AN EXPERIMENT IN PRACTICAL SEMANTICS*

Maurice Naftalin**

Abstract

me too is a functional language in which executable specifications of software components are constructed from abstract models and operations defined on them. The principal data type provided by the language is the set. This paper examines in detail the extent to which the objects manipulated by **me too** programs do in fact behave like the mathematical ideal of sets. It draws conclusions about the design of the language, and about the feasibility of such projects in 'applied semantics'.

* This work was supported in part by an SERC grant.

**Department of Computing Science, University of Stirling, Stirling, Scotland

1 INTRODUCTION

me too ([Henderson 84]) is a functional language for the specification of software components. It is similar to a pure Lisp called Lispkit Lisp ([Henderson 80]) except that it has sets, finite functions (mappings) and sequences for data objects instead of S-expressions. These types, together with the operations that are provided for manipulating them, are used to construct abstract model specifications, which are then executed via translation to Lisp.

A short example will illustrate some of its features. The problem of the bill of materials is well known from database applications. A factory management system is to record what components each manufactured item is built from. Each of these components may itself be an assembly. Some parts are atomic; they have no components (other than themselves). The operation of finding all the components of a given item, including the intermediate sub-assemblies, is called 'parts explosion'.

The model used here for the database is a finite function from component identifiers to sets of components. (A finite function in **me too** is a function with finite domain, specified by its graph). We define the primitive type P (for part identifier) and construct the type B (for bill of materials) as a finite function

    B = ff(P,set(P)).

The parts explosion operation can now be defined in **me too** as

    explode: P X B → set(P)

    explode(p,b) ≡ {p} ∪ union{explode(c,b):c ← b[p]}

where union denotes distributed union over a family of sets, and square brackets denote function application. The notation {F(I):I ← S} means the image of the set S formed by pointwise application of the function F.

This function definition is directly executable. The intended use of such definitions is to allow system specifications, in the style of functional programs, to be debugged at an early stage. Their execution then provides early feedback to the requirements analysis. If performance is not important, the corrected definitions can form a usable system. Otherwise, they are used as the specification of an imperative program.

The principal data type of **me too** is the set (its other data types are defined in terms of sets. They are not dealt with in this paper). A **me too** user is intended to think of objects of this data type simply as the sets familiar from classical set theory. Of course, in practice this type is implemented by completely different primitive types. So in order to have confidence that the specification as executed is faithful to the intentions of the user, it is necessary to show the correctness of the implementation with respect to classical set theory.

The usual way of showing that objects in some system 'behave like' sets is to demonstrate that the system forms a model for the axioms of set theory. This is done by relativising the axioms, that is to say by rewriting them with quantification restricted to the domain of the interpretation, and then verifying the resulting formulae. The problem with using this approach here is that the domain of the interpretation would be terms of **me too**, and these contain symbols claiming to correspond to the derived symbols of set theory as well as primitive ones. In order to provide a satisfactory semantics for **me too**, it would be necessary to extend set theory with an appropriate system of terms so as to treat as primitive many symbols that in classical theory are derived symbols.

An alternative approach, the one taken here, is to use the denotational model of **me too** as a model for set theory. We can then check the correctness of the implementation by ensuring that the effect of each **me too** operator on this model accords with the interpretation of the corresponding set theoretic operator.

The method is illustrated below:

$$\begin{tikzcd}[column sep=huge, row sep=huge]
\underline{\textbf{me too}} \atop \underline{\text{terms}} \arrow[rr, "\text{Desired correspondence}", "\text{defined by function Sval}"'] \arrow[dr, "\text{meaning given by Eval}"', "\text{of denotational semantics}"] & & \underline{\text{Sets}} \\
& \underline{\text{Infinite Lists}} \arrow[ur, "\text{Abstraction function}", "\text{Abs}"'] &
\end{tikzcd}$$

Section 2 defines the semantic function Eval for **me too.** Section 3 sketches a suitable (typed) version of set theory. Section 4 shows how to establish the correctness of the implementation. The principal result is that a function Abs can be defined so that Sval = Abs o Eval for every program, and that the domain of lists, viewed through the equivalence relation that Abs induces, then forms a model for (some of) the axioms of set theory. Section 5 explores the set theoretic consequences of some alternative implementations. Section 6 draws some conclusions about the design of the language and compares the two methods of defining its semantics outlined in the preceding paragraphs.

An important aspect of the project is the use of the automatic proof assistant Cambridge LCF to verify the formal conditions given in section 4 for the correctness of the implementation. This work is still in progress.

2 DENOTATIONAL SEMANTICS OF **me too**

The standard semantics of the current implementation of **me too** are defined in this section. Since the definitions are given via interpretation of **me too** into Lispkit Lisp, the semantics of this are given first. The notational conventions normal in denotational semantics ([Stoy 77]) are used. $\oplus$ and $\boxtimes$ denote respectively the coalesced sum and coalesced product constructors. A sum domain $S = A \oplus B$ has the following strict operators associated with it: two injection operators inS (one for each summand), projection operators outA and outB, and predicates isA and isB. A product domain $P = A \boxtimes B$ has the associated strict operators fst and snd. The lifting operator $_\perp$ is defined by

$$D_\perp = \{\langle true, x\rangle \mid x \in D\} \cup \{\perp\}$$

with associated operators lift: $D \rightarrow D_\perp$ and drop: $D_\perp \rightarrow D$, defined by

$$lift(x) = \langle true, x\rangle$$
$$drop(\langle true, x\rangle) = x$$
$$drop(\perp) = \perp$$

The binary operator + on environments is defined by:

$$\rho_0 + \rho_1 = \lambda I.\ \rho_1[I] = free \rightarrow \rho_0[I],\ \rho_1[I]$$

2.1 Semantics of Lispkit Lisp

Syntactic Domains

$A \in$ Ato = Nml + Sym + {T,F,nil}	Atoms
$I \in$ Ide	Identifiers
$E \in$ Exp	Expressions
$\Delta \in$ Dec	Declarations
$\Sigma \in$ Sxp	S-expressions

Abstract Syntax

$$\Sigma \rightarrow A \mid (\Sigma.\Sigma)$$
$$E \rightarrow \underline{quote}\ \Sigma \mid I \mid \underline{add}\ E\ E \mid \underline{car}\ E \mid \underline{cdr}\ E \mid \underline{cons}\ E\ E \mid \underline{eq}\ E\ E \mid \underline{atom}\ E \mid \underline{if}\ E\ \underline{then}\ E\ \underline{else}\ E \mid \underline{let}\ \Delta\ E \mid \underline{letrec}\ \Delta\ E \mid \underline{lambda}(I,I,\ldots,I)E \mid E(E,E,\ldots,E)$$
$$\Delta \rightarrow I = E \mid \Delta\ \underline{and}\ \Delta$$

Semantic Domains

$\mathbf{N}$	Integers
$\mathbf{T}$	Truth values
$\mathbf{X} = \{nil\}_\perp$	Nil
$\mathbf{Y}$	Other symbolic values
$\mathbf{A} = \mathbf{N} \oplus \mathbf{T} \oplus \mathbf{Y}$	Atomic values
$\mathbf{S} = \mathbf{A} \oplus \mathbf{L} \oplus \mathbf{X}$	S-expressions
$\mathbf{L} = \mathbf{S}_\perp \boxtimes \mathbf{S}_\perp$	Lists
$e \in \mathbf{E} = \mathbf{S} \oplus \mathbf{F} \oplus \{err\}$	Expressible values

$\mathbf{F} = \mathbf{E}^n \rightarrow \mathbf{E}$ Functions

$\rho \in \mathbf{U} = \mathrm{Ide} \rightarrow (\mathbf{E} \oplus \{\mathrm{free}\})$ Environments

<u>Semantic Functions</u>

Aval: $\mathrm{Ato} \rightarrow (\mathbf{A} \oplus \mathbf{S})$

$\mathrm{Aval}[\![T]\!] = \mathrm{inA(true)}$
$\mathrm{Aval}[\![F]\!] = \mathrm{inA(false)}$

$\mathrm{Aval}[\![nil]\!] = \mathrm{inS(nil)}$

Aval: $\mathrm{Nml} \rightarrow \mathbf{N}$

Aval: $\mathrm{Sym} \rightarrow \mathbf{Y}$

Eval: $\mathrm{Exp} \rightarrow \mathbf{U} \rightarrow \mathbf{E}$

(In the following definitions, inXY and outXY are abbreviations for $\mathrm{inX} \circ \mathrm{inY}$ and $\mathrm{outX} \circ \mathrm{outY}$ respectively, and e_i is an abbreviation for $\mathrm{Eval}[\![E_i]\!]\rho$).

$\mathrm{Eval}[\![\underline{\mathrm{quote}}\ A]\!]\rho = \mathrm{inES}(\mathrm{Aval}[\![A]\!])$

$\mathrm{Eval}[\![\underline{\mathrm{quote}}\ (\Sigma_0.\Sigma_1)]\!]\rho = \mathrm{inES}(\mathrm{pair}(\mathrm{lift}(\mathrm{outS}(\mathrm{Eval}[\![\Sigma_0]\!]\rho)),$
$\qquad\qquad \mathrm{lift}(\mathrm{outS}(\mathrm{Eval}[\![\Sigma_1]\!]\rho))))$

$\mathrm{Eval}[\![I]\!]\rho = \rho[\![I]\!]$

$\mathrm{Eval}[\![\underline{\mathrm{add}}\ E_0\ E_1]\!]\rho = \mathrm{checkN}(e_0)\ \mathbf{and}\ \mathrm{checkN}(e_1) \rightarrow$
$\qquad \mathrm{inESA}(\mathrm{outNAS}(e_0) + \mathrm{outNAS}(e_1)),$
$\qquad \mathrm{err}$

where $\mathrm{checkN} = \lambda e.(\mathrm{isS}(e) \rightarrow (\mathrm{isA}(\mathrm{outS}(e)) \rightarrow \mathrm{isN}(\mathrm{outAS}(e)),$
$\qquad \mathrm{inESA(false)}),$
$\qquad \mathrm{inESA(false)})$

and the other check functions are defined similarly.

$\mathrm{Eval}[\![\underline{\mathrm{car}}\ E]\!]\rho = \mathrm{checkL}(e) \rightarrow \mathrm{inE}(\mathrm{drop}(\mathrm{fst}(\mathrm{outLS}(e)))),$
$\qquad \mathrm{err}$

$\mathrm{Eval}[\![\underline{\mathrm{cdr}}\ E]\!]\rho = \mathrm{checkL}(e) \rightarrow \mathrm{inE}(\mathrm{drop}(\mathrm{snd}(\mathrm{outLS}(e)))),$
$\qquad \mathrm{err}$

$\mathrm{Eval}[\![\underline{\mathrm{cons}}\ E_0\ E_1]\!]\rho = \mathrm{isS}(e_0)\ \mathbf{and}\ \mathrm{isS}(e_1) \rightarrow \mathrm{inES}(\mathrm{pair}(\mathrm{lift}(e_0),\mathrm{lift}(e_1))),$
$\qquad \mathrm{err}$

$\mathrm{Eval}[\![\underline{\mathrm{eq}}\ E_0\ E_1]\!]\rho = \mathrm{checkA}(e_0)\ \mathbf{and}\ \mathrm{checkA}(e_1) \rightarrow \mathrm{inESA}(e_0 = e_1),$
$\qquad \mathrm{inESA(false)}$

$\mathrm{Eval}[\![\underline{\mathrm{atom}}\ E]\!]\rho = \mathrm{isS}(e) \rightarrow \mathrm{inESA}(\mathrm{isA}(\mathrm{outS}(e))),$
$\qquad \mathrm{inESA(false)}$

$\mathrm{Eval}[\![\underline{\mathrm{if}}\ E_0\ \underline{\mathrm{then}}\ E_1\ \underline{\mathrm{else}}\ E_2]\!]\rho = \mathrm{checkT}(e_0) \rightarrow (\mathrm{outTAS}(e_0) \rightarrow e_1, e_2),$
$\qquad \mathrm{err}$

$\mathrm{Eval}[\![\underline{\mathrm{let}}\ \Delta\ E]\!]\rho = \mathrm{Eval}[\![E]\!](\rho + \mathrm{Dval}[\![\Delta]\!]\rho)$

$\mathrm{Eval}[\![\underline{\mathrm{letrec}}\ \Delta\ E]\!]\rho = \mathrm{Eval}[\![E]\!](Y(\lambda\rho.\rho + \mathrm{Dval}[\![\Delta]\!]\rho))$
$\qquad$ where $Y =_{\mathrm{def}} \lambda h.((\lambda x.h(xx))\ (\lambda x.h(xx)))$

$\mathrm{Eval}[\![\underline{\mathrm{lambda}}(I_0,I_1,\ldots,I_n)E]\!]\rho =$
$\qquad \lambda\delta_0\delta_1\ldots\delta_n.\mathrm{Eval}[\![E]\!](\rho[\delta_0/I_0,\delta_1/I_1,\ldots,\delta_n/I_n])$

$\mathrm{Eval}[\![E_0(E_1,E_2,\ldots,E_n)]\!]\rho = \mathrm{isF}(e_0) \rightarrow (\mathrm{outF}(e_0))(e_1,e_2,\ldots,e_n),$
$\qquad \mathrm{err}$

Dval: $\mathrm{Dec} \rightarrow \mathbf{U} \rightarrow \mathbf{U}$

$\mathrm{Dval}[\![I = E]\!]\rho = \rho[e/I]$

$\mathrm{Dval}[\![\Delta_0\ \underline{\mathrm{and}}\ \Delta_1]\!]\rho = \mathrm{Dval}[\![\Delta_0]\!]\rho + \mathrm{Dval}[\![\Delta_1]\!]\rho$

2.2 Semantics of me-too

Because of space limitations, only part of the semantics is given here.

Syntactic Domains

A ε Ato = Nml + Sym + {T,F,nil}	Atoms
I ε Ide	Identifiers
E ε Exp	Lispkit Expressions
S ε Set	Set Expressions
G ε Gex = Set + Exp	General Expressions (Set + Lispkit)
Δ ε Dec	Declarations

Abstract Syntax

```
G    →    E | S
E is defined as in 2.1
S    →    {} |{G,G} | S U S |{G:I ← S} |
          {G:I ← S;E} | if E then S else | let Δ S |
          letrec Δ S | lambda(I,I,...,I)S | S(G,G,...G)
Δ    →    I = G | Δ and Δ
```

Semantic Domains

These are as given above for Lispkit.

Semantic Functions

The function definitions given in 2.1 all still apply. In addition a new semant function Pval is required:

```
Pval:  Gex → E
        Pval⟦G⟧ = Eval⟦G⟧[
                 Eval⟦letrec
                       member = lambda(e I)(
                           if atom I
                           then quote F
                           else
                                if eq e (car I)
                                then quote T
                                else member e (cdr I))
                      member⟧[]
                           /member,
                 Eval⟦letrec
                       map = lambda(f I)(
                             if atom I
                             then quote nil
                             else cons (f (car I))(map f (cdr I)))
                       map⟧[]
                             /map,
Eval⟦letrec
        filter = lambda(f I)(
              if atom I
              then quote nil
              else
                    if f (car I)
                    then cons (car I) (filter f (cdr I))
                    else filter f (cdr I))
        filter⟧[]
              /filter]
```

Eval is extended to act on the syntactic category Set also. Its functionality is

$\text{Eval}: \text{Gex} \rightarrow \mathbf{U} \rightarrow \mathbf{E}$

$\text{Eval}[\![\{\}]\!]\rho = \text{inES(nil)}$

$\text{Eval}[\![\{G_0,G_1\}]\!]\rho = \text{Eval}[\![\underline{\text{cons}}\ (\text{Eval}[\![G_0]\!]\rho)(\underline{\text{cons}}\ (\text{Eval}[\![G_1]\!]\rho)(\underline{\text{quote}}\ \underline{\text{nil}}))]\!]\rho$

```
Eval⟦S₀ U S₁⟧ρ = Eval⟦
        letrec
              union = lambda(a,b)(
                    if atom a
                    then b
                    else
                          if member (car a) b
                          then union (cdr a) b
                          else union (cdr a) (cons (car a) b))
              union S₀ S₁⟧ρ
```

$\text{Eval}[\![\{G{:}I \leftarrow S\}]\!]\rho = \text{Eval}[\![\text{map}\ (\underline{\text{lambda}}\ (I)\ G)\ S]\!]\rho$

$\text{Eval}[\![\{G{:}I \leftarrow S;\ E\}]\!]\rho =$
$\quad \text{Eval}[\![\text{map}\ (\underline{\text{lambda}}\ (I)\ G)\ (\text{filter}\ (\underline{\text{lambda}}\ (I)\ E)\ S)]\!]\rho$

Eval applied to the constructs if E then S else S, let Δ S, letrec Δ S, lambda(I,I,...,I)S and S(G,G,...,G) gives the same results as the corresponding cases for Lisp, with S substituted for E where appropriate.

$\text{Dval}: \text{Dec} \rightarrow \mathbf{U} \rightarrow \mathbf{U}$

$\text{Dval}[\![I = G]\!]\rho = \rho[\,g/I]$

$\text{Dval}[\![\Delta_0\ \underline{\text{and}}\ \Delta_1]\!]\rho = \text{Dval}[\![\Delta_0]\!]\rho + \text{Dval}[\![\Delta_1]\!]\rho$

3 SET THEORY

The theory used is liberally adapted from [Beeson 85]. It has been chosen to provide an axiomatisation which includes individuals, that is to say objects which are not sets. It is felt that a theory intended for practical use in computing science should view numbers and other primitive data types in terms of their own independent semantics rather than, in the conventional mathematical way, as special kinds of sets.

The theory presented here goes some way towards this goal, though of the individual types allowed only the natural numbers are axiomatised. Besides this drawback it has other features unsatisfactory to the computer scientist. One line of development envisaged from this work is the formulation of a more suitable theory; in section 6 we anticipate some of its features.

The replacement axioms have been omitted, because the scheme is not true in the list model. However, since the syntax of **me too** suggests that some version of the replacement axioms is still needed, a scheme is used which substitutes the graph of a function for the arbitrary function formula in the standard scheme. This weaker rule is a theorem of the system. It is not given here.

Logic and Language

First-order predicate calculus with equality; a binary predicate ε; unary predicates N and S (for numbers and sets); constants 0 and $\emptyset$; unary function symbol **s**.

A. Axioms on Numbers and Sets

1. $\sim(N(x) \ \& \ S(x))$
2. $N(0) \ \& \ (\forall x)(N(x) \to N(\mathbf{s}(x)))$
3. $x \ \varepsilon \ y \to S(y)$

B. Number-Theoretic Axioms

1. $N(x) \to \mathbf{s}(x) \neq 0$
2. $N(x) \ \& \ N(y) \ \& \ \mathbf{s}(x)=\mathbf{s}(y) \to x = y$
3. $A(0) \ \& \ (\forall x)(N(x) \ \& \ A(x) \to A(\mathbf{s}(x))) \to (\forall z)(N(z) \to A(z))$

C. Set-Theoretic Axioms

1. Extensionality	$S(x) \ \& \ S(y) \to ((\forall z)(z \ \varepsilon \ x \equiv z \ \varepsilon \ y) \to x = y)$	
2. Empty set	$S(\emptyset) \ \& \ (\forall z)\sim(z \ \varepsilon \ \emptyset)$	
3. Pairing	$(\exists u)(x \ \varepsilon \ u \ \& \ y \ \varepsilon \ u)$	
4. Infinity	$(\exists u)(S(u) \ \& \ (\forall z)(z \ \varepsilon \ u \equiv N(z)))$	
5. Union	$(\exists u)(S(u) \ \& \ (\forall z)(z \ \varepsilon \ u \equiv (\exists y)(y \ \varepsilon \ x \ \& \ z \ \varepsilon \ y))$	
6. Separation	$(\exists u)(S(u) \ \& \ (\forall z)(z \ \varepsilon \ u \equiv z \ \varepsilon \ x \ \& \ A(z)))$	(u not free in A)
7. Power set	$(\exists u)(S(u) \ \& \ (\forall z)(z \ \varepsilon \ u \equiv S(z) \ \& \ (\forall y)(y \ \varepsilon \ z \to y \ \varepsilon \ x))$	
8. Foundation	$S(x) \ \& \ x \neq \emptyset \to (\exists u)(u \ \varepsilon \ x \ \& \ (\forall z)(z \ \varepsilon \ u \to \sim(z \ \varepsilon \ x)))$	

4 IMPLEMENTATION CORRECTNESS

The result of this section will guarantee that the answer produced by exectuing a **me too** model is the same as would be obtained by using the formal theory. It is demonstrated by defining (in 4.1) the desired interpretation for **me too** in terms of sets, and (in 4.2) an abstraction function from lists to sets. We then discuss the constraints which the standard semantics must satisfy to ensure that the diagram in section 1 commutes.

This condition, however, is not sufficient to guarantee the correctness of the implementation, as Eval and Abs could be incorrect in ways that cancel each other out. Such a situation would only come to light through subsequent extension of Eval to new syntactic constructs. The possibility can be eliminated, however, if it can be shown that the axioms of set theory are true in the list model, with equality interpreted by the equivalence relation induced by the many-to-one function Abs. The interpretation is outlined in 4.3.

4.1 Set Semantics of **me too**

Syntax

This is as given in 3.2

Semantic Domains

Ind	Individuals
Set	Sets
$\rho \in$ **Env** = Ide $\rightarrow$ **Obj**	Environments
Obj = **Ind** $\oplus$ **Set** $\oplus$ **Fun**	Objects (in the universe of sets)
Fun = $\mathbf{Obj}^n \rightarrow \mathbf{Obj}$	Functions

It is intended that individuals shall be identified with the denotations of Lispkit expressions evaluated by the function Eval.

Semantic Functions

Sval: Gex $\rightarrow$ **Env** $\rightarrow$ **Obj**

For clarity and reasons of space, the definitions given now omit injections and projections for sum domains, and the error cases resulting from incorrect typing.

$$\mathrm{Sval}[\![\{\}]\!]\rho = \emptyset$$

$$\mathrm{Sval}[\![E]\!]\rho = \mathrm{Eval}[\![E]\!]\rho$$

$$\mathrm{Sval}[\![\{G_0,G_1\}]\!]\rho = \{\mathrm{Sval}[\![G_0]\!]\rho,\mathrm{Sval}[\![G_1]\!]\rho\}$$

$$\mathrm{Sval}[\![S_0 \cup S_1]\!]\rho = \cup\{\mathrm{Sval}[\![S_0]\!]\rho,\mathrm{Sval}[\![S_1]\!]\rho\}$$

$$\mathrm{Sval}[\![\{G{:}I \leftarrow S\}]\!]\rho = \{x \mid (\exists y{:}\mathrm{Sval}[\![S]\!]\rho)\ (x = (\mathrm{Eval}[\![\underline{\mathrm{lambda}}\ (I)\ G]\!]\rho)y)\}$$

$$\mathrm{Sval}[\![\{G{:}I \leftarrow S;E\}]\!]\rho = \{x \mid (\exists y{:}\mathrm{Sval}[\![S]\!]\rho)(((\mathrm{Eval}[\![\underline{\mathrm{lambda}}\ (I)\ E]\!]\rho)y)\ \&\ x = (\mathrm{Eval}[\![\underline{\mathrm{lambda}}\ (I)\ G]\!]\rho)y)\}$$

The last two equations are true in the list model only if the functions $\mathrm{Eval}[\![\underline{\mathrm{lambda}}\ (I)\ G]\!]\rho$ are represented by their graphs. Unfortunately the syntax does not enforce this restriction.

Once again, the constructs if E then S else S, let Δ S, letrec Δ S, lambda(I,I,...,I)S and S(G,G,...,G) are evaluated in the same way as Eval evaluates them. For example,

$$\mathrm{Sval}[\![\underline{\mathrm{if}}\ E\ \underline{\mathrm{then}}\ S_0\ \underline{\mathrm{else}}\ S_1]\!]\rho = \mathrm{Eval}[\![E]\!]\rho \rightarrow \mathrm{Sval}[\![S_0]\!]\rho,\ \ \mathrm{Sval}[\![S_1]\!]\rho$$

The definition of **me too** therefore consists of the definition of Lisp together with the definition of the data type of sets. The same method will be used for other data types 'embedded' into Lispkit Lisp.

4.2 Requirements for Implementation Correctness

Some preliminary definitions are required. The many-to-one function

$\text{Abs}: \mathbf{S} \to \mathbf{Obj}$

which maps the list interpretation of a **me too** term to its set interpretation is derived from the definitions of Eval and Sval. It is therefore implementation-dependent. The definition can be written

```
Abs(x) =def isA(x)  →  x,
                       {Abs(y)|member_of(y,x)}
```

where for the current implementation **member_of** is defined by

```
member_of: S → S → T
member_of(x,y) =def
          isL(y) →
                    object_equals(x,drop(fst(outL(y))))
                    or
                    member_of(x,drop(snd(outL(y)))),
          false
```

object_equals is derived from the equivalence relation induced by Abs on **S**.

```
object_equals: S → T → T
object_equals(x,y) =def isL(x) and isL(y)  →  Abs(x) = Abs(y),
                                 isA(x) and isA(y) → individual_equals(x,y),
                                                     false
```

individual_equals depends on the definitions of the domains **N** and **Y**. Its exact definition is unimportant here.

The major requirement for the correctness of the implementation is that

(+) $\text{Abs}(\text{Eval}[\![T]\!]\rho) = \text{Sval}[\![T]\!]\rho$

should hold for every **me too** term T. This is proved by induction on the structure of **me too** terms. First, we require a lemma, whose proof will come later:

Lemma For any **me too** term T composed of subterms $T_o,\ldots,T_i,\ldots,T_n$, if the inductive hypothesis holds for each subterm, so that for each i

$$\text{Abs}(\text{Eval}[\![T_i]\!]\rho) = \text{Sval}[\![T_i]\!]\rho$$

then

$$(\forall x{:}S)(\textbf{member\_of}(x,\text{Eval}[\![T]\!]\rho) \equiv \text{Abs}(x) \in \text{Sval}[\![T]\!]\rho)$$

The motivation for this equation can be shown diagrammatically as

$$\begin{tikzcd}
& T\ (\text{a } \mathbf{me\ too} \text{ term}) \arrow[dl, "\mathrm{Eval}"'] \arrow[dr, "\mathrm{Sval}"] & \\
\mathrm{Eval}[\![T]\!]\rho \arrow[rr, dashed, leftrightarrow, "\mathrm{Abs}"] & & \mathrm{Sval}[\![T]\!]\rho \\
x \arrow[u, "\mathbf{member\_of}"] \arrow[rr, "\mathrm{Abs}"] & & \mathrm{Abs}(x) \arrow[u, "\varepsilon"'] \\
\underline{S} & & \underline{\mathbf{Obj}}
\end{tikzcd}$$

The broken line represents the property to be established, and the lemma is the result of a "diagram chase" round the necessary conditions to establish it.

Using the lemma, the proof of (+) is straightforward. For the base step, assume T represents an individual or the empty set:

Abs(Eval⟦T⟧ρ)	= Eval⟦T⟧ρ	...defn Abs
	= Sval⟦T⟧ρ	...defn Sval

For the inductive step

Abs(Eval⟦T⟧ρ)	={Abs(x)|**member_of**(x,Eval⟦T⟧ρ)}	...defn Abs
	={Abs(x)|Abs(x) ε Sval⟦T⟧ρ}	...by the lemma
	= Sval⟦T⟧ρ	

It remains to establish the lemma. This must be done by cases, one case for each **me too** operator. The inductive hypothesis will appear in these proofs, applied to the constituent terms of T. The fact that Abs induces an equivalence relation on **S** allows us to avoid using **Obj**, as we can adjust the formulae to involve only terms and relations in **S**. For example, in the case that T = {G_0,G_1}

member_of(x,Eval⟦{G_0,G_1}⟧ρ) ≡ Abs(x) ε Sval⟦{G_0,G_1}⟧ρ

≡ Abs(x) = Sval⟦G_0⟧ρ v Abs(x) = Sval⟦G_1⟧ρ	...defn. Sval
≡ Abs(x) = Abs(Eval⟦G_0⟧ρ) v Abs(x) = Abs(Eval⟦G_1⟧ρ)	...ind. hyp.
≡ **object_equals**(x,Eval⟦G_0⟧ρ) v **object_equals**(x,Eval⟦ G_1⟧ρ)	...defn **object_equals**

Constraints on the evaluation of the other **me too** constructs can be deduced in a similar way:

member_of(x,Eval ⟦E⟧ρ) ≡ **false**

member_of(x,Eval⟦S_0 U S_1⟧ρ) ≡
  member_of(x,Eval⟦S_0⟧ρ) v **member_of**(x,Eval⟦S_1⟧ρ)

member_of(x,Eval⟦{G:I ← S}⟧ρ) ≡
  (∃y:S)(**member_of**(y,Eval⟦S⟧ρ) & x = (Eval⟦<u>lambda</u> (I) G⟧ρ)y)

member_of(x,Eval⟦{G:I ← S;E}⟧ρ) ≡
  (∃y:S)(**member_of**(y,Eval ⟦S⟧ ρ) & (Eval⟦<u>lambda</u> (I) E ⟧ρ)y &
    x = (Eval⟦<u>lambda</u> (I) G⟧ρ)y)

There are no such constraints on the evaluation of the constructs <u>if</u> E <u>then</u> S <u>else</u> S, <u>let</u> Δ S, <u>letrec</u> Δ S, <u>lambda</u>(I,I,...,I)S and S(G,G,...,G), as their evaluation is independent of the semantics of the particular data type in use.

Work is in progress to prove the above equivalences using Cambridge LCF. For reasons of space, it is not reproduced here.

4.3 <u>Interpreting Set Theory in **S**</u>

Only an outline of the interpretation is given. The language sketched in section 3 is interpreted as follows: the domain of the interpretation is **S**, = is interpreted by **object_equals**, $\in$ by **member_of**,

N(x) by isA → (isN o outA), false

S(x) by **not** isA(x)

and ∅ is interpreted by nil. Because in this work we are primarily interested in the subtheory of sets, the symbol **s** is interpreted by addition of 1 in an idealised model of Lisp arithmetic.

Interpreted in this way, the axioms of section 3 are satisfied, except for the Foundation and Power set axioms. Formal proofs of satisfaction are being produced using Cambridge LCF; they are again omitted for reasons of space.

5 <u>SEMANTIC VARIATIONS IN RECURSIVE TYPE MODELS FOR SET THEORY</u>

As explained in section 4, a condition for the correctness of the implementation is that the domain **S** should form a model for the axioms of set theory, when equality is interpreted by the equivalence relation induced by Abs. Clearly, not all the axioms can be true in **S**, since **S** can model at most countably infinite sets. In fact, the semantics of **S** may not be defined only in order that it can function as a model for set theory; implementation considerations will intrude. Such considerations, for example, led to the current implementation containing no type scheme and using lazy evaluation throughout. This section examines the effect of such decisions by presenting a series of recursive type models corresponding to different typing and evaluation strategies. Regarding each as an interpretation of the language of sets, we state which of the axioms are true in the interpretation. It is hoped that this study will show a way, for future work, of clearly relating the execution semantics of computer languages with sets to the power of the set theory they support.

The standard representation of sets as lists suggests, for a model of the theory of section 3, variations on the domain constructions:

Object = Individual $\oplus$ **Set**

Set = **Nonempty** $\oplus$ {nil}

Nonempty = **Object** $\otimes$ **Set**

where Individual is a type variable whose instantiation is implementation-dependent. Implementations will also differ in the strictness of the evaluation of the components of **Nonempty**. In the current implementation, **Object** is represented by S, Individual is instantiated by **A**, **Set** is represented by the anonymous domain **L** $\oplus$ **X**, **Nonempty** is represented by **L**, and both the components of **Nonempty** are evaluated lazily. (It also contains the anomaly that **L** is defined as $\mathbf{S}_\perp \otimes \mathbf{S}_\perp$ instead of $\mathbf{S}_\perp \otimes (\mathbf{L} \oplus \mathbf{X})$, corresponding to the fact that in Lisp a dot expression can have any atom as its second component).

We first consider variations in the strictness of evaluation of the components of **Nonempty**. To provide domain theory semantics for lazy evaluation, it is necessary to treat, instead of the simple domains given above, isomorphic subdomains of a universal domain which also contains subdomains of infinite Cartesian products. (Details are given in [Cartwright 82]). It follows that if either component of **Nonempty** is lifted, the resulting type will contain infinite objects. The truth of the Axioms of Foundation and Infinity, interpreted in the type, depend on this. The Axiom of Foundation will be true in a type if and only if that type is strict in the first component, and the Axiom of Infinity can be true in a type only if that type is lazy in the second.

The second dimension of variation is the instantiation used for Individual. This can be **A**, or either of the types **Finite_S_Exp** or **Infinite_S_Exp**, corresponding to Lisp S-expressions evaluated respectively strictly or lazily. The only theoretical interest in this variation is in whether equality between individuals is decidable, which depends on whether or not **Infinite_S_Exp** is used. It is only necessary to consider using **A** to instantiate Individual because the current implementation lacks a type scheme. To ensure that all objects can be unambiguously analysed, sum constructions must therefore use disjoint domains. In fact the semantics of the current implementation are rather unclear, since Individual is in effect instantiated by **Infinite_S_Exp**, which is isomorphic to **Object**.

Four types (labelled A – D) with different combinations of these factors have been considered. They are summarised in the table below, which is followed by brief discussion of each type. **Fse** and **Ise** are abbreviations for **Finite_S_Exp** and **Infinite_S_Exp** respectively.

TABLE 1

	Theory			
	A	B	C	D
Evaluation Rule for **Object**	Strict	Strict	Strict	Lazy
Evaluation Rule for **Set**	Strict	Strict	Lazy	Lazy
Instantiation of Individual	**A**	**Fse**	**Ise**	**Ise**

Theories A and B are similar from a set-theoretic point of view, both providing models for a finite theory, that is for all the axioms of section 3 except the Infinity Axiom. The difference between them is in their ease of implementation: theory A is included only because it can be implemented without a type scheme.

Theory C is a model for all the axioms except the Power set axiom. Theory D is similar to Theory C except that the lazy evaluation of **Object** renders the interpreted Foundation axiom false, raising the potentially interesting possibility of modelling non-well-founded sets. The primary motivation for studying this type, however, is that it corresponds to the semantics of the current implementation if a type scheme were added.

6 CONCLUSIONS

6.1 **me too**

One objective of the work has been to examine how far the assumption of the mathematical correctness of **me too** is actually justified. In general, it has been shown that the language could provide a dependable model for axiomatic set theory if certain changes were made. The defects that it has lie in two areas:

- The absence of a type scheme means that the 'best' semantics that can be given to the language prevents the use of S-expressions as individuals.
- The syntax of the **me too** replacement scheme permits the user to 'import' an arbitrary S-expression into the model of **me too**, and then to interpret it as a set. This is the reason why the interpreted Foundation Axiom is not true in the current system, although no **me too** operation can create a 'set' that falsifies it. The simplest remedy would be to allow only the graphs of functions (for example, **me too** finite functions) in the replacement formulae.

6.2 The investigation

The chief result obtained has been to establish the conditions for the correctness of the implementation. Moreover, the method used will provide an economical way to show the correctness of the implementation of any new data type embedded into Lisp.

The main theoretical difficulty encountered has been the problem of determining a suitable version of set theory. There seems to have been no attempt to deal systematically with individuals in an axiomatic framework, at least until [Beeson 85]. As pointed out in section 3, this deficiency is generally a problem for computer scientists hoping to implement systems modelling set theory. Further, a comprehensively useful theory would probably employ constructive logic and would certainly provide "an elaborate system of" terms (Beeson). Important work remains to be done in this area.

If such a theory had been used in this work, it would have been possible to devise a term model for **me too** which would also serve directly as a model for the axiomatic theory. This approach is attractive because it would make the language-theory relationship clear at the syntactic level, and it would establish immediately the expressive power of **me too** relative to the theory. The latter result has not yet been determined by the present method. However, checking the correctness of the implementation would involve comparing the term model with the execution semantics of the language, and this might well generate problems comparable with those encountered using the present approach.

The use of a proof assistant has greatly improved the rigour of the work. It has allowed proofs to be performed that would never have been attempted by hand. Learning to use Cambridge LCF is hard, however. For routine use, automatic theorem-provers and proof assistants need to be easier to work with, so that they do not require a user to devote weeks of work to acquiring specialist knowledge about them.

In conclusion, the work shows that a complete formal analysis of real-life software engineering tools is indeed possible, and that the design of such tools will benefit from it, if it is done at the outset. But more experiments along these lines are necessary before such anlysis becomes sufficiently cheap and easy to be done as a normal part of the design process.

7 ACKNOWLEDGEMENTS

I am grateful to Roy Dyckhoff, Alan Hamilton, Simon Jones and Muffy Thomas for their patience and good advice in many discussions. Peter Aczel, Cliff Jones and a referee made helpful comments on an earlier draft of this paper.

8 REFERENCES

[Beeson 85] Beeson M.J., Foundations of Constructive Mathematics, Springer-Verlag, 1985

[Cartwright 82] Cartwright R. and Donahue J., The Semantics of Lazy (and Industrious) Evaluation, ACM Symposium on Lisp and Functional Programming (1982), pp 253-264

[Henderson 80] Henderson P., Functional Programming: Application and Implementation, Prentice-Hall International, 1980

[Henderson 84] Henderson P., **me too** - a Language for Software Specification and Model Building, Report FPN-9, Department of Computing Science, University of Stirling

[Paulson 83] Paulson L., The Revised Logic PPLAMBDA: A Reference Manual, Technical Report No. 36, Computer Laboratory, University of Cambridge, 1983

[Stoy 77] Stoy J.E., Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, MIT Press, 1977