however, from the usual notion of an algorithm. It is in fact closer to that of a generalized type, or an algebraic sort, than to algorithms as they are known in complexity theory. Whence our term "sort" for what Nolin calls "algorithmes".

The author is going to give here his interpretation of sort theory, although this interpretation is influenced by what Nolin told him.

The aim here is twofold. First we would like to remedy the lack of literature by giving an accessible account of the basic ideas of sort theory. Second, we would like to give an illustration of the relevance of these ideas for solving some problems of program semantics, and outline a comparison with the usual cpo approach.

Our contribution in the present paper is a stepwise mathematical and critical analysis of various features of sort theory, including applications to applicative and logic programming. We show in a companion paper [6] how a consistent formalization of sort theory makes it necessary to use a specific algebraic structure called a double bundle structure.

1. Computing with sets:

When a computation is designed, or when a mathematical problem is solved, one starts from an ill-defined value of the result, belonging to some set. This set is usually specified by a declaration stating where the solution is to be found, and it corresponds to the initial state of uncertainty about the result. The larger the set, the larger the uncertainty. It is, in a way, the initial value of the result. Solving the problem amounts to restricting this set to some subset in order to obtain the final value of the result.

Example 1: Let $a , c \in \underline{Z}$ (where $\underline{Z}$ is the set of all integers)

$$\exists\ ?\ b \in \underline{Z} \qquad a = b * c$$

The result here is b. Its initial "value" is $\underline{Z}$. Its final value is

$$b \in \underline{\text{if}}\ c = 0\ \underline{\text{then}}\ \underline{\text{if}}\ a = 0\ \underline{\text{then}}\ \underline{Z}\ \underline{\text{else}}\ \phi\ \underline{\text{else if}}\ \frac{a}{c} \in \underline{Z}\ \underline{\text{then}}\ \frac{a}{c}\ \underline{\text{else}}\ \phi$$

or in a more compact form:

$$b \in \underline{\text{if}}\ c = 0\ \underline{\text{then}}\ (\underline{\text{if}}\ a = 0\ \underline{\text{then}}\ \underline{Z}\ \underline{\text{else}}\ \phi)\ \underline{\text{else}}\ \left\{\frac{a}{c}\right\} \cap \underline{Z} .$$

This solution is best expressed by using sets. The starting information about b is that b belongs to a "large" set $\underline{Z}$, and the computation consists in restricting this set in order to get more and more information (i.e. less and less uncertainty) about b. The restriction process is performed by using suitable intersections with other sets. The optimal information (from the point of view of most deterministic programming languages) is obtained when the set is reduced to a singleton

(here $b \in \left\{\frac{a}{c}\right\} \subseteq \underline{Z}$), although in general the solution may or may not be unique. The set can also be empty ($\frac{a}{c} \notin \underline{Z}$) and $b \in \left\{\frac{a}{c}\right\} \cap \underline{Z} = \phi$), which corresponds, in the case of programs, to contradictory specifications (overflow, underflow, errors).

In this example the initial and final sets were obvious, but the actual computation leading from the initial set to the final set was not given. Our second example will be a Prolog program, and will show the actual computation. The reader is referred to [4] for details of the Prolog syntax and computation rules.

Example 2: Consider the following Prolog program

1. sum (0 , x , x) ←
2. sum (s(N) , x , s(M)) ← sum (N , x , M)
3. ← sum (s^2(0) , y , z)

The first line asserts that $0 + x = x$, the second line that $N + x = M \Rightarrow s(N) + x = s(M)$ (here $s(\ )$ denote the successor function over natural numbers), and the third line that $\neg (s^2(0) + y = z)$ which amounts to the query

$$\exists\ ?\ y\ ,\ z\quad s^2(0) + y = z$$

A top-down execution of this program performed by the Prolog interpreter runs as follows:

4. ← sum (s(0) , y , m_1) if $z = s(m_1)$ (By lines 2, 3 and Modus Tollens)
5. ← sum (0 , y , m_2) if $m_1 = s(m_2)$ (By lines 2, 4 and Modus Tollens)

6. □ if $y = x$, $m_2 = x$ i.e. $z = s(m_1) = s^2(m_2) = s^2(x)$
(By lines 1, 5)

where □ denotes the contradictory clause. Thus the final result is: all y , z such that $z = y + 2$. If we consider the program (1., 2.) then the Herbrand universe H_u associated with it is the set of all variable free terms

$$H_u = \{0 , s(0) , s^2(0) ,..., s^n(0) ,...\}$$

The Herbrand base H_b is the set of all variable-free atomic formulae:

$$H_b = \{sum (a , b , c) : a , b , c \in H_u\}$$

since we are working with deductions, our universe ("large set") will be H_b and the initial value of the result is the set σ of variable free atomic formulae associated with the query

$$\leftarrow sum (s^2(0) , y , z)$$

namely

$$\sigma = \{sum (s^2(0) , d , e) : d , e \in H_u\}$$

The computation is going to transform this set, and the set transformation associated with the program is defined by

$$T : P(H_b) \rightarrow P(H_b)$$

$$T(S) = \{A' \in H_b \mid A' = A\theta , A \leftarrow B_1 \wedge ... \wedge B_m \text{ is in the program, and } B_1\theta ,..., B_m\theta \in S \text{ for some substitutions } \theta \}$$

Thus $T(S)$ is obtained from S by applying a one-step modus ponens to S, using the clauses contained in the program.

We now successively compute:

$$H_b = \{\text{sum}\ (a\ ,\ b\ ,\ c) : a\ ,\ b\ ,\ c \in H_u\}$$

$$T(H_b) = \{\text{sum}\ (0\ ,\ u\ ,\ u)\ ,\ \text{sum}\ (s(a)\ ,\ u\ ,\ s(c)):$$

$$u\ ,\ a\ ,\ c \in H_u\}$$

$$T^2(H_b) = \{\text{sum}\ (0\ ,\ u\ ,\ u)\ ,\ \text{sum}\ (s(0)\ ,\ u\ ,\ s(u))\ ,$$

$$\text{sum}\ (s^2(a)\ ,\ u\ ,\ s^2(c)) : u\ ,\ a\ ,\ c \in H_u\}$$

$$T^3(H_b) = \{\text{sum}\ (0\ ,\ u\ ,\ u)\ ,\ \text{sum}\ (s(0)\ ,\ u\ ,\ s(u))\ ,$$

$$\text{sum}\ (s^2(0)\ ,\ u\ ,\ s^2(u))\ ,\ \text{sum}\ (s^3(a)\ ,\ u\ ,\ s^3(c)):$$

$$u\ ,\ a\ ,\ c \in H_u\}$$

.....

$$T^n(H_b) = \{\text{sum}\ (0\ ,\ u\ ,\ u)\ ,\ \text{sum}\ (s^k(0)\ ,\ u\ ,\ s^k(u))\ ,$$

$$\text{sum}\ (s^n(a)\ ,\ u\ ,\ s^n(c)) : a\ ,\ u\ ,\ c \in H_u\ ;$$

$$k = 1,\ldots,n-1\}$$

Thus $T^n(H_b)$ excludes all false additions of the form $\text{sum}\ (s^n(0),\ldots)$. Now, as mentioned above, the initial value of the result is σ. We have also

$$\sigma \cap H_b = \sigma = \{sum\ (s^2(0)\ ,\ y\ ,\ z) : y\ ,\ z \in H_u\}$$

$$\sigma \cap T(H_b) = \{sum\ (s^2(0)\ ,\ u\ ,\ s(m)) : u\ ,\ m \in H_u\}$$

$$\sigma \cap T^2(H_b) = \{sum\ (s^2(0)\ ,\ u\ ,\ s^2(m)) : u\ ,\ m \in H_u\}$$

$$\sigma \cap T^3(H_b) = \{sum\ (s^3(0)\ ,\ u\ ,\ s^3(u)) : u \in H_u\}$$

....

$$\sigma \cap T^n(H_b) = \sigma \cap T^3(H_b) \quad \text{if} \quad n > 3 \ .$$

Now comparing the sequence of sets $\{\sigma \cap T^n(H_b)\}_{n \in IN}$, with the execution of the program we notice that:

(i) $\sigma = \sigma \cap T^0(H_b)$ corresponds exactly to the state of the computation in line 3. of the program

(ii) $\sigma \cap T^1(H_b)$ is in the same relation with line 4.

(iii) $\sigma \cap T^2(H_b)$ is in the same relation with line 5.

(iv) $\sigma \cap T^3(H_b)$ is in the same relation with line 6.

Thus the decreasing sequence of sets $\sigma \cap T^n(H_b)\}_{n \in IN}$ describes exactly the operational behaviour of the program during its execution by the top-down resolution method.

By comparison, a Scott-Strachey-like approach (bottom-up computation) would give:

$T(\phi) = \{sum\ (0\ ,\ u\ ,\ u) : u \in H_u\}$

$T^2(\phi) = \{sum\ (0\ ,\ u\ ,\ u)\ ,\ sum\ (s(0)\ ,\ u\ ,\ s(u)) : u \in H_u\}$

....

$T^n(\phi) = \{sum(0\ ,\ u\ ,\ u)\ ,\ldots,\ sum\ (s^{n-1}(0)\ ,\ u\ ,\ s^{n-1}(u)) : u \in H_u\}$

and the sequence $\{\sigma \cap T^n(\phi)\}_{n\in IN}$ has values

$\sigma \cap T^0(\phi) = \phi$

$\sigma \cap T(\phi) = \phi$

$\sigma \cap T^2(\phi) = \phi$

$\sigma \cap T^3(\phi) = \{sum\ (s^2(0)\ ,\ u\ ,\ s^2(u)) : u \in H_u\}$

$= \sigma \cap T^p(\phi) \quad \forall p \geq 3$

This sequence converges to the same limit but does not give as much information about the intermediate states of computation as $\{\sigma \cap T^n(H_b)\}_{n\in IN}$ if the Prolog program is executed as described in lines 4. through 6. above.

To further illustrate this approach, we give another Prolog example, which yields several results.

Example 3:

Consider the following Prolog program (lines 1, 2, 3) together with its execution. (MT is an abbreviation for Modus Tollens):

1. sum (0 , x , x) ←

2. sum (s(N) , x , s(M)) ← sum (N , x , M)

3. ← sum (x , y , $s^3(0)$)

4. □ x = 0 , y = $s^3(0)$ (1,3)

5. ← sum (x' , y , $s^2(0)$) , x = s(x') (2,4 MT)

6. □ x' = 0 y = $s^2(0)$ (x = s(0) (1,5)

7. ← sum (x" , y , s(0)) , x' = s(x") (2,5 MT)

8. □ x" = 0 y = s(0) (x = $s^2(0)$)

9. ← sum (x''' , y , 0) , x" = s(x''') (2,7 MT)

10. □ x''' = 0 y = 0 x = $s^3(0)$

It gives all the decompositions of number 3 into two numbers.

Here the Herbrand universe, the Herbrand base and the transformation T are the same as for example 2. The initial value of the result is now

$$\sigma = \{\text{sum } (x , y , s^3(0)) : x , y \in H_u\}$$

We do not need to recompute the $T^n(H_b)$, since they are the same as in example 2. The sequence $\{\sigma \cap T^n(H_b)\}_{n\in IN}$ yields:

$$\sigma \cap H_b = \{\text{sum } (x , y , s^3(0)) : x , y \in H_u\}$$

$$\sigma \cap T(H_b) = \{\text{sum } (0 , s^3(0) , s^3(0)) , \text{sum } (s(a) , b , s^3(0)) : a , b \in H_u\}$$

(this gives the result of line 4 and an anticipation of the other results)

$$\sigma \cap T^2(H_b) = \{\text{sum}\ (0\ ,\ s^3(0)\ ,\ s^3(0))\ ,\ \text{sum}\ (s(0)\ ,\ s^2(0)\ ,\ s^3(0))\ ,$$

$$\text{sum}\ (s^2(a)\ ,\ b\ ,\ s^2(0)) : a\ ,\ b \in H_u\}$$

(see line 6)

$$\sigma \cap T^3(H_b) = \{\text{sum}\ (0\ ,\ s^3(0)\ ,\ s^3(0))\ ,\ \text{sum}\ (s(0)\ ,\ s^2(0)\ ,\ s^3(0))\ ,$$

$$\text{sum}\ (s^2(0)\ ,\ s(0)\ ,\ s^3(0))\ ,\ \text{sum}\ (s^3(a)\ ,\ b\ ,\ s^3(0)) :$$

$$a\ ,\ b \in H_u\}$$

(compare with line 8)

$$\sigma \cap T^4(H_b) = \{\text{sum}\ (0\ ,\ s^3(0)\ ,\ s^3(0))\ ,\ \text{sum}\ (s(0)\ ,\ s^2(0)\ ,\ s^3(0))\ ,$$

$$\text{sum}\ (s^2(0)\ ,\ s(0)\ ,\ s^3(0))\ ,\ \text{sum}\ (s^3(0)\ ,\ 0\ ,\ s^3(0))\}$$

(compare with line 10)

As a comparison with the bottom-up approach we have:

$$\sigma \cap \phi = \phi =$$

$$\sigma \cap T(\phi) = \{\text{sum}\ (0\ ,\ s^3(0)\ ,\ s^3(0))\}$$

$$\sigma \cap T^2(\phi) = \{\text{sum}\ (0\ ,\ s^3(0)\ ,\ s^3(0))\ ,\ \text{sum}\ (s(0)\ ,\ s^2(0)\ ,\ s^3(0))\}$$

$$\sigma \cap T^3(\phi) = \{\text{sum}\ (0\ ,\ s^3(0)\ ,\ s^3(0)\ ,\ \text{sum}\ (s(0)\ ,\ s^2(0)\ ,\ s^3(0))\}\ ,$$

$$\text{sum}\ (s^2(0)\ ,\ s(0)\ ,\ s^3(0))\}$$

$$\sigma \cap T^4(\phi) = \{\text{sum}\ (0\ ,\ s^3(0)\ ,\ s^3(0)\ ,\ \text{sum}\ (s(0)\ ,\ s^2(0)\ ,\ s^3(0))\ ,$$

$$\text{sum}\ (s^2(0)\ ,\ s(0)\ ,\ s^3(0))\ ,\ \text{sum}\ (s^3(0)\ ,\ 0\ ,\ s^3(0))\}\ .$$

Thus it appears that $\{\sigma \cap T^n(\phi)\}_{n \in IN}$ records only the finite, successful computations once they have succeeded. It does not give any information about unfinished, current computations, whether these may eventually succeed or not. In particular it gives no information about the infinite computations. Indeed in Scott's theory the latter are all denoted by the undefined element $\perp$.

This approach applies to applicative programs as well, as we illustrate in the following examples.

Example 4:

Consider the program over the natural numbers:

$$F(n) \Leftarrow \underline{\text{if}}\ n = 0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ n * F(n - 1)\ \underline{\text{fi}}$$

If we define the transform T by

$$T = \lambda g \cdot \lambda n \cdot \underline{\text{if}}\ n = 0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ n * g(n - 1)\ \underline{\text{fi}}$$

then this can be rewritten as the functional fixpoint equation

$$F = T(F)$$

The universe ("large set") is here the set of functions:

$$H = \{f : IN \to IN\}$$

Transform T canonically maps $P(H)$ into $P(H)$ by:

$$T(S) = \{T(s) : s \in S \subseteq H\}$$

We now successively compute:

$H = \{h : IN \to IN\}$

$T(H) = \{h' \in H : h'(0) = 1 \ , \ h'(p+1) = (p+1) * h(p) \ , \ h \in H \ , \ p \in IN\}$

$T^2(H) = \{h'' \in H : h''(0) = 1 \ , \ h''(p+1) = (p+1) * h'(p) \ , \ h' \in T(H) \ , \ p \in IN\}$

.....

$$T^n(H) = \{h^{(n)} \in H : h^{(n)}(0) = 1 \ , \ h^{(n)}(p+1) = (p+1) * h^{(n-1)}(p) \ , \ h^{(n-1)} \in T^{n-1}(H) \ , \ p \in IN\} \ .$$

The sequence of sets $\{T^n(H)\}_{n \in IN}$ is strictly decreasing, and is constituted by the successive approximations of the set of solutions of the fixpoint equation $F = T(F)$. Its limit $\bigcap_{m \in IN} T^n(H)$ constains a single element, which is the factorial function, as may easily be seen.

The connection here with the Prolog examples is as follows. Consider the function call (= query) $F(4)$. Then what we must consider here is the sequence of sets $\{T^n(H)(4)\}_{n \in IN}$, where

$$T^n(H)(4) = \{h(4) : h \in T^n(H)\}$$

Thus we have:

$$H(4) = \{h(4) \mid h \in IN \to IN\} = IN$$

$$T(H)(4) = 4 \cdot IN = \{4\ p : p \in IN\}$$

$$T^2(H)(4) = 4 \cdot 3 \cdot IN = 12 \cdot IN$$

$$T^3(H)(4) = 4 \cdot 3 \cdot 2 \cdot IN = 24 \cdot IN$$

$$T^4(H)(4) = 4 \cdot 3 \cdot 2 \cdot 1 \cdot IN = 24 \cdot IN$$

$$T^5(H)(4) = \{24\}$$

$$T^{5+n}(H)(4) = \{24\} \quad \text{for any} \quad n \in IN \quad .$$

Notice that here again we have a desciption of the operational behaviour of the call F(4) :

F(4) (see H(4))

$\rightarrow 4 * F(3)$ (see $T(H)(4)$)

$\rightarrow 4 * (3 * F(2))$ (see $T^2(H)(4)$)

$\rightarrow 4 * (3 * (2 * F(1)))$ (see $T^3(H)(4)$)

$\rightarrow 4 * (3 * (2 * (1 * F(0))))$ (see $T^4(H)(4)$)

$\rightarrow 4 * 3 * 2 * 1 * 1$ (see $T^5(H)(4)$) .

Example 5:

Consider the applicative program:

$$F(n) \Leftarrow \underline{\text{if}}\ n = 0\ \underline{\text{then}}\ 1\ \text{else}\ F(n+1) \div (n+1)\ \underline{\text{fi}}$$

Here we have

$$T = \lambda g \cdot \lambda n\ \underline{\text{if}}\ n = 0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ \frac{g(n+1)}{(n+1)}\ \underline{\text{fi}}$$

and we take $H = [IN \rightarrow IN] = \{h : IN \rightarrow IN\}$ - The sequence $\{T^n(H)\}_{n \in IN}$ yields:

$H = \{h : IN \rightarrow IN\}$

$T(H) = \{h' : h'(0) = 1 \ ; \ h'(p+1) = \frac{h(p+2)}{p+2} \ , (h \in H) \ , \ p \in IN\}$

$$T^2(H) = \{h'' : h''(0) = 1 \ ; \ h''(p+1) = \frac{h'(p+2)}{p+2} \ , \ h' \in T(H), p \in \mathbb{N}\}$$

$$T^n(H) = \{h^{(n)} : h^{(n)}(0) = 1 \ , \ h^{(n)}(p+1) = \frac{h^{(n-1)}(p+2)}{(p+2)} \ , \ h^{(n-1)} \in T^{n-1}(H), p \in \mathbb{N}\}$$

The limit $\bigcap_n T^n(H)$ of this sequence is then

$$\bigcap_n T^n(H) = \{h : \mathbb{N} \to \mathbb{N} \mid h(0) = 1 \ , \ h(n+1) = a \cdot (n+1)! \ , \ a \in \mathbb{N}\}$$

It has infinitely many elements, and among those we have the factorial function. This is exactly the set of solutions of the functional equation defined by the program.

2. Data types as objects; normal functions:

We have been computing with sets until now. Intuitively sets correspond to data types. So we can already outline a semantic domain where data types would be objects like any other objects. This corresponds to the usual way typed programming languages are compiled, where types are attributes, as much as values are attributes of computation objects [1]. Besides there are at least three reasons to prefer such a type-system to those based on Church's theory of types [3] ([14]):

(i) a data object may have several types (intersection of sets), and a type may be a subtype of another (inclusion)

(ii) polymorphism is the rule rather than the exception: a function can have many types

(iii) types and data objects would belong to the same domain, and we could have some useful equalities such as:

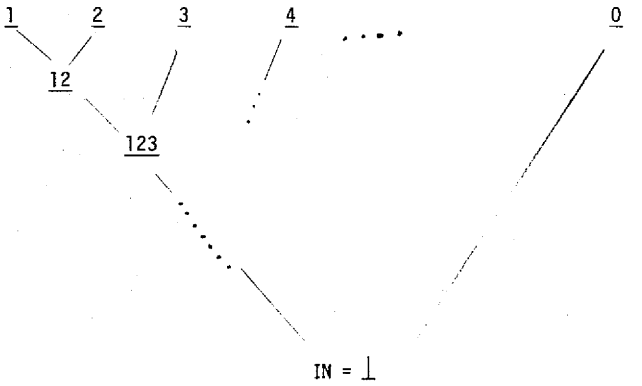
$$\underline{\text{eveninteger}} + 1 = \underline{\text{oddinteger}}$$

which make no sense in a Church system.

Carrying out such a scheme in the usual denotational semantics [15], which uses complete partial orders (c.p.o.) and continuous functions as its basic tools, leads to some serious difficulties.

We recall that a c.p.o. is a partially ordered set having a least element ($\perp$) , and where every increasing chain has a least upper bound. The least element denotes the least possible amount of information, i.e. the highest degree of uncertainty. If we compute with sets as in §1, this corresponds to the largest possible set, and the ordering is the inverse inclusion.

Consider a programming language L for computing with natural numbers : 0,1,2,... . Assume that in L numbers are either values, or indices in one-dimensional arrays. No index is 0 , and indices can only range over segments $[1,\ldots,n]$, $n \in \mathbb{N}$. Thus L has ground data types IN , $\underline{1}$, $\underline{12}$, $\underline{123}$,..., $\underline{12\ldots n}$, $n \in \mathbb{N} \setminus \{0\}$ corresponding to the fact that some numbers lies in IN , {1,2} , etc. Since IN is the largest set (= bottom element), we obtain the following c.p.o. where the objects and data types are incorporated together in a single unified domain:



This c.p.o. reflects faithfully our information-ordering. Now consider the program

$$F(n) \Leftarrow \underline{\text{if}}\ \ n < 2\ \ \underline{\text{then}}\ \ n\ \ \underline{\text{else}}\ \ F(n-1) + n\ \ \underline{\text{fi}}$$

According to [15], the function f computed by this program is the least fixpoint of the functional it defines. Now let us do some type checking.

$f(1) = 1$, $f(2) = 3$, so we guess $f(\underline{12}) \sqsubseteq \underline{123}$

Now:

$$f(\underline{12}) = \underline{\text{if}}\ \ \underline{12} < 2\ \ \underline{\text{then}}\ \ \underline{12}\ \ \underline{\text{else}}\ \ f(\underline{12} - 1) + \underline{12}$$

Taking the canonical extensions of $+$, $-$, $<$ we get:

$$\underline{12} < 2 = 1 < 2 \sqcap 2 < 2 = \underline{\text{true}} \sqcap \underline{\text{false}} = \underline{\text{boolean}}$$

$$\underline{12} - 1 = 1 - 1 \sqcap 2 - 1 = 0 \sqcap 1 = \text{IN} = \perp$$

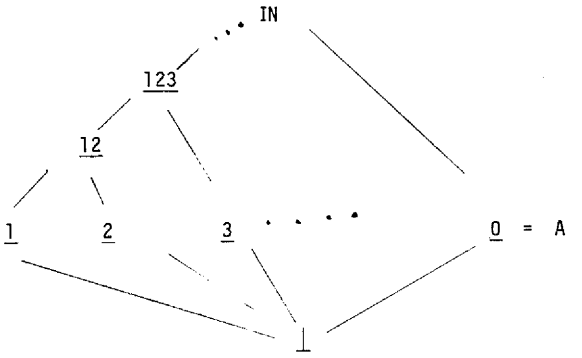
according to our information ordering.

Therefore

$$f(\underline{12}) = \underline{12} \sqcap (f(\underline{12} - 1) + \underline{12}) = \underline{12} \sqcap (\perp + \underline{12}) = \perp$$

Thus the least fixpoint f , evaluated at any type $\underline{12...n}$, always yields $\perp$ and gives no useful information for type checking purposes. The reason is that here the least fixpoint operates in a threshold manner, giving expected values at $1,2,3,...$ and $\perp$ elsewhere, thus ignoring completely the process of approximating sets through intersections of supersets described in §1.

Shamir and Wadge [14] patch the construction by putting the c.p.o. upside-down, and adding a new bottom element. The following complete lattice is then obtained; it will be called A in the sequel



The consequence of this is that the bottom element does not correspond any more to the least amount of information, and would rather correspond to the empty set, i.e. the intersection of contradictory information. (Compare with $\{T^n(\phi)\}_{n\in IN}$ and $\{T^n(H_b)\}_{n\in IN}$ in the Prolog examples).

In [14] the authors take as functions the least extensions of continuous functions defined over the sub c.p.o. $IN_\perp = IN \cap \{\perp\}$ (called tight functions). However continuous functions over $IN_\perp$ are just the monotonic ones, and there exist functions that are continuous over $IN_\perp$ and have no continuous least extension to the whole domain. As an example:

$$f = \lambda x \in IN_\perp \,.\, \underline{\text{if}}\ x \in \{0,\perp\}\ \underline{\text{then}}\ x\ \underline{\text{else}}\ 1$$

is computable and continuous over $IN_\perp$. Its least extension $\bar{f}$ is defined by:

$$\bar{f}(\underline{1\ldots n}) = f(1) \sqcup \ldots \sqcup f(n) = 1 \quad \forall\, n \in IN^+$$

$$\bar{f}(N) = f(0) \sqcup \bar{f}(\underline{12}) \sqcup \ldots \sqcup f(\underline{1\ldots n}) \sqcup \ldots = 0 \sqcup 1 = IN$$

and is discontinuous at point IN, since $\{\underline{1\ldots n} : n \in IN^+\}$ is an ascending chain converging to IN, and

$$\bar{f}(IN) \neq \sqcup \{\bar{f}(\underline{1\ldots n}) : n \in IN^+\} = \sqcup \{1 : n \in IN^+\} = 1$$

Another difficulty comes from programs containing conditionals. Consider the program [14]:

$$F(n) \Leftarrow \underline{\text{if}}\ \ n = 0\ \ \underline{\text{then}}\ \ n\ \ \underline{\text{else}}\ \ 0$$

If f is the least fixpoint then it seems that $f(IN) \sqsubseteq \underline{0}$, but this cannot be proved in the system since

$$f(IN) = \underline{\text{if}}\ \ IN = \underline{0}\ \ \underline{\text{then}}\ \ IN\ \ \underline{\text{else}}\ \ \underline{0}$$

$$= IN \sqcup \underline{0} = IN$$

The problem is in this step where no use is made of the test in the conditional. Here one may also follow [14] in introducing an extra type $\underline{nzi} = \{1,2,3,\ldots\}$ (non zero integer), and since $\underline{nzi} = 0$ is false we get the desired result:

$$f(IN) = f(0 \sqcup \underline{nzi}) = f(0) \sqcap f(\underline{nzi}) =$$

$$= (\underline{\text{if}}\ \ 0 = 0\ \ \underline{\text{then}}\ \ 0\ \ \underline{\text{else}}\ \ 0) \sqcup$$

$$(\underline{\text{if}}\ \ \underline{nzi} = 0\ \ \underline{\text{then}}\ \ \underline{nzi}\ \ \underline{\text{else}}\ \ 0) =$$

$$= 0 \sqcup 0 = 0$$

But this partition $IN = 0 \sqcup \underline{nzi}$ introduces a type which is outside of the domain, and has no theoretical justification in the construction. Thus, if the theory is to reflect reality, it appears that the regularity property which is needed here for our functions is not plain Scott-continuity, but must also take care of bizarre partitions such as $IN = \underline{1} \cup \underline{nzi}$. In other words we need alternative notions to continuous functions and cpo's. This leads to the concepts of normal function and domain, which were introduced in [5] in an algebraic setting(*).

We are computing with sets. So let T be a "large" set (total set), and $X \subseteq P(T)$. An element $x \in X$ is atomic iff

$$\forall \{y_i\}_{i \in I} \subseteq X \quad x = \bigcup_i y_i \Rightarrow \exists\, i \quad x = y_i$$

Intuitively an atomic element cannot be decomposed as the union of "smaller" elements for type checking purposes.

We say that X is a domain iff

(i) ϕ , $T \in X$

(ii) $\forall\, x \in X \quad x = \cup\{y \in X : y \text{ atomic} \subseteq x\}$

(iii) X is closed through intersection.

The motivation for our definition of domain is to abstract away the algebraic structure which we have seen at work in the examples discussed, in particular for the conditional if then else problem.

A function $f : X \to X$ is normal, iff

$$\forall\, x \in X \quad f(x) = \bar{\cup}\{f(y) : y \text{ atomic} \subseteq x\}$$

(*) These concepts were implicit in Nolin's work, but were first studied as such in [5].

where $\bar{\cup}$ denotes the least upper bound in the complete lattice X.

One easily verifies that normal functions are monotonic.

Now if the set A is as above, then $A \subseteq P(IN)$, $IN = T$, $\phi = \perp$ and A has ϕ, $\underline{0} = \{0\}$, $\underline{1} = \{1\}$,.... as its sets of atomic elements. And we easily verify that it is a domain.

Shifting the algebraic structure of A from a cpo to that of a domain, and considering only normal functions, solves all the problems we have met so far in handling data types as objects in the cpo/continuous function framework defined in [14]. In particular, the if then else does not appear any more like a teratological case which needs special care, but fits smoothly in this new general framework. All what is needed is the normal extension of McCarthy conditional to non atomic elements [*] pp. 168, and regular composition of functions [*] pp. 125.

The advance which is made here above [14], besides the solution of the technical difficulties encountered in [14], is a conceptual one. Indeed it appears that the algebraic structure we need in the present case is nothing but a particular case of another more general structure, which does appear everywhere in semantics and which is called a bundle [*].

In cpo theory, the function computed by a program is the least fixpoint of some functional, and computations are made upwards, as Cadiou theorem shows ([5],). But shifting from cpo's to domains leads to another important idea: since computations are made downwards through intersections, the function computed by a program is the greatest fixpoint of the functional it defines [5]. This was clear in the Prolog and applicative programs examples given in section I of this paper;

[*] A. Nait Abdallah, Faisceoux et sémantique des programmes, Thèse Etat Paris, 1980.

one may also check that this is true also for the applicative programs defined above. It may happen that the greatest fixpoint and the least fixpoint coincide, but this is not always the case.

3. Functional types and their intersections:

A further step in the theory is to take into account the fact that procedures can also be typed; for example

integer procedure p(x) integer x
    begin ... end

gives the set of functions where the value of the function f_p computed by p must be found:

$$f_p \in \text{FIN IN} = \{g \text{ computable} \mid g(\text{IN}) \subseteq \text{IN}\}$$

This is in a way, the "initial set" of the computation described by the body of p (cf. §1). This initial set can be sometimes given more precisely; for example:

(even → odd) and (odd → even) procedure p(x)
    begin ... end

means that

$$f_p \in (\text{F even odd}) \cap (\text{F odd even})$$

i.e.

$$f_p \in \{g \text{ computable} \mid \forall\, n \in \text{IN}\ g(2n) = 2p + 1 \text{ for } p \in \text{IN}\}$$

$$\cap \{g \text{ computable} \mid \forall\, n \in \text{IN}\ g(2n+1) = 2p \text{ for } p \in \text{IN}\}$$

This set is obviously smaller than the first one, and thus gives more information about f_p .

A limit case is the specification of an array, as a notion of a set is canonically associated with any array definition. Indeed, if

$$Fab = \{g \text{ computable} \mid g(a) \subseteq b\}$$

where a , b are sets of objects, then the set intersection

$$\underline{t} = F\{1\}\{a_1\} \cap F\{2\}\{a_2\} \cap \ldots \cap F\{8\}\{a_8\}$$

defines an eight element "array", containing a_i in store i . We will show that the evaluation of $\underline{t}$ at {j} yields:

$$\underline{t}\,[\{j\}] = \{a_i\} \text{ if } i = j \text{ , } T \text{ otherwise}$$

where T is the total set (= maximum amount of uncertainty.)

We note that this definition of array $\underline{t}$ is completely independent of any implementation or description in some language. One may as well say that it is a function, whose domain is a segment of integers.

If $\underline{t'} = \underline{t} \cap F\{9\}\{a_9\}$, then $\underline{t'}$ is more defined than $\underline{t}$ since it is a smaller set, and its domain is larger. This shows how functions computed by programs, or represented by arrays, are approximated in a natural way by sets of the form

$$F\,\underline{a}\,\underline{b} = \{g \text{ computable} \mid g(\underline{a}) \subseteq \underline{b}\}$$

where $g(\underline{a}) = \{g(\alpha) : \alpha \in \underline{a}\}$. Types are thus fundamental objects in the construction, and do not appear only later on as retracts, as in [13].

The definition of evaluation is closely related to the definition of $F\ \underline{a}\ \underline{b}$. In fact we have the following deduction

1. $f \in F\ \underline{a}\ \underline{b}$ (i.e. f is of type $F\ \underline{a}\ \underline{b}$)
2. $x \in \underline{c}$
3. $f(x) \in \underline{b}$ if $\underline{c} \subseteq \underline{a}$ by 1 , 2 .

{no information available about $f(x)$ if $\underline{c} \not\subseteq \underline{a}$.

This leads to the following definition of evaluation

$$(F\ \underline{a}\ \underline{b})\ [c] = \text{if } \underline{c} \subseteq \underline{a} \text{ then } \underline{b} \text{ else } T$$

where T denotes the least available amount of information (= total set).

If A is any intersection of $F\ \underline{a}\ \underline{b}$'s , this generalizes to

$$A[c] = \cap\{(F\ a\ b)[c] : A \subseteq F\ a\ b\} =$$

$$\cap\ \{b : A \subseteq F\ c\ b\}$$

because $$(F\ a\ b)[c] = b \iff c \subseteq a \text{ or } b = T\ .$$

4. Sort collections:

The next stage is to have a functionally closed space, containing objects and data types of any level of functionality. This is the most complicated part of sort theory.

The objects of the theory are types, or sets, like those we have met so far, and will be called sorts. We have used two operations on these objects:

(i) $F : x\ ,\ y \to \{f \ldots \mid f(x) \subseteq y\}$

(ii) intersection (possibly infinite)

F in this setting is defined later on. The aim here is to define a space which contains these objects as elements and is closed for these operations. The space is called a sort collection.(*) We give the technical details first, and a few comments afterwards.

The basic notion of the theory is that of a collection, its intuitive meaning being the space we have in mind and that we want to define.

Let T be a non-empty collection. Let P(T) be the power-collection of T (i.e. the collection of all subcollections), and $A \subseteq P(T)$ such that T, $\phi \in A$ and A is closed through infinite intersection.

An element $x \in A$ is atomic iff

$$\forall \{x_i\}_{i \in I} \subseteq A \quad x = \bigcup_i x_i \Rightarrow \exists\, i \quad x = x_i \quad .$$

If $\{x_i\}_{i \in I} \subseteq A$ then the completed union of this family is its least upper bound for the complete lattice structure of A, i.e.

$$\bar{\bigcup}_i x_i = \bigcap \{z \in A : \forall\, i \in I \quad x_i \subseteq z\}$$

A function $f : A \to A$ is normal iff

$$\forall\, x \in A \quad f(x) = \bar{\bigcup} \{f(y) : y \text{ atomic} \subseteq x\}$$

All these definitions we have met earlier in §2 for domains. What is new is the setting for obtaining the functional closure and which is as follows:

(*) "collection d'algorithmes" in [8].

Define

(i) $\forall\, x,y \in A \quad Fxy = \{f : A \to A \text{ normal} \mid f(x) \subseteq y\}$

(ii) A_F is the smallest collection containing all the fxy and closed by infinite intersection.

Definition of sort collections: [7]

If T is a non-empty collection, a subcollection $A \subseteq P(T)$ is sort collection if and only if it verifies the four conditions:

(i) there exists $\mathcal{B} \subseteq P(T)$ such that ϕ , $T \in \mathcal{B}$ and such that A is the closure of $\mathcal{B}$ under infinite intersection and under F .

(ii) $F \times T = T$

(iii) if $A_{\mathcal{B}}$ is the closure of $\mathcal{B}$ under infinite intersection then $\forall\, x \in A_{\mathcal{B}} \setminus \{T,\phi\}$, $\forall\, y \in A_F$, and x and y are incomparable for the inclusion relation

(iv) $\forall\, x \in A \quad x = \cup \{y : y \text{ atomic} \subseteq x\}$

(v) T is atomic. □

Another important definition is the evaluation, which generalizes the one given in §3 , and proceeds from the same idea:

$$\forall\, x,y \in A \quad x[y] = \cap \{z \in A : x \subseteq Fy\ z\}$$

We now give one theorem and the proof of this theorem in order to show how the theory works.

Theorem [8]: If $u = \bigcup_{j \in J} z_j \in A$, $I \neq \phi$

then

$$(\bigcap_{i \in I} Fx_i y_i)[u] = \bigcup_{j \in J} ((\bigcap_{i \in I} Fx_i y_i)[z_j])$$ □

We use the following lemmae in the proof of this theorem

Lemma 1: $Fxy \subseteq Fx_1y_1 \iff$ either $y_1 = T$ or $x_1 \subseteq x \wedge y \subseteq y_1$

Proof:

$\Rightarrow$: let $f = \lambda u$. if $z \subseteq x$ then y else T .

Then f is normal and $f \in Fxy \subseteq Fx_1y_1$.

Thus $f(x_1) =$ if $x_1 \subseteq x$ then y else $T \subseteq y_1$

i.e. either $y_1 = T$ or ($x_1 \subseteq x$ and $y \subseteq y_1$)

$\Leftarrow$: If $y_1 = T$, then

$\forall f \in Fxy$ $f(x) \subseteq y \subseteq T$ and $f(x_1) \subseteq T$ since T is the largest element,

i.e. $Fxy \subseteq Fx_1y_1$. If $x_1 \subseteq x$ and $y \subseteq y_1$, then

$$f(x_1) \subseteq f(x) \subseteq y \subseteq y_1$$

by monotonicity of normal functions. Thus

$$\forall f \quad f(x) \subseteq y \Rightarrow f(x_1) \subseteq y_1 \quad \text{i.e.}$$

$$Fxy \subseteq Fx_1y_1 \quad .$$ □

Lemma 2: (i) $\bigcap_i Fxy_i = Fx\,(\bigcap_i y_i)$

(ii) $F\,(\bigcup_i x_i)y = \bigcap_i Fx_iy$ □

proof: (i) $f \in \bigcap_i Fxy_i$: $\iff \forall i \quad f \in Fxy_i \iff$

$\forall i \; f(x) \subseteq y_i \iff f(x) \subseteq \bigcap_i y_i \iff f \in Fx\,(\bigcap_i y_i)$

(ii) $f \in F\,(\bigcup_i x_i)y \iff f\,(\bigcup_i x_i) \subseteq y \iff$

(f is normal) $\tilde{\bigcup}_i f(x_i \subseteq y \iff \forall i \quad f(x_i) \subseteq y$

$\iff f \in \bigcap_i Fx_iy$. □

Lemma 3:

$$(Fxy)\,[z] = \underline{\text{if}}\ \ z \subseteq x\ \ \underline{\text{then}}\ y\ \underline{\text{else}}\ T$$

proof: $(F\,xy)\,[z] = \cap\,\{u : F\,xy \subseteq F\,zu\}$ = (lemma 1)

$$\cap\,\{u : (u=T)\ \ \text{or}\ \ (z \subseteq x\ \ \text{and}\ \ y \subseteq u)\} =$$

$$\underline{\text{if}}\ \ z \subseteq x\ \ \underline{\text{then}}\ \ y\ \ \text{else}\ \ T\ .$$

Lemma 4: The operation

$$(u,v) \to u[v]$$

in monotonic in u and v .

proof:

$$u' \subseteq u \Rightarrow u'[v] = \cap\,\{b : u' \subseteq Fbv\} \subseteq$$

$$\cap\,\{b : u \subseteq Fbv\} = u[v]$$

We also have:

$$v' \subseteq v \Rightarrow u[v'] = \cap\,\{b : u \subseteq Fv'b\}$$

since, by Lemma 1, $v' \subseteq v \Rightarrow Fvb \subseteq Fv'b$

$$u[v'] \subseteq \cap\,\{b : u \subseteq Fvb\} = u[v]$$ □

Lemma 5:

$$x[\bigcup_i y_i] = \bar{\bigcup_i}\ x[y_i]$$

proof:

$$x[\bigcup_i y_i] = \cap\{b : x \subseteq F\,(\bigcup_i y_i)b\} =$$

$$(\text{lemma 2}) = \cap\,\{b : x \subseteq \bigcap_i Fy_i\ b\} =$$

$$\cap\,\{b :\ \forall i\ x \subseteq Fy_i\ b\} = \cap\,\{b :\ \forall i\ x[y_i] \subseteq b\}$$

$= \bar{\bigcup}_i\ x[y_i]$ by definition of the completed union. □

Lemma 6: $\bigcap_i Fx_iy_i \subseteq F(\bigcap_i x_i)\ (\bigcap_i y_i)$

proof: $\bigcap_i Fx_i\ y_i \subseteq \bigcap_i F\ (\bigcap_i x_i)y_i$ by lemma 1, which is equal to $F\ (\bigcap_i x_i)\ (\bigcap_i y_i)$ by lemma 2. □

Lemma 7: Let $v \in A$ be atomic. Then we have $\bigcap_{i\in I} Fx_iy_i \subseteq Fvw$ iff either $w = T$ or $\exists\ J \neq \phi\ \ J \subseteq I$, $v \subseteq \bigcap_{j\in J} x_j$ and $\bigcap_{j\in J} y_j \subseteq w$. □

proof: Let $v \in A$ atomic. Assume $\bigcap_{i\in I} Fx_iy_i \subseteq Fvw$. Define the normal function

$$f(a) = \cap\,\{y_i : a \subseteq x_i\} \quad \text{if } a \text{ is atomic}$$

$$= \bar{\cup}\,\{f(b) : b \text{ atomic} \subseteq a\} \quad \text{otherwise}$$

The evaluation of $f(v)$ gives two cases:

1st case: $\exists\ J \subset I\ \ v \subseteq \bigcap_{j\in J} x_j$. Take J maximal; then $f(v) = \bigcap_{j\in J} y_j \subseteq w$

2nd case: $\forall\ i \in I\ \ v \not\subseteq x_i$, then $f(v) = T \subseteq w$ i.e. $w = T$.

Conversely:

$$w = T \Rightarrow \bigcap_i Fx_iy_i \subseteq Fvw = FvT \quad \text{by lemma 1}$$

$$\exists\, J \subseteq I \;\; v \subseteq \bigcap_{j\in J} x_j \quad \text{and} \quad \bigcap_{j\in J} y_j \subseteq w \Rightarrow$$

$$\bigcap_i Fx_iy_i \subseteq F(\bigcap_i x_i)\,(\bigcap_i y_i) \quad \text{by lemma 6, which}$$

is included in Fvw by lemma 1 . □

proof of the theorem: Suppose u is atomic; then

$$(\bigcap_{i\in I} Fx_iy_i)\,[u] = \text{(definition)} \cap \{w \in A : \bigcap_{i\in I} Fx_iy_i \subseteq Fuw\}$$

$$= \text{(lemma 7)} =$$

$$\cap \{w : \exists\, J \neq \phi\,,\; J \subset I \; u \subseteq \bigcap_{j\in J} x_j\,,\; \bigcap_{j\in J} y_j \subseteq w\} =$$

$$\cap \{w : u \subseteq x_i \;\text{ and }\; y_i \subseteq w\} = \text{(lemma 1)}$$

$$\cap \{w : Fx_iy_i \subseteq Fuw\} = \text{(lemma 3)} \bigcap_i (Fx_iy_i[v])$$

Now if $u = \bigcup_{j\in J} z_j \in A$, z_j atomic,

$$(\bigcap_i Fx_iy_i)\,[\bigcup_{j\in J} z_j] = \text{(lemma 5)} \bar{\bigcup_j}\,((\bigcap_i Fx_iy_i)\,[z_j])$$

$$= \bar{\bigcup_{j\in J}} \bigcap_{i\in I} (Fx_iy_i[z_j]) \quad \text{(by the preceding argument.)}$$

$$= \bar{\bigcup_{j\in J}} \bigcap_{i\in I} \{y_i : z_j \subseteq x_i\} \;. \qquad \square$$

The definition of sort collections states a fixpoint equation

$$A = A_B + A_F$$

with equality, whose solution is implicitly assumed to be given. The notion of collection is not formally defined, and Nolin writes about this:

"At first sight at the very least, my "sets", except for the ground sets (*), cannot be described as such in any known set theory." ("A premiere vue tout au moins, mes "ensembles", à l'exception des ensembles de base (*), ne peuvent être qualifiés tels dans aucune théorie des ensembles connue" ([10] p. 267). An examination of the above definitions and the proof of the theorem shows that, explicitly or implicitly the following axioms from the Zermelo-Fraenkel set theory (ZF) are assumed to apply to collections:

1. Extensionality axiom:

$$\forall\, x,y\ (x = y \leftrightarrow \forall\, z\ (z \in x \leftrightarrow z \in y))$$

i.e. two collections are equal iff they have the same elements. This is applied in the proof of Lemmae 1, 2, etc...

2. Null set axiom:

$$\exists\, x\ \forall\, y\ \neg(y \in x)$$

i.e. there exists an empty collection. This appears in the definition of algorithm collections.

3. Pairing axiom:

$$\forall\, xy\ \exists\, z\ \forall u\,(u \in z \leftrightarrow u = x \vee u = y)$$

z is denoted $\{x,y\}$ and its uniqueness can be shown by using the extensionality axiom. Using this axiom we can define ordered pairs

$$\langle x,y\rangle = \{\{x\}, \{\{x,y\}\} \qquad (*)$$

Relations are defined from ordered pairs by: A is a relation iff

(*) i.e. elements of A_B

$\forall\, x(x \in A \to (\exists\, y)\ (\exists\, z)\ (x = \langle y,z\rangle)$ and functions by:

f is a function iff f is a relation and

$$\forall\, x\ \forall\, y\ \forall\, z\ (\langle x,y\rangle \in f \wedge \langle x,z\rangle \in f \to y = z)$$

Without something like (*), it is impossible to develop a theory of relations or normal functions unless the notion of ordered pairs is taken as primitive.

4. Union axiom

$$\forall\, x\ \exists\, y\ \forall\, z\ (z \in y \leftrightarrow \exists\, w\ (z \in w \wedge w \in x))$$

i.e. if x is a collection, so is $\cup x$. This appears in the definition of atomicity and in the statement of the theorem.

5. Powerset axiom:

$$\forall\, x\ \exists\, y\ \forall\, z\ (z \in y \leftrightarrow \forall\, w\ (w \in z \to w \in x))$$

i.e. if x is a collection, so is $P(x)$. This appears in the definition of $A \subseteq P(T)$.

6. For computation purposes, we need the natural numbers, i.e. the finite numbers. For obtaining cardinal numbers in ZF , there are at least two ways:

(i) introduce a special axiom for cardinal numbers, which merely states that cardinal numbers exist:

$$\forall\, xy\ (K(x) = K(y) \leftrightarrow \text{there exists a bijection between } x \text{ and } y)$$

(ii) define cardinal numbers as certain ordinal numbers, and use the

axiom of choice to show that every set has a cardinal number. The axiom of choice can be stated as follows:

$$\forall\, x\, \exists\, f \quad (f \text{ is a function with domain } x$$
$$\text{and } \forall\, z\,,\ z \text{ non-empty and } z \subseteq x \to f(z) \in z)$$

One of these ways has to be taken in order to show that we have the natural numbers.

7. Infinity axiom:

$$\exists\, x(\exists\, y(y \in x) \wedge \forall\, y(y \in x \to \exists z(y \in z \wedge z \in x)))$$

i.e. infinite collections exist.

This axiom is needed because the main purpose of algorithm theory is to provide a general theory of recursive definitions. Therefore there must exist an infinite collection in order to have the existence of the collection of natural numbers.

8. Separation axiom:

$$\forall\, x\, \exists\, y\, \forall\, z\, (z \in y \leftrightarrow z \in x \wedge \varphi(x))$$

where φ is a formula in which z has no free occurrence.

i.e. $\{z \in x : \varphi(x)\}$ is a collection.

This axiom is used for example in the statement of the theorem.

It appears from the above that the connection between sort theory and set theory needs to be made precise. Indeed in [11,12] the authors claim the existence of sets having algorithm collection structures. This matter is discussed in a companion paper [6].

5. An example : denotational semantics of a typed imperative program:

In this paragraph we assume that a set A is given, which contains all the sorts we need. For more details see [6]. The framework is similar to the one in [15].

(i) environments

Definition: Let A be a set of sorts, V a set of variables. An environment $K = (K_1 , K_2)$ is a couple of total functions $K_1 , K_2 : V \to A$ such that $\forall x \in V , K_1(x) \subseteq K_2(x)$.

For any $x \in V$, $K_1(x)$ is the value of x and $K_2(x)$ is the type of x . □

As an example $(\lambda x \in V . T , \lambda x \in V . T)$ is the least defined environment.

If K is an environment, $x \in V$ and $a , b \in A$, $K_{x \leftarrow (a,b)}$ is defined as being the environment k such that $\forall z \in V , z \neq x$ $k(z) = K(z)$ and $k(x) = (a , b)$. This notion of environemnt is one of the main features of the present semantics.

If e is an arithmetical or logical expression, and $K = (K_1 , K_2)$ is an environment, $Val(e)(K)$ will designate the value of expression e under the interpretation associated with function $K_1 : V \to A$. If I is a program instruction, $M(I)$ denotes the environment partial transformation associated with I , i.e. $M(I)(K)$ denotes the environment resulting from the execution of I in K , whenever this execution terminates.

(ii) procedure

Consider the following imperative procedure, inspired from [8]:

```
0.  procedure nat s (u : F2 nat (F nat nat)nat, v : F nat nat, x : nat)
        begin
1.          nat y , z ;
2.          y := 0 ; z := 0 ;
3.          while  y ≤ x do
4.              begin z := u(z , v(y)) ; y := y + 1  end ;
5.      return z
    end
```

Here we need a few explanations: nat means "natural number" ; F_2 nat (F nat nat)nat means F nat (F (F nat nat)nat). In the sequel each program line will be referred to with its number. Thus "1" means "nat y , z"

(iii) Semantic analysis of the procedure

0. As we have seen earlier (2.3) a procedure typing of the general form:

```
procedure  b  s (w : a )
    begin  B  end
```

is interpreted, if f_B is the function computed by the body B , as defining the functional sort

$$N_{a,b}(f_B) = \bigcap_W Fw((Fa\ f_B(w) \cap b)[w] \cap b) =$$
$$= \lambda w.\underline{if}\ w \subseteq a\ \underline{then}\ f_B(w) \cap b\ \underline{else}\ b$$

where a , b are the sorts named by a , b . (an alternative interpretation is:

$$N^*_{a,b}(f_B) = \lambda w.\underline{if}\ w \subseteq a\ \underline{then}\ f_B(w) \cap b\ \underline{else}\ T)$$

In the present case, calling B the body of our procedure, the function will be

$$N_{<F\ IN(F\ F\ IN\ IN\ IN)\ ,\ F\ IN\ IN\ ,\ IN>,IN}\ (f_B)$$

and the value of the function f_B will be given by:

$$f_B = \lambda\ UVX\ .\ Val(z)\ M(1;\ 2;\ 3;\ 4)\ (k)$$

with k being the initial environment defined by

$$k = (\lambda x.T,\lambda x.T)_{u \leftarrow (U,F\ IN\ F\ F\ \ IN\ IN\ IN)\ ,\ v \leftarrow (V,F\ IN\ IN),\ x \leftarrow (X,\ IN)}$$

and $M(1;\ 2;\ 3;\ 4)$ being the environment transformation defined by the body of the procedure.

1. The declaration statement

$$\underline{nat}\ y\ ,\ z$$

has a semantic

$$M(1) = \lambda K\ .\ K_{y \leftarrow (IN,\ IN)\ ,\ z \leftarrow (IN,\ IN)}$$

2. As for the procedure typing statement, there are two possible semantics for the assignment statement $u := e$, differing by the new value they give to u. The choice between these two possibilities is another main feature of the present semantics. The first one is:

$$(M(u := e)(K))_1(u) =$$
$$\underline{if}\ Val(e)(K_1) \subseteq K_2(u)\ \underline{then}\ Val(e)(K_1)\ \underline{else}\ K_2(u)$$

The other one is:

$$(M(u := e)(K))_1(u) = Val(e)(K_1) \cap K_2(u)$$

and seems more reasonable in the present case. (It is the one used in

Algol, Pascal,... compilers). Thus:

$$(M(2.)(K))_1(y) = \{0\} \cap IN = (M(2.)(K))_1(z)$$

3. $M(3)$ is being defined as the partial function

$$M(3_3) = \lambda K.M(\underbrace{4;\ldots;4}_{n \text{ times}})(K)$$

where n is the smallest natural number p such that

$$Val(y \leq x)M(\underbrace{4;\ldots,4}_{p \text{ times}})(K) = \text{false}$$

4. We have here

$$M(4)(K) = K_z \leftarrow (u[Vap(z)(K),\ v[Val(y)(K)]] \cap IN,\ IN)$$
$$y \leftarrow ((\bigcap_{n \in IN} F\{n\}\{n+1\})[Val(y)(K)] \cap IN,\ IN)$$

Thus the environment transformation defined by the body of the procedure is

$$M(1;\ 2;\ 3;\ 4) = \lambda K\ .\ M(3)(M(2)(M(1)(K)))$$

whence the functional sort computed by the procedure.

Acknowledgement: The author would like to thank Ed Ashcroft for his valuable comments.

References

[1] A. Aho, J. Ullman: Principles of Compiler Design, Addison Wesley, 1978.

[2] K. Apt, M.H. van Emden: Contributions to the theory of logic programming, University of Waterloo CS report #CS-80-12, October 1981.

[3] A. Church: A formulation of the simple theory of types, Journal of Symbolic Logic, Vol. 5, 1940, pp. 56-68.

[4] R. Kowalski: Logic for problem solving, North Holland, 1979.

[5] M.A. Nait Abdallah: Ordres élémentaires, 1er Colloque AFCET/SMF de Mathematiques Appliquées, Palaiseau, 1978, Vol. 2 pp. 115-123.

[6] M.A. Nait Abdallah: The necessity of double bundle structure in algorithm theory (to appear).

[7] L. Nolin: Systemes algorithmiques, systemes fonctionnels, 1st ICALP (1972).

[8] L. Nolin: Algorithmes universels, RAIRO, revue rouge, Mars 1974, pp. 5-18.

[9] L. Nolin: Les modèles informatiques du λ-calcul, in λ-calculus and computer science theory, Springer LNCS 19, 1975.

[10] L. Nolin: Pour le théorie des algorithmes, d'après A. Nait Abdallah, in Lambda-Calcul et Semantique formelle des langages de programmation (6 Ecole de Printemps d'Informatique théorique, La Chatre), LITP, 1978, pp. 267-275.

[11] L. Nolin, F. Le Berre: L'existence des espaces informatiques, C.R.A.S. t. 292, série I, pp. 499-502.

[12] L. Nolin, F. Le Berre: Les espaces informatiques, leur existence, leurs rapports avec la logique combinatoire et les λ-calculs, Revue technique Thomson /CSF Volume 13, No. 3, 1981, pp. 599-633.

[13] D. Scott: Data types as lattices, SIAM J. Comp. 5, 1976, pp. 522-587.

[14] A. Shamir, W. Wedge: Data types as objects, in Springer LNCS 52, 1977, pp. 465-479.

[15] J.E. Stoy: Denotational semantics: the Scott-Strachey approach to programming language theory, MIT Press, 1977.