Partial Objects in Constructive Type Theory

Robert L. Constable
Scott Fraser Smith
87-822

March 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

Partial Objects In Constructive Type Theory

Robert L. Constable Scott Fraser Smith
Department of Computer Science, Cornell University, Ithaca, NY 14853

Abstract

Constructive type theories generally treat only total functions; partial functions present serious difficulties. In this paper, a theory of partial objects is given which puts partial functions in a general context. Semantic foundations for the theory are given in terms of a theory of inductive relations. The domain of convergence of a partial function is exactly characterized by a predicate within the theory, allowing for abstract reasoning about termination. Induction principles are given for reasoning about these functions, and comparisons are made to the domain theoretic account of LCF. Finally, an undecidability result is presented to suggest connections to a subset of recursive function theory.

1 Introduction

1.1 Background

It has become widely accepted in the past few years that constructive type theory is relevant to many of the concerns of computer science. Such theories have become the core of functional programming languages [10,27] and have influenced the design of procedural languages as well [8,9,14,16]. They have served to organize the study of type disciplines generally [17,22,29], leading to new insights into polymorphism, modularity, and abstractions. Type theory has also played a significant role in proof checking and automated theorem proving [3,7,10,12,15,18,26], and it serves as a knowledge representation language for AI [10,35]. The theory has become central in the study of semantics [11,24,25,32], and it is influencing categorical and domain theoretic approaches to computing theory [21,24,28].

There are however important computing concepts that type theory does not presently treat in a satisfactory manner, in particular *partial functions*. Four of the most completely presented constructive type theories illustrate this point; they are Martin-Löf's Intuitionistic Type Theory [23], the Cornell type theory in the Nuprl system [10], and the Girard–Reynolds second order type theory [17,29] and its generalizations to the Theory of Constructions [15] and the Theory of Species [34]. Neither Martin-Löf's theory, call it ITT, nor the Theory of Constructions, say TC, nor the Theory of Species, say TS, treat the collection of partial functions from a type A to a type B as a distinct type. Nuprl does have a primitive type constructor $\sim>$ which given types A and B builds a type of partial computable functions from A to B, $A{\sim>}B$. This type constructor has shortcomings, however, that are described below.

In the next section we will examine some of the ways that partial functions can be treated in these theories. The lack of a partial function space constructor inhibits comparison of domain theory and type theory because the primitive function space constructor on domains builds just the space of all *partial* computable functions. The absence of a partial function space in type theory also inhibits comparing a type–theoretic computability theory with the classical theory developed by Turing, Kleene, Church, etc., which has become known as *Basic Recursive Function Theory*, BRFT. The novel treatment of partial functions presented here leads to results in both of these directions: we can show that the fixpoint induction rule from LCF, a domain-theoretic formalism, is meaningful and valid in type theory, and we show that there are unsolvable problems in the computability theory inside Nuprl.

Partial functions are also important because they arise naturally in practical programming languages. Even if one is interested exclusively in computing total functions, there are nevertheless compact notations which use constructors such as unbounded iteration and general recursion that in certain combinations define diverging computations, and these combinations cannot be computably separated from those which define total functions. Indeed, there are theoretical results which show that notation for partial functions is essential in any formalism which gives a compact interpretation of any r.e. class of total computable functions[5]. Theories such as ITT, TC, and TS do not permit these notations and so are significantly impoverished.

Partial functions play an especially interesting role in logics which allow programs as proofs, so-called *programming logics*. Type theories based on the propositions-as-types principle are examples of such logics (PL/CV[13] is another example). In these logics, partial functions represent proof attempts or *partial proofs*. It is possible to factor a problem into a partial proof and a demonstration that the partial proof is total. We give an induction rule below which produces partial proofs. This type of argument is interesting not only in programming but in mathematics generally.

Partial functions also pose a nontrivial challenge to logic designers. We will see shortly that there are many possible formalizations of the basic concept of partial functions, some easily accommodated in type theory and some not. In the remainder of this introduction, we will discuss four approaches which are sensible and progress toward the approach taken in this paper.

1.2 Approaches to Partial Functions

1.2.1 Total Functions on Subsets

In algebra a partial function f from a set A to a set B is often construed as a total function $f{:}D \to B$ where $D \subseteq A$ (D is the *domain* of f). This is the approach taken in Bourbaki[6], for example. This concept can be approximated in ITT by taking the type of partial functions from A to B, call it $\mathcal{P}(A \to B)$, as $\Sigma D \in (A \to U_1).\ \Pi x \in (\Sigma y \in A.\ D(y)).\ B$. A partial function is a pair $<D,f>$, where D is the domain of convergence of the algorithm f, and f maps pairs $<a,p>$ to elements of B, where $a \in A$ and $p \in D(a)$. This is awkward, especially since we want f to operate on elements of A. In Nuprl this rendering is somewhat less clumsy because the set type can be used, e.g. $\mathcal{P}$(A->B) is D:(A->U1)#{x:A|D(x)}->B[1]. Partial functions are still pairs, $<D,f>$, but f maps elements of A into B.

[1]The set type hides the witness to $D(x)$, so for example `0 ∈ {x:int|x<5}`.

1.2.2 Untyped Algorithms

A further refinement of the above idea is possible in Nuprl. All of the terms of the untyped lambda calculus are available in the computation system, including terms untypable directly in the theory such as the Y-combinator. The Nuprl term `fix(f,x.b)` is computationally equivalent to $Y(\lambda f.\lambda x.b)$, and is more concise. Terms built from the `fix` term can be typed in Nuprl (but not in ITT) because there are so-called *direct computation* rules which permit computation on terms as part of an argument to show that they are well-typed. The point is illustrated by a simple example:

$F \equiv$ `fix(f,x.if x=0 then λy.1 else f(x-1))`

is of type N->(N->N), where N is the natural numbers. This can be proved by induction on N. In the induction case, assume that F(x-1) is in N->N. To show F(x) $\in N$->N, perform a step of direct computation, which reduces the term to F(x-1), whereupon we may apply the induction hypothesis. In a system like ITT without direct computation rules, the reductions must be treated as equalities $a = b \in A$, and this requires that $a \in A$ and $b \in A$ are known. But in this case, $a \in A$ is not known until a is reduced to b.

1.2.3 Inductive Functions

The combination of untyped algorithms (λ-terms) and subtypes provides a flexible mechanism for defining partial functions, but this is not the only possibility. For example, in set-theoretic terms we can speak of defining a partial function and its domain of convergence by simultaneous induction. This will allow us to prove properties of the function by induction on the structure of the domain. For example, the function `fix(f,x.if even(x) then x/2 else f(f(3*x+1)))` can be treated as a graph, say $F(x,y)$, and defined inductively, or it can be treated as a function from D to N, where f and D are defined recursively. Examples of the two approaches are given below.

Functions-as-graphs:

$$F(x,y) = (\text{even}(x)\ \&\ y = x/2) \text{ or } (\text{odd}(x)\ \&\ \exists z.F(3*x+1,z)\ \&\ F(z,y))$$
$$D(x) = \exists y{:}N.F(x,y)$$

Functions-as-recursion:

$$f(x) = \text{if even}(x) \text{ then } f(x/2) \text{ else } f(f(3*x+1))$$
$$D = \{x{:}N \mid \text{even}(x) \text{ or } (\text{odd}(x)\ \&\ (3*x+1) \in D\ \&\ f(3*x+1) \in D)\}$$

In constructive type theory, functions cannot be identified with the graphs of their relations, and only the second style of mutual recursive definition is possible. This is expressible in theories such as Nuprl which permit the simultaneous definition of a type and a function over that type. The "$3x+1$" function and its graph are mutually defined in Nuprl as:

The function f:

```
fix(f,x.if x=1 then 0 else if even(x)
      then x/2 else f(3*x+1))
```

The graph D:

```
λx'.rec(F,x.if x=1 then true else
  if even(x) then F(x/2) else F(3*x+1);x')
```

We know that $f \in \{x{:}N|D(x)\}$->N. Although it is not known that this function is total, we can prove by induction on D that $\forall x{:}N.(D(x) \Rightarrow f(x) = 0 \in N)$. Such inductive arguments are not available in any of the treatments of partial functions discussed above.

This inductive approach to partial functions is discussed in [10]. In the case of partial functions from A to B, denoted A~>B, which do not use higher order functions, this theory appears adequate. For this restricted class of types, there is a procedure for automatically computing from the algorithm f a recursive predicate dom which defines exactly the subtype $\{x{:}A|\mathtt{dom}(f)(x)\}$ on which the algorithm converges. However, these exact domains are not computable in general. For example, a function such as

```
fix(f,x.if x=0 then λy.1 else λy.f(x+1)(y))
```

presents difficulties. `λy.f(x+1)(y)` is a canonical value, so it appears that the exact domain would be given by `λy.rec(F,x.if x=0 then true else true;y)`. However, the function clearly does not return a total function as its value.

1.2.4 Total Functionals

In an attempt to characterize the domain of a partial function exactly, we would like to introduce a relation such as $f(x)$ `in!` B to mean that $f(x)$ converges and the result lies in B. With this predicate defined, the exact domain of a partial function from A to B would be $\{x{:}A|f(x)$ `in!` $B\}$. But in order for the relationship to make sense, f must be assigned a type, say as a partial function. In order for that type to make sense, there must be an equality relation defined on it. But equality should mean that the functions are equal where they converge, and to speak of convergence, $f(x)$ `in!` B must be defined. One way to break the circularity is to only consider partial functions from A to B of the form `fix`$(f,x.b)$, where $\lambda f, x.b$ is a total functional mapping A->B to A->B, and to define `fix`$(f,x.b)$=`fix`$(f,x.b')$ iff $\lambda f, x.b = \lambda f, x.b' \in$ (A->B)->(A->B). Such a theory is sound, but it does not allow unrestricted nesting of partial functions in the bodies of other partial functions. However, it does show that a theory of partial functions with exact domains is possible. We offer the theory described below as a better alternative.

2 The Theory

2.1 Overview

Our goal is to define a general theory of partial objects with a wide range of applications. One of the key requirements is that it compare favorably with PPλ from LCF [18]. In the theories discussed here, we in fact have a class of types richer than those in LCF. It is also important to be able to exactly characterize termination within the theory and for there to be induction principles comparable to those of PPλ. It remains an open question to compare these function classes with higher type function classes studied by recursion theorists [19,31].

The key idea in solving this problem is to allow possibly nonterminating objects to inhabit some types; such a theory may be called a ***partial type theory.*** Corresponding to every type T in the underlying total theory there is a type $\overline{T}$ consisting of computations of elements of T. From a metalevel, the elements of $\overline{T}$ are terms in the underlying computation system which if they converge denote elements of T. Diverging terms are thus members of any type $\overline{T}$. From the object level, elements of $\overline{T}$ denote partial recursive computations, but the theory does not address the structure of the computation beyond giving a type to its final value. These computations are nevertheless mathematical objects. Given a type T of Nuprl, the *bar* type operator allows the type $\overline{T}$, which may contain diverging terms, to be formed. Types such as $T\#\overline{T'}$ may also be formed, allowing nontermination to be localized. We also define a termination predicate t `in!` T which asserts that t is a term that in fact terminates. This will allow us to abstractly reason about termination in the theory.

In this section, we will present the basic mathematical notions underlying the concept of partial types. We first will review the relational semantics of Stuart Allen [1,2] which serve as a foundation for predicative type theories. A simple theory will be defined which illustrates the essential ideas of this semantics. Then we will show how partial types may be added to this theory. Among the metatheorems proved are those which validate basic inference rules, including some forms of fixed point induction. This provides a comparison with LCF. We also illustrate a method of doing recursion theory in this context by proving a version of the unsolvability of the halting problem for functions in `int->`$\overline{\texttt{int}}$.

2.2 Relational Semantics for Type Theories

Elsewhere in this volume[2], Stuart Allen defines semantics for Martin-Löf's ITT[23]; this semantics will also serve as the foundation for the partial type theory that is the object of this study. The types and their inhabitants (i.e. members) are defined inductively, so for example if A and B[2] are known to be types and the inhabitants and equalities thereupon are also known, the function space A`->`B may be defined to be a type and its inhabitants and equality relation in turn defined. The situation is less obvious for dependent types, for it may not be clear that our definition is inductive. For this purpose we need to explicitly define the order in which the types of the theory are to be constructed. A type theory may be said to be a two-place relation Θ with $\Theta\ T\ \varepsilon$ meaning T is a type in the theory Θ and has equality and membership relation ε. Allen defines a monotone mapping $\mathcal{M}$ which given a type theory Θ defines a new theory Θ' which extends Θ. The least fixpoint of this mapping $\Psi = \mathcal{M}\Psi$ defines the complete type theory Ψ. By adding partial type constructors to the mapping, semantics for a partial type theory will be obtained. Here less formal semantics, which are not obviously inductive yet which are much more readable, will be used to define a Simple Type Theory, STT, which is a manageable subtheory of Nuprl. In section 2.3, STT will then be extended to a Simple Partial Type Theory, SPTT. The STT types are the type of integers `int`, the product constructor A`#`B, and the dependent function space constructor x`:`A`->`B (x may occur free in the type B), also known as a Π-type.

Beneath the type theory lies an untyped computation system which, in the case of STT, is simply the untyped λ-calculus with integers and pairing. The terms are listed in figure 1.

The reduction strategy used is lazy- or head-reduction, and many features of the theory hinge on this fact; other reduction strategies yield different theories. Terms are divided into two classes, *canonical* and *noncanonical* terms, as delineated in figure 1. Canonical terms are reduced no further in lazy evaluation, for the outermost form is not a computation. Types may themselves be objects of computation in

Canonical terms:
integers 0,1,-1,2,-2,...
λ-abstraction `\`x`.`b, pairing `<`a`,`b`>`
type expressions `int`, A`#`B, x`:`A`->`B

Noncanonical terms:
a *int_op* b (*int_op* $\equiv$ `+ - * mod`)
decision terms `int_eq(`a`;`b`;`t`;`f`)`, `less(`a`;`b`;`t`;`f`)`
application $a(b)$, projections $a.1$ and $a.2$

Figure 1: The Terms of STT

$a \leftarrow t$ iff eval(t)=a, where eval is defined as:

eval(t), where t is any canonical term $\equiv t$
eval(a *int_op* b)$\equiv$eval(a) int_op eval(b)
eval(`int_eq(`a`;`b`;`t`;`f`)`)$\equiv$ if eval(a)$=_{int}$eval(b) then eval(t) else eval(f)
eval(`less(`a`;`b`;`t`;`f`)`)$\equiv$ if eval(a)$<_{int}$eval(b) then eval(t) else eval(f)
eval($a(c)$)$\equiv$ if eval(a)=`\`x`.`b then eval($b[c/x]$) else error
eval($a.1$)$\equiv$ if eval(a)=`<`b`,`c`>` then eval(b) else error
eval($a.2$)$\equiv$ if eval(a)=`<`b`,`c`>` then eval(c) else error

Figure 2: The Evaluation Strategy of STT

this theory (`int_eq(5;7;int;int#int)` is a type.), so the type constructors are included in the class of canonical terms. We write $a \leftarrow t$ to indicate that term t head-reduces to a canonical term a. This relation is formally defined in figure 2. Note that $t \leftarrow t$ for all canonical t, and that there is no a where $a \leftarrow$ `<`a`,`b`>+3` or $a \leftarrow$ `(\`x`.`x`(`x`))(\`x`.`x`(`x`))`.

To define STT, we define relations for what our types are (T type), when objects inhabit a type ($t \in T$), and when objects are equal elements in a type ($t = t' \in T$). We can in fact combine the last two to simplify things: $t \in T$ means $t = t \in T$. It suffices to define these relations for closed terms only, for an open term has a meaning given by the meaning of a class of closed terms.

Definition 1 *STT is defined as follows:*

T type iff `int`$\leftarrow T$
or $\exists A, B.\ A$`#`$B \leftarrow T$ & A type & B type
or $\exists A, x, B.\ x$`:`A`->`$B \leftarrow T$ & A type & $\forall a.\ B[a/x]$ type if $a \in A$

$t = t' \in T$ iff `int`$\leftarrow T$ & $\exists n.\ n \leftarrow t$ & $n \leftarrow t'$

[2]In this discussion, x,y, and q range over variables, n and k range over integers, a, b, f, and t range over terms, and A, B, and T range over types. Types are actually just terms, and different variables are used for clarity only.

or $A\#B \leftarrow T$ & $A\#B$ type
& $\exists a, a', b, b'.$ <a,b>$\leftarrow t$ & <a',b'>$\leftarrow t'$ &
$a = a' \in A$ & $b = b' \in B$
or x:A->$B\leftarrow T$ & x:A->B type
& $\exists y, y', b, b'.$ \$y$.$b\leftarrow t$ & \$y'$.$b'\leftarrow t'$
& $\forall a, a'.\ b[a/y] = b'[a'/y'] \in B[a/x]$
if $a = a' \in A$

$t \in T$ iff $t = t \in T$

The following facts are true in STT:

Fact 1 `(3+(5*2))=13` $\in$ `(\x.x)(int)`
The type expression reduces to `int`; `13`$\leftarrow$`(3+(5*2))`, and 13 is an integer.

Fact 2 `(\x.<x.2,x.1>)(<7,6>)` $\in$`int#int`
`<<7,6>.2,<7,6>.1>`$\leftarrow$`(\x.<x.2,x.1>)(<7,6>)`, so for this term to be in `int#int` it must be the case that `<7,6>.2`$\in$`int` and `<7,6>.1`$\in$`int`, facts that follow trivially once we compute these terms to canonical form.

Fact 3 Y[3]`(\q,x.`
`int_eq(x;0;0;<x,q(x-1)>))` $\in$
`x:int->Y(\q,x.int_eq(x;0;int;int#q(x-1)))(x)`

Our type is well-formed because for every integer x, we get on the right hand side of the arrow an x-ary product of `int` types, each of which is a type. To prove membership, we must show that for an arbitrary integer n that the function applied to n is in the range type. This fact follows by induction on the integers.

We may use this semantics to justify rules for a logic. There will be a class of rules by which types are defined, and for each kind of type there are introduction and elimination rules. We also have a computation rule which allows terms to be reduced. If we wished to use the Y-combinator in the manner of the example above, rules for this would also be needed.

2.3 Partial Types

We wish to define a new class of types, *partial* types, embodied in the *bar* operator $\overline{T}$ on type T. The type $\overline{T}$ is inhabited by terms which may not converge under head-reduction. If a term $t \in \overline{T}$ does converge, however, it must inhabit T. The inhabitants of the types $\overline{T}$ are thus defined in terms of the inhabitants of T. This is the reverse of domain theory, where total objects are understood in terms of partial entities. As will be seen below, this approach fits in nicely with our semantic definitions.

[3] Y is the Y-combinator

The bar operator allows us to capture precisely the points in computation where divergence is accepted. For example, the type $\texttt{int\#}\overline{\texttt{int}}$ is inhabited by terms which all must converge to pairs <a,b> (such is the meaning of # types). The first element of the pair converges to an integer while the second element, *if* it converges, is an integer. The type $\overline{\texttt{int\#int}}$, however, contains terms which may diverge. Yet once a term in this type converges, the result must be a pair of `int`'s. The domain-theoretic partial objects amount to putting bars over all subtypes in an expression, as with $\overline{\overline{\texttt{int}}\texttt{\#}\overline{\texttt{int}}}$: diverging terms, pairs with one or two diverging elements, and pairs of integers all inhabit this type. A partial type theory thus has a more refined notion of convergence than domain theory.

It is not precise to say a function is a partial function, for there are many possibilities: $\overline{A\text{->}B}$, $A\text{->}\overline{B}$, $\overline{A}\text{->}\overline{B}$, and $\overline{\overline{A}\text{->}\overline{B}}$ could all be said to be partial functions, and each type has a slightly different meaning. However, the type which most closely corresponds to the programming language notion of partial function is $A\text{->}\overline{B}$: all inhabitants of the type converge to λ-terms \$x$.$b$, but upon applying elements $a \in A$, (\$x$.$b$)($a$) inhabits the type $\overline{B}$, admitting the possibility of divergence. This is the type we will hereafter refer to as a partial function space.

Definition 2 *The following equations extend STT to a Simple Partial Type Theory, SPTT:*

$t\downarrow$[4]iff $\exists a.\ a \leftarrow t$

$t \asymp t'$ iff $(t \downarrow \Leftrightarrow t' \downarrow)$

$\overline{T}$ type iff $T \downarrow \Rightarrow T$ type

$t = t' \in \overline{T}$ iff $\overline{T}$ type & $t \asymp t'$ & $(t, t' \downarrow$ [5]$\Rightarrow t = t' \in T)$

These equations are in fact very similar to those used to extend Nuprl to be a partial type theory; the semantic equations for Nuprl are discussed in section 2.5. $t \in \overline{T}$ means $t = t \in \overline{T}$, which can be seen to mean $t \downarrow \Rightarrow t \in T$. Two terms are said to be equal members in a bar type if when we know one converges, the other does, and if they both do converge, they are equal members of the (base) type T. One fact that follows from this is that any two diverging terms are equal members of any bar type. This corresponds to our intuition that one diverging term is as good as another, and hereafter they will all be represented by the generic symbol $\perp$.

[4] This relation is the formal definition of what is informally meant when we state that t terminates, converges, or halts.
[5] $t, t' \downarrow$ means $t \downarrow$ & $t' \downarrow$

Typehood for partial types should at least allow $\overline{T}$ to be a type when T is, but is this sufficient? Consider the possible type x:$\overline{\texttt{int->int\_eq}(x;0;\texttt{int};\texttt{int\#int})}$: if x diverges the int_eq computation will diverge as well, and the resulting expression will not be a type. It would be better to extend our notion of typehood for bar types: for $\overline{T}$ to be a type, T must be a type *provided* it converges. The above expression is then a type; so $\overline{(\perp)}$ is also a type.

Observe the following fact about the semantics: the meaning of type $\overline{T}$ is clearly defined in terms of the type T only. Thus, we will be able to formulate an inductively defined partial type theory in the spirit of Allen[2].

2.3.1 Facts About Partial Types

The following elementary facts about partial types give insights into the theory and also correspond to basic rules likely to be found in an implementation of a partial type logic.

Fact 1 *$t \in T$ implies $t \in \overline{T}$*

Fact 2 *$t \in \overline{T}$ and t is a canonical term implies $t \in T$*

Fact 3 $\forall T.\ T$ Type *implies* $\perp \in \overline{T}$

Fact 4 *$t = t' \in \overline{T}$ and $t \in T$ implies $t = t' \in T$*

Fact 5 *$f \in \overline{A\texttt{->}\overline{B}}$ and $a \in A$ implies $f(a) \in \overline{B}$*
Recall from the definition of -> types that $f' \in A\texttt{->}\overline{B}$ means $\forall a.\ a \in A$ implies $f'(a) \in \overline{B}$, which means $f'(a)\downarrow$ implies $f'(a) \in B$. To prove the fact above, suppose $f(a)\downarrow$; this means $f\downarrow$ because the first step of the reduction algorithm is to reduce f, so $f \in A\texttt{->}\overline{B}$; by the above, $f(a) \in B$ would hold.

Fact 6 *$f \in A\texttt{->}\overline{B}$ and $a \in \overline{A}$ does not necessarily imply that $f(a) \in \overline{B}$*
For example, take f =\x.<x,1>$\in$ int->$\overline{\texttt{int\#int}}$: $\perp \in \overline{\texttt{int}}$, but $f(\perp)$=<$\perp$,1> $\in \overline{\texttt{int\#int}}$ is a contradiction.

2.4 Inhabiting $\overline{T}$ Types

The interest in a partial type theory centers on the ability to define partial functions like Y(\f.\x. if x=0 then 1 else $f(x$-1)*x)$\in$int->$\overline{\texttt{int}}$. We need a principle which allows such functions to be formed freely in the logic. To express recursive computation concisely we introduce a new canonical term, fix(q,x.b), which has computational behavior identical to Y(\f.\x.b): fix(q,x.b)(a) reduces to b[fix(q,x.b),a/q,x].

In programming languages, recursive functions are typed by assuming all recursive calls are of the proper type:

$$\frac{q \in A\texttt{->}\overline{B} \Rightarrow \backslash x.b \in A\texttt{->}\overline{B}}{\texttt{fix}(q,x.b) \in A\texttt{->}\overline{B}}$$

We would like to have such a principle in type theory, but it does not follow trivially from the semantics.

To get started, we may observe the following: we know \x.$\perp\in A\texttt{->}\overline{B}$, so given the assumptions above the line and letting q be \x.$\perp$, we have \x.b[\x.$\perp/q$] $\in A\texttt{->}\overline{B}$; this function may in turn be q in the above, etcetera. Thus all of the unrollings of the function inhabit $A\texttt{->}\overline{B}$. These finite approximations for a functional f are often written as $f^k(\perp)$; we will notate unrollings over a fix body b with q and x free in it as b^k. By induction, we may show $\forall k.\ b^k \in A\texttt{->}\overline{B}$, so all finite approximations of a fix term are properly typed.

Consider the case where the type $A\texttt{->}\overline{B}$ is int->$\overline{\texttt{int}}$. For arbitrary $q \in$int->$\overline{\texttt{int}}$ and $x \in$ int, it is known that $b \in \overline{\texttt{int}}$; we wish to show fix($q$,$x$.$b$) $\in$ int->$\overline{\texttt{int}}$. By the definition of ->, this means showing $\forall n \in$ int. fix(q,x.b)(n) $\in \overline{\texttt{int}}$, which by the definition of the bar operator means fix(q,x.b)(n)$\downarrow$ implies fix(q,x.b)(n) $\in$ int. In programming language terms, if our recursive program terminates, the stack size must have had some maximum depth k over the computation. Analogously, given the fact that the fix term terminates, $b^k(n)$ must also terminate for some k, giving the same value as fix(q,x.b)(n). $b^k(n) \in$ int, so fix(q,x.b)(n) $\in$ int as well.

If the codomain type $\overline{B}$ is more complex, this approach falls short because there could be free occurrences of the fix term in the resulting canonical form. Consider showing fix(q,x.<$q(x)$.2,7>) $\in$ int->$\overline{(\overline{\texttt{int}}\texttt{\#}\overline{\texttt{int}})}$: this means that for arbitrary integer n, <fix(q,x.b)(n).2,7> $\in \overline{\texttt{int}}$#$\overline{\texttt{int}}$. However, all we have for the approximations is $\forall k,n$. <$b^k(n)$.2,7> $\in \overline{\texttt{int}}$#$\overline{\texttt{int}}$. In the case for the simple type $\overline{\texttt{int}}$ above, for large enough k the values of $b^k(n)$ and fix(q,x.b)(n) were *identical*. Here, however, the two terms will never be identical for any k. What we need to do is to apply the argument used above on each element of the pair: we know $\forall k$. $b^k(n) \in \overline{\texttt{int}}$ by the definition of #, and we would like to show fix(q,x.b)(n).2 $\in \overline{\texttt{int}}$. Assume the fix computation terminates: the reasoning then parallels the above, for $b^k(n)$ must terminate for some k. Thus, fix(q,x.b)(n).2 $\in \overline{\texttt{int}}$, which after putting the pair back together gives the result.

In general, the following principle can be proven valid in SPTT:

Theorem 1 (fix introduction)

$$\frac{\forall q{:}A\text{->}\overline{B}.\ \backslash x.b \in \overline{B}}{\texttt{fix}(q,x.b) \in A\text{->}\overline{B}}$$

The proof proceeds by induction on the semantic type structure. The **int** and $A\#B$ cases are sketched above, and the $x{:}A\text{->}B$ case proves no more difficult.

2.5 The Nuprl Type Theory

Ultimately, partial types should be a component of a powerful logic such as Nuprl, so after briefly describing some of the features of Nuprl not found in STT, we will explain how Nuprl may be extended to be a partial type theory.

The principle of treating types as propositions allows us to reason in the Nuprl theory. Empty types are false propositions, and inhabited types are true ones. Equality is extended to types as well; type equality is intensional in Nuprl as opposed to the extensional equality of ITT.

Some of the new types not found in STT are:

equality types t=t' **in** T are types which reflect the equalities $t = t' \in T$ back into the type theory. These types are well-formed when t and t' inhabit T and are inhabited only when $t = t' \in T$.

dependent products $x{:}A\#B$ (with x free in B), also known as dependent sums or sigma types, are useful as type constructors and also allow existential propositions to be represented.

empty type `void` , which is uninhabited and represents the unprovable proposition `false`.

other types $A|B$, $\{x{:}A|B\}$ and `list` A are disjoint unions which represent disjunction, set types which hide information, and list types.

universes $\texttt{U}k$ are a hierarchy of type universes which allow predicates and other higher–order objects to be treated abstractly. `U1` is a type which is inhabited by all "base" types; types inhabiting `U2` may treat `U1` itself as a type, so abstract predicates inhabit A->`U1`, which in turn inhabits `U2`. To be a type is to inhabit some universe $\texttt{U}k$. Equality on universe types is the intensional type equality mentioned previously.

The semantics for partial types in Nuprl are the same as the definitions for SPTT with in addition definitions of type equality and universe inhabitation for bar types:

$\overline{T} = \overline{T'} \in \texttt{U}k$ iff $T = T' \in \overline{\texttt{U}k}$

$T = T' \in \overline{\texttt{U}k}$ iff $T \asymp T'$ & $(T, T' \downarrow \Rightarrow T = T' \in \texttt{U}k)$

Rules for bar types have been formulated to extend Nuprl, although they have yet to be implemented. The Nuprl partial type theory will be the theory the rest of this paper refers to.

2.6 Abstract Reasoning About Termination

It is desirable to make abstract statements concerning the termination of a particular term, for example: "$f(7)$=5 provided f terminates everywhere". The first attempt at writing this in Nuprl is (∀`x:int.f(x) in int)->f(7)=5 in int`. For this term to be admitted as a proposition, it must be shown to be a type; this requires $\texttt{f}(n)$ `in int` to be a type for all integers n; by definition, ($f(n)$ `in int`) Type iff $\texttt{f}(n) \in$ `int`. Writing this statement thus asserts its truth, so another form of expression must be found. Attempts to directly reflect termination f(x)↓ into the theory will fail because there is no way we can admit such expressions as types and keep the theory sound. The solution is to define a typed termination predicate.

Definition 3 *The termination predicate* `in!` *is defined as follows:*
$(a\ \texttt{in!}\ A) \in \texttt{U}k$ iff $a \in \overline{A}$

$t \in (a\ \texttt{in!}\ A)$ iff $(a\ \texttt{in!}\ A) \in \texttt{U}k$ & $\texttt{axiom}{\leftarrow}t$ & $a\downarrow$

(t `in!` T) is a type when t inhabits $\overline{T}$ and is true when t converges; this allows us to sensibly hypothesize termination, writing the above example as (∀x:`int`.$f(x)$ `in! int)->`f`(7)=5 in int`, assuming that we know $f \in \texttt{int->}\overline{\texttt{int}}$. There are a multitude of uses for this predicate, some of which will be seen in the next section.

2.7 Induction Principles

There is a wide assortment of induction principles applicable to partial types. All of the principles discussed here will be for partial functions $A\text{->}\overline{B}$, and although principles for other types may be derived, what we do come up with is in fact quite general.

2.7.1 Computational Induction

If a recursive function $\texttt{fix}(q,x.b)$ is known to terminate upon application of a, the trace of the recursive calls will form a finite–depth, finitely–branching tree. This structure is well–founded, so induction principles may be formulated for it: if $\texttt{fix}(q,x.b)(a)$

terminates and we wish to show some property $P[\mathtt{fix}(q,x.b)(a)]$[6], it suffices to show $P[b]$ holds for arbitrary q and x, under the assumption that P is true for all values of the function called recursively in b. Since all recursive calls eventually terminate, our induction is well-founded. More formally[7],

$$\frac{\begin{array}{l}\mathtt{fix}(q,x.b)(a)\ \mathtt{in!}\ B\\ \forall q{:}A\text{->}\overline{B}.\ \forall x{:}A.\\ (\forall x'{:}A.\ \text{“}q(x')\text{ called in }b\text{”} \Rightarrow P[q(x')])\Rightarrow P[b]\end{array}}{P[\mathtt{fix}(q,x.b)(a)]}$$

The induction hypothesis is valid only for those $q(x')$ which are known to be on the computation path. If $q(x'')$ diverged, $P[q(x'')]$ could be false.

This principle may be proven sound by induction on the structure of the computation $\mathtt{fix}(q,x.b)(a)$, known to terminate by hypothesis.

For this rule to be implemented we need to be able to recognize when a particular $q(x')$ is on the computation path, and there is no mechanism in the logic which could perform such a task: intensional term analysis is required. But it is not difficult to recognize the class of terms which must recursively call $q(x')$ at the meta-level; for example, if b computes to $q(x')$ or $q(x')$+5 or $\mathtt{int\_eq}(q(x');b;t;f)$, $q(x')$ must be computed. However, not all terms with $q(x')$ free must compute it, for example <$q(x')$,b>: this computation has already terminated. In general, if b computes to any term t specified by the following grammar, we know $q(x')$ lies on the computation path, and the induction hypothesis is valid at that point:

$$\begin{array}{l}
t \rightarrow q(x')\\
X \rightarrow \text{(any term)}\\
t \rightarrow t\ \mathtt{+\ *\ -\ mod}\ X\\
t \rightarrow X\ \mathtt{+\ *\ -\ mod}\ t\\
t \rightarrow \mathtt{-}t\\
t \rightarrow \mathtt{int_eq}(t;X;X;X)\\
t \rightarrow \mathtt{int_eq}(X;t;X;X)\\
t \rightarrow \mathtt{less}(t;X;X;X)\\
t \rightarrow \mathtt{less}(X;t;X;X)\\
t \rightarrow \mathtt{ind}(t;u,v.X;X;u,v.X)\\
t \rightarrow \mathtt{decide}(t;x.X;y.X)\\
t \rightarrow \mathtt{spread}(t;x,y.X)\\
t \rightarrow t(X)
\end{array}$$

Because this language is decidable, our rule may check that t is of this form before allowing the induction hypothesis to be applied.

An interesting consequence of this approach is that the refinement-style sequent judgement >> of Nuprl needs to be reinterpreted to allow for such a rule. In the subtrees below the induction rule invocation, the judgement form is $\text{>>}^{P[b]}$, which has a new rule that allows for the induction hypothesis to be applied to terms t that meet the above criterion. Our sequents thus become dependent sequents. Perhaps there are other interesting dependent-sequent rules.

There is a classical computational induction rule worth mentioning:

$$\frac{\begin{array}{l}\mathtt{fix}(q,x.b)(a)\ \mathtt{in}\ \overline{B}\quad P[\bot]\\ \forall q{:}A\text{->}\overline{B},x{:}A.(\forall x'{:}A.P[q(x')])\Rightarrow P[b]\end{array}}{P[\mathtt{fix}(q,x.b)(a)]}$$

Because we know classically that our $\mathtt{fix}$ term either converges or diverges, we do not need to require that all applications of the induction hypothesis terminate; we merely prove $P[\bot]$, so our predicate is true even when the function diverges.

There is a novel form of induction for partial type theory, partial computational induction:

$$\frac{\begin{array}{l}\mathtt{fix}(q,x.b)(a)\ \mathtt{in}\ \overline{B}\\ \forall q{:}A\text{->}\overline{B},x{:}A.(\forall x'{:}A.\overline{P[q(x')]})\Rightarrow \overline{P[b]}\end{array}}{\overline{P[\mathtt{fix}(q,x.b)(a)]}}$$

This rule is similar to the classical rule in that it is not necessary to restrict attention to calls that terminate. We can justify this because our predicate $\overline{P}$ is a bar type, and its inhabiting object may not converge. In fact, if we apply the induction hypothesis to some $q(x'')$ that diverges, the object inhabiting $\overline{P}$ will diverge as well. Why bother to inhabit $\overline{P}$ when this type has no logical meaning as it is always inhabited? This is because if the inhabitant of $\overline{P}$ *does* converge, we will have a proof of the proposition P. Such a proof may be called a *partial proof*, for it is only a real proof if the inhabiting object converges. Such proofs give a different characterization of partial correctness, and deserve further study.

2.7.2 Fixpoint Induction

The most widely used induction principle in theories such as PPλ is fixpoint induction:

$$\frac{P[\lambda x.\ \bot]\quad \forall q.P[q] \Rightarrow P[\lambda x.b]}{P[\mathtt{fix}(q,x.b)]}$$

This rule is very concise and powerful, but it is only valid for a restricted subclass of the predicates P, the *admissable* P's. If P is continuous with respect to the limit of the finite approximations b^k, the rule is valid for that P. Because many predicates are continuous, this is a useful rule.

There is an analogue of fixpoint induction in partial type theory, although the class of admissable predicates is different. Because the type structure of Nuprl is so rich, even simple predicates may not be admissable if the type of the function is complicated. The types of PPλ are simple enough that this is not

[6] P is a one-place predicate with the argument in braces

[7] Under the propositions-as-types principle, $\forall x{:}A.B$ is represented by the type $x{:}A\text{->}B$, and the proposition $A \Rightarrow B$ is represented by the type $A\text{->}B$.

an issue, but it makes admissability conditions more baroque in Nuprl. Also, the term extracted to justify this rule would not be expressible in the current type theory because some intensional term analysis is required. Thus the proposition P could not be treated as a type if a fixpoint rule were to be implemented. Such problems illustrate disadvantages with the propositions–as–types paradigm.

3 Computability Theory

3.1 A Comparison to BRFT

Basic recursive function theory is a theory of partial computable functions on the natural numbers N; it is well–known and highly developed[30]. Our theory can be understood by comparison because facts about the type N->$\overline{N}$ constitute a theory of partial computable functions on N. First it is interesting to know that our theory is talking about the same objects. We can prove

Theorem 2 *If ϕ is a partial recursive function from N to N, then there is a term f of Nuprl such that $f \in N$->$\overline{N}$ and for all n, m in N, $\phi(n) = m$ iff $f(n) = m \in \overline{N}$.*

In BRFT we proceed by defining an indexing of the partial recursive functions, $\{\phi_i\}$. This would correspond to defining a function e:(n:N->N->(N^n->$\overline{N}$)) which is onto and which satisfies the S–m–n theorem and the universal machine theorem. It is possible to add axioms to Nuprl which approximate this concept. We may postulate an enumeration axiom:

$\exists e$:(n:N->N->(N^n->$\overline{N}$)).$\forall f$:N^n->$\overline{N}$.
$\quad\downarrow (\exists i{:}N.e(n)(i) = f$ in N^n->$\overline{N}$)

where the operation $\downarrow$ is called the *squash operator*. This operation uses the set type to hide the witness to $\exists i{:}N$ (see [10] for details of this operator), for the witness is not directly expressible in Nuprl. We then can also axiomatize the S–m–n relation. This realizable axiom leads to one version of recursive function theory, a theory which is similar to BRFT.

It is also possible to consider a theory in which N->$\overline{N}$ is not enumerable by postulating the negation of the enumeration axiom. This theory would be more like the classical theory of *all* partial functions.

3.2 Primitive Unsolvability

We may also approach some of the results of recursion theory directly without reference to enumerations. For example, the undecidability of the halting problem for terms in $\overline{\texttt{int}}$ may be proven inside the Nuprl partial type theory.

Theorem 3 (undecidability)
$\neg$($\forall$`t:`$\overline{\texttt{int}}$`.(t in! int)|`$\neg$`(t in! int))`[8]

If the symbols are given their computational meanings, this statement means that there is no algorithm that can decide whether a term in type $\overline{\texttt{int}}$ halts.
Proof. Assume that we could prove the body, i.e.

$\uparrow\downarrow\in\forall$`t:`$\overline{\texttt{int}}$`.(t in! int)|`$\neg$`(t in! int)`

Consider the function

$f \equiv$ `fix(q,x.decide(`$\uparrow\downarrow$`(q(x));a.`$\perp$`;b.1))`[9]

$f \in$ `int->`$\overline{\texttt{int}}$ follows because the result of the `decide` is either $\perp$ or 1.

By application of f to any integer (say 2) and computing, we have

f`(2)=decide(`$\uparrow\downarrow$`(`f`(2));x.`$\perp$`;y.1) in` $\overline{\texttt{int}}$

we know by hypothesis that

$\uparrow\downarrow$`(`f`(2)) in (`f`(2) in! int)|`$\neg$`(`f`(2) in! int)`

and can show that each case of this disjunction yields a contradiction.
`case` f`(2) in! int`: computing the `decide` term then gives f`(2)=`$\perp$ `in` $\overline{\texttt{int}}$; this contradicts the fact that f`(2)` terminates.
`case` $\neg$`(`f`(2) in! int)`: computing the `decide` term then gives f`(2)=1 in` $\overline{\texttt{int}}$; this contradicts the fact that f`(2)` diverges.

Acknowledgements

This work owes much to Stuart Allen's approach to type theory. We also thank Stuart, Doug Howe and Nax Mendler for helpful discussions.

References

[1] Stuart F. Allen, *The Semantics of Type Theoretic Languages.* Doctoral Dissertation, Computer Science Department, Cornell University, June 1987 (expected).

[2] Stuart F. Allen, A Non–Type–Theoretic Definition of Martin-Löf's types. (This volume).

[8] $\neg A$ is represented as A`->void`, and $\forall x{:}A.B$ is represented as x`:`A`->`B in Nuprl

[9] `decide` is the name of the decision procedure for disjoint unions; if $\uparrow\downarrow$`(q(x))` inhabits the left half of the union (meaning `t` terminates), the `decide` term evaluates to $\perp$; in the other case it evaluates to `1`.

[3] P. B. Andrews, Resolution in Type Theory. *J. Symbolic Logic*, v. 36 (1971), pages 414–432.

[4] Errett Bishop, *Foundations of Constructive Analysis.* McGraw–Hill, New York, 1967.

[5] M. Blum, On the Size of Machines. *Information and Control*,v. 11 (1967), pages 257–265.

[6] N. Bourbaki, *Elements of Mathematics.* Addison–Wesley, Reading, 1968.

[7] N. G. deBruijn, A Survey of the Project AUTOMATH. In *Essays in Combinatory Logic, Lambda Calculus, and Formalism*, J. P. Seldin and J. R. Hindley, eds. Academic Press, New York, 1980, pages 589–606.

[8] R. M. Burstall and B. Lampson, A Kernel Language for Abstract Data Types and Modules. In *Semantics of Data Types, Lecture Notes in Computer Science*, Vol. 173. Springer–Verlag, New York, 1984 pages 1–50.

[9] L. Cardelli, The Functional Abstract Machine. *Polymorphism: The ML/LCF/Hope Newsletter I* (1983).

[10] Robert L. Constable et. al., *Implementing Mathematics with the Nuprl Proof Development System.* Prentice–Hall, 1986.

[11] Robert L. Constable and N. P. Mendler, Recursive Definitions in Type Theory. In *Proceedings of Logics of Programs Conference, Lecture Notes in Computer Science*, Springer–Verlag, New York, 1985, pages 61–78.

[12] Robert L. Constable, Todd B. Knoblock, and Joseph L. Bates, Writing Programs That Construct Proofs. *J. Automated Reasoning*, v. 1 n. 3, (1984), pages 285–326.

[13] Robert L. Constable and Michael J. O'Donnell, *A Programming Logic.* Winthrop, Cambridge, MA, 1978.

[14] Robert L. Constable and Daniel R. Zlatin, The Type Theory of PL/CV3. ACM Trans. Prog. Lang. Sys., v. 7, n. 1 (1984), pages 72–93.

[15] Thierry Coquand and Gerard Huet, Constructions: A Higher Order Proof System for Mechanizing Mathematics. EUROCAL 85, Linz, Austria, April 1985.

[16] J. E. Donahue and A. J. Demers, Revised Report on Russell. *Computer Science Department Technical Report*, TR 79–389. Cornell University, Ithaca, NY, September 1979.

[17] J. Y. Girard, Une extension de l'interpretation de Godel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In *2nd Scandinavian Logic Symposium*, J. E. Fenstad, ed. North–Holland, Amsterdam, 1971, pages 63–92.

[18] Michael Gordon, Robin Milner, and Christopher Wadsworth, Edinburgh LCF: A Mechanized Logic of Computation. *Lecture Notes in Computer Science*, Vol. 78 Springer–Verlag, New York, 1979.

[19] Stephen C. Kleene, Recursive Functionals and Quantifiers of Finite Type. *Trans. Amer. Math. Soc.* v. 91 (1959), pages 1–52.

[20] Todd B. Knoblock and Robert L. Constable, Formalized Metareasoning in Type Theory. In *Proceedings of Symposium on Logic in Computer Science*, 1986, pages 237–248.

[21] J. Lambek and P. J. Scott, *Introduction to Higher Order Categorical Logic.* Cambridge University Press, 1986.

[22] D. B. MacQueen, Using Dependent Types to Express Modular Structure. In *13th ACM Symposium on Principles of Programming Languages*, 1986.

[23] Per Martin-Löf, Constructive Mathematics and Computer Programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science.* North–Holland, Amsterdam, 1982, pages 153–175.

[24] N. P. Mendler, P. Panangaden, and R. L. Constable, Infinite Objects in Type Theory. In *Proceedings of Symposium on Logic in Computer Science*, 1986, pages 249–255.

[25] Meyer and Reinhold, 'Type' is not a Type. In *13th ACM Symposium on Principles of Programming Languages*, 1986.

[26] Dale Miller, *Automating Higher Order Logic.* PhD Thesis, Carnegie–Mellon University, 1984.

[27] Bengt Nordstrom, Programming in Constructive Set Theory: Some Examples. In *Proceedings 1981 Conference on Functional Programming Languages and Computer Architecture.* Portsmouth, England, 1981, pages 141–153.

[28] P. Panangaden and M. I. Schwartzbach, Categorical Type Thoery. *Computer Science Department Technical Report*, TR 85–716. Cornell University, Ithaca, NY, 1985.

[29] John C. Reynolds, Towards a Theory of Type Structure. In *Proceedings Colloque sur la Programmation, Lecture Notes in Computer Science* Vol. 19. Springer–Verlag, New York, 1974, pages 408–423.

[30] Hartley Rogers Jr., *Theory of Recursive Functions and Effective Computability*. McGraw–Hill, 1967.

[31] Andre Scedrov, Kleene Computable Functionals and the Higher Order Existence Property. preprint, 1986.

[32] Dana Scott, Constructive Validity. In *Symposium on Automatic Demonstration, Lecture Notes in Mathematics*, Vol. 125. Springer–Verlag, New York, 1970, pages 237–275.

[33] Dana Scott, Data Types as Lattices. *SIAM J. Computing*, v. 5 (1976), pages 522–587.

[34] Sören Stenlund, *Combinators, λ-terms, and Proof Theory*. D. Reidel, Dordrecht, The Netherlands, 1972.

[35] Raymond Turner, *Logics for Artificial Intelligence*. Halsted Press (John Wiley & Sons), New York, 1984.