

Research Report

DENOTATIONAL SEMANTICS AND REWRITE RULES FOR FP: PRELIMINARY VERSION

Joseph Y. Halpern
John H. Williams
Edward L. Wimmers
Timothy C. Winkler

IBM Research Laboratory
San Jose, CA 95193

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties).



Research Division
Yorktown Heights, New York • San Jose, California • Zurich, Switzerland

Copies may be requested from:
IBM Thomas J. Watson Research Center
Distribution Services
Post Office Box 218
Yorktown Heights, New York 10598

**DENOTATIONAL SEMANTICS and REWRITE RULES for FP:
PRELIMINARY REPORT***

Joseph Y. Halpern
John H. Williams
Edward L. Wimmers
Timothy C. Winkler

IBM Research Laboratory
San Jose, California 95193

ABSTRACT: We consider languages whose operational semantics is given by a set of rewrite rules. For such languages, it is important to be able to determine that there are enough rules to completely reduce all meaningful expressions, but not so many that the system of rules is inconsistent. We develop a formal framework in which to give a precise treatment of these soundness and completeness issues. We believe our approach to be novel in that we make heavy use of denotational semantics in our proof of completeness. The particular language for which we answer these questions is an extended version of the functional programming language FP; however the applicability of these techniques extends beyond the realm of FP rewriting systems.

*This is a slightly revised version of a paper that will appear in the Proceedings of the Twelfth Principles of Programming Languages Conference.

Denotational Semantics and Rewrite Rules for FP: Preliminary Version

Joseph Y. Halpern
John H. Williams
Edward L. Wimmers
Timothy C. Winkler

IBM Research Laboratory
San Jose, CA 95193

Abstract: We consider languages whose operational semantics is given by a set of rewrite rules. For such languages, it is important to be able to determine that there are enough rules to completely reduce all meaningful expressions, but not so many that the system of rules is inconsistent. We develop a formal framework in which to give a precise treatment of these soundness and completeness issues. We believe our approach to be novel in that we make heavy use of denotational semantics in our proof of completeness. The particular language for which we answer these questions is an extended version of the functional programming language FP; however the applicability of these techniques extends beyond the realm of FP rewriting systems.

1. INTRODUCTION

Recently there have appeared in the literature a number of programming languages where *rewrite* (or *reduction*) rules have been used to provide operational semantics or a means of proving program equivalence ([O'D77, Ba78, Tu79, BMS80, Wi82, GMP82, HO84]).

When rewrite rules are used in this way, there are two obvious questions that can be asked:

1. Do we have too many rules, thus making equivalent two expressions that should be different?
2. Do we have enough rules to be able to compute the correct meaning of all expressions?

In this paper we develop a formal framework in

which to give precise answers to these soundness and completeness questions. We believe our approach to be novel in that we make heavy use of denotational semantics in our proof of completeness. The particular language for which we answer these questions is an extended version of the functional programming language FP [Ba78]; however the applicability of these techniques extends beyond the realm of FP rewriting systems.

Our first step is to give a formal denotational semantics to FP. This in itself is a nontrivial task. FP can be viewed as an untyped combinatory calculus, so we have the usual difficulties of giving semantics in such a way that every expression can be viewed as both a function and an argument to a function (cf. [Me82]), as well as the problems involved in representing infinite sequences.

Once we have given the semantics, answering the first question above becomes straightforward. We must simply show that any two expressions that we intend to be distinct are assigned different meanings, and then show that the reduction rules preserve meaning. Thus if $x \rightarrow y$ is a reduction rule, we must show that $\mu(x) = \mu(y)$, where μ is the semantic function that assigns a meaning to each FP expression.

Proving completeness is a somewhat more delicate task. Part of the problem is that it is not even clear what it means for there to be "enough" rules. We might hope to be able to prove that whenever two FP expressions x and y have the same meaning (i.e., $\mu(x) = \mu(y)$) then we could show that there exists some z such that both x and y reduce to z . This turns out to be too much to expect. Just as for combinatory calculus, we know that all recursive functions can be encoded in FP [Wi81], so doing this would be at least as hard as proving the equality

of two recursive functions, a problem well known not to be recursively enumerable (cf. [Ro67]). Thus we must aim for a somewhat weaker notion of completeness.

In FP, although programmers can write higher-order functions, programs are usually written to compute *observable expressions*, where an observable expression is one that can be formed from atoms using the sequence constructor. Roughly speaking, what we will show is that if the meaning of an FP expression x is the same as that of an observable expression y , then we have enough rewrite rules to reduce x to y . Thus, at least we have a notion of completeness for expressions of the type that a programmer is most interested in.

To understand the difficulties involved in the proof of completeness, recall that in an *untyped* λ -calculus or combinatory calculus, we can define a fixpoint operator fix in FP with the property that $\text{fix}:f$ reduces to $f:(\text{fix}:f)$. This already shows that FP is not even *weakly normalizable*: there are some expressions for which no sequence of rewriting rules terminates. It is well-known (see [FLO83] for a proof) that *typed* λ -calculus is *strongly normalizable*: every reduction sequence terminates. In our proof, we construct a variant of FP, called FP' , on which, using the semantic domain, we can impose a notion of type. Once we have done this, we can apply ideas from [FLO83] to show that FP' is strongly normalizable. Then we can show that for an expression x whose meaning is the same as that of an observable expression y , we can “pull back” an FP' reduction sequence for x to an FP reduction sequence for x . To our knowledge, this is the first use of semantic techniques to prove a completeness theorem of this kind.

In other related work, Dosch and Moller [DM84] develop a set of rewriting rules similar to ours, but they take for their semantic domain the term model underlying the algebra of rules. Because their semantic domain is so intimately related to the reduction rules, it does not seem possible for them to even state a completeness theorem of the same type as ours. For them, essentially *by definition*, the meaning of x is the same as that of y iff there is some z such that both x and y can be reduced to z . (For these and other reasons we also feel that the semantic

domain we introduce here is a more natural one for programmers to use in reasoning about their programs and for us to use in designing the language.)

The rest of the paper is organized as follows. In the next section we give the syntax and semantics of FP, and show that all the reduction rules are sound. We also show that when we remove any one of the operators `apply`, `cons`, and `comp` from FP, the resulting language is strongly normalizable. (This essentially strengthens the results of [DM84].) In Section 3 we prove the completeness result. We conclude in Section 4 with some consequences of the proof of completeness, and discuss how to extend these results when additional primitive functions are added to FP.

2. SYNTAX AND SEMANTICS OF FP

The language we treat in this paper differs from the “classical” FP of [Ba78]:

1. We allow expressions whose “meaning” is an infinite sequence. For example, in classical FP it is not possible to write an expression whose meaning is the sequence of all positive integers, but it is possible in our version of FP.
2. We drop the requirement that all functions be strict, thus allowing a “lazy” evaluation strategy (see [FW76,HM76] for a discussion of lazy evaluation).
3. We use slightly different notation for the composition and construction of functions. We also have an object level conditional rather than a function level conditional.

We begin this section with a description of the syntax of FP. We assume the existence of a set $\mathcal{A}$ of *atoms* which typically includes the integers and strings of letters. We require that T and F be atoms. We also assume the existence of a set $\mathcal{F}$ of primitive functions that is disjoint from $\mathcal{A}$. The set $\mathcal{F}$ we consider in this paper is $\{\text{al}, \text{first}, \text{tl}, \text{ar}, \text{last}, \text{tr}, \text{null}, \text{comp}, \text{cons}, \text{apply}, K, \text{id}, \text{cond}\}$. This set of functions is sufficiently rich to allow us to express all recursive functions; in Section 4 we discuss how to deal with extensions of FP. The FP expressions are defined to be the least set containing $\mathcal{A} \cup \mathcal{F}$ and closed under:

1. The formation of sequences: if $x_1, \dots, x_n$ are FP expressions (for $n \geq 0$), then $\langle x_1, \dots, x_n \rangle$ is also an FP expression.

2. Application of expressions: if f and x are FP expressions, then $f:x$ is also an FP expression.

Computation of an FP expression proceeds by using the rewrite rules to transform the expression. We give below the intuition behind the primitive functions, as well as some of the rewrite rules associated with them. (Rewrite rules are sometimes called reduction rules.) The entire set of rewrite rules is given in Figure 1.

1. al takes a pair as argument and forms a new sequence by adding the first argument to the left of the second argument (if the second argument is a sequence); thus
 $al: \langle x, \langle y_1, \dots, y_n \rangle \rangle \rightarrow \langle x, y_1, \dots, y_n \rangle$.
2. $first$ takes a sequence as argument and returns the first element of the sequence; thus
 $first: \langle x_1, \dots, x_n \rangle \rightarrow x_1$, $first: (al: \langle x, y \rangle) \rightarrow x$.
3. tl removes the first element of a sequence, so
 $tl: \langle x_1, \dots, x_n \rangle \rightarrow \langle x_2, \dots, x_n \rangle$.
4. There are symmetric operations ar , $last$, and tr which operate on the right side of the sequence instead of the left in an analogous manner to al , $first$, and tl .
5. $null$ returns T when given the empty sequence and F when given a non-empty sequence or an atom. Thus, for example, $null: \langle \rangle \rightarrow T$ and $null: \langle x_1, \dots, x_n \rangle \rightarrow F$.
6. $cons$ takes a sequence of functions $\langle f_1, \dots, f_n \rangle$ as an argument and returns the function that, when applied to x , returns the sequence $\langle f_1:x, \dots, f_n:x \rangle$. Thus $(cons: \langle f_1, \dots, f_n \rangle):x \rightarrow \langle f_1:x, \dots, f_n:x \rangle$. We view the expression $[f_1, \dots, f_n]$ of [Ba78] as an abbreviation for $cons: \langle f_1, \dots, f_n \rangle$.
7. $cond$ is the if-then-else function, so $cond: \langle T, x, y \rangle \rightarrow x$ and $cond: \langle F, x, y \rangle \rightarrow y$.
8. $comp$ denotes composition, so $(comp: \langle f, g \rangle):x \rightarrow f:(g:x)$. We view $f \circ g$ as an abbreviation for $comp: \langle f, g \rangle$.
9. $apply$, when given a pair of elements, applies the first to the second; thus $apply: \langle f, x \rangle \rightarrow f:x$.
10. K is the familiar combinator from combinatory calculus that converts an expression to a constant function which always returns that expression; thus $(K:x):y \rightarrow x$.
11. id is the identity function, so $id:x \rightarrow x$.

Before proceeding to a discussion of our semantics, we give an easy proof of strong normalizability for some large subsets of FP.

Definition 2.1. An expression is a *redex* iff it is an instance of the left-hand side of one of the rules; i.e., it is the result of instantiating a *schema variable* in the left-hand side of a rewrite rule (such as the f , g , and x in $(comp: \langle f, g \rangle):x$) by an FP expression. An expression is in *normal form* iff no subexpression is a redex. An expression is *weakly normalizable* (respectively, *strongly normalizable*) iff some (respectively every) reduction sequence for it leads to a normal

$$\begin{aligned}
 al: \langle x, \langle y_1, \dots, y_n \rangle \rangle &\rightarrow \langle x, y_1, \dots, y_n \rangle. \\
 ar: \langle \langle y_1, \dots, y_n \rangle, x \rangle &\rightarrow \langle y_1, \dots, y_n, x \rangle. \\
 first: \langle x_1, \dots, x_n \rangle &\rightarrow x_1 \quad (n \geq 1). \\
 first: (al: \langle x, y \rangle) &\rightarrow x. \\
 first: (ar: \langle al: \langle x, y \rangle, z \rangle) &\rightarrow x. \\
 first: (ar: \langle ar: \langle x, y \rangle, z \rangle) &\rightarrow first: (ar: \langle x, y \rangle). \\
 last: \langle x_1, \dots, x_n \rangle &\rightarrow x_n \quad (n \geq 1). \\
 last: (ar: \langle x, y \rangle) &\rightarrow y. \\
 last: (al: \langle x, ar: \langle y, z \rangle \rangle) &\rightarrow z. \\
 last: (al: \langle x, al: \langle y, z \rangle \rangle) &\rightarrow last: (al: \langle y, z \rangle). \\
 tl: \langle x_1, \dots, x_n \rangle &\rightarrow \langle x_2, \dots, x_n \rangle \quad (n \geq 1). \\
 tl: (al: \langle x, al: \langle y, z \rangle \rangle) &\rightarrow al: \langle y, z \rangle. \\
 tl: (al: \langle x, ar: \langle y, z \rangle \rangle) &\rightarrow ar: \langle y, z \rangle. \\
 tl: (ar: \langle al: \langle x, y \rangle, z \rangle) &\rightarrow ar: \langle y, z \rangle. \\
 tl: (ar: \langle ar: \langle x, y \rangle, z \rangle) &\rightarrow ar: \langle tl: (ar: \langle x, y \rangle), z \rangle. \\
 tr: \langle x_1, \dots, x_n \rangle &\rightarrow \langle x_1, \dots, x_{n-1} \rangle \quad (n \geq 1). \\
 tr: (ar: \langle ar: \langle x, y \rangle, z \rangle) &\rightarrow ar: \langle x, y \rangle. \\
 tr: (ar: \langle al: \langle x, y \rangle, z \rangle) &\rightarrow al: \langle x, y \rangle. \\
 tr: (al: \langle x, ar: \langle y, z \rangle \rangle) &\rightarrow al: \langle x, y \rangle. \\
 tr: (al: \langle x, al: \langle y, z \rangle \rangle) &\rightarrow al: \langle x, tr: (al: \langle y, z \rangle) \rangle. \\
 null: \langle \rangle &\rightarrow T. \\
 null: a &\rightarrow F \quad (\text{for any atom } a). \\
 null: \langle x_1, \dots, x_n \rangle &\rightarrow F \quad (n \geq 1). \\
 null: (al: \langle x, y \rangle) &\rightarrow F. \\
 null: (ar: \langle x, y \rangle) &\rightarrow F. \\
 (cons: \langle f_1, \dots, f_n \rangle):x &\rightarrow \langle f_1:x, \dots, f_n:x \rangle. \\
 cons: (al: \langle f, g \rangle):x &\rightarrow al: \langle f:x, cons:g:x \rangle. \\
 cons: (ar: \langle f, g \rangle):x &\rightarrow ar: \langle cons:f:x, g:x \rangle. \\
 cond: \langle T, x, y \rangle &\rightarrow x. \\
 cond: \langle F, x, y \rangle &\rightarrow y. \\
 (comp: \langle f, g \rangle):x &\rightarrow f:(g:x). \\
 apply: \langle f, x \rangle &\rightarrow f:x. \\
 (K:x):y &\rightarrow x. \\
 id: x &\rightarrow x.
 \end{aligned}$$

Figure 1: The FP rewrite rules.

form. A language is *weakly normalizable* (respectively, *strongly normalizable*) iff every expression in that language is weakly normalizable (respectively, strongly normalizable). $\rightarrow^*$ is the reflexive, transitive closure of $\rightarrow$.

Let $\text{FP}\{-\text{apply}\}$ (resp. $\text{FP}\{-\text{cons}\}$, $\text{FP}\{-\text{comp}\}$) consist of all FP expressions which do not mention the primitive function *apply* (resp. *cons*, *comp*).

Theorem 2.2. *FP{-apply}, FP{-cons}, and FP{-comp} are all strongly normalizable.*

Proof. For the case $\text{FP}\{-\text{apply}\}$, we assign numbers to all the FP expressions as follows:

$N(a) = 1$ if a is an atom or primitive function,

$N(\langle x_1, \dots, x_n \rangle) = N(x_1) + \dots + N(x_n) + 1$,

$N(f:g) = 2^{N(f)} \times N(g)$.

Whenever we apply a rewrite rule (except for the *apply* rule), the number of the expression goes down. We prove this for the *comp* rule:

$$\begin{aligned} N((\text{comp}: \langle x, y \rangle):z) &= 2^{2^1 \times (N(x) + N(y) + 1)} \times N(z) \\ &> 2^{N(x)} \times 2^{N(y)} \times N(z) \\ &= N(x:(y:z)). \end{aligned}$$

All the other rewrite rules (of $\text{FP}\{-\text{apply}\}$) are just as easy.

For the case $\text{FP}\{-\text{cons}\}$, we assign numbers to FP expressions in a slightly different fashion. Note that all the rules (other than the ones for *cons*) have a right-hand side whose length is no longer than the left-hand side. Indeed, all the rules except $\text{tl}:(\text{ar}: \langle \text{ar}: \langle x, y \rangle, z \rangle) \rightarrow \text{ar}: \langle \text{tl}:(\text{ar}: \langle x, y \rangle), z \rangle$ and the corresponding rule for *tr* decrease length. Thus, for any FP expression x , we define

$$M(x) = \langle |x|, N(x) \rangle,$$

where $|x|$ is the length of x viewed as a string of symbols. If we now define the obvious lexicographic order on pairs of integers ($\langle m, n \rangle < \langle m', n' \rangle$ iff $m < m'$ or $m = m'$ and $n < n'$), then it is an easy matter to check that for every rewrite rule $x \rightarrow y$ other than those involving *cons*, we have $M(x) > M(y)$. Thus $\text{FP}\{-\text{cons}\}$ is strongly normalizable.

For $\text{FP}\{-\text{comp}\}$, we need yet another way of assigning numbers to expressions. The following one works:

$L(a) = 1$ for any atom or primitive function a other than *null*, *K*, or *id*,

$$L(\text{null}) = L(K) = L(\text{id}) = 2,$$

$$L(\langle x_1, \dots, x_n \rangle) = 1 + L(x_1) + \dots + L(x_n),$$

$$L(f:g) = \begin{cases} L(f) \times L(g)^{L(g)} & \text{if } f \neq \text{al, ar} \\ L(g) & \text{otherwise.} \end{cases}$$

We again leave it to the reader to check that for any rewrite rule $x \rightarrow y$ other than that for *comp*, we have $L(x) > L(y)$. Thus $\text{FP}\{-\text{comp}\}$ is strongly normalizable. $\square$

This result is essentially optimal. Let $x = \text{comp}: \langle \text{apply}, \text{cons}: \langle \text{id}, \text{id} \rangle \rangle$. Then $x:x$ is not weakly normalizable; we leave it to the reader to check that for all z , $x:z \rightarrow^* z:z$. (We can also use *fix* to construct an example of an FP expression that is not weakly normalizable, namely $\text{fix}:\text{id}$.) This indicates that somehow it is the *interaction* of *apply*, *cons*, and *comp*, rather than any of them individually, that makes FP so complicated.

We now turn to defining the semantics for FP. The intuition behind the domain we present is relatively straightforward. As is the case for most semantic domains for combinatory calculi, ours is a cpo (complete partial order) (cf. [St77]), where syntactic expressions that look like sequences are interpreted as sequences, and functions are approximated by sets of input-output pairs, something along the lines of Engeler's "arrow" model [En71]. There are two related difficulties which must be dealt with. The first is the problem of expressing infinite sequences. Let f be the expression $\text{fix}:(\lambda g.\text{al} \circ [\text{id}, g])$. (Note we are using $\lambda x.y$ to denote the FP expression that, when applied to z , reduces to $y[z/x]$, the result of replacing all occurrences of x in the expression y by z . By results of [Wi81], an FP expression with these properties is guaranteed to exist.) Recall *fix* has the property that $\text{fix}:f \rightarrow^* f:(\text{fix}:f)$. Using this fact, it is not hard to show that $f:x \rightarrow^* \text{al}: \langle x, f:x \rangle$. This suggests that the meaning of $f:x$ should be an infinite sequence of all x 's where x is the meaning of x . Thus, our semantics must be able to represent such sequences. We also must give meaning to $\text{al}: \langle y, z \rangle$ when the meaning of z is $\perp$. We handle both of these problems by including sequences of the form $\langle t_1, \dots, t_n, \omega \rangle$ in the domain where, intuitively, ω is a place-holder for some (as yet unknown)

sequence of items. Roughly speaking, we will take the meaning of the expression $f:x$ defined above to be the "limit" or sup of the sequence $\langle x, \omega \rangle, \langle x, x, \omega \rangle, \dots$, where x denotes the meaning of x . Similarly, if the meaning of z is $\perp$ and the meaning of y is y , then we will take the meaning of $al: \langle y, z \rangle$ to be $\langle y, \omega \rangle$. Note that ω acts quite differently from $\perp$; ω stands for a list of items yet to be computed, while $\perp$ stands for a single item yet to be computed. Thus, as we shall see, $\langle a, \omega \rangle \leq \langle a, a, a \rangle$, while $\langle a, \perp \rangle$ and $\langle a, a, a \rangle$ are incomparable.

Definition 2.3. We inductively define a sequence of partially ordered domains $(D_0, \leq_0), (D_1, \leq_1), \dots$ so that $D_n \subseteq D_{n+1}$ and $\leq_n$ is the restriction of $\leq_{n+1}$ to D_n . The elements of D_n , $n = 0, 1, \dots$ are called *items*. D_0 consists of $\perp$ and for each atom $a \in \mathcal{A}$, a corresponding *atomic item* a . (Note that we have adopted the notational convention of using boldface for syntactic entities and italics for their semantic counterparts.) For $x, y \in D_0$, we define $x \leq_0 y$ iff $x = \perp$ or $x = y$.

Suppose we have defined $(D_n, \leq_n)$. Before we can define D_{n+1} , we will need the following:

Definition 2.4. A subset U of D_n is said to be *closed* with respect to D_n if, whenever a pair of elements $x, y \in U$ has an upper bound $z \in D_n$ (i.e. $x \leq_n z, y \leq_n z$), then the pair is also bounded by some $w \in U$ such that $w \leq_n z$.

The following lemma about closed sets will be useful in what follows:

Lemma 2.5. Every subset U of D has a unique smallest closed subset, denoted $Cl(U)$, containing it. If U is finite then so is $Cl(U)$.

We call $Cl(U)$ the *closure* of U .

Let D_{n+1} consist of all the items in D_n together with all items of the form:

1. $\langle x_1, \dots, x_k \rangle$, where $\{x_1, \dots, x_k\} \subseteq D_n \cup \{\omega\}$ and at most one $x_i = \omega$ (where ω is a distinct symbol not equal to any item); if some $x_i = \omega$ then we must have $k \geq 2$. Such an item is called a *sequence item*.
2. $\{x_1 \mapsto y_1, \dots, x_k \mapsto y_k\}$, where $\{x_1, \dots, x_k, y_1, \dots, y_k\} \subseteq D_n$ and:
 - a. $\{x_1, \dots, x_k\}$ is closed with respect to D_n ,

- b. if $x_i \leq_n x_j$ then $y_i \leq_n y_j$,
- c. $x_1, \dots, x_k$ are all distinct.

Such an item is called a *function item*; its elements are called *input-output pairs* and $\{x_1, \dots, x_k\}$ is called its *domain*. The domain of a function item u is denoted $dom(u)$. We will explain below the intuition behind the conditions we have imposed on function items.

Note that in clause 1 above we have explicitly disallowed $\langle \omega \rangle$ as an item; we will identify $\langle \omega \rangle$ with $\perp$. We could have distinguished them, viewing $\langle \omega \rangle$ as the unique minimal sequence item, but this would only have added extra details to our proofs. The reader will also notice that at the "top level" of a sequence item we allow only one ω . Again this was done mainly to simplify the proofs. Since, as will be seen when we give the meanings of the primitive functions, we have chosen to allow "access" to a sequence only from the left or right hand sides (via first and last), we can safely identify such sequences as $\langle x, \omega, y \rangle$ and $\langle x, \omega, z, \omega, y \rangle$ (since the z between the ω 's in the latter sequence can never be accessed).

We define $\leq_{n+1}$ to be the least reflexive, transitive relation on D_{n+1} satisfying:

1. $\perp \leq_{n+1} x$ for any item $x \in D_{n+1}$,
2. $\langle x_1, \dots, x_l \rangle \leq_{n+1} \langle y_1, \dots, y_l \rangle$ if $x_j \leq_n y_j$ or $x_j = \omega = y_j$ for $j = 1, \dots, l$,
3. $\langle x_1, \dots, x_l, \omega, y_1, \dots, y_m \rangle \leq_{n+1} \langle x_1, \dots, x_l, z_1, \dots, z_k, y_1, \dots, y_m \rangle$ for $k \geq 0$,
4. if u and v are function items, then $u \leq_{n+1} v$ iff for every input-output pair $x \mapsto y' \in u$, there exists a pair $x \mapsto y \in v$ such that $y' \leq_n y$.

Notice that function items, sequence items, and atomic items are pairwise incomparable. Restriction (c) on function items guarantees that distinct function items are not equivalent; i.e. if x and y are distinct function items, then it is not the case that both $x \leq_{n+1} y$ and $y \leq_{n+1} x$ can hold.

Definition 2.6. We define the partially ordered domain $(D, \leq)$ to be the limit of $(D_n, \leq_n)$; i.e., $D = \bigcup_n D_n$ and $\leq = \bigcup_n \leq_n$ (viewing $\leq_n$ as a relation on D). We define the operation $:$ (*function application*) on D via

$$x:y = \begin{cases} z & \text{if } x = \{u_1 \mapsto v_1, \dots, u_n \mapsto v_n\}, \text{ and for some } j, \\ & u_j = \max\{u_i \mid u_i \leq y, i = 1, \dots, n\} \text{ and } z = v_j, \\ \perp & \text{otherwise.} \end{cases}$$

Note that $x:y = \perp$ unless x is a function item and, for some input-output pair $u \mapsto v \in x$, we have $u \leq y$. It is easy to see from the definition of $\leq$ on function items that if $x' \leq x$, then $x':y \leq x:y$. What we now want to show is that functions are monotonic; namely

Proposition 2.7. *If $y' \leq y$ then $x:y' \leq x:y$.*

Proof. We need the following lemma, whose proof can be found in [HW³84]. A set U is said to be *directed* if any pair of elements in U has an upper bound in U .

Lemma 2.8. *If U is a finite subset of D_n which is closed (with respect to D_n) and $z \in D$, then $\{u \in U \mid u \leq z\}$ is directed.*

Proposition 2.7 now follows immediately from Lemma 2.8 and restrictions (a) and (b) on function items. For suppose $y' \leq y$. We want to show that $x:y' \leq x:y$. We can assume without loss of generality that x is a function item of the form $\{u_1 \mapsto v_1, \dots, u_n \mapsto v_n\}$, and for some j , we have $u_j = \max\{u \in U \mid u \leq y'\}$, where $U = \{u_1, \dots, u_n\}$, and $x:y' = v_j$. Since $y' \leq y$, $\{u \in U \mid u \leq y\}$ is a superset of $\{u \in U \mid u \leq y'\}$, and in particular contains u_j . Since U is closed (by restriction (a)), it follows by Lemma 2.8 that $\{u \in U \mid u \leq y\}$ is directed. Since this set is finite, it must have a maximum element, say u_k . Thus $x:y = v_k$. Since $u_j \leq u_k$ by construction, by restriction (b) we have $v_j \leq v_k$. Thus $x:y' \leq x:y$. $\square$

The observant reader may have noticed that the ordering we have defined on function items is *not* the most natural “approximation” ordering for functions. For example, suppose $x = \{u \mapsto v\}$ and $x' = \{u' \mapsto v'\}$ are function items with $u < u'$ and $v' \leq v$. It is easy to check that for all items y , we have $x':y \leq x:y$. However, x and x' are incomparable with respect to $\leq$. Indeed, it is easy to modify this example to construct two function items which are incomparable with respect to $\leq$, but act the same as functions. We could have easily modified our definition of $\leq$ so that these “anomalies” would not arise. However, our definition leads to an easier

proof of the completeness result of the next section; for example, it allows us to prove the following lemma:

Lemma 2.9. *For all $x \in D$, $\{y \mid y \leq x\}$ is finite.*

The domain D constructed above does not suffice as a semantic domain for FP because it is not *complete*: there are directed sets in D (such as $\{ \langle a, \omega \rangle, \langle a, a, \omega \rangle, \langle a, a, a, \omega \rangle, \dots \}$) which have no least upper bound in D . Let D^* be the *completion* of D ; i.e., the smallest complete set containing D . There are standard techniques for constructing such completions (cf. [Bi67]), so we only sketch a few of the details here. Roughly speaking, we can view the elements of D^* as directed sets of elements of D , in the same way that we can view a real number as a Cauchy sequence of rationals. We can extend the ordering $\leq$ to D^* by defining $\{x_1, x_2, \dots\} \leq \{y_1, y_2, \dots\}$ iff for all i there exists a j such that $x_i \leq y_j$. More precisely, just as a real can be viewed as an equivalence class of Cauchy sequences (since more than one Cauchy sequence can converge to a given real number), so an element of D^* can be viewed as an equivalence class of directed sets, where two directed sets U and V are equivalent if $U \leq V$ and $V \leq U$.

We can view D as a subset of D^* by identifying the item $x \in D$ with the equivalence class of the directed set $\{x\}$. If $x_1, \dots, x_n$ are elements of D^* , where x_i is the equivalence class of the directed set X_i , then we can view $\langle x_1, \dots, x_n \rangle$ as an element of D by identifying it with the equivalence class of the directed set $\{ \langle y_1, \dots, y_n \rangle \mid y_i \in X_i \}$. We leave it to the reader to check that this is well defined (i.e., independent of the representative of the equivalence class of x_i that is chosen). Thus we can view D^* as closed under the formation of finite sequences. Finally, we extend $:$ to D^* in the obvious way: if $x, y \in D^*$, where x is the equivalence class of the directed set X and y is the equivalence class of the directed set Y , then $x:y$ is the equivalence class of $\{x':y' \mid x' \in X, y' \in Y\}$. Again, we leave it to the reader to check that this is well defined and satisfies Proposition 2.7.

We are now ready to give a meaning in the semantic domain D^* to each syntactic expression. We define such a meaning function μ from syntactic

expressions to D^* as follows:

$$\mu(a) = a \text{ for any atom } a,$$

$$\mu(\langle x_1, \dots, x_n \rangle) = \langle \mu(x_1), \dots, \mu(x_n) \rangle,$$

$$\mu(x:y) = \mu(x):\mu(y).$$

All that remains is to define $\mu(f)$ for each primitive function f . Our first step in doing so is to define auxiliary sets S_f for each primitive function f . Intuitively, S_f consists of *all* the input-output pairs that make up the meaning of f . Since S_f is infinite, $\mu(f)$ is actually (the equivalence class of) the directed set of all function items which are approximations to S_f . More formally, let

$$S_{al} = \{x \mapsto y \mid x = \langle u, \langle v_1, \dots, v_n \rangle \rangle, y = \langle u, v_1, \dots, v_n \rangle, \\ u \neq \omega \text{ or } x = \langle u, \perp \rangle, y = \langle u, \omega \rangle, u \neq \omega\},$$

$$S_{ar} = \{x \mapsto y \mid x = \langle \langle u_1, \dots, u_n \rangle, v \rangle, y = \langle u_1, \dots, u_n, v \rangle, \\ v \neq \omega \text{ or } x = \langle \perp, v \rangle, y = \langle \omega, v \rangle, v \neq \omega\},$$

$$S_{first} = \{x \mapsto y \mid x = \langle u_1, \dots, u_n \rangle, n \geq 1, u_1 \neq \omega, y = u_1\},$$

$$S_{last} = \{x \mapsto y \mid x = \langle u_1, \dots, u_n \rangle, n \geq 1, u_n \neq \omega, y = u_n\},$$

$$S_{tl} = \{x \mapsto y \mid x = \langle u_1, \dots, u_n \rangle, n \geq 1, u_1 \neq \omega, \\ y = \langle u_2, \dots, u_n \rangle \text{ or } \\ n \geq 3, u_1 = \omega, y = \langle \omega, u_3, \dots, u_n \rangle\},$$

$$S_{tr} = \{x \mapsto y \mid x = \langle u_1, \dots, u_n \rangle, n \geq 1, u_n \neq \omega, \\ y = \langle u_1, \dots, u_{n-1} \rangle \text{ or } \\ n \geq 3, u_n = \omega, y = \langle u_1, \dots, u_{n-2}, \omega \rangle\},$$

$$S_{null} = \{x \mapsto y \mid x = \langle \rangle, y = T, \text{ or } x_i = a, a \text{ an atomic} \\ \text{item, } y = F, \text{ or } x = \langle u_1, \dots, u_n \rangle, y = F\},$$

$$S_{cons} = \{x \mapsto y \mid x = \langle f_1, \dots, f_n \rangle, y = \{u \mapsto \langle g_1, \dots, g_n \rangle \mid \\ u \in Cl(dom(f_1) \cup \dots \cup dom(f_n)), g_i = \omega \text{ if } \\ f_i = \omega, \text{ otherwise } g_i = f_i \cdot u, i = 1, \dots, n\},$$

$$S_{cond} = \{x \mapsto y \mid x = \langle T, u, v \rangle, y = u, \text{ or } \\ x = \langle F, u, v \rangle, y = v\},$$

$$S_{comp} = \{x \mapsto y \mid x = \langle f, g \rangle, f, g \neq \omega, \\ y = \{u \mapsto f \cdot v \mid u \mapsto v \in g\}\},$$

$$S_{apply} = \{x \mapsto y \mid x = \langle u, v \rangle, u, v \neq \omega, y = u \cdot v\},$$

$$S_K = \{x \mapsto \perp \mid x \in D\},$$

$$S_{id} = \{x \mapsto x \mid x \in D\}.$$

We now define $\mu(f)$ to be (the equivalence class of) the directed set consisting of all finite subsets of S_f whose domain is closed. Two things remain to be checked in order to show that this is well-defined. First, given a finite subset of S_f of the form $\{x_1 \mapsto y_1, \dots, x_k \mapsto y_k\}$ with a closed domain, we must check that it is indeed a function item; i.e., it

satisfies the three restrictions on function items described above. This is straightforward. That the domain is closed is true by definition, while the fact that $x_i \leq x_j$ implies $y_i \leq y_j$ and the fact that the x_i 's are all distinct can easily be checked for each of the functions. Secondly, we must show that $\mu(f)$ as defined is indeed an directed set. This follows from the following observation: for each primitive function f , if x and y are two function items in $\mu(f)$ with domains U_x and U_y respectively, then there is a function item z in $\mu(f)$ whose domain is the closure of $U_x \cup U_y$. From the form of the definition of $\mu(f)$, it now follows that $u \leq z$ and $v \leq z$ (and in fact $u \leq z, v \leq z$). Thus $\mu(f)$ is indeed a directed set.

A few remarks are in order regarding the semantics we have given to the primitive functions. Most of the definitions seem clearcut, given the intended meanings of the primitive functions. However, note that $\mu(al): \langle u, \perp \rangle = \langle u, \omega \rangle$ for any item u . This choice was necessary in order to ensure the soundness of the rule $first:(al: \langle x, y \rangle) \rightarrow x$ in the case that $\mu(y) = \perp$. Similar remarks hold for the definition of ar . For tl , note we have $\mu(tl): \langle \omega, u_2, \dots, u_n \rangle = \langle \omega, u_3, \dots, u_n \rangle$. This choice was made to ensure the soundness of the rule $tl:(ar: \langle ar: \langle x, y \rangle, z \rangle) \rightarrow ar: \langle tl:(ar: \langle x, y \rangle), z \rangle$ in the case that $\mu(x) = \perp$. Similar considerations apply to the definition of tr .

The reader who is familiar with such semantic domains as these may have noted that we could have given a somewhat more elegant definition of several of the functions. Consider for example $first$. Since $first$ returns the first element of a sequence and ignores the rest of the elements, we could have defined $S_{first}' = \{\langle x, \omega \rangle \mapsto x \mid x \in D\}$, and then defined $\mu(first')$ in an analogous fashion to $\mu(first)$ as (the equivalence class of) all the finite subsets of S_{first}' with closed domains. It is easy to check that $\mu(first)$ and $\mu(first')$ act the same as functions (although they are not the same element of D^* due to our "anomalous" ordering on function items). Similar simplifications are possible for $last$, tl , tr , $null$, and $cond$. We have chosen to use the definitions given above since they seem more intuitive and since their more compact counterparts do not make any of the following proofs easier.

How do we know that in some sense we have chosen reasonable definitions of the function items? It is straightforward to check that, for example, $(\mu(\text{comp}): \langle f, g \rangle).x = f:(g.x)$ for all $f, g, x \in D^*$. Similarly, $\mu(\text{apply}): \langle f, x \rangle = f:x$ for all $f, x \in D^*$. These observations do lend some weight to the claim that the semantics has captured the intended meaning of **comp** and **apply**. We can also check that analogous statements hold for all the primitive functions. Indeed, we can now state and prove the theorem that the rewrite rules are sound with respect to this semantics.

Theorem 2.10. *For every reduction rule $x \rightarrow y$, we have $\mu(x) = \mu(y)$.*

Proof. By a case-by-case analysis of the reduction rules. Details can be found in [HW³84]. $\square$

3. COMPLETENESS OF THE REDUCTION RULES

In this section we show that, in the precise sense discussed in the introduction, there are “enough” reduction rules.

Definition 3.1. Let the *observable expressions* be those that can be formed starting with atoms and closing off under the sequence constructor. More formally, the set of observable expressions is the least set $\mathcal{O}$ such that:

1. $\mathcal{A} \subseteq \mathcal{O}$,
2. if $x_1, \dots, x_n \in \mathcal{O}$, then $\langle x_1, \dots, x_n \rangle \in \mathcal{O}$.

To each observable expression there corresponds an *observable item* which is obtained by replacing boldface, $\langle$, and $\rangle$ wherever they occur in the observable expression by italics, $\langle$, and $\rangle$ respectively. Note that neither $\perp$ nor ω appears in any sequence item which is observable. Given an observable item x , let x^c be the observable expression to which it corresponds.

We are now ready to state the completeness theorem.

Theorem 3.2. *If x is an FP expression and y is an observable item such that $\mu(x) = y$ (or equivalently, $\mu(x) = \mu(y^c)$) then $x \rightarrow^* y^c$. Thus if z is an observable expression and $\mu(x) = \mu(z)$, then $x \rightarrow^* z$.*

Note that Theorem 3.2 implies that in a precise sense, (sound) rewrite rules such as $\text{first}:(\langle \text{cons}:x \rangle.y) \rightarrow (\text{first}:x).y$ or $\text{tl}:(\langle \text{cons}:x \rangle.y) \rightarrow (\text{cons}:(\text{tl}:x)).y$ are not needed in our system.

The proof of Theorem 3.2 is nontrivial, and will occupy us for the rest of this section. We sketch the main ideas here; further details can be found in [HW³84]. The key idea is to construct a variant of FP called FP' on which we impose a notion of *type*. The proof then proceeds in four steps. The first, quite simple, is to show that for any FP expression x such that $\mu(x) = y$ for some observable item y , there is a corresponding FP' expression x' such that $\mu(x') = y$. Secondly, we show that FP' reduction sequences correspond in a precise sense to FP reduction sequences. Next, by extending techniques of [FLO83], we show that FP' is strongly normalizable. Finally, we show that an FP' expression in normal form whose meaning is the same as that of an observable expression corresponds to an FP expression in normal form. The result follows from this last observation.

We now define FP' , the semantic function μ' which maps FP' expressions to elements of D , and a “pull-back” function ρ that associates with each FP' expression a corresponding FP expression. FP' expressions essentially look like FP expressions, except that primitive functions are subscripted by elements of D ; ρ removes these subscripts. For every FP' expression x , $\mu'(x)$ (which is in D) can be viewed as a finite approximation to $\mu(\rho(x))$ (which in general will be in D^*). For technical reasons we also allow expressions of the form $\perp(x)$ in FP' , where x is an FP expression, and elements of the form $\omega(z_1, \dots, z_k)$ to appear in FP' sequences, where $z_1, \dots, z_k$ are FP expressions. More formally we have:

Definition 3.3.

1. Every atom in $\mathcal{A}$ is an FP' expression; $\mu'(a) = a$ and $\rho(a) = a$.
2. If $f \in \mathcal{F}$, then f_i is an FP' expression for any item $i \leq \mu(f)$; $\mu'(f_i) = i$ and $\rho(f_i) = f$.
3. If x is an FP expression, then $\perp(x)$ is an FP' expression; $\mu(\perp(x)) = \perp$ and $\rho(\perp(x)) = x$.
4. If $x_1, \dots, x_n$ are FP' expressions, then $\langle x_1, \dots, x_n \rangle$ is an FP' expression;

$\mu'(\langle x_1, \dots, x_n \rangle) = \langle \mu'(x_1), \dots, \mu'(x_n) \rangle$ and

$\rho(\langle x_1, \dots, x_n \rangle) = \langle \rho(x_1), \dots, \rho(x_n) \rangle$.

5. If $x_1, \dots, x_n$, $n \geq 1$, are FP' expressions and $z_1, \dots, z_k$, $k \geq 0$, are FP expressions, then $\langle x_1, \dots, x_i, \omega(z_1, \dots, z_k), x_{i+1}, \dots, x_n \rangle$ is an FP' expression;

$\mu'(\langle x_1, \dots, x_i, \omega(z_1, \dots, z_k), x_{i+1}, \dots, x_n \rangle) = \langle \mu'(x_1), \dots, \mu'(x_i), \omega, \mu'(x_{i+1}), \dots, \mu'(x_n) \rangle$ and $\rho(\langle x_1, \dots, x_i, \omega(z_1, \dots, z_k), x_{i+1}, \dots, x_n \rangle) = \langle \rho(x_1), \dots, \rho(x_i), \rho(x_{i+1}), \dots, \rho(x_n) \rangle$.

6. If x and y are FP' expressions, then $x:y$ is an FP' expression; $\mu'(x:y) = \mu'(x):\mu'(y)$ and $\rho(x:y) = \rho(x):\rho(y)$.

Note that if x is an observable FP expression, then x is also an FP' expression, $\mu(x) = \mu'(x)$, and $\rho(x) = x$. In general we can show (by an easy induction on structure) that

Proposition 3.4. For any FP' expression x' , we have $\mu'(x') \leq \mu(\rho(x'))$. Conversely, if x is an FP expression, $d \leq \mu(x)$, and $d \in D$, then there exists an FP' expression x' such that $\mu'(x') = d$ and $\rho(x') = x$.

Since the meaning of any observable expression is in D , we immediately get:

Corollary 3.5. For any observable FP expression x , there is an FP' expression x' such that $\mu'(x') = \mu(x)$ and $\rho(x') = x$.

Besides the semantic order $\leq$ on D defined in the previous section, it will also be useful to have a more syntactically based ordering $\leq_e$ on FP' expressions. Intuitively, $x \leq_e y$ means that we can obtain x from y by "lowering" the subscripts on y (i.e. replacing a subscript by one of lower meaning).

Definition 3.6. $\leq_e$ is the least reflexive, transitive relation satisfying:

1. $f_i \leq_e f_j$ if f is a primitive FP function and $i \leq j$ (and $i, j \leq \mu(f)$).
2. $x_1:x_2 \leq_e y_1:y_2$ if $x_1 \leq_e y_1$ and $x_2 \leq_e y_2$.
3. $\langle x_1, \dots, x_n \rangle \leq_e \langle y_1, \dots, y_n \rangle$ if $x_i \leq_e y_i$ or $x_i = y_i = \omega(s)$ for some list s .
4. $\langle x_1, \dots, x_i, \omega(z_1, \dots, z_m), x_{i+1}, \dots, x_n \rangle \leq_e \langle x_1, \dots, x_i, z'_1, \dots, z'_m, x_{i+1}, \dots, x_n \rangle$ if $z_j = \rho(z'_j)$ for $j = 1, \dots, m$.
5. $\langle x_1, \dots, x_j, \omega(y_1, \dots, y_k, z_1, \dots, z_l, y_{k+1}, \dots, y_m), x_{j+1}, \dots, x_n \rangle$

$\leq_e \langle x_1, \dots, x_j, y'_1, \dots, y'_k, \omega(z_1, \dots, z_l), y'_{k+1}, \dots, y'_m, x_{j+1}, \dots, x_n \rangle$

if $y_i = \rho(y'_i)$ for $i = 1, \dots, m$.

6. $\perp(x) \leq_e x'$ if $x = \rho(x')$.
7. $f_i:x \leq_e f_j:x$ if there exists items u and v such that $i = \{\mu'(x) \rightarrow v\}$ and $j = \{u \rightarrow v\}$ where $u \leq \mu'(x)$, f is a primitive FP function (and $i, j \leq \mu(f)$).

We say $x <_e y$ if $x \leq_e y$ and $x \neq y$.

Example 3.7. Let $i = \{\langle a, \perp, \omega \rangle \rightarrow \langle a, \omega \rangle\}$, $j = \{\langle a, b, \perp \rangle \rightarrow \langle a, b \rangle\}$, $k = \{\langle a, b, c \rangle \rightarrow \langle a, \omega \rangle\}$, and $l = j \cup k$. Then

1. tr_i, tr_j, tr_k , and tr_l are all minimal (as is f_m for any primitive function f and $m \leq \mu(f)$),
2. $tr_l: \langle a, b, \perp(c) \rangle$ is not minimal, while $tr_j: \langle a, b, \perp(c) \rangle$ is,
3. $tr_i: \langle a, b, c \rangle$ is not minimal, while $tr_l: \langle a, \perp(b), \omega(c) \rangle$ is,
4. $tr_k: \langle a, b, c \rangle$ is not minimal since, by Clause 7, $tr_i: \langle a, b, c \rangle <_e tr_k: \langle a, b, c \rangle$.

Proposition 3.8. If $x \leq_e y$, then $\rho(x) = \rho(y)$ and $\mu'(x) \leq \mu'(y)$.

Proof. Follows from the definition of $\leq_e$. $\square$

Definition 3.9. An FP' expression x is *minimal* iff $y <_e x$ implies that $\mu'(y) < \mu'(x)$.

Reduction rules in FP' are essentially the same as those in FP (modulo the subscripts) except for two important differences: we will not allow FP -style rewriting to take place for any expressions whose meaning is $\perp$, and we only allow FP -style rules to be applied to *minimal* expressions. Thus we add another rule to allow us to reduce to a minimal expression so we can apply an FP -style rule.

Definition 3.10. We say $x \rightarrow y$ for FP' expressions x and y if $\mu'(x) = \mu'(y)$ and one of the following two conditions is met:

1. $\rho(x) \rightarrow \rho(y)$ is an FP rewrite rule, $\mu'(x) \neq \perp$, x is a minimal FP' expression, and every instantiation of a schema variable in y not of the form $\omega(s)$ for some list s is equal to the instantiation of the corresponding schema variable in x (this is called an *FP-style* rewrite rule).
2. $y <_e x$ (this is called a *lowering* rewrite rule).

Example 3.11. Let $i = \{\langle a, \perp, \omega \rangle \rightarrow \langle a, \omega \rangle\}$, $j = \{\langle \perp, \perp \rangle \rightarrow \langle \perp, \omega \rangle\}$, $k = \{\langle a, \langle \perp, \omega \rangle \rangle \rightarrow \langle a, \perp, \omega \rangle\}$. Then the following are FP -style reduction rules:

1. $tr_i: \langle a, \perp(b), \omega(c) \rangle \rightarrow \langle a, \omega(b) \rangle,$
2. $tr_i: \langle a, \perp(b), \omega() \rangle \rightarrow \langle a, \omega() \rangle,$
3. $al_j: \langle \perp(b), \perp(\langle c \rangle) \rangle \rightarrow \langle \perp(b), \omega(c) \rangle,$
4. $tr_i: (al_k: \langle a, al_j: \langle \perp(b), \perp(\langle c \rangle) \rangle \rangle) \rightarrow$
 $al_k': \langle a, tr_i': (al_j': \langle \perp(b), \perp(\langle c \rangle) \rangle) \rangle,$ where $i',$
 $j',$ and k' must be chosen so that
 $\mu'(al_k': \langle a, tr_i': (al_j': \langle \perp(b), \perp(\langle c \rangle) \rangle) \rangle) \rightarrow$
 $= \langle a, \omega \rangle.$

Suppose we have an FP' expression x such that $\mu'(x) \neq \perp$ and $\rho(x)$ is an instance of an FP rewrite rule. It is easy to see that by using the lowering rule we can reduce x to an expression to which an FP -style rewrite rule applies. (Note that from the definition of $\leq_e$ and Lemma 2.9 we can show that for all FP' expressions x there is a minimal FP' expression x' with $x' \leq_e x$.) The following straightforward lemma assures us that the FP' rewrite rules do in fact simulate the FP rewrite rules.

Proposition 3.12.

1. If x' is an FP' expression such that $\mu'(x') \neq \perp$ and y an FP expression such that $\rho(x') \rightarrow y$ is an instance of an FP rewrite rule, then $x' \rightarrow^* y'$ for some FP' expression y' with $\rho(y') = y$.
2. If x' and y' are FP' expressions such that $x' \rightarrow^* y'$, then $\rho(x') \rightarrow^* \rho(y')$.

We are now ready to define our notion of type. We first give the abstract properties that we want this notion to satisfy, and then prove (in [HW³84]) there is actually a function that satisfies these properties.

Definition 3.13. We say that τ is a type function iff it is a function from FP' expressions into a well-founded set (ordered by $\leq_t$) having the following properties:

1. If $x \rightarrow^* y$, then $\tau(y) \leq_t \tau(x)$,
2. $\tau(y_i) <_t \tau(\langle y_1, \dots, y_n \rangle)$ for $i = 1, \dots, n$,
3. If x is a minimal FP' expression of the form $al_i: \langle y, z \rangle$ or $ar_j: \langle y, z \rangle$, then $\tau(y) <_t \tau(x)$ and $\tau(z) <_t \tau(x)$,
4. If $\mu'(x)$ is a function item and $\mu'(y) \rightarrow \mu'(z) \in \mu'(x)$, then $\tau(y) <_t \tau(x)$ and $\tau(z) <_t \tau(x)$.

Proposition 3.14. *There is a type function for FP' .*

We take τ to be a fixed type function for the rest of the paper.

When one thinks of type functions (in typed λ -calculus, for example) some of the properties that one usually wants are that $x:y$ is well formed only if the type of x is greater than the type of y , in which case the type of x is also greater than the type of $x:y$. The former property does not hold for FP' , since $x:y$ is a well-formed FP' expression whenever x and y are, but the reduction rules essentially force it to hold. More precisely, we have:

Lemma 3.15.

1. We can only apply an FP -style reduction rule to $x:y$ if $\tau(y) <_t \tau(x)$.
2. For all FP' expressions x and y where $\mu'(x)$ is a function item, $\tau(x:y) <_t \tau(x)$.

We would like to prove FP' is strongly normalizable. One technique that suggests itself is induction on types. (Indeed, the whole point of defining a type function on expressions is to be able to do proofs by induction on types.) However, it seems that strong normalizability by itself is not a sufficiently strong inductive hypothesis. Following the lines of [FLO83], we remedy this problem by defining a notion of *stability*.

We first need

Definition 3.16. An FP' expression y is a *component* of the FP' expression x iff there is a minimal FP' expression z such that $x \rightarrow^* z$ and z has one the forms $al_i: \langle t, y \rangle$, $al_i: \langle y, t \rangle$, $ar_i: \langle y, t \rangle$, $ar_i: \langle t, y \rangle$, or $\langle \dots, y, \dots \rangle$.

We now define what it means for an FP' expression x to be stable by induction on $\tau(x)$.

Definition 3.17.

1. If $\tau(x)$ is minimal in the $\leq_t$ ordering, then x is stable iff x is strongly normalizable.
2. If $\tau(x)$ is not minimal, then x is stable iff it is strongly normalizable and satisfies the following properties:
 - a. If $\mu'(x)$ is a function item, then $x:y$ is stable for every stable y for which $\tau(y) <_t \tau(x)$,
 - b. If $\mu'(x)$ is a sequence item, then every component of x is stable.

Note that the concept of stability is well-defined since (by the properties of a type function) τ of a component is less than τ of the original expression and $\tau(x:y) < \tau(x)$ if $\mu'(x)$ is a function item.

Proposition 3.18. *Every FP' expression is stable.*

Proof. The proof depends on the following three claims, whose proof is in [HW³84]:

1. If $x_1, \dots, x_n$ are all stable, then so is $\langle x_1, \dots, x_n \rangle$.
2. If x and y are stable, then so is $x:y$.
3. Every primitive function symbol is stable.

Using the claims, we can show by an easy induction on structure that every FP' expression is stable. $\square$

Since every stable expression is by definition strongly normalizable, we immediately get:

Corollary 3.19. *FP' is strongly normalizable.*

The proof of the three claims in Proposition 3.18 is quite technical. To see the difficulty, suppose that, for example, we wish to prove that $\text{first}_i: \langle x_1, \dots, x_n \rangle$ is stable. This expression rewrites to x_1 , so it seems as if it should suffice to prove that x_1 is stable. But how do we know this suffices?

The following *Stability Criterion* (Theorem 3.21) gives this assurance, by providing a necessary and sufficient condition for proving that an expression is stable. We believe that this criterion may be of independent interest as a general technique for proving similar results about other languages.

Definition 3.20. An FP' expression x is a *subredex* if it unifies with a *nontrivial* (i.e. something other than a schema variable) proper subexpression of the left-hand side of an FP-style rewrite rule. An FP' expression x *filters through* a set of FP' expressions Y iff

1. $\forall y \in Y (x \rightarrow^* y)$
2. if $x \rightarrow^* z$ and z is a subredex, then $\exists y \in Y (y \rightarrow^* z)$

Thus, for example, if $\text{first}_i: \langle x_1, \dots, x_n \rangle$ is a minimal expression, then it filters through $\{x_1\}$.

Theorem 3.21. *Let x filter through the set Y . Then x is stable iff x is strongly normalizable and every element of Y is stable.*

We need one more proposition before proving our main result.

Proposition 3.22. *Let x' be an FP' expression in normal form, and let $x = \rho(x')$. Then:*

1. if $\mu'(x') = a$ for some atomic item a , then $x = a$,
2. if $\mu'(x') = \langle w_1, \dots, w_n \rangle$, then x is of the form $\langle y_1, \dots, y_n \rangle$, $\text{al}: \langle y, z \rangle$, or $\text{ar}: \langle y, z \rangle$; if $w_i \neq \omega$ for $i = 1, \dots, n$, then in fact x must be of the form $\langle y_1, \dots, y_n \rangle$, where $\mu(y_i) = w_i$,
3. if $\mu'(x')$ is a function item, then x is of the form f for some $f \in \mathcal{F}$, $K:y$, $\text{comp}:y$, or $\text{cons}:y$.

Corollary 3.23. *If x' is an FP' expression in normal form, and $\mu'(x') = y$, where y is an observable item, then $\rho(x') = y^c$.*

The proof of Theorem 3.2 is now relatively straightforward. Suppose x is an FP expression such that $\mu(x) = y$ for some observable item y . By Corollary 3.5, there is an FP' expression x' such that $\rho(x') = x$ and $\mu'(x') = y = \mu(x)$. By Corollary 3.19, there is some y' such that $x' \rightarrow^* y'$ and no further FP' reductions apply to y' . By Proposition 3.12, $x \rightarrow^* \rho(y')$, and by Corollary 3.23, $\rho(y') = y^c$ as desired. $\square$

We remark that we can squeeze more information out of our proof than is indicated by the statement of Theorem 3.2. For example, if $\mu(x) = \langle a, \omega \rangle$, then similar techniques to those used above can be used to show that $x \rightarrow^* \text{al}: \langle a, y \rangle$, where $\mu(y) = \perp$. Iterating this reasoning, if $\mu(x)$ is the directed set $\{\langle x_1, \omega \rangle, \langle x_1, x_2, \omega \rangle, \dots\}$, we can show that $x \rightarrow^* \text{al}: \langle x_1, \text{al}: \langle x_2, \dots \rangle \rangle$, where the approximation can be carried out to any desired depth. We can also get information out along the lines of part (3) of Proposition 3.22 if $\mu(x)$ is a directed set of function items. (See [HW³84] for further details.)

While Theorem 3.2 shows that there is always some strategy to reduce x to y^c if $\mu(x) = y$, where y is an observable item, it does not show that we can always find that strategy effectively. However, the proof of Theorem 3.2 actually shows that the following *breadth-first* strategy is guaranteed to work if anything does. We first number all the subexpressions of x which are redexes from left to right. We perform these reductions in order. It may be that by the time we get around to doing reduction 3, it may have "disappeared". For example, x might be of the form $K:y:z$, and redex 3 might be

a subexpression of z . Since $K:y:z \rightarrow y$, this redex no longer exists. If it does exist, we perform it. After all the numbered reductions have been performed, we repeat the process, numbering all the redexes in the resulting expression and performing them in order. This strategy guarantees that every redex that does not “disappear” will eventually be reduced.

Theorem 3.24. *If $\mu(x) = y$, where y is an observable item, then the breadth-first strategy described above will reduce x to y^c .*

4. CONCLUSIONS

We have answered the two questions posed in the introduction: we have given a denotational semantics for FP where distinct observable expressions get different meanings, shown that the rewrite rules are sound for that semantics, and shown that they are complete in the sense that whenever the meaning of an expression is an observable item, the rewrite rules allow us to reduce to the corresponding observable expression. While we have proved these results for FP, we believe that our techniques will apply to much more general languages, and we intend to try to use them to treat the general combinatory calculus. We also conjecture that this method will enable us to prove the strong normalizability of the second order λ -calculus without having to resort to the technique of “grounds” used in [FLO83].

As well as being applicable to other particular languages, we believe that this semantics-based approach can be generalized to show that a collection of rewrite rules is sound and complete with respect to an entire family of semantic domains, and we would like to explore this possibility.

While the language we have presented is universal, it is by no means sufficient as a useful programming language. There are many more functions to be added, some defined in terms of the primitives given and some as new primitives, and there are useful extensions that could be made to some of our functions. Although the addition of new primitives will necessarily make the proof longer (there being more cases to consider and rules to check), we believe that the structure of our proof is such that it will be relatively easy to make modifications to the language. In fact, in the development of this paper we have had occasion to make various perturbations in the definitions of the primitive functions, and we have found the proof of completeness (and in par-

ticular the stability criterion) to be remarkably insensitive to such modifications. Moreover, though it is computationally spartan, we believe that the language already includes instances of all the essential complexities. For example, it might be useful to extend the meaning of `comp` to allow the rule `comp:( $\lambda f.g$ ): $x \rightarrow f:(\text{comp}g:x)$` , but the complications that would result from such an extension have already been considered in our treatment of `cons`.

At the same time, while we don't expect that such additions will pose any significant mathematical difficulties for our proof technique, there is a great deal of interesting work to be done to make the language more useable computationally. We are presently engaged in developing an extension of this kernel language that will have the same richness of algebraic properties as the original language of [Ba78]. As we extend the language, we hope to be able to use the degree to which these mathematical considerations are complicated as a criterion for choosing among various alternative extensions.

5. ACKNOWLEDGEMENTS

We would like to thank John Backus, Ron Fagin, Albert Meyer, Mike O'Donnell, Moshe Vardi, and David Wise for their useful comments and criticisms.

6. REFERENCES

- [Ba78] Backus, J.W., “Can programming be liberated from the von Neumann style? A functional style and its algebra of programs”, *Commun. ACM* 21:8 (August, 1978), pp. 613-641.
- [Bi67] Birkhoff, G., *Lattice Theory*, American Math. Society Colloquium Publications, vol. 23, Third edition (1967).
- [BMS80] Burstall, R., D. MacQueen, and D. Sannella, “HOPE: An experimental applicative language”, *1980 LISP Conference*, Stanford, California (August, 1980), pp. 136-143.
- [DM84] Dosch, W. and B. Moller, “Busy and lazy FP with infinite objects”, *1984 ACM Symposium on LISP and Functional Programming*, Austin, Texas (August, 1984), pp. 282-292.
- [En71] Engeler, E., “Algebras and combinators”, *Algebra Universalis* 13 (1971), pp. 389-392.

- [FLO83] Fortune, S., D. Leivant, and M.J. O'Donnell, "The expressiveness of simple and second-order type structures", *Journal ACM* 30:1 (January, 1983), pp. 151-185.
- [FW76] Friedman, D. and D. Wise, "Cons should not evaluate its arguments", *Third International Colloquium on Automata, Languages, and Programming* (1976), pp. 257-284.
- [GMP82] Goguen, J., J. Meseguer, and D. Plaisted, "Programming with parameterized abstract objects in OBJ", in *Theory and Practice of Software Technology*, North-Holland (1982).
- [HW³84] Halpern, J.Y., J.H. Williams, E.L. Wimmers, and T.C. Winkler, "Denotational semantics and rewrite rules for FP", to appear as an IBM Research Report (1984).
- [HM76] Henderson, P. and J.H. Morris, "A lazy evaluator", *Third ACM Symposium on Principles of Programming Languages* (January, 1976), pp. 95-103.
- [HO84] Hoffmann, C.M. and M.J. O'Donnell, "Implementation of an interpreter for abstract equations", *Eleventh ACM Symposium on Principles of Programming Languages* (January, 1984), pp. 111-120.
- [Me82] Meyer, A.R., "What is a model of the λ -calculus?", *Information and Control* 52, (1982) pp. 87-122.
- [O'D77] O'Donnell, M.J., *Computing in Systems Described by Equations*, Lecture Notes in Computer Science No. 58, Springer-Verlag (1977).
- [Ro67] Rogers, H., Jr., *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York (1967).
- [St77] Stoy, J., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press (1977).
- [Tu79] Turner, D.A., "A new implementation technique for applicative languages", *Software practice and Experience* 9 (1979), pp. 31-49.
- [Wi81] Williams, J.H., "Formal representations for recursively defined functional programs", in *Formalization of Programming Concepts*, Lecture Notes in Computer Science No. 107, Springer-Verlag (April, 1981), pp. 460-470.
- [Wi82] Williams, J.H., "On the development of the algebra of functional programs", *ACM Trans. on Prog. Langs. and Sys.* 4:4 (October, 1982), pp. 733-757.

