On Storage Management for Advanced Programming Languages

by

Jack B. Dennis

This work was supported by the National Science Foundation under research grant GJ-34671.

October 1974

(Revised November 1, 1974)
On Storage Management for Advanced Programming Languages

Jack B. Dennis
Project MAC
Massachusetts Institute of Technology
545 Main Street
Cambridge, Massachusetts 02139
Tel: 617-253-6856

Abstract: Advanced programming languages developed with guidance from considerations of good program structure and proof of correctness have generally been implemented using the memory management scheme known as mark/sweep garbage collection. Such implementations are unlikely to be acceptable in practical computer systems intended to support the modular construction of large programs. Using an illustrative language designed with consideration to good program structure and clean semantics, we demonstrate the feasibility of using the reference count memory management technique in its implementation. The language is defined by its functional semantics, and its implementation is modelled by a formal interpreter based on acyclic directed graphs.

*This work was supported by the National Science Foundation under research grant GJ-34671.
1. Introduction

The software problem [18] has inspired increasing interest in the implications of structured programming concepts on the design of programming languages. The field of language design itself has moved into an era of self-criticism and evaluation rather than undisciplined innovation. Familiar programming constructs -- the goto, assignment to a formal parameter, global variables -- have been found wanting with respect to the construction of easily understood and certifiable programs. Hoare [9] has shown how the merit of programming constructs may be evaluated in terms of an axiomatic semantics: What is the method of program proof that justifies uses of a construct? Reynolds has argued* that a construct should not be introduced into a programming language unless the corresponding proof rule is understood.

The concept of levels of abstraction in composing programs has led to interest in the structure and communication of data in programming, and a new viewpoint on user defined data types that brings together and encapsulates the specification of object representation and the implementation of the operations of the abstract data type [11, 21].

Much criticism of programming constructs is concerned with independence of program parts and the programmer's ability to separate design decisions so as to minimize interaction of their consequences [15]. This is the essence of modular programming -- the building of programs from independently written components. Modular programming has far-reaching implications for the structure of computer systems: A computer system that supports modular programming must provide a uniform scheme for representing and accessing information. If the benefits of modularity are to be realized for very large programs, the uniformity must extend over information shared at all levels of the computer's memory hierarchy. Moreover, if modularity is to benefit a community of users, then the uniformity must hold over all programs written by the users. One interesting proposal for support of modular programming is the Binding Model developed by Henderson [8].

Advanced programming languages such as Pascal [10], PAL [6], Gedanken [16], and ECL [2], that have seriously sought to have clean semantics, call for

*In a presentation at the Symposium on the High Cost of Programming [18].
dynamic allocation and reallocation of storage to objects created during the progress of computation. The usual approach to implementation uses the memory management scheme known as mark/sweep garbage collection [7]. There are two serious problems with garbage collection that make it unlikely to be acceptable in a practical computer system designed to support modular programming: The mark/sweep scheme requires that all computation be stopped occasionally so all accessible information may be identified and storage occupied by inaccessible information released. This requirement is, of course, intolerable in any system intended to meet real time deadlines. The second problem is that no efficient mark/sweep scheme has been devised for storage management in a memory hierarchy.

In this paper we show that by restricting a programming language to constructs that are powerful, yet are not in conflict with considerations of good program structure and program verification, mark/sweep storage management can be avoided. We define a language L and discuss its functional semantics. By constructing interpreter using a directed graph formalism, we show that memory management by reference count suffices for the implementation of L.
2. A Language With Clean Semantics

A specific programming language L has been developed for an exposition on the methods of formal semantics [4], and illustrates the level of generality that can be achieved while omitting constructs having undesirable properties with respect to program structure and proof of correctness.

In choosing constructs for inclusion in L we have ignored many aspects such as input/output, precision of numerical computations, which would detract from our main purpose. Also we have omitted some syntactic sugar (for example, multiple arguments for procedures) merely to simplify the task.

We are concerned with the semantics of L— not its concrete syntax — and it suffices to present the structure of L by giving an abstract syntax for a class of abstract objects that correspond to the programs of L. We have adapted McCarthy's approach [14], as extended by the work of the IBM Vienna Laboratory [12]. The semantics of L are given by defining semantic functions, following the style of Scott and Strachey [17], that assign mathematical meaning to each construct of the language.

The constructs of L include both expression forms and command forms; the language shows how the two classes of linguistic constructs can combine harmoniously.

2.1. Objects

The general class of abstract objects is defined by the syntax

\[
\begin{align*}
\text{Obj} & := \emptyset | \text{elem} | \text{compound} & \quad \text{[the set of objects]} \\
\text{Elem} & := \text{truth} | \text{int} | \text{string} & \quad \text{[the elementary objects]} \\
\text{Truth} & := \text{true} | \text{false} & \quad \text{[the truth values]} \\
\text{Int} & := 0 | +1 | -1 | \ldots & \quad \text{[the integers]} \\
\text{String} & := < > | 'a' | 'b' | \ldots | 'aa' | \ldots & \quad \text{[strings on the alphabet of letters and digits]}
\end{align*}
\]

The first line of this syntax is intended to be read thus: An entity is an element of the set \text{Obj} if and only if the entity is the empty set \emptyset, or is an element of one of the sets \text{Elem} or \text{Compound}. The names of syntactic classes start with a capital letter; the uncapitalized name denotes an element of the class.
A compound object (an element of Compound) is a finite set of ordered pairs written
\[ \langle \text{sel}_1 : \text{obj}_1, \ldots, \text{sel}_k : \text{obj}_k \rangle \]
where
\[ \text{Sel} ::= \text{int} | \text{string} \]
is the set of selectors.

Objects may be thought of as trees with arcs labelled by selectors and with elementary objects associated with their leaf nodes (but not necessarily all leaf nodes).

We will use a scheme attributed to Strachey to make sure the syntactic form of an object is evident from its denotation. To illustrate, the class of compound objects representing the let expressions of L is defined by the form rule
\[ \text{Letexp} ::= \text{let} \ \text{id} = \text{exp}_0 \ \text{in} \ \text{exp}_1 \]
which means, by convention, that an object \text{letexp} in the class \text{Letexp} has the form
\[ \text{letexp} = \langle \text{'let'}: \text{let} \ * = * \ \text{in} \ *, \ \text{'id'}: \text{id}, \ \text{'exp'}_0: \text{exp}_0, \ \text{'exp'}_1: \text{exp}_1 \rangle \]
The \text{'let'}-component of this object is a tag that identifies the syntactic form satisfied by the object. Regarded as a tree, a let expression looks like Figure 1.

2.2. The Expressions of L

The abstract syntax for the expression forms of L is:

\[ \text{Exp} ::= \text{id} \]
\[ \text{cons} \]
\[ \text{upop} \ \text{exp} \]
\[ \text{exp}_1 \ \text{bop} \ \text{exp}_2 \]
\[ \text{let} \ \text{id} = \text{exp}_0 \ \text{in} \ \text{exp}_1 \]
\[ \text{if} \ \text{exp}_0 \ \text{then} \ \text{exp}_1 \ \text{else} \ \text{exp}_2 \]
\[ \text{iterate} \ \text{id} \ \text{from} \ \text{exp}_0 \ \text{by} \ \text{exp}_1 \ \text{while} \ \text{exp}_2 \]
\[ \text{proc}(\text{id}_1): \ \text{exp} \]
\[ \text{rec} \ \text{id}_0(\text{id}_1): \ \text{exp} \]
\[ \text{apply} \ \text{exp}_1(\text{exp}_2) \]
\[ \text{cmd} \ \text{res} \ \text{exp} \]

identifier
constant
unary operation
binary operation
let expression
conditional
iteration
procedure declaration
recursive procedure
application
result of command
Figure 1. An object in the syntactic class of let expressions.
The identifiers of \( L \) are strings:
\[
\text{Id ::= string}
\]
the constants are values:
\[
\text{Cons ::= value}
\]
and the values are typed:
\[
\text{Value ::= Boolean truth integer int string string}
\]
The sets \( \text{Hop} \) and \( \text{Bop} \) contain elements denoting unary and binary operations on
the values of \( L \). Discussion of structured values (for example, arrays and
records) is deferred to Section 7.

The meaning of an expression in \( L \) is given in terms of a mathematical
domain of values \( \mathbb{V} \), which is the "reflexive domain" of functional
semantics [17]:
\[
\mathbb{V} = \mathbb{E} + [\mathbb{V} \rightarrow \mathbb{V}]
\]
where
\[
\mathbb{E} = \mathbb{T} + \mathbb{N} + \mathbb{Q}
\]
\[
\mathbb{T} = \{ L, \text{true}, \text{false}, T \} \quad \text{[the domain of truth values]}
\]
\[
\mathbb{N} = \{ L, 0, +1, -1, \ldots, T \} \quad \text{[the domain of integers]}
\]
\[
\mathbb{Q} = \{ L, \langle \rangle, 'a', 'b', \ldots, 'aa', \ldots, T \} \quad \text{[the domain of quotations]}
\]
Elements of \( \mathbb{V} \) will be denoted by \( \alpha, \beta, \varphi \). For the semantics of \( L \) we take
\( L \) ("bottom") to be the meaning of nonterminating computations and \( T \) ("top") to be the meaning of expressions that fail to produce a normal value due to
mismatch of operator and operand (data type error), or absence of identifier
binding.

We assume a semantic function \( \mathcal{K} \) that maps the syntactic values of \( L \) into
the mathematical domain:
\[
\mathcal{K} : \text{Value} \rightarrow \mathbb{V} \text{ in } \mathbb{V}
\]
but we shall not further study its nature.

For an expression to have meaning, its identifiers must be bound to values;
this binding is given by a member \( \rho \) of a class of functions \( \mathbb{E} \) called environments:
\[
\rho \in \mathbb{E}, \quad \mathbb{E} = [\text{Id} \rightarrow \mathbb{V}]
\]
The meaning of an expression is given by the semantic function

\[ \mathcal{E} : \text{Exp} \rightarrow [\mathcal{E} \rightarrow \mathcal{V}] \]

that associates with each expression a mapping of environments (identifier bindings) into values. This function is conveniently specified by a kind of mathematical case statement:

\[
\mathcal{E}[[\text{exp}]] \rho = \begin{cases} 
\rho(\text{id}) & (\text{exp} = \langle \text{id} \rangle) \\
\mathcal{E}[[\text{cons}]] & (\text{exp} = \langle \text{cons} \rangle) \\
\vdots & 
\end{cases}
\]

This is usually abbreviated to

\[
\mathcal{E}[[\text{id}]] \longrightarrow \rho(\text{id}) \\
\mathcal{E}[[\text{cons}]] \longrightarrow \mathcal{K}[[\text{cons}]] \\
\vdots
\]

The first six forms of expression are familiar constructs -- they have their familiar meanings in L -- and we omit discussion of them. For iteration expressions, it is convenient to give their meaning in terms of translation into command form, which is done in Section 4.

The meaning of a procedure value in L is a function in \([\mathcal{V} \rightarrow \mathcal{V}]\) determined by the identifier bindings holding when the procedure value is created. We give the semantic equation for recursive procedures, which requires use of the fixed point operator \(Y\) of functional semantic theory:

\[
\mathcal{E}[[\text{rec } \text{id}_0(\text{id}_1) : \text{exp}]] \cdot \sigma = Y[\lambda \Phi, (\lambda \rho, \mathcal{E}[[\text{exp}]] \sigma) \in \mathcal{V}] \\
\rho' = \lambda \text{id}_0. (\text{id}_0 = \text{id}_0 \rightarrow \Phi, (\text{id}_0 = \text{id}_1 \rightarrow \sigma, \rho(\text{id}))]
\]

The argument of the fixed point operator is the function determined by the mapping of functions into functions that maps the function \(\Phi\) into the function determined by the procedure body \(\text{exp}\) for the environment \(\rho'\) in which \(\text{id}_0\) is bound to \(\Phi\) and the formal argument \(\text{id}_1\) is bound to the value \(\sigma\). The meaning of the recursive procedure is the function \(\Theta\) which, if bound to \(\text{id}_0\), yields the same function \(\Theta\) as the meaning of the procedure body.
The meaning of procedure application is given by

\[ E[apply \ exp_1 (exp_2)] = (E[exp_1] \rho) \nu \rightarrow (E[exp_2] \rho) \]

which says that values are obtained for both expressions, and the value of \( \exp_1 \) (which must be a function in \( \nu \rightarrow \nu \)) is applied to the value of \( \exp_2 \).

Finally, we have

\[ E[cmd \ join \ exp] \rho = E[exp](E[cmd] \rho) \]

which states that \( \exp \) is evaluated for the environment \( \rho' \) determined by the meaning of \( cmd \). Note that \( \rho' \) is created only for evaluation of \( \exp \) and can make no other contribution to the meaning of a program.

2.3. The Commands of \( \mathbf{L} \)

The abstract syntax for the command forms of \( \mathbf{L} \) is:

\[
\begin{align*}
\text{Cmd} & : = (\quad) & \text{the empty command} \\
& \mid \text{cmd}_1; \text{cmd}_2 & \text{sequencing} \\
& \mid \text{id} \ := \ \exp & \text{assignment} \\
& \mid \text{if } \exp \ \text{then } \text{cmd}_1 \ \text{else } \text{cmd}_2 & \text{conditional} \\
& \mid \text{while } \exp \ \text{do } \text{cmd} & \text{iteration}
\end{align*}
\]

The meaning of a command is a mapping of environments into environments:

\[ \mathcal{C} : \text{Cmd} \rightarrow [E \rightarrow E] \]

For example, the meaning of assignment is given by

\[ \mathcal{C}[\text{id}_0 := \exp] \rho = \lambda \text{id}.(\text{id} = \text{id}_0) \rightarrow E[\exp] \rho, \rho(\text{id}) \]

For the \textit{while} command, a fixed point must be found:

\[ \mathcal{C}[\text{while } \exp \ \text{do } \text{cmd}] \rho = \\
\lambda \text{\textit{U}}: \lambda \rho'.((E[\exp] \rho', \mathbf{E}) \rightarrow (\mathcal{C}[\text{cmd}] \rho'), \rho')) \rho \\
in \text{which } \lambda : E \rightarrow E.\]
The design of L follows many recommendations for clean semantics. The control structures of L are in correspondence with straightforward methods of proof — case analysis and induction; procedures denote functions — there are no side effects; consequently, the meaning of expressions is independent of the order of evaluation of subexpressions. As a result, a simple set of axioms for L may be derived from the semantic functions $\mathcal{E}$ and $\mathcal{C}$. (See [4] for details.)

The assignment command of L does not cause difficulty because its effect is limited to the result expression within which it occurs. The environment defined by a sequence of commands is used only for evaluation of the result expression, and does not otherwise influence the course of computation. The language has "free variables," but they can only transmit values into a procedure application because assignment merely creates a local binding of the identifier that temporarily occludes the external value.

3. Management of Storage for L

The function of a storage management scheme in a language implementation is to recover for reallocation space occupied by storage structures abandoned during the progress of computation and to which no future access is possible. The technique of mark/sweep garbage collection [7, 13] is used when the language implementer is unable to devise a scheme for recognizing the computational step at which a storage structure becomes inaccessible, and is therefore a candidate for reclamation of storage cells. Inherent in the use of mark/sweep garbage collection is the gradual accumulation of memory locations occupied by inaccessible information, the number of such locations being unknown to the storage management mechanism.

Garbage collection is expensive because every location in a program's address space occupied by accessible information must be touched for marking. There is a tradeoff between infrequent garbage collection which allows the number of occupied but inaccessible locations to become large — wasting memory, and frequent garbage collection which wastes processing power in marking accessible locations.

The desirable ability to share procedures and data structures among the programs of different users requires that all programs operate within the same address space and utilize a common storage management mechanism. If
mark/sweep garbage collection is used by the implementer of a computer system intended to support modular programming, then all users would share the problems of the scheme -- its inefficiency and the need to interrupt all computation at unpredictable times to reclaim storage cells. This would be intolerable in systems intended to meet real-time deadlines.

Another problem with mark/sweep garbage collection is that efficient realizations of the scheme for physical hierarchies of storage devices have not been found.

Our main objective is to show that the "reference count" storage management scheme [7, 20] is applicable to an efficient implementation of the language L. By an "efficient implementation" we mean that the storage cells used to hold values defined during execution of L programs are released once the values become inaccessible.*

3.1. Memory Management by Reference Count

In general, language implementations allocate storage in units called elements which contain elementary (unstructured) values and pointers that uniquely identify other elements. In using the reference count scheme, each element includes a count of the number of instances of pointers to the element located in other elements and in execution data structures (activation records or stack frames). The reference count is incremented for each instance of a pointer to the element created during program execution, and is decremented whenever an instance of a pointer to the element is deleted (by termination of a procedure activation and release of its stack frame, or by release of an element containing an instance of the pointer). When a reference count becomes zero, there are no pointers left in other elements or execution structures and the element can be deleted and the storage cells it occupies reclaimed. If cyclic chains of reference are present, groups of elements can become inaccessible while each element in the group retains a nonzero reference count. Thus the reference count scheme fails to be generally applicable. Nevertheless, if no cyclic reference chains are possible, it is usually possible to

---

*Implicit in this statement is a schematic model of L programs in which the primitive operations are replaced by uninterpreted operation letters and it is assumed that every path of control flow is possible for some choice of initial environment and meanings of the operation letters.
arrange that reference counts become zero when and only when the element
becomes inaccessible.

We will show that the reference count scheme is applicable to the lan-
guage L by presenting a formal interpreter in which the states are acyclic
directed graphs called s-graphs. The s-graphs are a simple abstraction of
storage structures composed of linked elements.

4. The Kernel-Language K

It is possible to give an interpreter for L that establishes our claim.
However, a much simpler interpreter suffices if we first show how L can be
translated into a subset of itself -- the kernel language K. The kernel lan-
guage embodies three simplifications of L:

1. Conditional, iteration and let expressions are omitted.
2. There is no nesting of expressions.
3. The test expressions in conditional and iteration commands are re-
  stricted to identifiers.

Abstract Syntax of K:

\[
\begin{align*}
\text{Exp} & ::= \text{id} | \text{cons} | \text{uop id} | \text{id}_1 \text{ bop id}_2 | \\
& \quad \text{proc(id}_1\text{): exp} | \\
& \quad \text{rec id}_1(id_1)\text{: exp} | \\
& \quad \text{apply id}_1(id_2) | \\
& \quad \text{cmd res exp} \\
\text{Cmd} & ::= ( \text{cmd}_1 ; \text{cmd}_2 | \\
& \quad \text{id} \text{: exp} | \\
& \quad \text{if id then cmd}_1 \text{ else cmd}_2 | \\
& \quad \text{while id}_1 \text{ do cmd}
\end{align*}
\]
4. A translation from L to K

The translation of programs in L to programs in K is accomplished by
a sequence of transformations each of which preserves meaning according to
the functional semantics of L. In some of the rules of transformation given
below, it is necessary to introduce one or two new identifiers. This is
indicated by writing \( id = \text{Newid} \). In Rule 4 it is essential that \( id_1 \neq id_2 \).
Otherwise, the choice of identifier does not matter except in Rule 5 where
the choice is explicitly restricted.

Translation rules 1, 2 and 3 replace expression forms with corresponding
command forms:

**Rule 1:** let expressions

\[
\text{let } id = \text{exp}_0 \text{ in } \text{exp}_1 \\
\text{becomes } \langle \langle id := \text{exp}_0 \rangle \rangle \text{ res } \text{exp}_1
\]

**Rule 2:** Conditional expressions

\[
\text{if } \text{exp}_0 \ \text{then } \text{exp}_1 \ \text{else } \text{exp}_2 \\
\text{becomes } \langle \langle \text{if } \text{exp}_0 \ \text{then } \langle \langle id := \text{exp}_1 \rangle \rangle \\
\text{else } \langle \langle id := \text{exp}_2 \rangle \rangle \rangle \text{ res } id
\]

where \( id = \text{Newid} \)

**Rule 3:** Iteration expressions

\[
\text{iterate } id \ \text{from } \text{exp}_0 \ \text{by } \text{exp}_1 \ \text{while } \text{exp}_2 \\
\text{becomes } \langle \langle \langle id := \text{exp}_0 \rangle \rangle ; \langle \langle \text{while } \text{exp}_2 \ \text{do} \langle \langle id := \text{exp}_1 \rangle \rangle \rangle \rangle \text{ res } id
\]

Rule 4 is used to substitute assignments with simple right-hand sides for
nested expressions.

**Rule 4:** Operations and Application

a. \( \text{uop exp} \)

\[ \text{becomes } \langle \langle id := \text{exp} \rangle \rangle \text{ res } \langle \langle \text{uop } id \rangle \rangle \]

b. \( \text{exp}_1 \ \text{bop } \text{exp}_2 \)

\[ \text{becomes } \langle \langle id_1 := \text{exp}_1 \rangle \rangle ; \langle \langle id_2 := \text{exp}_2 \rangle \rangle \text{ res } \langle \langle \text{bop } id_1 \rangle \rangle \langle \langle id_2 \rangle \rangle \]

c. \( \text{apply } \text{exp}_1 (\text{exp}_2) \)

\[ \text{becomes } \langle \langle id_1 := \text{exp}_1 \rangle \rangle ; \langle \langle id_2 := \text{exp}_2 \rangle \rangle \text{ res } \langle \langle \text{apply } id_1 (id_2) \rangle \rangle \]

where \( id = \text{Newid} \)

\( id_1 = \text{Newid} \)

\( id_2 = \text{Newid} \)
Rule 5 transforms conditional and iteration commands so the expression tested is an identifier.

Rule 5: Transformation of test expressions in conditional and iteration commands.

a. if \( \exp \) then \( \text{cmd}_1 \) else \( \text{cmd}_2 \)
   becomes \( <\text{id} := \exp; \text{if id then cmd}_1 \text{ else cmd}_2> \)

b. while \( \exp \) do \( \text{cmd} \)
   becomes \( <\text{id} := \exp; \text{while id do}<\text{cmd}; \text{id} := \exp> > \)

where \( \text{id} = \text{NewId} \) occurs neither in the command sequence containing the conditional or iteration command, nor in its result expression.

After making all possible applications of rules 1 - 5, programs satisfy the syntax of \( K \). The process terminates because each application of a rule removes an instance of a syntactic form allowed in \( L \) but not \( K \), without introducing any new instances of such forms. It is easy to show that the translation rules preserve the meaning of the expressions of \( L \) using the semantic equations of \( L \) -- a task we leave to the reader. However, the meaning of a command is not strictly preserved due to the possibility of extra identifiers having values in the environment produced by the translated command.

5: The Interpretive Normalism

We wish to use a formal model for the execution of programs in \( K \) from which claims can be made about implementations of \( K \), and hence also implementations of \( L \). For this purpose we need a formalism for constructing an interpreter for \( K \) that fairly represents considerations of storage management.

Two properties are important: The interpreter must not operate by making copies of information structures that may be arbitrarily large; and there must be a clear relationship between the state transitions of the interpreter and the operation of the storage management mechanisms of an implementation -- specifically, the reference count scheme.

The interpreters developed by the Vienna group [12] do not provide this combination of traits. The use of objects as interpreter states forces use of explicitly generated unique names to identify shared objects as components of a global structure. This scheme fails to provide any easy way of
determining whether a shared object is accessible. To identify inaccessible
shared objects the designer of an interpreter must in effect implement mark/
sweep garbage collection.

Instead of objects, we use directed graphs as interpreter states.
Directed graphs closely model the linked storage structures often used in
the implementation of advanced programming languages. The possibility of
different paths leading to the same node provides a direct way of modelling
sharing, and restricting interpreter states to acyclic graphs makes it simple
to demonstrate the sufficiency of reference count storage management. A simi-
lar formalism has been used by Ellis [5] in a study of the semantics of data
structures and sharing in a number of contemporary programming languages.

5.1. The Directed Graph Formalism

The directed graph formalism is developed from three sets of entities:

\[ \text{Elem} ::= \text{truth} | \text{int} | \text{string} \]
the elementary values

\[ \text{Sel} ::= \text{int} | \text{string} \]
the selectors

\[ \text{Unid}: \text{a denumerable set of unique identifiers} \]

A graph is a triple

\[ (\text{nodes}, \text{arcs}, \text{val}) \]

where

\[ \text{nodes} \subseteq \text{Unid} \]
\[ \text{arcs} \subseteq \text{nodes} \times \text{Sel} \times \text{nodes} \]
\[ \text{val} \subseteq \text{nodes} \times \text{Elem} \]

and represents in the usual way a directed graph in which the arcs are labelled
with selectors and the valuation \text{val} associates elementary values with certain
nodes.

We require that the set of arcs originating at each node \( \alpha \) be distinct

\[ (\alpha, \text{sel}_1, \beta_1) \in \text{arcs} \]
\[ (\alpha, \text{sel}_2, \beta_2) \in \text{arcs} \]
and \( \beta_1 \neq \beta_2 \) implies \( \text{sel}_1 \neq \text{sel}_2 \)

and that \text{val} associate at most one value with any node

\[ (\alpha, \text{elem}_1) \in \text{val} \]
\[ (\alpha, \text{elem}_2) \in \text{val} \]
\[ \text{implies } \text{elem}_1 = \text{elem}_2 \]
For representing interpreter states we use a restricted class of graphs called s-graphs:

1. An s-graph contains no closed paths (cycles).
2. The valuation of an s-graph \( \gamma \) associates elementary values only with leaf nodes of \( \gamma \).

Any node \( \alpha \) of an s-graph \( \gamma = (\text{nodes}, \text{arcs}, \text{val}) \) is said to denote the object \( \text{Object}(\alpha, \gamma) \) according to the following inductive rule:

Let \( \sigma \) be

\[
\{ \text{sel} | (\alpha, \text{sel}, \beta) \in \text{arcs} \text{ for some } \beta \in \text{nodes} \}
\]

and suppose

\[
\sigma = \{ \text{sel}_1, \ldots, \text{sel}_k \}
\]

If \( \sigma \) is not empty, then

\[
\text{Object}(\alpha, \gamma) = <\text{sel}_1; \text{obj}_1, \ldots, \text{sel}_k; \text{obj}_k>
\]

where \( \text{obj}_i = \text{Object}(\beta_i, \gamma) \) and \( \beta_i \) is the unique node of \( \gamma \) such that \((\alpha, \text{sel}_i, \beta_i) \in \text{arcs} \). If \( \sigma \) is empty and \((\alpha, \text{elem}) \in \text{val}\), then

\[
\text{Object}(\alpha, \gamma) = \text{elem}
\]

Otherwise

\[
\text{Object}(\alpha, \gamma) = \emptyset
\]

In words, this simply states that \( \text{Object}(\alpha, \gamma) \) contains paths to its leaves that correspond (in the sense of identical selector labelling of the arcs) to paths in \( \gamma \) originating at \( \alpha \) and terminating on leaves of \( \gamma \).

The states of an interpreter expressed in the formalism are pairs \((\rho, \gamma)\) where \( \rho \) is the unique root node of the s-graph \( \gamma \); that is, each node of \( \gamma \) is accessible by some directed path from node \( \rho \).

An interpreter specifies for each state \((\rho, \gamma)\) the construction of a new state \((\rho', \gamma')\) by a series of steps that modify the s-graph:

\[
\gamma_0 \rightarrow \gamma_1 \rightarrow \cdots \rightarrow \gamma_k
\]

Each step may access components of the current s-graph, choose between alternatives according to the outcome of a test, or add nodes and arcs to the s-graph. The final step produces the s-graph \( \gamma_k \) and a node \( \rho' \) of \( \gamma_k \) that defines the new interpreter state \((\rho', \gamma')\). The s-graph \( \gamma' \) of the new state is obtained from \( \gamma_k \) by deleting the old root node \( \rho \) and all nodes and arcs inaccessible from the new root node \( \rho' \).
Each state \((p, \gamma)\) determines the object \(\text{Object}(p, \gamma)\) denoted by the root node \(p\). Hence an interpreter may also be regarded as defining a function

\[
\text{Interp}: \text{State} \rightarrow \text{State}
\]

where \(\text{State}\) is a class of abstract objects that correspond to interpreter states. This class of objects may be defined by giving an abstract syntax, as we shall give below for the kernel language \(K\).

An interpreter is written according to the concrete syntax given in Figure 2. The effect of each construct is as follows:

- \(<\text{name}>\) names are bound to values which are either nodes of an s-graph or elementary values.

- \(<\text{selector}>\) yields a value which is a selector (an element of \(\text{Sel}\)).

- \(<\text{function name}>\)(<name>) = <construction> specifies, for any state \((p, \gamma)\), the successor state \((p', \gamma')\). The <construction> is performed with <name> bound to \(p\) yielding a transformed s-graph \(\gamma^*\) and the root node \(p'\) of the next state. The s-graph \(\gamma'\) is obtained from \(\gamma^*\) by deleting \(p\) and nodes inaccessible from \(p'\).

- <construction> builds the s-graph of the next state by adding nodes and arcs to the current s-graph. A <construction> has a value which is the node \(\beta\) that denotes the object whose representation has been constructed.

We consider in turn the six forms of a <construction>: 
\[ \text{<definition>} ::= \text{<function name>} (\text{<name>}) = \text{<construction>} \]

\[ \text{<construct\_expr>} ::= \text{<name>} | \]
\[ \quad \text{<elementary value>} | \]
\[ \quad \text{Select}(\text{<name>}, \text{<selector>}) | \]
\[ \quad \text{Append}(\text{<selector>}:\text{<construction}_1) \text{ to } \text{<construction}_2) \]
\[ \quad \llbracket \text{<constructive denotation>} \rrbracket \]
\[ \quad \{ \text{<test>} \rightarrow \text{<construction}_1, \text{<construction}_2} \} \]

\[ \text{<selector>} ::= \text{<name>} | \text{<integer>} | \text{<string>} \]

\[ \text{<constructive denotation>} ::= \text{<pattern>} | \]
\[ \quad \text{<pattern>}{\text{<construction>}{\text{<constructive denotation>}}} \]

\[ \text{<test>} ::= \text{<name>} \]
\[ \quad \text{Has}(\text{<name>}, \text{<selector>}) | \]
\[ \quad \text{<name>} = \llbracket \text{<binding denotation>} \rrbracket \]
\[ \quad \text{Select}(\text{<name>}, \text{<selector>}) = \llbracket \text{<binding denotation>} \rrbracket \]

\[ \text{<binding denotation>} ::= \text{<pattern>} | \text{<pattern>}{\text{<component>}{\text{<binding denotation>}}} \]

\[ \text{<component>} ::= \text{<name>} | \llbracket \text{<binding denotation>} \rrbracket \]

Figure 2. Concrete syntax for the text of an interpreter.
C1. Performing \( \langle \text{name} \rangle \) yields the value bound to \( \langle \text{name} \rangle \); the \( s \)-graph is not affected.

C2. Performing \( \langle \text{elementary value} \rangle \) adds a node \( \beta \) to the \( s \)-graph and makes \( \langle \text{elementary value} \rangle \) the value associated with \( \beta \). The value returned is \( \beta \).

C3. Performing \( \text{Select}(\langle \text{name} \rangle, \langle \text{selector} \rangle) \) does not affect the \( s \)-graph.

If \( \langle \text{name} \rangle \) is bound to node \( \alpha \) and \( \text{sel} \) is the value of \( \langle \text{selector} \rangle \), then the value returned is the unique node \( \beta \) such that \( (\alpha, \text{sel}, \beta) \) is an arc of the current \( s \)-graph. The effect is to select the \( \text{sel} \)-component of the object denoted by \( \alpha \) in the \( s \)-graph.

C4. Performing \( \text{Append}(\langle \text{selector} \rangle):\langle \text{construction}_1 \rangle \) to \( \langle \text{construction}_2 \rangle \) augments the current \( s \)-graph by performing \( \langle \text{construction}_1 \rangle \) and \( \langle \text{construction}_2 \rangle \) to yield nodes \( \alpha_1 \) and \( \alpha_2 \) that denote objects \( \text{obj}_1 \) and \( \text{obj}_2 \). If \( \text{sel} \) is the value of \( \langle \text{selector} \rangle \), the \( s \)-graph is further augmented with a node \( \beta \) and arcs such that \( \beta \) denotes the object formed by appending \( \text{sel}:\text{obj}_1 \) to \( \text{obj}_2 \). Specifically, the added arcs are

\[
(\beta, \text{sel}, \alpha_1)
\]

and

\[
\{(\beta, \text{sel}', \delta) \mid (\alpha_2, \text{sel}', \delta) \text{ is an arc of the } s\text{-graph and } \text{sel}' \neq \text{sel} \}
\]

The value returned is \( \beta \).

C5. Performing a \( \langle \langle \text{constructive denotation} \rangle \rangle \) of the form

\[
\langle \langle \text{pattern} \rangle \langle \text{construction}_1 \rangle \ldots \langle \text{construction}_k \rangle \langle \text{pattern} \rangle \rangle
\]

augments the current \( s \)-graph with a node \( \beta \) and arcs such that \( \beta \) denotes an object satisfying the abstract syntax of the denotation. Performing \( \langle \text{construction}_1 \rangle, \ldots, \langle \text{construction}_k \rangle \) augments the \( s \)-graph and yields nodes \( \alpha_1, \ldots, \alpha_k \). If \( \# \) represents the pattern of the denotation, and \( \text{sel}_1, \ldots, \text{sel}_k \) are the selectors for the components of objects satisfying the abstract syntax, the arcs added to the \( s \)-graph are
\((\beta, \textit{sel}_1, \alpha_1) \ldots (\beta, \textit{sel}_k, \alpha_k)\)
\((\beta, \textit{\emptyset}', \delta)\)

and the valuation of the s-graph is extended to associate \# with a new node \(\delta\). The value returned is \(\beta\).

C6. Performing \([\langle \text{test} \rangle \rightarrow \langle \text{construction}_1 \rangle, \langle \text{construction}_2 \rangle]\) causes \(\langle \text{construction}_1 \rangle\) or \(\langle \text{construction}_2 \rangle\) to be performed according to the outcome of \(\langle \text{test} \rangle\).

There are four forms of \(\langle \text{test} \rangle\). Performing a \(\langle \text{test} \rangle\) has no effect on the current s-graph.

T1. Performing a \(\langle \text{test} \rangle\) of the form \(\langle \text{name} \rangle\) has as its outcome the value to which \(\langle \text{name} \rangle\) is bound. This value must be a truth value.

T2. Performing \(\text{Has}(\langle \text{name} \rangle, \langle \text{selector} \rangle)\) with \(\langle \text{name} \rangle\) bound to node \(\alpha\) and \(\text{sel}\) the value of \(\langle \text{selector} \rangle\) tests whether the object denoted by \(\alpha\) in the s-graph has a \(\text{sel}\)-component. Specifically, the outcome is true if the s-graph contains an arc \((\alpha, \textit{sel}, \beta)\) for some node \(\beta\), and is false otherwise.

T3. Performing \(\langle \text{name} \rangle = \ll\langle \text{binding denotation} \rangle \gg\) has outcome true if the object \(\text{obj}\) denoted by the node \(\alpha\) bound to \(\langle \text{name} \rangle\) satisfies the abstract syntax of the \(\langle \text{binding denotation} \rangle\), and outcome false otherwise. If the outcome is true, the nodes of the s-graph that denote components objects of \(\text{obj}\) are bound to corresponding instances of \(\langle \text{name} \rangle\) in the \(\langle \text{binding denotation} \rangle\). These bindings hold throughout performance of the \(\langle \text{construction} \rangle\) determined by the \(\langle \text{test} \rangle\) outcome.

T4. A \(\langle \text{test} \rangle\) of the form \(\text{Select}(\langle \text{name} \rangle, \langle \text{selector} \rangle) = \ll\langle \text{binding denotation} \rangle \gg\) is performed as in T3, except node \(\alpha\) and the denoted object \(\text{obj}\) are determined by performing the left-hand side according to C3.
As an illustration of the formalism, consider the following text:

\[
\text{interpret(state) = }
\]
\[
\begin{align*}
\text{[state = } & \text{<<eval id in env; cont>> } \rightarrow \\
& \text{(Has(env, id) } \rightarrow \\
& \text{Append 'val':Select(env, id) to cont.} \\
& \text{<<error>>,} \\
& \text{<<error>>)}
\end{align*}
\]

In Figure 3 the nodes of the initial s-graph are solid. The effect of the outer conditional is to bind components of state to the names id, env and cont, as shown in brackets, if the given syntax is satisfied. The inner conditional tests whether \(y\) is bound in the environment \(env\). If both tests are satisfied, the Append (construction) creates a new node \(p\) that denotes the object formed by replacing the 'val'-component of cont with the 'y'-component of env. The new state consists of the root node \(p\) and the s-graph containing only those nodes accessible from \(p\).

If an interpreter state transition cannot be completed by following the above rules, no successor state is defined, and we consider the interpreter to be faulty unless the current state is a final state. I hope the interpreter for K is able to complete all essential state transitions!

6. An Interpreter for the Kernel Language

Our interpreter for the kernel language K has five modes of operation: expression evaluation; command performance; assignment; successful termination; and error termination. Accordingly, the state class is composed of five subclasses:

\[
\text{State ::= eval exp in env; cont|} \\
\text{perform cmd in env; cont|} \\
\text{assign val to id in env; cont|} \\
\text{done val|} \\
\text{error}
\]

For each state other than a terminal state, the 'cont'-component (continuation) is a skeleton of the state that is to hold following completion of the action called for by the present state. For example, the initial state of the interpreter is
Figure 3. Example of a state transition.
eval exp in env: «done ∅»

where the task to be performed is evaluation of the expression exp in a given environment env. The evaluation of exp may require many state transitions, but the ultimate effect is to append the value of exp as the 'val'-component of the continuation done ∅. In general, the continuation of an evaluation state has an empty 'val'-component, and the continuation of a perform state has an empty 'env'-component.

The interpreter for K is presented in Figure 4. Rules E1 through E8 concern the evaluation of expressions. Procedure values are represented according to the syntactic forms

\[
\text{function}(id_1): \text{exp in env}
\]

and

\[
\text{reccfun} \ id_0(id_1): \text{exp in env}
\]

These representations include the environment env in effect at the time the procedure value is created.

Rule E7 concerns procedure application and has cases for nonrecursive and for recursive procedures. The environment env_1 from the procedure value is appended with binding of id_3 to the argument value and, for recursive procedures, binding of id_0 to the procedure value itself. This environment is used for evaluating the procedure body exp.

Rules P1 through P5 concern the performance of commands. In the case of assignment, the right-hand side is evaluated with a skeleton assignment state as its continuation. The continuation is completed with the result of expression evaluation. The assignment state causes formation of a new environment which completes the continuation of the perform state for the assignment command.

The scheme embodied in Rule 7 for application of recursive procedures avoids representing the circularity of recursion by a cyclic structure in the interpreter state. This is, of course, essential to our main point — that reference count storage management suffices for implementation of L.

Study of the interpreter reveals that there is a fixed bound on the number of steps required to perform a state transition, and a fixed bound on the number of nodes added to the s-graph.
Note that this interpreter halts immediately on encountering an unbound identifier or a type error. In this respect the interpreter fails to faithfully implement the functional semantics of $K$: An expression of $K$ may have a normal value by the functional semantics if erroneous values (represented by $\tau$) generated are never used in producing the result value of the expression. However, the interpreter reflects the usual treatment of errors in language implementations that perform run-time checks.
Interpret(\textit{e}) =

E1. $\langle\langle\text{eval } \langle\langle\text{id} \rangle\rangle\text{ in env; cont}\rangle\rangle \rightarrow$

\[
(\text{Has env, id}) \rightarrow \\
\quad \text{Append 'val': Select(env, id) to cont,} \\
\quad \langle\langle\text{error} \rangle\rangle)
\]

E2. $\langle\langle\text{eval } \langle\langle\text{cons} \rangle\rangle\text{ in env; cont}\rangle\rangle \rightarrow$

\quad \text{Append 'val': cons to cont}

E3. $\langle\langle\text{eval } \langle\langle\text{uop id} \rangle\rangle\text{ in env; cont}\rangle\rangle \rightarrow$

\[
(\text{Has env, id}) \rightarrow \\
\quad (\text{Check1(uop, Select(env, id))} \rightarrow \\
\quad \quad \text{Append 'val': Oper1(uop, Select(env, id)) to cont,} \\
\quad \quad \langle\langle\text{error} \rangle\rangle, \langle\langle\text{error} \rangle\rangle)
\]

E4. $\langle\langle\text{eval } \langle\langle\text{id}_1 \text{ bop id}_2 \rangle\rangle\text{ in env; cont}\rangle\rangle \rightarrow$

\[
(\text{Has env, id}_1) \rightarrow (\text{Has env, id}_2) \rightarrow \\
(\text{Check2(bop, Select(env, id}_1), Select(env, id}_2)) \\
\quad \text{Append 'val': Oper2(bop, Select(env, id}_1), Select(env, id}_2)\text{ to cont,} \\
\quad \langle\langle\text{error} \rangle\rangle, \langle\langle\text{error} \rangle\rangle, \langle\langle\text{error} \rangle\rangle)
\]

E5. $\langle\langle\text{eval } \langle\langle\text{proc(id)}\rangle\text{: exp} \rangle\rangle\text{ in env; cont}\rangle\rangle \rightarrow$

\quad \text{Append 'val': \langle\langle\text{function(id)}\rangle\text{: exp in env}\rangle\rangle to cont'}

E6. $\langle\langle\text{eval } \langle\langle\text{rec id}_0(id)\rangle\text{: exp} \rangle\rangle\text{ in env; cont}\rangle\rangle \rightarrow$

\quad \text{Append 'val': \langle\langle\text{recfun id}_0(id)\rangle\text{: exp in env}\rangle\rangle to cont'}

Figure 4. Formal interpreter for the kernel language: Part 1
E7. \[\text{eval } \langle\text{apply id}_1 (\text{id}_2)\rangle \text{ in env}_0; \text{ cont} \rightarrow \]

\[
\{\text{Has(env}_0, \text{id}_1\} \rightarrow \{\text{Has(env}_0, \text{id}_2_}\rightarrow \\
\{\text{Select(env}_0, \text{id}_1\} = \langle\text{function(id}_3\rangle: \text{exp in env}_1\rangle \rightarrow \\
\langle\text{eval exp in} \\
\text{(Append id}_3; \text{Select(env}_0, \text{id}_2\text{ to env}_1); \text{ cont}\rangle, \\
\{\text{Select(env}_0, \text{id}_1\} = \langle\text{recfun id}_0 (\text{id}_3): \text{exp in env}_1\rangle \rightarrow \\
\langle\text{eval exp in} \\
\text{(Append id}_0; \text{Select(env}_0, \text{id}_1\text{ to } \text{ env}_1); \text{cont}\rangle, \\
\langle\text{error}\rangle\}, \\
\langle\text{error}\rangle\} \langle\text{error}\rangle\}
\]

E8. \[\text{eval } \langle\text{cmd req exp}\rangle \text{ in env; cont} \rightarrow \]

\[
\langle\text{perform cmd in env; eval exp in } \emptyset; \text{ cont}\rangle \rightarrow
\]

Figure 4. Part 2.
P1. \[\text{perform } \langle()\rangle \text{ in env; cont} \rightarrow \]
    Append 'env': env to cont

P2. \[\text{perform } \langle\text{cmd}_1; \text{cmd}_2\rangle \text{ in env; cont} \rightarrow \]
    \[\text{perform } \text{cmd}_1 \text{ in env; perform } \text{cmd}_2 \text{ in } \emptyset; \text{ cont} \rightarrow \]

P3. \[\text{perform } \langle\text{id} := \text{exp}\rangle \text{ in env; cont} \rightarrow \]
    \[\text{eval exp in env; assign } \emptyset \text{ to id in env; cont} \rightarrow \]

P4. \[\text{perform } \langle\text{if } \text{id} \text{ then } \text{cmd}_1 \text{ else } \text{cmd}_2\rangle \text{ in env; cont} \rightarrow \]
    \[\text{Has(env, id)} \rightarrow \{\text{Select(env, id)} = \text{Boolean truth} \rightarrow \]
    \[\text{truth} \rightarrow \langle\text{perform } \text{cmd}_1 \text{ in env; cont} \rangle, \]
    \[\langle\text{perform } \text{cmd}_2 \text{ in env; cont} \rangle \}, \langle\text{error}\rangle \}\]

P5. \[\text{perform } \langle\text{while } \text{id} \text{ do } \text{cmd}\rangle \text{ in env; cont} \rightarrow \]
    \[\text{Has(env, id)} \rightarrow \{\text{Select(env, id)} = \text{Boolean truth} \rightarrow \]
    \[\text{truth} \rightarrow \langle\text{perform } \text{cmd} \text{ in env; } \]
    \[\langle\text{perform Select(state, 'cmd') in } \emptyset; \text{ cont} \rangle, \text{ cont} \}, \langle\text{error}\rangle \}\]

A1. \[\text{assign val to id in env; cont} \rightarrow \]
    Append 'env': (Append id; val to env) to cont

Figure 4. Part 3.
7. **Data Structures**

For simplicity, data structures were not included in our language L. Now we shall extend L to include representations for structured data and an appropriate set of primitive operations.

Considerations of clean semantics and ease of program proof led us to design L so that no "side effects" are possible -- every procedure communicates to its caller only by returning a value. In adding data structures to L we will preserve this characteristic of the language. This decision is in contrast with use of a storage model based on the concept of cell [19, 8], or the concept of left-hand values [1].

Arrays and records are two general classes of data structures. We model both with classes of abstract objects: An array is an object where the selectors of the components are elements of \( \text{Int} \); a record is an object where the selectors of the components are members of \( \text{String} \).

To extend L we add a semantic domain \( S \) of compound values to the domain of values \( Y \). Mathematically, a value \( \alpha \) in \( S \) is a mapping

\[
\alpha : S \rightarrow [T \times Y]
\]

where \( S \) is the domain of selectors:

\[
S = N + Q
\]

A compound value \( \alpha \) maps each selector \( \sigma \) into the pair

\[
\alpha(\sigma) = <T, \beta>
\]

where \( T = \text{true} \) if \( \alpha \) has a \( \sigma \)-component, and \( \beta \) is the value of the \( \sigma \)-component if one exists; and \( T \) otherwise.

Thus the extended domain of values for L is defined by

\[
Y = B + [S \rightarrow [T \times Y]] + [Y \rightarrow Y]
\]

\[
\text{compound values} \quad \text{functions}
\]

where

\[
B = T + N + Q \quad \text{[the elementary values]}
\]

\[
S = N + Q \quad \text{[the selectors]}
\]

The essential operations on data structures are their construction from components, and their analysis into component values. Since we wish to regard data structures as values, it makes little sense to speak of modifying or
updating a data structure. Rather, we provide means for creating new data structures by selecting and combining existing values. To this end, we extend the syntax of L to include six additional forms for expressions. These are:

\[ \text{Exp ::= nil} \]

\[ \text{elem exp} \]

\[ \text{exp_1 has exp_2} \]

\[ \text{exp_1[exp_2]} \]

\[ \text{append exp_1;exp_2 to exp_3} \]

\[ \text{construct <sel_1:exp_1, ..., sel_k:exp_k>} \]

- the empty compound value
- test if elementary value
- test existence of component
- select component of compound value
- append value to compound value
- construct a compound value

The expression \text{nil} denotes the compound value that has no components; it is the function that maps each value that is a selector into the pair \langle false, T \rangle.

\[ E[\text{nil}]_\rho = \lambda \alpha. (\sigma: \beta \rightarrow \langle \text{false}, \text{T}, \text{T} \rangle) \]

The value of \langle elem exp \rangle is true if evaluation of exp produces an elementary value, and false if evaluation of exp produces any other value.

\[ E[\text{elem exp}]_\rho = \{ (E[\text{exp}]_\rho : \beta \rightarrow \text{true}, \text{false}) \}

An expression \langle exp_1 has exp_2 \rangle tests whether a compound value \( \alpha \) has a \( \sigma \)-component, where \( \alpha \) is the value of \text{exp_1} and \( \sigma \) is the value of \text{exp_2}. That is, the outcome is

\[ (\alpha(\sigma)) \downarrow 1 \]

which denotes application of \( \alpha \) to \( \sigma \) followed by selecting the first component of the result. The value of the expression is \text{T} if \( \alpha \) is not a compound value or \( \sigma \) is not a selector.

\[ E[\text{exp_1 has exp_2}]_\rho = ((E[\text{exp_1}]_\rho | \_ | (E[\text{exp_2}]_\rho | \_)) \downarrow 1 \]

An expression \langle exp_1[exp_2] \rangle selects the \( \sigma \)-component of \( \alpha \), where \( \alpha \) is the value of \text{exp_1} and \( \sigma \) is the value of \text{exp_2}. The result is

\[ (\alpha(\sigma)) \downarrow 2 \]

If \( \alpha \) is not a compound value, \( \sigma \) is not a selector or if \( \alpha \) does not have a \( \sigma \)-component the result is \text{T}. 
$E[exp_{1}[exp_{2}]]\rho = ((E[exp_{1}]\rho[C]) (E[exp_{2}]\rho[S]));$

Compound values are constructed from simpler values by appending, which is denoted by an expression $<<\text{append } exp_{1}:exp_{2} \text{ to } exp_{3}>>$. The result is the compound value

$\lambda\sigma'. (\sigma' = \sigma \rightarrow \beta, \alpha(\sigma'))$

where $\sigma, \beta$ and $\alpha$ are the values of $exp_{1}, exp_{2}$ and $exp_{3}$. The new compound value is $\alpha$ with its $\sigma$-component replaced by $\beta$. The value $\sigma$ must be a selector and $\alpha$ must be a compound value, but $\beta$ may be any value.

$E[\text{append } exp_{1}:exp_{2} \text{ to } exp_{3}]\rho =$

$\lambda\sigma'. ((\sigma'\rho[S]) \rightarrow E[exp_{2}]\rho, E[exp_{3}]\rho\sigma')$

A construction

$\text{construct } <\text{sel}_{1}:exp_{1}, \ldots, \text{sel}_{k}:exp_{k}>$

is syntactic sugar for the phrase

$\text{append } \text{sel}_{1}:exp_{1} \text{ to } <<\ldots<<\text{append } \text{sel}_{k}:exp_{k} \text{ to } \text{nil}>>\ldots>>$

We require that the selectors $\text{sel}_{1}, \ldots, \text{sel}_{k}$ be distinct, and therefore the order in which components are appended in constructing the new compound value does not matter.

The translation rules given in Section 4.1 remain valid for this extension of $L$; the corresponding extension of the kernel language $K$ incorporates the expression forms

$\text{Exp ::= nil|elem id|id}_{1} \text{ has id}_{2}|id}_{1}[id}_{2}] | \text{append id}_{1}:id}_{2} \text{ to id}_{3}$

Five corresponding rules must be added to the interpreter; the two most interesting rules -- for selection and appending -- are given in Figure 5. These rules have been simplified by considering only integer selectors. The reader may provide the extension to include string selectors.

It follows from the extended interpreter that the extension of $L$ has an efficient implementation (in our sense) using acyclic storage structures and for which reference count storage management is applicable.
(a) selection

\[
\text{\textbackslash{}eval \textless{}id}_1[\textless{}id}_2]\textgreater{} \text{ in env; cont} \rightarrow \\
(\text{Has(env, id}_1) \rightarrow (\text{Has(env, id}_2) \rightarrow \\
\quad [\text{Select(env, id}_1) = \text{\textless{}compound} \text{ obj}]) \rightarrow \\
\quad [\text{Select(env, id}_2) = \text{\textless{}integer} \text{ int}]) \rightarrow \\
\quad \text{Append 'val': Select(obj, int) to cont,} \\
\quad \text{\textless{}error}}\}, \\
\quad \text{\textless{}error}}\}, \text{\textless{}error}}\}
\]

(b) appending

\[
\text{\textbackslash{}eval \textless{}append id}_1:id_2 \text{ to id}_3\textgreater{} \text{ in env; cont} \rightarrow \\
(\text{Has(env, id}_1) \rightarrow (\text{Has(env, id}_2) \rightarrow (\text{Has(env, id}_3) \rightarrow \\
\quad [\text{Select(env, id}_1) = \text{\textless{}integer} \text{ int}]) \rightarrow \\
\quad [\text{Select(env, id}_2) = \text{\textless{}compound} \text{ obj}]) \rightarrow \\
\quad \text{Append 'val': (Append int:Select(env, id}_2 \text{ to obj}) to cont,} \\
\quad \text{\textless{}error}}\}, \\
\quad \text{\textless{}error}}\}, \text{\textless{}error}}\}, \text{\textless{}error}}\}
\]

Figure 5. Interpreter rules for selection and appending.
8. Conclusion

We have shown that it is possible to implement a rich language having clean semantic constructs without resorting to a mark/sweep mechanism for storage reclamation. By denying assignment to free variables, procedure values are included in a form that avoids difficulty, and a duality is achieved between command forms and expression forms. These characteristics of the language have significant bearing on the exploitation of parallelism in computation: Programs in the language are easily translated into the data flow procedural language we have recently described [3]. The data flow representation exposes many of the possibilities for parallel execution of program fragments. The simplicity of L's formalization and its exposure of parallelism are both consequences of the independence of program parts -- the absence of side effects.

Yet this language is incomplete as a general purpose applications language because it lacks provision for several programs to interact -- there is no means for sharing access to data structures between independent programs. However, all popular high-level languages are inadequate in this respect, and the design of appropriate mechanisms with clean semantics remains a challenge. The language does not incorporate the notion of user defined abstract data types [11, 22], but its extension to incorporate this concept (at least in the absence of shared objects) seems straightforward.

The interpretive model shows that strictly acyclic storage structures will suffice for implementation of L, but it also has the property that the object denoted by a node never changes. This suggests a potential simplification of memory systems, for many difficulties in the design of parallel processing computers arise from the possibility that information can be modified. The design of cache memories for a multiprocessor computer provides an example. Thus basing computer design on a language such as L offers the possibility of computer systems that support modular programming and can also exploit the parallelism of program fragments.

Acknowledgement

The design of the programming language discussed in this paper is the joint work of the author and Joe Stoy, who has been a valuable source of knowledge about the theory of functional semantics.
References


15. Parnas, D. L. On the criteria to be used in decomposing systems into modules. Comm. of the ACM 15, 12 (December 1972), 1053-1058.


