Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Modular Denotational Semantics Description

by

Roberto da Silva Bigonha

RT 004/94

Caixa Postal, 702

30.161 - Belo Horizonte - MG

November 24, 2013

Contents

1 Introduction

2 The Methodoly

In denotational semantics, the meaning of a language is given by associating with each construct in the language a corresponding semantic object, such as an abstract mathematical denotation. The language constructs are members of abstract syntactic domains and the mathematical objects in corresponding semantic domains. The association of language constructs with semantic objects is specified via mappings from syntactic to semantic domains. A denotational definition consists of the specification of syntactic and semantic domains together with the associated mappings.

Usually the first step in the formulation of a denotational semantic definition is the specification of the syntactic universe, i.e., the abstract syntax of the language. Principles of denotational semantics suggest that constructs with semantic similarities should be grouped in a single syntactic category. The underlying idea is to enhance conceptual clarity and to make the semantic definition more compact and elegant. This partitioning of the domain of constructs of a language into conceptually meaningful groups constitutes a key point in enhancing the *readability* of denotational definitions. A good partitioning reduces the number of semantic functions, and thus concentrates the definition of semantic concepts at specific points in the description as opposed to having pieces of them scattered throughout the entire definition. If one chooses to place each type of command of an imperative language into a separate syntactic domain, the meaning of the language's commands is given not by *one* semantic function but by a collection of possibly mutually recursive functions.

In general, the classification of a language's constructs into groups according to their semantic similarities is not a difficult task, as long as some knowledge about the semantics of the language is available in advance. In practice, this would not be much of a problem because at the time the definition of the language is being formulated, all or at least a major portion of the language must have already been designed, so that the definer should have good insight into its semantics. Conversely, for the case of a newly designed language, it has been recommended [**?**, **?**] that the very process of formulating the language definition should be also used to evaluate and reveal problems in its design. In any event, if the initial assumptions about the language's semantics do not turn out to be entirely valid, either a different syntactic classification is in order or the design should be changed.

The second step in the formulation of a denotational definition is the characterization of the necessary semantic domains. In contrast with syntactic domain specifications, it is more difficult to formulate all the necessary semantic domain equations in advance of the specification of the semantic functions. In general, the semantic domain structures depend on the way the associated mappings are defined. For instance, standard denotational semantic definitions are modeled on the notions of stores, environments and continuations. However, the need for these concepts and details of their internal structures vary from language to language. Apparently, the most natural approach is to provide the specification of semantic domains incrementally as they are demanded by semantic functions.

An important principle in denotational semantic definitions is that the meaning of a language construct should depend only on the meaning of its immediate constituents. This principle, called *referentially transparent property* [?], is related to the fact that in denotational semantics, the denotation of a construct is intended to be a complete representation of its semantics, and, the semantics of a construct should be a function of the denotations of its constituents and nothing else. Requiring referential transparency is tantamount to requiring that if two constructs have the same denotation, then they are semantically indistinguishable.

Moreover, the dependency between the denotation of a construct and those of its constituents need only be on the types and names of the associated semantic functions, and not on their complete definition. In terms of formulating language definitions the property of referential transparency provides the basis for applying *information hiding* techniques as suggested by modern object-oriented programming methodology. A module can be used to abstract away syntactic domains and associated semantic functions.

Another characteristic of denotational semantic definitions is that they are (abstract) syntax-directed. Therefore unlike programs in a general purpose programming language, denotational definitions have their *control structure* more or less fixed in advance by the language's abstract syntax. Consequently, the language's abstract syntax plays a very important role in the organization of denotational definitions. A methodology for formulating denotational semantic definitions should give more emphasis on the specifications of abstract syntaxes and semantic domains rather than on the *control structure* of semantic functions. In fact, the problems of formulating denotational semantic descriptions are well treated by the abstract data type and object-oriented methodologies [?, ?, ?, ?]. In particular, the readability problem of denotational semantics lies mainly on the way domains are presented. As in the object-oriented methodology, attention should be focused on *data* rather than *control*. Basically, the underlying strategy is to form modules consisting of the data structure, i.e., domain definitions and related semantic and auxiliary functions.

Fundamentally, methodology proposed has its basis in the use of the language's syntactic hierarchy as a criterion to separate the semantic details of the language into levels. This approach has the advantage that the module structure is automatically established when the language's abstract syntax is defined. Therefore, this immediately solves the most difficult task in the object-oriented methodology, namely, the problem of identifying the *best* collection of classes.

Similar criteria have been widely used to provide BNF-based informal definitions of programming languages [?, ?, ?, ?, ?, ?]. These definitions are, in general, organized into chapters and sections which can be viewed as *modules*, each of which dedicated to specific language concept while abstracting away details of others. The ALGOL 60 report [?], for instance, dedicates a section to each major concept in the language, such as expressions, commands, declarations, etc.. Abstractions are informally used throughout the report. Consider section 4.6 of the Algol 60 report, which is concerned with **for** statement. In this context, all that is required to be known about Algol 60 expressions are their types, while details about how they are constructed have been abstracted away into another section.

The structure of the Algol 60 report is essentially that of the syntactic specification of the language. In addition, there is a “module” for each semantically meaningful syntactic category in the grammar.

In summary, the proposed methodology for formulating structured denotational semantic definitions with the support of $\mathcal{SCRIPT}$ consists of the following steps:

1. From the concrete syntax of the language and from the language's semantics the definer has in mind, the corresponding abstract syntax shoud be defined.
2. One or more syntactic domains in the abstract syntax are associated with module which defines:
    - The internal structure of the domains.
    - Associated semantic functions.
3. Important semantic domains such as *Stores* and *Environments* should be identified and treated as “abstract data types” or object's classes, and thus, also entitled to their own modules which encapsulate their internal structure and define associated operators.
4. As modules are explicitly defined, the need for new types or semantic domains may arise. Like in the methodology, new modules should be created accordingly.

It should be emphasized that inherent properties of denotational semantics do not permit information hiding principles to be used to their full extent. The internal structure of syntactic domains may not be completely hidden inside modules because of the syntax-directed nature of the semantic definition style. Also, the internal structure of domains of continuations cannot be encapsulated because their internal details are needed at the various point where the continuation functions are defined. The important implication of this approach is that modules must be flexible in terms of visibility to information than a module that implements an abstract data type.

3 The Programming Language ASPLE

In order to show an application of the proposed methodology, a formal semantic definition of a simple language ASPLE [?] will be presented.

A denotational definition of ASPLE has already been published by D. Gouge [?]. This present definition borrows many of the solutions in [?] and highlights the proposed module organization.

The formal definition of ASPLE is presented in two parts: The first is concerned with type-checking ASPLE programs while the second describes the *dynamic* semantics of type-checked ASPLE programs.

Guided by extant semantic definitions of ASPLE, the domains of ASPLE constructs can be defined as tt Program, Dcl, Stmt and `Exp` for programs, declarations, commands and expressions, respectively. With every member of each of these domains will be associated two semantic functions: one to describe the type-checking of ASPLE constructs and the other to define their semantic interpretations.

Other syntactic domains easily identifiable in the concrete syntax of ASPLE are the domain `Id` of identifiers, the domain `Mode` of ASPLE type specifications and the domains `Num` and `Bool` for ASPLE constants.

3.1 Concrete and Abstract Syntaxes

The following module provides the concrete syntax of ASPLE and indicates how the corresponding abstract syntax is to be generated.

```
SYNTAX ASPLE
   program         ::= "begin" dcl-train stmt-train "end";

   dcl-train       ::= dcl+;
   dcl             ::= mode id+-"," ";" : [mode ide+];
   mode            ::= "bool" : "bool"
                     | "int"  : "int"
                     | "ref" mode ;

   stmt-train      ::= stmt+-";" ;
   stmt            ::= asgt-stmt : asgt-stmt
                     | cond-stmt : cond-stmt
                     | loop-stmt : loop-stmt
                     | transput-stmt : transput-stmt;
   asgt-stmt       ::= id ":=" exp ;
   cond-stmt       ::= "if" exp "then" stmt-train "fi"
                     | "if" exp "then" stmt-train
                       "else" stmt-train "fi"
   loop-stmt       ::= "while" exp "do" stmt-train "end" ;
   transput-stmt   ::= "input" id | "output" exp ;

   exp             ::= factor      : factor
                     | exp "+" factor ;
   factor          ::= primary     : primary
                     | factor "*" primary ;
   primary         ::= id
                     | constant    : constant
                     | "(" exp ")" : exp
                     | "(" compare ")" : compare ;
   compare         ::= exp "=" exp
                     | exp "#" exp ;
```

```
   constant        ::= bool | num ;

   bool            ::= "true"      : TT
                     | "false"     : FF;

DOMAINS

     dcl-train,dcl : Dcl;
     stmt-train,stmt : Stmt;
     exp, factor, primary, compare : Exp;

LEXIS

   UNIT            ::= layoutchart+   : ()
                     | id             : (OUT "ID", id)
                     | num            : (OUT "NUM", num);

   layoutchar      ::= " " | "\n" | "\f" ;
   id              ::= letter+  : QUOTE letter+
   letter          === "A" ... "Z";
   num             ::= digit+   : NUMBER digit+ ;
   digit           === "0" ... "9" ;

END ASPLE
```

3.2 The Static Type Checker

3.2.1 Introduction

The primary function of the ASPLE's type checker is to verify whether:

- all used identifiers are properly declared;
- all identifiers are declared only once within the same scope;
- the way identifiers are used agrees with the type assigned to them.

The type checker extends a given ASPLE program text by incorporating in it type information, to simplify the mappings of ASPLE constructs to their denotations. For example, *run-time* type-checking isv required during I/O operations. If the types of the variables involved are available locally, no type environment would be required in the *run-time* semantic definition. Moreover, type information is needed to carry out implicit coercions [?, ?]. Since all the necessary coercions can be identified at compile time, the type checker should make implicit coercions explicit.

Additionally, the type checker should use the type information it collects to perform operator identification. For example, in ASPLE, the same symbols, namely `"+"` and `"*"`, are used to denote integer and boolean operations. In particular, the type checker should replace `"+"` and `"*"` by `or` and `and`, respectively, whenever the corresponding operands have ASPLE type `bool`.

Now that we have decided to define the type checker as mappings from ASPLE constructs to extended ASPLE constructs three problems remain to be solved. The first problem regards the abstract syntax of the extended constructs. Notice that the fact that the type checker *extends* ASPLE programs does not affect the concrete syntax because *extended* programs are never parsed. It is nevertheless too early to make precise the needed extension because they are construct dependent. Their specifications will be delayed until the type checking functions for the involved constructs are defined.

The second problem has to do with error handling and the semantic style to be adopted. One possibility is to adopt continuation semantics, so that the type checker produces either an extended ASPLE program in abstract syntax form or an error message. The advantage of this approach is its convenient way of dealing with error conditions. Its main disadvantage is that the type checker terminates its processing upon encountering the first error. From the viewpoint of language definitions this would not be entirely bad because formal definitions are expected to provide the meaning of semantically correct programs while rejecting wrong ones. However, it seems wise to use formal definitions to establish a basis for standardizing the compiler's error messages. The addopted approach is the use direct semantics, and replacing ill-typed constructs by error indicators. Specifically, type checking functions are defined as mappings from parse trees, which are $\mathcal{SCRIPT}$ representations of abstract programs, to extended parse trees which may contain the undefined value (`?`) as subtrees. The value `"?"` is construed as an error indicator.

3.2.2 Machine Context

Since the formal definition of ASPLE is intended to be machine independent, certain parameters should be *passed* to the definition. These parameters are called *machine context*. In general, they should contain information that enables the formal definition to take into account certain machine dependent characteristics of the language such as the maximum value of integers. The module `Machine-context` encapsulates all these parameters as follows:

```
MODULE Machine-context
EXPORT
   maxint;
DOMAINS
   maxint : N;
DEFINITIONS
   DEF maxint = 32767
END Machine-context
```

3.2.3 Type Environment

The necessary type environment will be defined as a mapping from ASPLE identifiers to their respective type denotations. Module `Type-environ` defines the structure of the type environment and associated operators.

The type environment is defined as `Env = Id -> Den`, where `Id` is the domain of ASPLE identifiers and `Den` that of denotations. For the specific purpose of type checking ASPLE programs, the domain `Den` should include all ASPLE modes. In addition, `Den` must have elements that indicate error conditions. Module `Type-environ` is as follows:

```
MODULE Type-environ
EXPORTS
   *Env, *Den, initial-env;
IMPORTS
   Type-dcl(Mode);
DOMAINS
   Env = Id -> Den;
   Den = Mode | "err" ;
DEFINITIONS
   DEF initial-env : Env = LAM id . ?
END Type-environ
```

Function `initial-env` defines the initial environment in which all ASPLE identifiers are bound to the constant undefined (`?`).

3.2.4 Program Typing

Function `check-prog` below maps ASPLE programs to type-checked *extended* ASPLE programs.

```
MODULE Type-program
EXPORTS
    Program, check-prog;
IMPORTS
   Type-dcl(Dcl,check-dcl);
   Type-stmt(Stmt,check-stmt);
   Type-environ(*Env,initial-env);
DOMAINS
   Program  = ["begin" Dcl Stmt "end"];
   e : Env;
DEFINITIONS
  DEF check-prog(program) : Program =
      LET ["begin" dcl stmt "end"] = program
      LET (dcl',e) = check-dcl(dcl,initial-env)
```

```
    LET stmt' = check-stmt(stmt,e)
    IN  ["begin" dcl' stmt' "end"]
END Type-program
```

3.2.5 Declaration Typing

Function `check-dcl` below transforms ASPLE declarations into type-checked ASPLE declarations. In addition, it enriches the given type environment.

```
MODULE Type-dcl
EXPORTS
  Dcl, Mode, check-dcl;
IMPORTS
  Type-environ(Env);
  Type-id(Id,check-id-list);
DOMAINS
  Dcl  = [Dcl+] | [Mode Id+];
  Mode = "bool" | "int" | ["ref" Mode] ;
  e : Env;

DEFINITIONS
  DEF check-dcl(dcl,e) : (Dcl,Env) =
      CASE dcl
        /[dcl+] -> LET (dcl'+,e')=check-dcl-list(dcl+,e)
                   IN  ([dcl'+],e')
        /[mode id+] ->
                LET (id1+,e1) = check-id-list(id+,e,mode)
                IN  ([mode id1+],e1)
      END
  DEF check-dcl-list(dcl*,e) : (Dcl*,Env) =
       CASE dcl*
        /dcl1 PRE dcl1* ->
                LET (dcl1',e1) = check-dcl(dcl1,e)
                LET (dcl1'*,e2) = check-dcl-list(dcl1*,e1)
                IN  <dcl1' PRE dcl1'*, e2)
        /<> -> (<>,e)
       END
END Type-dcl
```

3.2.6 Identifier Typing

Function `check-id-list` below binds a list of identifiers to their declared modes in the environment. Identifiers which are declared more than once in the same scope are bound to "err".

```
MODULE Type-id
EXPORTS
  Id, check-id-list;
IMPORTS
  Type-dcl(Mode);
  Type-environ(*Env,*Den);
DOMAINS
  Id = Q;
  e : Env;
DEFINITIONS
  DEF check-id-list(id*,e,mode) : (Id*,Env) =
      CASE id*
        /id1 PRE id2* ->
            LET (id1',e1) = check-id(id1,e,mode)
            LET (id2'*,e2) = check-id-list(id2*,e1,mode)
            IN  (id1' PRE id2'*,e2)
        /<> -> (<>,e)
  END
  DEF check-id(id,e,mode) : (Id,Env) =
     e(id) NE ? ->(?,e{"err"/id}),(id,e{["ref" mode]/id})
END Type-id
```

3.2.7 Expression Typing

Function `check-exp` below maps ASPLE expressions in the presence of an environment to corresponding type checked expressions and their *computed* modes. During the process, all implicit coercions are made explicit by appropriate insertion of the operator `deref`. Moreover, `"+"` and `"*"` which denote boolean operators are conveniently replaced by `or` and `and`, respectively.

```
MODULE Type-exp
EXPORTS
  Exp, check-exp;

IMPORTS
  Type-environ(Env,Den);
  Coercion(base-level,deref);
  Machine-context(maxint);
  Type-id(Id)

DOMAINS
  Exp = [Exp "+" Exp]
      | [Exp "*" Exp]
```

```
      | [Exp "=" Exp]
      | [Exp "#" Exp]
      | [Exp "or" Exp] ! extension
      | [Exp "and" Exp]! extension
      | ["deref" Exp]  ! extension
      | [Id] | [Num] | [Bool] ;

DOMAINS
  e: Env;
  d: Den;

DEFINITIONS
  DEF check-exp(exp,e) : (Exp,Den) =
      CASE exp
        /[id] ->
           LET d = e(id)
           IN  (d EQ "err") OR (d EQ ?) ->
                        (?,"err") -- ill or undeclared
                        ([id],d)
        /[num] -> (num LE maxint -> ([num],"int"),
                   (?,"err"))
        /[bool] -> <[bool],"bool">
        /[exp1 "+" exp2] ->
                LET  (exp1,d1) = check-exp(exp1,e)
                ALSO (exp2,d2) = check-exp(exp2,e)
                LET  (d1',n1) = base-level(d1)
                ALSO (d2',n2) = base-level(d2)
                IN (d1' EQ d2') AND (d1' NE "err") ->
                        (LET exp1' = deref(exp1,n1)
                        ALSO exp2' = deref(exp2,n2)
                        IN (d1' EQ "int") ->
                                ([exp1' "+" exp2'],d1'),
                                ([exp1' "or" exp2'],d1'))
                        (?,"err")
        /[exp1 "*" exp2] ->
                LET (exp1,d1) = check-exp(exp1,e)
                LET (exp2,d2) = check-exp(exp2,e)
                LET (d1',n1)  = base-level(d1)
                ALSO (d2',n2) = base-level(exp2,n2)
                IN (d1' EQ d2') AND (d1' NE "err") ->
                   (LET exp1' = deref(exp1,n1)
                    ALSO exp2' = deref(exp2,n2)
                    IN (d1' EQ "int") ->
```

```
                    ([exp1' "*" exp2'],d1'),
                    ([exp1' "and" exp2'],d1'),
                 (?,"err")
        /[exp1 "=" exp2] ->
                LET  (exp1,d1) = check-exp(exp1,e)
                ALSO (exp2,d2) = check-exp(exp2,e)
                LET  (d1',n1) = base-level(d1)
                ALSO (d2',n2) = base-level(d2)
                IN (d1' EQ "int") AND (d2' EQ "int") ->
                        (LET  exp1' = deref(exp1,n1)
                         ALSO exp2' = deref(exp2,n2)
                         IN ([exp1' "=" exp2'],"bool")),
                        (?,"err")
        /[exp1 "#" exp2] ->
                LET  (exp1,d1) = check-exp(exp1,e)
                ALSO (exp2,d2) = check-exp(exp2,e)
                LET  (d1',n1)  = base-level(d1)
                ALSO (d2',n2)  = base-level(d2)
                IN  (d1' EQ "int") AND (d2' EQ "int") ->
                    (LET  exp1' = deref(exp1,n1)
                     ALSO exp2' = deref(exp2,n2)
                     IN ([exp1' "#" exp2'], "bool")),
                    (?,"err")
    END
END Type-exp
```

3.2.8 Type Coercion

Function `base-level` counts the number of *dereferencings* [?, ?] to which a given mode must be submitted to produce its basic mode. Specifically, `base-level` counts and eliminates all `ref`s that precede an `int` or `bool` in a given mode. It returns the basic mode and the number of dereferencings needed.

Function `deref` dereferences ASPLE expressions n times by appending a sequence of `deref`s in front of them. The value of n is given by function `base-level`.

```
MODULE Coercion

EXPORTS
  base-level, deref;

IMPORTS
  Type-environ(Den);
  Type-exp(Exp);
```

```
DEFINITIONS
  DEF base-level(d : Den) : (Den,N) =
      CASE d
          /["ref" mode] -> LET (d',n') = base-level(mode)
                           IN  (d',n' PLUS 1)
          /QUOTE ? -> (d,0)
          /?       -> ("err",0)
      END
  DEF deref(exp,n) : Exp =
      n EQ 0 -> exp, deref(["deref" exp], n MINUS 1)
END Coercion
```

3.2.9 Statement Typing

Function `check-stmt` transforms ASPLE statements in the presence of an environ to type checked ASPLE statements.

```
MODULE Type-stmt

EXPORTS
  Stmt, check-stmt;

IMPORTS
  Type-environ(Env,Den);
  Type-exp(Exp,check-exp);
  Type-dcl(Mode);
  Coercion(base-level,deref);

DOMAINS
  Stmt  = [Stmt+]
        | [Id ":=" Exp]
        | ["if" Exp "then" Stmt "else" Stmt "fi"]
        | ["if" Exp "then" Stmt "fi"]
        | ["while" Exp "do" Stmt "end"]
        | ["input" Id]
        | ["input" Exp Mode]    ! extension
        | ["output" Exp]
        | ["output" Exp Mode]; !extension

DOMAINS
  e : Env;
  d : Den;
```

```
DEFINITIONS
  DEF check-stmt(stmt,e) : Stmt =
      CASE stmt
        /[stmt+] -> LET stmt'+ = check-stmt-list(stmt+,e)
                    IN  [stmt'+]
        /["if" exp "then" stmt1 "fi"] ->
                LET (exp',d) = check-exp(exp,e)
                LET (d',n) = base-level(d)
                LET exp' = (d' EQ "bool" -> deref(exp',n),?)
                ALSO stmt' = check-stmt(stmt1,e)
                IN  ["if" exp' "then" stmt' "fi"]
        /["if" exp "then" stmt1 "else" stmt2 "fi"] ->
                LET (exp',d) = check-exp(exp,e)
                LET (d',n)   = base-level(d)
                LET exp' = (d' EQ "bool" -> deref(exp',n),?)
                ALSO stmt1' = check-stmt(stmt1,e)
                ALSO stmt2' = check-stmt(stmt2,e) IN
                ["if" exp' "then" stmt1' "else" stmt2' "fi"]
        /["while" exp "do" stmt "end"] ->
                LET (exp',d) = check-exp(exp,e)
                LET (d',n) = base-level(d)
                LET exp' = (d' EQ "bool" -> deref(exp',n),?)
                ALSO stmt' = check-stmt(stmt,e)
                IN ["while" exp' "do" stmt' "end"]
        /["input" id] ->
                LET ([id],d) = check-exp([id],e)
                LET (mode,exp) =
                        (d EQ "err") -> (?,?),
                        LET (d',n) = base-level(d)
                        IN  (d',deref([id], n PLUS 1))
                IN ["input" exp mode]
        /["output" exp] ->
                LET (exp',d) = check-exp(exp,e)
                LET (mode,exp'') =
                        d EQ "err" -> <?,?>,
                        LET (d',n) = base-level(d)
                        IN (d',deref(exp',n))
                IN ["output" exp'' mode]
        /[id ":=" exp] ->
                LET  ([id1],d1) = check-exp([id],e)
                ALSO (exp1,d2) = check-exp(exp,e)
                LET  (d1',n1)  = base-level(d1)
```

```
                ALSO (d2',n2)  = base-level(d2)
                IN (d1' EQ d2') AND (d1' NE "err") ->
                   ( (n1 LE (n2 PLUS 1)) ->
                        LET n' = (n2 PLUS 1) MINUS n1
                        LET exp1' = deref(exp1,n')
                        IN  [id1 ":=" exp1'], ?), ?
      END
  DEF check-stmt-list(stmt*,e) : Stmt* =
      CASE stmt*
        /<> -> <>
        /stmt1 PRE stmt2* ->
                (check-stmt(stmt1,e) PRE
                 check-stmt-list(stmt2*,e))
      END
END Type-stmt
```

3.3 The Semantics of ASPLE

Continuation semantics is used to define the semantics of ASPLE in order to facilitate the handling of the various run time errors, such as access to uninitialized variables, operation overflow, attempt to read past the end of file mark; invalid input data.

3.3.1 Continuations

With the three major constructs in ASPLE, namely declarations, commands and expressions, are associated the following domains of continuations: `Elab-cont`, `Exec-cont` and `Eval-cont`, which are defined in the following module:

```
MODULE Continuations
EXPORTS
  *Exec-cont, *Elab-cont, *Eval-cont, init-exec-cont, no-action;

IMPORTS
  Sem-environ(Env);
  Abstract-machine(State, Input, Value, Answer);

  DOMAINS Exec-cont = State -> Input* -> Answer;
         Elab-cont = Env   -> Exec-cont;
         Eval-cont = Value -> Exec-cont;

DEFINITIONS
   LET init-exec-cont (state)(input*) : Exec-cont = <>
   LET no-action(exec-cont) : Exec-cont= exec-cont
END Continuations
```

Notice that the internal structures of `State, Value, Answer, Input` and `Env` are imported from the cited modules, while the domains of continuation are exported opened.

3.3.2 The Run-time Environment

The run-time environment keeps track of the association of ASPLE identifiers to the *locations* assigned to them in the *memory* of the abstract machine. Operators for "updating" and accessing the environment are provided by $\mathcal{SCRIPT}$ directly. Acess to internal details of domain `Env` is granted to client modules.

```
MODULE Sem-environment
EXPORTS
  Env, initial-env;

IMPORTS
  Sem-id(Id);
  Abstract-machine(Loc);

DOMAINS
  Env = Id -> Loc;
  initial-env : Env;

DEFINITIONS
  DEF initial-env = LAM id. ?
END Sem environ
```

3.3.3 The Abstract Machine

The abstract machine that mimics the state to state transformations which model execution of ASPLE commands is defined by module `Abstract-machine`. This module encapsulates the concepts of machine state, memory locations, storable values, input/output and final answers. Moreover, this module provides all the required operators to handle the machine state and to perform input/output operations.

Notice that the particular structure chosen to define the domains of states, locations, final answers, and so forth does not affect the rest of the definition. Notably, no information about the internal structure of these domains is used outside module Abstract-machine. Operations are imported by other modules as needed. The following modules describes a possible abstract machine:

```
MODULE Abstract-machine
EXPORTS
  Answer, Loc, State, M, Value, Mode, Input,
  init-state, newloc, update, content, write, read, wrong;
```

```
IMPORTS
  Continuations(*Eval-cont,*Exec-cont);

DOMAINS
  Answer = Q*;            !Answers
  Loc    = N ;            ! Locations
  State  = (M,Loc);       ! States
  M      = Loc -> Value   ! Memories
  Value  = N | T | Loc;   ! Storable Values
  Mode   = "int" | "bool";! of printable values
  Input  = ?????

DEFINITIONS
  DEF wrong(q)(state)(input*) : Answer = (QUOTE <"ERROR :",q>)
  DEF init-state : State = (LAM loc. ?, 0)
  DEF newloc(eval-cont)(state) :(Input* -> Answer) =
        LET (m,last) = state
        LET last'    = last PLUS 1
        LET state'   = (m,last')
        IN  eval-cont(last')(state')
  DEF update(loc,value)(exec-cont)(state):(Input*->Answer)=
        LET (m,last) = state
        LET m'       = m{value/loc}
        LET state'   = (m',last)
        IN  exec-cont(state')
  DEF content(loc)(eval-cont)(state):(Input*->Answer)=
        LET (m,last) = state
        LET value    = m(loc)
        IN  value NE ? -> eval-cont(value)(state),
                          wrong "undefined" (state)
  DEF write(value,mode)(exec-cont)(state)(input*) : Answer=
        LET q = CASE mode
                /"int" -> LET NUMBER q'* = value
                          IN  QUOTE q'*
                /"bool"-> (value -> "true","false")
                END
        IN  <q> CAT (exec-cont(state)(input*))

  DEF  read(loc,mode)(exec-cont)(state)(input*) : Answer=
    CASE input*
     /input1 PRE input2* ->
        CASE (input1,mode)
```

```
          /(TRUTH ?,"bool")/(NUMBER ?,"int")->
               update(loc,input1)(exec-cont)(state)(input2*)
          /? -> wrong("input type")(state)(<>)
         END
     /? -> wrong ("end of file")(state)(<>)
    END
END Abstract-machine
```

In the module above, the state is composed of a pair `(m,last)`, where `m`, the memory, is a function from locations to storable values, and `last` is the last location in use.

Function `newloc` passes the next free location and the updated state to the given continuation.

Function `update` binds a given storable value to a given location in `m` and passes the updated state to the given continuation.

Function `content` passes the value bound to a given location in `m` to the continuation. However, if the given location happens to be uninitialized, an error error message is produced and the normal continuation is ignored.

Function `read` reads in the next value in the input file into a given location. This value is type-checked appropriately. If it passes the consistency check, the abstract machine state is updated and the updated state and the remainder of the input file are passed to the continuation. Otherwise, an error message is produced and the normal continuation is ignored.

Function `write` passes a given value to the continuation.

The value output is made part of the final answer produced by an ASPLE program.

Finally, `wrong` maps error messages to final answers.

3.3.4 The Concrete Machine

Module `Concrete-machine` is the run-time counterpart of module `Machine-context` defined in the ASPLE type-checker.

```
MODULE Concrete-machine
EXPORTS
  maxint;
DEFINITIONS
  DEF maxint:N = 32767
END Concrete-machine
```

3.3.5 The Semantics of Programs

The domain `Program` of ASPLE programs is associated with the main module defined below and the other domains are associated with other $\mathcal{SCRIPT}$ modules.

We have chosen to repeat the definition of the ASPLE syntactic domains in the following modules instead of importing them from the corresponding modules of the type checker.

We believe that at the expense of having to write more, this approach makes the run-time semantics more independent of the corresponding compile-time semantics, thus enhancing readability. Recall that the consistency among the various definitions of the same domain is automatically verified by the $\mathcal{SCRIPT}$ type-checker, which employs a particular kind of structural equivalence scheme of types.

In the main module `Sem-program`, a given program is type-checked first via the function `check-prog` and then its semantics is evaluated accordingly.

Function run maps type-checked ASPLE programs along with ! an "input file" to final answers. !

```
MODULE Sem-program
EXPORTS
  Program, run ;

IMPORTS
  Sem-dcl(Dcl,elaborate);
  Sem-stmt(Stmt,execute);
  Continuations(*Exec-cont,init-exec-cont);
  Sem-environ(Env,initial-env);
  Abstract-machine(init-state,Answer,Input);
  Type-program(check-prog);

DOMAINS
  Program = ["begin" Dcl Stmt "end"];

DEFINITIONS
  DEF run(program)(input*) : Answer =
      LET ["begin" dcl stmt "end"] = check-prog(program) IN
      elaborate(dcl)(initial-env); LAM env.
      execute(stmt)(env);
      init-exec-cont(init-state)(input*)

END Sem-program
```

3.3.6 The Semantics of Declarations

Function `elaborate` processes type-checked ASPLE declarations in order to allocate memory space for the declared variables. It also binds declared variables to their assigned locations in the environment. Details about *allocation of memory space* are given in module `Sem-id`.

```
MODULE Sem-dcl
EXPORTS
  Dcl, Mode, elaborate;
```

```
IMPORTS
  Sem-environ(Env)
  Continuations(Elab-cont,Exec-cont);
  Sem-id(Id,elaborate-id-list);

DOMAINS
  Dcl  = [Dcl+]
       | [Mode Id+];
  Mode = "bool" | "int" | ["ref" Mode] ;

DEFINITIONS
  DEF elaborate(dcl)(env)(elab-cont) : Exec-cont =
      CASE dcl
        /[dcl+] -> elaborate-list(dcl+)env;elab-cont
        /[mode id+] ->
             elaborate-id-list(id+,mode)env;elab-cont
      END
  DEF elaborate-list(dcl*)env;elab-cont : Exec-cont =
       CASE dcl*
        /dcl1 PRE dcl1* ->
              elaborate(dcl1)env;LAM env'.
              elaborate-list(dcl1*)env';
              elab-cont
        /<> -> elab-cont(env)
       END
END Sem-dcl
```

3.3.7 The Semantics of Identifiers

Function `elaborate-id` allocates a memory cell for a given ASPLE variable and calls function allocate to take care of the additional space needed for *references* to to the variable as indicated in the variable's mode.

Function `elaborate-id-list` is used in module `Sem-dcl` to allocate memory space for ASPLE variables and to bind these variables to their locations in the environment.

```
MODULE Sem-id
EXPORTS
  Id, elaborate-id-list;

IMPORTS
  Sem-dcl(Mode);
  Sem-environ(*Env,*Den);
```

```
  Continuations(Elab-cont,Exec-cont);
  Abstract-machine(new-loc,loc,update);

DOMAINS
  Id = Q;

DEFINITIONS
  DEF elaborate-id-list(id*,mode)(env)(elab-cont) =
    CASE id*
        /id1 PRE id2* ->
             elaborate-id(id1,mode)(env);LAM env'.
             elaborate-id-list(id2*,mode)env';elab-cont
        /<> ->  elab-cont(env)
    END
  DEF elaborate-id(id,mode)(env)(elab-cont) : Exec-cont =
       newloc;LAM loc.
       LET env' = env{loc/id}
       LET exec-cont' = elab-cont(env')
       IN  allocate(loc,mode); exec-cont'

  DEF allocate(loc,mode)(exec-cont) : Exec-cont =
     CASE mode
        /["ref" mode'] -> newloc;LAM loc'.
                          update(loc,loc');
                          allocate(loc',mode'); exec-cont
        /"int"/"bool" -> exec-cont
     END
END Sem-id
```

3.3.8 The Semantics of Expressions

Function evaluate evaluates ASPLE expressions in the presence of an environment and a *memory* and passes value produced to the continuation. In case an overflow condition is detected, the appropriate error message is produced.

```
MODULE Sem-exp
EXPORTS
  Exp, evaluate;

IMPORTS
  Sem-environ(Env,Den);
  Concrete-machine(maxint);
  Abstract-machine(Value,content,loc,wrong);
```

```
  Continuations(Eval-cont,Exec-cont);
  Sem-id(Id);

DOMAINS
  Exp = [Exp "+" Exp]
      | [Exp "*" Exp]
      | [Exp "=" Exp]
      | [Exp "#" Exp]
      | [Exp "or" Exp] ! extension
      | [Exp "and" Exp]! extension
      | ["deref" Exp]  ! extension
      | [Id] | [Num] | [Bool] ;

DEFINITIONS
  DEF evaluate(exp)(env)(eval-cont) : Exec-cont =
    CASE exp
        /[id] ->
           LET loc = env(id)
           IN  eval-cont(loc)
        /[num] -> eval-cont(num)
        /[bool] -> eval-cont(bool)
        /[exp1 "+" exp2] ->
           evaluate(exp1)(env);LAM value1.
           evaluate(exp2)(env);LAM value2.
           LET n = value1 PLUS value2
           IN n LE maxint -> eval-cont(n),wrong "overflow"
        /[exp1 "*" exp2] ->
           evaluate(exp1)(env);LAM value1.
           evaluate(exp2)(env);LAM value2.
           LET n = value1 MULT value2
           IN n LE maxint -> eval-cont(n),wrong "overflow"
        /[exp1 "or" exp2] ->
           evaluate(exp1)(env);LAM value1.
           evaluate(exp2)(env);LAM value2.
           LET t = value1 OR value2
           IN  eval-cont(t)
        /[exp1 "and" exp2] ->
           evaluate(exp1)(env);LAM value1.
           evaluate(exp2)(env);LAM value2.
           LET t = value1 AND value2
           IN  eval-cont(t)
        /[exp1 "=" exp2] ->
           evaluate(exp1)(env);LAM value1.
```

```
          evaluate(exp2)(env);LAM value2.
          value1 EQ value2 -> eval-cont(TT),eval-cont(FF)
       /[exp1 "#" exp2] ->
          evaluate(exp1)(env);LAM value1.
          evaluate(exp2)(env);LAM value2.
          value1 NE value2 -> eval-cont(TT),eval-cont(FF)
       /["deref" exp1] ->
          evaluate(exp1)(env);LAM loc.
          content(loc); eval-cont
    END
END Sem-exp
```

3.3.9 The Semantics of Statements

Basic operations such as input/output and storing a value in the memory of the abstract machine are carried out by calling the operators defined in module `Abstract-machine`

```
MODULE Sem-stmt
EXPORTS
  Stmt, execute;

IMPORTS
  Sem-environ(Env);
  Sem-exp(Exp,evaluate);
  Sem-dcl(Mode);
  Continuations(Exec-cont,no-action);
  Abstract-machine(read,write,loc,update,Value,wrong);

DOMAINS
  Stmt   = [Stmt+]
         | [Id ":=" Exp]
         | ["if" Exp "then" Stmt "else" Stmt "fi"]
         | ["if" Exp "then" Stmt "fi"]
         | ["while" Exp "do" Stmt "end"]
         | ["input" Exp Mode]   ! extension
         | ["output" Exp Mode]; !extension

DEFINITIONS
  DEF execute(stmt)(env)(exec-cont) : Exec-cont =
    CASE stmt
       /[stmt+] -> execute-list(stmt+)(env);exec-cont
       /["if" exp "then" stmt1 "fi"] ->
          evaluate(exp)(env);LAM t.
```

```
           (t->execute(stmt1)(env),no-action);
           exec-cont
        /["if" exp "then" stmt1 "else" stmt2 "fi"] ->
           evaluate(exp)(env); LAM t.
           (t -> execute(stmt1)(env), execute(stmt2)(env));
           exec-cont
        /["while" exp "do" stmt "end"] ->
           DEF exec-cont'=
                evaluate(exp)(env);LAM t.
                (t -> execute(stmt)(env);exec-cont',
                      exec-cont)
           IN  exec-cont'
        /["input" exp mode] ->
           evaluate(exp)(env);LAM loc.
           read(loc,mode); exec-cont
        /["output" exp mode] ->
           evaluate(exp)(env);LAM value.
           write(value,mode); exec-cont
        /[id ":=" exp] ->
           evaluate([id])(env);LAM loc.
           evaluate(exp)(env);LAM value.
           update(loc,value); exec-cont
    END
DEF execute-list(stmt*)(env)(exec-cont) : Exec-cont =
     CASE stmt*
        /<> -> exec-cont
        /stmt1 PRE stmt2* ->
          execute(stmt1)(env);
          execute-list(stmt2*)(env);
          exec-cont
     END
END Sem-stmt
```

4 The PROJECT Module

The entire semantic definition is introduced via the following module of PROJECT type:

```
PROJECT ASPLE

IMPORTS
   Program(run, Program);
   Abstract-machine(Input, Answer);
```

```
DOMAINS
   run := Program -> Input-data -> Answer;
   Input-data = Input* ;

INFILES
   Program = "prog.asp"
   Input-data = "prog.inp"

OUTFILE
   Answer = "prog.out"

COMPONENTS
    "Minil.scr", "Program.scr", "Env.scr", "Command.scr", "Expression.scr"
END ASPLE
```

5 Conclusion

The semantic definition of ASPLE has been decomposed into small pieces which are more or less independent of each other. Basic semantic concepts such as stores, environments and continuations have been isolated into separate modules so that the choice of a particular model for them does not affect the rest of the definition.

We have used the syntactic structure of the language being defined to guide the partitioning of the definition into small modules. Each module encapsulates the details of the definition of some domains and related functions, and makes their names and types available for use in other modules. This partitioning of the definition into *syntactic* modules corresponds to common practice.

We claim that such a partitioning of a denotational definition produces satisfactory results in the sense that the interfaces among the various modules are kept reasonably small.

Except for the emphasis we have placed on the use of the language's abstract syntax as one of the criteria to organizing denotational definitions, our approach is very similar to the **CLEAR-OBJ** approach of Goguen, Burstall and Parsaye [GOGUEN 77a, GOGUEN 80] and to Mosses' theories approach [MOSSES 79]. In essence, all of these approaches share the same underlying ideas proposed by the data abstraction methodology.

In fact, the basic differences are not in the methodology proposed but in the style of presentation of semantic descriptions. In the first place, we work with explicit definitions of data types and their related operations, while Goguen, Mosses, Burstall and Parsaye have favored implicit specifications of theories and abstract data types. Second, $\mathcal{SCRIPT}$ has more expressive power than CLEAR and OBJ in terms of modularization capabilities because $\mathcal{SCRIPT}$ allows definition of cyclic graphs of modules. Note that the restriction imposed by CLEAR or OBJ that only *acyclic* graphs of modules can be specified may force the definer to deviates from the most natural module organization.

Although $\mathcal{SCRIPT}$ is an object oriented functional language, the important mechanisms of *inheritance* and *dynamic binding* have not been used in the definition of ASPLE. A new methodology that will bring denotational to the realm of object oriented paradigm is certainly in order.