MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 109-1

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 of a uniform basis 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

Obj ::= ∅|elem|compound [the set of objects]

Elem ::= truth|int|string [the elementary objects]

Truth ::= true|false [the truth values]

Int ::= 0|+1|-1| ... [the integers]

String ::= < >|'a'|'b'| ... |'aa'| ...

[strings on the alphabet of letters and digits]

The first line of this syntax is intended to be read thus: An entity is an element of the set Obj if and only if the entity is the empty set ∅, or is an element of one of the sets Elem or 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

$$<sel_1:obj_1, \ldots, sel_k:obj_k> \qquad \text{[compound objects]}$$

where

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

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

$$Letexp ::= \text{let } id = exp_0 \text{ in } exp_1$$

which means, by convention, that an object letexp in the class Letexp has the form

$$letexp = <'\$':'\text{let} * = * \text{ in } *', 'id':id, 'exp_0':exp_0, 'exp_1':exp_1>$$

The '$'-component of this objects 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:

Exp ::= id	identifier
cons	constant
uop exp	unary operation
exp_1 bop exp_2	binary operation
let id = exp_0 in exp_1	let expression
if exp_0 then exp_1 else exp_2	conditional
iterate id from exp_0 by exp_1 while exp_2	iteration
proc(id_1): exp	procedure declaration
rec id_0(id_1): exp	recursive procedure
apply exp_1(exp_2)	application
cmd res exp	result of command

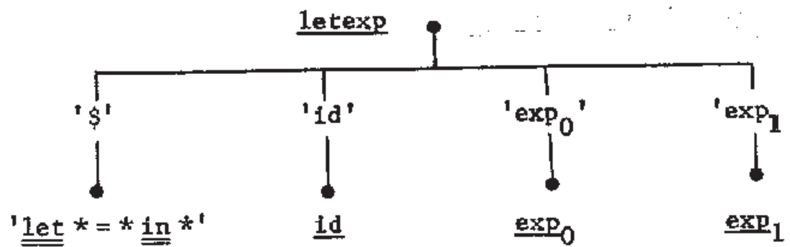

Figure 1. An object in the syntactic class of let expressions.

The identifiers of L are strings

$$\underline{Id} ::= \underline{string}$$

the constants are values

$$\underline{Cons} ::= \underline{value}$$

and the values are typed:

$$\underline{Value} ::= \underline{Boolean}\ \underline{truth} \mid \underline{integer}\ \underline{int} \mid \underline{string}\ \underline{string}$$

The sets Uop and 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 $\underline{\underline{V}}$, which is the "reflexive domain" of functional semantics [17]:

$$\underline{\underline{V}} = \underline{\underline{B}} + [\underline{\underline{V}} \to \underline{\underline{V}}]$$

where

$$\underline{\underline{B}} = \underline{\underline{T}} + \underline{\underline{N}} + \underline{\underline{Q}}$$

$$\underline{\underline{T}} = \{\bot, \text{true}, \text{false}, \top\}$$

[the domain of truth values]

$$\underline{\underline{N}} = \{\bot, 0, +1, -1, \ldots, \top\}$$

[the domain of integers]

$$\underline{\underline{Q}} = \{\bot, <>, \text{'a'}, \text{'b'}, \ldots, \text{'aa'}, \ldots, \top\}$$

[the domain of quotations]

Elements of $\underline{\underline{V}}$ will be denoted by α, β, φ. For the semantics of L we take $\bot$ ("bottom") to be the meaning of nonterminating computations and $\top$("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:

$$\mathcal{K} : \underline{Value} \to \underline{\underline{B}} \text{ in } \underline{\underline{V}}$$

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 ρ of a class of functions $\underline{\underline{E}}$ called environments:

$$\rho \in \underline{\underline{E}}, \quad \underline{\underline{E}} = [\underline{Id} \to \underline{\underline{V}}]$$

The meaning of an expression is given by the semantic function

$$\mathcal{E}: \underline{\text{Exp}} \to [\underline{\underline{E}} \to \underline{\underline{V}}]$$

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:

$$\mathcal{E}[[\text{exp}]]\rho = \{\underline{\text{exp}} = \ll\underline{\text{id}}\gg \longrightarrow \rho(\underline{\text{id}}), \quad \underline{\text{exp}} = \ll\underline{\text{cons}}\gg \longrightarrow \mathcal{K}[[\underline{\text{cons}}]], \quad \vdots \quad \vdots$$

This is usually abbreviated to

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

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 $[\underline{\underline{V}} \to \underline{\underline{V}}]$ 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:

$$\mathcal{E}[[\underline{\underline{\text{rec}}}\ \underline{\text{id}}_0(\underline{\text{id}}_1): \underline{\text{exp}}]]\rho = Y[\lambda\varphi.((\lambda\alpha.\ \mathcal{E}[[\underline{\text{exp}}]]\rho')\ \text{in}\ \underline{\underline{V}})]$$

$$\rho' = \lambda\underline{\text{id}}.\{\underline{\text{id}} = \underline{\text{id}}_0 \longrightarrow \varphi, \{\underline{\text{id}} = \underline{\text{id}}_1 \longrightarrow \alpha, \rho(\underline{\text{id}})\}\}$$

The argument of the fixed point operator is the function determined by the mapping of functions into functions that maps the function φ into the function determined by the procedure body $\underline{\text{exp}}$ for the environment ρ' in which $\underline{\text{id}}_0$ is bound to φ and the formal argument $\underline{\text{id}}_1$ is bound to the value α. The meaning of the recursive procedure is the function θ which, if bound to $\underline{\text{id}}_0$, yields the same function θ as the meaning of the procedure body.

The meaning of procedure application is given by

$$\mathcal{E}[\![\underline{\text{apply}}\ \underline{\text{exp}}_1(\underline{\text{exp}}_2)]\!] = (\mathcal{E}[\![\underline{\text{exp}}_1]\!]\rho|\underline{\underline{V}} \to \underline{\underline{V}})(\mathcal{E}[\![\underline{\text{exp}}_2]\!]\rho)$$

which says that values are obtained for both expressions, and the value of $\underline{\text{exp}}_1$ (which must be a function in $\underline{\underline{V}} \to \underline{\underline{V}}$) is applied to the value of $\underline{\text{exp}}_2$.

Finally, we have

$$\mathcal{E}[\![\underline{\text{cmd}}\ \underline{\underline{\text{res}}}\ \underline{\text{exp}}]\!]\rho = \mathcal{E}[\![\underline{\text{exp}}]\!]\underbrace{(\mathcal{C}[\![\underline{\text{cmd}}]\!]\rho)}_{\rho'}$$

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

2.3. The Commands of L

The abstract syntax for the command forms of L is:

Cmd ::= ( )	the empty command
$\underline{\text{cmd}}_1$; $\underline{\text{cmd}}_2$	sequencing
id := exp	assignment
if exp then $\underline{\text{cmd}}_1$ else $\underline{\text{cmd}}_2$	conditional
while exp do cmd	iteration

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

$$\mathcal{C}: \text{Cmd} \to [\underline{\underline{E}} \to \underline{\underline{E}}]$$

For example, the meaning of assignment is given by

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

For the while command, a fixed point must be found:

$$\mathcal{C}[\![\underline{\underline{\text{while}}}\ \underline{\text{exp}}\ \underline{\underline{\text{do}}}\ \underline{\text{cmd}}]\!]\rho =$$
$$Y[\lambda\psi.\lambda\rho'.\{(\mathcal{E}[\![\underline{\text{exp}}]\!]\rho'|\underline{\underline{T}}) \longrightarrow \psi(\mathcal{C}[\![\underline{\text{cmd}}]\!]\rho'),\ \rho'\}]\rho$$

in which $\psi: \underline{\underline{E}} \to \underline{\underline{E}}$.

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 language 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 language 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 restricted to identifiers.

Abstract Syntax of K:

Exp ::= id | cons | uop id | id_1 bop id_2 |
proc(id_1): exp |
rec id_0(id_1): exp |
apply id_1(id_2) |
cmd res exp

Cmd ::= ( ) | cmd_1; cmd_2 |
id := exp |
if id then cmd_1 else cmd_2 |
while id_1 do cmd

4.1. 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 = 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

let id = exp_0 in exp_1
becomes <<id := exp_0>> res exp_1

Rule 2: Conditional expressions

if exp_0 then exp_1 else exp_2
becomes <<if exp_0 then <<id := exp_1>>
else <<id := exp_2>> >> res id

where id = Newid

Rule 3: Iteration expressions

iterate id from exp_0 by exp_1 while exp_2
becomes << <<id := exp_0>>; <<while exp_2 do
<<id := exp_1>> >> >> res id

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

Rule 4: Operations and Application

a. uop exp becomes <<id := exp>> res <<uop id>>

b. exp_1 bop exp_2 becomes <<id_1 := exp_1; id_2 := exp_2>> res <<id_1 bop id_2>>

c. apply exp_1(exp_2) becomes <<id_1 := exp_1; id_2 := exp_2>> res <<apply id_1(id_2)>>

where id = Newid
id_1 = Newid
id_2 = 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 cmd_1 else cmd_2
   becomes <<id := exp; if id then cmd_1 else cmd_2>>

b. while exp do cmd
   becomes <<id := exp; while id do
   <<cmd; id := exp>> >>

   where id = 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 Formalism

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 similar 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:

$$\underline{\text{Elem}} ::= \underline{\text{truth}} \,|\, \underline{\text{int}} \,|\, \underline{\text{string}}$$
the elementary values

$$\underline{\text{Sel}} ::= \underline{\text{int}} \,|\, \underline{\text{string}}$$
the selectors

Unid: a denumerable set of unique identifiers

A graph is a triple

$$(\underline{\text{nodes}}, \underline{\text{arcs}}, \underline{\text{val}})$$

where

$$\underline{\text{nodes}} \subset \underline{\text{Unid}}$$
$$\underline{\text{arcs}} \in \underline{\text{nodes}} \times \underline{\text{Sel}} \times \underline{\text{nodes}}$$
$$\underline{\text{val}} \in \underline{\text{nodes}} \times \underline{\text{Elem}}$$

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

We require that the set of arcs originating at each node α be distinct

$$\left.\begin{array}{l}(\alpha, \underline{\text{sel}}_1, \beta_1) \in \underline{\text{arcs}} \\ (\alpha, \underline{\text{sel}}_2, \beta_2) \in \underline{\text{arcs}}\end{array}\right\} \text{ and } \beta_1 \neq \beta_2 \text{ implies } \underline{\text{sel}}_1 \neq \underline{\text{sel}}_2$$

and that val associate at most one value with any node

$$\left.\begin{array}{l}(\alpha, \underline{\text{elem}}_1) \in \underline{\text{val}} \\ (\alpha, \underline{\text{elem}}_2) \in \underline{\text{val}}\end{array}\right\} \text{ implies } \underline{\text{elem}}_1 = \underline{\text{elem}}_2$$

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 γ associates elementary values only with leaf nodes of γ.

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

Let σ be

$$\{\underline{sel} \mid (\alpha, \underline{sel}, \beta) \in \underline{arcs} \text{ for some } \beta \in \underline{nodes}\}$$

and suppose

$$\sigma = \{\underline{sel}_1, \ldots, \underline{sel}_k\}$$

If σ is not empty, then

$$\text{Object}(\alpha, \gamma) = \langle \underline{sel}_1 : \underline{obj}_1, \ldots, \underline{sel}_k : \underline{obj}_k \rangle$$

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

$$\text{Object}(\alpha, \gamma) = \underline{elem}$$

Otherwise

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

In words, this simply states that Object(α, γ) contains paths to its leaves that correspond (in the sense of identical selector labelling of the arcs) to paths in γ originating at α and terminating on leaves of γ.

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

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

$$\gamma_0 \longrightarrow \gamma_1 \longrightarrow \cdots \longrightarrow \gamma_k$$

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 γ_k and a node ρ' of γ_k that defines the new interpreter state (ρ', γ'). The s-graph γ' of the new state is obtained from γ_k by deleting the old root node ρ and all nodes and arcs inaccessible from the new root node ρ'.

Each state (ρ, γ) determines the object Object(ρ, γ) denoted by the root node ρ. Hence an interpreter may also be regarded as defining a function

Interp: <u>State</u> → <u>State</u>

where <u>State</u> 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:

⟨name⟩ names are bound to values which are either nodes of an s-graph or elementary values.

⟨selector⟩ yields a value which is a selector (an element of <u>Sel</u>).

⟨function name⟩(⟨name⟩) = ⟨construction⟩ specifies, for any state (ρ, γ), the successor state (ρ', γ'). The ⟨construction⟩ is performed with ⟨name⟩ bound to ρ yielding a transformed s-graph γ^* and the root node ρ' of the next state. The s-graph γ' is obtained from γ^* by deleting ρ and nodes inaccessible from ρ'.

⟨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 β that denotes the object whose representation has been constructed.

We consider in turn the six forms of a ⟨construction⟩:

```
⟨definition⟩ ::= ⟨function name⟩ (⟨name⟩) = ⟨construction⟩

⟨construction⟩ ::= ⟨name⟩|
                   ⟨elementary value⟩|
                   Select(⟨name⟩, ⟨selector⟩)|
                   Append⟨selector⟩:⟨construction_1⟩ to ⟨construction_2⟩
                   << ⟨constructive denotation⟩ >>
                   {⟨test⟩ ──→ ⟨construction_1⟩, ⟨construction_2⟩}

⟨selector⟩ ::= ⟨name⟩|⟨integer⟩|⟨string⟩

⟨constructive denotation⟩ ::= ⟨pattern⟩|
                              ⟨pattern⟩⟨construction⟩⟨constructive denotation⟩

⟨test⟩         ::= ⟨name⟩
                   Has(⟨name⟩, ⟨selector⟩)|
                   ⟨name⟩ = << ⟨binding denotation⟩>>
                   Select(⟨name⟩, ⟨selector⟩) = <<⟨binding denotation⟩>>

⟨binding denotation⟩ ::= ⟨pattern⟩|⟨pattern⟩⟨component⟩⟨binding denotation⟩

⟨component⟩   ::=  ⟨name⟩|<<⟨binding denotation⟩>>
```

Figure 2. Concrete syntax for the text of an interpreter.

C1. Performing ⟨name⟩ yields the value bound to ⟨name⟩; the s-graph is not affected.

C2. Performing ⟨elementary value⟩ adds a node β to the s-graph and makes ⟨elementary value⟩ the value associated with β. The value returned is β.

C3. Performing Select(⟨name⟩, ⟨selector⟩) does not affect the s-graph. If ⟨name⟩ is bound to node α and sel is the value of ⟨selector⟩, then the value returned is the unique node β such that $(\alpha, \underline{sel}, \beta)$ is an arc of the current s-graph. The effect is to select the sel-component of the object denoted by α in the s-graph.

C4. Performing Append⟨selector⟩:⟨construction$_1$⟩ to ⟨construction$_2$⟩ augments the current s-graph by performing ⟨construction$_1$⟩ and ⟨construction$_2$⟩ to yield nodes α_1 and α_2 that denote objects $\underline{obj}_1$ and $\underline{obj}_2$. If sel is the value of ⟨selector⟩, the s-graph is further augmented with a node β and arcs such that β denotes the object formed by appending $\underline{sel}$:$\underline{obj}_1$ to $\underline{obj}_2$. Specifically, the added arcs are

$$(\beta, \underline{sel}, \alpha_1)$$

and

$$\left\{(\beta, \underline{sel}', \delta) \;\middle|\; \begin{array}{l}(\alpha_2, sel', \delta) \text{ is an arc of the} \\ \text{s-graph and } \underline{sel}' \neq \underline{sel}\end{array}\right\}$$

The value returned is β.

C5. Performing a ≪⟨constructive denotation⟩≫ of the form ≪⟨pattern⟩⟨construction$_1$⟩ ... ⟨construction$_k$⟩⟨pattern⟩≫ augments the current s-graph with a node β and arcs such that β denotes an object satisfying the abstract syntax of the denotation. Performing ⟨construction$_1$⟩,...,⟨construction$_k$⟩ augments the s-graph and yields nodes $\alpha_1, \ldots, \alpha_k$. If # represents the pattern of the denotation, and $\underline{sel}_1, \ldots, \underline{sel}_k$ are the selectors for the components of objects satisfying the abstract syntax, the arcs added to the s-graph are

$(\beta, \underline{\text{sel}}_1, \alpha_1) \ldots (\beta, \underline{\text{sel}}_k, \alpha_k)$
$(\beta, \text{'\$'}, \delta)$

and the valuation of the s-graph is extended to associate # with a new node δ. The value returned is β.

C6. Performing [⟨test⟩ ⟶ ⟨construction$_1$⟩, ⟨construction$_2$⟩] causes ⟨construction$_1$⟩ or ⟨construction$_2$⟩ to be performed according to the outcome of ⟨test⟩.

There are four forms of ⟨test⟩. Performing a ⟨test⟩ has no effect on the current s-graph.

T1. Performing a ⟨test⟩ of the form ⟨name⟩ has as its outcome the value to which ⟨name⟩ is bound. This value must be a truth value.

T2. Performing Has(⟨name⟩, ⟨selector⟩) with ⟨name⟩ bound to node α and sel the value of ⟨selector⟩ tests whether the object denoted by α in the s-graph has a sel-component. Specifically, the outcome is true if the s-graph contains an arc (α, sel, β) for some node β, and is false otherwise.

T3. Performing ⟨name⟩ = ≪⟨binding denotation⟩≫ has outcome true if the object obj denoted by the node α bound to ⟨name⟩ satisfies the abstract syntax of the ⟨binding denotation⟩, and outcome false otherwise. If the outcome is true, the nodes of the s-graph that denote components objects of obj are bound to corresponding instances of ⟨name⟩ in the ⟨binding denotation⟩. These bindings hold throughout performance of the ⟨construction⟩ determined by the ⟨test⟩ outcome.

T4. A ⟨test⟩ of the form Select(⟨name⟩, ⟨selector⟩) = ≪⟨binding denotation⟩≫ is performed as in T3, except node α and the denoted object obj are determined by performing the left-hand side according to C3.

As an illustration of the formalism, consider the following text:

```
Interpret(state) =
    {state = <<eval id in env; cont>> ⟶
        {Has(env, id) ⟶
            Append 'val':Select(env, id) to cont,
            <<error>>},
    <<error>>}
```

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 ρ' 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 ρ' and the s-graph containing only those nodes accessible from ρ'.

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:

```
State ::= eval exp in env; cont|
          perform cmd in env; cont|
          assign val to id in env; cont|
          done val|
          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

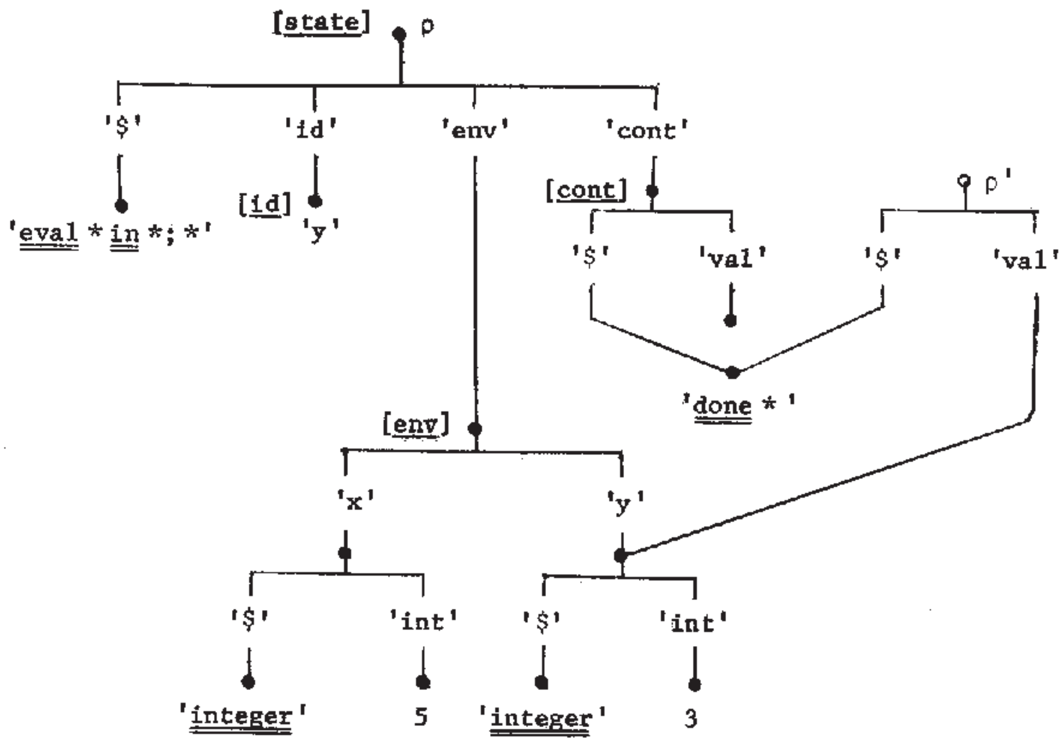

Figure 3. Example of a state transition.

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

function(id_1): exp in env

and

recfun id_0(id_1): 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 performance 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 $\top$) 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(state) =

E1.   <<eval <<id>> in env; cont>> --->
          {Has(env, id) --->
               Append 'val':Select(env, id) to cont,
          <<error>>}

E2.   <<eval <<cons>> in env; cont>> --->
          Append 'val':cons to cont

E3.   <<eval <<uop id>> in env; cont>> --->
          {Has(env, id) --->
               {Check1(uop, Select(env, id)) --->
                    Append 'val':Oper1(uop, Select(env, id)) to cont
          <<error>>}, <<error>>}

E4.   <<eval <<id_1 bop id_2>> in env; cont>> --->
          {Has(env, id_1) ---> {Has(env, id_2) --->
          {Check2(bop, Select(env, id_1), Select(env, id_2))
               Append 'val':Oper2(bop, Select(env, id_1), Select(env, id_2)) to cont,
          <<error>>}, <<error>>}, <error>>}

E5.   <<eval <<proc(id_1): exp>> in env; cont>> --->
          Append 'val':<<function(id_1): exp in env>> to cont

E6.   <<eval <<rec id_0(id_1): exp>> in env; cont>> --->
          Append 'val':<<recfun id_0(id_1): exp in env>> to cont
```

Figure 4. Formal interpreter for the kernel language: Part 1

E7. <<eval <<apply $id_1(id_2)$>> in env_0; cont>> ⟶

{Has(env_0, id_1) ⟶ {Has(env_0, id_2) ⟶

{Select(env_0, id_1) = <<function(id_3): exp in env_1>> ⟶

<<eval exp in

(Append id_3: Select(env_0, id_2) to env_1); cont>>,

{Select(env_0, id_1) = <<recfun $id_0(id_3)$: exp in env_1>> ⟶

<<eval exp in

(Append id_0: Select(env_0, id_1) to

(Append id_3: Select(env_0, id_2) to env_1)); cont>>,

<<error>>},

<<error>>} <<error>>}

E8. <<eval <<cmd res exp>> in env; cont>> ⟶

<<perform cmd in env; <<eval exp in ∅; cont>> >>

Figure 4. Part 2.

P1. <<perform <<( )>> in env; cont>> ⟶

Append 'env':env to cont

P2. <<perform <<cmd_1; cmd_2>> in env; cont>> ⟶

<<perform cmd_1 in env; <<perform cmd_2 in ∅; cont>> >>

P3. <<perform <<id := exp>> in env; cont>> ⟶

<<eval exp in env; <<assign ∅ to id in env; cont>> >>

P4. <<perform <<if id then cmd_1 else cmd_2>> in env; cont>> ⟶

{Has(env, id) ⟶ {Select(env, id) = <<Boolean truth>> ⟶
  {truth ⟶ <<perform cmd_1 in env; cont>>,
    <<perform cmd_2 in env; cont>>},
<<error>>}, <<error>>}

P5. <<perform <<while id do cmd>> in env; cont>> ⟶

{Has(env, id) ⟶ {Select(env, id) = <<Boolean truth>> ⟶
  {truth ⟶ <<perform cmd in env;
    <<perform Select(state, 'cmd') in ∅; cont>> >>, cont},
  <<error>>}, <<error>>}

A1. <<assign val to id in env; cont>> ⟶

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 Int; a record is an object where the selectors of the components are members of String.

To extend L we add a semantic domain $\underline{\underline{C}}$ of compound values to the domain of values $\underline{\underline{V}}$. Mathematically, a value α in $\underline{\underline{C}}$ is a mapping

$$\alpha : \underline{\underline{S}} \rightarrow [\underline{\underline{T}} \times \underline{\underline{V}}]$$

where $\underline{\underline{S}}$ is the domain of selectors:

$$\underline{\underline{S}} = \underline{\underline{N}} + \underline{\underline{Q}}$$

A compound value α maps each selector σ into the pair

$$\alpha(\sigma) = \langle \tau, \beta \rangle$$

where τ = true if α has a σ-component, and β is the value of the σ-component if one exists; and $\top$ otherwise.

Thus the extended domain of values for L is defined by

$$\underline{\underline{V}} = \underline{\underline{B}} + \underbrace{[\underline{\underline{S}} \rightarrow [\underline{\underline{T}} \times \underline{\underline{V}}]]}_{\text{compound values}} + \underbrace{[\underline{\underline{V}} \rightarrow \underline{\underline{V}}]}_{\text{functions}}$$

where

$\underline{\underline{B}} = \underline{\underline{T}} + \underline{\underline{N}} + \underline{\underline{Q}}$ [the elementary values]

$\underline{\underline{S}} = \underline{\underline{N}} + \underline{\underline{Q}}$ [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:

Exp ::= nil	the empty compound value
elem exp	test if elementary value
exp_1 has exp_2	test existence of component
$\text{exp}_1[\text{exp}_2]$	select component of compound value
append $\text{exp}_1:\text{exp}_2$ to exp_3	append value to compound value
construct $\langle \text{sel}_1:\text{exp}_1, \ldots, \text{sel}_k:\text{exp}_k \rangle$	construct a compound value

The expression 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, $\top\rangle$.

$$\mathcal{E}[\![\text{nil}]\!]\rho = \lambda\alpha.\{\alpha:\underline{S} \longrightarrow \langle\text{false}, \top\rangle, \ \top\}$$

The value of <<elem exp>> is true if evaluation of exp produces an elementary value, and false if evaluation of exp produces any other value.

$$\mathcal{E}[\![\text{elem exp}]\!]\rho = \{(\mathcal{E}[\![\text{exp}]\!]\rho):\underline{B} \longrightarrow \text{true, false}\}$$

An expression <<exp_1 has exp_2>> tests whether a compound value α has a σ-component, where α is the value of exp_1 and σ is the value of exp_2. That is, the outcome is

$$(\alpha(\sigma)) \downarrow 1$$

which denotes application of α to σ followed by selecting the first component of the result. The value of the expression is $\top$ if α is not a compound value or σ is not a selector.

$$\mathcal{E}[\![\text{exp}_1 \text{ has } \text{exp}_2]\!]\rho = ((\mathcal{E}[\![\text{exp}_1]\!]\rho)|\underline{C})(\mathcal{E}[\![\text{exp}_2]\!]|\underline{S})) \downarrow 1$$

An expression <<$\text{exp}_1[\text{exp}_2]$>> selects the σ-component of α, where α is the value of exp_1 and σ is the value of exp_2. The result is

$$(\alpha(\sigma)) \downarrow 2$$

If α is not a compound value, σ is not a selector or if α does not have a σ-component the result is $\top$.

$$\mathcal{E}[\![\underline{exp}_1[\underline{exp}_2]\;]\!]\rho = ((\mathcal{E}[\![\underline{exp}_1]\!]\rho|\underline{\underline{C}})(\mathcal{E}[\![\underline{exp}_2]\!]\rho|\underline{\underline{S}}))\downarrow 2$$

Compound values are constructed from simpler values by appending, which is denoted by an expression <<$\underline{\underline{append}}$ $\underline{exp}_1$:$\underline{exp}_2$ to $\underline{exp}_3$>>. The result is the compound value

$$\lambda\sigma'.\ \{\sigma' = \sigma \longrightarrow \beta,\ \alpha(\sigma')\}$$

where σ, β and α are the values of $\underline{exp}_1$, $\underline{exp}_2$ and $\underline{exp}_3$. The new compound value is α with its σ-component replaced by β. The value σ must be a selector and α must be a compound value, but β may be any value.

$$\mathcal{E}[\![\underline{\underline{append}}\ \underline{exp}_1:\underline{exp}_2\ \underline{\underline{to}}\ \underline{exp}_3]\!]\rho =$$
$$\lambda\sigma'.\{(\sigma'|\underline{\underline{S}}) = (\mathcal{E}[\![\underline{exp}_1]\!]\rho|\underline{\underline{S}}) \longrightarrow \mathcal{E}[\![\underline{exp}_2]\!]\rho,\ (\mathcal{E}[\![\underline{exp}_3]\!]\rho)\sigma'\}$$

A construction

$$\underline{\underline{construct}}\ \langle\underline{sel}_1:\underline{exp}_1,\ \ldots,\ \underline{sel}_k:\underline{exp}_k\rangle$$

is syntactic sugar for the phrase

$$\underline{\underline{append}}\ \underline{sel}_1:\underline{exp}_1\ \underline{\underline{to}}\ \langle\langle\ldots\langle\langle\underline{\underline{append}}\ \underline{sel}_k:\underline{exp}_k\ \underline{\underline{to}}\ \underline{\underline{nil}}\rangle\rangle\ldots\rangle\rangle$$

We require that the selectors $\underline{sel}_1, \ldots, \underline{sel}_k$ 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

$$\text{Exp} ::= \underline{\underline{nil}}\,|\,\underline{\underline{elem}}\ \underline{id}\,|\,\underline{id}_1\ \underline{\underline{has}}\ \underline{id}_2\,|\,\underline{id}_1[\underline{id}_2]\,|\,\underline{\underline{append}}\ \underline{id}_1:\underline{id}_2\ \underline{\underline{to}}\ \underline{id}_3$$

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

<<eval <<id$_1$[id$_2$]>> in env; cont>> →
    {Has(env, id$_1$) → {Has(env, id$_2$) →
        {Select(env, id$_1$) = <<compound obj>> →
            {Select(env, id$_2$) = <<integer int>> →
                Append 'val': Select(obj, int) to cont,
                <<error>>},
            <<error>>},
        <<error>>}, <<error>>}

(b) appending

<<eval <<append id$_1$:id$_2$ to id$_3$>> in env; cont>> →
    {Has(env, id$_1$) → {Has(env, id$_2$) → {Has(env, id$_3$) →
        {Select(env, id$_1$) = <<integer int>> →
            {Select(env, id$_3$) = <<compound obj>> →
                Append 'val': (Append int:Select(env, id$_2$) to obj) to cont,
                <<error>>},
            <<error>>},
        <<error>>}, <<error>>}, <<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 procedurelanguage 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

1. Barron, D. W., J. N. Buxton, D. F. Hartley, E. Nixon, and C. Strachey. The main features of CPL. Comp. J. 6, 2 (July 1963), 134-143.

2. Bobrow, D. G., and B. Wegbreit. A model and stack implementation of multiple environments. Comm. of the ACM 16, 10 (October 1973), 591-603.

3. Dennis, J. B. First version of a data flow procedure language. Proceedings of Symposium on Programming, Institut de Programmation, University of Paris, France, April 1974, 241-271.

4. Dennis, J. B., and J. E. Stoy. Paper in preparation.

5. Ellis, D. J. Semantics of Data Structures and References. Technical Report TR-134, Project MAC, M.I.T., Cambridge, Mass., August 1974.

6. Evans, A., Jr. PAL -- a language designed for teaching programming linguistics. Proceedings of the 23rd ACM National Conference, 1968, 395-403.

7. Fenichel, R. F. List tracing in systems allowing multiple cell-types. Comm. of the ACM 14, 8 (August 1971), 522-526.

8. Henderson, D. A., Jr. The Binding Model: A Semantic Base for Modular Programming. Ph.D Thesis, Department of Electrical Engineering, M.I.T., Cambridge, Mass., forthcoming.

9. Hoare, C. A. R. Procedures and parameters: an axiomatic approach. Symposium on Semantics of Algorithmic Languages, Lectures Notes in Mathematics, 118, Springer-Verlag 1971, 102-116.

10. Hoare, C. A. R., and N. Wirth. An Axiomatic Definition of the Programming Language Pascal. Computer Science Group Report, Eidgenossische Technische Hochschule, Zurich, November 1972.

11. Liskov, B. H., and S. N. Zilles. Programming with Abstract data types. SIGPLAN Notices 9, 4 (April 1974), 50-59.

12. Lucas, P., and K. Walk. On the formal description of PL/1. Annual Review in Automatic Programming 6, Pergamon Press 1969, 105-182.

13. McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, part 1. Comm. of the ACM 3, 4 (April 1960), 184-195.

14. McCarthy, J. A formal description of a subset of ALGOL. Formal Language Description Languages for Computer Programming, North-Holland Publishing Co., Amsterdam 1966, 1-12.

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.

16. Reynolds, J. C. GEDANKEN -- A simple typeless language based on the principle of completeness and the reference concept. Comm. of the ACM 13, 5 (May 1970), 308-319.

17. Scott, D. Outline of a Mathematical Theory of Computation. Technical Monograph PRG-2, Oxford University Computing Laboratory, Programming Research Group, Oxford, England, November 1960.

18. Stanford Research Institute, Proceedings of a Symposium on the High Cost of Software, Naval Postgraduate School, Monterey, Calif., September 1973.

19. Walk, K. Modelling of storage properties of higher level languages. Proceedings of a Symposium on Data Structures in Programming Languages, SIGPLAN Notices 6, 2 (February 1971), 146-170.

20. Weizenbaum, J. Symmetric list processor. Comm. of the ACM 6, 9 (September 1963), 524-536.

21. Zilles, S. N. Paper in preparation.