ALGEBRAIC DENOTATIONAL SEMANTICS USING PARAMETERIZED ABSTRACT MODULES

Joseph A. Goguen Kamran Parsaye-Ghomi
SRI International Computer Science Dept.
Menlo Park CA 94025 UCLA CA 90024

ABSTRACT: This paper describes a method for giving structured algebraic denotational definitions of programming language semantics. The basic idea is to use parameterized abstract data types to construct a directed acyclic graph of modules, such that each module corresponds to some feature of the language. A "feature" in this sense is sometimes a syntactic construction, and is sometimes a more basic language design decision. Our definitions are written in the executable algebraic specification language OBJT. Among the advantages of our approach are the following: it is relatively easier to understand the definitions because they are organized into modules and use flexible user-definable syntax; it is also relatively easy to modify or to extend the definitions, not only because of the modularity, but also because of the use of parameterization; it is possible to debug the definitions by executing test cases, which in this case are programs; the definitions are relatively compact; and they impose relatively little implementation bias. This paper illustrates these points with the definition of a modest programming language with integer and boolean expressions, blocks, iteration, conditional, input and output, and side-effect-only procedures, which can be assigned to variables and passed as parameters.

1. Introduction

There is a growing realization of the necessity for structuring semantic definitions in general, and programming language definitions in particular [Bjorner & Jones 78], [Burstall & Goguen 80], [Constable & Donahue 79], [Jones 80], [Mosses 79]. For it is impossible to correctly read, write, or modify large unstructured specifications. This paper describes a method for constructing hierarchically structured algebraic denotational definitions of programming language semantics using parameterized abstract data types. We have used the executable algebraic specification language OBJT [Goguen 77], [Goguen & Tardo 79].

Our semantic definitions are denotational, in the sense that meanings are always elements of particular abstract data types, or else are abstractly defined functions among them. Our definitions are compositional, in the sense that the meaning of each syntactic phrase is composed from the meanings of its component phrases in a way that respects syntactic structure. The importance of these ideas has been emphasized by the very successful work of the Scott-Strachey school of denotational semantics. The idea of compositionality can apparently be traced back to Frege.

In many ways, our definitions are standard denotational definitions, and the terminology used in the example in the Appendix reflects this. Our approach differs from standard denotational semantics first in that we structure definitions as directed acyclic graphs of modules, such that each module only makes use of modules lying below it in the

1) Work supported in part by Office of Naval Research, Contract N00014-80-C-0296 and National Science Foundation, Contract 442510-21713.
graph structure. Each module in the graph defining a programming language corresponds to some feature of the language, where a "feature" in this sense might be a syntactic feature with its semantic definition, or might represent some more basic aspect of the language design, such as its type-checker for expressions. Each module is an algebraic abstract data type definition, possibly parameterized. Our definitions also differ from standard denotational semantics in that they can be immediately executed (interpretatively), in their powerful exception handling capability, and in that they are only first order. (Section 5 indicates how to lift this restriction.) We have not found need for what is perhaps the most original aspect of the work of Scott and Strachey, namely recursive domain definitions [Scott 76]. But we have built upon ideas of how to define various language features which arose within their school; see [Gordon 79] or [Tennent 76] for good introductions to these techniques. See [Wand 77] for a similar, but unstructured, approach to defining programming language semantics with equations.

Our work suggests three principles for modularizing semantic definitions: (1) modules should be conceptually orthogonal; (2) module interfaces should be abstract, in the sense that no use is made of how structures and operations are defined; and (3) parameterized modules should be utilized wherever appropriate. None of these are standard techniques of denotational semantics. However, [Mosses 79] has recently done important work that can be seen to embody similar principles.

These three principles make semantic definitions easier to create, to understand, and to modify, because they help with partitioning into meaningful modules. These principles also ensure that modules are maximally reusable in definitions of other programming languages. For example, a module for defining DO-WHILE, that does not depend on how states are defined, such as whether or not there is a symboltable, can be used to define that feature in any language.

We have found that it is a great help to run test cases, because even fairly simple definitions are usually wrong as first written and need to be debugged in the same way that programs are. We have also found it valuable to structure the testing process to correspond with that of the semantic definition. Thus, one should test each module (i.e., each feature) before building other modules on top of it. Once the definition has evolved far enough, test cases are actually simple programs in the language, and one can see if the semantics one has defined is the semantics one really wants. Our experience with executing specifications has left us little trust in large formal specifications which have not been thoroughly tested. A further advantage of modularization is that the test cases can also often be reused.

Flexible user-definable syntax renders the expressions that occur in definitions and in test cases more easily readable, and thus facilitates the processes of incremental specification and debugging. Strong typing and mechanical type-checking are useful in preventing bugs.

Abstract algebraic definitions are relatively compact and impose relatively little implementation bias. Use of a high level programming language, such as ALGOL68, as advocated for example in [Pagan 79], would significantly diminish these advantages.

Parameterization is also useful in constructing definitions. For example, there are typically many instances of LIST, e.g., lists of identifiers in declarations, and lists of data items in an output file. One does not want to write the same axioms again for each of these; one wants a LIST module, the parameter of which can be instantiated for any desired kind of list. Parameterized modules can also be used in more complex ways. For example, one can construct a parameterized SYMBOLTABLE module by applying the parameterized STACK module to a parameterized ARRAY module. In general, the parameterized modules in a semantic definition form acyclic subgraphs which are used to efficiently construct the main graph.
OBJT's capabilities for defining exception conditions and exception handling are useful for defining the error messages which a compiler should produce. Our MODEST example in the Appendix does not go as far with this as we would like, but it does illustrate the technique. That previously defined modules generate specific and meaningful error messages is a great help in debugging. Furthermore, resource errors can be modeled in the same framework.

A powerful library of modules would constitute a "programming language designer's workbench." For example, one could reassemble the modules into new patterns, and then execute test cases to see how the resulting languages compare with one another; or one could substitute new modules for old ones and see what happens with some test cases.

The operational utility obtained by using OBJT in this way is similar to that provided by the SIS system of [Mosses 78]; the main differences are due to the modular and interactive capabilities of OBJT, and the higher order capabilities of SIS. For example, SIS produces a compiler, whereas OBJT provides an interpretative capability. It would be possible to give similar definitions in any sufficiently expressive structured specification language; for example, [Mosses 78] has used a variant of CLEAR. The advantages of OBJT are that it is currently implemented and is completely executable, so that one can actually run test cases. On the other hand, it lacks some desirable features, such as theorem proving, and higher-order specifications.

2. The Specification Language OBJT.

This section briefly describes the executable specification language OBJ [Goguen 77], in the form of the OBJT implementation by J. Tardo in LISP for PDP10 machines [Goguen & Tardo 79]. An OBJ "object" is similar to a SIMULA class, a CLU cluster, or an ALPHARD form, except that OBJ objects do not involve representations, and therefore provide truly abstract data type definitions. However, "relativized objects," which have been built on other objects, impart a rather abstract kind of representation bias, because of the specific ways in which they use previous objects.

OBJT objects have the syntactic form

```
OBJ <obj-name> / <obj-name-list>
SORTS <sort-list>
OK-OPS <op-form-list>
ERROR-OPS <op-form-list>
VARS <id-decl-list>
OK-EQNS <eq-list>
ERROR-EQNS <eq-list>
```

OBJ and JBO indicate the beginning and ending of definitions. The name of the object follows OBJ, and then, after a slash (/), any previously defined objects used in the definition must be listed. Following SORTS is a list of the new types (technically called "sorts" to avoid confusion) introduced in the object; new operations are introduced either as OK-OPS or ERR-OPS. Ordinary operations are OK-OPS, while operations which provide error messages are ERR-OPS. Constants are viewed as nullary operations; for example, NIL : -> LIST. OBJT permits users to define their own "mix-fix" syntax, using the underscore _ as a placeholder. For example, (_;_ : LIST LIST -> LIST) declares an infix operation on LISTS. If no underscores are used, then the default prefixed operator with parentheses around the arguments list is used. (ASSOCIATIVE) indicates that an operation is associative, both syntactically and semantically. Underbar by itself indicates a "coercion" between sorts; thus, _ : ELEMENT -> LIST indicates that each ELEMENT is also to be considered a singleton LIST. The sort and operation declarations, without the variable declarations or the equations, constitute what is called the signature of the object.
OBJ LIST
SORTS LIST ELEMENT
OK-OPS
  NIL : -> LIST
  _ : ELEMENT -> LIST
  _ _ : LIST LIST -> LIST (ASSOCIATIVE)
  FIRST : LIST -> ELEMENT
  REST : LIST -> LIST
ERR-OPS
  NO-FIRST : -> ELEMENT
  NO-REST : -> LIST
VARS
  L : LIST
  E : ELEMENT
OK-EQNS
  (NIL ; L = L)
  (L ; NIL = L)
  (FIRST(E ; L) = E)
  (REST(E ; L) = L)
  (FIRST(E) = E)
  (REST(E) = NIL)
ERR-EQNS
  (FIRST(NIL) = NO-FIRST)
  (REST(NIL) = NO-REST)
JBO

It is easy to treat resource errors in the same framework. To express that OVERFLOW
occurs for lists over 108 elements long, first add LENGTH : LIST -> INT as an OK-OP and
OVERFLOW as an ERR-OP to the signature; then define LENGTH with OK-EQNS; finally, add the
ERR-EQN (E : L = OVERFLOW IF LENGTH L > 107).

OBJT executes expressions by treating equations as rewrite rules and trying to match
the left-hand sides of equations. ERR-EQNS rules are always tried before OK-EQNS rules.
If an expression matches an error equation, then it is an error, and only error equations
can be applied to it. The syntax for executing an expression <exp> is RUN <exp> NUR.

Three objects are built into OBJT: ID, for identifiers, of the form 'A, 'B, etc.;
BOOL, for booleans, with constants T and F, and operations AND, OR, NOT, as usual; and
INT, the integers, with the expected operations, such as + and *. OBJT also provides an
equality operation of syntactic form (== : S S -> BOOL) for each sort S. Comments are
of the form *** <text> ***.

Parameterized objects are instantiated with OBJT's IMAGE facility with a "copy and edit" rule. For example,

IM (LIST => ID-LIST) / ID
SORTS (LIST => ID-LIST)
  (ELEMENT => ID)
MI

where * indicates zero or more instances of what precedes it.
OBJT has been used to specify many things other than programming languages. For example, [Goguen & Tardo] specifies an airline reservation information system.

3. The MODEST Programming Language.

MODEST has block structure, strong typing, and procedures with side effects, but not values. Procedures can be assigned to variables, and can appear as parameters to other procedures. MODEST is a simple language, but serves our illustrative purpose well. This section motivates the definition given in the Appendix as it came off SRI's KL. In many ways this is a "standard" denotational definition, and our terminology reflects this.

The definition begins with a library file of particularly basic parameterized objects: PAIR, ARRAY, LIST, and STACK. PAIRS of elements have the form \(<\text{LEFT};\text{RIGHT}>\) and ARRAYS are accessed with a mix-fix operation \([\_]\), so that \(A[1]\) is the contents of the 1-th location in array A. LISTS are constructed with an infix ";", operation, and STACKs are PUSHed and POPed in the usual way. The first three of these parameterized objects correspond to familiar operations on domains in denotational semantics: PAIR corresponds to the Cartesian product of domains; ARRAY corresponds to the domain of functions between two domains (note, however, that we only get finite functions); and LIST corresponds to *.

The storable values of MODEST, called ITEMS, are defined using IMAGE on PAIR, to consist of a TYPE and a VALUE. When ITEM is defined, there are not yet objects which define TYPE or VALUE; thus these are parameters.

Input and output appear on TAPEs, defined as LISTS of ITEMS; an I/O-TAPE is a PAIR of TAPEs, one for INPUTs and one for OUTPUTs. A STORE is an ARRAY of ITEMS, indexed by LOCations, which are created by a coercion from INTEGERS. For example, if the ITEM \(<\text{INT};3>\) is stored in LOCATION 25 of store S, then \(S[25] = <\text{INT};3>\). STATES are (STORE,I/O-TAPE) PAIRS. The operation ALLOCATE produces a new LOCATION, and INITIALIZE places an ITEM there with UNDEFINED value.

ENVironments associate the values of identifiers with LOCATIONS. For example, if the typed value \(<\text{INT};3>\) of identifier 'A' is in LOCATION 25, then the identifier 'A' is bound to 25. Because MODEST is block structured, its binding table must be a SYMBOLTABLE. We define them as STACKs of ARRAYS of INTEGERS, indexed by IDENTifiers. The EXITBLOCK operation POPs the latest LAYER when exiting a block. Given an identifier and an environment, GET produces the location where the typed value of the identifier is stored, and RETRIEVE produces the typed value.

The object EXPRESSION defines the operations TYPE and VALUE, which respectively return the type and the value of an expression, given an environment and a store. The objects INT-EXP and BOOL-EXP define the syntax and semantics of integer and boolean expressions respectively, and the object EXPRESSIONS provides an interface to the rest of the definition.

EXECUTION defines a PROGRAM as a statement list and an input tape. EXECUTing a program produces an output tape, using the operation EVAL, that takes a statement list, an environment, and a store, and returns the store resulting from execution of the statement list.

Identifiers are declared in DECL-LISTS, which are lists of pairs of the form (ID : TYPE). DECLARE-ENV binds an identifier to a location, and DECLARE places the TYPE of the identifier in the STORE with UNDEFINED value. A BLOCK is a statement, produced from a declaration list and a statement list, using the delimiters BEGIN and END.

ASSIGNMENT, INPUT-OUTPUT, CONDITIONAL, and WHILE statements are defined in succeeding objects. STATEMENTS provides an interface at the statement level with the rest of the definition.
MODEST procedure bodies, such as

```
BEGIN 'A : INT ; 'A := 'X t PRINT ('A + 'X) END
```

are defined in PROC-BODY to be (ID-LIST, STMT-LIST) PAIRS. Static bindings are modeled with (PROC-BODY, ENVironment) pairs, called CONTOURs. Recalling that the ITEMS are typed, we coerce CONTOURs to VALUES, and we create TYPES for them with PROC[], defined on procedure argument type lists. These definitions permit procedures to be assigned as VALUES, and passed as parameters to other procedures.

The object PROCEDURES coerces PROC-DECLs to DECLARATIONS, and PROC-DECLARATION extends the operations DECLARE-ENV and DECLARE. DECLARE-ENV associates a procedure name with the location where its CONTOUR is stored, and DECLARE places this contour in that location. For example, if

```
PROC 'P ['X : INT ] BEGIN 'A : INT ;
 'A := ('X + 'A); PRINT 'X END
```

is defined in environment El, then DECLARE-ENV binds the name 'P to a new location, say 31, in El; and DECLARE places the item containing the contour of 'P,

```
< PROC[ INT ] :
 < 'X ; < BEGIN 'A : INT ; 'A := ('X + 'A); PRINT 'X END > ;
 El > >
```

in location 31 of the store.

Call-by-value parameter passing is defined in the module PARAM-PASS-BY-VALUE. When a procedure is called, PASS-ENV binds its formal parameters to some new locations in the store, and PASS places the values of the actual parameters in these locations. For example, if procedure 'p above is called in environment E2, with CALL 'P ['Y], then PASS-ENV binds 'X to a new location, say 72, in the calling environment El of 'P, and then PASS places the value of 'Y in E2, say < INT : 3 >, in location 72. Procedure execution is defined in the module CALL-BY-VALUE. Parameters are passed by value, and procedure bodies are EXECUTEd in the declaring environment.


It is increasingly widely recognized that it is meaningless to give specifications unless the formalism in which they are expressed has a precise mathematical meaning. This section briefly discusses some difficulties and achievements in this area.

It is necessary to distinguish between two aspects of the semantics of a specification language. The first aspect is its underlying logical formalism (called its institution in [Burstell & Goguen 80]); this provides the basic notions of signature, axiom, and model (a theory over a signature is just a set of axioms over that signature). The second aspect of specification language semantics is the meaning of its structuring devices, such as modularization and parameterization.

Of course, specifications that are just sets of axioms in some standard institution, such as first order logic need no special semantic foundations. However, if a specification method uses either some unusual logic, or any nontrivial structuring devices, then a formal account must be given of how these work. It is shocking that so much of the literature on specification languages does not even attempt to define the basic logic involved, let alone the structuring devices.
OBJT is based on initial algebra semantics [Goguen, Thatcher, Wegner & Wright 77]. We hereafter identify the notions of object and presentation; both consist of a signature and a set of equational axioms over that signature. We take the denotation of an unparameterized basic OBJT object to be its initial algebra. (The semantics of a "relativized object", that is an object with a slash and some other object names after its name, is discussed later under the heading of structuring devices.) It is very convenient not to have to rely upon previously constructed subtle domains of various kinds as required in standard denotational semantics.

OBJT also embodies an operational semantics in which equations are regarded as rewrite rules; [Goguen 80] proves that under certain conditions these two semantics agree. Thus, despite the contrary claim in the generally excellent paper [Wand 77], such a rewrite rule semantics is also necessarily an initial algebra semantics. Incidentally, the conditions for agreement are general enough to include cyclic rules, such as arise from a commutative equation for a binary operation, and OBJT is able to execute such rules. [Reo & Vuillemin] show that operational rewrite semantics agrees with fix-point semantics for a restricted class of rewrite rules.

The structural semantics of OBJT is simple enough that it can easily be described and understood informally. It should also be described formally, but this has not yet been done. As far as we are aware, the only specification language having nontrivial structuring devices that has ever been given a formal semantics is CLEAR [Burstall & Goguen 77]. [Burstall & Goguen 80] [Mosses 79] uses CLEAR as a formal foundation. OBJ can be viewed as an implementation of an executable subset of CLEAR.

The meaning of a relativized unparameterized OBJT object is the reduct to the signature of that object, of the initial algebra of the combined signatures and equations of all the objects which lie below the given one in the graph structure. If an object has parameters in it, then its initial algebra generally has some empty carriers, so we do not want this algebra as the denotation of the object. Rather, we regard it as a purely formal entity, that does not have a denotation until its parameters have been instantiated using the IMAGE facility. The meaning of IMAGE is simply given by a copy and edit rule: it creates one formal object text from another formal object text by copying and then editing it in the indicated way. When there are no parameters left, it makes sense to consider the initial algebra denotation of the formal object. (A more sophisticated treatment of parameterized objects could be given along the lines of CLEAR, but OBJT lacks a mechanism for defining requirement theories that actual parameters must satisfy before the application can be considered reasonable.)

There are at least four important respects in which the simple institution of equational signatures and axioms with algebras as models is not adequate for OBJT. The first is relatively straightforward: we do not want just any algebra that satisfies the equations as a model for a set of equations, but only the initial algebra. Initiality can be incorporated into the equational institution using the "data constraints" of [Burstall & Goguen 80]. The second direction in which the equational institution must be extended is to handle errors or exceptions. [Goguen 77] indicates how this can be done, but there are still some gaps in the theoretical machinery. The third extension adds sub-, union-, and intersection- sorts, to handle the semantics of coercions [Goguen 78]. The fourth extension concerns equations which do not define finite structures, e.g., for nonterminating programs; [Goguen, Thatcher, Wegner & Wright] suggest using continuous theories and algebras for this problem. Much more work remains to be done in this area.

We use initial algebras to define programming language semantics differently than [Goguen 74], which noted that abstract syntax (in essentially the sense of [McCarthy 62]) is an initial " anarchic" algebra whose signature comes from the grammar, so that semantics can be given by a "semantic" algebra of the same signature, with meaning the unique
homomorphism. The approach of this paper is closer to that advocated by [Wand 77], in that the meaning function is an operation in a complex data type which includes the run-time environment of the programming language; i.e., it is essentially an interpreter [Reynolds 70], [Landin 66].

5. Conclusions and Extensions.

We have defined various programming language features using a hierarchy of parametrized abstract modules. While the syntax of features varies from language to language, their semantic definitions should not; moreover, the syntactic forms are easily changed if desired. A library of such modules can therefore be useful for language design and specification. The modules presented here are just a beginning, but we hope that a community of interest will arise to assist in extending and testing this collection.

This paper has not addressed program verification. It seems that a very different approach than Floyd-Hoare verification conditions will be appropriate. We suggest that to prove an assertion A about a program P in a language L with an algebraic denotational definition, one should first define P as a Boolean-valued function, and then prove the equation A(P) = T using the definitions of A, P and L. Since it may well be necessary to use induction in such proofs, it is important to find mechanical ways of doing so; see [Musser 80] and [Goguen 80].

An obvious extension of the MODEST definition would include procedures with values. There is no great difficulty in this; it should suffice to coerce procedures of value sort S to items of sort S, and to use continuations.

Finally, we note that the equational definition technique would be improved by using higher-order operations. For example, the equations

\[\text{VALUE}((\text{EXP} + \text{EXP'}), \text{ENV}, \text{STATE}) = \text{VALUE}((\text{EXP}, \text{ENV}, \text{STATE}) + \text{VALUE}(\text{EXP'}, \text{ENV}, \text{STATE})\]

\[\text{VALUE}((\text{EXP} \times \text{EXP'}), \text{ENV}, \text{STATE}) = \text{VALUE}((\text{EXP}, \text{ENV}, \text{STATE}) \times \text{VALUE}(\text{EXP'}, \text{ENV}, \text{STATE})\]

\[\text{VALUE}((\text{EXP} - \text{EXP'}), \text{ENV}, \text{STATE}) = \text{VALUE}((\text{EXP}, \text{ENV}, \text{STATE}) - \text{VALUE}(\text{EXP'}, \text{ENV}, \text{STATE})\]

are all of the same form. If we had a higher order variable \text{FUN}, ranging over the binary operation symbols in \text{EXPRESSION}, then the equation

\[\text{VALUE}(\text{FUN}(\text{EXP}, \text{EXP'}), \text{ENV}, \text{STATE}) = \text{FUN}(\text{VALUE}(\text{EXP}, \text{ENV}, \text{STATE}), \text{VALUE}(\text{EXP'}, \text{ENV}, \text{STATE}))\]

would subsume all such equations and significantly reduce the size of the specification.

Higher order variables would also be useful for denoting continuations; we could then avoid having to indirectly define continuations using the \text{ARRAY} parameterized type constructor. Thus, a higher order capability would enable algebraic denotational semantics to more completely unify the wisdom and experience of the denotational tradition with the structuring mechanisms of the algebraic tradition.

Of course, higher order algebraic specifications will not be meaningful without a theory of higher order algebraic data types. We have developed such a theory and hope to present it soon [Parsaye 81].
We thank Peter Dybjer, Dwight Harm, David Smallberg, and Maria Zamfir for their contributions to the specification project that has been going on at UCLA for some years now. The definition given here builds on earlier similar efforts by these people. We particularly thank Joe Tardo for his OBJT implementation of OBJ, without which the whole effort would have been impossible. We also wish to thank Prof. D. Stott Parker of UCLA for his kind help in facilitating the authors' communication.

We would also like to thank Peter Strachey, Dana Scott, Peter Mosses and Mitchell Wand for their inspiring work in this area. Goguen also wishes to thank Rod Burstall, Jim Thatcher, Eric Wagner, and Jesse Wright for their collaboration, encouragement, and friendship.
APPENDIX: Definition of MODEST

[PHOTO: Recording initiated Tue 30-Dec-80 11:27PM]

***OBJ 12/2/79

*** HERE IS A BASIC LIBRARY OF PARAMETERIZED TYPES ***

OBJ PAIR
SORTS PAIR LEFT RIGHT
OK-OPS

< ; • : LEFT RIGHT -> PAIR

LEFT : PAIR -> LEFT

RIGHT : PAIR -> RIGHT

VARS

LEFT : LEFT

RIGHT : RIGHT

P : PAIR

OK-EQNS

(LEFT < LEFT ; RIGHT > = LEFT)

(RIGHT < LEFT ; RIGHT > = RIGHT)

(< LEFT P ; RIGHT P > = P)

JBO

OBJ ARRAY / BOOL
SORTS ARRAY INDEX ELEMENT
OK-OPS

NIL-ARRAY : -> ARRAY

PUT : INDEX ELEMENT ARRAY -> ARRAY

[] : ARRAY INDEX -> ELEMENT

IN_ : INDEX ARRAY -> BOOL

ERR-OPS

UNDEF : INDEX -> ELEMENT

VARS

A : ARRAY

I I' : INDEX

ELM : ELEMENT

OK-EQNS

(PUT(I,ELM,A)[ I ] = ELM)

(PUT(I,ELM,A)[ I' ] = A [ I' ] IF NOT I = I')

(I IN NIL-ARRAY = F)

(I IN PUT(I',ELM,A) = I == I' OR I IN A)

ERR-EQNS

(A [ I ] = UNDEF(I) IF NOT I IN A)

JBO

OBJ LIST
SORTS LIST ELEMENT
OK-OPS

NIL : -> LIST

FIRST : LIST -> ELEMENT

REST : LIST -> LIST

_: LIST LIST -> LIST (ASSOCIATIVE)

ERR-OPS

NO-FIRST : -> LIST

NO-REST : -> LIST

VARS

L : LIST

E E' : ELEMENT

OK-EQNS

(NIL ; L = L)

(L ; NIL = L)

(FIRST(E ; L) = E)

(REST(E ; L) = L)

(REST(E) = NIL)

ERR-EQNS

(FIRST(NIL) = NO-FIRST)

(REST(NIL) = NO-REST)

JBO

OBJ STACK / BOOL
SORTS STACK ELEMENT
OK-OPS

EMPTY : -> STACK

POP : STACK -> STACK

PUSH : ELEMENT STACK -> STACK

TOP : STACK -> ELEMENT

EMPTY? : STACK -> BOOL

ERR-OPS

UNDERFLOW : -> STACK

NO-TOP : -> ELEMENT

VARS

ELM : ELEMENT

S : STACK

OK-EQNS

(PUSH(EIM,S) = E)

(TOP PUSH(EIM,S) = EIM)

(EMPTY? PUSH(EIM,S) = F)

(EMPTY? EMPTY = F)

ERR-EQNS

(PUSH EMPTY = UNDERFLOW)

(TOP EMPTY = NO-TOP)

JBO

*** WE NOW DEFINE THE BASIC COMPONENTS OF STATES ***

*** THE STORABLE VALUES OF MODEST ARE TYPE-VALUE PAIRS WHICH ARE CALLED ITEMS ***

IM (PAIR -> ITEM)

SORTS (PAIR -> ITEM)

(LEFT -> TYPE)

(RIGHT -> VALUE)

OPS (< ; • ; > : LEFT RIGHT -> PAIR = < ; • ; >)
(LEFT_ : PAIR -> LEFT => TYPE-OF_)

MI

IM (LIST > TAPE)/ ITEM
SORTS (LIST => TAPE)
(ELEMENT => ITEM)
OPS (NO-REST ; -> LIST => END-OF-TAPE)

MI

IM (PAIR > I/O-TAPES)/ TAPE
SORTS (PAIR => I/O-TAPES)
(LEFT => TAPE)
(RIGHT => TAPE)
OPS (LEFT : PAIR -> LEFT => INPUT-OF_)
(RIGHT : PAIR -> RIGHT => OUTPUT-OF_)

JI

*** AN ABSTRACT LOCATION MODULE ***

OBJ LOC / INT
SORTS LOC
OK-OPS

; INT -> LOC
NEXT_ : LOC -> LOC

VARS
I : INT
OK-EQNS
(NEXT I = I + 1)

JBO

IM (ARRAY => STORE)/ LOC ITEM BOOL
SORTS (ARRAY => STORE)
(ELEMENT => ITEM)
(INDEX => LOC)
OPS (NIL-ARRAY : -> ARRAY => NIL-STORE)

MI

IM (PAIR => STATE)/ STORE I/O-TAPES BOOL
SORTS (PAIR => STATE)
(LEFT => STORE)
(RIGHT => I/O-TAPES)
OPS (LEFT_ : PAIR -> LEFT => MEMORY-OF_)
(RIGHT_ : PAIR -> RIGHT => TAPE-OF_)

MI

OBJ MAKE-STATE / STATE
OK-OPS

PUT : LOC ITEM STATE -> STATE
[ ] : STATE LOC -> ITEM
IN : LOC STATE -> BOOL
NIL-STATE : -> STATE

VARS

ITEM : ITEM
STORE : STORE
L : LOC
TAPE : I/O-TAPES

OK-EQNS

(PUT(L, ITEM, < STORE ; TAPE >) = PUT(L, ITEM, STORE ; TAPE >)
( < STORE ; TAPE > [ L ] = STORE [ L ]
(L IN STORE ; TAPE > = L IN STORE)

JBO

OBJ I/O / MAKE-STATE
OK-OPS

NEXT-INPUT : STATE -> ITEM
NEXT-OUTPUT : ITEM STATE -> STATE
INITIAL-STATE : TAPE -> STATE
SET-INPUT : STATE -> STATE

VARS

STORE : STORE
IN-STATE OUT-TAPE : TAPE
STATE : STATE
ITEM : ITEM

OK-EQNS

(NEXT-INPUT(< STORE ; < IN-TAPE ; OUT-TAPE > ) = FIRST(IN-TAPE))
(NEXT-OUTPUT(ITEM, < STORE ; < IN-TAPE ; OUT-TAPE > ) = < STORE ; < IN-TAPE ; (OUT-TAPE ; ITEM) > )
(INITIAL-STATE(IN-TAPE) = < NIL-STORE ; IN-TAPE ; NIL > )
(SET-INPUT(< STORE ; < IN-TAPE ; OUT-TAPE > ) =< STORE ; < REST(IN-TAPE); OUT-TAPE > )

JBO

OBJ ALLOCATION / I/O
OK-OPS

ALLOCATE : STATE -> LOC
INITIALIZE : TYPE STATE -> STATE
INITIALIZE : ITEM STATE -> STATE
FIND-NEXT : LOC STATE -> LOC
UNDEFINED : ITEM -> VALUE

VARS

TYPE : TYPE
STATE : STATE
ITEM : ITEM
LOC : LOC

OK-EQNS

(ALLOCATE(STATE) = FIND-NEXT(1, STATE))
(FIND-NEXT(LOC, STATE) = IF NOT LOC IN STATE THEN LOC ELSE FIND-NEXT((NEXT LOC, STATE) FI)
(INITIALIZE(TYPE, STATE) = PUT(ALLOCATE(STATE), < TYPE : UNDEFINED ; STATE >))
(INITIALIZE(ITEM, STATE) = PUT(ALLOCATE(STATE), ITEM, STATE))
**JBO**

**IM (ARRAY => LAYER)/ ID LOC BOOL**

SORTS  (ARRAY => LAYER)
       (INDEX => ID)
       (ELEMENT => LOC)

OPS  (NIL-ARRAY : -> ARRAY => NIL-LAYER)
        MI

**IM (STACK => SYMBOL-TABLE)/ LAYER BOOL**

SORTS  (STACK => ENV)
       (ELEMENT => LAYER)

OPS  (EMPTY : -> STACK => NIL-ENV)
        (POP_ : STACK -> STACK => EXITBLOCK)
        MI

**IM (LIST => ID-LIST)/ ID**

SORTS  (LIST => ID-LIST)
       (ELEMENT => ID)

**MI**

**OBJ ENVIRONMENT / SYMBOL-TABLE ID-LIST ALLOCATION**

OK-OPS

ENTERBLOCK : ENV -> ENV
GET : ENV ID -> LOC
RETRIEVE : ID ENV STATE -> ITEM
BIND : ID-LIST ENV STATE -> ENV

ERR-OPS

UNDECL : ID -> LOC
ALREADY-DECLARED-IN-BLOCK : ID -> ENV

**VARS**

ENV : ENV
ID : ID
LAY : LAYER
STATE : STATE
ID-L : ID-LIST

OK-EQNS

(GET(ENV, ID)=(TOP ENV)[ ID ] IF ID IN TOP ENV)
(GET(ENV, ID)= GET(EXITBLOCK ENV, ID) IF NOT(ID IN TOP ENV))
(RETRIEVE(ID, ENV, STATE)= STATE[ GET(ENV, ID)])
(EXITBLOCK ENV = PUSH(NIL-LAYER, ENV))
(BIND(ID, PUSH(LAY, ENV), STATE)= PUSH(PUT(ID, ALLOCATE(STATE), LAY), ENV))
(BIND(ID; ID-L, ENV, STATE)= BIND(ID-L, BIND(ID, ENV, STATE), (INITIALIZE(UNDEFINED, STATE))))

ERR-EQNS

(GET(NIL-ENV, ID)= UNDECL(ID))
(BIND(ID, PUSH(LAY, ENV), STATE)= ID
ALREADY-DECLARED-IN-BLOCK IF(ID IN LAY))

**JBO***

**OBJ EXPRESSION / ENVIRONMENT**

SORTS EXP

OK-OPS

: ID -> EXP
   VALUE : EXP ENV STATE -> ITEM
   TYPE : EXP ENV STATE -> TYPE

ERR-OPS

* DOES-NOT-MATCH_ : TYPE TYPE -> VALUE

**VARS**

EXP : EXP
ENV : ENV
STATE : STATE
ID : ID

OK-EQNS

(TYPE(EXP, ENV, STATE)= TYPE-OF VALUE(EXP, ENV, STATE))
(VALEU(ID, ENV, STATE)= RETRIEVE(ID, ENV, STATE))

**JBO**

**OBJ INT-EXP / EXPRESSION**

OK-OPS

INT : -> TYPE
   : INT -> VALUE
   INT-VAL : ITEM -> INT
   : INT -> EXP
   + : EXP EXP -> EXP
   * : EXP EXP -> EXP
   - : EXP EXP -> EXP

**VARS**

I : INT
ENV : ENV
STATE : STATE
EXP EXP' : EXP
TYPE : TYPE
VALUE : VALUE

OK-EQNS

(INT-VAL(< INT : I >)= I)
(VALUE(I, ENV, STATE)= < INT : I >)
(VALUE(EXP + EXP', ENV, STATE)= < INT : ((INT-VAL(
   VALUE(EXP, ENV, STATE))+ INT-VAL(VALUE(EXP',
   ENV, STATE))))

(VALUE(EXP * EXP', ENV, STATE)= < INT : ((INT-VAL(
   VALUE(EXP, ENV, STATE))* INT-VAL(VALUE(EXP',
   ENV, STATE))))

(VALUE(EXP - EXP', ENV, STATE)= < INT : ((INT-VAL(
   VALUE(EXP, ENV, STATE))- INT-VAL(VALUE(EXP',
   ENV, STATE))))

ERR-EQNS

* (INT-VAL( TYPE : VALUE )= TYPE DOES-NOT-MATCH
   INT IF(NOT(TYPE == INT)))

**JBO**

**OBJ BOOL-EXP / EXPRESSION**
**OK-OPS**

- BOOL -> TYPE
- TYPE -> VALUE
- BOOL-VAL: ITEM -> BOOL
- BOOL -> EXP
- AND: EXP EXP -> EXP
- OR: EXP EXP -> EXP
- NOT: EXP -> EXP
- EQ: EXP EXP -> EXP

**ERR-OPS**

- TYPE-CONFLICT: ITEM

**VARS**

- B: BOOL
- TYPE: TYPE
- ENV: ENV
- STATE: STATE
- VALUE: VALUE
- EXP EXP': EXP

**OK-EQNS**

- (BOOL-VAL(< BOOL ~ B,)= B)
- (VALUE(B, ENV, STATE)= < BOOL :
  (BOOL-VAL(VALUE(EXP, ENV, STATE))AND BOOL-VAL(
    VALUE(EXP', ENV, STATE))))
- (VALUE(EXP OR EXP', ENV, STATE)= < BOOL :
  (BOOL-VAL
    (VALUE(EXP, ENV, STATE))OR BOOL-VAL(VALUE(
    EXP', ENV, STATE))))
- (VALUE(EXP EQ EXP', ENV, STATE)= < BOOL :
  (VALUE
    (EXP, ENV, STATE) == VALUE(EXP', ENV, STATE)))
- (VALUE(NOT EXP, ENV, STATE)= < BOOL :
  (NOT BOOL-VAL
    (VALUE(EXP, ENV, STATE))))

**ERR-EQNS**

- (BOOL-VAL(< TYPE : VALUE >)= TYPE DOES-NOT-MATCH
  BOOL IF NOT TYPE == BOOL)
- (VALUE(EXP EQ EXP', ENV, STATE)= TYPE-CONFLICT IF
  NOT TYPE(EXP, ENV, STATE) == TYPE(EXP', ENV, STATE))

**JBO**

***FIRST SEMICOLON***

- OBJ SEMICOLON / EXECUTION

**VARS**

- STMT: STMT
- STMT-L: STMT-LIST
- ENV: ENV
- STATE: STATE

**OK-EQNS**

- (EVAL(STMT-L, ENV, STATE)= EVAL(STMT-L, ENV)
  EVAL(STMT, ENV, STATE))

**JBO**

***DEFINE BLOCK STRUCTURE***

- IM (LIST -> DECL-LIST)
- SORTS (LIST -> DECL-LIST)
- ELEMENT -> DECLARATION
- OPS (NIL -> LIST -> NILDECL)

**VARS**

- D: DECLARATION
- DL: DECL-LIST
- ENV: ENV
- ID: ID
- TYPE: TYPE
- ID-L: ID-LIST
- STATE: STATE

**OK-EQNS**

- (DECLARE(D : DL, STATE, ENV)= DECLARE(DECLARE(D, STATE, ENV)
  DECLARE-ENV(D, ENV, STATE))
- (DECLARE-ENV(D : DL, ENV, STATE)= DECLARE-ENV(DL,
  DECLARE-ENV(D, ENV, STATE), INITIALIZE(
  UNDEFINED, STATE))
- (DECLARE(ID : TYPE, STATE, ENV)= INITIALIZE(TYPE, STATE))
- (DECLARE-ENV(ID : TYPE, ENV, STATE)= BIND(ID, ENV,
  STATE))

**JBO**

***THIS OBJECT PROVIDES AN INTERFACE FOR EXPRESSIONS***

- OBJ EXPRESSIONS / INT-EXP BOOL-EXP

**JBO**

***WE NOW DEFINE VARIOUS STATEMENTS AND THEIR MEANINGS***

- IM (LIST -> STMT-LIST)
- SORTS (LIST -> STMT-LIST)
- (ELEMENT -> STMT)

**VARS**

- TAPE: TAPE
- STMT-L: STMT-LIST

**OK-EQNS**

- (EXECUTE(STMT-L, TAPE)= OUTPUT-OF(TAPE-OF(
  EVAL(STMT-L, NIL-ENV, INITIAL-STATE(TAPE))))

**JBO**

***OBJ DECLARATION / DECL-LIST EXPRESSIONS***

**OK-OPS**

- ID TYPE -> DECLARATION
- DECLARE-ENV: DECL-LIST ENV STATE -> ENV
- DECLARE = DECL-LIST STATE ENV -> STATE

**VARS**

- D: DECLARATION
- DL: DECL-LIST
- ENV: ENV
- ID: ID
- TYPE: TYPE
- ID-L: ID-LIST
- STATE: STATE

**OK-EQNS**

- (DECLARE(D : DL, STATE, ENV)= DECLARE(DECLARE(D,
  STATE, ENV), DECLARE-ENV(D, ENV, STATE))
- (DECLARE-ENV(D : DL, ENV, STATE)= DECLARE-ENV(DL,
  DECLARE-ENV(D, ENV, STATE), INITIALIZE(
  UNDEFINED, STATE))
- (DECLARE(ID : TYPE, STATE, ENV)= INITIALIZE(TYPE,
  STATE))
- (DECLARE-ENV(ID : TYPE, ENV, STATE)= BIND(ID, ENV,
  STATE))

**JBO**

***OBJ EXECUTION / STMT-LIST ENVIRONMENT I/O***

**SORTS**

- PROGRAM

**OK-OPS**

- EXECUTE: PROGRAM -> TAPE
- EVAL: STMT-LIST ENV STATE -> STATE
- INPUT: STMT-L TAPE -> PROGRAM

**VARS**

- TAPE: TAPE
- STMT-L: STMT-LIST

**OK-EQNS**

- (EXECUTE(STMT-L, TAPE)= OUTPUT-OF(TAPE-OF(
  EVAL(STMT-L, NIL-ENV, INITIAL-STATE(TAPE))))

**JBO**
OBJ BLOCK / DECLARATION EXECUTION
SORTS BLOCK
OK-OPS
; : DECL-LIST STMT-LIST -> BLOCK
BEGIN END : BLOCK -> STMT
VARS
DCL-L : DECL-LIST
STMT-L : STMT-LIST
ENV : ENV
STATE : STATE
OK-EQNS
(EVAL(BEGIN DCL-L ; STMT-L END, ENV, STATE)= EVAL(STMT-L, DECLARE-ENV(DCL-L, ENTERBLOCK ENV, STATE), DECLARE(DCL-L, STATE, ENTERBLOCK ENV))
)

*** DEFINE ASSIGNMENT ***
OBJ ASSIGNMENT / EXECUTION EXPRESSIONS
OK-OPS
:= : ID EXP -> STMT
ASSIGN : ID ITEM ENV STATE -> STATE
ERR-OPS
TYPE-OF_CONFLICTS : ID ITEM -> STATE
VARS
ID : ID
EXP : EXP
STATE : STATE
ITEM : ITEM
OK-EQNS
(EVAL(ID := EXP, ENV, STATE)= ASSIGN(ID, VALUE(EXP, ENV, STATE), ENV, STATE))
(EVAL(ASSIGN(ID, ITEM, ENV, STATE)= PUT(GET(ENV, ID), ITEM, STATE))
ERR-EQNS
(EVAL(ASSIGN(ID, ITEM, ENV, STATE)= TYPE-OF ID CONFLICTS IF NOT TYPE-OF RETRIEVE(ID, ENV, STATE)== TYPE-OF ITEM)
)

*** DEFINE CONDITIONAL ***
OBJ CONDITIONAL / EXECUTION EXPRESSIONS STMT-LIST
OK-OPS
IF: THEN ELSE : EXP STMT-LIST STMT-LIST -> STMT
VARS
EXP : EXP
STMT-L : STMT-LIST
ENV : ENV
STATE : STATE
OK-EQNS
(EVAL(IF: EXP THEN STMT-L ELSE STMT-L': FI, ENV, STATE)= EVAL(STMT-L, ENV, STATE) IF BOOL-VAL(VALUE(EXP, ENV, STATE))== T)
(EVAL(STMT-L', ENV, STATE) IF BOOL-VAL(VALUE(EXP, ENV, STATE))== F)

*** DEFINE ITERATION ***
OBJ ITERATION / EXECUTION EXPRESSIONS STMT-LIST
OK-OPS
WHILE DO OD : EXP STMT-LIST -> STMT
VARS
EXP : EXP
STMT-L : STMT-LIST
ENV : ENV
STATE : STATE
OK-EQNS
(EVAL(WHILE EXP DO STMT-L OD, ENV, STATE)= EVAL(STMT-L, WHILE EXP DO STMT-L OD, ENV, STATE) IF BOOL-VAL(VALUE(EXP, ENV, STATE))== T)
(EVAL(STMT-L, ENV, STATE) IF BOOL-VAL(VALUE(EXP, ENV, STATE))== F)

*** THIS OBJECT SUMMARIZES ALL STATEMENT DEFINITIONS ***
OBJ STATEMENTS / EXECUTION SEMICOLON BLOCK
ASSIGNMENT CONDITIONAL INPUT-OUTPUT ITERATION

*** WE NOW BEGIN DEFINING PROCEDURES ***
IM (PAIR => PROC-BODY) / STMT-LIST ID-LIST BOOL
SORTS (PAIR => PROC-BODY)
  (LEFT => ID-LIST)
  (RIGHT => STMT-LIST)
OPS (LEFT : PAIR -> LEFT => FORMALS-OF )
  (RIGHT : PAIR -> RIGHT => STMT-OF )
MI

IM (PAIR => CONTOUR) / PROC-BODY ENVIRONMENT BOOL
SORTS (PAIR => CONTOUR)
  (LEFT => PROC-BODY)
  (RIGHT => ENV)
OPS (LEFT : PAIR -> LEFT => BODY-OF )
  (RIGHT : PAIR -> RIGHT => ENV-OF )
MI

IM (LIST => PARAM-LIST)
SORTS (LIST => PARAM-LIST)
  (ELEMENT => PARAMETER)
MI

IM (LIST => TYPE-LIST) / ITEM
SORTS (LIST => TYPE-LIST)
  (ELEMENT => TYPE)
OPS (NIL : LIST => VOID)
MI

IM (LIST => EXP-LIST) / EXPRESSIONS BOOL
SORTS (LIST => EXP-LIST)
  (ELEMENT => EXP)
MI

OBJ PROCEDURES / CONTOUR TYPE-LIST EXP-LIST
PARAM-LIST DECLARATION
SORTS PROC-DECL
OK-OPS
PROC[ ] : TYPE-LIST => TYPE
  - : CONTOUR => VALUE
  + : PROC-DECL => DECLARATION
CONTOUR-VAL : ITEM => CONTOUR
PROC[ ] END : ID PARAM-LIST STMT-LIST => PROC-DECL
  - : ID TYPE => PARAMETER
CALL[ ] : ID EXP-LIST => STMT
VARS
TYPE-L : TYPE-LIST
C : CONTOUR
VALUE : VALUE
OK-EQNS
(CONTOUR-VAL( < PROC[ TYPE-L ] ; C ) ) = C
ERR-EQNS
(CONTOUR-VAL( < TYPE : VALUE ) ) = TYPE

DOES-NOT-MATCH PROC[ VOID ] IF ( (TYPE == INT ) OR (TYPE == BOOL ) )
JBO

OBJ PROC-DECLARATION / PROCEDURES
OK-OPS
GET-ID : PARAM-LIST => ID-LIST
GET-TYPE : PARAM-LIST => TYPE-LIST
VARS
ID : ID
PM-L : PARAM-LIST
STMT-L : STMT-LIST
ENV : ENV
STATE : STATE
FM : PARAMETER
TYPE : TYPE
OK-EQNS
(DECLARE-ENV(PROC ID [ PM-L ] STMT-L END, ENV, STATE)= BIND(ID, ENV, STATE))

JBO

OBJ PARAM-PASS-BY-VALUE / PROC-DECLARATION
OK-OPS
PASS-ENV ; ID ENV STATE => ENV
PASS : EXP-LIST STATE ENV => STATE
GET-ENV : ID ENV STATE => ENV
GET-PARAMS : ID ENV STATE => ID-LIST
VARS
EXP : EXP
EXP-L : EXP-LIST
ID : ID
STMT-L : STMT-LIST
STATE : STATE
ENV : ENV
OK-EQNS
(PASS(EXP ; EXP-L, STATE, ENV)= PASS(EXP-L, PASS( EXP, STATE, ENV), ENV))
(PASS(EXP, STATE, ENV)= INITIALIZE(VALUE(EXP, ENV, STATE)))
(PASS-ENV(ID, ENV, STATE)= BIND(GET-PARAMS(ID, ENV, STATE)))
(GET-PARAMS(ID, ENV, STATE)= FORMALS-OF(BODY-OF CONTOUR-VAL(VALUE(ID, ENV, STATE))) )
*** A TEST FOR RECURSION ***

RUN EXECUTE((BEGIN('A : INT ; (PROC 'P [ 'B : INT ](IF:(NOT('B EQ 3))THEN(CALL 'P [('B + I)]) ELSE(PRINT('B + II)):FI))END));(READ 'A ; CALL 'P [('A + I)])END))INPUT:{< INT : 2 >})

AS TAPE: ({< INT : 5 >};{< INT : 10 >};{< INT : 33 >})

*** TESTS FOR PROCEDURES AS PARAMETERS ***

RUN EXECUTE((BEGIN('A : INT ; 'Q : PROC[ INT ] ; (PROC 'S ['C : INT ])(PRINT 'C)END));(PROC 'P ['R : PROC[ INT ] ; 'D : INT ](CALL 'R [('D + I)])END));(READ 'A ; 'Q := 'S ; (CALL 'P ['Q ; 'A ]);END))END)INPUT:{< INT : 22 >})

AS TAPE: ({< INT : 14 >})

*** TESTS FOR BLOCKS ***

RUN EXECUTE((BEGIN('S : INT ; 'M : INT ; PROC 'P ['A : INT ]('S :=('S * 'A))END);(PROC 'P ['R : PROC[ INT ] ; 'B : INT ; 'C : INT ](CALL 'B [('B + 'C)])END));('Q := 'P ; READ 'A ; (CALL 'Q ['S ; 'A ; 'A ]);END))END)INPUT:{< INT : 11 >})

AS TAPE: ({< INT : 22 >})

[PHOTO: Recording terminated Tue 30-Dec-80 11:45PM]

(JBO)

OBJ CALL-BY-VALUE / PARAM-PASS-BY-VALUE EXECUTION

OK-OPS

CALL-OK? : ID EXP-L ENV STATE -> BOOL
LIST-TYPE : EXP-L ENV STATE -> TYPE-LIST
GET-STMT : ID ENV STATE -> STMT-LIST

ERR-OPS

PARAMS-OF-MISMATCH : ID EXP-LIST -> BOOL

VARS

ID : ID
EXP-L : EXP-LIST
EXP : EXP
ENV : ENV
STATE : STATE

OK-EQNS

(EVAL(CALL ID [ EXP-L ], ENV, STATE) = EVAL(GET-STMT(ID, ENV, STATE), PASS-ENV(ID, EXP-L, ENV, STATE), PASS(EXP-L, STATE, ENV))IF CALL-OK?(ID, EXP-L, ENV, STATE))

(CALL-OK?(ID, EXP-L, ENV, STATE) = T IF(TYPE(ID, ENV, STATE) = PROC[ LIST-TYPE(EXP-L, ENV, STATE)]))

(GET-STMT(ID, ENV, STATE) = STMT-OF BODY-OF CONTOUR-VAL(VALUE(ID, ENV, STATE)))

(LIST-TYPE(EXP, ENV, STATE) = TYPE(EXP, ENV, STATE))

(LIST-TYPE(EXP-L, ENV, STATE) = TYPE(EXP-L, ENV, STATE))

ERR-EQNS

(CALL-OK?(ID, EXP-L, ENV, STATE) = PARAMS-OF-MISMATCH EXP-L IF(NOT(TYPE(ID, ENV, STATE) = PROC[ LIST-TYPE(EXP-L, ENV, STATE)]))

(JBO)

*** WE NOW SUM UP ALL THE FEATURES OF MODEST ***

OBJ MODEST / STATEMENTS CALL-BY-VALUE

JBO

*** TEST PROGRAMS FOR THE MODEST DEFINITION ***

*** FIRST A TEST FOR WHILE ***

RUN EXECUTE((BEGIN 'A : INT ; READ 'A ; WHILE(NOT('A EQ 4))DO(CALL 'P ['A ; 'M ]); 'M :=('M + I))OD ; PRINT 'S END))INPUT:{< INT : 2 >})

AS TAPE: ({< INT : 2 >};{< INT : 3 >})

*** A TEST FOR BLOCKS ***

RUN EXECUTE((BEGIN 'A : INT ; 'B : INT ; (READ 'A ; 'B :=('A + I)); PRINT 'B); (BEGIN 'A : INT ; READ 'A ; 'B :=('A + 5); PRINT 'B END); 'B :=('A + 22); PRINT 'B END))INPUT:{< INT : 11 >})

(GET-ENV(ID, ENV, STATE) = ENTERBLOCK ENV-OF CONTOUR-VAL(VALUE(ID, ENV, STATE)))

(JBO)

OBJ CALL-BY-VALUE / PARAM-PASS-BY-VALUE EXECUTION

OK-OPS

CALL-OK? : ID EXP-L LIST ENV STATE -> BOOL
LIST-TYPE : EXP-L ENV STATE -> TYPE-LIST
GET-STMT : ID ENV STATE -> STMT-LIST

ERR-OPS

PARAMS-OF-MISMATCH : ID EXP-LIST -> BOOL

VARS

ID : ID
EXP-L : EXP-LIST
EXP : EXP
ENV : ENV
STATE : STATE

OK-EQNS

(EVAL(CALL ID [ EXP-L ], ENV, STATE) = EVAL(GET-STMT(ID, ENV, STATE), PASS-ENV(ID, EXP-L, ENV, STATE), PASS(EXP-L, STATE, ENV))IF CALL-OK?(ID, EXP-L, ENV, STATE))

(CALL-OK?(ID, EXP-L, ENV, STATE) = T IF(TYPE(ID, ENV, STATE) = PROC[ LIST-TYPE(EXP-L, ENV, STATE)]))

(GET-STMT(ID, ENV, STATE) = STMT-OF BODY-OF CONTOUR-VAL(VALUE(ID, ENV, STATE)))

(LIST-TYPE(EXP, ENV, STATE) = TYPE(EXP, ENV, STATE))

(LIST-TYPE(EXP-L, ENV, STATE) = TYPE(EXP-L, ENV, STATE))

ERR-EQNS

(CALL-OK?(ID, EXP-L, ENV, STATE) = PARAMS-OF-MISMATCH EXP-L IF(NOT(TYPE(ID, ENV, STATE) = PROC[ LIST-TYPE(EXP-L, ENV, STATE)])))

(JBO)

*** WE NOW SUM UP ALL THE FEATURES OF MODEST ***

OBJ MODEST / STATEMENTS CALL-BY-VALUE

JBO

*** TEST PROGRAMS FOR THE MODEST DEFINITION ***

*** FIRST A TEST FOR WHILE ***

RUN EXECUTE((BEGIN 'A : INT ; READ 'A ; WHILE(NOT('A EQ 4))DO(CALL 'P ['A ; 'M ]); 'M :=('M + I))OD ; PRINT 'S END))INPUT:{< INT : 2 >})

AS TAPE: ({< INT : 2 >};{< INT : 3 >})

*** A TEST FOR BLOCKS ***

RUN EXECUTE((BEGIN 'A : INT ; 'B : INT ; (READ 'A ; 'B :=('A + I)); PRINT 'B); (BEGIN 'A : INT ; READ 'A ; 'B :=('A + 5); PRINT 'B END); 'B :=('A + 22); PRINT 'B END))INPUT:{< INT : 11 >})

(GET-ENV(ID, ENV, STATE) = ENTERBLOCK ENV-OF CONTOUR-VAL(VALUE(ID, ENV, STATE)))

(JBO)

OBJ CALL-BY-VALUE / PARAM-PASS-BY-VALUE EXECUTION

OK-OPS

CALL-OK? : ID EXP-L LIST ENV STATE -> BOOL
LIST-TYPE : EXP-L ENV STATE -> TYPE-LIST
GET-STMT : ID ENV STATE -> STMT-LIST

ERR-OPS

PARAMS-OF-MISMATCH : ID EXP-LIST -> BOOL

VARS

ID : ID
EXP-L : EXP-LIST
EXP : EXP
ENV : ENV
STATE : STATE

OK-EQNS

(EVAL(CALL ID [ EXP-L ], ENV, STATE) = EVAL(GET-STMT(ID, ENV, STATE), PASS-ENV(ID, EXP-L, ENV, STATE), PASS(EXP-L, STATE, ENV))IF CALL-OK?(ID, EXP-L, ENV, STATE))

(CALL-OK?(ID, EXP-L, ENV, STATE) = T IF(TYPE(ID, ENV, STATE) = PROC[ LIST-TYPE(EXP-L, ENV, STATE)]))

(GET-STMT(ID, ENV, STATE) = STMT-OF BODY-OF CONTOUR-VAL(VALUE(ID, ENV, STATE)))

(LIST-TYPE(EXP, ENV, STATE) = TYPE(EXP, ENV, STATE))

(LIST-TYPE(EXP-L, ENV, STATE) = TYPE(EXP-L, ENV, STATE))

ERR-EQNS

(CALL-OK?(ID, EXP-L, ENV, STATE) = PARAMS-OF-MISMATCH EXP-L IF(NOT(TYPE(ID, ENV, STATE) = PROC[ LIST-TYPE(EXP-L, ENV, STATE)])))

(JBO)

*** WE NOW SUM UP ALL THE FEATURES OF MODEST ***

OBJ MODEST / STATEMENTS CALL-BY-VALUE

JBO

*** TEST PROGRAMS FOR THE MODEST DEFINITION ***

*** FIRST A TEST FOR WHILE ***

RUN EXECUTE((BEGIN 'A : INT ; READ 'A ; WHILE(NOT('A EQ 4))DO(CALL 'P ['A ; 'M ]); 'M :=('M + I))OD ; PRINT 'S END))INPUT:{< INT : 2 >})

AS TAPE: ({< INT : 2 >};{< INT : 3 >})

*** A TEST FOR BLOCKS ***

RUN EXECUTE((BEGIN 'A : INT ; 'B : INT ; (READ 'A ; 'B :=('A + I)); PRINT 'B); (BEGIN 'A : INT ; READ 'A ; 'B :=('A + 5); PRINT 'B END); 'B :=('A + 22); PRINT 'B END))INPUT:{< INT : 11 >})

(GET-ENV(ID, ENV, STATE) = ENTERBLOCK ENV-OF CONTOUR-VAL(VALUE(ID, ENV, STATE)))

(JBO)

OBJ CALL-BY-VALUE / PARAM-PASS-BY-VALUE EXECUTION

OK-OPS

CALL-OK? : ID EXP-L LIST ENV STATE -> BOOL
LIST-TYPE : EXP-L ENV STATE -> TYPE-LIST
GET-STMT : ID ENV STATE -> STMT-LIST

ERR-OPS

PARAMS-OF-MISMATCH : ID EXP-LIST -> BOOL

VARS

ID : ID
EXP-L : EXP-LIST
EXP : EXP
ENV : ENV
STATE : STATE

OK-EQNS

(EVAL(CALL ID [ EXP-L ], ENV, STATE) = EVAL(GET-STMT(ID, ENV, STATE), PASS-ENV(ID, EXP-L, ENV, STATE), PASS(EXP-L, STATE, ENV))IF CALL-OK?(ID, EXP-L, ENV, STATE))

(CALL-OK?(ID, EXP-L, ENV, STATE) = T IF(TYPE(ID, ENV, STATE) = PROC[ LIST-TYPE(EXP-L, ENV, STATE)]))

(GET-STMT(ID, ENV, STATE) = STMT-OF BODY-OF CONTOUR-VAL(VALUE(ID, ENV, STATE)))

(LIST-TYPE(EXP, ENV, STATE) = TYPE(EXP, ENV, STATE))

(LIST-TYPE(EXP-L, ENV, STATE) = TYPE(EXP-L, ENV, STATE))

ERR-EQNS

(CALL-OK?(ID, EXP-L, ENV, STATE) = PARAMS-OF-MISMATCH EXP-L IF(NOT(TYPE(ID, ENV, STATE) = PROC[ LIST-TYPE(EXP-L, ENV, STATE)])))

(JBO)

*** WE NOW SUM UP ALL THE FEATURES OF MODEST ***

OBJ MODEST / STATEMENTS CALL-BY-VALUE

JBO

*** TEST PROGRAMS FOR THE MODEST DEFINITION ***

*** FIRST A TEST FOR WHILE ***

RUN EXECUTE((BEGIN 'A : INT ; READ 'A ; WHILE(NOT('A EQ 4))DO(CALL 'P ['A ; 'M ]); 'M :=('M + I))OD ; PRINT 'S END))INPUT:{< INT : 2 >})

AS TAPE: ({< INT : 2 >};{< INT : 3 >})

*** A TEST FOR BLOCKS ***

RUN EXECUTE((BEGIN 'A : INT ; 'B : INT ; (READ 'A ; 'B :=('A + I)); PRINT 'B); (BEGIN 'A : INT ; READ 'A ; 'B :=('A + 5); PRINT 'B END); 'B :=('A + 22); PRINT 'B END))INPUT:{< INT : 11 >})

(GET-ENV(ID, ENV, STATE) = ENTERBLOCK ENV-OF CONTOUR-VAL(VALUE(ID, ENV, STATE)))

(JBO)
REFERENCES

[Bjorner & Jones 78]

[Burstall & Goguen 77]

[Burstall & Goguen 80]

[Constable & Donahue 79]

[Goguen & Tardo 79]

[Goguen, Thatcher, Wagner & Wright 77]

[Goguen 74]

[Goguen 77]

[Goguen 78]

[Goguen 80]

[Gordon 79]

[Jones 80]

[Landin 66]

[McCarthy 62]

[Mosses 78]
[Mosses 79]

[Musser 80]

[Pagan 79]

[Parsaye 81]

[Raoult & Vuillemin 80]

[Reynolds 70]

[Scott 76]

[Tennent 76]

[Wand 77]