Infinite Objects in Type Theory*

N. P. Mendler
P. Panangaden
R. L. Constable

TR 86-743
March 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

* This research supported in part by the National Science Foundation under grant MCS-81-04018. This paper is to appear in the proceedings of the 1986 Logics in Computer Science Conference, Cambridge, Massachusetts.

Infinite Objects in Type Theory

N.P. Mendler P. Panangaden R.L. Constable

March 28, 1986

Abstract

In this paper we show how infinite objects can be defined in a constructive type theory. The type theory that we use is a variant of Martin-Lof's Intuitionistic Type Theory. We show how one can express the intuition that infinite objects are understood through a limiting process without having to introduce partial objects in the theory. This means that we can adhere to the propositions-as-types principle. The type of infinite objects thus contains only total elements. The approximation is expressed through a sequence of types that approximate the type of infinite objects. We give two semantic accounts of types of infinite objects. The first is lattice theoretic and shows how these types can be understood as fixed points. The second is category theoretic and shows the duality between types of infinte objects and the ordinary recursive type definitions.

1 Introduction

The results of this paper extend those in Recursive Definitions in Type Theory [2]. In that paper it was shown how recursive type definitions could be could be given meaning in the framework of type theory. The objects inhabiting the recursively defined types were all finite and no notion of approximation was used. In the present paper we shall show how the

This research supported in part by the National Science Foundation under grant MCS-81-04018. This paper is to appear in the proceedings of the 1986 Logics in Computer Science Conference, Cambridge, Massachusetts.

type theoretic framework can be used to define types containing infinite objects such as streams. The type theory we use is the NuPRL type theory developed by Constable and Bates [1] which is essentially a variant of Per Martin-Lof's Intuitionistic Type Theory [4,5].

In domain theory [8] it is well known how to define domains containing potentially infinite objects. The basic intuition behind the domain theoretic account of infinite objects is that they are limits of sequences of partial objects. The domains containing the infinite objects thus necessarily also contain partial objects. In particular, there is a single element called "bottom" and written $\perp$ which is an approximation to all other elements and a member of all domains. In type theory a key principle is that propositions may be viewed as types [3] and that the empty type ***void*** represents falsity. If the domain theoretic account of infinite objects were literally mimiced in type theory it would be impossible to have an empty type and the propositions as types principle would not be tenable. In this paper we describe how to define proof rules which do describe types containing infinite objects (or lazy types as we often call them) and we give a semantic account which gives a fixed point characterization of these types but which does not require a notion of partial object. We shall also briefly sketch a category theoretic explanation of the proof rules and point out the duality between lazy types and the recursive types of [2]. A more complete account of the category theoretic semantics is given in [6].

The basic notion from type theory is that a type is a method of construction [5]. To specify a type we say how to construct its members and what it means for two members of a type to be equal. Among the types are those built from $\times$ for cartesian product, $+$ for disjoint union, and $\rightarrow$ for function space, allowing us to form types such as $\mathit{int} \times \mathit{int}$, $\mathit{int} \times (\mathit{int} \times \mathit{int})$. Recursive types arise when the method of construction is used in its own description, as in "a tree over A is either a, or a triple $\langle a, t_1, t_2 \rangle$ where t_1 and t_2 are trees over A and a belongs to A." We write such a type as $\mathit{rec}(t.\ A \mid (A \times t \times t))$. In general we write $\mathit{rec}(t, x.T; a)$ where T is an expression which may use $t(e)$ and x. Such recursive types describe only finite constructions over the types used in building T.

Potentially infinite objects are useful when we are considering a process of generation, say as in producing a specific sentence according to the productions of a context free grammar or in forming the sequence of moves

of a Turing machine. In these cases the sequence, say of elements from A, can be defined also as a function $f{:}N \rightarrow A$. But such a definition might be very inefficient, causing recomputation of $f(1),\ldots,f(n)$ in order to define $f(n+1)$; it might also be inconvenient, causing us to compute a dependence on n which is inessential to the process. Sometimes the generation process appears to be quite arbitrary as in the sequenced of characters generated at a terminal or as in a sequence of bits from a physical process. To account for these streams one might want to go beyond generators specified as rules to completely lawless sequences. In this paper we shall not consider lawless sequences. It seems best to first understand the limitations imposed on the theory, if any, by the requirement that all generators are given by rules.

Typically one specifies an infinite object such as a stream by using a recursive equation. For example, the stream of 1's can be defined by the equation $x = cons(1,x)$. Such equations are understood through an iterative process of unwinding the equation. This generative approach to understanding infinite objects is embodied in our syntax for members of the lazy type. As notation for elements of lazy types we introduce elements called generators denoted as $gen(z.G)$. These belong to "infinite recursive types" or "lazy types" denoted by $inf(x.T)$. A stream of A's is written $inf(x.A \times x)$ and the generator producing an infinite list of a's is just $gen(z.\langle a,z\rangle)$. A generator produces its elements according to a lazy evaluation mechanism given by the rule $gen(z.G) = G[gen(z.G)/z]$, so $gen(z.\langle a,z\rangle) = \langle a, gen(z.\langle a,z\rangle)\rangle$. It is important to note that the *gen* notation just introduced does not define the *canonical members* of the lazy types. We have merely introduced a convenient notation for the elements of a lazy type. Indeed, since the *gen* forms can always be unwound they are not canonical. The introduction rule below describes the canonical elements. However, with lazy objects we cannot expect to have a notation for the canonical elements, at best we can give a rule for describing the element and that is exactly what the *gen* forms do.

The account of infinite objects to be described in section 2 is formal and proof-theoretic. We give the rules for building $inf(x.T)$ types and for showing that elements belong to these types and rules for computing with generators. In sections 3 and 4 we describe two semantic accounts of our rules. In the first we use lattice theoretic ideas and define infinite recursive types as fixed points and in the second we sketch a category theoretic

explanation in terms of limits.

2 The Proof Rules

This section describes the proof rules for reasoning about lazy objects in type theory. The intuition behind these rules is captured formally in the semantic accounts presented in the later sections. The essential idea is that lazy objects can be understood as fixed points in an appropriate lattice of types. All the lazy types are built by refining a single type written $\Box$ and called box, which is a trivial type and plays a role analogous to $\perp$ in the category of domains.

The rules are written in refinement style: the main goal is written above the subgoal(s).

2.1 Box Proof Rules

Formation

$H \vdash \Box \in U_k$ **by intro**

$H \vdash \Box = \Box \in U_k$ **by intro**

The type $\Box$ is a new type whose members are all closed canonical terms and whose equality is trivial – all members are considered to be equal. Thus this type is extensionally the same as a singleton type but establishing membership in this type is trivial.

These rules express the fact that $\Box$ is present in all universes.

Introduction

$H \vdash e \in \Box$ **by intro** (if e has canonical form)

$H \vdash e \in \Box$ **by intro using** E
$H \vdash e \in E$

These rules state that any member of any type is in $\Box$. In the second rule the "using E" phrase specifies that the type of e is E; in other words, E must be a type expression. Whether a term is canonical is a syntactic question and therefore easily checked.

$H \vdash e = e' \in \square$ **by intro**
(if e and e' have canonical form)

$H \vdash e = e' \in \square$ **by intro using** E, E'
$H \vdash e \in E$
$H \vdash e' \in E'$

These rules state that the equality in the type $\square$ is trivial.

2.2 Lazy Type Proof Rules

The types of lazy or infinite objects are constructed by refining the membership and equality in the type $\square$. Thus, in our proof rules, we mimic the idea that an infinite object is built through a sequence of successive approximations. Note that we are not doing this by introducing any "undefined" object or partial element. Since the type $\square$ has trivial membership we can prove that larger and larger pieces of an infinite object are in the appropriate approximate type, leaving the rest of the object as a member of $\square$. The introduction rule for a lazy type will thus have an inductive character. Because we do not introduce any partial elements, the objects that inhabit the lazy types are all total objects. This is in contrast to the domain theoretic constructions where the domain of streams (for example) contains all the partial streams in addition to the infinite streams.

The types of infinite objects are built using the new type constructor inf. The syntax is of the general form $inf(x.c)$, where x is a bound variable and c is an expression that may contain free occurrences of x. Such a type expression can be read as the greatest solution to the fixed point equation $x = c$. Members of the infinite type can have form $gen(z.b)$. These can also be viewed as solutions to fixed point equations of the form $z = b$.

There is a syntactic restriction on the free occurrences of x in $inf(x.c)$: x may not occur free in left hand side of a function space type, in the argument of a function application, or in the principal argument(s) of the other elimination forms. This restriction is the same as the restriction imposed on recursive type definitions and prohibits impredicative definitions.

Formation

$H \vdash inf(x.c) \in U_k$ by intro [new v]
$H, v{:}U_k \vdash c[v/x] \in U_k$

$H \vdash inf(x.c) = inf(y.d) \in U_k$ by intro [new v]
$H, v{:}U_k \vdash c[v/x] = d[v/y] \in U_k$

These rules state that an infinite type is well formed if the body of the type expression is a function from types to types of the appropriate universe level and equality of infinite types is defined in terms of extensional equality of the appropriate type functions.

Introduction

$H \vdash e \in inf(x.c)$ by intro at U_k new n
$H, n{:}\text{int} \vdash e \in (\lambda x.c)^n(\Box)$
$H \vdash inf(x.c) \in U_k$

$H \vdash e = e' \in inf(x.c)$ by intro at U_k new n
$H, n{:}\text{int} \vdash e = e' \in (\lambda x.c)^n(\Box)$
$H \vdash inf(x.c) \in U_k$

These are the inductive introduction rules. For an element e to be in the appropriate infinite type it has to be in every "approximation" of the infinite type. Thus we are viewing $\Box$ as an approximation of every type as is suggested by its trivial membership and equality relations. The lazy type then is essentailly the "limit" of the types obtained by applying the type function $\lambda x.c$ repeatedly to the $\Box$ type. These intuitive justifications of the proof rules will be made precise in the semantic accounts of the inf type constructor.

$H \vdash e \in inf(x.c)$ by intro at U_k
$H \vdash e \in c[inf(x.c)/x]$
$H \vdash inf(x.c) \in U_k$

$H \vdash e = e' \in inf(x.c)$ by intro at U_k
$H \vdash e = e' \in c[inf(x.c)/x]$
$H \vdash inf(x.c) \in U_k$

These are the non-inductive introduction rules. They state that $inf(x.c)$ and $c[inf(x.c)/x]$ are extensionally equal types. This is what characterizes the lazy types as fixed points.

Elimination

$H, v{:}inf(x.c), H' \vdash E$ `by elim` v
$H, v{:}c[inf(x.c)/x], H' \vdash E$

Computation

$H \vdash gen(z.b) = e \in E$ `by reduce 1`
$H \vdash b[gen(z.b)/z] = e \in E$

The elimination and computation rules merely reflect the idea that one computes with infinite objects by unwinding their recursive definitions in a demand driven fashion. Note that since *gen* forms can be rewritten they cannot be canonical forms.

2.3 An Example

We shall illustrate these proof rules by showing that the function *zip*, which interleaves the elements of two streams, like teeth on an infinite zipper, is a (curried) binary operation on streams. On streams a and b we should have

$zip(a)(b) = \text{spread}(a; h, t.\ \langle h, zip(b)(t)\rangle)$

where the spread term is the logic's pair destructor: $\text{spread}(\langle c, d\rangle; x, y.t)$ evaluates to $t[c/x, d/y]$. This suggests the following definition for *zip*.

$zip =_{df} gen(z.\ \lambda a.\lambda b.\text{spread}(a; h, t.\ \langle h, z(b)(t)\rangle))$

Now if we define

$Str(A) =_{df} inf(s.\ A \times s)$

we can state our goal as

$A{:}U_1, a{:}Str(A), b{:}Str(A) \vdash zip(a)(b) \in Str(A)$.

Applying the lazy type introduction rule we generate the subgoal

$\ldots n{:}\text{int} \vdash zip(a)(b) \in (\lambda s.A \times s)^n(\Box)$.

We would like to show this by induction on n, but a stronger form is needed to push the induction through. So we cut, or sequence, in the formula

$$\forall a'{:}Str(A).\forall b'{:}Str(A).zip(a')(b') \in (\lambda s.A \times s)^n(\Box)$$

and proceed by induction on n. The base case is

$$\ldots \vdash \forall a'{:}Str(A).\forall b'{:}Str(A).zip(a')(b') \in \Box.$$

By introducing a' and b', computation on the *gen* form, and two β reductions, we arrive at

$$\ldots a'{:}Str(A) \ldots$$
$$\vdash \text{spread}(a'; h, t.\ \langle h, zip(b')(t)\rangle) \in \Box.$$

Now we eliminate on a', which yields the subgoal

$$\ldots a'{:}A \times Str(A) \ldots$$
$$\vdash \text{spread}(a'; h, t.\ \langle h, zip(b')(t)\rangle) \in \Box$$

and now we can use the $\times$ elimination rule and get

$$\ldots u{:}A, v{:}Str(A)$$
$$\vdash \text{spread}(\langle u, v\rangle; h, t.\ \langle h, zip(b')(t)\rangle) \in \Box$$

which by computation of the spread form yields

$$\ldots u{:}A, v{:}Str(A) \vdash \langle u, zip(b')(v)\rangle \in \Box.$$

Since $\langle u, zip(b')(v)\rangle$ has canonical form, this sequent holds by the introduction rule for $\Box$.

In the inductive step we must show

$$\forall a'{:}Str(A).\forall b'{:}Str(A).zip(a')(b') \in (\lambda s.A \times s)^n(\Box)$$
$$\vdash \forall a'{:}Str(A).\forall b'{:}Str(A).zip(a')(b')$$
$$\in A \times (\lambda s.A \times s)^n(\Box).$$

We follow the same steps as above and arrive at

$$\ldots u{:}A, v{:}Str(A)$$
$$\vdash \langle u, zip(b')(v)\rangle \in A \times (\lambda s.A \times s)^n(\Box).$$

By the $\times$ introduction rule we must show

$$\ldots u{:}A \ldots \vdash u \in A$$

which holds by A introduction, and

$\ldots \vdash zip(b')(v) \in (\lambda s.A \times s)^n(\Box)$

which follows from the induction hypothesis.

Most of the reasoning in the preceeding example is simple-minded. The standard "default" tactic of the current nuprl system would have completed the proof after the cut rule, so, to the user, this would have been a two step long proof.

The point to note here is that the induction proceeds by establishing that a given term is in a closer and closer "approximation" to the type of infinite streams rather than that approximations to the final infinite stream are all in a single domain. Thus we do express the intuition that infinite objects are understood through increasingly refined approximations but we do not admit partial objects into the theory to do so.

3 Lazy Types as Fixed Points

In this section we shall describe how lazy types are modelled as fixed points over an appropriate lattice. We shall build a complete lattice where types are viewed extensionally and lazy types turn out to be the least fixed points of continuous functions. Recursive types turn out to be the least fixed points of monotonic functions when we turn this lattice upside-down [2]. This duality between the two constructions will be made sharper in the following section in which we sketch a category theoretic account of the two constructions. The duality also appears in the proof rules. The lazy types have introduction rules which have an inductive character whereas the recursive types have an inductive elimination rule.

Define the complete lattice as follows. The elements are ordered pairs $\langle A, \sim_A \rangle$, where A is any set of closed, canonical nuprl terms that is closed under variable renaming, and $\sim_A$ is any equivalence relation on A which relates α-equivalent terms. For a lattice element $\gamma = \langle A, \sim_A \rangle$, it is convenient to introduce the notation

$$a \text{ in } \gamma =_{df} a \in A$$
$$a = b \text{ in } \gamma, \text{ or } a =_\gamma b \ =_{df} a \sim_A b.$$

The order relation is

$$\langle A, \sim_A \rangle \sqsubseteq \langle B, \sim_B \rangle \Leftrightarrow A \supseteq B \text{ and } \sim_A \supseteq \sim_B,$$

thus, for any set of elements Γ,

$$\bigsqcup \Gamma = \langle \bigcap_{\langle A, \sim_A \rangle \in \Gamma} A, \bigcap_{\langle A, \sim_A \rangle \in \Gamma} \sim_A \rangle.$$

We associate a given type T with the element $\langle A, \sim_A \rangle$, where A is the set of elements of T and $\sim_A$ is the equality relation for members of T. For example, $\Box$ is associated with the bottom element of the lattice, and `void` with the top, and extensionally equal types, such as $inf(x.\text{int} \times x)$ and $\text{int} \times inf(x.\text{int} \times x)$, are identified.

For a given type $inf(x.c)$, we view $\lambda x.c$ as a function on the lattice and show

> $\lambda x.c$ is a continuous function on the lattice, and $inf(x.c)$ is associated with its least fixed point.

This is proven by showing all the type constructors are continuous when viewed as operations on the lattice and that composition preserves continuity. These proofs are straightforward. Here we show the proof for the function type constructor. Consider $\lambda g.T \to g$ (recall that g may not occur free in T). Let $\langle A, \sim_A \rangle$ be the element associated with T. Given any element $\langle B, \sim_B \rangle$, we define $T \to \langle B, \sim_B \rangle$ to be the element $\langle C, \sim_C \rangle$, where:

1. C is the set of all lambda terms which, when evaluated on terms in A related by $\sim_A$, reduce to terms in B related by $\sim_B$.

2. $\sim_C$ is the equivalence relation on C asserting that two lambda terms in C are related iff, when evaluated on any term of A, they reduce to terms related by $\sim_B$.

This is the expected generalization of the function space type constructor. $\lambda g.T \to g$ is continuous: given a set of elements Γ, we now show

$$\bigsqcup \{T \to \gamma \mid \gamma \in \Gamma\} = T \to \bigsqcup \Gamma.$$

Proof. Given $\lambda x.b = \lambda x.c$ in $\bigsqcup\{T \to \gamma \mid \gamma \in \Gamma\}$, by the definition of $\bigsqcup$, $\lambda x.b = \lambda x.c$ in $T \to \gamma$ for every $\gamma \in \Gamma$, thus given any terms t, u for which $t =_T u$, $(\lambda x.b)(t) =_\gamma (\lambda x.b)(u)$, $(\lambda x.c)(t) =_\gamma (\lambda x.c)(u)$ and $(\lambda x.b)(t) =_\gamma (\lambda x.c)(t)$ for any $\gamma \in \Gamma$; so $\lambda x.b = \lambda x.c$ in $T \to \bigsqcup \Gamma$.

Given $\lambda x.b = \lambda x.c$ in $T \to \bigsqcup\Gamma$, by the definitions of $\to$ and $\bigsqcup$, for any terms t, u for which $t =_T u$, $(\lambda x.b)(t) =_\gamma (\lambda x.b)(u)$, $(\lambda x.c)(t) =_\gamma (\lambda x.c)(u)$ and $(\lambda x.b)(t) =_\gamma (\lambda x.c)(t)$ for any $\gamma \in \Gamma$; so $\lambda x.b = \lambda x.c$ in $T \to \gamma$ for any $\gamma \in \Gamma$, and therefore $\lambda x.b = \lambda x.c$ in $\bigsqcup\{T \to \gamma \mid \gamma \in \Gamma\}$.

However, in the complete lattice generated by inverting our order relation, this equality may not hold: thus the recursive types turn out to be least fixed points of monotonic, but not necessarily continous, functions.

Now we see that for a type to correspond to the least fixed point of $\lambda x.c$, its members must be members of $(\lambda x.c)^n(\Box)$ for every natural number n, and two terms will be equal in this type exactly when they are equal in each $(\lambda x.c)^n(\Box)$. This is what one would call an "extensional" equality and is precisely what the introduction rule for $\mathit{inf}(x.c)$ asserts. Since $\mathit{inf}(x.c)$ is associated with the least fixed point of $\lambda x.c$, the types $\mathit{inf}(x.c)$ and $c[\mathit{inf}(x.c)/x]$ are extensionally equal, and so the other rules for lazy types are sensible.

4 Category Theoretic Semantics

In this section we give a category theoretic account of the lazy type constructor. This is based on the correspondence between types and objects in certain categories discussed in [6]. It turns out that types may be viewed as objects in a category and the inhabitants of a type can be viewed as morphisms from the terminal object to the type. The morphisms in the category are functions between the types. The category is cartesian closed since the function spaces are themselves types. It can be shown that the standard type constructions such as product and disjoint sum correspond to various universal constructions in the category such as categorical product and categorical sum respectively. In this category we can easily see that the type $\Box$ is the terminal (or final) object whereas the type ***void*** is the initial object.

The recursive type construction described in [2] and the lazy type construction described in the present paper can be viewed as colimit and limit constructions respectively. First note that the empty type ***void*** and the type $\Box$ are initial and final objects in the category of types. Suppose we are given a unary type constructor T. This can be viewed as an endo-

functor on the category of types. For example, suppose we define the type constructor $\lambda A.int \times A$ which takes the type A and returns the type $int \times A$. As a functor its action on objects is already defined. The action of T on a morphism f from A to B is given by $\langle id, f \rangle$ from $int \times A$ to $int \times B$. Of course we have a certain freedom in defining the action of the functor on morphisms in the sense that given a unary type constructor T the action of T on morphisms is not specified. We have to make a sensible choice to make the construction below work. However, usually the obvious choice is the right choice to make.

Given this view of a type constructor as a functor we can build an ω diagram in a manner very similar to that of Plotkin and Smythe [7]. For the lazy types we proceed as follows. By repeatedly applying the functor T to the final object $\Box$ we obtain the sequence of objects

$$\Box \longleftarrow T(\Box) \longleftarrow T^2(\Box) \longleftarrow \ldots \longleftarrow T^n(\Box) \longleftarrow \ldots$$

Since $\Box$ is the final object there is a unique morphism ! from $T(\Box)$ to $\Box$. Applying T this morphism repeatedly generates a sequence of morphisms between the successive members of the sequence of objects. Now we have an ω-diagram. The limit of this is exactly the type of objects that satisfy the equation $x = T(x)$.

We can justify the inductive introduction rule on the basis of this construction. First note that saying that a type A is inhabited is interpreted as there being a morphism from $\Box$ to A. Now suppose we establish that a particular element is in every one of the types in the sequence of objects above. We now have a morphism from $\Box$ to every type in the sequence such that the entire diagram commutes. Since the type $inf(t.T)$ is the limit of the diagram it must be the case that there is a unique morphism from $\Box$ to $inf(t.T)$; in other words we have established that we have indeed an element of the lazy type. It is easy to mimic the arguments of the previous section to show that the usual type constructions are ω-continuous functors.

The construction for recursive types is exactly the dual construction starting from the initial object *void*. The inductive elimination rule in that case can be explained in terms of the co-universality of the recursive type construction. Suppose we wish to show that a type B is inhabited then we can do this by proving that the type $T(B) \rightarrow B$ is inhabited and that

the $rec(t.T)$ is inhabited. Why is this justified? First note that the type $rec(t.T)$ is the co–limit of the diagram

$$0 \longrightarrow T(0) \longrightarrow T^2(0) \longrightarrow \ldots \longrightarrow T^n(0) \longrightarrow \ldots$$

where we are using 0 as an abbreviation for *void*. If we have an element of $rec(t.T)$ we have an arrow from $\Box$ to $rec(t.T)$ so we now need an arrow from $rec(t.T)$ to B so that composition will give an arrow from $\Box$ to B. Since *void* is initial there is a unique arrow from *void* to B. Applying the functor T we get an arrow from $T(void)$ to $T(B)$. Composing this arrow with the arrow from $T(B)$ to B we get an arrow from $T(void)$ to B. Repeating this process we get arrows from $T^n(void)$ to B for all n. Since $rec(t.T)$ is the co–limit of the diagram above, co–universality induces a unique morphism from $rec(t.T)$ to B. This version of the inference rule states that the *rec* types form an initial T–algebra. The dual rule for *inf* types states that lazy types form a final T–coalgebra.

5 Conclusion

In this paper we have shown how we can describe infinite objects in type theory without using the concept of partial elements. We do capture the domain theoretic intuition that one understands infinite objects through a limit process, but we do not introduce any partial objects into the theory. This allows us to keep the propositions-as-types principle intact since we are not required to inhabit every type with $\perp$. We have given a lattice theoretic semantics which shows that lazy types are fixed points on an appropriate complete lattice. We also give category theoretic semantics that makes manifest the duality between infinite recursive types and finite recursive types. It is interesting to note that one construction requires a limit construction and a final object while the other requires a colimit construction and an initial object. Categories that contain universal objects, such as are used to explain type constructions in domain theory [8], cannot have both initial objects and final objects otherwise one can easily show that the category is degenerate [6]. It is straightforward to extend the constructions of the present paper to type constructions that are parametrized. Implementations of the type constructions described here are being carried out as part of the NuPRL project at Cornell.

6 Acknowledgements

We would like to thank Peter Aczel, Bob Harper and Michael Schwartzbach for helpful discussions.

References

[1] Constable, R.L. et al.
Implementing Mathematics in the NuPrl System.
To appear. Prentice-Hall 1986.

[2] Constable, R.L., Mendler, N.P.
Recursive Definitions in Type Theory.
In Proc. Logic of Programs, Lecture Notes in Computer Science Vol. 193.
Springer-Verlag 1985.

[3] Howard, W.A.
The Formulae-as-Types Notion of Construction.
Essays on Combinatory Logic, Lambda Calculus and Formalism.
Academic Press, 1980.

[4] Martin-Lof, P.
An Intuitionistic Theory of Types: Predicative Part.
In Proc. Logic Colloquium '73. North-Holland 1975.

[5] Martin-Lof, P.
Constructive Mathematics and Computer Programming.
In Proc. 6th International Congress for Logic, Method and Philosophy of
Science. Hanover 1979.

[6] Panangaden, P., Schwartzbach, M.I.
Categorical Type Theory.
TR 85-716, Cornell University, Computer Science Technical Report, 1985.

[7] Plotkin, G.D.,Smyth, M.B.
The Category Theoretic Solution of Recursive Domain Equations.
Siam Journal of Computing Vol 11 No 4, 1982.

[8] Scott, D.S.
Data types as lattices.
Siam J. Computing, Vol. 5, 1976.