A Logical Basis for Functional Programming (Extended Abstract)

N. Shankar
Computer Science Department
Stanford University
Stanford, CA 94305, USA.

The work reported here was supported by NSF Grant No. CCR-8718605.

1 Introduction

Functional programming provides a powerful medium for specifying computations and is frequently employed as a foundational tool to explain other forms of programming [13, 1, 5]. Many of the existing axiom systems which have been proposed for functional programming are either limited in scope or too complicated from the point of view of a programmer. We present an axiomatic framework for proving properties of functional programs which is, on the one hand, simpler than the LCF family of logics [10, 14], and on the other, more powerful than the Boyer-Moore logic [3] in its ability to deal with partial and higher-order functions. Plotkin [16] describes an axiomatic framework which is similar to the one given here. The objective here is to design a logic which is simple and yet powerful enough for proving the program properties which arise in practice, and which can be effectively mechanized. The differences among the various approaches are described as well as their weaknesses. Illustrative examples are provided.

1.1 Functional programs and their properties

We assume familiarity with some higher-order functional programming language such as Scheme, SASL, or ML. The syntax of Scheme is adopted in the introductory outline. A simple example of a function definition is presented below in which we define a function which returns the square of a given number.

Function Definition 1 *(square)*

```
(define (square x) (times x x))
```

The above definition could be seen as stating as an axiom, the equality

```
(square x) = (times x x).
```

We use, $\triangleq$, for definitional equalities like the one above.

A more complicated example is the following recursive definition of addition, where `coerce` coerces non-numbers to `0` and maps numbers to themselves.

Function Definition 2 *(plus)*

```
(define (plus x y)
   (if (positive?  x)
       (add1 (plus (sub1 x) y))
       (coerce y)))
```

A standard property of `plus` is associativity which is expressed by the equality below.

Program Property 1 *(Associativity)*

```
(plus (plus x y) z) = (plus x (plus y z))
```

There are several classifications of functional programming languages according to whether

1. Functions are *first-order* or *higher-order*. An example of a higher-order operation is the Scheme function `map` which when used as `(map square lst)` returns the list of squares of the numbers in `lst`. Pure Lisp is a first-order language whereas Scheme, SASL, and ML are higher-order languages.

2. The order of evaluation is *call-by-value* or *call-by-name*. The call-by-value order evaluates the arguments to a function before substituting the values into the body of the function definition. The call-by-name order merely substitutes the unevaluated arguments into the body of the function definition and is also called *lazy evaluation*. Scheme employs call-by-value evaluation whereas SASL employs lazy evaluation.

3. Terms are *typed* or *untyped*. Types can be simple, recursive, polymorphic, and so on. Type-checking is useful for catching errors at compile-time at the expense of some inflexibility. ML is a typed language whereas SASL and Scheme are untyped.

We assume below that we are working with a higher-order, untyped, call-by-value language, e.g., pure Scheme. The operation of *lambda abstraction* constructs a function defined by a given term, with respect to the abstracted variables. For example, the `square` function could be represented by `(lambda (x) (times x x))`. Consequently the term, `(map` (lambda (x) (times x x)) lst), has the same value as (map square lst). The identity function could be written as (lambda (x) x). In an untyped, higher-order functional language, *self-application* is possible. If I $\triangleq$ (lambda (x) x), then (I I) evaluates to I. Self-application leads inevitably to *non-termination*. If omega $\triangleq$ (lambda (f) (f f)), then the computation of (omega omega) never halts. Non-termination is, of course, possible in the first-order or the typed framework as well. For simplicity, we do not distinguish between computations which are abortive and those that do not terminate. The assertion that a term **t** converges to a value upon computation is written as **t** $\downarrow$. The assertion, '**t** diverges or aborts' is written as **t** $\uparrow$.

We list below some additional example program properties which one might hope that an axiomatic theory would establish.

Program Property 2 *(Ninetyone)*

(f91 x)$\downarrow$ $\rightarrow$ (f91 x) = (h91 x),

where

```
(f91 x) ≜
        (if (> x 100)
            (difference x 10)
            (f91 (f91 (plus x 11))))
(h91 x) ≜
        (if (> x 100)
            (difference x 10)
            91).
```

Program Property 3 *(Subst-var-var)*

(subst var var exp) = exp,

where

```
(subst term var exp) ≜
                     (if (pair?  exp)
                         (cons (car exp)
                               (map (lambda (x) (subst term var x))
                                    (cdr exp)))
                         (if (equal?  var exp)
                             term
                             exp))
```

Program Property 4 *(Omega)*

$$\texttt{(omega omega)}\uparrow,$$

where `omega` $\triangleq$ `(lambda (f) (f f))`.

Program Property 5 *(Plus-plus1)*

$$\texttt{(plus x y) = (plus1 x y plus1)},$$

where

```
(plus1 x y f) ≜
          (if (positive? x)
              (add1 (f (sub1 x) y f))
              y)
```

Note how the term `(plus1 x y plus1)` employs self-application to mimic the recursion in `plus`. We are not assuming that either `plus` or `plus1` is known to be total.

In the next subsection we examine three previous axiomatizations proposed for functional programming to see how they perform with respect to the above program properties.

1.2 Approaches to proving properties of functional programs

The descriptions of the three formalizations of functional programming sketched below will be useful for understanding the logic presented in the next section. There are a number of other formalisms [8, 12, 9] which are not considered below for lack of space. All the formalisms below are first-order theories.

1.2.1 The Boyer-Moore logic

The Boyer-Moore logic is a quantifier-free first-order theory of total recursive first-order functions [3]. Function definitions are admitted as axioms only if a total function results. The logic provides a rule of induction which is actually a powerful form of McCarthy's rule of *recursion-induction* [13]. Let $\overline{\mathtt{x}}$ represent $\mathtt{x}_1 \ \ldots \ \mathtt{x}_n$. The rule of recursion-induction is

$$\frac{(\mathtt{f}\ \ \overline{\mathtt{x}}) \triangleq \tau[\mathtt{f}, \overline{\mathtt{x}}];\ \ (\mathtt{f}\ \ \overline{\mathtt{x}})\downarrow;\ \ \mathbf{s} = \tau[(\lambda\overline{\mathtt{x}}.\ \mathbf{s}), \overline{\mathtt{x}}];\ \ \mathbf{t} = \tau[(\lambda\overline{\mathtt{x}}.\ \mathbf{t}), \overline{\mathtt{x}}]}{\mathbf{s} = \mathbf{t}.}$$

The use of first-order lambda abstraction above is eliminable, but convenient. The rule essentially says that if two terms satisfy the defining equation of a total recursive function, then they must be equal. Termination is proved by induction over the ordinals up to ϵ_0. The reasoning about termination is done outside the Boyer-Moore logic, though it relies on facts which are proved within the logic.

The Boyer-Moore logic cannot formalize program properties *Omega, Subst-var-var,* and *Plus-plus1*, since they involve higher-order functions. The logic can establish the associativity of `plus`, if `plus` is defined to be a total function. It cannot establish program property *Ninetyone*, since `f91` has to be proved total before it can be admitted, and program property *Ninetyone* is needed to prove that `f91` is total. A recent extension to the logic makes some progress towards these problems [4].

1.2.2 LCF: Logic for Computable Functions

LCF actually refers to a family of theories which grew out of Scott's investigations of models of the lambda calculus [17, 10, 14]. The primary logic motivating much of the development of LCF is called PPLAMBDA which formalizes typed, call-by-name functions. In PPLAMBDA, terms are written along with their types as $\mathtt{s}{:}\sigma$. Types are omitted since they can be inferred from context. Each type σ contains an undefined element $\perp : \sigma$ which represents non-termination. The variables of type σ range over this undefined element as well. To each type σ, there is a least fixpoint operator, FIX_σ of type $(\sigma \rightarrow \sigma) \rightarrow \sigma$, which is used to define recursive functions. (FIX `F`) computes to (`F` (FIX `F`)). For example `plus` would be defined as

```
plus ≜ (FIX (lambda (f)
                    (lambda (x y)
                            (if (positive?  x)
                                (add1 (f (sub1 x) y))
                                y))))
```

For a given `F`, (FIX `F`) yields a least (defined) solution to the equation, `f = (F f)`, with respect to the Scott ordering on the domain corresponding to σ. Thus, (FIX `(lambda` `(x:`σ`)` `x))` is equal to $\perp : \sigma$.

Proofs by induction are carried out by Scott's rule of computational induction. The intuition behind this rule is that a function `f` defined recursively as (FIX `F`), is the limit of the monotonically increasing sequence of partial approximations,

$$\perp, (\mathtt{F}\ \perp), (\mathtt{F}\ (\mathtt{F}\ \perp)), \ldots, (\mathtt{F}^{\mathtt{n}} \perp), \ldots\ .$$

Now, in order to prove a property $A[(FIX\ \mathtt{F})]$, one instead proves that $A[(\mathtt{F}^n\ \perp)]$ holds for each n, and ensures that A is *admissible*, i.e., if for each n, $A[(\mathtt{F}^n\ \perp\ )]$, then $A[(FIX\ \mathtt{F})]$ holds. rule can be depicted as

$$\frac{A[\perp]\ ;\ \forall \mathrm{g}. A[g] \rightarrow A[(\mathtt{F}\ \ \mathtt{g})]}{A[(FIX\ \mathtt{F})].}$$

LCF can formalize the proofs of program properties *Associativity*, *Ninetyone*, and *Subst-var-var*, but not those of properties *Omega* and *Plus-plus1*. LCF without recursive types does not admit self-application so a function like `plus1` would not be definable. The newer versions of LCF [14] do contain recursive types. The proof of *Associativity* requires the derivation of structural induction over the naturals from Scott induction. Scott induction is only able to deal with inductions suggested by recursions which are explicity defined by the fixpoint operator. It is unable to deal with recursions arising from self-application as in program properties *Omega* and *Plus-plus1*. The fact that variables range over $\bot$ is awkward when reasoning about total functions and inappropriate for a formalizing a functional language with call-by-value evaluation. The need to prove induction formulas to be admissible is a significant nuisance. Plotkin's formalization [16] attacks the problem of $\bot$ somewhat by employing a logic of partial existence[1]; variables still range over $\bot$, but only defined terms "exist." Plotkin also specifies syntactic criteria for checking formulas for admissibility and employs a type framework in the style of the Martin-Löf type theories.

1.2.3 The McCarthy-Cartwright first-order theory of programs

A major contribution of McCarthy and Cartwright's first-order theory of programs (abbreviated here as MC) [6] was to show that the Scott ordering is stronger than is needed for reasoning about first-order functional programs. As with PPLAMBDA, MC is a first-order logic with variables ranging over $\bot$. However there are no higher-order types and all functions are first-order. Recursive function definitions are simply written down as equations, $(\mathtt{f}\ \bar{\mathtt{x}}) \triangleq \tau_{\mathtt{f}}[\mathtt{f},\ \bar{\mathtt{x}}]$, where $\tau_{\mathtt{f}}$ is a term in the given parameters. The rule of induction, called *minimization*, is a first-order version of Park's rule and it has the form

$$\frac{\tau_{\mathtt{f}}[(\lambda\bar{\mathtt{x}}.\ \mathbf{s}), \bar{\mathtt{x}}] \sqsubseteq \mathbf{s}}{(\mathtt{f}\ \ \bar{\mathtt{x}}) \sqsubseteq \mathbf{s}}$$

where $\mathbf{r} \sqsubseteq \mathbf{s}$ abbreviates $\mathbf{r} \downarrow \rightarrow \mathbf{r} = \mathbf{t}$ and again only first-order lambda abstraction is used.

MC can formalize proofs of program properties *Associativity* and *Ninetyone* but not the others. Program properties *Omega* and *Plus-plus1* employ higher-order functions, `omega` and `plus1`, which are inexpressible in the logic.

2 A Logic for Functional Programming

The theory described in this section draws heavily on the work described in the previous section. It is able to formalize proofs of the same properties which LCF can establish. Later, we discuss how the logic can be strengthened (perhaps not very elegantly) to cope with the other properties. The logic is a first-order theory of higher-order programs. The theory is

[1] In a logic of partial terms [2], variables (free or bound) only range over defined terms, whereas in a logic of partial existence [18], free variables range over all terms but quantified variables range only over defined terms

specified for a type-free language with a call-by-value mode of evaluation, but it can easily be adapted to a typed or a call-by-name framework. The theory is built on a logic of partial terms [2], i.e., only defined terms can instantiate the free variables in a theorem. The axioms will first be listed and the semantic soundness of these axioms will be argued in the next section. Rules of inference are also called axioms below for uniformity.

2.1 The syntax

The symbols of the language consist of the *variables* `f, g, x, y, z`; the *constants* including the undefined constant, $\perp$; λ or `lambda`; the *relation* symbols: $\downarrow$ (definedness or convergence), and $=$ (extensional equality); the *predicate* symbols, and the usual logical connectives and quantifiers: $\neg, \vee, \wedge, \rightarrow, \leftrightarrow, \exists$, and $\forall$. There is an implicit dyadic function symbol, *apply*, which never appears since juxtaposition is used to represent application. Note the distinction between relation and predicate symbols which will be explained later.

Terms are either constants; *applications* of the form $(\mathbf{f}\ \mathbf{s})$, where $\mathbf{f}$ and $\mathbf{s}$ are terms; or *lambda abstractions* of the form $(\lambda\mathbf{x}.\ \ \mathbf{s})$, where $\mathbf{x}$ is a variable and $\mathbf{s}$ is a term. Parentheses will be omitted from left-associatively nested application terms so that the expression $(\mathbf{f}\ \mathbf{s}_1\ \ldots\ \mathbf{s}_n)$ is to be parsed as $(\ldots(\mathbf{f}\ \mathbf{s}_1)\ \ldots\ \mathbf{s}_n)$. The bold-face letters **a, b, f, g, r, s,** and **t** will be used as syntactic variables to represent terms. The syntactic variables **u, v, x, y,** and **z** will be used to represent variables.

Atomic formulas are of one of the forms: $\mathbf{s} = \mathbf{t}$; $\mathbf{s}\downarrow$; or $\mathbf{p}(\mathbf{s}_1, \ldots, \mathbf{s}_n)$, where $\mathbf{p}$ is an n-ary predicate symbol.

Formulas are either atomic formulas or are of one of the forms: $(\neg\mathbf{A})$, $(\mathbf{A} \vee \mathbf{B})$, $(\mathbf{A} \wedge \mathbf{B})$, $(\mathbf{A} \rightarrow \mathbf{B})$, $(\mathbf{A} \leftrightarrow \mathbf{B})$, $(\forall\mathbf{x}.\ \mathbf{A})$, or $(\exists\mathbf{x}.\ \mathbf{A})$; where $\mathbf{A}$ and $\mathbf{B}$ are formulas. Parentheses will be omitted when they do not add to the clarity of the formula.

Substitution, $[\mathbf{a}/\mathbf{x}]\mathbf{B}$ or $[\mathbf{a}/\mathbf{x}]\mathbf{s}$, is defined so that the bound variables in the expression being substituted into, are renamed to avoid free/bound variable clashes. $\mathbf{A}[\mathbf{a}]$ represents $[\mathbf{a}/\mathbf{x}]\mathbf{A}[\mathbf{x}]$. Expressions which are identical modulo the renaming of bound variables will be regarded as syntactically identical, thereby dispensing with the α-rule.

It is important to observe that since the variables in the logic only range over the defined terms, a number of the axioms below are actually axiom schemes written using syntactic variables which range over all terms. The distinction between formal and syntactic variables is crucial for a correct reading of the axioms.

2.2 The logical axioms and rules

Since the non-logical axioms are the ones that are of interest, we devote very little attention to the logical axioms. The equality axioms appear among the non-logical axioms though some of the axioms are logical. The logical axioms formalize a logic of partial terms, i.e., variables only range over defined terms. Since the logical axioms are not the central issue, they receive only cursory treatment.

Axiom 1 Tautology rule: *If* $\mathbf{B}$ *is a tautological consequence of* $\mathbf{A_1}, \ldots, \mathbf{A_n}$*, then*

$$\frac{\mathbf{A_1};\ \ldots;\ \mathbf{A_n}}{\mathbf{B}}.$$

Axiom 2 Instantiation: *Variables can only be instantiated with defined terms.*

$$(\forall \mathbf{x}.\ \mathbf{A}) \wedge\ (\mathbf{s}\downarrow) \rightarrow\ [\mathbf{s}/\mathbf{x}]\mathbf{A}.$$

Axiom 3 Generalization: *If* $\mathbf{x}$ *does not occur free in* $\mathbf{A}$*, then*

$$\frac{\mathbf{A} \rightarrow \mathbf{B}}{\mathbf{A} \rightarrow (\forall \mathbf{x}.\ \mathbf{B})}.$$

Axiom 4 Existential quantifier:

$$(\exists \mathbf{x}.\ \mathbf{A}) \leftrightarrow \neg(\forall \mathbf{x}.\ \neg\mathbf{A})$$

Axiom 5 Domain of predicates: *A predicate fails to hold if any of its arguments fail to denote. If* $\mathbf{p}$ *is an n-ary predicate symbol, then*

$$\mathbf{p}(\mathbf{s}_1, \ldots, \mathbf{s}_n) \rightarrow (\mathbf{s}_1\downarrow \wedge \ldots \wedge \mathbf{s}_n\downarrow).$$

2.3 Definedness

Axiom 6 Variables: *Since variables only range over defined terms, they are themselves defined.*

$$\mathbf{x}\downarrow$$

Axiom 7 Functions: *Functions are always defined since a machine stops its evaluation when it encounters a lambda-expression.*

$$(\lambda \mathbf{x}.\ \mathbf{s})\downarrow$$

Axiom 8 Substitutivity:

$$\mathbf{s} = \mathbf{t} \rightarrow \mathbf{s}\downarrow \rightarrow \mathbf{t}\downarrow$$

Axiom 9 Function predicate: *The class of functions can defined as below.*

$$\mathbf{Function}(\mathbf{x}) \leftrightarrow \mathbf{x} = (\lambda \mathbf{y}.\ (\mathbf{x}\ \mathbf{y})).$$

Axiom 10 Application definedness: *The call-by-value mode of evaluation is specified below. The converse of the axiom clearly need not hold.*

$$(\mathbf{f}\ \mathbf{s}) \downarrow \rightarrow \mathbf{Function}(\mathbf{f}) \wedge \mathbf{s} \downarrow .$$

Axiom 11 The Fixpoint operation: *This operation will be used for specifying recursive definitions.*

$$\mathbf{Function}((FIX\ \mathbf{x}))$$

Axiom 12 Undefined:

$$\neg \bot \downarrow$$

2.4 Equality

Axiom 13 Reflexivity:

$$\mathbf{s} = \mathbf{s}$$

Axiom 14 Symmetry:

$$\mathbf{s} = \mathbf{t} \rightarrow \mathbf{t} = \mathbf{s}$$

Axiom 15 Transitivity:

$$\mathbf{r} = \mathbf{s} \wedge \mathbf{s} = \mathbf{t} \rightarrow \mathbf{r} = \mathbf{t}$$

A number of free logics use an equality predicate which is restricted to defined terms. These two equalities are inter-definable, but the one employed here seems to match the programmers view of the equality of programs. The expression $\mathbf{s} \uparrow$ abbreviates $\neg \mathbf{s} \downarrow$.

Axiom 16 Equality of undefined:

$$(\mathbf{s} \uparrow \wedge \mathbf{t} \uparrow) \rightarrow \mathbf{s} = \mathbf{t}$$

The call-by-value mode of evaluation is implicit here since $\mathbf{y}$ only ranges over defined terms. To formalize a call-by-name evaluation scheme, the logic must be changed to a total term logic, and the **Application definedness** axiom appropriately modified. The axiom below remains unchanged.

Axiom 17 The β-rule:

$$((\lambda \mathbf{x}.\ \mathbf{s})\ \mathbf{y}) = [\mathbf{y}/\mathbf{x}]\mathbf{s}$$

Axiom 18 The ξ-rule or extensionality:

$$\frac{\mathbf{s} = \mathbf{t}}{(\lambda \mathbf{x}.\mathbf{s}) = (\lambda \mathbf{x}.\mathbf{t})}$$

Axiom 19 FIX:

$$(FIX\ \mathbf{x}\ \mathbf{y}) = (\mathbf{x}\ (FIX\ \mathbf{x})\ \mathbf{y})$$

Axiom 20 Application substitutivity:

$$\mathbf{f} = \mathbf{g} \wedge \mathbf{s} = \mathbf{t} \rightarrow (\mathbf{f}\ \mathbf{s}) = (\mathbf{g}\ \mathbf{t})$$

Axiom 21 Predicate substitutivity:

$$\mathbf{x}_1 = \mathbf{y}_1 \wedge \ldots \wedge \mathbf{x}_n = \mathbf{y}_n \rightarrow \mathbf{p}(\mathbf{x}_1, \ldots, \mathbf{x}_n) \rightarrow \mathbf{p}(\mathbf{y}_1, \ldots, \mathbf{y}_n)$$

2.5 Induction

The induction axiom is a special case of Scott induction where the form of the induction formula is restricted to being of the form $(\forall \overline{\mathbf{x}}.\ (\mathbf{u}\ \overline{\mathbf{x}}) \downarrow \rightarrow \mathbf{A}[(\mathbf{u}\ \overline{\mathbf{x}}), \overline{\mathbf{x}}])$. The intuitive idea is that if one wishes to prove a statement over the domain of a partial recursive function, then the inductive scheme induced by the recursion scheme of the function is a valid one. This form of recursion corresponds very closely to what is implemented in the Boyer-Moore prover, where all functions are total and hence all recursion schemes generate valid recursion schemes. The induction formula is easily seen to be admissible, since if for some $\mathbf{x}$, $((FIX\ \mathbf{f})\ \overline{\mathbf{x}}) \downarrow$, then forsome i, $((\mathbf{f}^i\ \bot)\ \overline{\mathbf{x}}) = ((FIX\ \mathbf{f})\ \overline{\mathbf{x}})$. Plotkin [16] has a syntactic criterion for checking admissibility which encompasses the above form. The main empirical claim is that this form essentially covers all the cases of Scott induction which are actually used in practice and would be more easily mechanizable. It can also be viewed as a generalization of the Floyd-Hoare method to functional programs, i.e., the induction scheme applied to first-order tail-recursive programs yields the inductive assertion method. Clark's induction scheme [7] for logic programs can also be seen as an analogue of the scheme below.

Axiom 22

$$\frac{(\forall \mathbf{u}.\ (\forall \overline{\mathbf{x}}.\ (\mathbf{u}\ \overline{\mathbf{x}}) \downarrow \rightarrow \mathbf{A}[(\mathbf{u}\ \overline{\mathbf{x}}), \overline{\mathbf{x}}]) \rightarrow (\forall \overline{\mathbf{x}}.\ ((\mathbf{f}\ \mathbf{u})\ \overline{\mathbf{x}}) \downarrow \rightarrow \mathbf{A}[((\mathbf{f}\ \mathbf{u})\ \overline{\mathbf{x}}), \overline{\mathbf{x}}]))}{(\forall \mathbf{x}.\ ((FIX\ \mathbf{f})\ \overline{\mathbf{x}}) \downarrow \rightarrow \mathbf{A}[((FIX\ \mathbf{f})\ \overline{\mathbf{x}}), \overline{\mathbf{x}}])}$$

2.6 Discussion

The call-by-value lambda calculus formalized in the previous subsections is known to comprise a complete programming language, but programming with it could get quite tedious. To extend the language to something closer to the language used in the programming examples, we need axioms for `if` and the usual Peano-like axioms for s-expressions. Assuming that we have the axioms for s-expressions, the axioms for `if` can be written as below.

Axiom 23 If left:

$$\mathbf{x} \neq \#\mathtt{f} \rightarrow (\mathtt{if}\ \mathbf{x}\ \mathbf{r}\ \mathbf{s}) = \mathbf{r}.$$

Axiom 24 If right:

$$\mathbf{x} = \#\mathtt{f} \rightarrow (\mathtt{if}\ \mathbf{x}\ \mathbf{r}\ \mathbf{s}) = \mathbf{s}.$$

Axiom 25 If test:

$$\mathbf{t} \uparrow \rightarrow (\mathtt{if}\ \mathbf{t}\ \mathbf{r}\ \mathbf{s}) \uparrow.$$

Now the proofs of properties *Associativity*, *Ninetyone*, and *Subst-var-var* can easily be worked out as can most natural examples. In the case of *Ninetyone*, the induction formula is

$$\forall \mathbf{x}.\ (\mathbf{u}\ \mathbf{x}) \downarrow \rightarrow (\mathbf{u}\ \mathbf{x}) = (\mathtt{h91}\ \mathbf{x}).$$

The induction formula for *Subst-var-var* is

$$\forall \mathbf{x}.\ (\mathbf{u}\ \mathbf{term}\ \mathbf{var}\ \mathbf{exp}) \downarrow \rightarrow (\mathtt{subst}\ \mathbf{var}\ \mathbf{var}\ \mathbf{exp}) = \mathbf{var}.$$

The proofs of program properties *Omega* and *Plus1-plus* fail just as they do in LCF[2].

Types could be added to the logic; the only change is that we need a FIX operator for each type. The Scott approximation is not a primitive relation as it is in LCF. A simpler ordering, $(\mathbf{s} \sqsubseteq_{\mathbf{0}} \mathbf{t}) \leftrightarrow (\mathbf{s} \downarrow \rightarrow \mathbf{s} = \mathbf{t})$, can first be defined. The Scott approximation is of course needed for constructing models. If needed, Scott approximation could be defined as, $(\mathbf{s} \sqsubseteq \mathbf{t}) \leftrightarrow \forall \mathbf{f}.\ ((\mathbf{f}\ \mathbf{s}) \sqsubseteq_{\mathbf{0}} (\mathbf{f}\ \mathbf{t}))$. However, the $\sqsubseteq_{\mathbf{0}}$ relation is the one that is really used in proving most program properties. The induction scheme can be viewed as a generalization of Park induction (and hence minimization) and the Boyer-Moore rule of induction as well. Termination could be proved either by simple induction (when quantifiers are present) or by introducing a scheme for ϵ_0-induction, otherwise. Mathematical induction over the natural numbers can be derived from the above induction scheme by asserting the termination of a copying function over the natural numbers. Though slightly complicated[3], the fixpoint operator can be used to define mutually recursive functions as well, and the induction rule provides the appropriate simultaneous induction.

[2]These properties seem to require stronger logics with induction schemes involving computation and subcomputation relations between terms [11, 19]. These relations are necessarily intensional, i.e., they are not substitutive with respect to the extensional equality relation. If $\mathbf{s} \prec \mathbf{t}$ asserts that $\mathbf{s}$ is a sub-computation of $\mathbf{t}$, then the following adaptation of Talcott's rule seems sufficient.

$$\frac{\forall \overline{\mathbf{x}}.\ [(\mathbf{s}[\overline{\mathbf{x}}] \downarrow \wedge (\forall \overline{\mathbf{y}}.\ (\mathbf{s}[\overline{\mathbf{y}}] \prec \mathbf{s}[\overline{\mathbf{x}}]) \rightarrow \mathbf{A}[\overline{\mathbf{y}}])) \rightarrow \mathbf{A}[\overline{\mathbf{x}}]]}{\forall \overline{\mathbf{x}}.\ \mathbf{s}[\overline{\mathbf{x}}] \downarrow \rightarrow \mathbf{A}[\overline{\mathbf{x}}]}$$

[3]Mutual recursive definition is a little easier with the call-by-name FIX operator.

The soundness of the logic can be proved by means of a term model. The universe of the model consists of all the closed (variable-free) terms. Terms which are identical modulo the renaming of bound variables are regarded as identical. We can then give a simultaneous inductive definition of a computation relation between these terms which provides a machine for reducing closed terms to values, and of the class of defined terms. The reason for defining them simultaneously is that the computation rules, e.g., the β-rule, require that their arguments be defined. In the base case, the class of defined terms consists of all the values (including all the lambda-expressions), and the computation relation is empty. In the inductive case, the defined terms are enlarged to include all the terms which reduce (by the computation relation) to defined terms; and the computation relation is enlarged by closing under the defined instances of the computation rules (the β rule, the FIX rule, etc.).

There are several crucial observations about the computation relation. The equality relation can be defined by taking the reflexive, symmetric, and transitive closure of the computation relation with respect to the ξ rule, $(\forall \mathbf{x}.\ \mathbf{s} = \mathbf{t}) \rightarrow (\lambda \mathbf{x}.\ \mathbf{s}) = (\lambda \mathbf{x}.\ \mathbf{t})$, and equating all undefined closed terms. The Church-Rosser theorem for the λ-v-δ-calculus [15] ensures that resulting term model is non-trivial. The defined closed terms form the well-founded part of the universe with respect to the computation relation. Since the computation relation is monotone and compact, the ordinal corresponding to the class of defined terms is ω. In particular, if for element $\mathbf{a}$ of the universe, $((FIX\ \mathbf{f})\ \mathbf{a}) \downarrow$, then for some natural number i, $((\mathbf{f^i}\ \bot)\ \mathbf{a}) = ((FIX\ \mathbf{f})\ \mathbf{a})$. The well-foundedness of the class of defined closed terms justifies the induction axiom.

3 Conclusions

A logic for axiomatizing functional programming has been described. It is based on a logic of partial terms and contains axioms which reflect the operational rules of the language in a straightforward way. All the concepts in the logic are those already familiar to programmers. The logic can be used to prove termination as well as partial correctness. The proofs of program properties can be formulated in elegant and clear terms. The logic can easily be modified for a typed and/or call-by-name framework. The powerful induction heuristics of the Boyer-Moore theorem prover can also be adapted to the present framework.

I am grateful to Carolyn Talcott, Matt Kaufmann, Gordon Plotkin, Sol Feferman, Gian Luigi Bellin, Larry Paulson, Lou Galbiati, and Jussi Ketonen, for useful discussions.

References

[1] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM*, 21:616–641, August 1978.

[2] M. J. Beeson. Proving programs and programming proofs. In *International Congress on Logic, Methodology and Philosophy of Science*, Amsterdam, 1983. North-Holland.

[3] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, NY, 1979.

[4] R. S. Boyer and J. S. Moore. The addition of bounded quantification and partial functions to a computational logic and its theorem prover. Technical Report ICSCA-CMP-52, University of Texas at Austin, Institute of Computing Science, January 1987.

[5] W. H. Burge. *Recursive Programming Techniques*. Addison-Wesley, Reading, MA, 1975.

[6] R. Cartwright and J. McCarthy. Recursive programs as functions in a first order theory. Technical Report STAN-CS-79-717, Computer Science Department, Stanford University, 1979.

[7] K. L. Clark. Predicate logic as a computational formalism. Technical Report 79/59, Department of Computing, Imperial College, 1979.

[8] R. L. Constable, *et al*. *Implementing Mathematics with the Nuprl*. Prentice-Hall, New Jersey, 1986.

[9] S. Feferman. Finitary inductively presented logics. In *Logic Colloquium '88*, pages 191–220, Amsterdam, 1988. North-Holland.

[10] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1979.

[11] M. J. C. Gordon. Operational reasoning and denotational semantics. Technical Report Stanford Artificial Intelligence Laboratory Memo AIM-264, Computer Science Department, Stanford University, 1975.

[12] S. Hayashi and H. Nakano. PX — a computational logic. Technical Report 573, Research Institute for Mathematical Sciences, Kyoto University, 1987.

[13] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hershberg, editors, *Computer Programming and Formal Systems*. North-Holland, 1963.

[14] L. C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, Cambridge, England, 1987.

[15] G. Plotkin. Call-by-name, call-by-value and the lambda-v-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[16] G. D. Plotkin. (Towards a) logic for computable functions. Unpublished Lecture Notes from the ASL Summer School, 1985.

[17] D. S. Scott. A type-theoretic alternative to CUCH, ISWIM, OWHY. Typed notes, Oxford, 1969.

[18] D. S. Scott. Identity and existence in intuitionistic logic. In M. P. Fourman, C. J. Mulvey, and D. S. Scott, editors, *Applications of Sheaves*, volume 753 of *Lecture Notes in Mathematics*, pages 660–696, Berlin, 1979. Springer-Verlag.

[19] C. L. Talcott. *The Essence of RUM: A theory of the intensional and extensional aspects of Lisp-type computation.* PhD thesis, Stanford University Computer Science Department, 1985.