SIAM J. COMPUT.
Vol. 13, No. 2, May 1984

RECURSIVE PROGRAMS AS DEFINITIONS IN FIRST ORDER LOGIC*

ROBERT CARTWRIGHT†

Abstract. Despite the reputed limitations of first order logic, it is easy to state and prove almost all interesting properties of recursive programs within a simple first order theory, by using an approach we call "first order programming logic". Unlike higher order logics based on fixed-point induction, first order programming logic is founded on deductive principles that are familiar to most programmers. Informal structural induction arguments (such as termination proofs for LISP append, McCarthy's 91-function, and Ackermann's function) have direct formalizations within the system.

The essential elements of first order programming logic are:

(1) The data domain D must be a finitely generated set that explicitly includes the "undefined" object $\perp$ (representing nontermination) as well as ordinary data objects.

(2) Recursive programs over D are treated as logical definitions augmenting a first order theory of the data domain.

(3) The interpretation of a recursive program is the least fixed-point of the functional corresponding to the program.

Since the data domain D is a finitely generated set, the first order axiomatization of D includes a structural induction axiom scheme. This axiom scheme serves as the fundamental "proof rule" of first order programming logic.

The major limitation of first order programming logic is that *every* fixed-point of the functional corresponding to a recursive program is an acceptable interpretation for the program. The logic fails to capture the notion of *least* fixed-point. To overcome this limitation, we present a simple, effective procedure for transforming an arbitrary recursive program into an equivalent recursive program that has a unique fixed-point, yet retains the logical structure of the original. Given this transformation technique, it is our experience that first order programming logic is sufficiently powerful to prove almost any property of practical interest about the functions computed by recursive programs.

Key words. programming logic, recursive programs, recursive definitions, rewrite rules, semantics, verification, program transformations

1. Introduction.

It is a widely accepted part of computer science folklore that first order logic is too limited a formalism for stating and proving the interesting properties of recursive programs. Hitchcock and Park [16], 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** augmented by the recursive definition. As a result of this criticism, most researchers developing programming logics for recursive programs have rejected first order logic in favor of more complex higher order systems, e.g., Milner [19], [20], [21], Park [23], deBakker [11], Gordon et al. [15], Scott and deBakker [25], Scott [26], deBakker and deRoever [12]. Nevertheless, we will show that a properly chosen, axiomatizable first order theory is a natural programming logic for recursive programs. In fact, we will present evidence which suggests that first order logic may be a more appropriate formalism for reasoning about specific recursive programs than higher order logics.

* Received by the editors September 1, 1978, and in final revised form July 15, 1983.

† Computer Science Program, Department of Mathematical Sciences, Rice University, Houston, Texas 77251. This research was partially supported by the National Science Foundation under grants MCS76-14293, MCS78-05850, and MCS81-04209.

[1] A brief synopsis of the important definitions from mathematical logic (such as theory) appears in the next section.

2. Logical preliminaries. As a foundation for the remainder of the paper, we briefly summarize the important definitions and notational conventions of first order logic. Readers who are unfamiliar with the fundamental concepts of first order predicate calculus are encouraged to consult Enderton's excellent introductory text [14].

In first order programming logic, recursive definitions are expressed within a conventional *first order logical language L with equality* determined by a countable set of function symbols G, a countable set of predicate symbols R, and an associated "arity" function $\#: G \cup R \to \text{Nat}$ (where Nat denotes the set of natural numbers) specifying the *arity* $\# p$ (required number of arguments) for each function and predicate symbol p. Nullary function symbols serve as constants. The function and predicate symbols are the names of the primitive operations of the data domain. The first order language L determined by G, R, and $\#$ contains two classes of strings: a set of *terms* constructed from variables and function symbols G, and a set of *formulas* constructed from predicate symbols $\{=\} \cup R$ applied to terms (forming *atomic formulas*) and from logical connectives $\{\forall, \wedge, \vee, \neg\}$ applied to simpler formulas. Each function and predicate symbol p is constrained to take exactly $\# p$ arguments.

A context free grammar defining the (context free) syntax of terms and formulas appears below.

$$
\begin{array}{ll}
\langle\text{term}\rangle & \to \langle\text{constant}\rangle \mid \langle\text{variable}\rangle \mid \langle\text{function-symbol}\rangle\ (\langle\text{termlist}\rangle) \\
\langle\text{termlist}\rangle & \to \langle\text{term}\rangle \mid \langle\text{term}\rangle, \langle\text{termlist}\rangle \\
\\
\langle\text{atomic-formula}\rangle & \to \langle\text{predicate-symbol}\rangle\ (\langle\text{termlist}\rangle) \mid \langle\text{term}\rangle = \langle\text{term}\rangle \\
\langle\text{formula}\rangle & \to \langle\text{atomic-formula}\rangle \mid \forall\langle\text{variable}\rangle\langle\text{formula}\rangle \mid \neg\langle\text{formula}\rangle \mid \\
 & \quad (\langle\text{formula}\rangle \wedge \langle\text{formula}\rangle) \mid (\langle\text{formula}\rangle \vee \langle\text{formula}\rangle)
\end{array}
$$

An occurrence of a variable v in a formula α is *bound* if the occurrence is contained within a subformula of the form $\forall v\beta$ or $\exists v\beta$. An occurrence of a variable is *free* iff it is not bound. Terms and formulas containing no occurrences of free variables are called *variable-free terms* and *sentences*, respectively. Let $\alpha(x)$ denote a formula possibly containing the variable x and let t denote an arbitrary term. Then $\alpha(t)$ denotes the formula obtained from $\alpha(x)$ by replacing every free occurrence of x by t.

The additional logical connectives $\{\oplus, \supset, \equiv, \exists, \exists!\}$ are defined as abbreviations for combinations of primitive connectives as follows

$$
\begin{array}{lll}
(\alpha \oplus \beta) & \text{abbreviates} & ((\alpha \wedge \neg\beta) \vee (\neg\alpha \wedge \beta)) \\
(\alpha \supset \beta) & \text{abbreviates} & (\neg\alpha \vee \beta) \\
(\alpha \equiv \beta) & \text{abbreviates} & ((\alpha \supset \beta) \wedge (\beta \supset \alpha)) \\
\exists v\alpha & \text{abbreviates} & \neg\forall v \neg\alpha \\
\exists! v\alpha(v) & \text{abbreviates} & \exists v(\alpha(v) \wedge \forall u(\alpha(u) \supset u = v))
\end{array}
$$

where α and β denote arbitrary formulas and v denotes an arbitrary variable.

A formula with elided parentheses abbreviates the fully parenthesized formula generated by giving unary connectives precedence over binary ones, ranking binary connectives in order of decreasing precedence: $\{\wedge\} > \{\vee, \oplus\} > \{\supset\} > \{\equiv\}$, and associating adjacent applications of connectives of equal precedence to the right. For the sake of clarity, we will occasionally substitute square brackets {[,]} for parentheses within formulas. In place of a sentence, a formula α abbreviates the sentence $\forall \bar{v}\alpha$ where $\bar{v}$ is a list of the free variables of α. Similarly, the forms $\forall x{:}p\,\alpha$ and $t{:}p$, where p is a unary predicate symbol, abbreviate the formulas $\forall x[p(x) \supset \alpha]$ and $p(t)$, respectively.

Let S denote a (possibly empty) set of function and predicate symbols (with associated arities) not in the language L. Then $L \cup S$ denotes the first order language determined by the function and predicate symbols of L augmented by S; $L \cup S$ is called an *expansion* of L.

Although logicians occasionally treat first order logic as a purely syntactic system (the subject of *proof theory*), we are interested in what terms and formulas *mean.* The meaning of a first order language L is formalized as follows. A *structure* **M** *compatible with* L is a triple $\langle |M|, \mathbf{M_G}, \mathbf{M_R} \rangle$ where $|M|$ (called the *universe*) is a set of (data) values; $\mathbf{M_G}$ is a function mapping each function symbol $g \in G$ into a $\#g$-ary function on $|M|$; and $\mathbf{M_R}$ is a function mapping each predicate symbol $r \in R$ into a $\#r$-ary *predicate* on $|M|$—a function mapping $|M|^{\#r}$ into the set Tr of truth values $\{\mathbf{TRUE}, \mathbf{FALSE}\}$. The universe $|M|$ must be disjoint from Tr. Given a structure **M** compatible with L and a *state* s (often called an *interpretation function*) mapping the variables of L into $|M|$, every term in L denotes an object in $|M|$ and every formula denotes a truth value. The meaning of terms and formulas of L is defined by structural induction in the obvious way; for a rigorous definition, consult Enderton's text.

Let H be a subset of the function symbols of the first order language L. A structure **M** compatible with the language L is called an *H-term structure* iff the universe $|M|$ consists of equivalence classes of variable-free terms in L constructed solely from the function symbols in H. A structure compatible with L is *finitely generated* iff there exists a finite subset H of the function symbols of L such that **M** is isomorphic to an H-term structure.

Let **M** be a structure compatible with the language L and let **S** denote a set of functions and predicates over $|M|$ interpreting a set S of function and predicate symbols not in L. Then $\mathbf{M} \cup \mathbf{S}$ denotes the structure consisting of **M** augmented by the functions and predicates **S**; $\mathbf{M} \cup \mathbf{S}$ is called an *expansion* of **M**.

In mathematical logic, it is often important to make a clear distinction between a function symbol and its interpretation. To cope with this issue, we will use the following notation. Function symbols appear in ordinary type and stand for themselves. In contexts involving a single structure **M**, a function or predicate symbol p written in **boldface (p)** denotes $\mathbf{M_G}(p)$ or $\mathbf{M_R}(p)$, the interpretation of p in **M**. In more general contexts, $\mathbf{M}[p]$ denotes the interpretation of the symbol p in the structure **M**. Similarly, $\mathbf{M}[\alpha][s]$ denotes the meaning of the formula or term α in **M** under the state s. If α is a variable-free term or a sentence, then its meaning in a structure is independent of the particular choice of state s. In this case, the abbreviated notation $\mathbf{M}[\alpha]$ denotes the meaning of α in **M**.

Let T be a set of sentences in the first order language L. A *model* of T is a structure **M** compatible with L such that every sentence of T is **TRUE** in **M**. We say that a structure **M** *satisfies* T or alternatively, that T is an *axiomatization* of **M**, iff **M** is a model of T. The set of sentences T forms a *theory* iff it satisfies the following two properties:

(i) *Semantic consistency*: there exists a model of T.

(ii) *Closure under logical implication*: every sentence that is **TRUE** in all models of T is a member of T.

Given a structure **M** compatible with L, the set of sentences in L that are **TRUE** in **M** (denoted *Th* **M**) obviously forms a theory; it is called the *theory of* **M**. Given an arbitrary set A of sentences of L, the *theory generated by* A is the set of sentences that are logically implied by A. A theory T is *axiomatizable* iff there exists a recursive set of sentences $A \subseteq T$ such that A generates T. In this case, the set of sentences A is called an *effective axiomatization* of T.

A theory T typically has an intended model called the *standard model.* Any model that is not isomorphic to the standard model is called a *nonstandard model.* Two structures compatible with the same language L are *elementarily distinct* iff there exists a sentence S in L such that S is true in one structure but not in the other. A theory is *incomplete* iff it has elementarily distinct models; otherwise, it is *complete*. For any structure $\mathbf{M}$, *Th* $\mathbf{M}$ is obviously complete.

Given a recursively enumerable set of axioms A, there is a mechanical procedure for enumerating all of the sentences that are logically implied by A. A *first order deductive system* Γ is a finite set of syntactic rules (often formulated as productions in a phrase structure grammar) that generates a set of sentences from A. A *proof* of a sentence α from A in the deductive system Γ is simply its derivation in Γ from A. A deductive system Γ is *sound* iff every sentence derivable from an axiom set A is logically implied by A. A deductive system Γ is *complete* iff every sentence in the theory generated by A is derivable (provable) in Γ from A. A remarkable property of first order logic is the existence of sound, complete deductive systems for arbitrary axiom sets A. Higher order logics generally do not share this property.

There are many different ways to formulate a sound, complete deductive system for first order logic. Two approaches that are well known to computer scientists are resolution and Gentzen natural deduction [17]. Of course, every first order deductive system that is sound and complete derives exactly the same set of sentences. In this paper, we will leave the choice of deductive system unspecified, since we are not interested in the syntactic details of formal proofs. In our examples, we will present proofs in informal (yet rigorous) terms that readily translate into formal proofs in a Gentzen natural deduction system.

Let $\mathbf{A}$ and $\mathbf{B}$ be two structures compatible with the languages L_A and L_B, respectively, where $L_A \subseteq L_B$ (i.e., L_B is an expansion of L_A). $\mathbf{B}$ is an *extension* of A iff $|B| \supseteq |A|$ and every operation (function or predicate) of $\mathbf{A}$ is the restriction of the corresponding operation of $\mathbf{B}$ to $|A|$. If $|B|$ is identical to $|A|$, then $\mathbf{B}$ is obviously an *expansion* of $\mathbf{A}$. Otherwise, $|B|$ properly contains $|A|$, and $\mathbf{B}$ is called a *proper extension of* $\mathbf{A}$.

Let $S = \{s_1, \cdots, s_n\}$ be a finite set of function and predicate symbols not in the language L_A. A *definition for S over the structure* $\mathbf{A}$ is a collection of sentences Δ in the language $L_A \cup S$ such that $\mathbf{A}$ can be expanded—by adding interpretations for the new function and predicate symbols in S—to a model for Δ. An *unambiguous definition for S over* $\mathbf{A}$ is a definition that determines a *unique* expansion of $\mathbf{A}$.

A formula $\alpha(x_1, \cdots, x_k)$ in L_A *defines the k-ary predicate* $\mathbf{r}$ *in* $\mathbf{A}$ iff α contains no free variables other than $x_1, \cdots, x_k$ and for all states s over $|A|$, $\mathbf{A}[\alpha(x_1, \cdots, x_k)][s] = \mathbf{r}(s(x_1), \cdots, s(x_k))$. Similarly, a formula $\alpha(x_1, \cdots, x_k, y)$ in L_A *defines the k-ary function* $\mathbf{g}$ in $\mathbf{A}$ iff α contains no free variables other than $x_1, \cdots, x_k$ and for all states s over $|A|$, $\mathbf{A}[\alpha(x_1, \cdots, x_k, y)][s] = \mathbf{TRUE}$ iff $\mathbf{g}(s(x_1), \cdots, s(x_k)) = s(y)$. A set $\mathbf{S} = \{\mathbf{s_1}, \cdots, \mathbf{s_n}\}$ of predicates and functions interpreting the symbols S is *definable in* $\mathbf{A}$ iff there exist formulas $\alpha_1, \cdots, \alpha_n$ defining $\mathbf{s_1}, \cdots, \mathbf{s_n}$, respectively.

Let T be a semantically consistent set of sentences in the language L_A and let $\mathbf{A}$ be a model of T. A *definition for S augmenting T* is a collection of sentences Δ in the language $L_A \cup S$ such that every model of T can be expanded—by adding interpretations for the new function and predicate symbols in S—to a model for $T \cup \Delta$. An *unambiguous definition for S augmenting T* is a definition that determines a *unique* expansion in *every* model of T. Note that a definition for S over a model of T is not necessarily a definition augmenting T. Similarly, an unambiguous definition for S over a model of T is not necessarily an unambiguous definition augmenting T.

A set $\mathbf{S} = \{\mathbf{s_1}, \cdots, \mathbf{s_n}\}$ of predicates and functions over $|A|$ interpreting the symbols S is *implicitly definable in the theory generated by T* iff there an unambiguous definition

Δ for S augmenting T such that $\mathbf{S}$ is interpretation of S determined by Δ in the structure $\mathbf{A}$. The set $\mathbf{S}$ is *explicitly definable in* T iff there exists a set of formulas $\alpha_1, \cdots, \alpha_n$ in L_A, defining $\mathbf{s_1}, \cdots, \mathbf{s_n}$, respectively, in $\mathbf{A}$. One of the most important results in the theory of definitions, Beth's Definability Theorem [2], asserts that a set $\mathbf{S}$ of functions and predicates over $|A|$ is implicitly definable in T iff it is explicitly definable in T. Hence, we are justified in dropping the modifiers "implicitly" and "explicitly" when discussing the issue of definability in a theory.

In first order programming logic, we formalize data domains as structures in first order logic. In this context, a recursive program is simply a particular form of logical definition over the data domain. Before proceeding with the development of the formal theory, we will first examine and refute a widely accepted argument asserting that first order logic is incapable of expressing and proving that functions defined in recursive programs are total.

3. Hitchcock and Park's critique of first order logic. As motivation for developing a higher order logic for reasoning about recursive programs, Hitchcock and Park [16] claim that first order logic is too weak to express and prove that the functions defined in a recursive program are total. As justification, they consider the following recursive program over the natural numbers:

$$\text{(1)} \qquad \text{zero}\,(n) = \text{IF } n = 0 \text{ THEN } 0 \text{ ELSE zero}\,(n-1)$$

where IF-THEN-ELSE is interpreted as a logical connective (as in reference [17]). This program (1) can be expressed within the usual language of first order number theory (eliminating the special IF-THEN-ELSE connective) by the sentence:

$$\text{(2)} \qquad \forall n[(n = 0 \supset \text{zero}\,(n) = 0) \wedge (n \neq 0 \supset \text{zero}\,(n) = \text{zero}\,(n-1))].$$

While they concede that it is very easy to prove informally by induction that the zero function is total on the natural numbers they claim that no sentence provable in a first order theory of the natural numbers augmented by (2) can state that zero is total. To justify this claim, they propose the following argument.

Let $\mathbf{N}$ denote the structure consisting of the natural numbers, the constants (0-ary functions) $\{\mathbf{0}, \mathbf{1}\}$, the binary functions $\{+, \times, -\}$ and the binary predicates $\{=, <\}$. By the upward Lowenheim–Skolem theorem, the theory (set of true sentences) of $\mathbf{N}$ has a nonstandard model $\bar{\mathbf{N}}$ that is a proper extension of $\mathbf{N}$. The additional objects in the universe $|\bar{N}|$ are "nonstandard" natural numbers that are greater than all standard integers (the elements of the universe $|N|$). Hitchcock and Park assert that the recursive definition for zero obviously does not terminate for all elements of $|\bar{N}|$, since the nonstandard numbers in this model have infinitely many predecessors. Given this assertion, no sentence θ provable in a first order theory for $\mathbf{N}$ can state zero is total since θ must be true in $\bar{\mathbf{N}}$.

The flaw in Hitchcock and Park's analysis is their assumption that the interpretation of the function symbol zero in a nonstandard model must be obtained by applying standard computation (reduction) rules to (1). In the theory of program schemes [15], where the concept of program execution is embedded in the formalism (just as the meaning of logical connectives such as $\wedge$ and $\vee$ is embedded in first order logic), this point of view makes sense. But in first order logic, there is no notion of execution constraining the interpretation of recursively defined functions. Recursive definitions are simply equations that introduce new function symbols; they do not necessarily have a computational interpretation. In fact, they may have no interpretation at all (see example (4) below). In first order programming logic, we prevent potential inconsistencies by restricting logical theories to a form that guarantees that arbitrary recursive definitions have computational interpretations in the standard model.

We can gain additional insight into the difference between first order logic and the theory of program schemes by examining Hitchcock and Park's example in more detail. Let A be the standard first order Peano axiomatization for the natural numbers including an axiom scheme expressing the induction principle (such an axiomatization appears in Appendix I). Given A and the recursive definition of zero, we can easily prove the sentence

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

by induction on n[2]. Both the base case and induction step are trivial consequences of (2). Consequently, the function zero defined by (2) is identically zero in every model of A—including $\bar{\mathbf{N}}$. Furthermore, since the models of A do not contain an object (usually denoted $\perp$) representing a divergent computation,[3] all of them must, by definition, interpret every function symbol by a (total) function on the universe of the model (the set of standard or nonstandard natural numbers). Hence, no recursion equation augmenting A can define a nontotal function in any model of A. For the same reason, some recursion equations such as

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

define no function at all because they are inconsistent with the original theory.

The situation is more interesting if we start with an axiomatization of the structure $\mathbf{N}^+$ consisting of $\mathbf{N}$ augmented by the undefined object $\perp$, instead of an axiomatization for $\mathbf{N}$. In this case, the interpretation for a function symbol f may be partial in the sense that it maps some elements of the data domain into $\perp$. Note that $\perp$ is an ordinary constant which is forced by the axiomatization to behave like a "divergent" or "undefined" data object. It is *not* a new logical primitive.

Within the first-order language for $\mathbf{N}^+$, we can assert that f is total on $|N|$ by simply stating

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

Let A^+ be an axiomatization (including an induction axiom scheme) for $\mathbf{N}^+$ analogous to Peano's axioms (in first order form) for $\mathbf{N}$. A suitable formulation of A^+ appears in Appendix I. Given A^+ and the recursive definition (2), we can easily establish that the zero function is total on $|N|$ by proving the sentence

$$\forall n[n \neq \perp \supset \text{zero}(n) \neq \perp].$$

The proof (which appears in Appendix II) is a direct translation of the informal structural induction proof that Hitchcock and Park cite in their paper. Consequently, we are forced to conclude that a careful analysis of Hitchcock and Park's example actually supports the thesis that the totality of recursively defined functions can be naturally expressed and proven within first order logic. We will rigorously establish this result in the next section.

4. Basic concepts of first order programming logic. As we suggested in the previous section, the undefined object $\perp$ plays a crucial role in first order programming

[2] Note that adding new function symbols to L implicitly augments A by additional instances of the induction axiom scheme (containing the new symbols).

[3] No object in *any* model of A—including those within nonstandard integers—has the same properties as the divergent object $\perp$. For instance, successor $(\perp)=\perp$, yet in any model of A, $\forall x$ successor $(x)\neq x$.

logic, just as it does in higher order logics such as LCF [15], [19], [20], [21]. If we fail to include the undefined object $\perp$ in the data domain, recursive definitions like

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

on the natural numbers **N** are inconsistent with the axiomatization of the domain; the interpretation of f must be a (total) function on the natural numbers, yet no such function exists.

Consequently, first order programming logic imposes certain constraints on the data domain (and hence on any corresponding theory). In particular, the program data domain must be *continuous*. The following collection of definitions defines this property and several related concepts.

DEFINITION. A *complete partial ordering* $\subseteq$ on a set S is a binary relation over S such that:

(i) $\subseteq$ is a partial ordering on S (a reflexive, antisymmetric, and transitive relation on S).
(ii) The set S contains a least element $\perp$ (under the partial ordering $\subseteq$).
(iii) Every chain (denumerable sequence ordered by $\subseteq$) $x_0 \subseteq x_1 \subseteq x_2 \subseteq \cdots$ has a least upper bound.

A set S with a corresponding complete partial ordering $\subseteq$ is called a *complete partial order* (abbreviated *cpo*); the partial ordering $\subseteq$ is called the *approximation ordering* for S.

DEFINITION. Given cpo's A and B, a function $f: A \rightarrow B$ is *continuous* iff the image of an arbitrary chain $X = x_0 \subseteq x_1 \subseteq x_2 \subseteq \cdots$ in A is a chain in B and the image of the least upper bound of X is the least upper bound of the chain image.

There are two standard methods for building composite cpo's from simpler ones. First, given the cpo's $A_1, \cdots, A_m$ under the approximation orderings $\subseteq_1, \cdots, \subseteq_m$, respectively, the Cartesian product $A_1 \times \cdots \times A_m$ forms a cpo under the ordering $\subseteq$ defined by

$$\bar{x} \subseteq \bar{y} \equiv \bigwedge_{1 \leq i \leq n} [x_i \subseteq_i y_i].$$

Second, given the cpo A under $\subseteq_A$ and the cpo B under $\subseteq_B$, the set of continuous functions mapping A into B forms a cpo under the ordering $\subseteq$ defined by

$$g \subseteq h \equiv \forall \bar{x} \in A[g(\bar{x}) \subseteq_B h(\bar{x})].$$

DEFINITION. A structure **D** including the constant $\perp$ is *continuous* under the binary relation $\subseteq$ on $|D|$ iff $|D|$ forms a complete partial order under $\subseteq$ and every function $\mathbf{f}: |D|^{\#f} \rightarrow |D|$ in **D** is continuous.

DEFINITION. Given a continuous data domain **D** compatible with the language L_D determined by the function symbols G and the predicate symbols R, a *recursive program P* over $|\mathbf{D}|$ has the form:

$$\{f_1(\bar{x}_1) = t_1, f_2(\bar{x}_2) = t_2, \cdots, f_n(\bar{x}_n) = t_n\}$$

where $n > 0$; the set F of function symbols $\{f_1, f_2, \cdots, f_n\}$ is disjoint from $G \cup R$; $\bar{x}_1, \bar{x}_2, \cdots, \bar{x}_n$ are lists of variables; and $t_1, t_2, \cdots, t_n$ are terms in the language $L_D \cup F$ such that each term t_i contains no variables other than those in $\bar{x}_i$. The intended *meaning* of the n-tuple of function symbols $[f_1, \cdots, f_n]$ introduced in the program P is the least fixed-point of the functional

$$\mathbf{P} = \lambda f_1, \cdots, f_n \cdot [\lambda \bar{x}_1 \cdot t_1, \cdots, \lambda \bar{x}_n \cdot t_n]$$

corresponding to P.

By Kleene's recursion theorem (most broadly formulated by Tarski [27]), **P** must have a least fixed-point $[\mathbf{f_1}, \cdots, \mathbf{f_n}]$, because it is a continuous mapping from the cpo$(|D|^{\# f_1} \to |D|) \times \cdots \times (|D|^{\# f_n} \to |D|)$ into itself. A proof that **P** is continuous can be found in either Cadiou [5] or Vuillemin [28].

Although continuity ensures that recursive programs are well-defined, it does not guarantee that they can be implemented on a machine. For this reason, program data domains typically satisfy several additional constraints which we lump together under the label *arithmeticity*. The most important difference between an arithmetic domain and a continuous domain is that the former must be finitely generated. First order programming logic critically depends on this property, because it presumes that the domain obeys the principle of structural induction. The remaining properties that distinguish arithmetic domains from continuous ones (items (i) and (iii) in the definition below) are not essential; they are included solely to simplify the exposition.

DEFINITION. A structure **D** is *flat* iff it is continuous under the binary relation $\sqsubseteq$ defined by the identity

$$a \sqsubseteq b \equiv [a = b \vee a = \bot].$$

DEFINITION. A continuous function $f: A_1 \times \cdots \times A_m \to B$ is *strict* iff $f(x_1, \cdots, x_m) = \bot$ when any argument $x_i = \bot$.

DEFINITION. Let **D** be a data domain (structure) compatible with the language L_D determined by the function symbols G and the predicate symbols R. **D** is an *arithmetic domain* iff it satisfies the following three properties:

(i) **D** is flat.

(ii) **D** is finitely generated. Hence, every element $d \in |D|$ has at least one *name* consisting of a variable-free term α such that $\mathbf{D}[\alpha] = d$. Note that the finite generation property implies that **D** obeys induction on the structure of names (often called "structural induction" or "generator induction"). We can formulate this principle as follows. Let Gen $= \{g_1, \cdots, g_k\}$ denote a minimal subset of G (the function symbols of L) that generates $|D|$. Generator induction asserts that for every unary predicate $\varphi(x)$ over **D**,

$$(*) \qquad \left[\bigwedge_{1 \leq i \leq k} \forall x_1, \cdots, x_{\# g_i}[\varphi(x_1) \wedge \cdots \wedge \varphi(x_{\# g_i}) \supset \varphi(g_i(x_1, \cdots, x_{\# g_i}))]\right] \supset \forall x\, \varphi(x).$$

In the literature on programming languages, the generator symbols Gen are often called constructors. Note that a minimal set of generators Gen for a domain **D** is not necessarily unique.

(iii) The set of functions **G** includes the constants {**true, false**} and the special function **if-then-else** which partitions $|D|$ into three nonempty disjoint subsets D_{true}, D_{false}, $D_\bot$ such that

$$\begin{aligned}
&\mathbf{true} \in D_{\text{true}}\\
&\mathbf{false} \in D_{\text{false}}\\
&\bot \in \mathrm{D}_\bot\\
&\mathbf{if}\ p\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta = \alpha \text{ if } p \in D_{\text{true}}\\
&\mathbf{if}\ p\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta = \beta \text{ if } p \in D_{\text{false}}\\
&\mathbf{if}\ p\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta = \bot \text{ if } p \in D_\bot.
\end{aligned}$$

All functions in **G** other than **if-then-else** must be strict.

With the exception of the induction principle (*) appearing in property (ii), the preceding list of conditions on **D** can be formally expressed by a finite set of sentences in the language L_D. The induction principle (*) cannot be expressed in L_D, because it asserts that induction holds for all unary predicates—an uncountable set with many members that cannot be defined within L_D. We will explore this issue in depth in § 7.

Fortunately, confining our attention to arithmetic data domains does not significantly limit the applicability of first order programming logic. With the exception of domains including an extensional treatment of higher order data objects (such as functions), the data domain of any plausible recursive programming language has a natural formalization as an arithmetic structure. At the end of this section, we will discuss how to extend an arbitrary, finitely generated domain **D** excluding $\perp$ to form an arithmetic domain **D**$'$ with universe $|D| \cup \{\perp\}$.

Before we state and prove the fundamental theorem of first order programming logic, we must resolve a subtle issue concerning the status of induction in arithmetic domains that are augmented by definitions. Formalizing induction in first order logic requires an axiom scheme: a template with a free formula parameter. The scheme represents the infinite recursive set of sentences consisting of all possible instantiations of the template. Let **D** be a structure that is finitely generated by the function symbols $\mathrm{Gen} = \{g_1, \cdots, g_k\}$. Obviously, the corresponding induction principle (*) holds in **D**. In a first order axiomatization A_D for **D**, we typically include the following axiom scheme formalizing the induction principle (*)

$$(**) \quad \left[\bigwedge_{1 \leqq i \leqq k} \forall x_1, \cdots, x_{\#g_i}[\varphi(x_1) \wedge \cdots \wedge \varphi(x_{\#g_i}) \supset \varphi(g_1(x_1, \cdots, x_{\#g_i}))]\right] \supset \forall x\, \varphi(x)$$

where $\varphi(x)$ is an arbitrary formula in L_D defining a unary predicate. The scheme asserts that structural induction holds for every *definable* unary predicate in the domain. Any structure satisfying the structural induction scheme (**) is called an *inductive domain* with generator set Gen. The only difference between an inductive domain and a finitely generated domain is that induction may fail in an inductive domain for predicates that are not definable.

When we augment a finitely generated domain **D** by a definition Δ introducing new function and predicate symbols S, how should we interpret the induction scheme? Does the formula parameter in the scheme range over formulas in the augmented language or formulas in the original language? Assuming that we are interested in constructing the strongest possible theory for the expanded structure, the answer to the question is clear. Since the universe of the expanded structure is identical to the universe of the original domain, the induction scheme must hold for all predicates that are definable in the expanded structure (using the augmented language). Consequently, we follow the convention that a definition Δ over a finitely generated domain **D** implicitly includes all of the new instances of the structural induction scheme (**) for **D** corresponding to the language extension. In this context, $A_D \cup \Delta$ denotes the set of axioms containing A_D, Δ, and *all new instances* of the induction scheme (**). For reasons that will become clear when we discuss nonstandard models, we follow exactly the same convention for definitions over inductive domains: a definition Δ over an inductive domain **D** implicitly includes all new instances of the induction scheme (**) for **D**. To emphasize this convention, we will use the term *arithmetic definition* to refer to any definition that implicitly includes new instances of the corresponding induction scheme.

THEOREM (fundamental theorem). *Let P be a recursive program*

$$\{f_1(\bar{x}_1) = t_1, f_2(\bar{x}_2) = t_2, \cdots, f_n(\bar{x}_n) = t_n\}$$

over an arithmetic domain **D**, *and let F denote the least fixed-point of the functional of P. Then P is an arithmetic definition over* **D** *satisfying the model* **D** $\cup$ **F**.

Proof. By Kleene's recursion theorem [27], the functional for P has a least fixed-point $\mathbf{F} = [\mathbf{f_1}, \cdots, \mathbf{f_n}]$. Hence,

$$\bigwedge_{1 \leq i \leq n} [\mathbf{f_i} = \lambda \bar{x}_i \cdot \mathbf{t_i}],$$

where $\mathbf{t_i}$ denotes the interpretation of t_i given that the primitive function symbols in t_i are interpreted by the corresponding functions in $\mathbf{D}$ and the function symbols $f_1, \cdots, f_n$ are interpreted by $\mathbf{f_1}, \cdots, \mathbf{f_n}$, respectively. This fact can be restated in the form

$$\bigwedge_{1 \leq i \leq n} \mathbf{D}\dagger[f_i(\bar{x}_i)][s] = \mathbf{D}\dagger[t_i][s],$$

where $\mathbf{D}\dagger$ denotes the structure $\mathbf{D} \cup \mathbf{F}$, and s is an arbitrary state over $|D\dagger|$. Since the universe $|D\dagger|$ is identical to the universe $|D|$, the induction principle (*) holds for all unary predicates $\varphi(x)$ over $|D\dagger|$ including those defined by formulas in $L_D \cup \{f_1, \cdots, f_n\}$. Hence, $\mathbf{D}\dagger$ is a model for $P \cup$ (**) extending $\mathbf{D}$. □

This theorem formally establishes that we can interpret recursive programs as definitions in first order logic. Consequently, given a recursive program P over an arithmetic domain $\mathbf{D}$, we can prove properties of P by applying ordinary first order deduction to a suitable first order axiomatization A_D of $\mathbf{D}$ (including the structural induction scheme (**) for $\mathbf{D}$) augmented by the equations in P (which are simply first order formulas). Using this approach, we can prove almost any property of practical importance about P, including totality. Most proofs strongly rely on structural induction.

A minor impediment to the practical application of first order programming logic is the fact that most axiomatizations appearing in the literature (e.g., the first order formulation of Peano's axioms) specify domains that exclude the special object $\perp$. In structures that are not specifically intended to serve as domains for computation, an object representing a divergent computation is superfluous.

Fortunately, it is easy to transform a first order axiomatization A_D for a finitely generated data domain $\mathbf{D}$ that excludes $\perp$ into an axiomatization A'_D for a corresponding arithmetic domain $\mathbf{D}'$ that includes $\perp$. Although the syntactic details of the transformation are beyond the scope of this paper, the main features warrant discussion. The transformation breaks down into three parts.

First, to satisfy the continuity property required by Kleene's theorem, the transformation designates two distinct elements of $|D|$ as the constants {**true, false**}, adds the undefined object $\perp$ to $|D|$, and replaces the computable predicates of $\mathbf{D}$ (those that can appear in program text) of the data domain by corresponding strict boolean functions.

Second, the transformation extends each primitive function $\mathbf{g}$ in $\mathbf{D}$ to its strict analog over $\mathbf{D} \cup \{\perp\}$. Specifically, for each axiom in the original set A_D, the transformation generates a corresponding axiom with restrictive hypotheses that prevent variables from assuming the value $\perp$. The transformation also generates new axioms asserting that each function $\mathbf{g}$ is strict.

Third, to accommodate nontrivial recursive definitions, the transformation adds the standard ternary conditional function **if-then-else** to the collection of primitive functions. While **if-then-else** is not strict (since **if true then** x **else** $\perp = x$), it is continuous. Without **if-then-else**, every recursive function definition (that actually utilizes recursion) diverges for all inputs, because all the other primitive functions are strict.

The new data domain $\mathbf{D}'$ retains the structure of the original one, yet it is clearly an arithmetic domain. Appendix I presents a sample axiomatization for a data domain (the natural numbers) that does not include $\perp$ and transforms it into one for an arithmetic domain that does.

5. A sample proof. As an illustration of the utility of first order programming logic, consider the following simple example. Let flat and flat1 be recursively defined functions over the domain of LISP S-expressions defined by the following equations:

$$\begin{aligned}&\text{flat}\,(x) = \text{flat1}\,(x, \text{NIL})\\ &\text{flat1}\,(x, y) = \text{if atom } x \text{ then cons}\,(x, y) \text{ else flat1}\,(\text{car } x, \text{flat1}\,(\text{cdr } x, y)).\end{aligned}$$

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

$$\begin{aligned}&\text{flat}\,[(A \cdot B)] &&= (A\ B),\\ &\text{flat}\,[(A \cdot (B \cdot A))] &&= (A\ B\ A),\\ &\text{flat}\,[(A)] &&= (A),\\ &\text{flat}\,[((A \cdot C) \cdot B)] &&= (A\ C\ B).\end{aligned}$$

We want to prove that flat1 (x, y) terminates for arbitrary S-expressions x and y (obviously implying flat (x) is total for all S-expressions x). In the theory of S-expressions augmented by $\{\perp\}$, we can formally state and prove this property in the following way, given that Sexpr (x) abbreviates the formula $x \neq \perp$.

THEOREM. $\forall x, y : \text{Sexpr}\,[\text{flat1}\,(x, y) : \text{Sexpr}]$.

Proof. We prove the theorem by structural induction on x.

Basis: x is an atom.

Simplifying flat1 (x, y) yields cons (x, y) which must be an S-expression since x and y are S-expressions.

Induction step: x has the form cons (hd, tl) where hd: Sexpr and tl: Sexpr. Given the hypotheses

(a) $\forall y : \text{Sexpr}\,[\text{flat1}\,(\text{hd}, y) : \text{Sexpr}]$, and
(b) $\forall y : \text{Sexpr}\,[\text{flat1}\,(\text{tl}, y) : \text{Sexpr}]$,

we must show

$$\forall y : \text{Sexpr}\,[\text{flat1}\,(\text{cons}\,(\text{hd}, \text{tl}), y) : \text{Sexpr}].$$

Since hd, tl, and y are S-expressions,

$$\text{flat1}\,(\text{cons}\,(\text{hd}, \text{tl}), y) = \text{flat1}\,(\text{hd}, \text{flat1}\,(\text{tl}, y)).$$

By induction hypothesis (a), flat1 (tl, y) is an S-expression. Given this fact, we immediately deduce by induction hypothesis (b) that flat1 (hd, flat (tl, y)) is an S-expression. □

Some additional examples appear in Appendix II.

6. Computations in first order programming logic. Although we have shown that recursive programs can be interpreted as definitions over an arithmetic data domain, we have not yet established that there is a plausible definition of computation that is consistent with our logical interpretation. Since conventional first order logic does not include any notion of computation (proof is the closest analogue), we must invent one specifically for first order programming logic. Fortunately, there is a simple syntactic definition of the concept based on "term rewriting systems" that makes sense in the context of first order logic. The critical idea is that computation is a uniform (possibly nonterminating) procedure for transforming a variable-free term into its meaning[4] in the standard model using ordinary first order deduction.

[4] More precisely, into a "canonical" term denoting its meaning.

Term rewriting systems for recursive programs have been extensively investigated by Cadiou [4], Vuillemin [28], [29], Rosen [24], Downey and Sethi [13], and O'Donnell [22], but not in the context of first order theories of program data. The following formulation of first order computation is a distillation and adaptation of the work of Cadiou [5] and Vuillemin [28], [29] recast in the terminology of first order logic.

DEFINITION. A structure $\mathbf{D}$ with function set $\mathbf{G}$ is *effective* iff it satisfies the following two conditions:

(i) Every element $d \in |D|$ has a unique canonical name can (d) that is a variable-free term denoting d in the first order language L_D excluding if-then-else. For simplicity, we require that can $(\perp)$ be $\perp$. In addition, the set of canonical names must be a recursive subset of the set of all variable-free terms in L_D.

(ii) The graph of every function $\mathbf{g}$ in $\mathbf{G}$ is recursive, given that we denote objects of the universe $|D|$ by their canonical names.

All data domains in conventional programming languages satisfy these constraints.

DEFINITION. Let P be an arbitrary recursive program over an effective, arithmetic domain $\mathbf{D}$ defining the function symbols $F=\{f_1, \cdots, f_n\}$, and let L_P be the language $L_D \cup F$. A set Φ of *productions* (*or rewrite rules*) *over* L_P is a set of ordered pairs $u \rightarrow v$ where u and v are variable-free terms of L_P. Let Υ denote the set of left-hand sides of Φ: $\{u \mid u \rightarrow v \in \Phi\}$. Φ is *effective* iff the following three conditions hold.

(i) Every left-hand side in Υ corresponds to a unique production in Φ.

(ii) Υ is a recursive subset of L_P.

(iii) Every noncanonical variable-free term t in L_P contains a subterm in Υ.

DEFINITION. Let Φ be an effective set of productions in the language L_P. Given an arbitrary variable-free term t in L_P, the *Φ-reduction of t* is the countable (finite or infinite) sequence of variable-free terms $\tau = t_0, \cdots, t_k, \cdots$ such that $t_0 = t$, each term t_i has a successor if and only if it is noncanonical, and each successor term t_{i+1} is generated from its predecessor t_i by locating the *left-most* subterm that matches a left-hand side of some production α in Φ and replacing it by the right-hand side of α. The *result* of a reduction τ is the *meaning* in $\mathbf{D}$ of the last term t' of τ ($\mathbf{D}[t']$) if τ is finite and $\perp$ otherwise.

Remark. A reduction τ is finite iff the last term in τ is a canonical name.

6.1. Call-by-name computation. The reduction scheme that directly corresponds with the logical meaning of recursive programs (as defined in § 4) is called *call-by-name* computation.

DEFINITION. Let L_P be the first order language corresponding to an effective, arithmetic domain $\mathbf{D}$ augmented by a recursive program P defining the function symbols $F=\{f_1, \cdots, f_n\}$. The *call-by-name production set* Φ_P for P is the set $\Phi_G \cup \Phi_F \cup \Phi_{if}$, where Φ_G, Φ_F, and Φ_{if} are defined as follows.

(i) Φ_G is the set of productions

$$\{g(\bar{c}) \rightarrow v \mid g \in G - \{\text{if-then-else}\};\ \bar{c} \text{ is a } \#g\text{-tuple of canonical names};\ v \text{ is the canonical name for } \mathbf{g}(\bar{\mathbf{c}})\}$$

where $\mathbf{G}$ denotes the set of primitive functions of $\mathbf{D}$. Since $\mathbf{D}$ is an effective domain, Φ_G is a recursive set.

(ii) Φ_{if} is the set of productions

$$\{\text{if } t \text{ then } u \text{ else } v \rightarrow u \mid t \in \text{can}(D_{\text{true}});\ u, v \text{ are variable-free terms in } L_P\} \cup$$
$$\{\text{if } t \text{ then } u \text{ else } v \rightarrow v \mid t \in \text{can}(D_{\text{false}});\ u, v \text{ are variable-free terms in } L_P\} \cup$$
$$\{\text{if } t \text{ then } u \text{ else } v \rightarrow \perp \mid t \in \text{can}(D_{\perp});\ u, v \text{ are variable-free terms in } L_P\}$$

where can (S) stands for $\{\gamma_d \mid \gamma_d$ is the canonical name for an element $d \in S\}$. Φ_{if}

specifies how to reduce conditional expressions where the first argument (the Boolean test) is in canonical form. It is clearly recursive.

(iii) Φ_F is the set of productions

$$\{f_i(\bar{u}) \to t_i(\bar{u}) \mid f_i(\bar{x}) = t_i(\bar{x}) \in P;\ \bar{u} \text{ is a } \#f_i\text{-tuple of variable-free terms in } L_P\}$$

specifying how to expand an arbitrary function application.

LEMMA. *Φ_P is effective.*

Proof. Immediate from the definition of Φ_P and the fact that **D** is effective. The proofs of conditions (i) and (ii) are trivial. Condition (iii) is a routine induction on the structure of terms. □

DEFINITION. The *call-by-name computation* (with respect to the program P) for a variable-free term t in L_P is the Φ_P-reduction of t.

The following theorem establishes that call-by-name computation transforms variable-free terms in L_P into their meanings in the structure $\mathbf{D} \cup \mathbf{F}$.

THEOREM. *Let* **D** *be an effective, arithmetic domain, and let P be an arbitrary recursive program over* **D** *defining function symbols F. For every variable-free term t in L_P, the result of the call-by-name computation for t with respect to P is identical to* $\mathbf{D} \cup \mathbf{F}[t]$ *where* **F** *denotes the least fixed-point of the functional* **P** *for P.*

Proof. See reference [9]. It is a straightforward but tedious induction on an appropriate measure of the complexity of the term t. □

6.2. Call-by-value computation. Up to this point, we have confined our attention to program semantics consistent with call-by-name computation. However, most practical programming languages (e.g., LISP, PASCAL, C) employ *call-by-value* computation which has slightly different semantics. Call-by-value computation is identical to call-by-name computation, except for the productions concerning the expansion of program functions F.

DEFINITION. Let L_P be the first order language corresponding to an effective, arithmetic domain **D** augmented by a recursive program P defining the function symbols $F = \{f_1, \cdots, f_n\}$. The *call-by-value production set* $\Phi_{P_\perp}$ for P is the set $\Phi_G \cup \Phi_{F_\perp} \cup \Phi_{if}$ where Φ_G and Φ_{if} are defined exactly as they are in call-by-name computation, and $\Phi_{F_\perp}$ is defined as the set of productions

$$\{f_i(\bar{c}) \to t_i(\bar{c}) \mid f_i(\bar{x}) = t_i(\bar{x}) \in P;\ \bar{c} \text{ is a } \#f_i\text{-tuple of canonical terms excluding can } (\perp)\}.$$

LEMMA. *$\Phi_{P_\perp}$ is effective.*

Proof. Immediate from the definition of effective production set. □

DEFINITION. The *call-by-value computation* (with respect to program P) for a variable-free term t in L_P is the $\Phi_{P_\perp}$-reduction of t.

The main consequence of this change is that an application of a program function f_i is not expanded until all the arguments are reduced to canonical form. Fortunately, there is a simple semantic relationship between call-by-value and call-by-name computations. In fact, it is trivial to transform a program P into a slightly different program $P_\perp$—called the *strict transform* of P—such that the call-by-name meaning of $P_\perp$ is identical to the call-by-value meaning of P.

DEFINITION. Let P be an arbitrary recursive program

$$\{f_1(\bar{x}_1) = t_1, \cdots, f_n(\bar{x}_n) = t_n\}$$

over an arithmetic domain **D**. The *strict transform* $P_\perp$ corresponding to P is the program

$$\{f_1(\bar{x}_1) = \text{if } \delta(\bar{x}_1) \text{ then } t_1 \text{ else } \perp, \cdots, f_n(\bar{x}_n) = \text{if } \delta(\bar{x}_n) \text{ then } t_n \text{ else } \perp\}$$

where δ is the primitive "is-defined" function[5] such that:

$$\delta(\bar{x}_i) = \begin{cases} \textbf{true} & \text{if } \bot \notin \bar{x}_i, \\ \bot & \text{otherwise.} \end{cases}$$

Given this transformation, the *call-by-value functional* $\mathbf{P}_\bot$ for P is simply the (call-by-name) functional for $P_\bot$:

$$\lambda f_1, \cdots, f_n \cdot [\lambda \bar{x}_1 \cdot \text{if } \delta(\bar{x}_1) \text{ then } t_1 \text{ else } \bot, \cdots, \lambda \bar{x}_n \cdot \text{if } \delta(\bar{x}_n) \text{ then } t_n \text{ else } \bot].$$

The following theorem establishes that call-by-value computation transforms variable-free terms t in L_P into meanings in the expansion of $\mathbf{D}$ determined by the functional $\mathbf{P}_\bot$.

THEOREM. *Let* $\mathbf{D}$ *be an effective, arithmetic domain, and let P be an arbitrary recursive program over* $\mathbf{D}$ *defining the function symbols F. For every variable-free term t in L_P, the result of the call-by-value computation for t is identical to* $\mathbf{D} \cup \mathbf{F}[t]$ *where* $\mathbf{F}$ *denotes the least fixed-point of the call-by-value functional* $\mathbf{P}_\bot$ *for P.*

Proof. A proof of this theorem, an induction on the complexity of t, appears in [9]; proofs of similar theorems appear in [4] and [26]. □

To avoid confusion between the call-by-name and call-by-value interpretations for recursive programs, we will use the following terminology. Unless we specifically use the qualifier "call-by-value", the intended meaning of a program P defining the function symbols F is the least fixed-point of the (call-by-name) functional $\mathbf{P}$ for P—the call-by-name interpretation for F. In contrast, the intended meaning of a *call-by-value* program P is the least fixed-point of the *call-by-value* functional $\mathbf{P}_\bot$ for P—the call-by-value interpretation for F.

6.3. Proving properties of call-by-value programs. Since it is trivial to translate call-by-value recursive programs into equivalent (call-by-name) recursive programs, first order programming logic obviously accommodates call-by-value semantics. Given a first order axiomatization A_D for the domain $\mathbf{D}$ and a call-by-value recursive program P over $\mathbf{D}$, we augment A_D by the recursion equations $P_\bot$ and the definition of the function δ

$$[x \neq \bot \supset \delta(x) = \text{true}] \wedge \delta(\bot) = \bot.$$

A logically equivalent but conceptually simpler approach is to directly augment A_D by a set of axioms P_{ax} characterizing P; it eliminates the function δ and the construction of the strict transform $P_\bot$. In the direct approach, each function definition

$$f_i(\bar{x}_i) = t_i$$

in the original program P, generates two axioms

$$x_i \neq \bot \wedge \cdots \wedge x_{\#f_i} \neq \bot \supset f_i(\bar{x}_i) = t_i,$$
$$x_i = \bot \vee \cdots \vee x_{\#f_i} = \bot \supset f_i(\bar{x}_i) = \bot$$

defining f in P_{ax}. Note that P_{ax} and $P_\bot$ are logically equivalent sets of formulas. A proof that the expanded domain $\mathbf{D} \cup \mathbf{F}$, where $\mathbf{F}$ denotes the least fixed-point of the call-by-value functional $\mathbf{P}_\bot$, is a model for the augmented axiomatization $A_D \cup P_{\text{ax}}$ appears in reference [7].

Call-by-value programs are an attractive alternative to call-by-name programs because they are easier to implement and programmers seem more comfortable with

[5] Technically, $\boldsymbol{\delta}$ is a countable family of functions $\boldsymbol{\delta}_m$, $m = 1, 2, \cdots$ where $\boldsymbol{\delta}_m$ is the m-ary version of the "is-defined" function. It should be obvious from context which instance of $\boldsymbol{\delta}$ is required.

their semantics. In addition, we will show in § 8 and Appendices III and IV that the complete recursive program corresponding to an arbitrary call-by-value program is easier to describe and understand than the equivalent construction for a call-by-name program.

7. Metamathematics of first order programming logic. Although the fundamental theorem of first order programming logic clearly establishes that recursive programs are arithmetic definitions over an arithmetic domain **D**, it ignores two important issues. First, can recursive programs be ambiguous (as definitions over the domain **D**)? Second, do recursive programs have a plausible interpretation in nonstandard models (of a first order theory for **D**)?

The answer to the first question is significant. In fact, it motivates one of the major technical results of this paper: the complete recursive program construction. Assume that we are given a recursive program P over the domain **D** defining the function symbols $F=\{f_1, \cdots, f_n\}$. If we augment **D** by any fixed-point **F** of the functional **P** for P, the expanded structure $\mathbf{D} \cup \mathbf{F}$ is a model for $P \cup (^{**})$. Interpreting P as an arithmetic definition for **F** over **D** captures the fact that **F** is a fixed-point of **P**, but not the fact that it is the *least* fixed-point.

What are the implications of this form of incompleteness? If every function in the least fixed-point of the functional for P is total, the problem does not arise because the least fixed-point is the only fixed-point. On the other hand, if some function in the least fixed-point is partial, there may or may not be additional fixed-points. In the former case, we cannot prove any property of the least fixed-point that does not hold for all fixed-points. For example, we cannot prove anything interesting about the function **f** defined by

$$(5) \qquad f(x)=f(x)$$

since any interpretation for f over the domain satisfies (5), not just the everywhere undefined function.

In contrast, the program

$$(6) \qquad f(x)=f(x)+1,$$

which determines exactly the same function **f**, is unambiguous. Consequently, given program (6), we can easily prove that

$$\forall x f(x)=\bot$$

in first order programming logic.

There are several possible solutions to this problem. John McCarthy [18] has suggested adding a "minimization" axiom scheme φ_P (containing a free function parameter for each function symbol) to the definition of a program P. The scheme φ_P asserts that **F** approximates every definable set of functions $\mathbf{F}'$ satisfying the equations P ($P_\bot$ if P is a call-by-value program). In this paper, we will develop a more direct approach to the problem: a method for mechanically translating an arbitrary recursive program into an equivalent recursive program with a unique fixed-point.

DEFINITION. A (call-by-name or call-by-value) recursive program P over the domain **D** is *complete* iff the corresponding functional has a unique fixed-point.

In the next section of this paper, we will prove that every recursive program can be effectively transformed into an equivalent complete recursive program. As a result, we can reason about recursive programs that define partial functions by first transforming the programs into equivalent complete programs. Fortunately, the transformation process leaves the logical structure of the original program intact—so that the programmer can understand it.

The answer to the second metamathematical question raised at the beginning of this section is even more interesting than the first one, although its practical significance is less apparent. The behavior of recursive programs in nonstandard structures does not concern programmers interested in proving properties of recursive programs over the standard arithmetic domains supported by language processors. Regardless of the meaning of recursive programs in nonstandard models, first order programming logic provides a sound, yet intuitively appealing formal system for deducing the properties of recursive programs. On the other hand, as computer scientists interested in the deductive and expressive power of various logics, we can gain insight into the relative strength of first order programming logic by settling the question of how to interpret recursive programs over nonstandard structures.

Before we can make precise statements about nonstandard models, we need to introduce some additional terminology.

Definition. Assume that we are given an arithmetic domain **D** generated by the finite set of function symbols Gen. A set T sentences in the language L_D compatible with **D** is an *arithmetically complete* axiomatization of **D** iff

(i) **D** is a model of T.

(ii) T logically implies all the sentences expressing the arithmetic properties of **D** (listed in the definition of arithmetic domain in § 4) except the structural induction principle (*) (which cannot be expressed within a first order theory).

(iii) T logically implies that the structural induction axiom scheme (**) for **D** (stated in § 4), holds for every formula $\varphi(x)$ in L_D.

(iv) For every pair of variable-free terms u and v in L_D, either the sentence $u = v$ or the sentence $u \neq v$ is derivable from T.[6]

Remark. The axiomatization of $\mathbf{N}^+$ in Appendix I is arithmetically complete. Given an arithmetic domain **D**, it is a straightforward but tedious (and error-prone) exercise to devise an effective, arithmetically complete axiomatization for **D**. Note that *Th* **D** is an arithmetically complete axiomatization for **D**. Unfortunately, Gödel's incompleteness theorem implies that for nontrivial domains **D**, *Th* **D** is not recursively enumerable.

An arithmetically complete axiomatization T for an arithmetic domain **D** has many distinct (nonisomorphic) models. The nonstandard models (models other than **D**) are not necessarily arithmetic, since induction may fail for unary predicates that are not definable. On the other hand, they are inductive, because they satisfy the structural induction scheme (**) for **D**.

Definition. A structure $\mathbf{D}'$ is *weakly arithmetic* iff it is a model of an arithmetically complete set of sentences T.

The only difference between an arithmetic and a weakly arithmetic model of T is that induction may fail in a weakly arithmetic model for predicates that are not definable in T. Obviously, the nonstandard models corresponding to an arithmetic domain **D** are weakly arithmetic. Given a recursive program P defining the function symbols F over the arithmetic domain **D**, we can interpret P as a definition over an arbitrary nonstandard model $\mathbf{D}'$ if we can find an interpretation $\mathbf{F}'$ for F such that $\mathbf{D}' \cup \mathbf{F}'$ is a model for $P \cup (**)$.

[6] Although none of the theorems that we prove in this paper depend on this property, it ensures that an arithmetically complete axiomatization has a unique (up to isomorphism) arithmetic model. In addition, it guarantees that arithmetically complete axiomatizations for nontrivial arithmetic domains support elementary syntax (see below).

At first glance, the proof of the fundamental theorem appears to generalize without modification to nonstandard models, because it does not explicitly rely on the fact that the domain **D** is arithmetic rather than weakly arithmetic. Let P be a recursive program over an arithmetic domain **D** defining the function symbols F, let $\mathbf{D}'$ be a nonstandard model corresponding to **D**, and let $\mathbf{P}'$ denote the functional for P over $\mathbf{D}'$. Then the least fixed-point $\mathbf{F}'$ of $\mathbf{P}'$ obviously satisfies the equations P, implying that $\mathbf{D}' \cup \mathbf{F}'$ is a model for P. Hence, the only remaining step in confirming that the proof generalizes to nonstandard models is to show that the induction scheme (**) holds for definable predicates in the extended language $L_D \cup F$—a property that appears plausible, if not obvious.

Nevertheless, there is a simple counterexample to this conjecture. Consider the following program defined over a nonstandard model $\bar{\mathbf{N}}$[7] of the natural numbers augmented by $\perp$ (axiomatized as in Appendix I):

(7) $\quad \text{zero}(n) = \text{if } n \text{ equal } 0 \text{ then } 0 \text{ else zero}(n-1).$

This program is essentially identical to the one that Hitchcock and Park [16] used to argue that first order logic was incapable of expressing and proving that the function defined by (7) is total. The least fixed-point of the corresponding functional is the function **zero** defined by:

$$\mathbf{zero}(x) = \begin{cases} \mathbf{0} & \text{if } x \text{ is a standard natural number,} \\ \perp & \text{otherwise } (x \text{ is } \perp \text{ or nonstandard}). \end{cases}$$

Yet, we have already established the fact in refuting Hitchcock and Park's argument (§ 3 and Appendix II) that we can prove (using structural induction) that the zero function defined in equation (7) is identically zero everywhere except at $\perp$.

Clearly, our naive approach to generalizing the proof of the fundamental theorem to nonstandard models will not work. Our assumption that a recursive program P over a weakly arithmetic structure $\mathbf{D}'$ can be interpreted as a definition introducing a set of functions $\mathbf{F}'$ that

(i) forms the least fixed-point of the functional for P, and
(ii) obeys the structural induction principle (**)

leads to a contradiction. Where did we go wrong?

Ironically, we made essentially the same mistake as Hitchcock and Park: we assumed that a recursive program over a nonstandard structure should be interpreted as the least fixed-point of the corresponding functional. In the preceding example, the least fixed-point of the functional for equation (7) over $\bar{\mathbf{N}}$ is not definable in $\bar{\mathbf{N}}$. For this reason, it need not obey the structural induction principle.

In order to generalize the fundamental theorem to nonstandard models, we must develop a more sophisticated interpretation for recursive programs than the least fixed-point of the corresponding functional. Since first order programming logic formalizes recursive programs as arithmetic definitions over the data domain, we must find an interpretation for recursive programs over nonstandard models that satisfies the structural induction principle. From the preceding example, it is obvious that the least fixed-point interpretation does not. What is a reasonable alternative? For induction to hold, the interpretation must be definable in the original domain **D**. Hence, we must limit our attention to definable fixed-points of the functional for a recursive program—abandoning our reliance on the familiar least fixed-point construction from Kleene's recursion theorem. In its place, we must develop a new approach to constructing fixed-points of functionals that always determines definable functions.

[7] A model containing an element with infinitely many predecessors.

A detailed, systematic development of the subject of definable fixed-points is beyond the scope of this paper (the interested reader is encouraged to consult reference [10]). However, the correct formulation of the generalized fundamental theorem rests on a single lemma which is easy to explain and to justify. The critical lemma is a generalization of Kleene's recursion theorem; it asserts that every continuous functional over an appropriate domain **D** has a least definable-fixed-point. The lemma applies to weakly arithmetic domains that support what John McCarthy calls *elementary syntax.* Fortunately, nonstandard models of nontrivial, arithmetically complete theories possess this property.

Elementary syntax is a definition that introduces functions for encoding finite sequences over the universe as individual elements of the universe. We formalize the notion as follows.

DEFINITION. An arithmetically complete axiomatization T *supports elementary syntax* iff there exists an unambiguous definition Elem augmenting T introducing a set of functions and predicates **Seq** including the constant **0**; unary functions **last**, **mkseq**, **length**, and **suc**; the binary function $\circ$ (append); and unary predicates **seq** and **nat** satisfying the following sentences:

(a) $\mathrm{suc}(\perp)=\perp$

(b) $\forall x[x:\mathrm{nat}\,(x=0 \oplus \exists! y[x=\mathrm{suc}(y) \wedge y:\mathrm{nat}])]$

(c) $\varphi(0) \wedge \forall x:\mathrm{nat}\,[\varphi(x) \supset \varphi(\mathrm{suc}(x))] \supset \forall x:\mathrm{nat}\; \varphi(x)$
for every formula $\varphi(x)$ in $L_D \cup \mathrm{Seq}$

(d) $\mathrm{mkseq}(\perp)=\perp$

(e) $\forall x[x:\mathrm{seq} \equiv \exists! d(d \neq \perp \wedge [\mathrm{mkseq}(d)=x \oplus \exists! y:\mathrm{seq}\; x=\mathrm{mkseq}(d)\circ y])]$

(f) $\forall x, y, z:\mathrm{seq}\,[x\circ(y\circ z)=(x\circ y)\circ z]$

(g) $\forall x, y[x=\perp \vee y=\perp \supset x\circ y=\perp]$

(h) $\forall d[\mathrm{last}(\mathrm{mkseq}(d))=d]$

(i) $\forall y:\mathrm{seq}\,\forall d[d\neq\perp \supset \mathrm{last}(\mathrm{mkseq}(d)\circ y)=\mathrm{last}(y)]$

(j) $\forall d[d\neq\perp \supset \mathrm{length}(\mathrm{mkseq}(d))=\mathrm{suc}(0)]$

(k) $\forall y:\mathrm{seq}\,\forall d[d\neq\perp \supset \mathrm{length}(\mathrm{mkseq}(d)\circ y)=\mathrm{suc}(\mathrm{length}(y))]$

(l) $\forall y[\varphi(\mathrm{mkseq}(y))] \wedge \forall x:\mathrm{seq}\,[\varphi(x) \supset \forall y\; \varphi(\mathrm{mkseq}(y)\circ x)] \supset \forall x:\mathrm{seq}\; \varphi(x)$
for every formula $\varphi(x)$ in $L_D \cup \mathrm{Seq}$.

Remark. Elem implicitly determines a representation function embedding the finite, nonempty sequences over $|D|$ in $|D|$. In the sequel, we will denote the element of $|D|$ representing the sequence $[s_1, \cdots, s_k]$ by $\langle s_1, \cdots, s_k\rangle$.

DEFINITION. A weakly arithmetic domain **D**$'$ *supports elementary syntax* iff there exists a corresponding arithmetically complete axiomatization A_D that supports elementary syntax.

Given this terminology, we can succinctly state the lemma as follows.

LEMMA (generalized recursion theorem). *For every recursive program P over a weakly arithmetic domain* **D**$'$ *supporting elementary syntax, the corresponding functional* **P**$'$ *has a least definable-fixed-point.*

Proof. A detailed proof of this lemma appears [10]; we will only sketch the main ideas. Let **D** and T be the arithmetic domain and the arithmetically complete axiomatization, respectively, corresponding to **D**$'$. By Kleene's least fixed-point theorem for continuous functionals, the functional **P** for P in **D** has a least fixed-point $[\mathbf{f}_1, \cdots, \mathbf{f}_n]$. It is a straightforward, but tedious exercise to construct formulas (called *program formulas*) $\psi_1(\bar{x}_1, y), \cdots, \psi_n(\bar{x}_n, y)$ in $L_D \cup \mathrm{Seq}$ such that $\mathbf{D}[\psi_i(\bar{x}_1, y)][s]$ is **TRUE** iff $\mathbf{f}_i(s(\bar{x}_i))=s(y)$. The key idea underlying the construction of the program formulas is that for each pair $(\bar{a}, b)$ in the graph of a program defined function $\mathbf{f}_i$, there exists a set of finite graphs $G_1 \subset \mathrm{Graph}(\mathbf{f}_1), \cdots, G_n \subset \mathrm{Graph}(\mathbf{f}_n)$, such that:

(i) $(\bar{a}, b) \in G_i$.

(ii) The collection of graphs $G_1, \cdots, G_n$ is *computationally closed under P*: for each pair $(\bar{u}, v)$ in a graph G_j, every reduction of a function application $f_k(\bar{t})$ in the (call-by-name) computation reducing $f_j(\bar{u})$ to v (according to the definition presented in § 6) has the property that $(\mathbf{\bar{t}}, \mathbf{f_k}(\mathbf{\bar{t}}))$ appears in G_k.

Since a finite graph $\{(\bar{a}_1, b_1), \cdots, (\bar{a}_m, b_m)\}$ can be represented by the sequence $\langle\langle\bar{a}_1\rangle, b_1, \cdots, \langle\bar{a}_m\rangle, b_m\rangle$, the formula $\psi_i(\bar{x}_i, y)$ can be expressed in the form

$$\exists g_1, \cdots, g_n(\mathrm{Clos}_P(g_1, \cdots, g_n) \wedge \mathrm{member}(\langle x_i\rangle, y, g_i))$$

where each g_j is a sequence $\langle\langle\bar{a}\rangle, b_1, \cdots, \langle\bar{a}_m\rangle, b_m\rangle$, $\mathrm{Clos}_P(g_1, \cdots, g_n)$ is a formula expressing the fact that $g_1, \cdots, g_n$ represent a set of finite graphs that are computationally closed under P, and member $(\langle\bar{x}\rangle, y, g)$ is a formula expressing the fact that $\langle\langle\bar{x}\rangle, y\rangle$ is an element of the graph represented by the sequence g. Roughly speaking, the formula Clos_P is generated from the text of P by replacing all references to program function symbols f_j by corresponding references to finite graphs g_j.

In the standard model $\mathbf{D}$, the formulas $\psi_1(\bar{x}_1, y), \cdots, \psi_n(\bar{x}_n, y)$ characterize the least fixed-point of $\mathbf{P}$. What do they mean in the weakly arithmetic model $\mathbf{D}'$? From T, we can prove that the n-tuple of functions $[\mathbf{f}'_1, \cdots, \mathbf{f}'_n]$ determined by $\psi_1(\bar{x}_n, y), \cdots, \psi_n(\bar{x}_n, y)$ satisfies the definition P. Moreover, given another collection of formulas $\varphi_1(\bar{x}_1, y), \cdots, \varphi_n(\bar{x}_n, y)$ determining an n-tuple of functions $[\tilde{\mathbf{f}}'_1, \cdots, \tilde{\mathbf{f}}'_n]$ over $|D'|$ that satisfies the definition P, we can prove that $[\mathbf{f}'_1, \cdots, \mathbf{f}'_n]$ approximates $[\tilde{\mathbf{f}}'_1, \cdots, \tilde{\mathbf{f}}'_n]$. Hence, the n-tuple of functions determined by $\psi_1(\bar{x}_1, y), \cdots, \psi_n(\bar{x}_n, y)$ must be the least definable-fixed-point of $\mathbf{P}'$. □

Given the generalized recursion theorem, the following generalization of the fundamental theorem is a simple consequence.

THEOREM (generalized fundamental theorem). *Let P be a recursive program*

$$\{f_1(\bar{x}_1) = t_1, f_2(\bar{x}_2) = t_2, \cdots, f_n(\bar{x}_n) = t_n\}$$

over a weakly arithmetic domain $\mathbf{D}'$ *supporting elementary syntax, and let* $\mathbf{F}'$ *denote the least definable fixed-point of the functional for P. Then P is an arithmetic definition over* $\mathbf{D}'$ *satisfying the model* $\mathbf{D}' \cup \mathbf{F}'$.

Proof. By the generalized recursion theorem, the functional $\mathbf{P}'$ over $\mathbf{D}'$ for P has a least definable fixed-point $[\mathbf{f}'_1, \cdots, \mathbf{f}'_n]$. Hence, the relationship

$$\bigwedge_{1 \le i \le n} \mathbf{f_1} = \lambda\bar{x}_i \cdot \mathbf{t_i}$$

holds where $\mathbf{t_i}$ denotes the interpretation of t_i given that the primitive function symbols in t_i are interpreted by the corresponding functions in $\mathbf{D}'$ and the function symbols $f_1, \cdots, f_n$ are interpreted by $\mathbf{f}'_1, \cdots, \mathbf{f}'_n$, respectively. We can restate this fact as follows

$$\bigwedge_{1 \le i \le n} \mathbf{D}'\dagger[f_i(\bar{x}_i)][s] = \mathbf{D}'\dagger[t_i][s]$$

where $\mathbf{D}'\dagger$ denotes the structure $\mathbf{D}' \cup \mathbf{F}'$ and s is an arbitrary state over $|D'\dagger|$. By the same construction used in the proof of the generalized recursion theorem, every formula over $\mathbf{D}'\dagger$ can be translated into an equivalent formula over $\mathbf{D}'$. Consequently, the induction principle (**) holds for all formulas over $\mathbf{D}'$, implying $\mathbf{D}'\dagger$ is a model for $P \cup (**)$ extending $\mathbf{D}'$. □

8. Construction of complete recursive programs. In this section, we will show how to construct a complete recursive program P^* equivalent to a given call-by-value program P. We will also verify that the constructed program P^* actually is complete and equivalent to P. We relegate the analogous construction and proof for call-by-name programs to Appendix III, since they are similar but somewhat more complex.

The intuitive idea underlying the construction is to define for each function f in the original call-by-value program a corresponding function f^* such that $f^*(x_1, \cdots, x_n)$ constructs the computation sequence for the call-by-value evaluation of $f(x_1, \cdots, x_n)$. In fact, constructing the actual computation sequence really is not necessary; the values of the elements in the sequence, except for the final one (the value of $f(x_1, \cdots, x_n)$), are irrelevant. It is the expanding structure of the sequence that is significant, because it prevents an arbitrary fixed-point solution from filling in points where the computed (least) fixed-point diverges.

For example, consider the trivial program

(8) $f(x) = \text{if } x \text{ equal } 0 \text{ then } 0 \text{ else } f(g(x))$

over the domain of LISP S-expressions where $\mathbf{g}$ is any unary function with fixed-points (i.e., $\mathbf{g}(y) = y$ for some S-expression y). The corresponding functional obviously has multiple fixed-points, although the intended meaning of the definition is the least fixed-point $\mathbf{f}$. If we define f^* by the program

(9) $f^*(x) = \text{if } x \text{ equal } 0 \text{ then cons}(0, \text{NIL}) \text{ else cons}(g(x), f^*(g(x)))$,

then f^* constructs a sequence containing the argument for each call on f in the call-by-value evaluation of $f(x)$, assuming that $f(x)$ terminates. If $\mathbf{f}(x)$ does not terminate (e.g., $\mathbf{g}(x) = x$), then every fixed-point $\mathbf{f}^*$ of the functional for definition (9) must be undefined ($\perp$) at x. Otherwise, $\mathbf{f}^*(x)$ has length greater than any integer which contradicts the fact that every sequence in the data domain is finite.[8] Given (9), we can redefine f by the recursion equation

(10) $f(x) = \text{last}(f^*(x))$

where **last** is the standard LISP function that extracts the final element in a list. Now, by substituting the definition consisting of equations (9) and (10) for the original program (8), we can force f to mean $\mathbf{f}$. We generalize this idea to arbitrary recursive programs as follows.

DEFINITION. Let P be a call-by-value recursive program defining the function symbols F over the arithmetic domain $\mathbf{D}$ supporting elementary syntax. Let L_P and L_{D^*} denote the first order languages $L_D \cup F$ and $L_D \cup \text{Seq}$, respectively, and let $\mathbf{D}^*$ denote domain $\mathbf{D} \cup \mathbf{Seq}$. Let t be an arbitrary term in the language L_P. The *call-by-value computation sequence term* t^* (in the extended language $L_{P^*} = L_{D^*} \cup \{f_1^*, \cdots, f_n^*\}$) corresponding to t is inductively defined as follows:

(i) If t is a constant or a variable x,

$$t^* = \text{mkseq}(x).$$

(ii) If t has the form $g(u_1, \cdots, u_{\#g})$ where $g \in G - \{\text{if-then-else}\}$,

$$t^* = u_1^* \circ \cdots \circ u_{\#g}^* \circ \text{mkseq}(g(\text{last}(u_1^*), \cdots, \text{last}(u_{\#g}^*))).$$

(iii) If t has the form $f_i(u_1, \cdots, u_{\#f_i})$,

$$t^* = u_1^* \circ \cdots \circ u_{\#f_i}^* \circ f_1^*(\text{last}(u_1^*), \cdots, \text{last}(u_{\#f_i}^*)).$$

[8] Although this argument is cast in terms of standard S-expressions, it generalizes to arbitrary models of a simple first order theory of S-expressions. Given the usual recursive definition for **length**, the sentence $\forall y: \text{list } \exists n: \text{integer}[\text{length}(y) < n]$ is a theorem of the theory. Hence it must hold for arbitrary models—including those with infinite objects.

(iv) 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 *call-by-value complete recursive program* P^* equivalent to P is the call-by-value program

$$\{f_1^*(\bar{x}_1) = t_1^*, \cdots, f_n^*(\bar{x}_n) = t_n^*\}$$

over $\mathbf{D}^*$.

A similar construction generates the complete recursive program equivalent to an arbitrary call-by-name recursive program; it appears in Appendix IV. The following theorem, called the *fixed-point normalization theorem*, shows that the complete recursive program construction preserves the meaning of the original program and produces a program that is in fact complete.

THEOREM (fixed point normalization theorem). *Let P be a call-by-value recursive program over an arithmetic data domain* $\mathbf{D}$ *supporting elementary syntax and let* $[\mathbf{f}_1, \cdots, \mathbf{f}_n]$ *denote the least fixed-point of the call-by-value functional* $\mathbf{P}_\perp$ *for P. The complete recursive program* P^* *equivalent to P has the following properties*:

(i) P^* *is complete, i.e. the corresponding call-by-value functional* $\mathbf{P}_\perp^*$ *has a unique fixed-point* $[\mathbf{f}_1^*, \cdots, \mathbf{f}_n^*]$.

(ii) *For* $i = 1, \cdots, n$, $\mathbf{last}(\mathbf{f}_i^*(\bar{d})) = \mathbf{f}_i(\bar{d}))$ *for all* $\# f_i$*-tuples* $\bar{d}$ *over* $|D|$.

Proof. See Appendix III. □

The corresponding theorem for call-by-name programs and a sketch of its proof appear in Appendix IV.

The fixed-point normalization theorem has an important corollary relating complete recursive programs to first order theories. In informal terms, the corollary asserts that a complete recursive program over an arithmetic domain $\mathbf{D}$ is an unambiguous, arithmetic definition augmenting a suitable first order theory for $\mathbf{D}$ (a theory closely resembling Peano arithmetic[9]).

COROLLARY. *Let* A_D *be an arithmetically complete first order axiomatization for the arithmetic domain* $\mathbf{D}$. *Then for every call-by-value program P over* $\mathbf{D}$, *the equivalent complete recursive program* P^* (*expressed in the form* P^*_{ax}) *is an unambiguous, arithmetic definition augmenting* $A_D \cup$ Elem.

Proof. The key idea underlying the proof of the corollary is to generalize the fixed-point normalization theorem to cover programs defined over weakly arithmetic (not just arithmetic) domains. Since all the models of $A_D \cup$ Elem are weakly arithmetic, the generalized normalization theorem implies that P^*_{ax} determines a unique expansion of every model of $A_D \cup$ Elem—immediately establishing the corollary. □

The only obstacle to extending the fixed-point normalization theorem to weakly arithmetic domains is the same complication that we encountered in generalizing the fundamental theorem of first order programming logic in § 7: the least fixed-point of the functional corresponding to a recursive program may not be definable. We overcame this problem in § 7 by substituting the notion of least definable fixed-point for the standard notion of least fixed-point. The same strategy works here.

THEOREM (generalized fixed point normalization theorem). *Let P be a call-by-value recursive program over a weakly arithmetic data domain* $\mathbf{D}$ *supporting elementary syntax and let* $[\mathbf{f}_1, \cdots, \mathbf{f}_n]$ *denote the least definable fixed-point of the call-by-value functional* $\mathbf{P}_\perp$ *for P. The call-by-value complete recursive program* P^* *equivalent to P has the following properties*:

[9] The first order theory generated by Peano's axioms for the natural numbers.

(i) *P^* is complete, i.e. the corresponding call-by-value functional $\mathbf{P}_\perp^*$ has a unique definable-fixed-point $[\mathbf{f}_1^*, \cdots, \mathbf{f}_n^*]$.*

(ii) *For $i = 1, \cdots, n$, $\mathbf{last}\,(\mathbf{f}_i^*(\bar{d})) = \mathbf{f}_i(\bar{d})$ for all $\# f_i$-tuples $\bar{d}$ over $|D|$.*

Proof. The proof of the generalized fixed-point normalization theorem is essentially identical to the proof of the original one, except that it must invoke the generalized recursion theorem described in § 7 instead of Kleene's recursion theorem—substituting the notion of least definable-fixed-point for least fixed-point. □

We will use the term *extended first order programming logic* for P to refer to conventional first order programming logic for P augmented by Elem (the definition of functions **Seq**), the axioms defining the equivalent complete program P^*, and the axioms asserting that each function $\mathbf{f}_i$ is identical to $\mathbf{last} \circ \mathbf{f}_i^*$.

9. Simplifying complete recursive programs. If we carefully examine the proof of the fixed-point normalization theorem, it is clear that we can simplify the structure of the constructed program without affecting the proof of the theorem. It is easy to verify that the same proof works if we substitute the following definition of computation sequence term for the original one.

DEFINITION. Let t be arbitrary term in the language L_D. The *simplified computation sequence term* t^* corresponding to t is given by the inductive definition:

(i) If t is a constant or a variable x,

$$t^* = \text{mkseq}\,(x).$$

(ii) If t has the form $g(u_1, \cdots, u_{\#g})$ where $g \in G - \{\text{if-then-else}\}$,

$$t^* = u_{i_1}^* \circ \cdots \circ u_{i_k}' \circ \text{mkseq}\,(g(\text{last}\,(u_1^*), \cdots, \text{last}\,(u_{\#g}^*))),$$

where $u_{i_1}, \cdots, u_{i_k}$ are all the arguments containing invocations of some program function f_i.

(iii) If t has the form $f_i(u_1, \cdots, u_{\#f_i})$,

$$t^* = u_{i_1}^* \circ \cdots \circ u_{i_k}^* \circ f_i^*\,(\text{last}\,(u_1^*), \cdots, \text{last}\,(u_{\#f_i}^*)),$$

where $u_{i_1}, \cdots, u_{i_k}$ are all the arguments containing invocations of some program function f_i, except when this list is empty. In this case, $k = 1$ and $u_{i_1}^*$ is mkseq (true).

(iv) If t has the form if u_0 then u_1 else u_2,

$$t^* = \text{if last}\,(u_0^*) \text{ then } u_1^* \text{ else } u_2^*,$$

when u_0^* does not contain invocations of some program function f_i. Otherwise,

$$t^* = u_0^* \circ (\text{if last}\,(u_0^*) \text{ then } u_1^* \text{ else } u_2^*).$$

In subsequent examples, we will always use this construction since it produces simpler translations.

10. Sample proofs involving complete recursive programs. To illustrate how complete programs can be used to prove theorems about partial functions, we present two examples.

10.1. Example 1. A divergent function. Let f be the partial function on the natural numbers defined by the recursive program

(11) $\quad f(x) = f(x+1).$

We want to prove the following theorem.

THEOREM. $\forall x[f(x)=\perp]$.

Proof. Although the intended meaning of f (the least fixed-point of the functional for the program (11)) is everywhere undefined, we cannot establish this property using conventional first order programming logic since f is total in many other models. On the other hand, in extended first order programming logic, we can prove the theorem by using the equivalent complete program

$$f^*(x)=\text{mkseq}\,(x+1)\circ f^*(x+1)$$

and the axiom

$$\forall x\, f(x)=\text{last}\,(f^*(x))$$

relating it to the original program (11). Since last is strict, the theorem is an immediate consequence of the following lemma. □

LEMMA. $\forall x[f^*(x)=\perp]$.

Proof. We prove the lemma by structural induction on the (possibly $\perp$) sequence $f^*(x)$.

Basis: $f(x)=\perp$.
In this case, the theorem is true by assumption.

Induction step: $f^*(x)\neq\perp$.
By induction, we may assume that the lemma holds for all x_0 such that $f^*(x_0)$ is a proper tail of $f^*(x)$. Since $x\in N$, $f^*(x)$ expands into the expression mkseq $(x)\circ f^*(x+1)$. Instantiating the induction hypothesis for $x_0=x+1$ yields $f^*(x+1)=\perp$, implying $f^*(x)=\perp$. □

10.2. Example 2. Equivalence of two program schemes. A more interesting example is the proof that the following two iterative program schemes are equivalent.

```
Program A                        Program B
{                                {
 x ← f(x);                        repeat x ← f(x) until p(x);
 while ¬p(x) do x ← f(x);         return (x)
 return (x)
}                                }
```

Expressed as recursive programs, program A and program B have the following form:

```
progA (x) ← whileA (f(x))
whileA (x) ← if p(x) then x else whileA (f(x))

progB (x) ← repeatB (x)
repeatB (x) ← if p(f(x)) then f(x) else repeatB (f(x)).
```

We wish to prove the following formal theorem.

THEOREM. $\forall x[\text{progA}\,(x)=\text{progB}\,(x)]$.

Proof. By simplification, the theorem trivially reduces to the statement

$$\forall x[\text{whileA}\,(f(x))=\text{repeatB}\,(x)]. \tag{12}$$

If $f(x)$ is not total, this statement is not provable in conventional first order programming logic, because the recursive definitions of whileA and repeatB may have extraneous fixed-points. However, in extended first order programming logic, the proof is straightforward. Given the equivalent complete programs

```
whileA* (x) ← if p(x) then mkseq (x) else cons (x, whileA* (f(x)))
repeatB* (x) ← if p(f(x)) then mkseq (f(x)) else cons (x, repeatB* (f(x)))
```

and the axioms (from extended first order programming logic)

$$\begin{aligned}&\text{cons}\,(x, y) = \text{mkseq}\,(x) \circ y\\&\text{last}\,(\text{mkseq}\,(x)) = x\\&x \neq \perp \supset \text{last}\,(\text{cons}\,(x, y)) = \text{last}\,(y)\\&\text{whileA}\,(x) = \text{last}\,(\text{whileA}^*\,(x))\\&\text{repeatB}\,(x) = \text{last}\,(\text{repeatB}^*\,(x))\end{aligned}$$

relating the complete program to the original one, statement (12) reduces to the sentence:

(13) $\forall x[\text{last}\,(\text{whileA}^*\,(f(x)) = \text{last}\,(\text{repeatB}^*\,(x))]$.

As in higher order logics based on fixed-point induction (e.g., Edinburgh LCF), the proof of (13) breaks down into two parts:

(13a) $\forall x[\text{last}\,(\text{whileA}^*\,(f(x)) \subseteq \text{last}\,(\text{repeatB}^*\,(x))]$

(13b) $\forall x[\text{last}\,(\text{repeatB}^*\,(x)) \subseteq \text{whileA}^*\,(f(x))]$

where $\alpha \subseteq \beta$ intuitively means "α approximates β" (as defined in § 8) and formally abbreviates the formula

$$\alpha = \perp \vee \alpha = \beta.$$

The proof of (13a) proceeds as follows.

First, we can assume that $\text{last}\,(\text{whileA}^*\,(f(x))) \neq \perp$; otherwise, (13a) trivially holds. Given this assumption and the fact that last is strict (which follows immediately from the definition of last), we can apply structural induction on $\text{whileA}^*\,(f(x))$. As the induction hypothesis we assume that

$$\text{last}\,(\text{whileA}^*\,(f(x'))) \subseteq \text{last}\,(\text{repeatB}^*\,(x'))$$

for all x' such that $\text{whileA}^*\,(f(x'))$ is a proper subtail of $\text{whileA}^*\,(f(x))$. By the definition of whileA*,

$$\text{whileA}^*\,(f(x)) = \text{if } p(f(x)) \text{ then mkseq}\,(f(x)) \text{ else cons}\,(f(x), \text{whileA}^*\,(f(f(x)))).$$

A three-way case analysis on the value of $p(f(x))$ completes the proof of (13a).

Case (a). $p(f(x)) \in D_{\perp}$.

In this case, $\text{whileA}^*\,(f(x)) = \perp$, which is a contradiction.

Case (b). $p(f(x)) \in D_{\text{true}}$.

By simplification,

$$\text{last}\,(\text{whileA}^*\,(f(x))) = f(x) \text{ and last}\,(\text{repeatB}^*\,(x)) = f(x)$$

establishing (13a).

Case (c). $p(f(x)) \in D_{\text{false}}$.

Obviously,

$$\text{whileA}^*\,(f(x)) = \text{cons}\,(f(x), \text{whileA}^*\,(f(f(x))))$$

and

$$\text{repeatB}^*\,(x) = \text{cons}\,(f(x), \text{repeatB}^*\,(f(x)))$$

implying that $\text{whileA}^*\,(f(f(x)))$ is shorter than $\text{whileA}^*\,(f(x))$. Consequently, the induction hypothesis holds for $x' = f(x)$, yielding the following chain of simplifications:

$$\begin{aligned}\text{last}(\text{whileA}^*(f(x))) &= \text{last}(\text{cons}(f(x), \text{whileA}^*(f(f(x)))))\\ &= \text{last}(\text{whileA}^*(f(f(x))))\\ &\subseteq \text{last}(\text{repeatB}^*(f(x))) \quad \text{(by induction)}\\ &= \text{last}(\text{repeatB}^*(x)),\end{aligned}$$

which proves (13a).

Since the proof of (13b) is nearly identical, it is omitted. □

An interesting property of proofs based on complete recursive programs is their similarity to the corresponding proofs based on fixed-point induction in higher order logics. Although fixed-point induction is an awkward rule for reasoning about total functions, it appears well suited to proving many properties of partial functions.

11. Advantages of first order programming logic. Although first order programming logic is narrower in scope than higher order logics (since it does not accommodate functions that take functions as arguments), it is a powerful, yet natural formalism for proving properties of recursive programs. The primary advantage of first order programming logic is that it relies on the familiar principle of structural induction, the most important proof technique in discrete mathematics. In first order programming logic, programmers can develop proofs that are direct formalizations of familiar informal structural induction arguments. In contrast, higher order logics for recursive programs (e.g. Edinburgh LCF [15]) typically rely on fixed-point induction, a rule that is more obscure and difficult to use.

A particularly troublesome aspect of fixed-point induction is that it is valid only for admissible formulas.[10] Edinburgh LCF [15], for example, restricts the application of fixed-point induction to formulas that pass a complex syntactic test ensuring admissibility. Unfortunately, it is often difficult to predict whether a particular formula will pass the test. Moreover, the test does not necessarily produce consistent results on logically equivalent formulas.

As an illustration, consider the sample proof presented in § 5: the termination of the recursive program flat. The proof using first order programming logic is a direct translation of the obvious informal proof. In contrast, a proof of the same theorem in Edinburgh LCF (or similar higher order logic) must introduce a retraction characterizing the domain of S-expressions and simulate structural induction by performing fixed-point induction on the retraction. In the fixed-point induction step, the programmer (or theorem prover) must check that the formula is admissible (by applying the syntactic test) before applying the rule.

For the reasons cited above, we believe that first order programming logic—rather than a higher order logic—is the appropriate formal system for proving properties of recursive programs. Both Boyer and Moore [3], [4] and Cartwright [6], [7] have successfully applied first order programming logic to prove the correctness of sophisticated LISP programs with relative ease. On the other hand, the practical utility of extended first order programming logic for reasoning about non-total functions (such as interpreters) has yet to be determined. Moreover, it is not obvious that the particular complete recursive program construction presented in this paper is the best way to translate an arbitrary program into an equivalent complete program. There are many different equivalence preserving program transformations that generate complete programs. The few examples that we have done manually are encouraging, but we

[10] The formula $\alpha[f]$ in the language $L_D \cup \{f\}$ is *admissible with respect to f over the continuous domain* **D** iff for every ascending chain $\mathbf{f}_0 \subseteq \mathbf{f}_1 \subseteq \cdots \subseteq \mathbf{f}_k \subseteq \cdots$ over **D**, $\mathbf{D} \cup \mathbf{f}[\alpha[f]]$ is TRUE for all functions **f** in the chain implies that $\mathbf{D} \cup \mathbf{f}\,[\alpha[f]]$ is **TRUE** for **f** defined as $\mathbf{lub}\{\mathbf{f}_k \mid k \geqq 0\}$.

cannot reach a firm conclusion until we build a machine implementation and experiment with various schemes for translating arbitrary programs into complete ones.

12. Related and future research. A group of Hungarian logicians—Andreka, Nemeti, and Sain—have independently developed a programming logic [1] with metamathematical properties similar to first order programming logic, although the pragmatic details are completely different. Their logic formalizes flowchart programs as predicate definitions within a first order theory of the data domain excluding $\perp$. Given a flowchart program P, they generate a formula $\pi_P(x, y)$ that is true (in the standard model of the data domain) iff y is the output produced by applying program P to input x. In contrast to first order programming logic, the notion of computation embedded in their logic applies to *all* models of the data domain theory.[11] We are confident that an analogous result holds for first order programming logic; we intend to formulate and prove it in a subsequent paper.

As a formal system for reasoning about recursive programs, the major limitation of first order programming logic as formulated in this paper is that it does not accommodate "higher order" data domains—structures that are not flat. In practice, this restriction may not be very important since higher order objects can always be modeled by intensional descriptions (e.g., computable functions as program text). Nevertheless, we believe that an important direction for future research is to extend first order programming logic to "higher-order" domains. With this objective in mind, we are exploring the implications of allowing lazy (nonstrict) constructors (e.g., lazy cons in LISP) in the data domain.

Appendix I. Sample first order axiomatizations. A conventional first order axiomatization A for the structure $\mathbf{N}$, the natural numbers with functions $\{\mathbf{0}, \mathbf{suc}, +, \times\}$, is:

(1) $\forall x[x=0 \oplus \exists! y\, x=\text{suc}(y)]$.
(2) $\forall y[0+y=y]$.
(3) $\forall x, y[\text{suc}(x)+y=\text{suc}(x+y)]$.
(4) $\forall x, y[0\times y=0]$.
(5) $\forall x, y[\text{suc}(x)\times y=y+(x\times y)]$.
(6) $(\alpha(0)\wedge \forall x[\alpha(x)\supset \alpha(\text{suc}(x))])\supset \forall x\alpha(x)$ for every formula $\alpha(x)$.

The corresponding axiomatization A^+ for the arithmetic domain $\mathbf{N}^+$, consisting of the universe $|N|\cup\{\perp\}$ and functions $\{\mathbf{0}, \mathbf{true}, \mathbf{false}, \mathbf{suc}, \mathbf{equal}, +, \times, \mathbf{if\text{-}then\text{-}else}\}$, is:

(1) $\forall x:N[x=0\oplus \exists! y:N x=\text{suc}(y)]$.
(2) $\forall y:N[0+y=y]$.
(3) $\forall x, y:N[\text{suc}(x)+y=\text{suc}(x+y)]$.
(4) $\forall x, y:N[0\times y=0]$.
(5) $\forall x, y:N[\text{suc}(x)\times y=y+(x\times y)]$.
(6) $\alpha(0)\wedge \forall x:N[\alpha(x)\supset \alpha(\text{suc}(x))]\supset \forall x:N\alpha(x)$ for every formula $\alpha(x)$.
(7) $0:N$.
(8) $\forall x:N[\text{suc}(x):N]$.
(9) $\text{suc}(\perp)=\perp$.
(10) $\forall y[\perp+y=\perp\wedge y+\perp=\perp]$.
(11) $\forall x[x\times\perp=\perp\wedge\perp\times x=\perp]$.
(12) $\text{true}=\text{suc}(0)\wedge \text{false}=0$.

[11] Presumably, their notion of nonstandard computation is closely related to the concept of least definable-fixed-points presented in this paper.

(13) $\forall x, y : N[(x = y \supset (x \text{ equal } y) = \text{true}) \wedge (x \neq y \supset (x \text{ equal } y) = \text{false})]$.
(14) $\forall x[(\perp \text{ equal } x) = \perp \wedge (x \text{ equal } \perp) = \perp]$.
(15) $\forall y, z[\text{if false then } y \text{ else } z = z]$.
(16) $\forall x : N\ \forall y, z[\text{if suc}(x) \text{ then } y \text{ else } z = y]$.
(17) $\forall y, z\ [\text{if } \perp \text{ then } y \text{ else } z = \perp]$.

where $x : N$ abbreviates the formula $x \neq \perp$.

Appendix II. Sample proofs in first order programming logic.

Example 1. *Termination of the countdown function.* Let the function zero over the natural numbers **N** augmented by $\{\perp\}$ be defined by the (call-by-name) recursive program:

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

which is logically equivalent (on $|N|$) to the definition for zero in § 3:

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

We will prove a theorem asserting that the function zero equals 0 for all natural numbers.

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

Proof. The proof proceeds by induction on n.

Basis: $n = 0$.

This case is trivial by simplification: $\text{zero}(n) = \text{if } 0 \text{ equal } 0 \text{ then } 0 \text{ else zero}(n-1) = 0$.

Induction step: $n > 0$.

We assume by hypothesis that 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.

Example 2. *Termination of an Ackermann function.* Let the function ack over the natural numbers **N** augmented by $\{\perp\}$ be defined by the call-by-value recursive program:

```
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.

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

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

Basis: $x = 0$.

By assumption, $y \neq \perp$. Hence, $\text{ack}(x, y) = \text{suc}(y) \neq \perp$.

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 $y \neq \perp$ by assumption,

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

Case (a). $y = 0$.

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

Case (b). $y > 0$.

In this case,

$$\text{ack}(x, y) = \text{ack}(\text{pred}(x), \text{ack}(x, \text{pred}(y))).$$

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

Example 3. *McCarthy's* 91*-function.* Let the function $f91$ over the integers augmented by $\{\perp\}$ be defined by the (call-by-name) recursive program

$$\begin{aligned} f91(n) = &\text{if } n > 100 \text{ then } n-10 \\ &\text{else } f91(f91(n+11)). \end{aligned}$$

We will prove the following theorem implying $f91$ is total over the integers.

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

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

$$x \ominus y = \text{if } (x-y) > 0 \text{ then } x-y \text{ else } 0.$$

Basis: $101 \ominus n = 0$.

Clearly, $n > 100$, implying $f91(n) = n - 10$, which is exactly what the theorem asserts.

Induction step: $101 \ominus n > 0$.

By hypothesis, we assume the theorem holds for n' such that $101 \ominus n' < 101 \ominus < n$, i.e. $n' > n$. By the definition of $f91$,

$$f91(n) = f91(f91(n+11)) = f91(\text{if } n+11 > 100 \text{ then } n+1 \text{ else } 91)$$

(by induction since $n+11 > n$).

By assumption, $n \leqq 100$. Consequently, there are two cases we must consider.

Case (a). $n+11 > 100$.

Obviously, $100 \geqq n > 89$, implying

$$f91(n) = f91(n+1) = \text{if } n+1 > 100 \text{ then } n-9 \text{ else } 91 = 91 \text{ (since } n \leqq 100).$$

Case (b). $n+11 \leqq 100$.

By assumption, $n \leqq 89$, implying

$$f91(n) = f91(91) = \text{if } 91 < 100 \text{ then } 91-10 \text{ else } 91$$

(by induction since $91 > n) = 91$. □

Appendix III. Proof of call-by-value fixed point normalization theorem.

THEOREM. *Let P be a call-by-value recursive program over an arithmetic data domain* $\mathbf{D}$ and *let* $\mathbf{F}$ (*abbreviating* $[\mathbf{f}_1, \cdots, \mathbf{f}_n]$) *denote the least fixed-point of the call-by-value functional* $\mathbf{P}_\perp$ *for P. The complete recursive program P^* corresponding to P has the following properties*:

(i) *P^* is complete, i.e. the corresponding call-by-value functional* $\mathbf{P}'_\perp$ *as a unique fixed-point* $[\mathbf{f}_1^*, \cdots, \mathbf{f}_n^*]$.

(ii) *For $i = 1, \cdots, n$,* $\mathbf{last}(\mathbf{f}_i^*(\bar{d})) = \mathbf{f}_i(\bar{d})$ *for all $\# f_i$-tuples $\bar{d}$ over $|D|$.*

Proof. By definition, $\mathbf{F}$ is the least upper bound of the chain of approximating n-tuples of functions

$$\mathbf{F}^{(0)} \subseteq \mathbf{F}^{(1)} \subseteq \cdots \subseteq \mathbf{F}^{(k)} \subseteq \cdots$$

where $\mathbf{F}^{(k)} = [\mathbf{f}_1^{(k)}, \cdots, \mathbf{f}_n^{(k)}]$ is inductively defined $\forall k \geqq 0$ by

$$\mathbf{F}^{(0)} = [\lambda \bar{x}_1 \cdot \perp, \cdots, \lambda \bar{x}_n \cdot \perp]$$

$$\mathbf{F}^{(k+1)} = \mathbf{P}_\perp(\mathbf{F}^{(k)}).$$

Let $\mathbf{D}^{(k)}$ denote the structure $\mathbf{D} \cup \mathbf{F}^{(k)}$. In informal terms, $\mathbf{D}^{(k)}$ is $\mathbf{D}$ augmented by the call-by-value evaluation of P to depth k. Similarly, let $\mathbf{D}^{*(k)}$ denote the structure $\mathbf{D}^* \cup \mathbf{F}^{*(k)}$ where $\mathbf{F}^{*(k)}$ is the depth k approximation for P^* analogous to $\mathbf{F}^{(k)}$.

In the course of this proof, we will frequently employ the following lemma without explicitly citing it.

LEMMA 0. *For every term t in L_P, every state s over $|D|$, and all $k \geqq 0$,* $\mathbf{D}^{*(k)}[\mathrm{seq}\,(t^*)][s] = \mathbf{TRUE}$, i.e., *$t^*$ denotes a sequence code in $\mathbf{D}^*$.*

Proof of Lemma 0. A routine induction on the structure of t. Omitted. □

Property (ii) of the main theorem is an immediate consequence of the following lemma.

LEMMA 1. *For every term t in L_P, every state s over $|D|$, and all $k \geqq 0$, $\mathbf{D}^{(k)}[t][s] = \mathbf{D}^{*(k)}[\mathrm{last}\,(t^*)][s]$.*

Proof of Lemma 1. The proof proceeds by induction on the pair $[k, t]$. By hypothesis, we may assume that the lemma holds for all $[q, u]$ where either $q < k$ and u is arbitrary, or $q = k$ and u is a proper subterm of t.

Case (a). t is a constant or a variable x. Then t^* has the form mkseq(x) implying $\mathbf{D}^{*(k)}[\mathrm{last}\,(t^*)][s] = \mathbf{D}^{*(k)}[x][s] = \mathbf{D}^{(k)}[t][s]$ for arbitrary $k \geqq 0$.

Case (b). t has the form $g(u_1, \cdots, u_{\#g})$, $g \in G \cup \mathrm{Seq}$. Then

$$t^* = u_1^* \circ \cdots \circ u_{\#g}^* \circ \mathrm{mkseq}\,(g(\mathrm{last}\,(u_1^*), \cdots, \mathrm{last}\,(u_{\#g}^*))).$$

By hypothesis, $\mathbf{D}^{*(k)}[\mathrm{last}\,(u_i^*)][s] = \mathbf{D}^{(k)}[u_i][s]$, for all $s, i = 1, \cdots, \#g$. If for some i, $\mathbf{D}^{(k)}[u_i][s] = \bot$, then $\mathbf{D}^{*(k)}[\mathrm{last}\,(u_i^*)] = \bot$, implying $\mathbf{D}^{(k)}[t][s] = \mathbf{D}^{*(k)}[\mathrm{last}\,(t^*)][s] = \bot$, since $\mathbf{g}$, $\circ$, and **last** are all strict functions. On the other hand, if $\mathbf{D}^{(k)}[u_i][s] \neq \bot$ for all i, we conclude by the induction hypothesis and the definition of last that $\mathbf{D}^{*(k)}[u_i^*][s] \neq \bot$ for all i, implying

$$\begin{aligned}\mathbf{D}^{*(k)}[\mathrm{last}\,(t^*)][s] &= \mathbf{D}^{*(k)}[g(\mathrm{last}\,(u_i^*), \cdots, \mathrm{last}\,(u_{\#g}^*))][s] \\ &= \mathbf{D}^{(k)}[g(u_1, \cdots, u_{\#g})][s] \quad \text{(by induction)} \\ &= \mathbf{D}^{(k)}[t][s].\end{aligned}$$

Case (c). t has the form $f_i(u_1, \cdots, u_{\#f_i})$. If $k = 0$, the proof is trivial. For positive k, the proof breaks down into two cases. First, if $\mathbf{D}^{(k)}[u_j][s] = \bot$ for some j, then the lemma holds by exactly the same reasoning as in case 2. On the other hand, given $\mathbf{D}^{*(k)}[u_j^*][s] \neq \bot$ for all j, we deduce that

$$\begin{aligned}\mathbf{D}^{*(k)}[\mathrm{last}\,(t^*)][s] &= \mathbf{D}^{*(k)}[\mathrm{last}\,(f_i^*\,(\mathrm{last}\,(u_1^*), \cdots, \mathrm{last}\,(u_{\#f_i}^*)))][s] \\ &= \mathbf{D}^{*(k)}[\mathrm{last}\,(f_i^*\,(\bar{x}))][s^*]\end{aligned}$$

where s^* is a state binding each variable x_j to $\mathbf{D}^{*(k)}[\mathrm{last}\,(u_j^*)][s] = \mathbf{D}^{(k)}[u_j][s]$, $j = 1, \cdots, \#f_i$. Since no x_j is bound to $\bot$, we can expand $f_i^*\,(\bar{x})$ to produce

$$\begin{aligned}\mathbf{D}^{*(k)}[\mathrm{last}\,(t^*)][s^*] &= \mathbf{D}^{*(k-1)}[\mathrm{last}\,(t_i^*)][s^*] \\ &= \mathbf{D}^{(k-1)}[t_i][s^*] \quad \text{(by induction)} \\ &= \mathbf{D}^{(k)}[f_i(\bar{x})][s^*] \\ &= \mathbf{D}^{(k)}[t][s].\end{aligned}$$

Case (d). t has the form if u_0 then u_1 else u_2. By definition $t^* = u_0^* \circ$ (if last (u_0^*) then u_1^* else u_2^*). If $\mathbf{D}^{(k)}[u_0][s] = \bot$, then by induction $\mathbf{D}^{*(k)}[\mathrm{last}\,(u_0^*)] = \bot$, implying $\mathbf{D}^{*(k)}[\mathrm{last}\,(t^*)][s] = \bot$ and $\mathbf{D}^{(k)}[t][s] = \bot$. *On the other hand, if $\mathbf{D}^{(k)}[u_0][s] \neq \bot$*, then it denotes either "true" or "false". If $\mathbf{D}^{(k)}[u_0][s]$ is a "true" element of $\mathbf{D}$ then $\mathbf{D}^{(k)}[t][s] = \mathbf{D}^{(k)}[u_1][s]$. By induction,

$$\mathbf{D}^{(k)}[u_0][s] = \mathbf{D}^{*(k)}[\text{last}\,(u_0^*)][s]$$

implying

$$\begin{aligned}\mathbf{D}^{*(k)}[t^*][s] &= \mathbf{D}^{*(k)}[\text{last}\,(u_1^*)][s]\\ &= \mathbf{D}^{(k)}[u_1][s] \quad \text{(by induction)}\\ &= \mathbf{D}^{(k)}[t][s].\end{aligned}$$

An analogous argument proves the "false" case. □ (Lemma 1)

We prove property (ii) of the theorem as follows. By Lemma 1,

$$\begin{aligned}&\mathbf{D}^{(k)}[f_i(x_1, \cdots, x_{\#f_i})][s]\\ &\quad = \mathbf{D}^{*(k)}[\text{last}\,(\text{mkseq}\,(x_1) \circ \cdots \circ \text{mkseq}\,(x_{\#f_i}) \circ f_i^*\\ &\qquad\qquad (\text{last}\,(\text{mkseq}\,(x_1)), \cdots, \text{last}\,(\text{mkseq}\,(x_{\#f_i}))))][s]\end{aligned}$$

for all $k \geqq 0$, all states s over $|D|$. Simplifying the right-hand side of the preceding equation yields

$$\mathbf{D}^{(k)}[f_i(x_1, \cdots, x_{\#f_i})][s] = \mathbf{D}^{*(k)}[\text{last}\,(f_i^*\,(x_1, \cdots, x_{\#f_i}))][s]$$

for all states s over $|D|$. Since $\mathbf{D}$ and $\mathbf{D}^*$ are both flat domains, the functions $\mathbf{f}_i$ and $\mathbf{f}_i^*$ have the following property. For any $\#f_i$-tuple d over $|D|$ there exists $p \geqq 0$ such that

$$\mathbf{f}_i^{(p)}(\bar{d}) = \mathbf{f}_i(\bar{d})$$

and

$$\mathbf{f}_i^{*(p)}(\bar{d}) = \mathbf{f}_i^*(\bar{d}).$$

Let s_d be a state mapping $\bar{x}$ into $\bar{d}$. Then

$$\begin{aligned}\mathbf{f}_i(\bar{d}) = \mathbf{f}_i^{(p)}(\bar{d}) &= \mathbf{D}^{(p)}[f_i(\bar{x})][s_d]\\ &= \mathbf{D}^{*(p)}[\text{last}\,(f_i^*(\bar{x}))][s_d] = \textbf{last}\,(\mathbf{f}_i^{*(p)}(\bar{d})) = \textbf{last}\,(\mathbf{f}_i^*(\bar{d}))\end{aligned}$$

proving property (ii).

To prove property (i), we must introduce some new definitions. Let $\mathbf{H}$ be an n-tuple of strict functions $[\mathbf{h}_1, \cdots, \mathbf{h}_n]$ over $\mathbf{D}$ corresponding to the n-tuple of function symbols $[f_1^*, \cdots, f_n^*]$. We define $\mathbf{D}_H^{*(k)}$ for all $k \geqq 0$ as the structure corresponding to L_P^* that is identical to $\mathbf{D}^{*(k)}$ except that $\mathbf{D}_H^{*(k)}$ interprets $[f_1^*, \cdots, f_n^*]$ by $\mathbf{F}_H^{*(k)}$ where $\mathbf{F}_H^{*(k)}$ is inductively defined by

$$\mathbf{F}_H^{*(0)} = \mathbf{H},$$

$$\mathbf{F}_H^{*(k+1)} = \mathbf{P}_\perp(\mathbf{F}_H^{*(k)}).$$

Informally $\mathbf{f}_{i_H}^{*(k)}$ is the function computed by applying call-by-value evaluation to the recursion equation for f_i where invocations of f_j, $j = 1, \cdots, n$, at depth k are interpreted by the function $\mathbf{h}_j$ instead of ordinary evaluation. If all the functions $\mathbf{h}_j$ in $\mathbf{H}$ are everywhere undefined, then $\mathbf{F}_H^{*(k)} = \mathbf{F}^{*(k)}$.

Property (i) is a simple consequence of the following lemma.

LEMMA 2. *Let t be any term in L_P. Let $\mathbf{H}$ be an n-tuple of strict functions $[\mathbf{h}_1, \cdots, \mathbf{h}_n]$ corresponding to $[f_1^*, \cdots, f_n^*]$. Then for any $k \geqq 0$ and any state s over $|D|$, $\mathbf{D}^*[t^*][s] = \perp$ implies either $\mathbf{D}_H^{*(k)}[t^*][s] = \perp$, or* $\textbf{length}\,(\mathbf{D}_H^{*(k)}[t^*][s]) \geqq k$.

Proof of Lemma 2. In the course of the proof, we will use the following lemmas which are easily proven by structural induction on t.

LEMMA 2a. *For any term t in L_{P^*}, $\mathbf{D}^{*(k)}[t][s] \neq \perp$ implies $\mathbf{D}^{*(k)}[t][s] = \mathbf{D}_H^{*(k)}[t][s]$.*

LEMMA 2b. *For any term t in L_{P^*} not containing any recursive function symbol f_i^*, $\mathbf{D}_H^{*(k)}[t][s] = \mathbf{D}^{*(k)}[t][s]$ for arbitrary state s.*

Proof of Lemmas 2a *and* 2b. Omitted. □

To prove lemma 2, we apply induction on the pair $[k, t]$.

Basis: $k = 0$. In this case, the lemma is a trivial consequence of the definition of **length** and the fact that the meaning of any computation sequence term t^* under any state s in the structure $\mathbf{D}^*$ is either $\perp$ or a sequence code.

Induction step: $k > 0$. We perform a case split on the structure of t.

Case (a). t is a constant or variable x. Then $t^* = \text{mkseq}(x)$, implying by Lemma 2b that $\mathbf{D}_H^{*(k)}[t^*][s] = \mathbf{D}^{*(k)}[t^*][s]$ which is $\perp$ by assumption.

Case (b). t has the form $g(u_1, \cdots, u_{\#g})$ where $g \in G$. In this case, $t^* = u_1^* \circ \cdots \circ u_{\#g}^* \circ \text{mkseq}(g(\text{last}(u_1^*), \cdots, \text{last}(u_{\#g}^*)))$. If $\mathbf{D}^{*(k)}[u_j^*][s] = \perp$, then by induction, either $\mathbf{D}_H^{*(k)}[u_j^*][s] = \perp$ or $\textbf{length}(\mathbf{D}_H^{*(k)}[u_j^*][s]) \geqq k$, implying the lemma holds (since $\circ$ is strict). On the other hand, if $\mathbf{D}^{(k)}[u_j^*][s] \neq \perp$ for all j, then by Lemma 2a, $\mathbf{D}_H^{*(k)}[u_j^*][s] = \mathbf{D}^{*(k)}[u_j^*][s]$ for all j. Consequently,

$$\begin{aligned}\mathbf{D}_H^{*(k)}[t^*][s] &= \mathbf{D}_H^{*(k)}[u_1^* \circ \cdots \circ u_{\#g}^* \circ \text{mkseq}(g(\text{last}(u_1^*), \cdots, \text{last}(u_{\#g}^*)))][s] \\ &= \mathbf{D}^{*(k)}[u_1^* \circ \cdots \circ u_{\#g}^* \circ \text{mkseq}(g(\text{last}(u_1^*), \cdots, \text{last}(u_{\#g}^*)))][s] \\ &= \mathbf{D}^{*(k)}[t^*][s].\end{aligned}$$

implying the lemma holds.

Case (c). t has the form $f_i(u_1, \cdots, u_{\#f_i})$. If $\mathbf{D}^{*(k)}[u_j^*][s] = \perp$ for some j, the proof is identical to the analogous section of the previous case. On the other hand, when $\mathbf{D}^{*(k)}[u_j^*][s] \neq \perp$ for all j,

$$\begin{aligned}\mathbf{D}_H^{*(k)}[t^*][s] &= \mathbf{D}^{*(k)}[u_1^*][s] \circ \cdots \circ \mathbf{D}^{*(k)}[u_{\#f_i}^*] \\ &\quad [s] \circ \mathbf{f}_{i_H}^{*(k)}(\textbf{last}(\mathbf{D}^{*(k)}[u_1^*][s]), \cdots, \textbf{last}(\mathbf{D}^{*(k)}[u_{\#f_i}^*][s])) \\ &= \mathbf{D}^{*(k)}[u_1^*][s] \circ \cdots \circ \mathbf{D}^{*(k)}[u_{\#f_i}^*][s] \circ \mathbf{D}_H^{*(k)}[f_i^*(\bar{x})][s^*] \\ &= \mathbf{D}^{*(k)}[u_1^*][s] \circ \cdots \circ \mathbf{D}^{*(k)}[u_{\#f_i}^*][s] \circ \mathbf{D}_H^{*(k-1)}[t_i^*][s^*]\end{aligned}$$

where s^* maps x_j into $\textbf{last}(\mathbf{D}^{*(k)}[u_j^*][s])$, $j = 1, \cdots, \#f_i$. By induction, either $\mathbf{D}_H^{*(k-1)}[t_i^*][s^*] = \perp$ or $\textbf{length}(\mathbf{D}_H^{*(k-1)}[t_i^*][s^*]) \geqq k-1$. Hence the lemma clearly holds (since $\textbf{length}(u_j^*) \geqq 1$ for all j).

Case (d). t has the form if u_0 then u_1 else u_2. If $\mathbf{D}^{*(k)}[u_0^*][s] = \perp$, the proof is identical to the analogous section of case (b). On the other hand, when $\mathbf{D}^{(k)}[u_0][s] \neq \perp$, $\mathbf{D}^{(k)}[u_0][s]$ is either a "true" element of $|D|$ or a "false" one. In the former case, $\mathbf{D}^{*(k)}[t][s] = \mathbf{D}^{*(k)}[u_1^*][s]$ and $\mathbf{D}_H^{*(k)}[t^*][s] = \mathbf{D}_H^{*(k)}[u_1^*][s]$. By induction, $\mathbf{D}^{*(k)}[u_1^*][s] = \perp$, simplying either $\mathbf{D}_H^{*(k)}[u_1^*] = \perp$ or $\textbf{length}(\mathbf{D}_H^{*(k)}[u_1^*]) \geqq k$. Hence the lemma holds in this case. An analogous argument holds for the "false" case. □ (Lemma 2)

Given Lemma 2, we prove that property (i) holds by the following argument. Let $\mathbf{H} = [\mathbf{h}_1, \cdots, \mathbf{h}_n]$ be any fixed-point of the call-by-value function $\mathbf{P}_\perp$ for the complete recursive program P^*, i.e., for all i

$$\mathbf{D}_H^{*(0)}[f_i^*(\bar{x}_i)][s] = \mathbf{D}_H^{*(0)}[t_i^*][s].$$

By induction on $[k, t]$ we can easily show for all $k \geqq 0$, all terms t in L_{P^*}, and all states s that

$$\mathbf{D}_H^{*(k)}[t^*][s] = \mathbf{D}_H^{*(0)}[t^*][s].$$

Now assume $\mathbf{H}$ is not the least fixed-point of $\mathbf{P}_\perp$, i.e., $\mathbf{f}_i^*(\bar{d}) = \perp$ but $\mathbf{h}_i(\bar{d}) \neq \perp$ for some i and some $\#f_i$-tuple $\bar{d}$ over $|D|$. Then,

$$\forall k \geqq 0[\mathbf{D}_H^{*(k)}[f_i^*(\bar{x}_i)][s_d] = \mathbf{D}_H^{*(0)}[f_i^*(\bar{x}_i)][s_d] = \mathbf{h}_i(\bar{d})]$$

where s_d binds $\bar{x}_i$ to $\bar{d}$. By Lemma 2, the length of $\mathbf{h}_i(\bar{d})$ is greater than any number k, contradicting the fact that

$$\exists k[\mathbf{length}\,(\mathbf{D}_H^{*(0)}[f_i^*(\bar{x}_i)](s_d)) \leqq k]. \quad \square$$

Appendix IV. Call-by-name complete recursive programs. The call-by-value recursive program construction described in § 8 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 to construct complete call-by-name programs.

Unfortunately, call-by-name computation sequences have a more complex structure than their call-by-value counterparts. The chief complication is that the collection of arguments in recursive call $f_i(\bar{t})$ that are actually evaluated depends on the particular arguments $\bar{t}$. To accommodate this complication, we adopt the convention that the new functions f_i^* in the complete recursive program take computation sequences corresponding to the arguments of f as inputs instead of the arguments themselves. Hence, the original functions $f_1, \cdots, f_n$ are related to the new functions $f_1^*, \cdots, f_n^*$ by the equations

$$f_i(x_1, \cdots, x_{\#f_i}) = \text{last}\,(f_1^*(\text{mkseq}\,(x_1), \cdots, \text{mkseq}\,(x_{\#f_i}))), i = 1, \cdots, n$$

instead of

$$f_i(x_1, \cdots, x_{\#f_i}) = \text{last}\,(f^*(x_1, \cdots, x_{\#f_i})), i = 1, \cdots, n,$$

which hold for call-by-value complete recursive programs.

This convention gives the body of each new function f_i^* control over the process incorporating particular argument evaluations into the output computation sequence. If a particular argument is never evaluated, its computation sequence is discarded. Recall that in the call-by-value construction, the computation sequences for the arguments of a function call are unconditionally inserted in the computation sequence by the calling expression.

We construct the call-by-name complete recursive program P^* corresponding to P as follows.

Definition. Let $\mathbf{D}$, L_D, P, L_P, $\mathbf{F}$, $\mathbf{D}^{(k)}$, and $\mathbf{D}^*$ (including **Seq**) be defined as in § 8 except that P is a call-by-name program and, consequently, $\mathbf{F}$ is the least fixed-point of the (call-by-name) functional $\mathbf{P}$ for P over $\mathbf{D}$. Let t be an arbitrary term in the language L_P. The *call-by-name computation sequence term* t^* (in the extended language L_{P^*}) corresponding to t is inductively defined by:

(i) If t is a variable v,

$$t^* = v.$$

(ii) If t is a constant symbol c,

$$t^* = \text{mkseq}\,(c).$$

(iii) If t has the form $g(u_1, \cdots, u_{\#g})$ where $g \in G$,

$$t^* = u_1^* \circ \cdots \circ u_{\#g}^* \circ \text{mkseq}\,(g(\text{last}\,(u_1^*), \cdots, \text{last}\,(u_{\#g}^*))).$$

(iv) If t has the form $f_i(u_1, \cdots, u_{\#f_i})$,

$$t^* = \text{mkseq}\,(\text{true}) \circ f_i^*\,(u_1^*, \cdots, u_{\#f_i}^*).$$

(v) 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 P^** corresponding to P is the (call-by-name) program

$$\{f_1^*\,(\bar{x}_1) = t_1^*, \cdots, f_n^*\,(\bar{x}_n) = t_n^*\}$$

over $\mathbf{D}^*$.

THEOREM (call by-name fixed point normalization theorem). *Let P be a call-by-name recursive program over an arithmetic data domain $\mathbf{D}$ supporing elementary syntax and let $[\mathbf{f_1}, \cdots, \mathbf{f_n}]$ denote the least fixed-point of the corresponding (call-by-name) functional $\mathbf{P}$. The complete recursive program P^* corresponding to P has the following properties*:

(i) *P^* is complete, i.e. the corresponding (call-by-name) functional $\mathbf{P}^*$ has a unique fixed-point $[\mathbf{f}_1^*, \cdots, \mathbf{f}_n^*]$.*

(ii) For $i = 1, \cdots, n$, $\textbf{last}\,(\mathbf{f}_i^*(\textbf{mkseq}\,(d_1), \cdots, \textbf{mkseq}\,(d_{\#f_i}))) = \mathbf{f}_i(\bar{d})$ for all $\#f_i$-tuples $\bar{d}$ over $|D|$.

Proof. The proof follows the same outline as the proof of the corresponding theorem for call-by-value fixed-points in § 8. Recall that $\mathbf{F}$ is the least upper bound of the chain of function n-tuples

$$\mathbf{F}^{(0)} \subseteq \mathbf{F}^{(1)} \subseteq \cdots \subseteq \mathbf{F}^{(k)} \subseteq \cdots$$

where $\mathbf{F}^{(k)} = [\mathbf{f}_1^{(k)}, \cdots, \mathbf{f}_n^{(k)}]$ is defined $\forall k \geqq 0$ by

$$\mathbf{F}^{(0)} = [\lambda \bar{x}_1 \cdot \perp, \cdots, \lambda \bar{x}_n \cdot \perp],$$

$$\mathbf{F}^{(k+1)} = \mathbf{P}(\mathbf{F}^{(k)}),$$

and that $\mathbf{D}^{(k)}$ denotes the structure $\mathbf{D} \cup \mathbf{F}^{(k)}$. Let $\mathbf{D}^{(k)*}$ denote the structure $\mathbf{D} \cup \mathbf{F}^{*(k)}$ for the call-by-name program P^* over $\mathbf{D}^*$ analogous to $\mathbf{D}^{(k)}$ for the program P over $\mathbf{D}$.

Property (ii) is a simple consequence of the following lemma.

LEMMA 1. For every term t in L_P, every state s over $|D^*|$, and all $k \geqq 0$, $\mathbf{D}^{(k)}[t][s] = \mathbf{D}^{*(k)}[\text{last}\,(t^*)][s]$.

Proof (*of Lemma* 1). The proof is essentially identical to the proof of the corresponding lemma in § 8, which proceeds by induction on the pair $[k, t]$. The details are left to the reader. □

Property (ii) follows immediately from Lemma 1 by the following argument. For any function f_i and $\#f_i$-tuple $\bar{d} = [d_1, \cdots, d_{\#f_i}]$ over D there exists $p \geqq 0$ such that

$$\mathbf{f}_i^{(p)}(\bar{d}) = f_i(\bar{d})$$

and

$$\mathbf{f}_i^{*(p)}(\textbf{mkseq}\,(d_1), \cdots, \textbf{mkseq}\,(d_{\#f_i})) = \mathbf{f}_i^*(\textbf{mkseq}\,(d_1), \cdots, \textbf{mkseq}\,(d_{\#f_i})).$$

Let s be a state mapping $\bar{x}$ into $\bar{d}$, and s_{mkseq} be the state mapping each variable x_j

into **mkseq** $(s(x_j))$. Then

$$\begin{aligned}\mathbf{f}_i(\bar{d}) &= \mathbf{f}_i^{(p)}(\bar{d})\\ &= \mathbf{D}^{(p)}[f_i(\bar{x})][s]\\ &= \mathbf{D}^{*(p)}[\text{last}\ (\text{mkseq}\ (d^*) \circ f_i^*\ (\bar{x}))][s_{\text{mkseq}}]\\ &= \mathbf{D}^{*(p)}[\text{last}\ (f_i^*\ (\bar{x}))][s_{\text{mkseq}}]\\ &= \mathbf{last}\ (\mathbf{f}_i^{*(p)}(\mathbf{mkseq}\ (d_1), \cdots, \mathbf{mkseq}\ (d_{\#f_i})))\\ &= \mathbf{last}\ (\mathbf{f}_i^{*}(\mathbf{mkseq}\ (d_1), \cdots, \mathbf{mkseq}\ (d_{\#f_i})))\end{aligned}$$

proving property (ii).

To prove property (i), we must utilize the definitions introduced in the analogous proof in Appendix II. Let $\mathbf{H}$, $\mathbf{D}_H^{*(k)}$, and $\mathbf{F}_H^{*(k)}$ be defined exactly as in §8, except substitute call-by-name semantics for call-by-value semantics in the definition of $\mathbf{F}_H^{*(k)}$ and $\mathbf{D}_H^{*(k)}$. In other words,

$$\mathbf{F}_H^{*(0)} = \mathbf{H},$$

$$\mathbf{F}_H^{*(k+1)} = \mathbf{P}(\mathbf{F}_H^{*(k)}).$$

Since the remaining details of the proof of property (i) are nearly identical to those found in the corresponding proof in Appendix III, they are omitted. □

It is a straightforward exercise to formulate and prove call-by-name analogues to the corollary and the generalization of the fixed-point normalization theorem presented in § 8 for call-by-value programs. The details are left to the reader.

Acknowledgments. After studying a very early draft of the paper, Mike O'Donnell suggested that the fixed-point normalization theorem might generalize to all models (not just the standard ones) of suitable theories, motivating the corollary and generalization of the theorem. Two of my graduate students, Alex and Vladimir Yakhnis, discovered numerous errors and omissions in earlier drafts of the paper. Immediately prior to publication, Albert Meyer suggested several technical refinements, noticed the connection between extended first order programming logic and recent research by Andreka, Nemeti and Sain [1], and proposed many improvements in the presentation.

REFERENCES

[1] H. ANDREKA, I. NEMETI AND I. SAIN, *A complete logic for reasoning about programs via nonstandard model theory*, Theoret. Comput. Sci., 17 (1982), pp. 193–212, 259–278.

[2] J. BELL AND M. MACHOVER, *A Course in Mathematical Logic*, North-Holland, New York, 1977, pp. 316–324.

[3] R. BOYER AND J. MOORE, *Proving theorems about* LISP *functions*, J. Comput. Mach., 22 (1975), pp. 129–144.

[4] R. BOYER AND J. MOORE, *A Computational Logic*, Academic Press, New York, 1979.

[5] J. CADIOU, *Recursive definitions of partial functions and their computations*, Technical Report AIM-163, Computer Science Dep., Stanford Univ., Stanford, CA, 1972.

[6] R. CARTWRIGHT, *User defined data types as an aid to verifying* LISP *programs*, Proc. Third International Colloquium on Automata, Languages, and Programming, S. Michaelson and R. Milner, eds., Edinburgh Press, Edinburgh, 1976, pp. 228–256.

[7] ———, *A practical formal semantic definition and verification system for* TYPED LISP, Stanford A.I. Memo AIM-296, Stanford Univ., Stanford, CA, 1976 (also published as a monograph in the Outstanding Dissertations in Computer Science series, Garland Publishing Company, New York, 1979).

[8] ———, *First order semantics: A natural programming logic for recursively defined functions*, Technical Report TR 78-339, Computer Science Dept., Cornell University, Ithaca, NY, 1978.

[9] ———, *Computational models for programming logics*, Technical Report, Computer Science Program, Mathematical Sciences Department, Rice University, 1983.

[10] ———, *Nonstandard fixed-points*, Proc. of Logics of Programs Workshop, Carnegie-Mellon Univ., June 1983, Lecture Notes in Computer Science, Springer-Verlag, New York, 1983.

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

[12] 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.

[13] P. Downey and R. Sethi, *Correct computation rules for recursive languages*, this Journal, 5 (1976), pp. 378–401.

[14] H. Enderton, *A Mathematical Introduction to Logic*, Academic Press, New York, 1972.

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

[16] P. Hitchcock and D. M. R. Park, *Induction rules and proofs of program termination.* Proc. First International Colloquium on Automata, Languages, and Programming, M. Nivat, ed., North-Holland, Amsterdam, 1973, pp. 225–251.

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

[18] J. McCarthy and R. Cartwright, *Representation of recursive programs in first order logic*, Stanford Artificial Intelligence Memo AIM-324, Stanford Univ., Stanford, CA, 1979.

[19] R. Milner, *Logic for computable functions—description of a machine implementation*, Stanford A.I. Memo AIM-169, Stanford Univ., Stanford, CA, 1972.

[20] ———, *Models of* LCF, Stanford A.I. Memo AIM-186, Stanford Univ., Stanford, CA, 1973.

[21]. R. Milner, L. Morris and M. Newey, *A logic for computable functions with reflexive and polymorphic types*, Proc. Conference on Proving and Improving Programs, Arc-et-Senons, 1975.

[22] M. O'Donnell, *Computing in Systems Described by Equations*, Lecture Notes in Computer Science, 58, Springer-Verlag, New York, 1977.

[23] D. M. R. Park, *Fixpoint induction and proof of program semantics*, in Machine Intelligence 5, B. Meltzer and D. Michie, eds., Edinburgh Press, Edinburgh, 1970, pp. 59–78.

[24] B. K. Rosen, *Tree-manipulating systems and Church–Rosser theorems*, J. Assoc. Comput. Mach., 20 (1973), pp. 160–187.

[25] D. Scott and J. W. deBakker, *A theory of programs*, unpublished notes, IBM seminar, Vienna, 1969.

[26] D. Scott, *Data types as lattices*, this Journal, 5 (1976), pp. 522–586.

[27] A. Tarski, *A lattice-theoretical fixpoint theorem and its applications*, Pacific J. Math., 5 (1955), pp. 285–309.

[28] J. Vuillemin, *Proof techniques for recursive programs*, Stanford A.I. Memo AIM-218, Stanford Univ., Stanford, CA, 1973.

[29] ———, *Correct and optimal implementations of recursion in a simple programming language*, J. Comput. Syst. Sci., 9 (1974), pp. 332–354.