Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

E.A. van der Meulen

Algebraic specification of a compiler for a language with pointers

Computer Science/Department of Software Technology Report CS-R8848 December

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Research (N.W.O.).

Algebraic Specification of a Compiler for a Language with Pointers

E. A. van der Meulen

Department of Software Technology, Centre for Mathematics and Computer Science
P.O.Box 4079, 1009 AB Amsterdam, The Netherlands (email: emma@cwi.nl)

ASPLE is a simple programming language that has been used as an example in several studies on formal language definitions. The major characteristic of ASPLE is a pointer system in the spirit of Algol68. We present a complete definition of ASPLE in the formalism ASF+SDF. First, a complete specification is given of syntax, as well as static and dynamic semantics. Next, the simple stack machine language SML is introduced together with a definition of its dynamic semantics. Finally, the translation from ASPLE to SML is specified.

Key Words & Phrases: software engineering, algebraic specification, specification languages, typechecking and evaluation of languages, pointer system.

1980 Mathematical subject classification: 68Bxx **[Software]**.

1987 CR Categories: D.2.1 **[Software engineering]:** Requirements/Specifications - *Languages*; D3.1 **[Programming languages]:** Formal Definitions and Theory; F.3.1 **[Logics and Meanings of Programs]:** Specifying and Verifying and Reasoning about Programs - *Specification techniques*; F.3.2 **[Logics and Meanings of Programs]:** Semantics of Programming Languages - *Algebraic approaches to Semantics.*

Note: partial support received from the European Communities under ESPRIT project 348 (Generation of Interactive Programming Environments - GIPE).

Report CS-R8848
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Table of contents

1. Introduction

The goal of ESPRIT Project 348 (GIPE - Generation of Interactive Programming Environments) is to generate interactive programming environments from formal language definitions. In this context the Algebraic Specification Formalism ASF [BHK85, BHK87] and the Syntax Definition Formalism SDF [HK86, HHKR] have been developed. A shortcoming of ASF is that it only allows the use of prefix functions and a limited form of unary and binary operators. Therefore it has been combined with SDF that permits a more liberal use of syntax. We call the combination of both formalisms ASF+SDF. A typechecker of Mini-ML has been specified in ASF+SDF [Hen87].

In [HK88] an attempt has been made to make ASF+SDF specifications shorter by introducing negative conditions and omitting the specification of error cases. The toy language PICO has been specified in ASF+SDF using these features.

ASPLE is a simple programming language originally introduced in [CU73]. In [BML76] four formal definitions of ASPLE were given with the purpose of comparing specification formalisms. In [CDDHK85] a specification of ASPLE is given using TYPOL. Here we present a specification of ASPLE in ASF+SDF to be compared with the one given in [CDDHK85].

2. Organization of this paper

Section 3 presents a brief description of the specification formalism used in this paper.

In Section 4 the programming language ASPLE is introduced. We give a short description of the language and in particular of its pointer system in Section 4.1. The specification of the syntax of ASPLE is given in Section 4.2. The static and dynamic constraints on programs are formulated in Section 4.3 and a specification of the typechecking of ASPLE programs is presented in Section 4.4. In Section 4.5.1 we extend the ASPLE syntax with constructs that allow us to annotate programs with the type information that is needed during execution. The translation from ASPLE programs to annotated programs is specified in Section 4.5.2. These annotated programs are used in the specification of the dynamic semantics of ASPLE in Section 4.6.

In Section 5 we introduce the simple machine language SML. The specification of the syntax of SML is given in Section 5.1, while Section 5.2 contains the specification of its dynamic semantics.

Finally, the compilation of ASPLE programs to SML programs is specified in Section 6.

3. The specification formalism

We give a brief description of the algebraic specification formalism ASF, of the syntax definition formalism SDF and of the combined formalism ASF+SDF.

3.1. ASF

An ASF specification consists of a sequence of modules. Each module defines a signature and a set of conditional or unconditional equations over the signature. A signature consists of sorts and functions over these sorts.

Features in ASF to support the modular structure of a specification are:

- Exports: A module may have an exports-section with a (possibly incomplete) signature. The sorts and functions in the exports section are visible outside the module.
- Hiding: Sorts and functions local to a module are declared in the sort and function section outside the export section.
- Imports and Renamings: If a module uses other modules the names of those modules are given in the imports section. Upon import of a module it is possible to bind its parameters and to rename its signature.
- Parameters: A module may have a parameter section. It consists of (possibly incomplete) signatures which are formal parameters of the module and can be bound to actual parts of a module when the parameterized module is imported.

The semantics of an ASF specification is the initial algebra of its "normal form", i.e. the flat specification obtained by textual expansion of all modular constructs in the specification. The initial algebra can be represented by a term model constructed in the following way. Take all closed terms over the signature and

divide these into sets according to their sorts. A congruence relation is defined on closed terms by defining two closed terms to be equal if and only if their equality can be deduced from the equations using many-sorted equational logic. Finally each set of terms is divided into congruence classes.

In [BHK87] is described how an ASF specification with compound modules can be normalized. The result of such a normalization is one module that provides the initial algebra of the original specification.

An ASF system has been developed for compiling and testing ASF specifications [Hen88]. Equations are interpreted as rewrite rules for a conditional term rewriting system. This leads to a discrepancy between the formal semantics of a specification and the semantics that result from its implementation. The generated term rewriting system is sound but not complete. For all open terms t1 and t2 the following holds: if the implementation of the specification returns t2 as the result of evaluating t1, the equality of t1 and t2 can be deduced from the equations of the specifications (sound); on the other hand if two terms t1 and t2 can be proved equal using the equations of the specification they cannot always be shown equal using the implementation (not complete).

A first version of the specification presented in this paper has been written in ASF, implemented and tested using the ASF-system. Its incompleteness did not cause problems.

As a preparation for the implementation of ASF+SDF some new features were added to ASF as well as to the ASF implementation [Hen]. A second version of our specification has been written and tested using this extended version of ASF. We will now briefly describe the new features: lists, partial functions and negative conditions.

For each sort S lists can be defined: S+ denotes a list of one or more elements of sort S, S* denotes a list of zero or more elements of sort S.

Partial functions are denoted by the attribute {partial} following a function declaration. If a function is defined as partial the specification of error cases (e.g. the typechecking of incorrect programs) can be avoided. It is not clear yet how partial functions should be interpreted in the formal semantics of a specification. In the ASF system we used they have been implemented as follows: If a function f is defined as partial and a term say f(a) is offered to the resulting rewrite system and no equation is available to reduce the term f(a), f(a) is considered undefined and a message "unable to reduce f(a)" will be given. If f is not defined as partial, the reduction system will return the term f(a) as the normal form of f(a).

Negative conditions have been discussed in [HK88] as well. Using them does reduce the number of equations considerable. However, this may introduce ambiguity in the initial algebra. For instance it is not clear which elements can be put in one congruence class as result of the equation $a = c$ when $a \neq b$ [HK88, Kap87]. The implementation of negative conditions is as follows: Two terms x and y in a condition $x \neq y$ can be compared if both terms are defined. In that case their normal forms are compared. If at least one of the terms is undefined the condition $x \neq y$ fails.

3.2. SDF

In the syntax definition formalism SDF concrete and abstract syntax of a language can be defined simultaneously. The abstract syntax is a signature, the concrete syntax is described in the form of 'BNF rules in reverse order'. SDF will be described extensively in [HHKR].

The following information can be derived from an SDF definition:

A derived regular grammar and a derived BNF grammar, defining a set of well-formed strings.

A derived signature, consisting of the sorts defined in the specification, a prefix version of the functions declared and some derived and predefined sorts and functions. The derived signature defines a set of well-formed abstract syntax trees.

An SDF definition consists of five sections, the contents of which we describe briefly:

- Sorts: A listing of sort names.
- Lexical syntax: Here the lexical tokens of the language are defined. The section consists of one or more function declarations, each consisting of a regular expression and a result sort. Character classes like [a-z] or [0-9] can be used for abbreviating an enumeration of characters. The operators * and + can be used to indicate a repetition. The sort LAYOUT is predefined and can be used only as a result sort in the lexical section. Layout functions serve the purpose of defining layout characters, comments etc.

- Context-free syntax: A list of functions is defined. Functions are declared by giving their syntax and their output type. Lists with or without separators can be defined using {} and the operators + and *. Four attributes can be used to resolve ambiguities. The attribute {bracket} is used to define a bracket function, which allows us to introduce parentheses in the concrete syntax without affecting the underlying abstract syntax. It may also be used to improve readability of the specification. The attributes {assoc}, {left} and {right} indicate the associativity, left associativity or right associativity of a function.
- Priorities: The priority of functions can be declared. Functions with a higher priority bind more strongly. This too helps in resolving ambiguities.
- Variables: A list of declarations of variables together with their sorts can be defined.

3.3. ASF+SDF

In the combination formalism ASF+SDF the signature definition in an ASF module is replaced by an SDF definition. Partial functions and negative conditions are features of ASF+SDF as well. To find the semantics of a specification in ASF+SDF it has to be translated to ASF. The initial algebra of the resulting ASF specification is the meaning of the ASF+SDF specification. Translation of an ASF+SDF specification to ASF implies:

replacing the SDF definition in each module by its derived signature,
changing all equations so that each function is written as a prefix function.

3.4. Booleans, naturals and identifiers

We will not give the specifications for the Booleans, natural numbers and the identifiers here, but only list the required sorts and functions. The sort BOOL, with constants `true` and `false` is defined in a module Bool-con. This module is imported by the module Booleans, in which the logical functions "~" (negation), "|" (disjunction) and "&" (conjunction) are defined. This way the constants `true` and `false` can be imported by modules (e.g ASPLE syntax) in which we need those constants but not Boolean expression like `true & false`. For the same reason we define the natural numbers, sort NAT, in a separate module Nat-con. Nat-con, in its turn, is imported by the module Naturals, in which we define the functions "+" and "*".

The module Identifiers only consists of a lexical syntax. Any string of characters starting with a letter followed by zero or or more letters or numbers is an identifier.

4. ASPLE

ASPLE is a simple programming language derived from Algol68. Especially the pointer system of Algol68 is complicated and therefore interesting to specify even though this feature will turn out to be quite useless in ASPLE itself. ASPLE has assignment, if-then, if-then-else, while-do, input and output statements. There are two primitive types: integer and Boolean. Operators +, *, = and ≠ apply to both integer and Boolean expressions. Applied to integers, + and * represent addition and multiplication, respectively. Applied to Boolean expressions, they represent the logical 'or' and 'and' operations. The context-free syntax of ASPLE is defined in [BML76].

An ASPLE program consists of a declaration section and a statement section. Identifiers that are used in the statements must be defined together with their modes in the declaration section. An identifier can have mode integer, Boolean or reference-to-a-mode. We will discuss this pointer system in Section 4.1 An example of an ASPLE program, which computes and prints the factorial of the input value `x`, is given in example 1 [BML76].

ASPLE-program

```
begin
   int x, y, z;
   input x;
   y := 1;
   z := 1;
   if (x ≠ 0)
     then
          while (z ≠ x) do
                z := z + 1;
                y := y * z
          end
   fi;
   output y
end
```

example 1

4.1. Modes

The *mode* of an identifier defines the number of indirection steps between the identifier and its value as well as the type, integer or Boolean, of that value.

With each of the identifiers `x`, `y`, `z` in example 1, a single pointer is associated. If, for instance, identifier `i` has been given mode `ref ref bool` in the declaration section, three pointers form the path from `i` to a Boolean value. This identifier can refer to an identifier `j` of mode `ref bool`, that, in turn, can refer to an identifier `k` of mode `bool` (figure 1).

We will call the mode that has been assigned to an identifier in the declaration section the declared mode. It is clear that an identifier that has been given mode `int` is not an integer itself but refers to some integer. So the declared mode has one reference less than the *actual mode*. In all figures the actual number of references will be shown!

figure 1

If the last pointer in the path associated with an identifier points to an integer or a Boolean value this value is called the *primitive value* of the identifier. The *primitive mode* of an identifier is the type of the values that can be reached by its last pointer. We will also speak of the primitive mode of a mode meaning the primitive mode of identifiers of that mode.

Modes are specified in the module Mode-con. The bracket function for modes is defined for reasons of readability and will be used in combination with `ref`.

In the module Modes the `result-mode` of two modes is specified as their common primitive mode. It may be integer or Boolean. We will need this notion when typechecking assignment statements or expressions.

A mode M is smaller than mode M1 if they have the same primitive mode and the number of ref's in M is smaller than the number of ref's in M1. The next section will make clear why comparing the number of pointers of two modes is relevant.

The `is-bool` predicate will be used to typecheck if and while statements.

The functions `result-mode` and ≤ have the predicate **{partial}**. Equations for those functions are given only for modes that have the same primitive mode.

```
module Mode-con
begin

  exports
  begin
    sorts MODE
    context-free syntax
      bool                                  -> MODE
      int                                   -> MODE
      ref MODE                              -> MODE
      "(" MODE ")"                          -> MODE       {bracket}
    end

end Mode-con

module Modes
begin

  exports
  begin
    context-free syntax
      MODE "≤" MODE                         -> BOOL       {partial}
      is-bool MODE                          -> BOOL       {partial}
      result-mode "(" MODE "," MODE ")"     -> MODE       {partial}
  end

  imports
    Mode-con, Booleans

  variables
    M, M1                  -> MODE

  equations

  [M1]    M ≤ M  = true

  [M2]    M ≤ M1  = true
          ---------------
          M ≤ ref(M1)  = true

  [M3]    bool ≤ M  = true
          ----------------
          is-bool M = true

  [M4]    result-mode (M1, M) = M
          ------------------------------
          result-mode (ref(M1), M) = M

  [M5]    result-mode (M, M1) = M
          ------------------------------
          result-mode (M, ref(M1)) = M

  [M6]    result-mode (int,int) = int

  [M7]    result-mode (bool,bool) = bool

end Modes
```

4.2. The syntax of ASPLE

The specification of the syntax of ASPLE is rather straightforward.

The sort STMS is introduced because lists are not allowed as output sort of functions. Though we do not need the sort STMS yet, we will need it in the specification of the translation to annotated programs in Section 4.5.2.

Due to the use of `fi` to finish an if statement no ambiguity problems with nested if statements (the dangling else problem) will occur when parsing an ASPLE program. Operators + and * are left associative which is indicated by the attribute **{left}**. The operator * binds more strongly than the operator +. This is defined in the priorities section. The bracket function for expressions is needed to be able to change the priorities of * and +. In ASPLE, it is obligatory to write parentheses around expressions containing = or ≠.

```
module ASPLE-syntax
begin

exports
  begin
    sorts PROGRAM, DECL, STM, STMS, EXP
    lexical  syntax
      [\t\n\r]                              -> LAYOUT     -- tab, new line, return
    context-free  syntax
      begin {DECL ";"}* ";" STMS end        -> PROGRAM
      MODE {ID ","}+                        -> DECL
      {STM ";"}+                            -> STMS
      ID ":=" EXP                           -> STM
      input ID                              -> STM
      output EXP                            -> STM
      if EXP then STMS fi                   -> STM
      if EXP then STMS else STMS fi         -> STM
      while EXP do STMS end                 -> STM
      EXP "+" EXP                           -> EXP        {left}
      EXP "*" EXP                           -> EXP        {left}
      "(" EXP "=" EXP ")"                   -> EXP
      "(" EXP "≠" EXP ")"                   -> EXP
      "(" EXP ")"                           -> EXP        {bracket}
      NAT                                   -> EXP
      BOOL                                  -> EXP
      ID                                    -> EXP

    priorities
      {"*"} > {"+"}
  end

  imports
    Mode-con, Bool-con, Nat-con, Identifiers

end ASPLE-syntax
```

4.3. Static and dynamic constraints on ASPLE programs

We formulate the constraints on ASPLE programs that have to be checked either during typechecking or at run time.

R1 An identifier can occur only once in the declaration section.

R2 An identifier must have been given a primitive value before it can be used in an operator expression.

R3 a.The operators +, *, = and ≠ can be applied to any pair of identifiers or expressions with equal primitive mode.

Since **R2** has to be satisfied an operator applied to a pair of identifiers can actually act on their primitive values and hence yield an integer or Boolean value. So

b. operator expressions and expressions consisting of integers or Boolean constants have zero references.

We need two auxiliary notions before we can formulate the remaining constraints. We introduce the notation *n(exp)* to indicate the number of actual references (pointers) of an expression. We will also need the definition of a chain. A *chain* from x_1 to x_k is a series $\{x_1...x_k\}$ in which

- x_k is a constant or an identifier, $x_1..x_{k-1}$ are identifiers,
- $n(x_i) = n(x_{i+1}) + 1$,
- all x_i have equal primitive mode,
- x_i refers to x_{i+1}.

A chain can be constructed by means of assignment and input statements.

R4 An assignment statement x := exp is correct if

a. x and exp have the same primitive mode, and $n(x) \leq n(exp) + 1$,

b. if $n(x) \leq n(exp)$, *which means exp is an identifier*, say y, a chain must exist from y to some identifier or constant a such that $n(x) = n(a) + 1$.

R5 a. An input value must have the same primitive mode t as the identifier it is assigned to.

b. A chain must exist from the identifier in the input statement to some identifier of mode ref t.

R6 If an output statement has an *identifier* as argument, a chain must exist from this identifier to an identifier of actual mode ref t, with t a primitive mode.

R7 The expressions in if and while statements must be of mode Boolean.

Knowing **R4** we can reformulate the first part of **R2** as: a chain must exist from an identifier to a primitive value. When we combine **R3b** and **R4** it is clear that a statement like x:= x + 1 is only correct when the declared mode of x is int. Similarly x:= y * z is correct when x, y and z have the same primitive mode t and the actual mode of x is ref t. Example 2 illustrates **R2**, **R3** and **R4**.

```
begin
   int x, u;
   ref int v;
   u := 5;                          -- R4a
   x := u;                          -- R4b
   v := u;                          -- R4a
   u := x + v                       -- R2,3,4a
end
```

example 2

The statement `v := u` can be pictured as making the first pointer of the chain emanating from `v` point to the first pointer of the chain emanating from `u` (figure 2a). Changing the value of `u` in the assignment `u := x + v` also changes the primitive value of `v` (figure 2b).

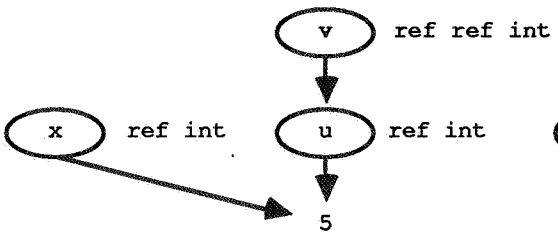


figure 2a

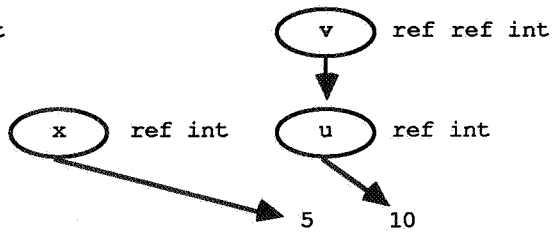
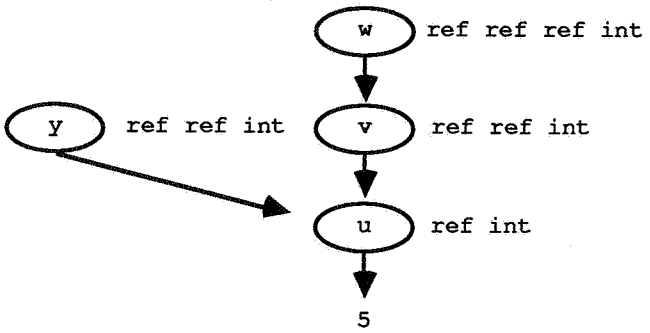


figure 2b

Example 3 illustrates **R4b**: n(u) = n(y) - 1 so **R4b** for statement `y := w` is satisfied. The primitive value of `y` is the primitive value of `w`.

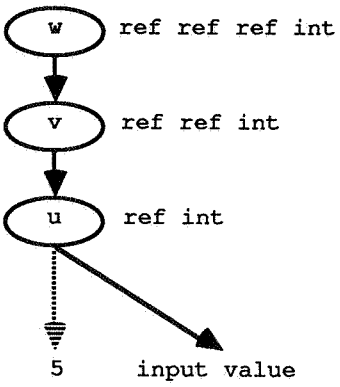
```
begin
  int u;
  ref int v, y;
  ref ref int w;
  u := 5;
  v := u;
  w := v;
  y := w      -- R4b
end
```



example 3 figure 3

Example 4 shows that evaluating the statement `input w` can be seen as making the last pointer in the chain emanating from `w` point at the input value (figure 4).

```
begin
  int u;
  ref int v;
  ref ref int w;
  u := 5;
  v := u;
  w := v;
  input w;
  output u;
  output v;
  output w
end
```



example 4 figure 4

When running the program in example 4 with some integer, say 8, as input, the output will be `8, 8, 8`. Note that replacing `input w` by `input v` or `input u` would lead to the same output.

When an ASPLE program is evaluated **R1**, **R3**, **R4a** and **R7** are checked in the typechecking phase. Obviously, **R5a** can only be checked at run time since we need to know the value that occurs as input. As for the checking of the required construction of chains in **R2**, **R4b**, **R5b** and **R6**: since the evaluation of assignment or input statements may depend on the evaluation of conditions in if statements or while loops, chains can not be detected during typechecking. So whether or not a chain has been constructed has to be checked at run time as well.

4.4. Typechecking ASPLE

4.4.1. Mode environments

When typechecking the declaration section of an ASPLE program a table called *Mode-environment* is constructed for the declared identifiers together with their mode. First, we define a parameterized module *Tables*. Next we instantiate this module to obtain *Mode-environments*.

```
module Tables
begin

  parameters
  begin
    sorts KEY, ENTRY
  end
```

```
exports
begin
  sorts PAIR, TABLE
  context-free  syntax
    KEY ":" ENTRY                          -> PAIR
    table "(" {PAIR "#"}* ")"              -> TABLE
    lookup KEY in TABLE                    -> ENTRY      {partial}
    add PAIR to TABLE                      -> TABLE
    modify PAIR in TABLE                   -> TABLE
    KEY is in TABLE                        -> BOOL
    "(" PAIR ")"                           -> PAIR       {bracket}
end

imports
  Booleans

variables
  Pair                -> PAIR              Pairs, Pairs'       -> {PAIR "#"}*
  Key, Key'           -> KEY               Ent, Ent'           -> ENTRY

equations

[T1]    add Pair to table (Pairs) = table (Pair # Pairs)

[T2]    lookup Key in table ((Key : Ent) # Pairs) = Ent

                          Key ≠ Key'
[T3]    ------------------------------------------------------
        lookup Key in table ((Key': Ent) # Pairs) =
        lookup Key in table (Pairs)

[T4]    modify (Key : Ent) in table ((Key : Ent') # Pairs) =
        table ((Key : Ent) # Pairs)

                               Key ≠ Key'
[T5]    ------------------------------------------------------------------
        modify (Key : Ent) in table ((Key' : Ent') # Pairs) =
        table ((Key' : Ent') # modify (Key : Ent) in table (Pairs))

[T6]    modify (Key : Ent) in table () =  table (Key : Ent)

[T7]    Key is in table () = false

[T8]    Key is in table ((Key : Ent) # Pairs) = true

                                    Key ≠ Key'
[T10]   ------------------------------------------------------------------
        Key is in table ((Key' : Ent) # Pairs) = Key is in table (Pairs)

end Tables

module Mode-environments
begin

  imports
    Tables
      Keys bound by
           sorts
                KEY -> ID to Identifiers
      Entries bound by
           sorts
                ENTRY -> MODE to Mode-con
      renamed  by
           sorts
                TABLE => MENV
                PAIR => MPAIR
           functions
```

```
            table => menv
    end  renaming

end Mode-environments
```

4.4.2. ASPLE-tc

Typechecking an ASPLE program consists of two phases. First, all declaration information is collected in a mode environment (Tc2-5). Next, this information is used to check the statement section of the program (Tc6-19). In the mode environment the actual mode of an identifier is recorded so one more `ref` is added to the declared mode of each identifier (Tc5). The condition of Tc5 states that an identifier cannot be declared more than once. This forms a check of **R1**.

In the typechecking of operator expressions **R3** is checked (Tc16-19). Since the result-mode of two modes is defined for modes with equal primitive mode only, the conditions in Tc16-19 make sure that operators are applied to pairs of expressions with equal primitive mode (**R3a**). The mode of an operator expression is the result-mode of the modes of its arguments. This result-mode is a mode without references (**R3b**).

The mode environment is used to typecheck statements. Tc7 specifies the typechecking of assignment statements. In the condition of Tc7 **R4a** is checked. In Tc10-12 the `is-bool` function is used to check that the expression in if and while statements are Boolean expressions. This forms a check of **R7**.

```
module ASPLE-tc
begin

  exports
  begin
    context-free  syntax
      tc "[" PROGRAM "]"                  -> BOOL      {partial}
      tc "[" {DECL ";"}*"]" in MENV       -> MENV      {partial}
      tc "[" {STM ";"}+ "]" in MENV       -> BOOL      {partial}
      tc "[" EXP "]" in MENV              -> MODE      {partial}
  end

  imports
    ASPLE-syntax, Modes, Mode-environments

  variables
    Id                  -> ID          Id-list            -> {ID ","}+
    Decl                -> DECL        Decls              -> {DECL ";"}*
    Mode, M, M1, M2     -> MODE
    Stm                 -> STM         Stms, Stms1, Stms2 -> STMS
    Exp,Exp1,Exp2       -> EXP
    Nat                 -> NAT         Bool, Bool1        -> BOOL
    E                   -> MENV

  equations

-- typechecking programs

  [Tc1]    tc[Decls] in menv() = E,  tc[Stms] in E = true
           ------------------------------------------------
                 tc[begin Decls ; Stms end] = true

-- typechecking declarations

  [Tc2]    tc[Decl; Decls] in E = tc[Decls] in tc[Decl] in E

  [Tc3]    tc[] in E = E

  [Tc4]    tc[Mode Id, Id-list] in E =
           tc[Mode Id-list] in tc[Mode Id] in E

  [Tc5]    Id is in E = false
           ------------------------------------------
           tc[Mode Id] in E = add (Id:ref(Mode)) to E
```

```
-- typechecking statements

[Tc6]   tc[Stm] in E = true,  tc[Stms] in E = true
        ---------------------------------------------
              tc[Stm; Stms] in E = true

[Tc7]   tc[Id] in E = M1,  tc[Exp] in E = M2,  M1 ≤ ref(M2) = true
        ---------------------------------------------------------
                    tc[Id := Exp] in E = true

[Tc8]      tc[Id] in E = M
        ---------------------
        tc[input Id] in E = true

[Tc9]      tc[Exp] in E = M
        ---------------------
        tc[output Exp] in E = true

[Tc10]  is-bool (tc[Exp] in E) = true,  tc[Stms1] in E = true
        ---------------------------------------------------
              tc[if Exp then Stms1 fi] in E = true

[Tc11]   is-bool (tc[Exp] in E) = true,
         tc[Stms1] in E = true,  tc[Stms2] in E = true
        ---------------------------------------------
        tc[if Exp then Stms1 else Stms2 fi] in E = true

[Tc12]  is-bool (tc[Exp] in E) = true,  tc[Stms] in E = true
        ---------------------------------------------------
              tc[while Exp do Stms end] in E = true

-- typechecking expressions

[Tc13]  tc[Nat] in E = int

[Tc14]  tc[Bool] in E = bool

[Tc15]  tc[Id] in E = lookup Id in E

[Tc16]  tc[Exp1] in E = M1,  tc[Exp2] in E = M2,  result-mode(M1,M2) = M
        ---------------------------------------------------------------
                    tc[Exp1 + Exp2] in E = M

[Tc17]  tc[Exp1] in E = M1,  tc[Exp2] in E = M2,  result-mode(M1,M2) = M
        ---------------------------------------------------------------
                    tc[Exp1 * Exp2] in E = M

[Tc18]  tc[Exp1] in E = M1,  tc[Exp2] in E = M2,  result-mode(M1,M2) = M
        ---------------------------------------------------------------
                    tc[(Exp1 = Exp2)] in E = bool

[Tc19]  tc[Exp1] in E = M1,  tc[Exp2] in E = M2,  result-mode(M1,M2) = M
        ---------------------------------------------------------------
                    tc[(Exp1 ≠ Exp2)] in E = bool

end ASPLE-tc
```

4.5. Translation to annotated ASPLE programs

Annotated ASPLE programs are now introduced to simplify the specification of both the dynamic semantics of ASPLE and the translation of ASPLE to SML. An annotated program contains all the type information needed during its execution.

The most complicated part of the translation from ASPLE programs to annotated ASPLE programs regards the *dereferencing* of identifiers. To come closer to or obtain the primitive value of an identifier we can dereference the identifier to another mode. So we go down the pointer chain from the identifier towards its primitive value. We describe dereferencing by introducing a new sort VAR (variable) and a function *deref*. All identifiers are variables. The function deref applied to a variable yields another variable that is one step closer to the primitive mode of the former (figure 5).

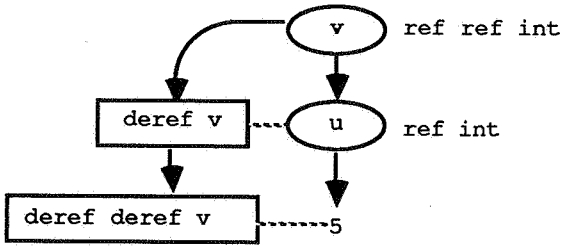


figure 5

The following rules are applied in the translation:

- Identifiers in expressions are dereferenced to their primitive modes. This will be used to check **R2** at run time.
- The operators +, *, = and ≠ are overloaded as they apply to integer as well as to Boolean expressions. In the machine language SML the operators = and ≠ are overloaded likewise but + and * apply to integers only, whereas | and & are the corresponding operators for Boolean expressions. In ASPLE-stat + and * applied to Boolean expressions are translated to | and & respectively.
- In case the expression on the right hand side of an assignment statement is an identifier, it is dereferenced to a mode that has one reference less than the mode of the identifier on the right hand side. We will use this to check **R4b** at runtime.
- The identifier in an input statement is dereferenced to an actual mode ref t, where t is a primitive mode. Type t is added as annotation to the input statement. We will use this to check **R5** at run time.
- In the statement output x, for any identifier x, x must be dereferenced to its primitive mode. We will use this to check **R6** at runtime.

We will first describe extensions to the ASPLE-syntax needed for the annotations, next we describe the translation from ASPLE programs to annotated ASPLE programs.

4.5.1. ASPLE-extended-syntax

Sorts and functions that are needed in annotated ASPLE programs are introduced in the module ASPLE-extended-syntax.

```
module ASPLE-extended-syntax
begin

  exports
  begin
    sorts VAR
    context-free  syntax
      EXP "&" EXP                          -> EXP        {left}
      EXP "|" EXP                          -> EXP        {left}
      deref VAR                            -> VAR
      tinput VAR ":" MODE                  -> STM
      VAR                                  -> EXP
      ID                                   -> VAR

    priorities
      {"*", "&"} > {"+", "|"}
  end

  imports
    ASPLE-syntax

end ASPLE-extended-syntax
```

4.5.2. ASPLE-static semantics

As an example of the translation of ASPLE programs, we give the translation of the program in example 1. Note that the dereferencing of identifiers is made explicit and that type information has been added to input statements.

program

```
begin
   int x, y, z;
   input x;
   y := 1;
   z := 1;
   if (x ≠ 0)
    then
     while (z ≠ x) do
           z := z + 1;
           y := y * z
     end
   fi;
   output y
end
```

annotated program

```
begin
   int x, y, z;
   tinput x : int;
   y := 1;
   z := 1;
   if (deref x ≠ 0)
    then
       while (deref z ≠ deref x) do
            z := deref z + 1;
            y := deref y * deref z
    end
   fi;
   output deref y
end
```

example 5

The translation to annotated programs is specified in the module ASPLE-stat. The translation function uses the typechecking function defined in Section 4.4.2. When an ASPLE program is translated, a mode environment is constructed by typechecking the declaration section. Statements will be translated when they are "type-correct" only, since no equations are given for incorrect statements.

We first have a look at the translation of expressions (Tr11-23). Of course, translating integer and Boolean values does not cause any change (Tr11,12). Translating an identifier in a context of a certain mode may require dereferencing it (Tr13-15). The function *dereference* then yields a variable that is some references closer to the primitive mode of the identifier, as required by the context. Dereference uses the function deref (figure 6).

```
dereference v from ref ref int to int = deref deref v
```

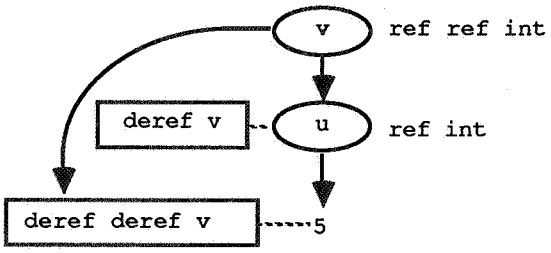


figure 6

Translating an operator expression implies dereferencing of all its identifiers to their primitive mode and resolving the overloading of the operators * and + (Tr16-23).

The translation of an assignment statement is the translation of the expression on the right hand side into an expression with a mode that has one reference less than the mode of the identifier on the left hand side (Tr3). The conditions in Tr3 correspond to **R4a.**

Identifiers in input statements are dereferenced and their primitive mode is added to the annotated statement (Tr4,5). The expression in output statements (Tr6,7) is translated to its primitive mode. In case this expression is a single identifier the identifier is dereferenced to its primitive mode.

The translation of if statements and while loops is the translation of the expressions to primitive mode bool and the translation of the statements.

```
module ASPLE-stat
begin

  exports
  begin
    context-free  syntax
      tr "[" PROGRAM "]"                  -> PROGRAM    {partial}
      tr "[" STMS "]" in MENV             -> STMS       {partial}
      tr "[" EXP "]" in MENV to MODE      -> EXP        {partial}
      dereference ID from MODE to MODE    -> VAR        {partial}
  end

  imports
    ASPLE-extended-syntax,  ASPLE-tc

  variables
    Decl                  -> DECL          Decls                 -> {DECL ";"}*
    Id, Id'               -> ID            Id-list               -> {ID ","}+
    M, M1                 -> MODE
    Stm, Stm'             -> STM
    Stms, Stms1, Stms2    -> STMS          Stms',Stms1',Stms2'   -> STMS
    Exp, Exp1, Exp2,      -> EXP           Exp',Exp1',Exp2'      -> EXP
    Nat                   -> NAT           Bool                  -> BOOL
    E                     -> MENV          Var                   -> VAR

  equations

-- translation of programs

  [Tr1]   tc[Decls] in menv () = E, tr[Stms] in E = Stms'
          ------------------------------------------------
          tr[begin Decls ; Stms end] = begin Decls ; Stms' end

-- translation of statements

  [Tr2]   tr[Stm; Stms] in E = tr[Stm] in tr[Stms] in E

  [Tr3]   lookup Id in E = ref(M), tr[Exp] in E to M = Exp'
          ------------------------------------------------
          tr[Id := Exp] in E = Id := Exp'

  [Tr4]   tr[input Id] in E = tinput tr[Id] in E to ref(int) : int

  [Tr5]   tr[input Id] in E = tinput tr[Id] in E to ref(bool) : bool

  [Tr6]   tr[output Exp] in E = output tr[Exp] in E to int

  [Tr7]   tr[output Exp] in E = output tr[Exp] in E to bool

  [Tr8]   tr[if Exp then Stms1 fi] in E =
          if tr[Exp] in E to bool then tr[Stms1] in E fi

  [Tr9]   tr[if Exp then Stms1 else Stms2 fi] in E =
          if tr[Exp] in E to bool then tr[Stms1] in E
          else tr[Stms2] in E fi

  [Tr10]  tr[while Exp do Stms end] in E =
          while tr[Exp] in E to bool do tr[Stms] in E end

-- translation of expressions

  [Tr11]  tr[Nat] in E to int = Nat

  [Tr12]  tr[Bool] in E to bool = Bool
```

[Tr13] $$\frac{\text{lookup Id in E = M,}\quad \text{M1} \leq \text{M = true}}{\text{tr[Id] in E to M1 = dereference Id from M to M1}}$$

[Tr14] $$\frac{\text{M1} \leq \text{M = true}}{\text{dereference Id from ref(M) to M1 = deref dereference Id from M to M1}}$$

[Tr15] dereference Id from M to M = Id

[Tr16] $$\frac{\text{tr[Exp1] in E to int = Exp1', tr[Exp2] in E to int = Exp2'}}{\text{tr[Exp1 + Exp2] in E to int = Exp1' + Exp2'}}$$

[Tr17] $$\frac{\text{tr[Exp1] in E to bool = Exp1', tr[Exp2] in E to bool = Exp2'}}{\text{tr[Exp1 + Exp2] in E to bool = Exp1' | Exp2'}}$$

[Tr18] $$\frac{\text{tr[Exp1] in E to int = Exp1', tr[Exp2] in E to int = Exp2'}}{\text{tr[Exp1 * Exp2] in E to int = Exp1' * Exp2'}}$$

[Tr19] $$\frac{\text{tr[Exp1] in E to bool = Exp1', tr[Exp2] in E to bool = Exp2'}}{\text{tr[Exp1 * Exp2] in E to bool = Exp1' \& Exp2'}}$$

[Tr20] $$\frac{\text{tr[Exp1] in E to int = Exp1', tr[Exp2] in E to int = Exp2'}}{\text{tr[(Exp1 = Exp2)] in E to bool = (Exp1' = Exp2')}}$$

[Tr21] $$\frac{\text{tr[Exp1] in E to bool = Exp1', tr[Exp2] in E to å = Exp2'}}{\text{tr[(Exp1 = Exp2)] in E to bool = (Exp1' = Exp2')}}$$

[Tr22] $$\frac{\text{tr[Exp1] in E to int = Exp1', tr[Exp2] in E to int = Exp2'}}{\text{tr[(Exp1} \neq \text{Exp2)] in E to bool = (Exp1'} \neq \text{Exp2')}}$$

[Tr23] $$\frac{\text{tr[Exp1] in E to bool = Exp1', tr[Exp2] in E to bool = Exp2'}}{\text{tr[(Exp1} \neq \text{Exp2)] in E to bool = (Exp1'} \neq \text{Exp2')}}$$

end ASPLE-stat

4.6. ASPLE dynamic semantics

When executing an ASPLE program a *value environment*, a table with identifiers and their values is used. Input values will be taken from a list called INPUT and output values will be added to a list called OUTPUT.

First, we will specify the auxiliary notions Values, Value-environments and States. Then we will present the specification of the dynamic semantics of ASPLE.

4.6.1. Values and Value-environments

Values for input and output can be integer or Boolean constants. We also need identifiers as values, since they can be assigned to other identifiers.

```
module Values
begin

  exports
  begin
    sorts IO-VAL, VAL
    context-free  syntax
      BOOL                              -> IO-VAL
      NAT                               -> IO-VAL
      IO-VAL                            -> VAL
      ID                                -> VAL
  end

  imports
```

```
    Bool-con, Nat-con, Identifiers

end Values

module Value-environments
begin

  imports
    Tables
      Keys bound by
           sorts
                KEY -> ID to Identifiers
      Entries bound by
           sorts
                ENTRY -> VAL to Values
      renamed by
           sorts
                TABLE => VALENV
           functions
                table => valenv
      end renaming

end Value-environments
```

4.6.2. ASPLE-states

A *state* consists of a value environment, a list of input values and an output list. The effect of evaluating a statement is described by modifications to a given state.

```
module ASPLE-states
begin

  exports
  begin
    sorts STATE, INPUT, OUTPUT
    context-free syntax
      input "("{IO-VAL ","}* ")"                    -> INPUT
      output "(" {IO-VAL ","}* ")"                  -> OUTPUT
      "<" VALENV "," INPUT "," OUTPUT ">"           -> STATE
  end

  imports
    Value-environments

end ASPLE-states
```

4.6.3. ASPLE-ds

The dynamic semantics of ASPLE is defined in module ASPLE-ds. A program is translated (and thus type-checked) before it is executed (Ev1).

We will now discuss the evaluation of expressions and of statements in some detail.

The equations Ev13-15 are straightforward: the evaluation of an integer value, a Boolean value or an identifier is the value itself. Ev16 handles the evaluation of dereferenced identifiers. Example 6 illustrates how the value environment is affected by the evaluation of the statements in a program. Only assignment and input statements cause updates of this table.

annotated program	value environment
begin	
int u;	valenv ()
ref int v, y;	valenv ()
ref ref int w	valenv ()
u := 5;	valenv (u:5)
v := u;	valenv (u:5 # v:u)
w := v;	valenv (u:5 # v:u # w:v)
y := deref deref w;	valenv (u:5 # v:u # w:v # y:u)
tinput deref deref w : int;	valenv (u:8 # v:u # w:v # y:u)
output deref u;	valenv (u:8 # v:u # w:v # y:u)
output deref deref v;	valenv (u:8 # v:u # w:v # y:u)
output deref deref deref w	valenv (u:8 # v:u # w:v # y:u)
end	
with input 8	

example 6

We explain the evaluation of `deref deref w` in example 6:

In trying to fulfill the first condition of Ev16 all 'deref's' are peeled off. `Ev[w] = w` in any value environment (Ev15). In the value environment of example 6 the value of `w` is `v`, so `ev[deref w] = v`. The value of `v` is `u`, so `ev[deref deref w] = u`.

Note that a dereferenced identifier x with k deref's can be evaluated only if a chain exists from x to some y with n(y) = n(x) - k.

Equations Ev17-23 describe the evaluation of expressions with operators. Since all identifiers occurring in expressions have already been dereferenced (Section 4.5.2) Ev13-16 are used to check **R2** and to obtain the primitive value of an expression. Consider the program in example 7 and its translation.

program	annotated program
begin	begin
bool k, l, m;	bool k, l, m;
ref bool p;	ref bool p;
k := false;	k := false;
l := true;	l := true;
p := l;	p := l;
m := k + p	m := deref k | deref deref p
end	end

example 7

We explain the evaluation of the last statement of this program.

Using Ev14-16 `deref k` evaluates to `false` and `deref deref p` evaluates to `true`. Ev18 then results in `ev[deref k | deref deref p] = false | true`, which is `true`.

Ev15 and Ev16 are also used to evaluate dereferenced identifiers in assignment, input and output statements (Ev3-6). This forms a check of **R4b**, **R5b** and **R6** .

The mode annotation of an identifier in an input statement is used to check that the mode of the input value matches the one of the identifier in the input statement (Ev4,5). This forms a check of **R5a**.

The evaluation of if statements and while loops is straightforward (Ev7-12).

```
module ASPLE-ds
begin

  exports
  begin
    context-free  syntax
      ev "[" PROGRAM "]" with INPUT                    -> OUTPUT    {partial}
      ev "[" {STMS ";"}+ "]" in STATE                  -> STATE     {partial}
      ev "[" EXP "]" in STATE                          -> VAL       {partial}
  end
```

imports
ASPLE-stat, ASPLE-states, Naturals

variables

Id	-> ID	Id-list	-> {ID ","}+
Var	-> VAR	Mode	-> MODE
Decls	-> {DECL ";"}*	Stm	-> STM
Stms,Stms',Stms1,Stms2	-> {STM ";"}+	Exp,Exp1,Exp2	-> EXP
S, S1, S2	-> STATE	E, E'	-> VALENV
I, I'	-> INPUT	O, O'	-> OUTPUT
Nat, Nat1, Nat2	-> NAT	Bool, Bool1, Bool2	-> BOOL
Val, Val1, Val2	-> VAL	Vals	-> {VAL ","}*

equations

-- evaluation of programs

[Ev1] tr[begin Decls ; Stms end] = begin Decls; Stms' end,
ev[Stms'] in <valenv (), I, output ()> = <E, I', O>
ev[begin Decls ; Stms end] with I = O

-- evaluation of annotated statements

[Ev2] ev[Stm; Stms] in S = ev[Stms] in ev[Stm] in S

[Ev3] $\dfrac{\text{ev[Exp] in <E, I, O> = Val}}{\text{ev[Id := Exp] in <E, I, O> = <modify (Id:Val) in E, I, O>}}$

[Ev4] $\dfrac{\text{ev[Var] in <E, input (Nat, Vals), O> = Id}}{\begin{array}{l}\text{ev[tinput Var : int] in <E, input (Nat, Vals), O> =}\\ \quad\text{<modify (Id:Nat) in E,input (Vals), O>}\end{array}}$

[Ev5] $\dfrac{\text{ev[Var] in <E, input (Bool, Vals), O> = Id}}{\begin{array}{l}\text{ev[tinput Var : bool] in <E, input (Bool, Vals), O> =}\\ \quad\text{<modify (Id:Bool) in E, input (Vals), O>}\end{array}}$

[Ev6] $\dfrac{\text{ev[Exp] in <E, I, output (Vals)> = Val}}{\text{ev[output Exp] in <E, I, output (Vals)> = <E, I, output (Vals, Val)>}}$

[Ev7] $\dfrac{\text{ev[Exp] in S = true}}{\text{ev[if Exp then Stms1 else Stms2 fi] in S = ev[Stms1] in S}}$

[Ev8] $\dfrac{\text{ev[Exp] in S = false}}{\text{ev[if Exp then Stms1 else Stms2 fi] in S = ev[Stms2] in S}}$

[Ev9] $\dfrac{\text{ev[Exp] in S = true}}{\text{ev[if Exp then Stms1 fi] in S = ev[Stms1] in S}}$

[Ev10] $\dfrac{\text{ev[Exp] in S = false}}{\text{ev[if Exp then Stms1 fi] in S = S}}$

[Ev11] $\dfrac{\text{ev[Exp] in S = true, ev[Stms] in S = S1,}\quad \text{ev[while Exp do Stms end] in S1 = S2}}{\text{ev[while Exp do Stms end] in S = S2}}$

[Ev12] $\dfrac{\text{ev[Exp] in S = false}}{\text{ev[while Exp do Stms end] in S = S}}$

-- evaluation of annotated expressions

[Ev13] ev[Nat] in S = Nat

[Ev14] ev[Bool] in S = Bool

```
[Ev15]   ev[Id] in S = Id

[Ev16]   ev[Var] in <E,I,O> = Id,  lookup Id in E = Val
         ------------------------------------------------
              ev[deref Var] in <E,I,O> = Val

[Ev17]   ev[Exp1 + Exp2] in S = ev[Exp1] in S + ev[Exp2] in S

[Ev18]   ev[Exp1 | Exp2] in S = ev[Exp1] in S | ev[Exp2] in S

[Ev19]   ev[Exp1 * Exp2] in S = ev[Exp1] in S * ev[Exp2] in S

[Ev20]   ev[Exp1 & Exp2] in S = ev[Exp1] in S & ev[Exp2] in S

[Ev21]   ev[Exp1] in S = Val1, ev[Exp2] in S = Val2,  Val1 = Val2
         ------------------------------------------------------
                    ev[(Exp1 = Exp2)] in S = true

[Ev22]   ev[Exp1] in S = Val1,  ev[Exp2] in S = Val2,  Val1 ≠ Val2
         ------------------------------------------------------
                    ev[(Exp1 = Exp2)] in S = false

[Ev23]   ev[(Exp1 ≠ Exp2)] in S = ~ev[(Exp1 = Exp2)] in S

end ASPLE-ds
```

5. SML

SML is a simple stack machine language with commands for

- loading constants and identifiers on the stack,
- looking up values (of identifiers) in a value environment and put them on the stack,
- taking values from the stack to update a value environment,
- replacing two values on top of the stack by another one,
- input and output.

Operators + and * act on integers, & and | on Booleans; operators = and ≠ act on both types.

SML programs may also contain jumps. Commands exist for conditional and unconditional jumps and for defining labels. In case of a conditional jump, the value true or false on top of the stack determines whether or not a jump is executed. Jumps and labels only have a meaning inside a *block* of commands. The scope of a label ranges from the command following the label till the end of the block. Labels are denoted by natural numbers.

In accordance with the specification in TYPOL [CDDHK85] identifiers are used as addresses. So the mapping from ASPLE identifiers to SML addresses in ASPLE-SML will be the identity.

5.1. The syntax of SML

```
module SML-syntax
begin

  exports
  begin
    sorts PROGRAM, COM, CONSTANT, OPER
    context-free  syntax
      {COM ";"}*                          -> PROGRAM
      ldci CONSTANT                       -> COM  -- load constant on stack
      lao ID                              -> COM  -- load identifier on stack
      ldo ID                              -> COM  -- load value of identifier on
                                                     -- stack
      ind                                 -> COM  -- replace identifier on stack
                                                     -- by its value
```

```
      sro ID                                -> COM  -- modify value env with id and
                                                       -- top value from stack
      sto                                   -> COM  -- modify value env with id and
                                                       -- value both from stack
      s-read                                -> COM  -- modify value env with id and
                                                       -- input value
      s-write                               -> COM  -- write top value from stack
      nop                                   -> COM  -- dummy operator
      block "(" {COM ";"}* ")"              -> COM
      ujp NAT                               -> COM  -- unconditional jump
      fjp NAT                               -> COM  -- false jump
      tjp NAT                               -> COM  -- true jump
      lbl NAT                               -> COM  -- label
      OPER                                  -> COM  -- operators
      NAT                                   -> CONSTANT
      BOOL                                  -> CONSTANT
      "+"                                   -> OPER
      "*"                                   -> OPER
      "&"                                   -> OPER
      "|"                                   -> OPER
      "="                                   -> OPER
      "≠"                                   -> OPER
  end

  imports
    Bool-con, Nat-con, Identifiers

end SML-syntax
```

5.2. The dynamic semantics of SML

Three auxiliary notions are needed in the specification of the dynamic semantics of SML: label environments, stacks and states.

5.2.1. Label environments

Label-environments associate labels with commands in an SML block. Note that the function `is-label` has not been defined as partial. So the term `is-label (Com)` in the condition of L3 is defined for every SML command, and hence can be compared to `true` (see Section 3.1).

```
module Label-environments
begin

  exports
  begin
    context-free  syntax
      cons-env "("{COM ";"}* ")"          -> LENV
      is-label "(" COM ")"                -> BOOL
  end

  imports
    Booleans,
    Tables
      Keys bound by
           sorts
                KEY -> NAT to Nat-con
      Entries bound by
           sorts
                ENTRY -> PROGRAM to SML-syntax
      renamed  by
           sorts
                TABLE => LENV
                PAIR => LPAIR
```

```
        functions
             table => lenv
    end  renaming

  variables
    Com                  -> COM              Coms                -> {COM ";"}*
    Nat                  -> NAT              Le, Le1             -> LENV

  equations

  [L1]     cons-env () = lenv ()

  [L2]     cons-env (Coms) = Le1,  Nat is in Le1 = false
           ---------------------------------------------
           cons-env (lbl Nat; Coms) = add (Nat: Coms) to Le1

  [L3]     is-label (Com) ≠ true
           ---------------------
           cons-env (Com; Coms) = cons-env (Coms)

  [L4]     is-label (lbl Nat) = true

end Label-environments
```

5.2.2. Evaluation stack

A stack of values will be maintained during the execution of an SML program.

```
module Stack
begin

  parameters
    Items
    begin
      sorts ITEM
    end Items

  exports
  begin
    sorts STACK
    context-free  syntax
      stack "(" {ITEM ","}* ")"          -> STACK
  end

end Stack

module Evaluation-stack
begin

  imports
    Stack
      Items bound by
           sorts
                ITEM -> VAL to Values
      renamed  by
           sorts
                STACK => EVSTACK
      end  renaming

end Evaluation-stack
```

5.2.3. SML-states

SML-states completely describe the execution state of an SML program.

```
module SML-States
begin

  exports
  begin
    sorts STATE, INPUT, OUTPUT
    context-free  syntax
      "<"VALENV "," EVSTACK "," INPUT "," OUTPUT "," LENV">"        -> STATE
      input "(" {IO-VAL ","}* ")"                                   -> INPUT
      output "(" {IO-VAL ","}* ")"                                  -> OUTPUT
  end
  imports
    Value-environments, Evaluation-stack, Label-environments

end SML-States
```

5.2.4. SML-ds

The evaluation of SML commands for stack manipulation, updating the value environment, performing input and output is straightforward (Es2-13).

Evaluating a block of commands (Es14) amounts to constructing a (new) label environment and evaluating the commands one by one using this label environment. At the end of the block the new label environment is thrown away and the original one is used to evaluate the commands following the block. So in case of nested blocks, commands are evaluated in the context of the smallest block they are part of.

Equations Es15-19 describe the evaluation of jumps. If a jump has to be executed the label it refers to is looked up in the label environment and the statements associated with the label are evaluated.

An operator is applied to the two values on top of the stack. The result replaces those values on the stack (Es21-29).

```
module SML-ds
begin

  exports
  begin
    context-free  syntax
      es "[" PROGRAM "]" with INPUT                 -> OUTPUT     {partial}
      es "[" {COM ";"}* "]" in STATE                -> STATE      {partial}
      es "[" OPER VAL "," VAL "]"                   -> VAL        {partial}
  end

  imports
    SML-syntax, SML-States, Naturals

  variables
    Id                  -> ID              C                   -> CONSTANT
    Com                 -> COM             Coms, Coms1         -> {COM ";"}*
    Nat, Nat1, Nat2     -> NAT             Bool, Bool1, Bool2  -> BOOL
    Op                  -> OPER
    S, S1, S2           -> STATE           E, E1               -> VALENV
    I, I1, I2           -> INPUT           O, O1, O2           -> OUTPUT
    Val, Val1, Val2     -> VAL             Vals,Vals1          -> {VAL ","}*
    K, K1               -> EVSTACK         Le, Le1             -> LENV

  equations

-- evaluation of programs

  [Es1]     es[Coms] in <valenv(),stack(),I,output(),lenv()> = <E,K,I1,O,lenv()>
            ------------------------------------------------------------------
                                  es[Coms] with I = O

-- evaluation of commands
```

[Es2] es[] in S and Le = S

[Es3] es[ldci Nat; Coms] in <E, stack(Vals), I, O, Le> =
es[Coms] in <E, stack(Nat, Vals), I, O, Le>

[Es4] es[ldci Bool; Coms] in <E, stack(Vals), I, O, Le> =
es[Coms] in <E, stack(Bool, Vals), I, O, Le>

[Es5] es[lao Id; Coms] in <E, stack(Vals), I, O, Le> =
es[Coms] in <E, stack(Id, Vals), I, O, Le>

[Es6] $\dfrac{\text{lookup Id in E = Val}}{\text{es[ldo Id; Coms] in <E, stack(Vals), I, O, Le> = es[Coms] in <E, stack(Val, Vals), I, O, Le>}}$

[Es7] $\dfrac{\text{lookup Id in E = Val}}{\text{es[ind, Coms] in <E, stack(Id,Vals), I, O, Le> = es[Coms] in <E, stack(Val,Vals) , I, O, Le>}}$

[Es8] es[sro Id; Coms] in <E, stack(Val,Vals), I, O, Le> =
es[Coms in <modify (Id: Val) in E ,stack(Vals), I, O, Le>

[Es9] es[sto; Coms] in <E, stack(Val, Id, Vals), I, O, Le> =
es[Coms] in <modify (Id: Val) in E , stack(Vals), I, O, Le>

[Es10] es[s-read; Coms] in <E, stack(Id,Vals), input(Nat,Vals1), O, Le> =
es[Coms] in <modify (Id: Nat) in E, stack(Vals), input(Vals1), O, Le>

[Es11] es[s-read; Coms] in <E, stack(Id,Vals), input(Bool,Vals1), O, Le> =
es[Coms] in <modify (Id: Bool) in E, stack(Vals), input(Vals1), O, Le>

[Es12] es[s-write; Coms] in <E1, stack(Val, Vals), I, output(Vals1), Le> =
es[Coms] in <E1, stack(Vals), I, output(Vals1, Val), Le>

[Es13] es[nop; Coms] in S = es[Coms] in S

[Es14] cons-env (Coms) = Le1,
$\dfrac{\text{es[Coms] in <E, K, I, O, Le1> = <E1,K1,I1,O1, Le1>}}{\text{es[block(Coms); Coms1] in <E, K, I, O, Le> = es[Coms1] in <E1, K1, I1, O1, Le>}}$

[Es15] $\dfrac{\text{lookup Nat in Le = Coms1}}{\text{es[ujp Nat; Coms] in <E, K, I, O, Le> = es[Coms1] in <E, K, I, O, Le>}}$

[Es16] es[fjp Nat; Coms] in <E,stack(true,Vals), I, O, Le> =
es[Coms] in <E,stack(Vals), I, O, Le>

[Es17] $\dfrac{\text{lookup Nat in Le = Coms1}}{\text{es[fjp Nat; Coms] in <E, stack(false,Vals), I, O, Le> = es[Coms1] in <E, stack(Vals), I, O, Le>}}$

[Es18] $\dfrac{\text{lookup Nat in Le = Coms1}}{\text{es[tjp Nat; Coms] in <E, stack(true,Vals), I, O, Le> = es[Coms1] in <E,stack(Vals), I, O, Le>}}$

[Es19] es[tjp Nat; Coms] in <E, stack(false,Vals), I, O, Le> =
es[Coms] in <E,stack(Vals), I, O, Le>

[Es20] es[lbl Nat; Coms] in S = es[Coms] in S

[Es21] $\dfrac{\text{es[Op Val1, Val2] = Val}}{\text{es[Op; Coms] in <E, stack(Val1,Val2, Vals), I,O, Le> =}}$

```
          es[Coms] in <E, stack(Val, Vals), I, O, Le>

-- evaluation of operators

  [Es22]   es[+ Nat1, Nat2] = Nat1 + Nat2

  [Es23]   es[* Nat1, Nat2] = Nat1 * Nat2

  [Es24]   es[& Bool1, Bool2] = Bool1 & Bool2

  [Es25]   es[| Bool1, Bool2] = Bool1 | Bool2

  [Es26]   Val1 = Val2
           ------------------------
           es[= Val1, Val2] = true

  [Es27]   Val1 ≠ Val2
           ------------------------
           es[= Val1, Val2] = false

  [Es28]   Val1 = Val2
           ------------------------
           es[≠ Val1, Val2] = false

  [Es29]   Val1 ≠ Val2
           ------------------------
           es[≠ Val1, Val2] = true

end SML-ds
```

6. Compilation of ASPLE to SML

The compilation from ASPLE to SML is defined in the module ASPLE-SML.

Besides functions `cp` for compilation also a function `exe` on ASPLE programs is specified. `Exe` first compiles an ASPLE program to an SML program and a value environment, then the SML program is evaluated (C1).

An ASPLE program is translated to an annotated ASPLE program and is thus typechecked before it is compiled. Only the statements in an ASPLE program are translated to SML commands (C2). We consider the compilation of expressions first.

C13 and 14 handle the compilation of dereferenced identifiers. It may seem strange not to have the same translation for all the deref's but this is because the command `ldo Id` is equivalent to `lao Id` followed by `ind`. For instance the compilation of `deref deref w` from example 6 is `ldo w; ind`, which may be evaluated to:

Put the value of `w`, `v` in example 6, on the stack. Replace the identifier `v` on top of the stack by its value `u`.

Note that the SML commands that correspond to a dereferenced identifier x with k 'deref's' can only be evaluated if a chain exists from x to some y with n(y) = n(x) - k.

C15-20 are simple. Remember that the operator takes as arguments the two top values from the stack and returns the result. Since identifiers in expressions are dereferenced to their primitive modes C13 and C14 are used to compile expressions. This implies that **R2** is checked when the SML commands are evaluated.

C13 and C14 are used again in the compilation of dereferenced identifiers in assignment, input and output statements. Upon evaluation of the resulting SML commands **R4b**, **R5b** and **R6** are checked.

In SML the read command cannot be typed so the mode of the given input value cannot be compared to the mode of the identifier. This means **R5a** cannot be checked for compiled ASPLE programs.

In the compilation of if and while statements jumps, and as a consequence blocks and labels are needed (C7-9). The result of compiling the if statement in the factorial program (example 1) is shown in example 8.

annotated statements

```
if
(deref x ≠ 0)
then
  while

        (deref z ≠ deref x) do

        z := deref z + 1;
        y := deref y + deref z

  end
fi
```

compiled to

```
block(
      ldo x; ldci 0 ; ≠;
      fjp 1;
      block(
            lbl 1;
            ldo z; ldo x; ≠;
            fjp 2;
            ldo z; ldci 1; +; sro z;
            ldo y; ldo z; *; sro y;
            ujp 1;
            lbl 2);
      lbl 1)
```

example 8

Note the double occurrence of `lbl 1`. The first one is in the inner block and the jump `ujp 1` refers to it. The second one is in the outer block and the jump `fjp 1` refers to this one.

```
module ASPLE-SML
begin

  exports
  begin
    context-free  syntax
    exe "[" ASPLE-PROGRAM "]" with INPUT -> OUTPUT                {partial}
    cp "[" ASPLE-PROGRAM "]"             -> SML-PROGRAM           {partial}
    cp "[" {STM ";"}* "]"                -> SML-PROGRAM           {partial}
    cp "[" EXP "]"                       -> SML-PROGRAM           {partial}
  end

  imports
    ASPLE-stat,
    SML-ds
      renamed  by
          sorts
              PROGRAM => SML-PROGRAM
      end  renaming

  variables
    Decls               -> {DECL ";"}*      Var                  -> VAR
    Stm                 -> STM              Stms, Stms1, Stms2   -> STMS
    Mode                -> MODE             Exp,Exp1,Exp2        -> EXP
    Nat                 -> NAT              Bool                 -> BOOL
    Com                 -> COM              Coms, Coms1, Coms2   -> {COM ";"}+
    Op                  -> OPER
    S, S1, S2           -> STATE            K, K1, K2            -> EVSTACK
    I, I1, I2           -> INPUT            O, O1, O2            -> OUTPUT
    E, E1, E2           -> VALENV           Val, Val1, Val2      -> VAL
    Le, Le1, Le2        -> LENV

equations

-- execution of ASPLE programs

  [C1]      cp[begin Decls ; Stms end] = Coms
            ---------------------------------
            exe[begin Decls ; Stms end] with I = es[Coms] with I

-- compilation of ASPLE programs

  [C2]      tr[begin Decls ; Stms end] = begin Decls; Stms1 end,  cp[Stms1] = Coms
            ---------------------------------------------------------------------
                              cp[begin Decls ; Stms end] = Coms
```

```
-- compilation of annotated statements

  [C3]     cp[Stm; Stms] = cp[Stm]; cp[Stms]

  [C4]     cp[Id := Exp] = cp[Exp]; sro Id

  [C5]     cp[tinput Var : Mode] = cp[Var]; s-read

  [C6]     cp[output Exp] = cp[Exp]; s-write

  [C7]     cp[if Exp then Stms1 fi] =
           block (cp[Exp]; fjp 1; cp[Stms1]; lbl 1)

  [C8]     cp[if Exp then Stms1 else Stms2 fi] =
           block(cp[Exp]; fjp 1; cp[Stms1]; ujp2; lbl 1; cp[Stms2]; lbl 2)

  [C9]     cp[while Exp do Stms1 end] =
           block (lbl 1; cp[Exp]; fjp 2; cp[Stms1]; ujp 1; lbl 2)

-- compilation of expressions

  [C10]    cp[Bool] = ldci Bool

  [C11]    cp[Nat] = ldci Nat

  [C12]    cp[Id] = lao Id

  [C13]    cp[deref Id] = ldo Id

  [C14]    cp[deref deref Var] = cp[deref Var]; ind

  [C15]    cp[Exp1 + Exp2] = cp[Exp1]; cp[Exp2]; +

  [C16]    cp[(Exp1 ≠ Exp2)] = cp[Exp1]; cp[Exp2]; ≠

  [C17]    cp[Exp1 & Exp2] = cp[Exp1]; cp[Exp2]; &

  [C18]    cp[Exp1 | Exp2] = cp[Exp1]; cp[Exp2]; |

  [C19]    cp[(Exp1 = Exp2)] = cp[Exp1]; cp[Exp2]; =

  [C20]    cp[(Exp1 ≠ Exp2)] = cp[Exp1]; cp[Exp2]; ≠

end ASPLE-SML
```

7. Conclusions

We have given a complete specification of ASPLE in ASF+SDF: the syntax, the typechecker, a translation to annotated programs, the dynamic semantics and the compilation to SML. This problem has also been specified in Typol [CDDHK85]. Comparing the two specifications we can observe the following.

In Typol relations and inference rules are used where ASF+SDF has functions and conditional equations. The definition of the syntax of functions in ASF+SDF is given in the context-free syntax section of modules. Predicates in the Typol specification, for instance -> and :, are predefined. Their syntax is not defined explicitly but must be deduced from their usage in the inference rules. Other minor differences between the two specifications concern the modular structure and the declaration of variables.

Specifications in ASF+SDF consist of one type of modules only. A module can import other modules by using the import section or by binding parameters to sorts from other modules. The Typol specification consists of two types of modules: "abstract syntax" and "programs". In programs "sets", sets of inference rules, act as a kind of auxiliary modules. Programs can import abstract syntax modules using "Use", other programs using "Import", and sets from other programs using "Import from".

In ASF+SDF long lists of variables have to be declared in some modules. (Note that [HHKR] presents a method for abbreviating such lists). In Typol variables need not be declared when their type can be deduced from the context.

In both formalisms one can specify the aspects of ASPLE we wanted to specify.

Acknowledgements

I wish to thank Paul Hendriks for answering all questions concerning ASF and the ASF-system, and for his useful remarks on the specification. I also wish to thank Paul Klint for commenting on preliminary versions of this paper.

References

[BHK85] J.A. Bergstra, J. Heering, and P. Klint, "Algebraic definition of a simple programming language," Report CS-R8504, Centre for Mathematics and Computer Science, Amsterdam (1985).

[BHK87] J.A. Bergstra, J. Heering, and P. Klint, "ASF – an algebraic specification formalism," Report CS-R8705, Centre for Mathematics and Computer Science, Amsterdam (1987).

[BML76] G.V. Bochmann, H.F. Ledgard, and M.Marcotty, "A Sampler of Formal Definitions," *Computing Surveys*, vol. 8, no 2, pp. 191–276, (1976).

[CDDHK85] D. Clément, J. Despeyroux, T. Despeyroux, L. Hascoet, and G. Kahn, "Specification in Natural Semantics," in Generation of interactive programming environments – GIPE Intermediate report, Report CS-8620, pp. D4.A2, Centre for Mathematics and Computer Science, Amsterdam (1986)

[CU73] J. Cleveland, and R. Uzgalis "What every programmer should know about grammar," Department of Computer Science, University of California, Los Angeles (1973).

[HHKR] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers, "The syntax definition formalism SDF – reference manual," Centre for Mathematics and Computer Science, Amsterdam, to appear.

[HK86] J. Heering, and P. Klint, "A syntax definition formalism," pp. 619-630 in *ESPRIT '86 -- Results and Achievements*, Directorate General XIII (Editors), North-Holland, Amsterdam (1987), also as Report CS-R8633, Centre for Mathematics and Computer Science, Amsterdam (1986).

[HK88] J. Heering, and P. Klint, "Towards shorter algebraic specifications: a simple language definition and its compilation to Prolog," pp. 365-379 in *ESPRIT '88 - Putting the Technology to Use* 1, Directorate General XIII (Editors), North-Holland, Amsterdam (1988), also as Report CS-R8814, Centre for Mathematics and Computer Science, Amsterdam (1988).

[Hen87] P.R.H. Hendriks, "Typechecking mini-ML: an algebraic specification with user-defined syntax," Report CS-R8737, Centre for Mathematics and Computer Science, Amsterdam, (1987), Extended abstract in: *Conference Proceedings of Computing Science in the Netherlands*, pp. 21-38, SION (1987).

[Hen88] P.R.H. Hendriks, "ASF system user's guide," Report CS-R8823, Centre for Mathematics and Computer Science, Amsterdam, (1988), Extended abstract in: *Conference Proceedings of Computing Science in the Netherlands* 1, pp. 83-94, SION (1988).

[Hen] P.R.H. Hendriks, "Lists and associative functions in algebraic specifications - semantics and implementation," Centre for Mathematics and Computer Science, Amsterdam, to appear.

[Kap87] S. Kaplan, "Positive/negative conditional rewriting,"Technical report 87–10, Leibniz Center for Research in Computer Science, Hebrew University, Jerusalem (1987).