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 $\underline{D}$ cannot be expressed by a sentence in a first order theory[1] of $\underline{D}$ 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 DeRoever [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:[2]

$$\text{zero}(n) = \text{if } n=0 \text{ then } 0 \text{ else } \text{zero}(n-1). \qquad (*)$$

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 $T\cup\{(*)\}$.. 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 θ provable from $T\cup\{(*)\}$ can state zero is total since θ 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:

$$\forall n \, [\ (n=0) \supset \text{zero}(n)=0) \wedge (\neg(n=0) \supset \text{zero}(n)=\text{zero}(n-1))].$$

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 $T\cup\{(*)\}$[3]. Consequently, the function zero is identically zero in every model of $T\cup\{(*)\}$. Furthermore, since the models of $T\cup\{(*)\}$ do not contain a special undefined element (usually denoted $\perp$), 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 $N\cup\{\perp\}$ (where $\perp$ 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 $\perp$). We can assert (within the corresponding first-order language) that f is total on N by simply stating:

$$\forall x_1,\ldots,x_n[(x_1\neq\perp)\wedge\ldots\wedge(x_n\neq\perp) \supset f(x_1,\ldots,x_n)\neq\perp].$$

Let $T_\perp$ be a "Peano-like" axiomatization (including an induction axiom schema) for $N\cup\{\perp\}$.

Given $T_\perp$, and the recursion equation (*), we can easily establish that Hitchcock and Park's zero function is total on N by proving the sentence $\forall n[n\neq\perp) \supset zero(n)\neq\perp]$. 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,...,x,aa,... 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 $\perp$ 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 $\perp$ 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 $\perp$ 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[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 $D-\{\bot\}$ into two non-empty subsets D_T and D_F consisting of the objects representing true and false respectively. For example, if our data domain is simply the natural numbers N augmented by $\bot$, then we could let $D_F = \{0\}$ and $D_T = \{x \in N \mid x \neq 0\}$. In LISP, $D_F = \{NIL\}$ and $D_T = \{x \in \text{S-expressions} \mid x \neq NIL\}$.

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

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

---

[5] An n-ary function is strict if it is undefined ($\bot$) whenever any of its arguments is undefined ($\bot$).

$$f_1(x_1,\ldots,x_{m_1}) = \tau_1$$
$$f_2(x_1,\ldots,x_{m_2}) = \tau_2$$
$$\cdots$$
$$f_n(x_1,\ldots,x_{m_n}) = \tau_n$$

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 τ_i contains no variables other than $x_1, \ldots, x_{m_i}$; $\underline{i} = 1, 2, \ldots, n$.

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 flatl be recursively defined functions over the domain of S-expressions defined by the following equations:

```
flat(x)=flatl(x,NIL).
flatl(x,y)=if atom x. then cons(x,y)
           else flatl(car x,flatl(cdr x,y)).
```

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

```
flat((A.B))= (A B)
flat((A.(B.A))) = (A B A)
flat(A) = (A)
flat(((A.C).B)) = (A C B)
```

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

$$x \neq \perp \wedge y \neq \perp \supset \text{flatl}(x,y) \neq \perp .$$

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

$$\text{is-Sexpr}(x) \wedge \text{is-Sexpr}(y) \supset \text{is-Sexpr}(\text{flatl}(x,y)).$$

We prove the theorem by applying structural induction on x.

Base-case. x is an atom.

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

Induction-step. Given

1) $\forall y$ is-Sexpr(x1) $\wedge$ is-Sexpr(y) $\supset$ is-Sexpr(flatl(x1,y)), and

2) $\forall y$ is-Sexpr(x2) $\wedge$ is-Sexpr(y) $\supset$ is-Sexpr(flatl(x2,y))

we must show

$\forall y$ is-Sexpr(cons(x1,x2)) $\wedge$ is-Sexpr(y) $\supset$
    is-Sexpr(flatl(cons(x1,x2),y))

In this case, flatl(cons(x1,x2),y) = flatl (x1,flatl,(x2,y)) Since cons(x1,x2) is an S-expression, x1 and x2 must be S-expressions. Hence, by induction hypothesis 1, flatl(x2,y) is an S-expression. Given this fact we can apply induction hypothesis 2 to deduce that flatl(x1,flat(x2,y)) is an S-expressic 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 $D-\{\perp\}$, 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

$$f(x) = f(x) \qquad (***)$$

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 Φ_f (containing a free function symbol) for each function f defined in the recursive program. The schema Φ_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.[6] 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[7] solution of the recursion equations.

2. For each recursion equation

   $$f(x_1,\ldots,x_n) = \tau$$

   we add the two axioms

   $$x_1 \neq \perp \wedge \ldots \wedge x_n \neq \perp \supset f(x_1,\ldots,x_n)=\tau$$

   $$x_1 = \perp \vee \ldots \vee x_n = \perp \supset f(x_1,\ldots,x_n)=\perp$$

   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.

---

[7]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 $f(x_1,\ldots,x_n)=$ if $\partial(x_1)\wedge\ldots\wedge\partial(x_n)$ then τ else $\perp$

where ∂ is the primitive "is-defined" function with the property: $\partial(x)$ = true if $x \neq \perp$
$\partial(\perp) = \perp$

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

$$x \sqsubseteq y \text{ iff } x = \perp .$$

The set D^+ is clearly a complete partial order[8] under the partial ordering $\sqsubseteq$.

---

[3] A set S under the partial order $\sqsubseteq$ is a complete partial order iff

1) is a partial ordering (for a definition, see [8])
2) every ascending sequence $x_0 \sqsubseteq x_1 \sqsubseteq x_2$ ... has a least upper bound.

Let F be a set of recursion equations $\{f_1(\bar{x}_1) = t_1, \ldots, f_n(\bar{x}_n) = t_n\}$ where $\bar{x}_i$ 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 $\bar{x}_i$ appear in t_i, $i=1,\ldots,n$.

The call-by-value least fixed point solution of F (denoted $[\hat{f}_1, \ldots, \hat{f}_n]$) is the least upper bound of the ascending sequence of n-tuples of functions $[f_{1_{(0)}}, \ldots, f_{n_{(0)}}] \subseteq [\hat{f}_{1_{(1)}}, \ldots, \hat{f}_{n_{(1)}}] \subseteq \ldots \subseteq [\hat{f}_{1_{(k)}}, \ldots, \hat{f}_{n_{(k)}}] \subseteq \ldots$ 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:

$$\hat{f}_{i_{(0)}}(\bar{d}) = \perp \text{ for all tuples } \bar{d} \text{ over } D^+.$$

$$\hat{f}_{i_{(k)}}(\bar{d}) = \begin{cases} M_{k-1} [\![t_i]\!] [s] & \text{for all tuples } \bar{d} \text{ over } D. \\ \perp & \text{otherwise} \end{cases}$$

where s is a state vector mapping the variables $\bar{x}_i$ into $\bar{d}$; and $M_k [\![t_i]\!] [s]$, $k \geq 0$ 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_F to accomodate sequences. Let SEQ(D) denote the set of finite, non-empty sequences over D and D^+_{SEQ} denote the extended data domain $D^+ \cup$ SEQ(D). We extend every function $\hat{g} \epsilon G$ to the domain D^+_{SEQ} as follows:

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

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

$$[a_1,\ldots,a_\ell] \hat{\oplus} [b_1,\ldots,b_m] = [a_1,\ldots,a_\ell,b_1,\ldots,b_m].$$

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

$$x\hat{\oplus}y = \bot \text{ if } x \epsilon \text{SEQ}(D) \text{ or } y \epsilon \text{SEQ}(D)$$

$$\widehat{last}([a_1,\ldots,a_\ell]) = a_\ell \text{ for } [a_1,\ldots,a_\ell] \in \text{SEQ}(D)$$

$$\widehat{last}(x) = \bot \text{ for } x \notin \text{SEQ}(D)$$

$$\widehat{seq}(d) = [d] \text{ for } d \epsilon D$$

$$\widehat{seq}(x) = \bot \text{ for } x \notin D$$

Let $L_{D^+_{SEQ}}$ denote the first order language $L_{D^+} \cup \{\oplus, last, seq\}$ corresponding to D^+_{SEQ}.

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 $L_{F'} = L_{D^+_{SEQ}} \cup \{f_1',\ldots,f_n'\}$) corresponding to t is inductively defined as follows:

1. If t is a constant or a variable x,

$$t' = seq(x).$$

2. If t has the form $g(u_1,\ldots u_m)$ where $\hat{g} \epsilon G$,

$$t' = u_1' \oplus \ldots \oplus u_m' \oplus seq(g(last(u_1'),\ldots,last(u_m'))).$$

3. If t has the form $f_i(u_1,\ldots,u_m)$,

$$t' = u_1' \oplus \ldots \oplus u_m' \oplus f_i'(last(u_1'),\ldots, last(u_m')).$$

4. If t has the form if u_0 then u_1 else u_2,

$$t' = u_0' \oplus (\text{if } last(u_0') \text{ then } u_1' \text{ else } u_2').$$

The complete recursive program F' corresponding to F is the set of recursion equations $\{f_1'(\bar{x}_1) = t_1', \ldots, f_n'(\bar{x}_n) = t_n'\}$.

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

1. $\widehat{last}(\hat{f}_i'(\bar{d})) = \hat{f}_i(\bar{d})$ for all m_i-tuples $\bar{d}$ 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 1. For every term t in L_F and every state s over D^+,

$$M_k [\![t]\!] [s] = M_k' [\![last(t')]\!] [s], \ \forall k \geq 0.$$

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 seq(x) implying $M_k' [\![last(t')]\!] [s] = M_k' [\![x]\!] [s] = M_k [\![t]\!] [s]$ for arbitrary $k \geq 0$.

Case 2. t has the form $g(u_1, \ldots, u_m)$, $g \in G \cup \{\hat{\oplus}, \widehat{last}, \widehat{seq}\}$.

Then $t' = u_1' \oplus \ldots \oplus u_m' \oplus seq(g(last(u_1'), \ldots, last(u_m')))$.

By hypothesis $M_k' [\![last(u_i')]\!] [s] = M_k [\![u_i]\!] [s]$, for all s, $i = 1, \ldots, m$.

If for some i, $M_k [\![u_i]\!] [s] = \bot$ then $M_k' [\![last(u_i')]\!] = \bot$

implying $M'_k \llbracket \mathrm{last}(t') \rrbracket [s] = \bot$ (since $\hat{g}$, $\hat{\Phi}$, and $\widehat{\mathrm{last}}$ are strict). Hence we assume $M_k \llbracket u_i \rrbracket [s] \neq \bot$ for all i. By the induction hypotheses and the definition of $\widehat{\mathrm{last}}$, we conclude $M'_k \llbracket u'_i \rrbracket [s] \in SEQ(D)$ for all i, implying

$$\begin{aligned} M'_k\llbracket \mathrm{last}(t')\rrbracket[s] &= M'_k\llbracket g(\mathrm{last}(u'_1),\dots,\mathrm{last}(u_m'))\rrbracket[s] \\ &= M_k\llbracket g(u_1,\dots,u_m)\rrbracket[s] \quad \text{(by induction)} \\ &= M_k\llbracket t\rrbracket[s]. \end{aligned}$$

Case 3. t has the form $f_j(u_1,\dots,u_m)$. By an argument analogous to the one presented in the previous case, we can assume $M_k\llbracket u_i\rrbracket[s] \neq \bot$ and $M'_k\llbracket u'_i\rrbracket[s] \in SEQ(D)$ for all i. Consequently,

$$\begin{aligned} M'_k\llbracket \mathrm{last}(t')\rrbracket[s] &= M'_k\llbracket \mathrm{last}(f'_j(\mathrm{last}(u'_1),\dots,\mathrm{last}(u_m')))\rrbracket[s] \\ &= M_k'\llbracket \mathrm{last}(f'_j(x_1,\dots,x_m))\rrbracket[s'] \end{aligned}$$

where s' is a state vector binding x_i to $M'_k\llbracket \mathrm{last}(u'_i)\rrbracket[s] = M_k\llbracket u_i\rrbracket[s]$, $i=1,\dots,m_j$. Since no x_i is bound to $\bot$, we may expand $f'_j(x_1,\dots,x_m)$ yielding

$$\begin{aligned} M'_k\llbracket \mathrm{last}(t')\rrbracket[s'] &= M'_{k-1}\llbracket \mathrm{last}(t'_j)\rrbracket[s'] \\ &= M_{k-1}\llbracket t_j\rrbracket[s'] \quad \text{(by induction)} \\ &= M_k\llbracket f_j(x_1,\dots,x_m)\rrbracket[s'] \\ &= M_k\llbracket t\rrbracket[s]. \end{aligned}$$

Case 4. t has the form if u_0 then u_1 else u_2. By definition $t' = u_0' \Phi (\text{if } \mathrm{last}(u'_0) \text{ then } u'_1 \text{ else } u'_2)$.

Subcase a. $M_k\llbracket u_0\rrbracket[s] = \bot$. By induction $M'_k\llbracket \mathrm{last}(u'_0)\rrbracket = \bot$. Since $\hat{\Phi}$ and $\widehat{\mathrm{last}}$ are strict, and $\widehat{\text{if-then-else}}$ is strict in its first argument, $M'_k\llbracket \mathrm{last}(t')\rrbracket[s] = \bot$ and $M_k\llbracket t\rrbracket[s] = \bot$.

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

$$M_k\llbracket u_0\rrbracket[s] = M'_k\llbracket \mathrm{last}(u_0')\rrbracket[s] \text{ implying}$$

$$\begin{aligned} M'_k\llbracket t'\rrbracket[s] &= M'_k\llbracket \mathrm{last}(u_1')\rrbracket[s] \\ &= M_k\llbracket u_1\rrbracket[s] \quad \text{(by induction)} \\ &= M_k\llbracket t\rrbracket[s]. \end{aligned}$$

An analogous argument proves the "false" case. Q.E.D.

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

$$M_k[\![f_i(x_1,\ldots,x_{m_i})]\!][s] = M'_k[\![\text{last}(\text{seq}(x_1)\oplus\ldots\oplus\text{seq}(x_{m_1})\oplus f'(\text{last}(\text{seq}(x_1)),\ldots,\text{last}(\text{seq}(x_n))))]\!][s]$$

for all $k\geq 0$, all state vectors s over D^+. Property 1 trivially holds for m_i-tuples $\bar{d}$ where some element of $\bar{d}$ is $\bot$ since the functions $\hat{f}_i$, $\hat{f}'_i$ and $\widehat{\text{last}}$ 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

$$M_k[\![f_i(x_i,\ldots,x_{m_i})]\!][s] = M'_k[\![\text{last}(f'(x_1,\ldots,x_n))]\!][s]$$

for state vectors s binding $x_1,\ldots,x_{m_i}$ to values in D. Since D^+ and D^+_{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_0 such that $\hat{f}_{i_{(k_0)}}(\bar{d}) = \hat{f}_i(\bar{d})$ and $\hat{f}'_{i_{(k_0)}}(\bar{d}) = \hat{f}'_i(\bar{d})$. Let $\bar{d}$ be an arbitrary m_i-tuple over D and s be a state mapping $\bar{x}_i$ into $\bar{d}$.

Then

$$\hat{f}_i(\bar{d}) = \hat{f}_{i_{(k_0)}}(\bar{d}) = M_{k_0}[\![f_i(x_1,\ldots,x_{m_i}]\!][s] =$$

$$M'_{k_0}[\![\text{last}(f'_i(x_1,\ldots,x_{m_i}))]\!][s] = \widehat{\text{last}}(\hat{f}'_{i_{(k_0)}}(\bar{d})) = \widehat{\text{last}}(\hat{f}'_i(\bar{d}))$$

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^+_{SEQ} corresponding to the function symbols $f'_1,\ldots,f'_n$. We define M'_{k_H} $k>0$ as the meaning function for terms of L_F identical to M_k except that M'_{k_H} interprets f'_i, $i=1,\ldots,n$ by $\hat{f}'_{i_{(k)_H}}$ where $\hat{f}'_{i_{(k)_H}}$

is inductively defined by

$$\hat{f}'_{i_{(0)_H}} = \hat{h}_i$$

$$\hat{f}'_{i_{(k)_H}}(\bar{d}) = \begin{cases} \bot \text{ if } \bot \in \bar{d} \\ M'_{k-1_H}[\![t'_i]\!][s_d] \text{ otherwise} \end{cases}$$

where s_d maps $\bar{x}_i$ into $\bar{d}$. Informally, $\hat{f}'_{i_{(k)_H}}$ 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 L'_F. 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 D^+_{SEQ} $M'_k[\![t]\!][s] = \bot$ implies either $M'_{k_H}[\![t]\!][s] = \bot$ or $\widehat{\text{length}}(M_{k_H}[\![t]\!][s]) \geq k$ where $\widehat{\text{length}}$ is the function mapping D^+_{seq} into the extended natural members $(N\cup\{\bot\})$ defined by

$$\widehat{\text{length}}(\bot) = \bot$$

$$\widehat{\text{length}}(d) = 0 \text{ for } d \in D$$

$$\widehat{\text{length}}([d_1,\ldots,d_\ell]) = \ell \text{ for } [d_1,\ldots,d_\ell] \in SEQ(D).$$

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 L_F', $M'_k[\![t]\!][s] \neq \bot$ 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_i', $M_{k_H}[\![t]\!] = M_k[\![t]\!]$.

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 $\widehat{\text{length}}$.

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

$$M'_{k_H}[\![t']\!][s] = M'_{k_H}[\![seq(x)]\!][s] = M'_k[\![seq(x)]\!][s] = \widehat{seq}(M_k[\![x]\!][s]).$$

Consequently, if $M_k[\![x]\!][s] = \bot$, then $M'_{k_H}[\![seq(x)]\!][s] = \bot$.

Subcase b. t has the form $g(u_1,\ldots,u_m)$ where $\hat{g}\in G$. In this case, $t' = u_1'\oplus\ldots\oplus u_m'\oplus seq(g(last(u_1'),\ldots,last(u_m')))$. If $M_k[\![u_j]\!][s] = \bot$ for some j, then by induction $M'_{k_H}[\![u_j']\!] = \bot$ or $\widehat{\text{length}}\,(M'_{k_H}[\![u_j]\!][s]) \geq k$, implying the lemma holds (since $\hat{\oplus}$ is strict). On the other hand, if $M_k[\![u_j]\!][s]\neq\bot$ for all j, then by lemmas 1 and 3a, $M'_{k_H}[\![u_j]\!][s] = M'_k[\![u_j]\!][s] \in SEQ(D)$ for all j. Consequently,

$$\begin{aligned} M'_{k_H}[\![t']\!][s] &= M'_{k_H}[\![u_1'\oplus\ldots\oplus u_m'\oplus seq(g(last(u_1'),\ldots,last(u_m')))]\!][s] \\ &= M'_k[\![u_1'\oplus\ldots u_m'\oplus seq(g(last(u_1'),\ldots,last(u_m')))]\!][s] \\ &= M'_k[\![t']\!][s]. \end{aligned}$$

If $M_k[\![t]\!][s] = \bot$, then by lemma 1 $M_k'[\![t']\!][s] = \bot$ implying $M'_{k_H}[\![t']\!][s] = \bot$.

Subcase c. t has the form $f_i(u_1,\ldots,u_{m_i})$. If $M_k[\![u_j]\!][s] = \bot$ for some j, the proof is identical to the analogous section of the previous case. On the other hand, when $M_k[\![u_j]\!][s] \neq \bot$ for all j,

$$\begin{aligned} M'_{k_H}[\![t']\!][s] &= M'_k[\![u_1']\!][s]\hat{\oplus}\ldots\hat{\oplus}M'_k[\![u'_{m_i}]\!][s]\hat{\oplus} \\ &\quad \hat{f}_{i_{k_H}}(\widehat{\text{last}}\,(M'_k[\![u_1']\!][s]),\ldots,\widehat{\text{last}}(M'_k[\![u'_{m_i}]\!][s])) \\ &= M'_{k-1_H}[\![x_1\oplus\ldots\oplus x_{m_i}\oplus t_i']\!][s'] \end{aligned}$$

where s' maps x_j into $M_k'[\![u_j']\!][s]$.

By induction $M'_{k-1_H}[\![t'_i]\!][s'] = \bot$ or $\widehat{\text{length}}\ (M'_{k-1_H}[\![t'_i]\!][s'] \geq k-1$. Hence the lemma clearly holds (since each x_j has length ≥ 1).

Subcase d. t has the form if u_0 then u_1 else u_2. If $M_k[\![u_0]\!][s] = \bot$, the proof is identical to the analogous section of subcase b. On the other hand, when $M_k[\![u_0]\!][s] \neq \bot$, 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 $M'_{k_H}[\![t']\!][s] = M'_{k_H}[\![u'_1]\!][s]$. By induction, $M_k[\![u_1]\!] = \bot$ implies either $M'_{k_H}[\![u'_1]\!] = \bot$ or $\widehat{\text{length}}\ (M'_{k_H}[\![u'_1]\!]) \geq k$. 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 $H = \{\hat{h}_1,\ldots,\hat{h}_n\}$ be any call-by-value fixed point solution of the set of recursive equations F', i.e. for $i=1,\ldots,n$

$$M'_{0_H}[\![f'_i(\bar{x}_i)]\!] = M'_{0_H}[\![t'_i]\!].$$

By induction on $[k,t']$ we can easily show for all $k\geq 0$ and all terms t' in $L_F{}'$, $M'_{k_H}[\![t']\!] = M'_{0_H}[\![t']\!]$

Now assume H is not the least call-by-value fixed point solution of F', i.e. $f'_i(\bar{d}) = \bot$ but $h_i(\bar{d}) \neq \bot$ for some i, some $\bar{d}\in D^+_{SEQ}$. Then $\forall k\geq 0\ M'_{k_H}[f'_i(\bar{x}_i)][s_d] = M'_{0_H}[f'_i(\bar{x}_i)][s_d]$ where s_d binds $\bar{x}_i$ to $\bar{d}$. Hence the length of $\hat{h}'_i(\bar{d})$, i.e. $\widehat{\text{length}}\ (M'_{0_H}[\![f'_i(\bar{x}_i)]\!][s_d])$ is greater than any positive k, which is impossible since all sequences in D^+_{SEQ} 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

$$f'(x) = seq(x+1) \oplus f'(x+1),$$

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

$$\forall x(x \in N) \supset last(f'(x)) = \bot \qquad (1)$$

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

$$\forall x(x \in N) \supset f(x) = \bot \qquad (2)$$

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

$$\forall x(x \in N) \supset f'(x) = \bot \qquad (3)$$

We prove lemma (3) by structural induction on f'(x).

Base case $f'(x) = \perp$. Trivial.

Induction step. $f'(x) \neq \perp$. By hypothesis, the lemma holds for all x_0 such that $f'(x_0)$ is a proper tail of $f'(x)$. Since $x \in N$, $f'(x) = seq(x) \bullet f'(x+1)$. By hypothesis, $f'(x+1) = \perp$. Hence $f'(x) = \perp$. 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 constrast, 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 severly restrict the syntax of formulas (banning negation and general quantication 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 appropriate 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

[1] R. Boyer and J. Moore. Proving Theorems about LISP Functions, JACM 22, 1 (January 1975), pp. 129-144.

[2] R. Cartwright. User-Defined Data Types as an Aid to Verifying LISP Programs. Automata, Languages, and Programming, S. Michaelson and R. Milner, eds. Edinburgh Press, Edinburgh, 1976, pp. 228-256.

[3] R. Cartwright. A Practical Formal Semantic Definition and Verification System for TYPED LISP, Stanford A. I. Memo AIM-296, Stanford University, Stanford, California, 1976.

[4] J.W. DeBakker. The Fixed Point Approach in Semantics: Theory and Applications, Mathematical Centre Tracts 63, Free University, Amsterdam, 1975.

[5] J.W. DeBakker and W.P. DeRoever. A Calculus for Recursive Program Schemes. Automata, Languages, and Programming, M. Nivat, ed. North-Holland, Amsterdam, 1973, pp. 167-196.

[6] M. Gordon, R. Milner, and C. Wadsworth. EDinburgh LCF. Edinburgh Technical Report CSR-11-77, University of Edinburgh, Edinburgh, 1977.

[7] P. Hitchcock and D.M.R. Park. Induction Rules and Proofs of Program Termination. Automata, Languages, and Programming, M. Nivat, ed. North-Holland, Amsterdam, 1973, pp. 225-251.

[8] Z. Manna. Introduction to the Mathematical Theory of Computation, McGraw-Hill, New York, 1974.

[9] Z. Manna and J. Vuillemin. Fixpoint Approach to the Theory of Computation, CACM 15 (1972), pp. 528-536.

[10] J. McCarthy. Representation of Recursive Programs in First Order Logic, unpublished draft, Stanford University, Stanford, California, 1977.

[11] R. Milner. Logic for Computable Functions--Description of a Machine Implementation, Stanford A.I. Memo AIM-169, Stanford University, Stanford, California, 1972.

[12] R. Milner. Models of LCF, Stanford A.I. Memo AIM-186, Stanford University, Stanford, California, 1973.

[13] R. Milner, L. Morris, and M. Newey. A Logic for Computable Functions with Reflexive and Polymorphic Types. Proceddings Conference on Proving and Improving Programs, Arc-et-Senons, 1975.

[14] Park, D.M.R. Fixpoint Induction and Proff of Program Semantics, in Machine Intelligence 5, B. Meltzer and D. Michie (eds.), Edinburgh Press, 1970, pp. 59-78.

[15] Scott, D. and J.W. DeBakker. A Theory of Programs, unpublished notes, IBM seminar, Vienna, 1969.

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.

$\forall n\ [n \neq \bot \supset \text{zero}(n) = 0].$

The proof proceeds by induction on n.

Base case. n=0. In this case,

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

Induction step. n>0. By hypothesis, the theorem holds for all n'<n. Since n>0,

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

ack(x,y) = if x equal 0 then suc(y)
else if y equal 0 then ack(pred(x),1)
else ack(pred(x), ack(x,pred(y))).

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

$\forall x,y\ [x \neq \bot \wedge y \neq \bot \supset \text{ack}(x,y) \neq \bot].$

The proof proceeds by induction on the pair [x,y].

Base case. $x=0$. By assumption, $y \neq \bot$. In this case, $ack(x,y) = suc(y) \neq \bot$.

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 $y \neq \bot$ by assumption,

```
ack(x,y) = if y equal 0 then ack(pred(x),1)
                  else ack(pred(x), ack(x,pred(y)))
```

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

Subcase b. $y>0$. In this case,

```
ack(x,y) = ack(pred(x), ack(x,pred(y)).
```

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

Q.E.D.

Example 3: McCarthy's 91-function.

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

```
f91(n) = if n>100 then n-10
             else f91(f91(n+1))
```

We will prove the following theorem (implying f91 is total)

$$\forall n\ n \neq \bot \supset f91(n) = \text{if } n>100 \text{ then } n-10 \text{ else } 91.$$

The proof proceeds by induction on $101 \ominus n$ where the binary operator $\ominus$ (funny minus) is defined by the equation

```
xΘy = if (x-y) >0 then x-y
          else 0.
```

Base case. $101 \ominus n = 0$, i.e. $n > 100$. In this case,

$$f91(n) = n-10$$

which is exactly what the theorem asserts.

Induction step. $101 \ominus n > 0$, i.e. $n \le 100$. By hypothesis, we assume the theorem holds for n' such that $101 \ominus n' < 101 \ominus n$, i.e. $n' > n$. In this case,

$$\begin{aligned} f91(n) &= f91(f91(n+11)) \\ &= f91(\text{if } n+11>100 \text{ then } n+1 \text{ else } 91) \\ &\qquad \text{(by induction since } n+11>n\text{)}. \end{aligned}$$

Subcase a. $n+11>100$, i.e. $n>89$. In this case,

$$\begin{aligned} f91(n) &= f91(n+1) \\ &= \text{if } n+1>100 \text{ then } n-9 \text{ else } 91 \\ &= 91 \text{ (since } n \le 100\text{)}. \end{aligned}$$

Subcase b. $n+11 \le 100$, i.e. $n \le 89$. In this case,

$$\begin{aligned} f91(n) &= f91(91) \\ &= \text{if } 91>100 \text{ then } 91-10 \text{ else } 91 \\ &\qquad \text{(by induction since } 91>n\text{)} \\ &= 91. \end{aligned}$$

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 $f_i(\bar{t})$ 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:

$$f_i(x_1,\ldots,x_{m_i}) = \mathrm{last}(f_i'(\mathrm{seq}(x_1),\ldots,\mathrm{seq}(x_{m_i})))$$

instead of the simpler relationship

$$f_i(x_1,\ldots,x_{m_i}) = \mathrm{last}(f_i'(x_1,\ldots,x_{m_i}))$$

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^+_{SEQ} (including $\hat{\oplus}$, $\widehat{last}$, $\widehat{seq}$), 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

$$\hat{f}_{i_{(0)}}(\bar{d}) = \perp \text{ for all } m_i\text{-tuples } \bar{d} \text{ over } D^+.$$

$$\hat{f}_{i_{(k)}}(\bar{d}) = M_{k-1} [\![t_i]\!] [s_d]$$

where s_d is a state binding $\bar{x}_i$ to $\bar{d}$ 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 term 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'=seq(c).
3. If t has the form $g(u_1,\ldots,u_m)$ where $\hat{g}\in G$,
   $t' = u_1' \oplus\ldots\oplus u_m' \oplus seq(g(last(u_1'),\ldots,last(u_m')))$.
4. If t has the form $f_i(u_1,\ldots,u_m)$,
   $t = seq(d') \oplus f_i(u_1',\ldots,u_m')$
   where d' is any element of D.
5. If t has the form if u_0 then u_1 else u_2,
   $t' = u_0' \oplus$ (if last (u_0') then u_1' else u_2').

The complete recursive program F' corresponding to F is the set of recursion equations $\{f'_1(\bar{x}_1) = t'_1, \ldots, f'_n(\bar{x}_n) = t'_n\}$.

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

1. $\widehat{\text{last}}\ (\hat{f}'_i\ (\widehat{\text{seq}}(d_1), \ldots, \widehat{\text{seq}}(d_{m_i}))) = \hat{f}_i(d_1, \ldots, d_{m_i})$ 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^+,

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

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

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 $\bar{d}=[d_1, \ldots, d_{m_i}]$ over D^+ there exists $k_0>0$ such that $\hat{f}_{i_{(k_0)}}(\bar{d}) = \hat{f}_i(\bar{d})$ and

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

Let s be a state mapping $\bar{x}_i$ into $\bar{d}$ and s_{seq} be the state mapping each variable x into $\widehat{\text{seq}}(s(x))$.

$$\text{Then } f_i(\bar{d}) = f_{i_{(k_0)}}(\bar{d}) = M_{k_0}[\![f_i(\bar{x}_i)]\!][s]$$
$$= M'_{k_0}[\![\text{last}(\text{seq}(d')\oplus f'_i(\bar{x}_i))]\!][s_{seq}]$$
$$= M'_{k_0}[\![\text{last}(f'_i(\bar{x}_i))]\!][s_{seq}]$$
$$= \widehat{\text{last}}\,(\hat{f}'_{i_{(k_0)}}(\widehat{\text{seq}}(d_1),\ldots,\widehat{\text{seq}}(d_{m_i})))$$
$$= \widehat{\text{last}}\,(\hat{f}'_i(\widehat{\text{seq}}(d_1),\ldots,\widehat{\text{seq}}(d_{m_i}))).$$

To prove property 2 we must utilize the definitions introduced for the analogous proof in Section 7. Let H, M'_{k_H} (k=0,1,...), $\hat{f}'_{i_{(k)_H}}$ (k=0,1,...; i=1,...,n) be defined exactly as in Section 7, except that $\hat{f}'_{i_{(k)_H}}$ and M'_{k_H} are defined using call-by-name semanties instead of call-by-value semantics, i.e.

$$\hat{f}'_{i_{(0)_H}} = \hat{h}_i$$

$$\hat{f}_{i_{(k)_H}}(\bar{d}) = M_{k-1_H}[\![t'_i]\!][s_d] \text{ for all } k>0, \text{ all } m_i\text{-tuples}$$

$\bar{d}$ over D^+_{SEQ} where s_d maps $\bar{x}_i$ into $\bar{d}$.

The critical lemma for proving property 2 is lemma 2' which is identical to lemma 2 in Section 7 (although the definitions of $\hat{f}'_i$, $\hat{f}'_{i_{(k)}}$, M'_k, and M'_{k_H} 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.