FIRST ORDER SEMANTICS: A NATURAL
PROGRAMMING LOGIC FOR RECURSIVELY
DEFINED FUNCTIONS

by

Robert Cartwright, Jr.†

TR78-339

Department of Computer Science
Cornell University
Ithaca, N.Y. 14853

†This research has been partially supported by National Science
Foundation grant MCS76-14293.
1. Introduction

It is commonly believed that first order logic is too limited a formalism for stating and proving the interesting properties of recursively defined functions. Hitchcock and Park [7], for example, claim that the termination (totality) of a recursively defined function on a data domain $D$ cannot be expressed by a sentence in a first order theory$^1$ of $D$ extended by the defining equations. As a result of this criticism, most researchers developing programming logics for recursive languages have rejected first order logic in favor of more complex systems—notably least fixed point logics, e.g. Milner [11, 12, 13], Park [14], DeBakker [4,15], Scott and DeBakker [15], DeBakker and DeRoeve [5]. Nevertheless, in this paper we will show that a properly chosen, axiomatizable first order theory is a viable programming logic for recursively defined functions. In fact, we will present evidence which suggests that first order logic is a more appropriate formalism for reasoning about specific recursive programs than least fixed point logics.

---

$^1$A theory is a set of sentences (closed formulas) closed under logical implication.
2. Hitchcock and Park's Critique of First Order Logic

In [7], Hitchcock and Park consider the following recursively defined function on the natural numbers N:\n
\[
\text{zero}(n) = \text{if } n=0 \text{ then } 0 \text{ else } \text{zero}(n-1). \quad (*)
\]

While they admit that it is very easy to informally prove by induction that zero terminates on N, they claim that no sentence provable from an axiomatization T of N augmented by (*) can state that zero is total. Let N' denote the natural numbers extended to include the zero function. N' is clearly a model for T∪{(*)}. By the upward Lowenheim-Skolem theorem, the theory (set of true sentences) of N' has a non-standard model \( \hat{N}' \) which is a proper extension of N'. Hitchcock and Park assert that zero obviously does not terminate for all elements of \( \hat{N}' \). Given this assertion, no sentence \( \theta \) provable from T∪{(*)} can state zero is total since \( \theta \) must be true in \( \hat{N}' \).

The flaw in Hitchcock's and Park's analysis is their assumption that the interpretation of zero in a non-standard model must be obtained by applying computation rules to (*). In the example above, if we use a Peano style axiomatization for the natural numbers (including an induction axiom schema) then we can prove the sentence

\[
\forall n \ [ \ \text{zero}(n)=0 ]
\]

\(^2(*)\) abbreviates the sentence:

\[
\forall n \ [ \ (n=0) \Rightarrow \text{zero}(n)=0 ) \land \neg ((n=0) \Rightarrow \text{zero}(n)=\text{zero}(n-1))].
\]
FIRST ORDER SEMANTICS: A NATURAL PROGRAMMING LOGIC FOR RECURSIVELY DEFINED FUNCTIONS

by

Robert Cartwright, Jr.

78-339

Department of Computer Science
Cornell University
Ithaca, N.Y. 14853

Abstract

Despite the widespread belief to the contrary, virtually any interesting property of recursively defined total functions on a data domain D can be stated and proved in a simple first order logic for D, by using an approach we call "first order semantics". In particular, it is easy to prove within this formalism that common recursive functions (such as standard LISP functions, McCarthy's 91-function, and Ackermann's function) are total. The primary features of first-order semantics are:

1. The data domain D must be a well-founded set which explicitly includes the undefined object (representing non-termination) as well as ordinary data objects.

2. Recursive definitions of functions on D are interpreted as axioms augmenting the first order axiomatization of the data domain.
3. The interpretation of a system $F$ of recursive function definitions on $D$ is simply the least fixed point solution of $F$.

Since the data domain $D$ is a well founded set, the first order axiomatization of $D$ includes a structural induction axiom schema. This axiom schema serves as the fundamental "proof rule" of first-order semantics.

The major weakness of first order semantics is its failure to capture the notion of least fixed point. In fact, any fixed point solution of a set of recursive function definitions is consistent with the augmented axiomatization of the data domain $D$. To alleviate this problem, we develop an effective procedure for transforming any set of recursive function definitions into an equivalent set of definitions which has a unique fixed point. When augmented by this transformation technique, first order semantics is sufficiently powerful to prove virtually any extensional property of any system of recursively defined functions.

Keywords: semantics, verification, program transformations, programming logic, recursively defined functions.
from $T_u\{(*))\}$\(^3\). Consequently, the function zero is identically zero in every model of $T_u\{(*))\}$. Furthermore, since the models of $T_u\{(*))\}$ do not contain a special undefined element (usually denoted $\bot$), every model must, by definition, interpret every function symbol by a (total) function. Hence, no recursion equation augmenting $T$ can define a non-total function.

The situation is more interesting if we axiomatize $N_u\{i\}$ (where $i$ denotes the undefined object) instead of $N$. In this case, the interpretation for an $n$-ary function $f$ may be non-total (in the sense that is maps some elements of the data domain into $i$). We can assert (within the corresponding first-order language) that $f$ is total on $N$ by simply stating:

$$\forall x_1, \ldots, x_n [(x_1 \neq i) \land \ldots \land (x_n \neq i) \Rightarrow f(x_1, \ldots, x_n) = i].$$

Let $T_1$ be a "Peano-like" axiomatization (including an induction axiom schema) for $N_u\{i\}$.

Given $T_1$, and the recursion equation (*), we can easily establish that Hitchcock and Park's zero function is total on $N$ by proving the sentence $\forall n [n \neq i] \Rightarrow \text{zero}(n) = i$. The proof (which appears in Appendix I) is a direct translation of the informal structural induction proof that Hitchcock and Park cite in their paper. One is forced to conclude that the totality of recursively functions like zero is easily expressed and proven within first order logic.

\(^3\)Assuming the language of $T$ includes for each arity $n > 0$ a countably infinite collection of function symbols $a, b, \ldots, x, a, a, \ldots$ including zero. Otherwise, we must augment $T$ by additional instances (involving the new function symbol zero) of the induction axiom schema.
3. Basic Concepts of First Order Semantics

As we hinted in the previous section, the undefined object \( i \) plays a crucial role in first order semantics. In order to use first order logic to define the meaning of arbitrary recursively defined functions, we must include the undefined object \( i \) in the data domain. Otherwise, recursive definitions like

\[
f(x) = f(x)+1 \quad (**)
\]
on the natural numbers are inconsistent with the axiomatization of the data domain (the interpretation of \( f \) must be a (total) function on the natural numbers, yet no such function exists). If a set of recursion equations contains an inconsistent definition like (**), then the axiomatization of the data domain augmented by the equations is inconsistent (has no model).

Fortunately, if we include the undefined object \( i \) in the data domain, force all primitive functions to be continuous\(^4\), and slightly modify our treatment of conditional and boolean expressions, then we can guarantee every set of recursion equations \( F \) is consistent with the axiomatization of the data domain. This property is an immediate consequence of Kleene's fixed point theorem for continuous functionals.

\(^4\)For a definition of continuity, see [Manna 74]. All strict functions and the standard if-then-else function are continuous
Under the stated assumptions, we can extend the model of the data domain to include F by interpreting the functions defined in F as the least fixed points of their recursion equations.

To satisfy the continuity property required by Kleene's theorem, we must replace the predicates of the data domain by corresponding strict\textsuperscript{5} boolean functions. If our data domain D is divided into types (sorts in the terminology of first order logic), then we can simply include a boolean type in D. For the sake of simplicity, we will assume that the data domain consists of a single type. In this case, we embed boolean values in the data domain D by partitioning D-(⊥) into two non-empty subsets D\textsubscript{T} and D\textsubscript{F} consisting of the objects representing true and false respectively. For example, if our data domain is simply the natural numbers ℕ augmented by ⊥, then we could let D\textsubscript{F} = \{0\} and D\textsubscript{T} = \{x∈ℕ | x≠0\}. In LISP, D\textsubscript{F} = \{NIL\} and D\textsubscript{T} = \{x∈S-expressions | x≠NIL\}.

We augment the set of primitive functions on D by adding the standard if-then-else function mapping D\textsuperscript{3} into D. While if-then-else is not strict (since if true then x else ⊥ = x), it is continuous.

Given these modifications to the data domain D and its axiomatization, a recursive program F on D has the form:

\textsuperscript{5}An n-ary function is strict if it is undefined (⊥) whenever any of its arguments is undefined (⊥).
\[\begin{align*}
    f_1(x_1, \ldots, x_{n_1}) &= \tau_1 \\
    f_2(x_1, \ldots, x_{n_2}) &= \tau_2 \\
    \ldots \\
    f_n(x_1, \ldots, x_{n_n}) &= \tau_n
\end{align*}\]

where \(f_1, f_2, \ldots, f_n\) are function symbols distinct from the primitive function symbols of \(D\) and \(\tau_1, \tau_2, \ldots, \tau_n\) are terms constructed from the primitive function symbols of \(D\) augmented by \(f_1, f_2, \ldots, f_n\) such that \(\tau_i\) contains no variables other than \(x_1, \ldots, x_{n_i}; i = 1, 2, \ldots, n\).

The meaning of the functions \(f_1, f_2, \ldots, f_n\) is the least fixed solution over \(D\) of the system of recursion equations comprising \(F\). The corresponding deductive system for reasoning about \(f_1, \ldots, f_n\) is simply standard first order implication given the axiomatization of \(D\) augmented by the equations in \(F\) (which are ordinary first order formulas). Given any recursive program \(F\) defining total functions \(f_1, \ldots, f_n\), we can prove virtually any interesting property of the functions \(f_1, \ldots, f_n\) totality by using ordinary first order deduction. Most proofs strongly rely on structural induction.

4. A Sample Proof

As an illustration, consider the following simple example. Let flat, and flat1 be recursively defined functions over the domain of \textit{S-expressions} defined by the following
equations:

\[
\begin{align*}
\text{flat}(x) &= \text{flatl}(x, \text{NIL}). \\
\text{flatl}(x, y) &= \text{if atom } x, \text{ then cons}(x, y) \\
&\quad\text{ else flatl(car } x, \text{flatl(cdr } x, y)).
\end{align*}
\]

The function flat returns a linear "in-order" list of the atoms appearing in the S-expression x. For example

\[
\begin{align*}
\text{flat}((\text{A.B})) &= (\text{A B}) \\
\text{flat}((\text{A.(B.A)})) &= (\text{A B A}) \\
\text{flat}((\text{A})) &= (\text{A}) \\
\text{flat}((\text{A.C}.B)) &= (\text{A C B})
\end{align*}
\]

We want to prove that \(\text{flatl}(x, y)\) terminates for arbitrary S-expressions x and y (obviously implying \(\text{flat}(x)\) terminates for any S-expression x). Formally, we state the theorem in the first order theory of S-expressions \(\mathcal{U}\{1\}\) as follows:

\[
x \neq 1 \land y \neq 1 \Rightarrow \text{flatl}(x, y) \neq 1.
\]

By using the unary predicate is-Sexpr (which is true for elements of the domain except 1) we can restate the theorem in the more readable form:

\[
is\text{-Sexpr}(x) \land is\text{-Sexpr}(y) \Rightarrow is\text{-Sexpr}(\text{flatl}(x, y)).
\]

We prove the theorem by applying structural induction on x.

Base-case. \(x\) is an atom.

Then \(\text{flatl}(x, y) = \text{cons}(x, y)\) which must be an S-expression since \(x\) and \(y\) are S-expressions.

Induction-step. Given

1) \(\forall y \ is\text{-Sexpr}(x_1) \land is\text{-Sexpr}(y) \Rightarrow is\text{-Sexpr}(\text{flatl}(x_1, y))\), and

2) \(\forall y \ is\text{-Sexpr}(x_2) \land is\text{-Sexpr}(y) \Rightarrow is\text{-Sexpr}(\text{flatl}(x_2, y))\)

we must show

\[
\forall y \ is\text{-Sexpr}(\text{cons}(x_1, x_2)) \land is\text{-Sexpr}(y) \Rightarrow is\text{-Sexpr}(\text{flatl}(\text{cons}(x_1, x_2), y))
\]
In this case, \( \text{flat1}(\text{cons}(x_1, x_2), y) = \text{flat1}(x_1, \text{flat1}(x_2, y)) \)

Since \( \text{cons}(x_1, x_2) \) is an S-expression, \( x_1 \) and \( x_2 \) must be S-expressions. Hence, by induction hypothesis 1, \( \text{flat1}(x_2, y) \) is an S-expression. Given this fact we can apply induction hypothesis 2 to deduce that \( \text{flat1}(x_1, \text{flat}(x_2, y)) \) is an S-expression.

Q.E.D.

Some additional examples appear in Appendix I.

5. Incompleteness of First Order Semantics

Besides the inescapable Godel incompleteness of any axiomatization of a non-trivial data domain \( D \), there is a more fundamental kind of deductive incompleteness inherent in first order semantics. The problem is that the axiomatization of the data domain augmented by a set of recursion equations is satisfied by any model consisting of the data domain augmented by a fixed point solution for the recursion equations. The augmented axiomatization fails to capture the concept of least fixed point.

What are the implications of this form of incompleteness? If the least fixed point solution for the recursion equations is total on \( D-(1) \), the problem does not exist because the equations have a unique fixed point solution. On the other hand, if some function in the least fixed point solution is partial, then we cannot prove any property of the least fixed point which does not hold for all fixed points. For example, we cannot prove anything interesting about the function \( f \) defined by

\[
f(x) = f(x) \quad (***)
\]
since any interpretation for $f$ over the data domain satisfies (***) , not just the everywhere undefined function.

There are several possible solutions to this problem. John McCarthy [10] suggests adding an axiom schema $\phi_f$ (containing a free function symbol) for each function $f$ defined in the recursive program. The schema $\phi_f$ asserts that $f$ is the least function satisfying the recursion equation for $f$. Since McCarthy's approach verges on converting first order semantics into a second order system, our inclination is to follow a different approach employing the concept of complete recursive programs.

Briefly, a set of recursion equations is a complete recursive program if and only if it has a unique fixed point solution. In a subsequent section of the paper, we will prove that every recursive program can be effectively transformed into an equivalent complete recursive program (in the sense that the two programs have identical least fixed point solutions). As a result, we can reason about recursively defined partial functions using first order semantics by first transforming the recursive program into an equivalent complete recursive program.

6. Reasoning about Call-By-Value Fixed Points

Computing the standard least fixed point of a recursively defined function requires "call-by-name" evaluation. On the

---

6. Assuming that the primitive functions meet certain mild restrictions. Otherwise a more sophisticated evaluation mechanism is required. See Manna [4].
other hand, most practical programming languages (e.g. LISP, PASCAL) employ "call-by-value" evaluation which has slightly different semantics. Fortunately; first order semantics readily adapts to call-by-value recursive programs. The only changes required to handle call-by-value programs are:

1. The meaning of the functions defined in the program is the least call-by-value fixed point solution of the recursion equations.

2. For each recursion equation
   \[ f(x_1, \ldots, x_n) = \tau \]
   we add the two axioms
   \[
   \begin{align*}
   x_1 = 1 \land \ldots \land x_n = 1 & \Rightarrow f(x_1, \ldots, x_n) = \tau \\
   x_1 = 1 \lor \ldots \lor x_n = 1 & \Rightarrow f(x_1, \ldots, x_n) = 1
   \end{align*}
   \]
   to the data domain axiomatization instead of adding the recursion equation itself.

The proof that the augmented data domain for a call-by-value recursive program is a model for the augmented axiomatization appears in [Cartwright 76b].

Least call-by-value fixed points are an attractive alternative to standard least fixed points because they are easier to compute and programmers seem more comfortable with their semantics. In addition, we will show in the subsequent section and Appendix II that the complete recursive program corresponding to an arbitrary call-by-value recursive program is easier to describe and to understand than the equivalent construction for standard recursive programs.

---

The call-by-value least fixed point of the recursion equation
\[ f(x_1, \ldots, x_n) = \tau \]
is the standard least fixed point of the modified recursion equation
\[
\begin{align*}
\delta(x_1) \land \ldots \land \delta(x_n) & \Rightarrow f(x_1, \ldots, x_n) = \tau \\
\delta(1) & \land \ldots \land \delta(1) & \Rightarrow f(x_1, \ldots, x_n) = 1
\end{align*}
\]
where \( \delta \) is the primitive "is-defined" function with the property:
\[
\begin{align*}
\delta(x) & = \text{true if } x \neq 1 \\
\delta(1) & = 1
\end{align*}
\]
7. Construction of Complete Recursive Programs

In this section, we will describe the procedure for constructing the complete recursive program corresponding to an arbitrary call-by-value recursive program, and prove that the constructed program has the desired properties. We relegate the analogous construction and proof for standard recursive programs to Appendix II, since they are similar but somewhat more complex.

The intuitive idea underlying the construction is to define for each function \( f \) in the original program a corresponding function \( f^* \) such that \( f^*(x_1, \ldots, x_n) \) constructs the computation sequence for \( f(x_1, \ldots, x_n) \). Constructing the actual computation sequence really is not necessary; the value of all elements in the sequence except for the final one (the value of \( f(x_1, \ldots, x_n) \)) are irrelevant. It is the structure of the sequence of the sequence which is significant, since it prevents a fixed point solution from "looping back" on itself.

For example, consider the trivial recursion equation

\[
f(x) = \text{if } x \text{ equal } 0 \text{ then } 0 \text{ else } f(g(x))
\]

over the domain of LISP S-expressions where \( g \) is any unary function with fixed points. If we define \( f^* \) by

\[
f^*(x) = \text{if } x \text{ equal } 0 \text{ then } \text{list}(0) \\
\text{else cons}(g(x), f^*(g(x)))
\]

then \( f^* \) constructs a sequence containing an element for each
expansion of \( f \) in the call-by-value evaluation of \( f(x) \), assuming \( f(x) \) terminates. If \( f(x) \) does not terminate, then every fixed point solution for \( f^* \) must be undefined (1) at \( x \). Otherwise, \( f^*(x) \) would have to be infinitely long which is impossible (all S-expressions are finite). The recursion equation

\[
f'(x) = \text{last}(f^*(x))
\]

where \text{last} is the standard LISP function which extracts the final element in a list, clearly defines a function identical to \( f \). Furthermore, the equations defining \( f^* \) and \( f' \) form a complete recursive program.

Before we can define the general form of the complete recursive program construction, we need to introduce some notation and terminology.

In the sequel, we will continually need to distinguish between a function symbol \( f \) and its interpretation \( \hat{f} \). We will use a circumflex sign ("^") over a function symbol to denote its interpretation.

Let \( D^+ = D \cup \{1\} \) be a data domain with operations if-then-else and a collection \( G \) of strict primitive functions. Let \( L_{0^+} \) be the corresponding first order language. On \( D^+ \), we define the partial ordering \( \leq \) on \( D^+ \) as follows:

\[
x \leq y \text{ iff } x = 1.
\]

The set \( D^+ \) is clearly a complete partial order \(^8\) under the partial ordering \( \leq \).

\(^3\) A set \( S \) under the partial order \( \leq \) is a complete partial order iff

1) \( \leq \) is a partial ordering (for a definition, see [31])
2) every ascending sequence \( x_0 \leq x_1 \leq x_2 \ldots \) has a least upper bound.
Let $F$ be a set of recursion equations $\{f_1(\vec{x}_1) = t_1, \ldots, f_n(\vec{x}_n) = t_n\}$ where $\vec{x}_i$ is a vector of variables $x_1, \ldots, x_{m_i}$; $f_1, \ldots, f_n$ are new function symbols not in $L_D^+$; and $t_1, \ldots, t_n$ are terms in the augmented language $L_F = L_D^+ \cup \{f_1, \ldots, f_n\}$ such that no variables other than $\vec{x}_i$ appear in $t_i$, $i=1, \ldots, n$.

The call-by-value least fixed point solution of $F$ (denoted $[f_1, \ldots, f_n]$) is the least upper bound of the ascending sequence of $n$-tuples of functions $[f_1(0), \ldots, f_n(0)] \subseteq [f_1(1), \ldots, f_n(1)] \subseteq \cdots \subseteq [f_1(k), \ldots, f_n(k)] \subseteq \cdots$ where we have extended the partial ordering $\subseteq$ to $n$-tuples of functions in the usual way 

$([r_1, \ldots, r_n] \subseteq [s_1, \ldots, s_n]$ iff $r_i[d] \subseteq s_i[d]$ for all $d \in D$, $i=1, \ldots, n$). We inductively define $[f_1(k), \ldots, f_n(k)]$, $k=0, 1, \ldots$ as follows:

$\hat{f}_i(\vec{a}) = 1$ for all tuples $\vec{a}$ over $D^+$.

$\hat{f}_i(k) (\vec{a}) = \sum_{k=0}^{n-1} \left[ t_i \right] s \text{ for all tuples } \vec{a} \text{ over } D.$

Otherwise

where $s$ is a state vector mapping the variables $\vec{x}_i$ into $\vec{a}$; and $M_k \left[ t_i \right] s, k=0$ is the interpretation of $t_i$ under $s$ when $f_1, \ldots, f_n$ are interpreted by $\hat{f}_1(k), \ldots, \hat{f}_n(k)$, respectively, and the function symbols and constants of $L_D^+$ are interpreted by their meanings in $D^+$. In informal terms, $\hat{f}_i(k)$ is the call-by-value evaluation for $f_i$ to depth $k$.

Before we can define functions which construct computation sequences, we must extend the data domain $D^+$ and the augmented
language \( L_D \) to accommodate sequences. Let \( \text{SEQ}(D) \) denote the set of finite, non-empty sequences over \( D \) and \( D^+ \) denote the extended data domain \( D^+ \cup \text{SEQ}(D) \). We extend every function \( \hat{g} \in G \) to the domain \( D^+ \) as follows:

\[
\hat{g}(d_1, \ldots, d_m) = 1 \text{ if some } d_i \notin D^+
\]

On \( D^+_{\text{SEQ}} \) we define the new binary function \( \hat{\hat{\circ}} \) (append) and unary functions \( \hat{\text{last}} \) and \( \hat{\text{seq}} \) as follows:

\[
[a_1, \ldots, a_k] \hat{\hat{\circ}} [b_1, \ldots, b_m] = [a_1, \ldots, a_k, b_1, \ldots, b_m]
\]

for \( [a_1, \ldots, a_k], [b_1, \ldots, b_m] \in \text{SEQ}(D) \)

\[
x \hat{\hat{\circ}} y = 1 \text{ if } x \in \text{SEQ}(D) \text{ or } y \in \text{SEQ}(D)
\]

\[
\hat{\text{last}} ([a_1, \ldots, a_k]) = a_k \text{ for } [a_1, \ldots, a_k] \in \text{SEQ}(D)
\]

\[
\hat{\text{last}} (x) = 1 \text{ for } x \notin \text{SEQ}(D)
\]

\[
\hat{\text{seq}} (d) = [d] \text{ for } d \in D
\]

\[
\hat{\text{seq}} (x) = 1 \text{ for } x \notin D
\]

Let \( L_{D^+}^{\text{SEQ}} \) denote the first order language \( L_{D^+}^{\text{SEQ}} \) corresponding to \( D^+ \).

Now we are finally ready to construct the complete recursive program \( F' \) equivalent to \( F \). Let \( t \) be an arbitrary term in the language \( L_F \). The computation sequence term \( t \) (in the extended language \( L_F, = L_{D^+}^{\text{SEQ}} \cup \{f_1', \ldots, f_n'\} \)) corresponding to \( t \) is inductively defined as follows:

1. If \( t \) is a constant or a variable \( x \),

\[
t' = \text{seq}(x).
\]

2. If \( t \) has the form \( g(u_1, \ldots, u_m) \) where \( g \in G \),

\[
t' = u_1' \hat{\hat{\circ}} \ldots \hat{\hat{\circ}} u_m' \hat{\hat{\circ}} \text{seq}(g(\text{last}(u_1'), \ldots, \text{last}(u_m'))).
\]

3. If \( t \) has the form \( f_i'(u_1, \ldots, u_m) \),

\[
t' = u_1' \hat{\hat{\circ}} \ldots \hat{\hat{\circ}} u_m' \hat{\hat{\circ}} f_i'(\text{last}(u_1'), \ldots, \text{last}(u_m')).
\]
4. If $t$ has the form if $u_0$ then $u_1$ else $u_2$, 
   \[ t' = u_0 \cdot \sigma (\text{if last}(u_0')) \text{ then } u_1' \text{ else } u_2' \]. 

The complete recursive program $F'$ corresponding to $F$ is the 
set of recursion equations \( \{ f_1'(x_1) = t_1', \ldots, f_n'(x_n) = t_n' \} \).

Theorem. The (call-by-value) complete recursive program $F'$ constructed from 
the set of recursion equations $F$ has the following properties:

1. $\text{last}(\hat{\bar{f}}_i'(\bar{a})) = \hat{\bar{f}}_i(\bar{a})$ for all $m_i$-tuples $\bar{a}$ over $D^+$.
2. $F'$ has a unique call-by-value least fixed point solution.

Proof. Let $M'_k$ denote the call-by-value interpretation function 
for $F'$ analogous to $M_k$ for $F$. Property 1 is an immediate consequence of the following lemma.

Lemma. For every term $t$ in $L_F$ and every state $s$ over $D^+$, 
\[ M_k[t][s] = M'_k[\text{last}(t')][s], \forall k \geq 0. \]

Proof of Lemma. The proof proceeds by induction on the pair 
$[k,t]$. By hypotheses, we may assume that the lemma holds for 
all $[k_0,t_0]$ such that either $k_0 < k$, or $k_0 = k$ and $t_0$ is a proper subterm of $t$.

Case 1. $t$ is a constant or a variable $x$. Then $t'$ has 
the form $\text{seq}(x)$ implying $M'_k[\text{last}(t')][s] = M'_k[x][s] = M_k[t][s] \text{ for arbitrary } k \geq 0.$

Case 2. $t$ has the form $g(u_1', \ldots, u_m')$, $g \in \mathcal{G} \cup \{ \sigma, \text{last}, \text{seq} \}$. 
Then $t' = u_1' \cdot \sigma \ldots u_m' \cdot \text{seq}(g(\text{last}(u_1'), \ldots, \text{last}(u_m'))).$

By hypothesis $M'_k[\text{last}(u_i')][s] = M_k[u_i][s]$, for all $s$, $i = 1, \ldots, m$. 
If for some $i$, $M_k[u_i][s] = 1$ then $M'_k[\text{last}(u_i')] = 1$.
implying $M_k'[\text{last}(t')] [s] = 1$ (since $\hat{g}$, $\hat{e}$, and $\text{last}$ are strict). Hence we assume $M_k [u_i] [s] \neq 1$ for all $i$. By the induction hypotheses and the definition of $\text{last}$, we conclude $M_k'[u_i][s] \in \text{SEQ}(D)$ for all $i$, implying $M_k'[\text{last}(t')] [s] = M_k'[g(\text{last}(u_i'), \ldots, \text{last}(u_m'))] [s]$

$$= M_k'[g(u_1', \ldots, u_m')] [s] \quad \text{(by induction)}$$

$$= M_k [t] [s].$$

Case 3. $t$ has the form $f_j(u_1', \ldots, u_m')$. By an argument analogous to the one presented in the previous case, we can assume $M_k[u_i][s] \neq 1$ and $M_k'[u_i'][s] \in \text{SEQ}(D)$ for all $i$. Consequently,

$$M_k'[\text{last}(t')] [s] = M_k'[\text{last}(f_j'(\text{last}(u_1'), \ldots, \text{last}(u_m')))] [s]$$

$$= M_k'[\text{last}(f_j(x_1', \ldots, x_m'))][s']$$

where $s'$ is a state vector binding $x_i$ to $M_k'[\text{last}(u_i')] [s] = M_k[u_i][s]$ for $i = 1, \ldots, m_j$. Since no $x_i$ is bound to $i$, we may expand $f_j(x_1', \ldots, x_m')$ yielding $M_k'[\text{last}(t')] [s'] = M_k'[\text{last}(t'_j)] [s']$

$$= M_k[-1][t_j][s'] \quad \text{(by induction)}$$

$$= M_k[f_j(x_1', \ldots, x_m')][s']$$

$$= M_k [t] [s].$$

Case 4. $t$ has the form if $u_0$ then $u_1$ else $u_2$. By definition $t' = u_0' \hat{e}$ (if last($u_0'$) then $u_1'$ else $u_2'$).

Subcase a. $M_k[u_0][s] = 1$. By induction $M_k'[\text{last}(u_0')] = 1$.

Since $\hat{e}$ and $\text{last}$ are strict, and if-then-else is strict in its first argument, $M_k'[\text{last}(t')] [s] = 1$ and $M_k[t][s] = 1$.

Subcase b. $M_k[u_0][s] \in D$. If $M_k[u_0][s]$ is a "true" element of $D$ then $M_k[t][s] = M_k[u_1][s]$. By induction,

$$M_k[u_0][s] = M_k'[\text{last}(u_0')][s] \quad \text{implying}$$

$$M_k'[t'][s] = M_k'[\text{last}(u_1')][s]$$

$$= M_k[u_1][s] \quad \text{(by induction)}$$

$$= M_k [t] [s].$$
An analogous argument proves the "false" case. Q.E.D.

We prove property 1 of the theorem as follows. By lemma 1:

\[ M_k[f_i(x_1, ..., x_{m_i})][s] = M_k'[\text{last}(\text{seq}(x_1)) \circ \ldots \circ \text{seq}(x_{m_i})] \\
\hat{f}_i'(\text{last}(\text{seq}(x_1)), \ldots, \text{last}(\text{seq}(x_{n_i})))][s] \]

for all \( k \geq 0 \), all state vectors \( s \) over \( D^+ \). Property 1 trivially holds for \( m_i \)-tuples \( \bar{a} \) where some element of \( \bar{a} \) is \( i \) since the functions \( \hat{f}_i, \hat{f}_i' \) and \( \text{last} \) are all strict. So we can restrict our attention to state vectors \( s \) that bind \( x_1, \ldots, x_{m_i} \) to values in \( D \). Simplifying the right hand side of the equation above yields

\[ M_k[f_i(x_1, ..., x_{m_i})][s] = M_k' [\text{last}(f_i'(x_1, ..., x_{n_i}))][s] \]

for state vectors \( s \) binding \( x_1, \ldots, x_{m_i} \) to values in \( D \). Since \( D^+ \) and \( D_{\text{seq}}^+ \) are both flat domains, the functions \( \hat{f}_i \) and \( \hat{f}_i' \) have the following property. For any \( m_i \)-tuple \( d \) over \( D \) there exists \( k_n \) such that \( \hat{f}_i(k_0) = \hat{f}_i(d) \) and \( \hat{f}_i'(k_0) = \hat{f}_i'(d) \). Let \( d \) be an arbitrary \( m_i \)-tuple over \( D \) and \( s \) be a state mapping \( x_i \) into \( d \).

Then

\[ \hat{f}_i'(d) = \hat{f}_i'(d) = M_k[f_i(x_1, ..., x_{m_i})][s] = \]

\[ M_k' [\text{last}(f_i'(x_1, ..., x_{m_i}))][s] = \text{last}(\hat{f}_i'(d)) = \text{last}(\hat{f}_i'(d)) \]

proving property 1.

To prove property 2, we must introduce some new definitions. Let \( H \) be a set of strict functions \( \hat{h}_i, i=1, \ldots, n \), over \( D^+_{\text{SEQ}} \) corresponding to the function symbols \( f_1', \ldots, f_n' \). We define \( M_K \) \( k \geq 0 \) as the meaning function for terms of \( L_F \) identical to \( M_k \) except that \( M_K \) interprets \( f_i', i=1, \ldots, n \) by \( \hat{f}_i'(k) \) where \( \hat{f}_i'(k) \).
is inductively defined by
\[ f'_i(0)_H = h_i \]
\[ f'_i(k)_H(\bar{d}) = \begin{cases} \text{if } \bar{t} \uparrow \bar{d} & M'_{k-1}_H \circ [t][s]_d \\ \text{otherwise} & \end{cases} \]

where \( s_d \) maps \( \bar{x}_i \) into \( \bar{d} \). Informally, \( f'_i(k)_H \) is the call-by-value evaluation of the recursion equation for \( f_i \)

where \( \hat{h}_j, j=1,\ldots,n \) interprets \( f_j \) in calls of depth \( \geq k \). If all the functions \( \hat{h}_j \) in \( H \) are everywhere undefined \( \hat{f}'_i(k)_H = \hat{f}'_i(k) \).

Property 2 is a simple consequence of the following lemma.

Lemma 2. Let \( t \) be any term in \( \mathbb{L}' \). Let \( H \) be a set of strict functions \( \hat{h}_1,\ldots,\hat{h}_n \) corresponding to \( f'_1,\ldots,f'_n \). Then for any \( k \geq 0 \) and any state vector \( s \) over \( \mathbb{N}^+_{\text{SEQ}} \)
\[ M'_k[t][s] \neq 1 \] \text{implies either } \begin{align*}
\text{length } (M'_{k_H}[t][s]) & \geq k \\
\text{or } & \\
\text{length } (d) & = 0 \text{ for } d \in D
\end{align*}

Proof of Lemma 2. The proof proceeds by induction on \([k, t]\). In the course of the proof, we will use the following lemmas which are easily proven by structural induction on \( t \):

Lemma 3a. For any term \( t \) in \( \mathbb{L}' \), \( M'_k[t][s] \neq 1 \) implies \( M'_k[t][s] = M'_{k_H}[t][s] \).
Lemma 3b. For any term t in L_F' not containing any recursive function symbol \( f_1' \), \( \mathcal{M}_K[t] = \mathcal{M}_{K_H}[t] \).

The proof of lemma 2 breaks down into two cases.

Case 1. \( k=0 \). The lemma in this case is a trivial consequence of the definition of length.

Case 2. \( k>0 \). We perform a case split on t.

Subcase a. t is a constant or variable x. Then t = seq(x), implying
\[
\mathcal{M}_K'[t'][s] = \mathcal{M}_K'[\text{seq}(x)][s] = \mathcal{M}_K'[\text{seq}(x)][s] = \text{seq}(\mathcal{M}_K[x][s]).
\]
Consequently, if \( \mathcal{M}_K[x][s] = 1 \), then \( \mathcal{M}_K'[\text{seq}(x)][s] = 1 \).

Subcase b. t has the form \( g(u_1, \ldots, u_m) \) where \( \hat{g} \in G \). In this case, t' = \( u_1 \hat{e} \ldots u_m \hat{e} \text{seq}(g(\text{last}(u_1), \ldots, \text{last}(u_m))) \).

If \( \mathcal{M}_K[u_j][s] = 1 \) for some j, then by induction \( \mathcal{M}_K'[u_j'] = 1 \) or length \( \mathcal{M}_K'[u_j][s] \geq k \), implying the lemma holds (since \( \hat{e} \) is strict). On the other hand, if \( \mathcal{M}_K[u_j][s] \neq 1 \) for all j, then by lemmas 1 and 3a, \( \mathcal{M}_K'[u_j][s] = \mathcal{M}_K'[u_j'][s] \in \text{SEQ}(D) \) for all j. Consequently,
\[
\mathcal{M}_K'[t'][s] = \mathcal{M}_K'[u_1 \hat{e} \ldots u_m \hat{e} \text{seq}(g(\text{last}(u_1), \ldots, \text{last}(u_m)))][s]
\]
\[
= \mathcal{M}_K'[u_1 \hat{e} \ldots u_m \hat{e} \text{seq}(g(\text{last}(u_1), \ldots, \text{last}(u_m)))[s]
\]
\[
= \mathcal{M}_K'[t'][s].
\]

If \( \mathcal{M}_K[t][s] = 1 \), then by lemmas \( \mathcal{M}_K'[t'][s] = 1 \) implying \( \mathcal{M}_K'[\hat{d}][s] = 1 \).

Subcase c. t has the form \( f_i(u_1, \ldots, u_m) \). If \( \mathcal{M}_K[u_j][s] = 1 \) for some j, the proof is identical to the analogous section of the previous case. On the other hand, when \( \mathcal{M}_K[u_j][s] \neq 1 \) for all j,
\[
\mathcal{M}_K'[t'][s] = \mathcal{M}_K'[u_j'][s] \hat{e} \ldots \hat{e} \mathcal{M}_K'[u_j'][s] \hat{e}
\]
\[
\hat{f}_{ik} \left( \text{last} \left( \mathcal{M}_K'[u_1][s] \right), \ldots, \text{last} \left( \mathcal{M}_K'[u_m][s] \right) \right)
\]
\[
= \mathcal{M}_K'[x_1 \hat{e} \ldots x_m \hat{e} t'][s']
\]
where s' maps \( x_j \) into \( \mathcal{M}_K'[u_j'][s] \).
By induction $M'_{k-1}[t'][s'] = 1$ or length $(M'_{k-1} H^1_{-1} t')[s'] \geq k-1$.

Hence the lemma clearly holds (since each $x_j$ has length $\geq 1$).

Subcase d. $t$ has the form if $u_0$ then $u_1$ else $u_2$. If $M_k[u_0][s] = 1$, the proof is identical to the analogous section of subcase b. On the other hand, when $M_k[u_0][s] \neq 1$, either $M_k[u_0][s]$ is a "true" element of $D$ or a "false" one. In the

former case, $M_k[t][s] = M_k[u_1][s]$ and $M_k[t'][s] = M_k[u_1'][s]$. By induction, $M_k[u_1] = 1$ implies either $M_k[u_1'] = 1$ or

length $(M_k H_{-1} u_1') \geq k$. Hence the lemma holds in this case.

An analogous argument holds for the "false" case. Q.E.D.

We prove that Property 2 follows from lemma 2 as follows.

Let $H = \{\hat{h}_1, ..., \hat{h}_n\}$ be any call-by-value fixed point solution of the set of recursive equations $F'$; i.e. for $i=1, ..., n$

$M'_{0H} [f'_i(\tilde{x}_i)] = M'_{0H} [t'].$

By induction on $[k, t]$ we can easily show for all $k \geq 0$ and all terms $t'$ in $L_{F'}, M'_{H^1} [t] = M'_{0H} [t]$. 

Now assume $H$ is not the least call-by-value fixed point solution of $F'$; i.e. $f'_i(\tilde{a}) = 1$ but $h_i(\tilde{a}) = 1$ for some $i$, some $\tilde{a} \in D_{\Sigma E}$. Then $\forall k \geq 0$ $M'_{kH} [f'_i(\tilde{x}_i)] [s_d] = M'_{0H} [f'_i(\tilde{x}_i)] [s_d]$ where $s_d$ binds

$\tilde{x}_i$ to $\tilde{a}$. Hence the length of $\hat{h}_i(\tilde{a})$, i.e. length $(M'_{0H} [f'_i(\tilde{x}_i)] [s_d])$

is greater than any positive $k$, which is impossible since all sequences in $D'_{\Sigma E}$ are finite. Q.E.D.
A similar construction generates a complete recursive program corresponding to an arbitrary set of call-by-name recursion equations. This construction and a sketch of the corresponding proof are presented in Appendix II.

8. A Sample Proof Involving Complete Recursive Programs

To illustrate how complete programs can be used to prove theorems about recursively defined partial functions, we present the following example. Let $f$ be the partial function on the natural numbers defined by the recursion equation:

$$f(x) = f(x+1)$$

Although $f$ is everywhere undefined in the standard model, we can not establish this property of using ordinary first order semantics since $f$ is total in numerous non-standard models. However, we can prove the equivalent property for the corresponding complete recursive program

$$f'(x) = \text{seq}(x+1) \odot f'(x+1).$$

By the complete recursive program theorem proved in the preceding section, the statement

$$\forall x(x \in \mathbb{N}) \implies \text{last}(f'(x)) = 1 \quad (1)$$

is true in the standard model for $f'$ if and only if the statement

$$\forall x(x \in \mathbb{N}) \implies f(x) = 1 \quad (2)$$

is true in the standard model for $f$.

We prove statement (1) as follows. Since last is strict, statement (1) is an immediate consequence of the lemma

$$\forall x(x \in \mathbb{N}) \implies f'(x) = 1 \quad (3)$$

We prove lemma (3) by structural induction on $f'(x)$. 
Base case $f'(x) = 1$. Trivial.

Induction step. $f'(x) \neq 1$. By hypothesis, the lemma holds for all $x_0$ such that $f'(x_0)$ is a proper tail of $f'(x)$. Since $x \in \mathbb{N}$, $f'(x) = \text{seq}(x) \cdot f'(x+1)$. By hypothesis, $f'(x+1) = 1$. Hence $f'(x) = 1$. Q.E.D.

9. Advantages of First Order Semantics

While first order semantics is more limited in scope than least fixed point logics, we believe it is a simple, intuitively appealing formalism which is well-suited for reasoning about programs written in applicative languages. Unlike proofs in least fixed point logics, proofs in first-order semantics closely correspond to their informal counterparts. As an illustration, consider the sample proof presented in section 4. The proof using first order semantics is a straightforward formalization of the obvious informal proof. In contrast, a proof of the same theorem in a least fixed point logic requires introducing a retraction characterizing the domain of S-expressions and simulating a structural induction by performing fixed point induction on the retraction. The first order semantics proof is significantly shorter and more direct.

Besides facilitating simpler formal proofs, first order semantics avoids the admissibility problem plaguing least fixed point logics. To ensure the soundness of fixed point induction a least fixed point logic must either severely restrict the syntax of formulas (banning negation and general quantification
as in Stanford LCF[11]) or restrict the application of fixed
point induction to formulas satisfying a complex admissibility
criterion (as in Edinburgh LCF [6]).

For the reasons cited above, we believe that first order
semantics—rather than a least fixed point logic—is the ap-
propriate formal system for verifying programs in recursive
languages like LISP. Both [Cartwright 76ab] and [Boyer and
Moore 75] have successfully applied first order semantics to
prove the correctness of moderately complex LISP programs
with relative ease. On the other hand, the feasibility of
using first order semantics to formally reason about non-
trivial partial functions (such as interpreters) has yet to
be investigated. We believe, however, that both McCarthy's
minimization schema and complete recursive program construction
described in this paper show considerable promise as methods
for reasoning about partial functions.
References


APPENDIX I

Sample Proofs in the Logic of First Order Semantics

Example 1: Termination of the countdown function.

Let the (partial) function zero over the natural numbers

N be defined by the call-by-name recursion equation:

zero(n) = if n equal 0 then 0 else zero(n-1).

We will prove that the function zero equals 0 on N, everywhere

i.e.

\[ \forall n [n \neq 1 \Rightarrow \text{zero}(n) = 0]. \]

The proof proceeds by induction on n.

Base case. n=0. In this case,

\[ \text{zero}(n) = \text{if } 0 \text{ equal } 0 \text{ then } 0 \text{ else zero}(n-1) = 0. \]

Induction step. n>0. By hypothesis, the theorem holds

for all n'<n. Since n>0,

\[ \text{zero}(n) = \text{if } n \text{ equal } 0 \text{ then } 0 \text{ else zero}(n-1) = \text{zero}(n-1) \]

which is 0 by hypothesis.

Q.E.D.

Example 2: Termination of an Ackermann function.

Let the (partial) function ack over the natural numbers

N be defined by the call-by-value recursion equation:

\[ \text{ack}(x, y) = \text{if } x \text{ equal } 0 \text{ then } \text{suc}(y) \]

\[ \text{else if } y \text{ equal } 0 \text{ then } \text{ack}(\text{pred}(x), 1) \]

\[ \text{else } \text{ack}(\text{pred}(x), \text{ack}(x, \text{pred}(y))). \]

We will prove that ack is total on N, i.e.

\[ \forall x, y [x \neq 1 \land y \neq 1 \Rightarrow \text{ack}(x, y) \neq 1]. \]

The proof proceeds by induction on the pair \{x, y\}. 
Base case. \( x=0 \). By assumption, \( y \neq 1 \). In this case, \( \text{ack}(x,y) = \text{suc}(y) \neq 1 \).

Induction step. \( x>0 \). By hypothesis, we assume the theorem holds for all \( x' \), \( y' \) such that either \( x'<x \) or \( x'=x \) and \( y'<y \). Since \( y \neq 1 \) by assumption,

\[
\text{ack}(x,y) = \begin{cases} 
\text{if } y \text{ equal } 0 \text{ then } \text{ack}(\text{pred}(x),1) \\
\text{else } \text{ack}(\text{pred}(x), \text{ack}(x,\text{pred}(y)))
\end{cases}
\]

Subcase a. \( y=0 \). In this case, \( \text{ack}(x,y) = \text{ack}(\text{pred}(x),1) \)
which by hypothesis is a natural number (not 1).

Subcase b. \( y>0 \). In this case,

\[
\text{ack}(x,y) = \text{ack}(\text{pred}(x), \text{ack}(x,\text{pred}(y)))
\]

By hypothesis, \( \text{ack}(x,\text{pred}(y)) \) is a natural number implying (by the induction hypothesis) that \( \text{ack}(\text{pred}(x), \text{ack}(x,\text{pred}(y))) \)
is a natural number.

Q.E.D.

Example 3: McCarthy's 91-function.

Let the (partial) function \( \text{f91} \) over the integers be defined by the call-by-name recursion equation

\[
\text{f91}(n) = \begin{cases} 
\text{if } n>100 \text{ then } n-10 \\
\text{else } \text{f91}(\text{f91}(n+1))
\end{cases}
\]

We will prove the following theorem (implying \( \text{f91} \) is total)

\[ \forall n \neq 1 \Rightarrow \text{f91}(n) = \begin{cases} 
\text{if } n>100 \text{ then } n-10 \text{ else } 91.
\end{cases} \]

The proof proceeds by induction on 101\text{en} where the binary operator \( \diamond \) (funny minus) is defined by the equation

\[
\text{\textit{x\diamond y} = if } (x-y)>0 \text{ then } x-y \\
\text{else } 0.
\]
Base case. \(1010n = 0\), i.e. \(n > 100\). In this case,

\[ f_{91}(n) = n - 10 \]

which is exactly what the theorem asserts.

Induction step. \(1010n > 0\), i.e. \(n \leq 100\). By hypothesis, we assume the theorem holds for \(n'\) such that \(1010n' < 1010n\), i.e. \(n' > n\). In this case,

\[ f_{91}(n) = f_{91}(f_{91}(n+11)) \]
\[ = f_{91}(\text{if } n+11 > 100 \text{ then } n+1 \text{ else } 91) \]
\[ \quad \text{(by induction since } n+11 > n). \]

Subcase a. \(n+11 > 100\), i.e. \(n > 89\). In this case,

\[ f_{91}(n) = f_{91}(n+1) \]
\[ = \text{if } n+1 > 100 \text{ then } n - 9 \text{ else } 91 \]
\[ = 91 \text{ (since } n \leq 100). \]

Subcase b. \(n+11 \leq 100\), i.e. \(n \leq 89\). In this case,

\[ f_{91}(n) = f_{91}(91) \]
\[ = \text{if } 91 > 100 \text{ then } 91 - 10 \text{ else } 91 \]
\[ \quad \text{(by induction since } 91 > n) \]
\[ = 91. \]

Q.E.D.
APPENDIX II

Call-by-name Complete Recursive Programs

The call-by-value recursive program construction described in Section 7 exploited the idea of defining a new function \( f'_i \) for each function \( f_i \) in the original program such that \( f'_i \) constructs the call-by-value computation sequence for \( f_i \). We will utilize essentially the same idea in the call-by-name complete recursive program construction.

Unfortunately, call-by-name computation sequences have a more complex structure than the corresponding call-by-value computation sequences. The chief complication is that the set of arguments evaluated in a recursive call \( f_i(t) \) in the original program depends on the particular values of the arguments. The solution is to adopt the convention that the new functions \( f'_i \) take computation sequences for arguments of \( f \) as input instead of the arguments themselves. Consequently, the body of each new function \( f'_i \) is free to incorporate in the final result only the computation sequences for arguments of \( f_i \) which are actually evaluated.

As a result of this complication, the original functions \( f_1, \ldots, f_n \) are related to the constructed functions \( f'_1, \ldots, f'_n \) by the equations:

\[
f_i(x_1, \ldots, x_{m_i}) = \text{last}(f'_i(\text{seq}(x_1), \ldots, \text{seq}(x_{m_i})))
\]

instead of the simpler relationship

\[
f_i(x_1, \ldots, x_{m_i}) = \text{last}(f'_i(x_1, \ldots, x_{m_i}))
\]
which holds for call-by-value complete recursive programs.

Let $D^+$, $L_{D^+}$, $F$, $L_F$, $\hat{f}_i (i=1, \ldots, n)$, $M_k (k=0, 1, 2, \ldots)$, $D_{\text{SEQ}}$ (including $\hat{\Theta}$, $\hat{\text{last}}$, $\hat{\text{seq}}$), and $L_F$ be defined as in section 7 with the single exception that $[\hat{f}_1, \ldots, \hat{f}_n]$ is the standard call-by-name (rather than call-by-value) least fixed solution over $D^+$ of the system of recursion equations $F$, i.e. that $[\hat{f}_1, \ldots, \hat{f}_n]$ is the least upper bound of the ascending sequence of function $n$-tuples $[\hat{f}_{1(k)}, \ldots, \hat{f}_{n(k)}]$, $k=0, 1, \ldots$

where

\[
\hat{f}_i (\bar{a}) = 1 \text{ for all } m_i\text{-tuples } \bar{a} \text{ over } D^+. \\
\hat{f}_i (0) (k)
\]

\[
\hat{f}_i (\bar{a}) = M_{k-1} [t_i] [s_d]
\]

where $s_d$ is a state binding $\bar{x}_i$ to $\bar{a}$ and $M_k$ is defined in terms of $\hat{f}_{i(k)}$, $i=1, \ldots, n$ exactly as in section 7.

We construct the call-by-name complete recursive program $F'$ corresponding to $F$ as follows. Let $t$ be an arbitrary term in the language $L_F$. The call-by-name computation sequence $t\mapsto t'$ (in the extended language $L_{F'}$) corresponding to $t$ is inductively defined by:

1. If $t$ is a variable $v$, $t'=v$.
2. If $t$ is a constant symbol $c$, $t'=\text{seq}(c)$.
3. If $t$ has the form $g(u_1, \ldots, u_m)$ where $g\in G$,
   
   \[
t'= u_1 \circ \ldots \circ u_m \circ \text{seq}(g(\text{last}(u_1'), \ldots, \text{last}(u_m'))).
\]
4. If $t$ has the form $f_i(u_1, \ldots, u_m)$,
   
   \[
t = \text{seq}(d') \circ f_i(u_1', \ldots, u_m')
\]
   
   where $d'$ is any element of $D$.
5. If $t$ has the form if $u_0$ then $u_1$ else $u_2$,
   
   \[
t' = u_0' \circ (\text{if last}(u_0') \text{ then } u_1' \text{ else } u_2').
\]
The complete recursive program $F'$ corresponding to $F$ is the set of recursion equations $\{f'_1(x_1) = t'_1, \ldots, f'_n(x_n) = t'_n\}$.

**Theorem.** The call-by-name complete recursive program $F'$ constructed from the set of recursion equations $F$ has the following properties:

1. $\text{last}(\hat{f}'_i(\text{seq}(d_1), \ldots, \text{seq}(d_{m_i}))) = \hat{f}'_i(d_1, \ldots, d_{m_i})$ for all $m_i$-tuples $[d_1, \ldots, d_{m_i}]$ over $D^+$.

2. $F'$ has a unique (call-by-name) least fixed point solution.

**Proof.** The proof follows the same outline as the corresponding call-by-value proof in Section 7. Let $M'_k$ denote the call-by-name meaning function for $F'$ analogous to $M_k$ for $F$, $k=0,1,\ldots$. To prove property 1, we first prove the lemma:

**Lemma 1'.** For every term $t$ in $L_F$ and every state $s$ over $D^+$,

$$M'_k[t][s] = M'_k[\text{last}(t')][s_{\text{seq}}]$$

where $s_{\text{seq}}$ is the state mapping each variable $x$ into $\text{seq}(s(x))$.

**Proof of lemma.** Since the proof of lemma 1' follows the same lines as the proof of lemma 1 in section 7, it is omitted.

Property 1 follows immediately from lemma 1' by the following argument. For any $m_i$-tuple $d=[d_1, \ldots, d_{m_i}]$ over $D^+$ there exists $k_0>0$ such that $\hat{f}'_i(k_0) = \hat{f}'_i(d)$ and

$$\hat{f}'_i(k_0)(\text{seq}(d_1), \ldots, \text{seq}(d_{m_i})) = \hat{f}'_i(\text{seq}(d_1), \ldots, \text{seq}(d_{m_i}))$$

Let $s$ be a state mapping $\hat{f}'_i$ into $d$ and $s_{\text{seq}}$ be the state mapping each variable $x$ into $\text{seq}(s(x))$. 
Then \( f_i(\vec{d}) = f_i(k_0) = M'_{k_0} [f_i(\vec{x})][s] \)
\[ = M'_{k_0} [\text{last(seq}(d)) \circ f_i(\vec{x})][s_{\text{seq}}] \]
\[ = M'_{k_0} \text{last}(f_i(\vec{x}))[s_{\text{seq}}] \]
\[ = \text{last}(\hat{f}_i^{(k_0)}(\vec{d}), \text{seq}(d_1), ..., \text{seq}(d_m))) \]
\[ = \text{last}(\hat{f}_i^{(k_0)}(\text{seq}(d_1), ..., \text{seq}(d_m))). \]

To prove property 2 we must utilize the definitions introduced for the analogous proof in Section 7. Let \( H, \ M'_k \ (k=0,1,\ldots, \), \( \hat{f}^i_k \ (k=0,1,\ldots; i=1,\ldots,n) \) be defined exactly as in Section 7, except that \( \hat{f}^i_k \) and \( M'_k \) are defined using call-by-name semantics instead of call-by-value semantics, i.e.
\[ \hat{f}^i_0 = h_i \]
\[ \hat{f}^i_k(\vec{d}) = M'_{k-1} [\hat{f}^i_k][s_d] \text{ for all } k>0, \text{ all } m_i\text{-tuples } \vec{d} \text{ over } D_{\text{SEQ}}^+ \text{ where } s_d \text{ maps } \vec{x}_i \text{ into } \vec{d}. \]

The critical lemma for proving property 2 is lemma 2' which is identical to lemma 2 in Section 7 (although the definitions of \( \hat{f}^i_k, \hat{f}^i_k, M'_k, \text{ and } M'_k \) are different). Since the proof of lemma 2' is very similar to the proof of lemma 2, it is omitted.

Property 2 follows immediately from lemma 2' by the same argument used to prove property 2 from lemma 2 in Section 7. Q.E.D.