PARTIAL FUNCTIONS IN CONSTRUCTIVE
FORMAL THEORIES*

Robert L. Constable

TR 83-549
March 1983

Department of Computer Science
Cornell University
Ithaca, New York 14853

*This work was supported in part by NSF grant MCS-81-04018 at Cornell
University and in part by SERC grant GRB 59471 from Great Britain at the
University of Edinburgh, Scotland.
PARTIAL FUNCTIONS IN CONSTRUCTIVE FORMAL THEORIES*

Robert L. Constable
Cornell University

ABSTRACT

Partial functions abound in modern computing theory, and so any system which purports to naturally formalize it must treat them. Surprisingly the most common treatments do not work well for constructive formal systems, i.e., for those with computational content. Since constructive formal systems are significant in computer science, it is important to give an account of partial functions in them. This paper does that by construing a partial function $\phi: N \rightarrow N$ as a total function $f: D \rightarrow N$ for $D$ an inductively defined set generated simultaneously with $f$. This idea has appeared in other guises, at least in the author's previous work, but here it is presented in a pure form. It is compared to Scott's method of using total functions on domains. A formal system of arithmetic is defined to illustrate the ideas. The system is shown consistent relative to constructive type theory; from this result important corollaries are drawn about using the theory as a programming language.

KEY WORDS AND PHRASES

automated logic, Heyting arithmetic, constructivity, intuitionistic predicate calculus, partial functions, recursive functions, programming logics, program verification, type theory, type checking

I. INTRODUCTION

1.1 Primitive and General Recursion

In 1888 or so Dedekind considered a class of recursive function definitions over the natural numbers $N = \{0,1,2,\ldots\}$ which we now call primitive recursive. Essentially they have the form

$$f(n,x) = \text{if } n=0 \text{ then } g(x) \text{ else } h(n,f(n-1,x),x) / \text{fi.}$$

*This work was supported in part by NSF grant MCS-81-04018 at Cornell University and in part by SERC Grant GRB 59471 from Great Britain at the University of Edinburgh, Scotland.
In the early 1930's Herbrand considered a more general class of recursive definition, say of the form

\[ f(n,x) = \text{exp} \]

where \text{exp} can contain arbitrary occurrences of \( f(t_1,t_2) \) for expressions \( t_1, t_2 \). These are now called \textit{general recursive definitions} and we recognize in them the style of recursive definition allowed in programming languages such as Lisp or Algol.

The meaning of primitive recursive definitions is immediately clear. They define functions from numbers or n-tuples of numbers to numbers. But because it is possible to write general recursive definitions of the form \( f(n) = f(n+1) \), it is clear that these definitions do not denote such functions. One approach to their meaning which has been extensively studied considers them to be \textit{partial functions}, that is functions which may be \textit{undefined} at some (or all) of their arguments. The subject of Basic Recursive Function Theory (BRFT) as founded by Kleene takes the concept of partial recursive function as fundamental. One of the formative results of the theory states that there is no recursive enumeration of those partial recursive functions which are defined on all inputs, i.e., the \textit{total recursive functions}.

1.2 Formalizing General Recursion - Difficulties

Although the theory of partial recursive functions is very elegant, there are certain difficulties with formalizing it that stem from the fact that the total functions are not recursively enumerable. Essentially the difficulty is how to treat a term such as \( f(t) \) if \( f \) is not defined at \( t \). This is a special case of the linguistic issue of "terms that do not denote" such as "the king of France in 1982."

1.3 Avoiding the Problem

There are formal theories of partial recursive functions so the difficulties are not absolute. For instance any formalization of set theory will solve the problem, but it usually does so by treating \( f(t) \) as an "abuse of notation." A function is defined to be a special kind of relation, so the basic notation is that a certain pair \( <x,y> \) belongs to the relation \( f \). If \( t \) is not in the domain of \( f \), i.e., if \( f \) is not defined at \( t \), then \( <t,y> \) does not belong to \( f \) for any \( y \).
Kleene showed how to formalize BRFT in elementary number theory in terms of the relation \( f(x) \approx g(x) \) which means "if either \( f(x) \) or \( g(x) \) is defined then so is the other and their values are the same." But again the term \( f(x) \) is a fiction in the theory and does not have meaning itself.

1.4 Extensions to Total Functions

One way to treat \( f(t) \) when \( f \) is undefined at \( t \) might be to regard it as some arbitrary number, about which we know nothing. In this view the definition \( f(x) = \exp \) denotes any of a class of total functions consistent with \( f(x) = \exp \) when \( f(x) \) is defined. Classically this interpretation is acceptable (see [2]), and it obviates complex type checking. But constructively the interpretation is flawed since from the equality \( f(x) = f(x) \) for any \( x \) in \( N \), we can deduce \( \exists y, (f(x) = y) \). From this we expect to be able to calculate the number \( y \), giving the value of \( f(x) \) which is not possible for \( x \not\in D_f \).

Another classical approach to this problem is to extend the partial function to a unique total function. These extensions are not always very useful, e.g., as in the case of \( 1/x \), but they solve the problem of type checking. However constructively not even this option is possible because there are partial recursive functions which cannot be extended to any total recursive function.

1.5 Extended Domains

John McCarthy who in the early 1960's was laying the foundations for a theory of general computer programs recognized the difficulties caused by partial functions, [12]. In the early 1970's Dana Scott forcefully advocated a theory of partial functions obtained by extending the domain of numbers, for instance, to include a special element, called bottom, written \( \bot \). One says that \( f(t) \) is undefined by writing \( f(t) = \bot \). Now the term \( t(n) \) has precise meaning for all numbers \( n \). This approach as been extensively pursued \([4,5,10,13,17]\) and has been rigorously formalized in the LCF system of Robin Milner and his colleagues [7].

One difficulty with the McCarthy, Scott, Milner, etc. approach is that the new element \( \bot \) complicates the basic theory, say in this case that of the extended natural numbers \( N^+ = \{ \bot, 0, 1, 2, \ldots \} \). Another difficulty is that unlike the Kleene approach, this theory is not constructive as written, i.e., where \( \bot \) is considered a canonical element of \( N^+ \). That is, given a partial
recursive function $\phi : N \rightarrow N$, its natural extension to $\phi^+ : N^+ \rightarrow N^+$ obtained by putting $\phi^+(n) = \bot$ if $\phi(n)$ is undefined is not effectively computable. It is possible to interpret such a theory constructively as Scott advocated [15] or as done by Herbert Egli and R. Constable [5]. Indeed LCF stands for "Logic of Computable Functions." But the conceptual cost of the constructive theory of $N^+$ is high, e.g., the basic objects are not numbers but computations of numbers and the basic equality is no longer decidable, so $N^+$ is not discrete, and the notion of a computation returning a value involves fundamentally the idea of approximations, even in the natural number case.

1.6 Inductively Defined Domains

A common informal way of dealing with partial function such as $1/x$ in algebra and analysis is to regard them as total functions on a subset of their apparent domain, e.g., $1/x$ maps from the subset of nonzero reals to reals. In the case of a general recursive definition $f(x) = \exp$ we can find uniformly an inductively defined subset of $N$, $D_f$, whose members are precisely those numbers on which $f$ is defined, $D_f$ may be empty. The definition $f(x) = \exp$ determines a unique function $f : D_f \rightarrow N$. This function we can take to be the meaning of the definition, and we call $D_f$ the domain of the function.

Although we know what $f(x) = \exp$ denotes and we have granted $f(t)$ real status for $t \in D_f$, there remains the question about the meaning of $f(t)$ when $t \notin D_f$. We might say that $f(t)$ is not a well-defined term and has no semantic content. Thus the formula $\forall x. (P(x) \Rightarrow f(x) = f(x))$ might be meaningless unless $P(x) \Rightarrow D_f(x)$. However such an approach would mean that knowing whether a term or formula is well-defined may be an extremely difficult mathematical problem in some cases, because one would have to know in general whether $P(x) \Rightarrow D_f(x)$ for arbitrary predicates $P$. This is not the customary way of treating syntactic issues in formal systems, especially those of programming languages. On the contrary, custom dictates that such issues are efficiently decidable. Are we to view this custom as mere happenstance or as well learned principle?

1.7 Outline of the Paper

In section II we define a theory called General Recursive Arithmetic, GRA, which treats partial
functions $f$ as mappings from $D_f \rightarrow N$. The theory is presented in such a manner that it can easily be generalized to a type theory which we consider to be the type theory generated by GRA.

In section III we show that all partial recursive functions are definable in GRA. In section IV we interpret GRA in the Intuitionistic Type Theory (ITT) of Per Martin-Lof thereby revealing GRA's constructive content. This involves giving a type theoretic treatment of least fixed points.

II. GENERAL RECURSIVE ARITHMETIC

2.1 Syntax Conventions

The syntax is presented using BNF with the following conventions. A term such as "proof" will denote a particular element of the syntactic category "Proof." After each definition we list the metavariables used to denote arbitrary members of the class when we want compact notation, e.g., $t, t_1, t_2, \ldots$ for terms. A vertical bar, $|$, is used to separate clauses.

The disjoint union of syntax classes $A$ and $B$ is $A + B$, and the ordinary union is $A \cup B$.

We assume that identifiers (id and Id) comprise at least finite sequences of upper and lower case letters and numbers. But we may also allow other alphabets and special symbols. We also assume that the concept of a list of elements from a set $A$, written $A$ list, is understood. Lists are written $(a_1, a_2, \ldots, a_n)$ with $a_n$ the head and $(a_1, \ldots, a_{n-1})$ the tail; concatenation is denoted by juxtaposition, i.e., $(a_1, a_2, a_3, a_4) = (a_1, a_2, a_3) a_4$.

2.2 Syntax Equations

1. Let Nexp abbreviate "numerical expression."

   $\text{Nexp ::= id | id(Nexp list) | (nexp \rightarrow nexp, nexp) | (nexp \text{ id nexp})}$

   $t, t_1, t_2, \ldots$ denote numerical expressions. The expression

   $(t \rightarrow t_1, t_2)$ is a form of conditional, if $t = 0$ then $t_1$ else $t_2$.

2. Constants ::= a certain list of reserved identifiers such as $+, *, -, /, <, >, 0, 1, N, \text{false}$, etc.

3. For now we put $\text{Term ::= Nexp.}$
4. Atomic formula ::= id | $t_1 = t_2$ | id(Term list) | $D_{id}(Term list)$ | $t_1 \operatorname{id} t_2$

We call id and $D_{id}$ as they occur here predicate names.

P, $P_1$, $P_2$,... are used to denote predicate names.

5. Formula ::= atomic formula | formula & formula | formula $\lor$ formula

$\Rightarrow$ formula $\Rightarrow$ formula | $\forall Id list.formula$ | $\exists Id list.formula$ | (formula)

F, $F_1$, $F_2$,... denote formulas.

The operators have their usual precedence: $\{\forall, \exists\}, \&, \lor, \Rightarrow$

($\Rightarrow$ is right associative), so $\forall z. z = 0 \lor P(x) \Rightarrow Q \& R \Rightarrow S$ is

$(\forall z. (z = 0) \lor P(x)) \Rightarrow ((Q \& R) \Rightarrow S)$.

6. We say that a formula F is positive in predicate names $P_1,...,P_n$, written $\mathrm{Pos}(P_1,...,P_n)$, iff none of the $P_i$ occur in it or if it has the form $P_i(Term list)$ or if A and B are $\mathrm{Pos}(P_1,...,P_n)$ and F is $A \& B$, $A \lor B$, $\forall Id list.A$ or $\exists Id list.B$ or if A is $\mathrm{Neg}(P_1,...,P_n)$ and B is $\mathrm{Pos}(P_1,...,P_n)$ and F is $A \Rightarrow B$.

We say that F is negative in predicate names $P_1,...,P_n$, written $\mathrm{Neg}(P_1,...,P_n)$ iff none of the $P_i$ occur in it (in which as it is positive also) or if A and B are $\mathrm{Neg}(P_1,...,P_n)$ and F is $A \& B$, $A \lor B$, $\forall Id list.A$, $\exists Id list.B$ or if A is $\mathrm{Pos}(P_1,...,P_n)$ and B is $\mathrm{Neg}(P_1,...,P_n)$ and F is $A \Rightarrow B$.

Note, some formulas such as $P_1(x) \Rightarrow P_1(x)$ are neither $\mathrm{Pos}(P_1)$ nor $\mathrm{Neg}(P_1)$.

7. Let Idt denote typed identifiers defined as

Idt ::= id:formula

8. Proof ::= id | id(Proof list) | $\lambda$ Idt list.proof

The necessary reserved identifiers and the type constraints on proofs will be defined later. We will also discuss the use of lambda terms.

9. Let "Fd line" abbreviate "function definition line."

Fd line = id(Id list) = term | id id id = term

We call id(Id list) the left side or the definiendum and "term" the right side or definiens. We call the underlined id the defined function (symbol), it can occur as an infix binary operator.
10. Function definition ::= \textit{def} (Pd line) list \textit{fed}

\textit{def} id \textit{from} \forall \textit{Id list} \exists \textit{id.formula fed}

In the first clause each defined function can occur only once in a left hand side.

11. Let Pd line abbreviate "predicate definition line."

Pd line ::= \textit{id(Id list)} = \textit{formula} | \textit{id id} = \textit{formula}

We call underlined id the \textit{defined predicate}

12. Predicate definition ::= \textit{def} (Pd line) list \textit{fed} where each defined predicate occurs exactly once on the left side and each formula on the right is \textit{positive in the defined predicates.}

13. Definitions ::= Predicate definition \textbf{+} Function definition

14. Claim ::= \textit{formula} \textit{by proof}

We discuss later \textit{valid claims.}

15. Book ::= (Definition \textbf{\textit{\cup}} Claim) list

We discuss below a \textit{correct book} which is one for which all definitions are well-formed and all claims are valid.

In the type theory section we will allow Term = Nexp + Proof. We will also replace Id list in 5, 6, 9 and 11 to be a typed id list. With these changes we will be able to define constructive type theory.

2.3 Constants

The only constants that we really need for arithmetic are \textit{zero}, 0, and \textit{successor} s and the constants for proofs of 2.6. But the theory has a pleasing form if we also allow the formula \textit{false} and the formula N used to denote "truth."

We also reserve certain symbols for infix operators to be defined later, namely + , *, /.

2.4 Well-Formed Terms and Well-Formed Definitions

Every term such as \textit{f(t) occurs in a specific context which includes definitions, theorems and variables. Indeed each term has a unique address consisting of the line number of the book in which}
it occurs plus the "decimal address" in the tree representation of the formula or proof in which it occurs.

Given a particular occurrence of a term (at an address), we can list all of the theorems that are written up to that point and all of the assumptions which "govern" that occurrence. For example in these lines

\[
\begin{align*}
    f(2) &> 0 \\
    \forall x.(x > 0 \Rightarrow x^2 > 0) \\
    \forall x.(x > 0 \Rightarrow f(2)/x > 0)
\end{align*}
\]

if we examine the occurrence of \(f(2)/x\), it is governed by the theorem \(f(2) > 0\) and the assumption \(x > 0\).

In order to state the conditions under which an occurrence of a term is well-formed, we must be able to say precisely what its context is. For each occurrence of a term in a book, say specified by address \(p\), we define its context which consists of all theorems previous to the line in which \(p\) occurs plus all assumptions which govern \(p\) in its line and all variable bindings in whose scope \(p\) lies. To define those formulas which govern a position, denoted \(\text{Gov}(p,A)\), let \(A_p\) denote a formula \(A\) which contains the occurrence at address \(p\). Let \(\text{new}(x)\) denote a variable name which has not been used previously in any formula up to the line we are examining.

\[
\begin{align*}
    \text{Gov}(p, A_p \& B) &= \text{Gov}(p, A_p), \text{Gov}(p, B \& A_p) = \Gamma(B) \cup \text{Gov}(p, A_p) \\
    \text{Gov}(p, A_p \lor B) &= \text{Gov}(p, A_p) = \text{Gov}(p, B \lor A_p) \\
    \text{Gov}(p, A \Rightarrow B_p) &= \Gamma(A) \cup \text{Gov}(p, B_p), \text{Gov}(p, A_p \Rightarrow B) = \phi \\
    \text{Gov}(p, \forall x. A_p) &= \{\text{new}(x)\} \cup \text{Gov}(p, A[\text{new}(x)/x]) \\
    \text{Gov}(p, \exists x. A_p) &= \{\text{new}(x)\} \cup \text{Gov}(p, A[\text{new}(x)/x]) \\
    \Gamma(A \& B) &= \Gamma(A) \cup \Gamma(B) \quad \Gamma(A \lor B) = \{A \Rightarrow B\} \cup \Gamma(t) \text{ for all terms of } A \Rightarrow B \\
    \Gamma(P(t)) &= P(t) \cup \Gamma(t) \\
    \Gamma(f(t)) &= D_f(t) \cup \Gamma(t)
\end{align*}
\]

Let \(A\) be \(\forall x.B\) or \(\exists x.B\) and let \(t\) be any term of \(A\) which contains no bound variable, then \(\Gamma(A) = A \cup \Gamma(t)\).

Given any set of formulas and variable bindings \(F\), we define its immediate closure \(F\) inductively as follows (writing \(d \in F\) if \(d\) is any free term appearing in a formula or binding of \(F\)):
(i) \( F \subseteq F \)
(ii) if \( A \in F \), \( A \Rightarrow B \in F \) then \( B \in F \)
(iii) if \( A \& B \in F \) then \( A \in F, B \in F \)
(iv) if \( d \in F \) and \( \forall x.A(x) \in F \), then \( A(d) \in F \)
(v) if \( t\Rightarrow t' \in F \), \( A(t) \in F \), then \( A(t') \in F \)
(vi) and if \( t\Rightarrow t' \in F \) and \( t'\Rightarrow t'' \in F \), then \( t'' \Rightarrow t \in F \)

Fact: If \( F \) is finite, say \( n \) elements (largest of depth \( m \)) then \( F \) is finite with at most \( O(n, \max(n, m)) \) elements.

We say that a function term \( f(t) \) is well-formed at occurrence \( p \) in \( A \) iff \( \text{Gov}(p, A) \) contains \( D_f(t) \).

well-formed definitions

A definition is well-formed at a line in a book if and only if any functions or predicates which occur on the right but not on the left are previously defined in the book and each occurrence of them in the definition is well-formed and moreover no variable occurs on the left that does not occur on the right of the same equation. (All occurrences of the defined function on the right are considered well-defined.)

2.5 Domains

The key to treating partial functions in this system is the simultaneous definition of functions \( f \) and their domains \( D_f \). The domain definition can be generated automatically from the function definition. The rules for doing this are given here inductively. They are stated for one argument functions but generalize to \( f(x_1, ..., x_n) \) trivially. Also for simplicity we assume that \( f \) is defined in terms of \( g_i \) which are defined or base functions. (Sometimes for typographical simplicity we write \( g_i \) as \( g_i \).)

Let \( N(x) \) be the predicate meaning "\( x \) is a nonnegative integer," i.e., \( x = x \) since there is only one type.

1. \( D_1(x) = N(x) \)
\( D_1(x, y) = N(x) \& N(y) \) for \( c = *, +, - \)
\( D_1(x, y) = N(x) \& y \neq 0 \)
2. If \( t(x) = h(g,1(x), \ldots, g_n(x)) \) then
\[
D_f(x) = D_h(g,1(x), \ldots, g_n(x)) \& D_{g,1}(x) \& \ldots \& D_{g_n}(x)
\]

3. If \( t(x) = (b(x) \rightarrow g_1(x), g_2(x)) \) then
\[
D_f(x) = D_h(b(x) \rightarrow 0 \& D_{g,1}(x) \lor b(x) \neq 0 \& D_{g,2}(x))
\]

These three clauses will be used in giving axioms for the system. Notice that all the formulas on the right side are positive in \( D_f \).

2.6 Axioms and Rules

We present a Hilbert style proof system based on three rules of inference (two of which express essentially one idea, application) and several axiom schemes listed below. For each axiom we also provide a function which is its constructive realization. Proofs are terms built from these realizing primatives. Each proof has a type which is the formula that it proves.

To state some of these rules we need the usual concepts of free and bound variables, scope of quantifiers and definitions and the idea of a term being free for a variable in a formula. We assume all of these definitions as presented in Kleene [9].

Inference Rules

\[
\begin{align*}
\text{application} & : A, A \Rightarrow B & \forall z.A(z) \text{ t a well-formed term} & : A(t) \\
\text{existential} & : E_A & A(t) \text{ t a well-formed term} & : \exists z.A(z)
\end{align*}
\]

Axioms - propositional

\[
\begin{array}{lll}
\text{realizing term} & \text{name} & \text{axiom} \\
K_{AB} & \text{constants} & A \Rightarrow (B \Rightarrow A) \\
S_{ABC} & \text{composition} & (A \Rightarrow (B \Rightarrow C)) \Rightarrow (A \Rightarrow B) \Rightarrow (A \Rightarrow C) \\
P_{AB} & \text{pairing} & A \Rightarrow B \Rightarrow A \& B \\
1_{AB} & \text{1st projection} & A \& B \Rightarrow A \\
2_{AB} & \text{2nd projection} & A \& B \Rightarrow B \\
L_{AB} & \text{left injection} & A \Rightarrow A \lor B \\
R_{AB} & \text{right injection} & B \Rightarrow A \lor B \\
N_A & \text{nil} & (0=1) \Rightarrow A \\
C_{ABC} & \text{cases} & (A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow (A \lor B \Rightarrow C)
\end{array}
\]
axioms - predicate

\[ \lambda f. \lambda z. (\lambda z.f(z)) \text{ generalization } (C \Rightarrow A(z)) \Rightarrow (C \Rightarrow \forall z.A) \]
C has no free x

\[ W_{AC} \text{ witness } \exists x. A(x) \Rightarrow (\forall z. (A(z) \Rightarrow C) \Rightarrow C) \]

axioms - equality

ref reflexivity \[ \forall z.(z=z) \]
tran transitivity \[ \forall z,y,z. (z=y \Rightarrow y=z) \Rightarrow z=z \]
com commutativity \[ \forall z,y. (z=y \Rightarrow y=z) \]

axioms - functions for \( f \) a defined function

def definition \[ \forall z. (D_f(z) \Rightarrow f(x) = \exp(x)) \]
fun\(_f\) functionality \[ \forall z,y (z=y \land D_f(z) \Rightarrow f(x) = f(y)) \]
cond conditional \[ \forall x. (b(x)=0 \Rightarrow (b(x) \rightarrow f(x),g(x)) = f(x)) \land \]
\[ b(x) \neq 0 \Rightarrow (b(x) \rightarrow f(x),g(x)) = g(x)). \]

We need also the schemes 2. and 3. for \( D_f \) from section 2.4 as axioms about functions.

For \( D \) an inductively defined predicate defined by \( D(x) = F(D,x) \)

\[ D_{\text{def}} \text{ D-definition } \forall z. (D(z) \Rightarrow F(D,z)) \]
\[ D_{\text{ind}} \text{ D-induction } \forall z. (F(P,z) \Rightarrow P(z)) \Rightarrow \forall z. (D(z) \Rightarrow P(z)) \]

There are axioms for functions of \( n \) arguments as well.

axioms - arithmetic

\[ \text{dom}_a \text{ domain-s } \forall z. (D_a(z) \Rightarrow z=z) \]
\[ \text{fun}_a \text{ functionality-s } \forall z,y. (z=y \Rightarrow s(z)=s(y)) \]
\[ \text{inj}_a \text{ injectivity-s } \forall z,y. (s(z)=s(y) \Rightarrow z=y) \]
\[ N_{\text{ind}} \text{ N-induction } A(0) \land \forall z. (A(z) \Rightarrow A(s(z))) \Rightarrow \forall z. A(z) \]

For brevity we omit the axioms for +, -, *, and / since these functions can all be defined from successor.

III. REPRESENTATION OF THE PARTIAL RECURSIVE FUNCTIONS

3.1 Syntactic Representation

Suppose we have defined the partial recursive functions from recursion equations as in [9]. These definitions have precisely the same form as those allowed in the theory. So there is an isomo-
phism from a recursion equation, say $E_F$, defining a partial recursive function $F(\ )$ and a function
definition say $def$. We will use this notation generally, allowing $F$ to denote the principle function
letter defined by the equation $E_F$ and letting $f$ be the corresponding identifier of the formal system.

We claim that the computation rules of the system permit the deduction $f(n)=m$ precisely
when $F(n)=m$. To prove this we must show that we can deduce $D_f(n)$ if and only if $n$ is in the
domain of $F(\ )$. We turn to this next.

3.2 Semantic Equivalence

We let $\vdash A$ mean that $A$ is provable in the system. We use the same notation for numbers and
their numerical representation in the system. We can prove:

Theorem: (a) For all $n$ and recursively defined $F$,

$$F \text{ is defined at } n \text{ iff } \vdash D_f(n)$$

(b) $F(n)=m$ iff $\vdash f(n)=m$.

Proof:

We are looking only at a single function definition given by equations $E_F$ which is represented
in the system as

$$def f(x) = \exp(x) fed$$

We assume that the definition is well-formed so that all functions occurring in $\exp$, except for $f$,
are previously defined. For all of these functions we assume the result because we can easily prove it
for the base function $s$, and we can treat definitions as they appear in order in a book (i.e., we are
doing an induction on the number of definitions in a particular book).

We proceed by induction on the number of recursive calls of $F$ in any computation.

Base Case:

Consider all those numbers $b$ such that $F(b)$ is computed without any calls to $F$. If there are
not such $b$ then $F$ is nowhere defined. For these $b$ the value of $F(b)$ is computed using only previously
defined functions, say $g_i$. 
Since the computation of \( F(b) \) does not call \( F \), the proposition \( D_f(b) \) can be proved without reference to \( D_f(x) \), that is, given proofs of \( D_k(y_j) \) for the appropriate numbers \( y_j \), we can build a proof of \( D_f(b) \). Thus part (a) is proved. By axiom def-f we can prove \( f(b) = \exp(b) \) and by the isomorphism with \( F \) we obtain that \( \exp(b) = F(b) \) which proves part (b).

**Induction Case:**

Assume results (a) and (b) for all those integers \( x \) for which the computation of \( F(x) \) is defined and requires \( k \) calls to \( F \).

Consider now a value \( n \) such that \( F(n) \) is defined and the computation requires \( k + 1 \) calls. Suppose that in the computation of \( F(n) \), the first equation requires the computation of \( F(z_1), \ldots, F(z_\ell) \) as well as various values \( g_i(y_j) \). Then since the computations of the \( F(z_i) \) must all terminate and can require at most \( k \) steps, we know

\[
D_f(z_i) \text{ are provable and} \quad f(z_i) = F(z_i).
\]

Now consider the form of \( F \).

**composition cases:** \( F(x) = h(\epsilon_1, \ldots, \epsilon_\ell) \)

Then \( D_f(x) = D_k(\epsilon_1, \ldots, \epsilon_\ell) \& G \) where the \( G \) term contains \( D_f(z_i) \) and various \( D_k(y_j) \) each of which is provable.

Since \( h \) is previously defined and since \( F(x) \) is defined, we know that the \( \epsilon_i \) reduce to integers and that \( h \) is defined on them. So \( D_f(x) \) is provable.

**conditional case:** \( F(x) = b(x) \rightarrow \epsilon_1(x), \epsilon_2(x) \)

Since \( F(x) \) is defined, so is \( b(x) \) and any occurrence of \( F(z_i) \) in \( b(x) \) requires only \( k \) reductions.

Therefore we know that \( D_k(x) \) is provable. By an analysis like that in the conditional case we know that \( D_f = D_k(x) \& (D_1(x) \lor D_2(x)) \) and one of \( D_1(x) \) or \( D_2(x) \) is provable. So \( D_f(x) \) is provable.

We conclude \( f(x) = F(x) \) as before.

**QED**
IV. INTERPRETATION IN TYPE THEORY

4.1 Constructive Soundness

To know that a theory is constructively sound we might provide a computational semantics for it. We might do this informally or by interpreting it in a known constructive theory. If the known theory is implemented, then the interpretation can provide an implementation of the given theory. We say that a constructive theory is implemented when there is a computer program which will execute all of the functions definable in the theory and all of the functions which are proofs.

In this section we will interpret GRA in Per Martin-Löf's Intuitionistic Theory of Types, ITT-79 [11], a theory which is being implemented. We will also consider interpreting GRA in the closely related type theory V3 [3] which takes an intensional view of functions and this permits an especially simple treatment of partial functions.

4.2 General Recursive Arithmetic Type Theory

There is a surprisingly simple way to interpret GRA in type theory. The first step, which we consider in this section, is to define a type theory which we characterize as the type theory generated from GRA by the propositions as types principle. The second step is to map this "arithmetic type theory", denoted GRATE, into Martin-Löf's ITT-79. The interesting step is the treatment of recursive functions and inductive definitions. In this paper we have room to treat only the first notion.

The syntactic changes needed to define a type theory from GRA are very simple.

1. Let Term = Nexp + Proof

2. In lines 5, 6, 9 and 11 replace Id list by typed Id list, that is sequences of id:formula.

With these changes we must consider type-correctness as well as well-formedness of formulas. Indeed we must worry about whether a type expression is well-formed, which is not a common concern in logical systems.

A formula F, whether it occurs in type position, \( \forall z:F \), or an assertion position, \( \forall z:T.F \), will be well-formed under the same conditions as in GRA except that when assuming a formula such as \( \forall z:T(y).A(z) \) or \( \exists z:T(y).A(z) \) we assume \( T(y) \) is well-formed. This means we define an operation
\( \Gamma_t \) similar to \( \Gamma \) except that in forming \( \Gamma(\forall z : T(y) . A) \) or \( \Gamma(\exists x : T(y) . A) \) we take \( \Gamma_t(T(y)) \cup \{ \text{new}(x) : T(y) \} \cup \Gamma(A(\text{new}(x)/(z))) \).

For the purposes of type checking the formula \( N \) is considered to be the type of nonnegative integers and \textit{false} is the empty type. So the numerical constants have their obvious types: \( 0 : \text{N} \), \( s(t) : \text{N} \) for \( t : \text{N} \). We also agree that the proof constants have as type the formulas which they prove according to the table of section 2.6. Moreover we interpret \( A \Rightarrow B \) as the function \( A \rightarrow B \), and \( A \& B \) as the cartesian product \( A \times B \); so \( s \) has type \( N \rightarrow N \) and defined recursive functions \( f(n, x_1, \ldots, x_p) \) have type \( N \& N \& \ldots \& N \Rightarrow N \) where \( N \& \ldots \& N \) means \( N \& \ldots \& N \) for \( N \) repeated \( p \) times.

The universal quantifier \( \forall z : T \) is interpreted as the \( \Pi z \in T \) operator of type theory, so we have also the type checking rule that if \( f : (\forall z : T . A(z)) \) and \( t : T \), then \( f(t) : A(t) \). \( \exists x : T \) is interpreted as \( \Sigma x \in T \).

Here are some examples of formulas in this type theory which we call GRATE for General Recursive Arithmetic TypE.

1. \( \exists f : N \rightarrow N . \forall x : N . f(x) = f(y) \).
2. \( \exists x : (\exists z : N . A(z) \Rightarrow N) . \forall x : (\exists z : N . A(z)) . A(f(x)) \).

Number 2 says that there is among the functions from \( \exists x : N . A(x) \) into \( N \) one which is the witness function for the quantifier.

Notice that this is a formula scheme because we have not specified \( A \), but \( A \) must be a formula which is well-formed as a function of \( x \), so it cannot for example be \( (1/x = 1/x) \).

In this report we do not consider the axioms of GRATE in detail, but they include reduction axioms for terms, such as \( 1(P(x, y)) = x \), \( 2(P(x, y)) = y \), \( Kxy = x \), etc., and they include axioms relating propositions and types such as \( \exists x : A(x = a) \Leftrightarrow A \) as well as the axioms of GRA.

4.3 Partial Functions and Their Domains in Intuitionistic Type Theory

From the discussion in 4.2 it is clear how we are going to translate into ITT-79. We will take \textit{false} to be the empty type, \( \phi \), \( N \) to be the type of natural numbers, \& as cartesian product, \( \forall \) as
disjoint union, \( \exists \) as \( \Sigma \) the infinite union and \( \forall \) as \( \Pi \) the infinite product. But how will we translate partial functions \( f \) and their domains? How will we translate inductive definitions?

In this section we treat partial functions and their domains assuming the translator of proofs mentioned above. Given a function definition such as \( f(x) = \text{exp} \) we want to treat the right side as a functional, say \( \lambda f. \lambda x. \text{exp} \). But if we do this without care \( \lambda f \lambda x. \text{exp} \) will not be type correct. To insure that such a functional makes sense we must be able to determine the domain of numbers \( x \) for which \( \lambda x. \text{exp} \) is defined as a function of the domain of \( f \). In order to do this, we must supply the functional domain information about \( f \). This we do by providing a predicate \( F \) of type \( N \rightarrow V_1 \).

The domain of \( f \) will be \( \{ x : N | F(x) \} \). So we will build a mapping denoted \( \mathcal{E} \) from \( F \) and \( f \) to a new predicate, denoted \( D(F,f) \) and a new function denoted \( E(F,f) \). We write \( \mathcal{E}(F,f) = <D(F,f), E(F,f)> \).

The basic idea is that we define a function \( \mathcal{E} \) from predicates and functions \( F,f \) such that \( f : \{ x : N | F(x) \} \rightarrow N \) whose value is a pair \( <D(F,f), E(F,f)> \) such that \( E(F,f) \epsilon \{ x : N | D(F,f)(x) \} \rightarrow N \).

We use \( \mathcal{E} \) to define a sequence of predicates and functions starting with \( F_0 = \text{false}, D_0 = \{ x : N | F_0(x) \}, F_0 \epsilon D_0 \rightarrow N \). We take \( F_{i+1} = D(F_i, f_i), D_i = \{ x : N | F_{i+1}(x) \}, f_{i+1} = E(F_i, f_i) \). The fixed point is defined by taking a limit of the \( f_i \) which is essentially but not exactly

\[
F_\omega(x) = \exists n : N. F_n(x) \\
f_\omega(x) = f_n(x) \text{ where } F_n(x).
\]

The details of how this is done in type theory are somewhat tedious. In the first place, if we are to take a general approach to this, then we allow arbitrary predicates \( F : N \rightarrow V_1 \). But this means that \( \mathcal{E} \) and consequently \( F_\omega \) and \( f_\omega \) will not be of small type (of level \( V_1 \)) but will be of level \( V_2 \).

Another complication is that each \( f_i \) maps from \( \{ x : N | F_i(x) \} \) which in type theory is \( \Sigma x \epsilon N. F_i(x) \), i.e., a type of pairs. So we must keep track of the proof component of \( x \). Our convention will be that \( x_1 \) is the numerical component and \( x_2 \) is the proof; so \( x = <x_1, x_2> \). We will have to iterate to form \( x_{1,2} \) etc.

Here are the details of the translation. Assign to each defined function \( f \) of GRA a level as follows. The level of each expression containing only base functions is 0. The level of a recursive defin-
ition \( f(x) = \exp \) is one greater than the level of \( \exp \).

Let \( S = \Sigma F \epsilon (N \rightarrow V_1)((\Sigma x \epsilon N.F(x)) \rightarrow N) \). Given a well-formed expression \( \exp \) of GRA with free \( f \), that is with \( f \) a function symbol not previously defined, we want to associate with \( \exp \) a mapping \( \mathcal{E} : S \rightarrow S \). Given \( F \) we will treat each occurrence of \( f \) in \( \exp \) as a mapping from \( \Sigma n \epsilon N.F(n) \) to \( N \). To do this we must regard each occurrence of \( f(t) \) in \( \exp \) as an occurrence of \( f(<t,p>) \) where \( p \) proves that \( t \) satisfies \( F \), i.e., \( <t,p> \epsilon \Sigma n \epsilon N.F(n) \). We know that such \( p \) must exist because we build the domain of \( E(F,f) \) to guarantee this. We show inductively how to build the appropriate functional expression \( E(F,f) \) once we have built its domain \( D(F,f) \).

The key point in making this work is that we regard \( \mathcal{E} \) as a function which computes from \( \exp \) the domain on which \( E(F,f) \) is defined. Thus if we consider an expression such as

\[
b(x) \rightarrow g(x), h(f(j(x)))
\]

then if \( b(x) \) is false but \( f \) is not defined at \( j(x) \), i.e., if \( b(x) \) is false, yet \( F(j(x)) \) is not provable, then \( x \) is not in the domain of \( \mathcal{E}(F,f) \). Whenever \( x \) is in the domain \( D(F,f) \), we will be able to extract from the proof of \( D(F,f) \) the necessary proof that \( F(j(x)) \). Define from well-formed \( \exp \) with free \( f \) a mapping \( \mathcal{E} \) from \( S \) to \( S \) as follows for \( F \epsilon (N \rightarrow V_1) \) and \( f \epsilon (\Sigma x \epsilon N.F(x)) \rightarrow N \). \( \mathcal{E}(F,f)(x) \) will be \( <D(F,f)(x_1), E(F,f)(x)> \) for \( E \) and \( D \) as below:

- if \( \exp \) is \( g(n) \) for \( g : N \rightarrow N \)
  then put \( D(F,f)(n) = D_g(n) \). Note \( E(F,f) \) is \( \lambda x.g(x_1) \).

- if \( \exp \) is \( h(t) \) for \( h \epsilon \Sigma x \epsilon N.H(x) \) and \( t \) an expression such that \( H(t) \), then assume \( t \) has been translated by the procedure to have the form \( T(F,f)(x) \) for \( T(F,f) = <D_T(F,f), T(F,f)> \)
  then \( D(F,f)(x_1) = D(F,f)(n) = \Sigma q \epsilon E_T(F,f)(n) \). \( H(T(F,f)<n,q>) \) and \( E(F,f) = h(<t,p>) \)
  where \( p \) is the translator of the proof of \( H(t) \).

- if \( \exp \) is \( f(t) \), then assume \( t \) has been translated say \( T(F,f)(x) = <D_T(F,f), T(F,f)> \) so that \( t = T(F,f)(x) \), then put \( D(F,f)(n) = \Sigma q \epsilon D_T(F,f)(n) \). \( F(T(F,f)<n,q>) \) and \( E(F,f)(x) = f(<t(x_1),p(x)>) \) where \( p \) extracts from \( x \epsilon \{ y : N \} | D(F,f)(y) \} \) the proof of \( F(t) \), i.e.,
p(x) is the second component of z₂.

(We use t(z₁) to denote t with z₁ for x.) Notice that E(F,t) ∈ {y:N|D(F,f)(y)} → N.

if exp is b → t₁, t₂ and b, t₁, t₂ have been translated so that

\[
\begin{align*}
  b &= B(F,f)(x) \text{ with domain } D_B(F,f) \\
  t₁ &= T₁(F,f)(x) \text{ with domain } D_T₁(F,f) \\
  t₂ &= T₂(F,f)(x) \text{ with domain } D_T₂(F,f)
\end{align*}
\]

then D(F,f)(n) = D_B(F,f)(n) & (b=0 & D_T₁(F,f)(n)) v b=1 & D_T₂(F,f)(n)) and E(F,f)(x) = B(F,f)(x) → T₁(F,f)<x.1,p>, T₂(F,f)<x.1,q> where p selects the proof of D_T₁(F,f)(x.1) and q the proof of D_T₂(F,f)(x.1).

Example

Suppose we have defined the functions zero(x)=0 iff x=0, one(x) which is 0 iff x=1 and even(x)=0 iff x is even. Then in GRA we can define the well-known "3x+1" function by the line

\[
f(x) = (\text{one}(x) → 0, \text{even}(x) → f(x/2), f(3x+1))
\]

The function \( \mathcal{E} \) corresponding to this definition is built as follows.

\[
D(F,f)(n) = D_{\text{one}}(n) & (\text{(one}(n)=0 & D_{\text{zero}}(n)) v (\text{one}(n)\neq0 & D_T(F,f)(n))
\]

where

\[
D_T(F,f)(n) = D_{\text{even}}(n) & [(\text{even}(n)=0 & \Sigma q ∈ N . F(<n/2,q>.1)) \]
\[
(\text{even}(n)\neq0 & \Sigma q ∈ N . F(<3n+1,q>.1))]
\]

for p ∈ D(F,f)(n)

\[
E(F,f)(<n,p>) = \text{one}(<n,p>.1)→\text{zero}(<n,p>.1), \text{even}(<n,p>.1)→f(<n,p>.1/2), f(3(<n,p>.1)+1)
\]
\[
= \text{one}(n) → \text{zero}(n), \text{even}(n) → f(n/2), f(3n+1)
\]

Notice that p does not enter into the computation of the value E(F,f)(<n,p>), but without p it is not possible to evaluate E(F,f). Thus before we can compute f(n) we must know that f terminates on n.

We now define the fixed point apparatus.

**Definition:** Given \( \mathcal{E} ∈ S → S \) define
\[ F_0(n) = \text{false}, \quad D_i = \sum_{n \in N} F_i(n), \quad f_{\omega} \in D_{\omega} \Rightarrow N, \]
\[ F_{i+1}(n) = D(F_i, f_i)(n), \quad f_{i+1} = E(F_i, f_i) \]
\[ F_{\omega}(n) = \sum_{m \in N} F_m(n) \]

We define \( f_{\omega}(x) = f_{zz,1}(<x_1, x_{z2,2}>) \). In order to show that \( f_{\omega} \) is a well-defined function on \( N \) we need to prove:

**Lemmas:**

1. For all \( n \), \( F_i(n) \Rightarrow F_{i+1}(n) \).

**Definition:** If \( F(n) \Rightarrow G(n) \) for all \( n \), and if \( f \in \sum_{n \in N} F(n) \Rightarrow N \), \( g \in \sum_{n \in N} G(n) \Rightarrow N \) and if \( f(<n, p>) = g(<n, p']) \) for all \( n \) such that \( <n, p> \in \sum_{i \in N} F(i) \), \( <n, p'> \in \sum_{i \in N} G(i) \) then we write \( f \subseteq g \) and \( <F, f> \subseteq <G, g> \). If \( <F, f> \subseteq <G, g> \) and \( <G, g> \subseteq <F, f> \) then we write \( <F, f> \equiv <G, g> \). In this circumstance we allow writing \( <F, f>(x) = <G, g>(x) \) for any \( x \) for which \( z_2 \in F(x_1) \) or \( z_2 \in G(x_1) \).

2. If \( <F, f> \subseteq <G, g> \), then \( \mathcal{E}(F, f) \subseteq \mathcal{E}(G, g) \).

3. If \( F_i(n) \) then \( f_{i+1}(<n, p>') = f_i(<n, p>) \) where \( p' \) is the proof of \( F_{i+1}(n) \) known from (1) and \( F_i(n) \Rightarrow F_{i+1}(n) \).

4. Given any sequence of predicates \( F_i \) and functions \( f_i \) such that \( F_i(n) \Rightarrow F_{i+1}(n) \) for all \( n \), \( i \), and \( f_i \subseteq f_{i+1} \), then \( D(\sum_{n \in N} F_n, f_\omega)(m) \Rightarrow \sum_{n \in N} D(F_n, f_\omega)(m) \) for all \( m \).

**Definition:** For \( x \in D_\omega \) define
\[ f_\omega(x) = f_{zz,1}(<x_1, x_{z2,2}>) \]

We can now prove a compactness lemma.

5. **Compactness**

   If \( \mathcal{E}(F_\omega, f_\omega)(x) = y \), then \( \exists n \mathcal{E}(F_n, f_n)(x) = \mathcal{E}(F_\omega, f_\omega)(x) \).

   From these lemmas we can prove

**Least Fixed Point Theorem:**

1. \( F_\omega(n) \equiv D(F_\omega, f_\omega)(n) \) for all \( n \)
(b) for all \( z \in D_\omega \), \( f_\omega(x) = E(F_\omega.f_\omega)(x) \).

Combining (a) and (b) we conclude

(c) \( <F_\omega,f_\omega> \equiv \mathcal{E}(F_\omega,f_\omega) \)

(d) If \( <G,g> \equiv \mathcal{E}(G,g) \),

then \( <F_\omega,f_\omega> \subseteq <G,g> \).

To show that \( f_\omega \) defines a function \( D_f \rightarrow N \) in GRA we must know that the value of \( f_\omega(z) \) does not depend on the proof component. That is:

Independence of Proofs Theorem:

For \( \mathcal{E} \) defined from a GRA function definition \( f(x)=\exp \),
if \( x,y \) belong to \( D_\omega \) and \( x_1 = y_1 \)
then \( f_\omega(x) = f_\omega(y) \).

Proof:

We can see from the construction of \( \mathcal{E} \) that the proof component never enters into a value.

QED

(5) Compactness

If \( \mathcal{E}(F_\omega,f_\omega)(x)_z = y \) then \( \exists m. \exists z'. x_1 = z_1' \) & \( \mathcal{E}(F_{m,f_m})(x') \equiv \mathcal{E}(F_\omega,f_\omega)(x) \)

Proof by induction on the structure of \( \mathcal{E} \)

(1) Base case, \( E(F,f)(x) = h(x) \) and \( D(F,f)(n) = D_h(n) \). The result holds because \( E \) is independent of \( F \) and \( f \), so any \( n \) will do, e.g., \( (F_\omega,f_\omega)(x) = y \).

(2) In the induction case we assume the result for subterms and proceed by a case analysis on the structure of \( E \).

(i) \( E(F,f)(x) = h(T(F,f)(x)) \) and the result holds for \( T \).

Recall that \( D(F_\omega,f_\omega)(n) = \Sigma q \in D_T(F_\omega,f_\omega)(n).D_h(T(F_\omega,f_\omega)(<n,q>)) \)

By the induction hypothesis we know for any \( n \) and \( q \)

\( \exists m. (T(F_\omega,f_\omega)(<n,q>) = T(F_{m,f_m})(<n,q>). \) &

\( D_T(F_\omega,f_\omega)(n) \equiv D_T(F_{m,f_m})(n)) \)
So

\[ \Sigma q \in D_T(F_m,f_m)(n).D_k(T(F_m,f_m)(<n,q>)) \text{ and} \]

\[ h(T(F_\omega,f_\omega)(x)) = h(T(F_m,f_m)(x)) \text{ as was to be proved.} \]

(ii) \( E(F,f)(x) = f(T(F,f)(x)) \)

Recall that \( D(F_\omega,f_\omega)(n) = \Sigma q \in D_T(F_\omega,f_\omega)(n).F_\omega(T(F_\omega,f_\omega)(<n,q>)) \)

By definition \( F_\omega(x) \) is \( \Sigma m \in N.F_m(x) \).

By the induction hypothesis we know for any \( n \) and \( q \)

\[ \exists m. (T(T_\omega,f_\omega)(<n,q>) = T(F_m,f_m)(<n,q>) \& \]

\[ D_T(F_\omega,f_\omega)(n) \Leftrightarrow D_T(F_m,f_m)(n)) \]

Now choose an \( m_1 \) and \( m_2 \) as witnesses for \( \Sigma m \in N. \). Then let \( m \) be \( \max(m_1,m_2) \). We claim that

(i) \( \Sigma q \in D_T(F_m,f_m)(n).F_m(T(F_m,f_m)(<n,q>)) \) and

(ii) \( T(F_\omega,f_\omega)(<n,q>) = T(F_m,f_m)(<n,q>) \& \)

\[ D_T(F_\omega,f_\omega)(n) \Leftrightarrow D_T(F_m,f_m)(n). \]

By monotonicity of \( T \) and \( D_T \) (lemma 2) we know that

\[ T(F_{m_1},f_{m_1})(<n,q>) = T(F_{m_2},f_{m_2})(<n,q>) \]

\[ D_T(F_{m_1},f_{m_1})(n) \Leftrightarrow D_T(F_{m_2},f_{m_2})(n). \]

By the property \( F_i(n) \Rightarrow F_{i+1}(n) \) for all \( n \), we know \( F_m(T(T_m,f_m)(<n,q>)) \) hence the result.

(iii) \( E(F,f)(x) = F(F,f)(x) \rightarrow T_1(F,f)(<x_1,p_1>), T_2(f,f)(<x_1,p_2>) \)

Recall that \( D(F,f)(n) = D_B(F,f)(n) \& (B(F,f)(n)=0 \& D_{T_1}(F,f)(n) \lor \)

\[ B(F,f)(n) \neq 0 \& D_{T_2}(F,f)(n)) \]

and \( p_1 \in D_{T_1}(F,f)(n), p_2 \in D_{T_2}(F,f)(n). \)

By the induction hypothesis we know for any \( n,q \)
\[ \exists m \in \mathcal{D}(F_\omega, f_\omega)(n) \leftrightarrow D_B(F_{m}, f_{m})(n) \& \\
B(F_\omega, f_\omega)(<n, q>) = B(F_{m}, f_{m})(<n, q>) \]
\[ \exists m \in \mathcal{D}(F_\omega, f_\omega)(n) \leftrightarrow D_{T_i}(F_{m}, f_{m})(n) \& \\
T_i(F_\omega, f_\omega)(<n, q>) = T_i(F_{m}, f_{m})(<n, q>) \quad i=1,2. \]

For \( x, <x_1, p_1>, <x_2, p_2> \) choose \( m_0, m_1, m_2 \) respectively, let \( n=x_1 \), and take \( m=\max(m_0, m_1, m_2) \), then by monotonicity of \( T_1, T_2, B, D_{T_1}, D_{T_2}, D_B \) we know
\[ D(F_\omega, f_\omega)(n) = D_B(F_{m}, f_{m})(n) \& (B(F_{m}, f_{m})\exists N=0 \& D_{T_i}(F_{m}, f_{m})(n) \vee \\
B(f_{m}, f_{m})(n) \neq 0 \& D_{T_2}(F_{m}, f_{m})(n)) \]
and we know
\[ E(F_\omega, f_\omega)(x) = E(F_{m}, f_{m})(x) = \\
B(F_{m}, f_{m})(x) \rightarrow T_1(F_{m}, f_{m})(<x_1, p_1>), T_2(F_{m}, f_{m})(<x_2, p_2>). \]

QED

We can now prove the main theorem.

Proof of Least Fixed Point Theorem:

(a) We claim \( F_\omega(n) \leftrightarrow D(F_\omega, f_\omega)(n) \)

Suppose \( F_\omega(n) \), that is \( \exists m. F_{m}(n) \). Choose such an \( m \), say \( F_{m}(n) \). From (1) we know \( F_{m+1}(n) \)
as well, and by definition of \( F_{m+1}, F_{m+1}(n) = D(F_{m}, f_{m})(n) \).

Since \( F_{m}(n) \Rightarrow F_\omega(n) \) and \( f_{m} \subseteq f_\omega \), we know from (2) that \( D(F_{m}, f_{m})(n) \Rightarrow D(F_\omega, f_\omega)(n) \).

So the only if part \( (\Rightarrow) \) is proved.

Suppose \( D(F_\omega, f_\omega)(n) \). Then immediately from compactness we know \( \exists m. D(F_\omega, f_\omega)(n) \), so by definition \( F_{m+1}(n) \) which means \( \exists m. F_{m}(n) \) which is by definition \( F_\omega(n) \).

(b) We claim that for all \( x \in D_{f_\omega}, f_\omega(x) = E(F_\omega, f_\omega)(x) \).

From (a) we know that \( f_\omega \) and \( E(F_\omega, f_\omega) \) have the same domains so that the applications are sensible, i.e., \( D(F, f) \) is always the domain of \( E(F, f) \). By definition \( f_\omega(x) = f_{m+1}(<x_1, x_2, x_3>) \) and by compactness \( E(F_\omega, f_\omega)(x) = E(F_{m}, f_{m})(x') \) for some \( m \). Take \( k = \max(m, x_2, x_3) \).
Then by (2) \( f_{x_{1}}(<x_{1},q>) = f_{k}(<x_{1},q>) \) and \( E(F_{k},f_{k})(<x_{1},q'>) = E(F_{m},f_{m})(x) \) where \( q,q' \) are built from the implications \( F_{x_{1}}(z_{1}) \Rightarrow F_{k}(x_{1}), F_{m}(z_{1}) \Rightarrow F_{k}(z_{1}) \). Hence \( f_{m}(x) = E(F_{m},f_{m})(x) \).

(c) \(<F_{m},f_{m}>> \equiv \mathcal{E}(F_{m},f_{m}) \) follows by definition from (a) and (b).

(d) We claim that if \( <G,g> \equiv \mathcal{E}(G,g) \), then \( <F_{m},f_{m}>> \subseteq <G,g> \).

We have that \( <F_{m},f_{m}>> \subseteq <G,g> \), and we know

...\( <F_{m+1},f_{m+1}>> = \mathcal{E}(F_{m},f_{m}) \subseteq \mathcal{E}(G,g) \equiv <G,g> \) by monotonicity of \( \mathcal{E} \).

For any \( x \) by compactness we know

\( \mathcal{E}(F_{m},f_{m})(x) \equiv \mathcal{E}(F_{m},f_{m})(x) \) and by applying

\( \mathcal{E}(F_{m},f_{m})(x_{1}) \Rightarrow \mathcal{E}(F_{m},f_{m})(x_{1}) \Rightarrow F_{m+1}(x_{1}) \Rightarrow G(x_{1}) \)

and

\( \mathcal{E}(F_{m},f_{m})(x_{2}) = \mathcal{E}(F_{m},f_{m})(<x_{1},q>)_{1} = f_{m+1}(<x_{1},q>) \)

\( = g(<x_{1},q>) \)

where \( q \) is the proof of \( F_{m}(x_{1}) \).

QED

4.4 Remarks on the Interpretation

A method similar to that described above can be used to interpret inductive definitions of predicates. In this paper there is no room to present the method. It is worth noting however that the interpretation into type theory provides an implementation of GRA as a programming language. Not only can any partial recursive function \( f \) be executed, but any function implicitly defined by the proof of a claim such as \( \forall x.(P(x) \Rightarrow \exists y.Q(x,y)) \) can be executed. In this sense GRA can serve as a theory for program development as in the current PRL system at Cornell whose implementation has been led by my colleague Joseph L. Bates

ACKNOWLEDGEMENTS

I would like to express my appreciation to Stuart Allen who has constructively criticized much
of what I have to say on this topic. I also thank Donette Isenbarger for so efficiently and carefully preparing the manuscript.

REFERENCES


