Universally Closed Classes of Total Computable Functions

Robert L. Constable

TR 84-640
September 1984

Department of Computer Science
Cornell University (and Edinburgh Universit)
Ithaca, New York 14853

Universally Closed Classes of Total Computable Functions

Robert L. Constable
Cornell University/Edinburgh University

ABSTRACT

One of the important characteristics of universal programming languages is that they can express their own semantics, that is universal partial functions, say interpreters, can be defined in the language. A fundamental result of computability theory is that no class of *total* functions can contain its own universal function. In practical terms this is disappointing, one must give up the advantages of universality to gain those of totality. In this paper the fundamental negative result is circumvented for programming languages with a sufficiently rich type structure. A new method is given for building a universally closed class of total computable functions starting with any class of such functions. These results have direct practical application in programming logics, and they raise new questions for computability theory.

1. INTRODUCTION

The following well-known results are interpretations of the same basic idea.

Cantor's Theorem: The set of functions $N \rightarrow N$ is not enumerable.

Kleene's Theorem: The set of total recursive functions is not recursively enumerable.

Grzegorczyk's Theorem: No universal function for the class of primitive recursive functions is primitive recursive.

All of these theorems are negative and are proved by diagonalizing over a hypothetical enumeration $f: N \rightarrow \mathbf{C}$ for $\mathbf{C}$ a set of functions from $N \rightarrow N$. That is, given f, a new diagonal function $d = \lambda x. f(x)(x) + 1$ is defined which cannot be in the range of f. In the case of Cantor's theorem, since $\mathbf{C}$ includes all functions, d must be in $\mathbf{C}$ so f cannot be onto. In Kleene's theorem f is forced to be onto, but if f is computable, then d is computable, so the conclusion is that f is not computable. In the final theorem, $\lambda y \lambda x. f(y)(x)$ is not primitive recursive.

The last theorem is the most delicate and the most discouraging. The universal function $\lambda y \lambda x.f(y)(x)$ requires some fixed enumeration of the class **C** usually obtained by enumerating a class C of expressions e which denote functions of **C**. There is an evaluation function, eval, which defines the meaning of expressions e in C. Clearly $\lambda x.eval(e)(x)$ is in **C** for each expression e. But it seems impossible to have the function $\lambda u \lambda x.eval(u)(x)$ in **C** because by composing some onto function $enum: N \rightarrow C$, the universal function $\lambda y \lambda x.eval(enum(y))(x)$ then belongs to **C**; and thus the diagonal function d does as well.

There seems to be no way out of this situation. Yet it is very desirable to have functions like eval in a programming language. So people are led to believe that so-called subrecursive programming languages are undesirable. One is thus led inevitably to *universal programming languages* in which eval is a program but in which partial functions are required.

These results have been known for so long and have been examined from so many angles that they seem natural and inevitable. But that is not the case. There is another perspective on them. Suppose we consider a programming language in which there are types of the form $\{x:N \mid P(x)\}$ denoting the set of those natural numbers satisfying P. Suppose also that type equality is defined so that

$$\{x:N \mid P(x)\} = \{x:N \mid Q(x)\} \text{ iff}$$
$$\text{for all } x \text{ in } N,\ P(x) \Leftrightarrow Q(x).$$

Under these circumstances it can happen that there is no enumeration of all expressions e which denote functions $f: N \rightarrow N$ because there is no enumeration of those expressions P for which we know for all x in N, $P(x)$. Thus it becomes possible that we can define an evaluation function, eval.

We show in the next section that this possibility can be realized. As a practical matter it means we can build a programming language in which all functions are total, including those of type $N \rightarrow N$, and yet there is a universal function for the class; that is, for any term t of type $N \rightarrow N$, $eval(t)$ is a function of type $N \rightarrow N$. However, it is not possible to enumerate all terms t of type $N \rightarrow N$.

What we have done is trade the ability to enumerate all terms of type $N \rightarrow N$, which is of dubious value, for the ability to evaluate any term of that type within the language, which is very valuable. Indeed, this capability will be used in an extension of our programming system *NuPRL* [3].

In the next section we outline a construction of eval for a minimal language of the kind suggested in this discussion. We also show that such languages can be built from any base class of total functions such as the primitive recursive functions or polynomial time computable functions.

2. CLOSURE CONSTRUCTIONS

2.1. Using Universal Machines

The most direct way to see that there can be a class of total recursive functions containing an eval function is to impose the right type structure on the class of partial recursive functions. For example, let F be the class of expressions for the μ-recursive function [6]. Let $eval(f)$ denote the partial recursive function denoted by f.

Let the types be defined over N, the natural numbers, and over the expressions F by the condition that N and F are types and if T is a type, then so are $T \times N$, $T \times F$ and $\{x{:}T \mid P(x)\}$ for P a predicate on n-tuples of natural numbers and expressions. In the expression {x:T | P(x)}, T is called the *base type*.

If we use the Kleene T-predicate [6], then we can define the domain type of a partial recursive function on N as $\{x{:}N \mid \exists y{:}N.T(f,x,y)\}$. The same idea applies over numbers and expressions. Now we say that f has the type $f{:}\{x{:}N \mid \exists y{:}N.T(f,x,y)\} \rightarrow N$, and is thus a *total function*. The function $\lambda f\, \lambda x.eval(f)(x)$ has the type

$$\{(e,x){:}F \times N \mid arg(e)=1 \;\&\; dom(e)=N \;\&\; \exists y{:}N.T(e,x,y)\} \rightarrow N$$

where arg(e) is the number of arguments of e and dom(e) denotes the domain type of e.

The details of this construction are not interesting in this paper, but see Constable [1] for the method of treating partial recursive functions with this type structure. The resulting system is similar to Lisp with types. The power of this approach comes from the theory which allows us to express and prove the predicates P which characterize the domains.

2.2. Subrecursive Closure

1. *Outline*

In this section we outline a method of building a universally closed class starting with a base of total functions. The idea is to reflect the expressions for functions into the data objects of the programming language. Suppose we start with a class of expression F which denote functions, say for $f \in F$, f denotes a function from A to B, written $f:A \rightarrow B$. Let there be a type of terms, term, in addition to any other atomic types. Suppose there is a mapping

inside:F $\rightarrow$ term .

This is a function in the metalanguage. We also define a form value(x) such that for each type T

value:{x:term | typeof x is T}→T .

The forms $value(t)$ belong to the object theory. They are thus themselves reflected into term, e.g. inside $(value(t)) \in$ *term*. In general we will have

value(inside(f)) = f

for any expression of type T.

We will use the word "expression" to speak of the metalanguage concepts and the word "term" to refer to the data type used to represent expressions in the language. Thus we have an expression for the empty list, say nil. We also have a term denoting this, say (*nil*), So inside (nil) = *nil* and value((*nil*)) = nil.

The key to the success of this method is the type construct $\{x:term \mid typeof(x) = T\}$ and the attendant use of "value". We can express the value of terms of a known type T, but we cannot write an arbitrary "universal" function of the form $\lambda x,y.value(x)(y)$ mapping $(term \times D \rightarrow D)$.

With types of the form $\{x:A \mid P\}$ we are prepared to give up the possibility of decidable type checking. In the case of the method outlined in section 2.1, it is clearly not decidable in general whether an expression e is of type $\{x:N \mid \exists y:N.T(f,x,y)\}$. But this is not a major drawback in programming logics, such as [3], where properties of expression are proved not decided.

We present the closure construction for a simple artificial language. There is an atomic data type called atoms, there are lists, cartesian products of types and subtypes. Certain lists represent terms.

All the functions of the language are *total recursive.* To make the description of value(t) easy, we define all functions using combinators instead of lambda terms.

2. *Language Definition*

Types

As a first approximation we think of the types of this language as follows:

1. list is a base type
2. atom is a base type
3. $\{x:list \mid typeof\ \ x\ \ is\ \ T\}$ is a base type
4. If B is a base type and P is a predicate

   then $\{x:B \mid P\}$ is a base type.
5. If B_i are base types

   then $B_1 \times \cdots \times B_n \rightarrow B_{n+1}$ is a type
6. If T is a type

   then $\{x:list \mid typeof\ \ x\ \ is\ \ T\} \rightarrow T$ is a type.

We can abbreviate $\{x:list \mid typeof\ \ x\ \ is\ \ T\}$ as list$|T$.

Forms

Functions are built from certain constant functions, cons, car, cdr, ls, at. The types of these functions are

cons : list $\times$ list $\rightarrow$ list

car : list $\rightarrow$ list

cdr : list $\rightarrow$ list

ls : atom $\rightarrow$ list

at : {x:list | singleton(x)} $\rightarrow$ atom where singleton(x) is a predicate defining lists of the form (a) where a is a term.

Functions are defined using the following combinators: rec(n), scomp(n), comp(n,m), proj(n,i), const(n)(a), and *value*. There are infinitely many basic combinators distinguished by the parameters n, m, i, a. In each case, n gives the *arity* of the function. To keep the type structure simple we do not assign values to these basic combinators but only to compound forms built using them. These forms are described as follows:

F1: If b has arity n and h has arity $n+2$, then $rec(n)(b,h)$ is an n ary *form*.

This form defines the function given by $rec(n)(b,h)(nil,y_1,\ldots,y_n) =$

$$b(y_1, \ldots, y_n)$$

$rec(n)(b,h)(cons(a,d),y_1,\ldots,y_n) =$

$$h(cons(a,d),rec(n)(b,h)(d,y_1,\ldots,y_n),y_1,\ldots,y_n)$$

F2: If $f_1,\ldots,f_m$ have arity n and h has arity m then $comp(n,m)(h,f_1,\ldots,f_m)$ has arity n. This form denotes the function defined by $comp(n,m)(h,f_1,\ldots,f_m)(x_1,\ldots,x_n) = h(f_1(x_1,\ldots,x_n),\ldots,f_m(x_1,\ldots,x_n))$

The form $proj(n,i)$ is of arity n and is defined by $proj(n,i)(x_1,\ldots,x_n) = x_i$ for $1 \leq i \leq n$ (there is no form when n and i do not satisfy $1 \leq i \leq n$).

F3: The form $scomp(n)(f,g)$ is of arity n and is defined by scomp(n)(f,g)($x_1,\ldots,x_n$) = f(x_1)($g(x_1,\ldots,x_n)$).

F4: The form $const(n)(a)$ is of arity n and is defined by $const(n)(a)(x_1,\ldots,x_n) = a$.

Atoms

There are infinitely many atoms. In particular the following are atoms:

nil, quote, cons, car, cdr, at, ls,
const(n)
comp(n,m)
proj(i,j)
rec(n)
scomp(n)
value

Terms

Terms will be elements of types of the form {x:list|typeof x is T} where T is a type expression; these are the terms of type T. Their definition depends on the rules for defining the predicate "typeof t is T" on lists. This predicate is inductively defined; intuitively typeof t is T iff t denotes an object of type T. Thus type of *nil* is list.

We are interested in the defined operator called ' "quote" on lists. For any t $\in$ {x:list | typeof x is T} the element (*quote* t) is a list, and

typeof(*quote* t) is {x:list | typeof x is T}.

Typeof predicate

The predicate "typeof _ is _" is inductively defined by these rules:

(T1) typeof (*nil*) is list
typeof (*car*) is list $\rightarrow$ list
typeof (*cdr*) is list $\rightarrow$ list
typeof (*cons*) is list $\times$ list $\rightarrow$ list
typeof (*ls*) is list $\rightarrow$ list
typeof (*at*) is {x:list | singleton(x)} $\rightarrow$ atom

(T2) typeof (*value*) is {x:list | typeof x is T}$\rightarrow$T for any T.

(T3) For any types $T_1,\ldots,T_n$,
typeof *proj*(n,i) is $T_1 \times \ldots \times T_n \rightarrow T_i$
for $1 \leq i \leq n$.

(T4) For any types $T_1,\ldots,T_n$, T and t for which typeof t is T
then typeof (*const*(n)(t)) is $T_1 \times \ldots \times T_n \rightarrow T$

(T5) For any types $T_1,\ldots,T_n$, $S_1,\ldots,S_m$, T,
if typeof f_i is $T_1 \times \ldots \times T_n \rightarrow S_i$ for i=1,...,m
and typeof h is $S_1 \times \ldots \times S_m \rightarrow T$,
then typeof (*comp*(n,m) h $f_1 \ldots f_m$) is $T_1 \times \ldots \times T_n \rightarrow T$

(T6) For any types $T_1,\ldots,T_n$, T, S
if typeof f is $T_1 \to (T \to S)$ and
typeof g is $T_1 \times \ldots \times T_n \to T$
then typeof (*scomp*(n) f g) is $T_1 \times \ldots \times T_n \to T$.

(T7) For any types $T_1,\ldots,T_n$, T,
if typeof b is $T_1 \times \ldots \times T_n \to T$ and
typeof h is list $\times$ T $\times$ $T_1 \times \ldots \times T_n \to T$
then typeof(*rec*(n) b h) is list $\times$ $T_1 \times \ldots \times T_n \to T$

(T8) For any types $T_1,\ldots,T_n$, T
if typeof f is $T_1 \times \ldots \times T_n \to T$ and
if typeof a_i is T_i for i=1,...,n
then typeof (f $a_1 \ldots a_n$) is T

(T9) If t is a list and P(t)
then typeof (*quote* t) is {x:list |P(x)}
This includes the special case:

For any type T
if typeof t is T
then typeof (*quote* t) is {x:list |typeof x is T}

3. *Definition of Value*

The form value(t) is well-defined only for t of the right type. For any type T and term t for which we can prove typeof t is T, we have value(t) is of type T. We know in fact value:{x:list | typeof x is T} → T for any T, but value itself is not a function of any fixed type, as is the case for proj(n,i) or const(n)(t).

The computation rules for value are these:

constants:
value((*nil*)) = nil
value((*cons*)) = cons
.
.
.
value((*quote* t)) = (t)

functions:
value(($comp(n,m)$ h $g_1 \ldots g_m$)) =
comp(n,m)(value(h),value(g_1),...,$value(g_m)$)

value(($proj(i,j)$)) = proj(i,j)

value(($rec(n)$ b h)) =
rec(n)(value(b),value(h))

value(($scomp(n)$ v f)) =
scomp(n)(value(v),value(f))

applications:
value((f $a_1 \ldots a_n$)) =
value(f)(value(a_1),...,$value(a_n)$)

3. CLOSURE RESULTS

3.1. Overview

We now consider two important properties of the subrecursive closure. First we define a mapping called inside which takes expressions, which are metalanguage, into terms, which are object language. We then show that value is the inverse of inside. That is we prove

Theorem 1: For every expression e of type T

value(inside(e)) = e

Next we show that all function expressions are total. That is

Theorem 2: If e is an expression of type $T_1 \times \ldots \times T_n \rightarrow T$ and t_i are of type T_i, i=1,...,n then e(t_1,...,t_n) denotes an object of type T.

The totality result is subtle because with minor changes in the language it will fail. For example if there were an onto function e:list→{x:list | typeof x is list→list} then one could define the diagonal function d(x) = (*nil* value(e(x))(x)). The combinator form, say d = comp(1,2)(cons,const(1)(*nil*),scomp(1)(comp(1,1)(value,e),proj(1,1))) would satisfy d(inside(d)) = (*nil* d(inside(d)) which is contradictory. So it must be impossible to enumerate in the language all terms denoting total functions. This happens because the type structure is so rich that type equality is not decidable.

3.2. Definition of "inside"

The function "inside" maps metalanguage into object language, in particular expressions to terms. To define it, consider the forms of expression:

constants:
  list: nil
  atom: *nil,quote,cons,car,...*

forms:
  comp(n,m)
  scomp(n)
  rec(n)

functions:
  cons,car,cdr,value,...
  proj(n,i),const(n)(t)
  comp(n,m)(h,$g_1,\ldots,g_m$)
  scomp(n)(v,g)
  rec(n)(b,h)

application:
  f($a_1,\ldots,a_n$)
  where f is a function expression and a_i are expressions

Definition of inside:

constants:
  inside(nil) = (*nil*)
  inside(*nil*) = (*quote nil*)
  inside(*cons*) = (*quote cons*)
  inside(a) = (*quote* a) for any atom a.
  .
  .
  .
  inside((*quote* a)) = (*quote* (*quote* t))

forms:
  inside(comp(n,m)) = *comp*(n,m)
  inside(scomp(n)) = *scomp*(n)
  inside(rec(n)) = *rec*(n)
  .
  .
  .

functions:

inside(cons) = (*cons*)
inside(car) = (*car*)

.
.
.

inside(value) = (*value*)

inside(comp(n,m)(h,g_1,...,g_m)) =
(inside(comp(n,m)) inside(h) inside(g_1)...inside(g_m))

inside(rec(n)(b,h)) =
(inside(rec(n)) inside(b) inside(h))

applications:

inside(f(a_1,...,a_n)) =
(inside(f) inside(a_1)...inside(a_n))

3.3. Reflection

Theorem 1 (Reflection): For all types T and any t∈T,

value(inside(t)) = t.

Proof (by structural induction on t):

base case (constants):

value(inside(nil)) = value((*nil*)) = nil

value(inside(*nil*)) = value((*quote nil*)) = *nil*

.
.
.

value(inside(proj(n,i))) = value((*proj*(n,i))) = proj(n,i)

The other cases are all similar.

induction case (forms, functions, applications):

value(inside (const(n)(t))) = value((*const*(n) inside(t)))

= value(*const*(n))(value(inside(t)))

= const(n)(t) by the induction hypothesis that

value(inside(t)) = t.

value(inside(comp(n,m)(h,g_1,...,g_m))) =

value(($comp$(n,m) inside(h) inside(g_1)...inside(g_n))) =

comp(n,m)(value(inside(h)), value(inside(g_1)),...,value(inside(g_m))) =

comp(n,m)(h,g_1,...,g_m) by the induction hypothesis.

value(inside(f(a))) = value((inside(f) inside(a))) =

= value(inside(f))(value(inside(a)))

= f(a) by induction hypothesis.

The other cases are all similar.

Qed.

3.4. Termination

We now show that all functions are total, arguing by induction on the structure of the proof that the inputs to the function are of the right type. The essential difficulty with a termination proof of this kind is that value is defined recursively, and in considering expressions of the form value(e(a))(b) one must exclude the possibility that e(a) evaluates to a form "more complex than e(a)" so that the successive calls to value diverge. In this setting that possibility is precluded because e must be proved to have type A → {x:list | typeof x is B}. Such proofs can be arranged hierarchically depending on the nesting of proofs of the predicates typeof x is B. At the base there are only those proofs which make no appeal to such predicates. Thus in this case we know that the proof that e:A → {x:list | typeof x is B} involves value only in cases where it is known to converge.

This hierarchical arrangement of type checking proofs is possible because there is no interaction between computation and type checking. For example there is no rule of the form

$$\frac{f(x) = a}{typeof\ \ x\ \ is\ \ B}.$$

In particular, there is no function typeof(t) = T.

Theorem 2 (Totality): If f: $T_1 \times \ldots \times T_n \rightarrow$ T and if

$a_i \in T_i$ for i=1,...,n, then

f(a_1,...,a_n) reduces to some t $\in$ T

in a finite number of steps.

Proof

Consider the proof that $a_i \in T_i$. This proof involves the type checking rules and rules for proving predicates P(t) including those of the form "typeof t is T." We proceed by induction on the structure of the proofs of the "typeof t is T" predicates.

In the base case there are no such predicates. We argue by induction on the structure of f. We know that f does not contain any occurrence of "value" because in order to be type correct, an application of value must satisfy "typeof t is T." The demonstration of totality in this case is straightforward (similar to proofs in [6,7]).

In the induction case, consider a proof that $a_i \in T_i$ and look at the outer most proof of the form "typeof t is T." We know that this is to establish value:{x:list | typeof x is T} → T and that all imbedded occurrences of value are known to terminate (these are the induction hypotheses). Consider how "typeof t is T" is established, there are 9 possibilities, rules T1 - T9.

Suppose t is the constant (*value*) which is being assigned type {x:list | typeof x is T}, then T must be {y:list | typeof y is S}→S. In this case we know that value(*value*) = value and value:{y:list | typeof y is S}→S. Notice that S cannot include reference to T since T contains S as a strict subcomponent. So this apparent self-evaluation which in the absence of types would lead to nonterminating behavior in this case does not.

Suppose t is (f a) where "typeof f is A→B" and "typeof a is A" are proved. Then we know that value((f a)) = value(f)(value(a)). By the induction hypothesis we have value(a) is defined and has value b in B. Likewise value(f) is defined and has type A→B.

All of the other noninductive proofs of "typeof t is T" are handled similarly. A different kind of case arises with a term t of the form (rec(n)(b,h)). Suppose this is proved to have the type list×B→A where "typeof b is B→A" and "typeof h is list×A×B→A." We know that value((*rec*(n)(b,h))) is rec(n)(value(b),value(h)) and by the induction hypothesis, value(b),value(h) are defined and have the proper type for rec(n). We also know that rec(n)(value(b),value(h)) defines a total function.

4. FURTHER REFLECTION

4.1. Discussion

Although the results of section 3 show that there is no way to build diagonal functions and that universal total functions exist, still the results are weaker than they might appear because there is so little that can be done in this calculus to build internal forms of terms. Suppose, for example, that we wanted to build a function which took a list of n atoms, say (a...a), representing the integer n, and returned as value a term denoting the function λx.car(car(...car(x)...)), that is, car composed with itself n times. The pattern of definition is simple; in the base case, we produce (*car*) and in the induction case, if f represents n-1 compositions, then (*comp*(1,1) *car* f) will represent n compositions. So we want the function

rec(0) ((*car*), comp(1,2)(cons,const(1)(*comp*(1,1)),

comp(1,2)(cons,const(1)(*car*),prog(1,1)))) .

This function, call it g, has the right property, but we will not be able to prove that g:list→{x:list | typeof x is list→list} because we cannot use the fact that if the input, say y, is of type {x:list | typeof x is list→list} then (comp(1,1)*car* y) is of the right type. What we need is a rule of the form

(*) comp(1,2)(cons,const(1)(*comp*(1,1)),comp(1,2)(cons,const(1)(*car*),prog(1,1))):
{x:list | typeof x is list→list}→{x:list | typeof x is list→list}

or more generally of the form

(**) comp(2,2)(cons,const(1)(*comp*(n,m)),comp(2,2)(cons,proj(2,1),proj(2,2))):
{x:list | typeof x is $S^{(m)} \rightarrow R$}
$\times$ {x:list | length(x)=n & $\forall$i.(1$\leq$i$\leq$n$\Rightarrow$typeof x[i] is $T_1 \times \ldots \times T_n \rightarrow S$)}$\rightarrow$
{x:list | typeof x is $T_1 \times \ldots \times T_n \rightarrow R$} where $S^{(m)} =$ S$\times \ldots \times$S m times.

Ideally, the types of the elements in the input sequence, the x[i], could depend on i. But to accomplish this we would need to consider types as values. Such a concept is provided in systems such as Nuprl [3], but we do not wish to consider such complexities here.

The lack of a rule such as * does not spoil the reflection result because there is no term in the theory which takes a list of arguments of arbitrary length as input. Instead there are an unbounded number of separate rules and separate forms. Thus for example we do not need a term such as "comp" which takes n,m and then two lists of length n and m as arguments, but we only need *comp*(n,m) as atoms for each n,m and a list of rules for each n,m. Nevertheless, we can increase the power of the calculus by adding new forms and new rules without loosing the termination property. In the next section, we sketch a method of doing this.

4.2. Internal Constructors

To obtain the capabilities mentioned in 4.1, we generalize rules T4-T9 to apply to variables. Specifically we add new functions such as these:

T'4: inconst(n): {x:list | typeof x is T_2}$\rightarrow${x:list | typeof x is $T_1 \rightarrow T_2$}

T'5: incomp(n,m): {x:list | typeof x is $S^{(m)} \rightarrow R$} $\times$
{x:list | length(x)=n & $\forall$i.(1$\leq$i$\leq$n$\Rightarrow$typeof x[i] is $T_1 \times \ldots \times T_n \rightarrow S$)}
$\rightarrow$ {x:list | typeof x is $T_1 \times \ldots \times T_n \rightarrow R$}

where $S^{(m)} =$ S$\times \ldots \times$S m times.
and so forth for T'6, T'7, ..., T'9.

These new functions are definable. For instance incomp(n,m) is the function of (**) in 4.1. But we add new names to make statement of the rules simple. The new content are rules for typing terms.

4.3. Termination Revisited

We now argue that when viewed as new type inference rules these new functions do not lead to nonterminating behavior. We can see this by noticing that the rules are supported by function definitions in the system which are known to terminate by 3.4. For instance, incomp(n) is defined by the function in (**). We can show that if we have a proof that a is in {x:list | typeof x is $T \times \ldots \times T \rightarrow R$} and a proof that b is in $\{x{:}list \mid length(x)=n$ & $\forall i.(1 \leq i \leq n \Rightarrow$ typeof x[i] is $S_1 \times \ldots \times S_n \rightarrow T)\}$, then we have a proof that incomp(n)(a,b) is in {x:list | typeof x is $S_1 \times \ldots \times S_m \rightarrow R$}. Thus for any specific inputs, the new type inferences needed can be reduced to repeated applications of the rules known to allow only terminating computations.

5. CONCLUSION

These simple results can be extended to more elaborate theories such as Nuprl [3] in which types are values. In this setting such results are quite useful. They permit meta-reasoning about the system to be conducted in it, and they allow users of the system to write proof checking and proof finding algorithms for the logic in the logic itself. This capability of writing proof manipulating programs in the logic is known from experience to be a very powerful way to raise the level of proof presentation [4,8,9].

These results also suggest purely theoretical questions about computability theory. For example, if we begin with the class P of polynomial time computable functions as the base of total functions and then we form the closure as defined in section 3, what complexity class bounds the resulting functions?

Acknowledgements

This work was supported in part by NSF grants MCS80-03349, MCS81-04018 and by the University of Edinburgh. I want to thank Todd Knoblock for a careful reading of the manuscript, Robin Milner and Gordon Plotkin for listening to me formulate these ideas and Donette Isenbarger for putting the manuscript in order.

References

[1] Constable, Robert L., Partial Functions in Constructive Formal Theories, *Proceedings of 6th G.I. Conference Lecture Notes in Computer Science,* vol. 145, Springer-Verlag, 1983.

[2] Constable, R.L. and D.R. Zlatin, The Type Theory of PL/CV3, *ACM Trans. on Prog. Lang. and Systems,* vol. 6, 1, 1984, 94-117.

[3] Constable, R.L. and J.L. Bates, The Nearly Ultimate PEARL, Cornell University Computer Science Department, TR83-551, 1984.

[4] Gordon, M., R. Milner and C. Wadsworth, *Edinburgh LCF: A Mechanized Logic of Computation, Lecture Notes in Computer Science, Vol.* 78, Springer-Verlag, 1979.

[5] Grzegorczyk, A., Some Classes of Recursive Functions, *Rozprawy Mathmatcyzne,* Warsaw, 1953, 1-45.

[6] Kleene, S.C., *Introduction to Metamathematics,* D. Van Hostrand, Princeton, 1952.

[7] Machtey, M., and P. Young, *An Introduction to the General Theory of Algorithms,* North-Holland, NY, 1978.

[8] McCarthy, John, A Basis for a Mathematical Theory of Computation, *Computer Programming and Formal System,* P. Braffort, D. Hirschberg [eds.], North-Holland, Amsterdam, 1963, 33-70.

[9] Proofrock, J.A., PRL: Proof Refinement Logic Programmer's Manual (Lambda PRL VAX Version), Computer Science Department, Cornell University, 1983.

[10] Troelstra, A.S., Metamathematical Investigations of Intuitionistic Arithmetic and Analysis, *Lecture Notes in Mathematics,* 344, Springer-Verlag, NY, 1973.