MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 90

A Semantics for Structured Programs

by

D. Austin Henderson, Jr.

October 1973

This research was supported in part by the National Science Foundation under research grant GJ-3467L, and in part by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 433 which was monitored by ONR Contract No. N00014-70-A-0362-0001.

The major idea underlying much of the work on both structured programming and structured programs is that of expressing programs in terms of abstractions and then either finding or creating programs which support (implement) those abstractions.

Some work concerns methodology for choosing and specifying operations [e.g. Dijkstra, Parnas]. More recently others have concentrated on the abstraction of data types [Simula 67, McKeag, Zilles, Liskov and Zilles] .

Abstractions appear in programs as identifiers. Support for abstraction appear as programs. Hence the use of abstraction must be supported by a system capable of implementing associations between abstraction-denoting identifiers and their supporting programs.

This note discusses an abstract computation system in terms of which the abstractions of structured programming and the associations which that methodology implies are expressible. This system is explained by defining a class of expressions, and an interpreter for them which converts a system "state" into some set of alternative resultant states. (The set of outcomes explains possible non-deterministic behavior of the interpreter.) That is, it is explained by defining an abstract machine.

This machine is offered as a functional specification of computing hardware augmented by a software nucleus.which will support structured programming. It therefore also offers a set of semantics for the system support of structured programming; that is, it offers a set of concepts adequate to define the creation and meanings of structured programs.

The machine has seven primitive data types ; every value it computes is either a boolean, integer, string, structure, cell, functional or object. Each type has associated with it a collection of machine "instructions" with which instances of it can be manipulated. This machine's action is described by defining the effect of each instruction on the machine's state. An informal description of these effects is given in the following paragraphs.

Boolean, integer, and string values are constant: no "instructions" ("operators" will be used synonymously) can effect the results of other operations on them. Thus

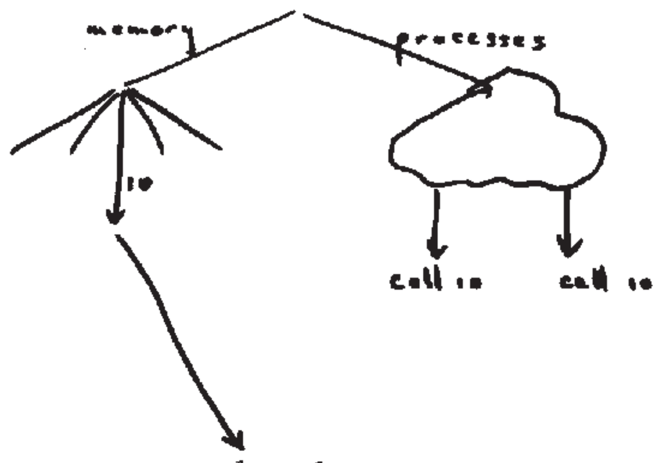
not(true)

yields *false*, but does not change *true* in any way. Apart from this, the operators on values of these types are intellectually uninteresting, and so will not be described here.

Structures are also constant values. A structure has a set of selectors--values of type string, each of which "selects" a component which is a value of any type. The *select* operator accesses component values. The *augment* operator constructs a new structure with an additonal component; informally, all components but the additional one share with the components of the "old" structure where it makes any difference (see below). *Augment* leaves the "old" structure unchanged.

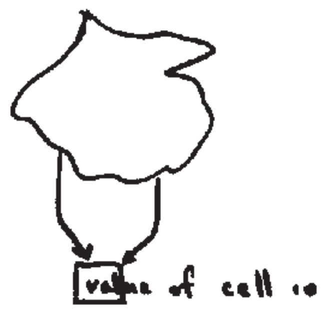
Cells are the changeable memory of the machine. These values have state information associated with them. That state is any other value of the machine. A cell is created with *newcell*; its state is accessed with *contents* and changed with *update*.

More formally, the semantics of cells are described by having a "memory section" of the machine state. A value of type cell is a "reference" to some part of the memory section; that part is by interpretation the contents of the cell value.



value of cell 10

Usually the memory section of the interpreter state is not explicitly mentioned and cellsvalues are envisioned as enclosing their contents.



Augmenting a structure with a cell value as a component duplicates the reference to the cell, but does not "copy" the cell. Thus "value sharing" can be introduced by using cells.

Program text is introduced into the machine as string values having the special form:

'"{extends | operations}"

"Operations" is interpreted as an expression specifying a sequence of primitive operations to be performed. Sequencing is provided by syntactic forms indicating sequential, conditional, and iterative control strategies. The primitive operators are indicated by a set of reserved identifiers (see Appendix 1). The operands of the primitive operators are either other expressions (this yields nested applicative forms), literals (e.g. true, -4, 'WLS3P'), or identifiers.

Identifiers are user-doined names. Identifiers may be coined "locally" using the declare operator, in which case the value they denote is an argument to that operator. Otherwise, the value of an identifier must be determined by external action, in which case it must appear in "externals", the list of external identifiers of the text.

For example

```
'{x |
   declare('y', multiply(x,x))
   resultis (plus(y,x))
}'
```

A value of type functional results from the action of the install primitive on a string which has the form of program text. The action of install may be thought of as including some sort of compilation process so that the resulting functional may be more efficiently interpreted. Semantically, however, install simply "changes" a string to an unbound functional. The external identifiers of this functional do not denote any values.

New functionals can be created by using the bind primitive to give one of those externals meaning.

bind(functional, identifer, value)

Like, augment, bind does not alter its argument; it yields a new functional with another of its externals bound. Functionals are "partially closed" programs.

A functional may be evaluated using the eval operator to compute a value.

These mechanisms are sufficient to describe abstract operators. The "programs" supporting a particular abstraction is described as a functional. It may be partially bound, and then bound as the meaning of an external of another functional. That functional binds the remaining externals and evaluates the result.

Notice that the usual distinction between free (non-local) variables and parameters is not made in the machine. This distinction is reintroduced by interpretation of where, with respected to evaluation, the external in question is bound: parameters are bound by the evaluating functional, free variables by some other functional.

Notice also that the structured programming concept of supporting an abstract operation with a program has been extended in the machine; functionals-partially closed programs--support abstract operations. This permits abstract operations which use other abstractions on data bases. These needed supporting values are bound into the program to create a functional which supports the desired abstraction.

The last primitive data type of the machine is the object. These are non-primitive values (cf. functionals are non-primitive operators). They can be used to support abstract data types and data values.

Recent work by Liskov and Zilles has crystalized the notion that a data type is characterized by the operators which manipulate instances of that type. In fact, they define a data type by giving a cluster of functions (operators) which support it. The machine given here carries this idea one step further: a data type is characterized by the collection of operators which implement it, described mathematically. A cluster of functionals is identified with a single abstract non-primitive value: the members of the cluster are the operations for manipulating that particular value. The value is considered to be a correct instance of some type if the cluster of functionals satisfy the mathematical description required of operators for that type.

An object is created with the <u>construct</u> operator

construct(name: functional, name: function, ...)

The functionals are associated with their names, and the whole is returned as a new non-primitive value having primitive type object. It is also regarded by interpretation as a value of some non-primitive type.

An object may be manipulated by employing the functionals which make up the object. These functionals are accessed through the operator operator.

operator(object, name)

Many representations of the same data type may be present in the machine at the same time. By judicious choice of abstractions, they may be made to interact quite successfully. This is because every correct object meets the (mathematical) requirements of the data type of which it is an instance; programs using the abstraction rely on the definition, and not on the implementation. For example, accessing the push operator for a stack gets different functionals for different stacks; perhaps even the text of the functionals is different because the stacks are differently represented.

It seems that the concept of type should be associated with each object of the system. This creates syntactic and semantic problems, not least of which is figuring a way (like bind?) to associate the abstraction with the 'program' which supports it. Another level of binding for types seems indicated. To do that, a primitive type type is being considered for inclusion in the machine. It is not easy; but it does seem strange not to have dynamically-**computated** types. In the machine, data types are not synonymous with the cluster which support them (cf. Liskov and Zilles); instead, many objects with different representations which are all instances of the same data type can be defined. The data type is independent of the cluster(s) supporting it.

This machine is adequate to model both abstract operations (functionals) and abstract values (objects). To support this claim, in the face of the difference (outlined above) between objects in this machine and the clusters of Liskov and Zilles, evidence is now offered that the piece of text which is a function. . cluster can be modelled in the machine. A cluster is a functional which takes the cluster parameters as its arguments, generates appropriate functionals (usually sharing the denotations of some of their externals) using those arguments, constructs from those functionals an object and returns it to its caller. This functional can be evaluated at run time to create objects.

In such object-producing functionals when the functionals used to create a cluster are given as literals, and the identifiers of the functional are used to bind externals of the operators, a syntactic shorthand can be used. This shorthand is called a template, because of its intent. When templates are used to create objects, implementations of this more general definition of data types can be reduced to one quite similar to that suggested by Liskov and Zilles.

I now include two examples. One is text for non-primitive operator which converts an integer to a string. The other is text of a template for objects of type rational; a rational in this cluster is represented as a pair of integers, a fact which should be of no interest to a user of the template.

integer-to-string

```
{ i, radix |
  j≡[Ø]
  k:[i]
  t≡['']
  if i eq Ø do
    resultis('Ø')
  if i ls Ø do
    k←-i
  { t←↑t‖((↑k rem radix)+αØ)
    k←(↑k)+radix } repeatuntil ↑k eq Ø
  if i ls Ø do
    t←↑t‖α-
  u≡['']
  n≡length(↑t)
  for i≡Ø to n-1 do
    u←↑u‖(get-character(↑t,(n-i)))
  resultis ↑u
}
```

template for rationals

```
{ n, d, rat-temp, gcd |
  return
  numerator { | resultis(n) }
  denominator { | resultis(d) }
  integer-part { | resultis(n+d) }
  fractional-part
    { |
      fn≡n-((n+d)×d)
      resultis((↑rat-temp)⟦'n'≡fn,'d'≡d⟧)
    }
  equal
    { v |
      vn≡v∘'numerator'⟦⟧
      vd≡v∘'denominator'⟦⟧
      resultis((n×vd)eq(d×vn))
    }
  greater-than
    { v |
      vn≡v∘'numerator'⟦⟧
      vd≡∘'denominator'⟦⟧
      resultis((n×vd)gr(d×vn))
    }
plus
  { v |
    vn=v∘'numerator'⟦⟧
    vd=v∘'denominator'⟦⟧
    sn=n*vd+v*dn
    sd=d*vd
    k=gcd⟦'a'=sn,'b'=sd⟧
    resultis((↑rat-temp)⟦'n'=sn+k,'d'=sd+k⟧)
  }
minus
  { v |
    vn=v∘'numerator'⟦⟧
    vd=v∘'denominator'⟦⟧
    sn=n*vd - v*dn
    sd=d*vd
    k=gcd⟦'a'=sn,'b'=sd⟧
    resultis((↑rat-temp)⟦'n'=sn+k,'d'=sd+k⟧)
  }
multiply
  { v |
    vn=v∘'numerator'⟦⟧
    vd=v∘'denominator'⟦⟧
    sn=n*vn
    sd=d*vd
    k=gcd⟦'a'=sn,'b'=sd⟧
    resultis((↑rat-temp)⟦'n'=sn+k,'d'=sd+k⟧)
  }
divide
  { v |
    vn=v∘'numerator'⟦⟧
    vd=v∘'denominator'⟦⟧
    sn=n*vd
    sd=d*vn
    k=gcd⟦'a'=sn,'b'=sd⟧
    resultis((↑rat-temp)⟦'n'=sn+k,'d'=sd+k⟧)
  }
}
```

gcd is an integer greatest-common-divisor functional. It is used to keep the representation of a rational in "lowest terms".

rat-temp is a cell containing a template for creating rationals - a template created from this very code perhaps. That is, this template is recursive: the functionals implementing a rational must be able to create other rationals - the sum and fractional-part, for example.

Just to complete the picture, the following operations would produce a template for rationals, assuming gcd denotes the needed integer gcd routine. The use of the cell and updating here is required to achieve the desired recursion.

```
rat-code≡' the code just given '
rat-t   ≡installc(rat-code)
rat-t1  ≡bindc(rat-t,'gcd',gcd)
rat-t2  ≡[Ø]
rat-t2  ←bindc(rat-t1,'rat-temp',rat-t2)
rat-temp≡↑rat-t2
```

The machine as described is not complete; it does not provide an adequate semantics for a complete computation system (e.g. a time sharing system). It needs primitives for creating and co-ordinating processes (a fork operator, and a primitive type semaphore (?)), and for talking to the outside world (a primitive type stream and a number of instances already implemented when the machine "starts" modelling its input and output devices).

Despite this incompleteness, the machine does provide a semantics for structured programming and programs, with explicit means (binding) for discussing the association of abstraction-denoting identifiers in programs (functionals) with the values supporting those abstractions (functionals, templates, and objects).

APPENDIX 1

Primitive Operators

This is a list of the primitive operators of the machine. The domains of each is given. U is the domain of all values. At the right is the special characters used in examples to shorten the expressions.

1. booleans (B):	and $(B,B) \rightarrow B$	$\wedge$
	or $(B,B) \rightarrow B$	$\vee$
	not $(B) \rightarrow B$	~
2. integers (Z):	plus(($Z,Z) \rightarrow Z$	+
	minus $(Z,Z) \rightarrow Z$	−
	multiply $(Z,Z) \rightarrow Z$	*
	divide $(Z,Z) \rightarrow Z$	/
	remainder $(Z,Z) \rightarrow Z$	rem
	equal $(Z,Z) \rightarrow B$	eg
	not equal $(Z,Z) \rightarrow B$	re
	less than $(Z,Z) \rightarrow B$	ls
	less than or equal $(Z,Z) \rightarrow B$	le
	greater than $(Z,Z) \rightarrow B$	gr
	greater than or equal $(Z,Z) \rightarrow B$	ge
3. strings (S):	extend $(S,Z) \rightarrow S$	‖
	equal string $(S,S) \rightarrow B$	
	length $(S) \rightarrow Z$	
	get character $(S,Z) \rightarrow Z$	
4. structures (N):	augment $(N,S,U) \rightarrow N$	
	select $(N,S) \rightarrow U$	.
	length $(N) \rightarrow Z$	
	selector $(N,Z) \rightarrow S$	
	is selector $(N,S) \rightarrow B$	
	equal structure $(N,N) \rightarrow B$	
5. cells (C):	newcell $(U) \rightarrow C$	[ ]
	update $(C,U) \rightarrow U$	←
	contents $(C) \rightarrow U$	↑
	same $(C,C) \rightarrow B$	

6. functionals (F):

```
install (S) → F
bind (F,S,U) → F
eval (F) → U
declare (S,U) → U
undeclare (S) → U
```

7. objects (O):

```
construct (S:F, S:F,...,S:F) → O
operator (O,S) → F
```

Some other notations used in the examples:

nil	the empty structure
⟨S=U, S=U,...,S=U⟩	a structure with the indicated selectors
F⟦S=U, S=U,...,S=U⟧	functional application: indicated bindings in the functional followed by evaluation
{	begin
}	end
αletter	the integer character code of letter