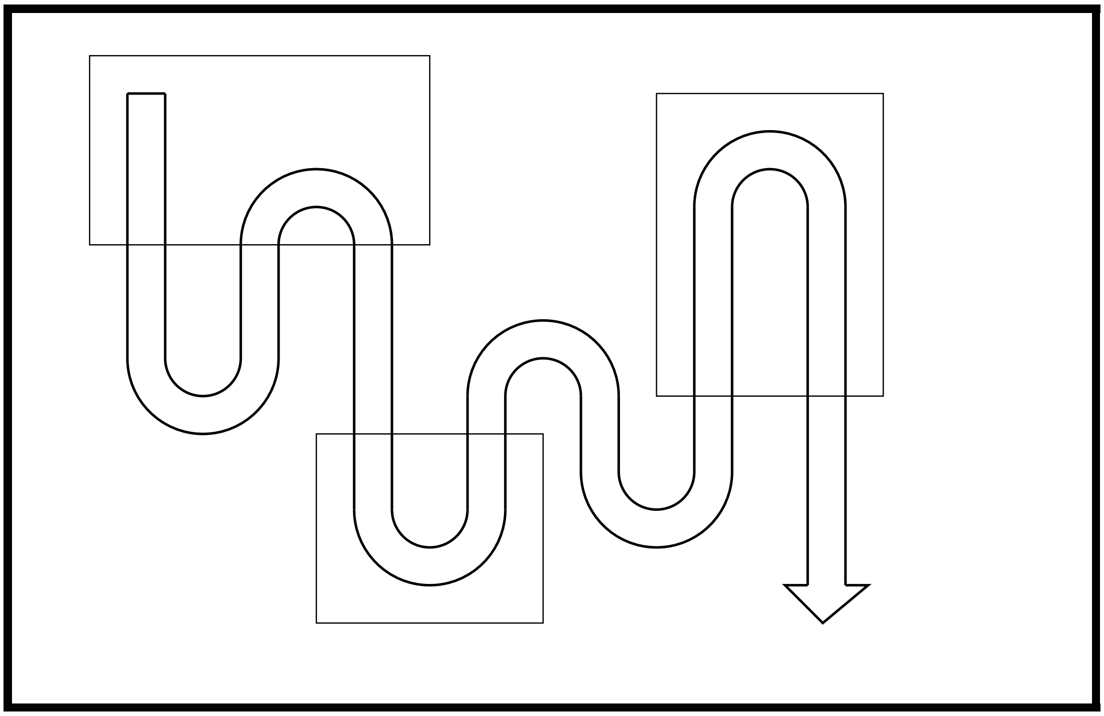
University of Amsterdam
Programming Research Group



Annotated Algebraic Specification of the Syntax and Semantics of the Programming Language Alma-0

Jacob Brunekreef

Report P9803 april 1998

University of Amsterdam

Department of Computer Science

Programming Research Group

Annotated Algebraic Specification of the Syntax and Semantics of the Programming Language Alma-0

Jacob Brunekreef

Report P9803 april 1998

J.J. Brunekreef

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7588
e-mail: jacob@wins.uva.nl

Universiteit van Amsterdam, 1998

Annotated algebraic specification of the syntax and semantics of the programming language Alma-0.

Jacob Brunekreef

Programming Research Group, University of Amsterdam

Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

e-mail: jacob@wins.uva.nl

Abstract

The syntax and semantics of the programming language Alma-0 have been specified in the ASF+SDF formalism. This report contains the interesting parts of the specification, embedded in text explaining its structure, various technical details and the choices that have been made during the specification process.

The report also contains a short introduction to the language Alma-0, including some examples of meaningful Alma-0-programs.

1 The programming language Alma-0

1.1 Introduction

Alma-0 combines the advantages of logic and imperative programming in order to deal in a natural way with algorithmic problems that involve search. Alma-0 is an extension of a subset of the imperative programming language Modula-2. In this report we present an executable formal specification of the semantics of Alma-0.

In this section we briefly describe the language Alma-0 and give some examples of Alma-0-programs. The remainder of this report deals with the syntax and semantics of Alma-0, specified in the algebraic formalism ASF+SDF of Klint ([Kli93]). In section 2 an overview of the specification is given. The sections that follow focus on the various parts of the specification: basic datatypes (section 3), the definition of the syntax of Alma-0 (section 4), the state of the virtual machine on which programs are executed (section 5) and the semantics of the various language constructs (section 6). In a final section some concluding remarks are made.

The language Alma-0 has been introduced by Apt & Schaerf ([AS97]). It is described in more detail in Apt et al. ([ABPS97]). This paper also reports about a compiler for Alma-0. The compiler is described in more detail by Partington ([Par97]). Information about Alma-0 can also be found on the Web at `http://www.cwi.nl/alma/`.

1.2 Alma-0

Alma-0 is an imperative language that supports backtracking. A program consists of a sequence of statements. The *computation* of a program consist of the execution of the statement sequence. The execution of a single statement can be successful or fail. In the latter case, backtracking is tried. This means that program execution resumes at a point that has been 'stored' during the execution of the preceding part of the program. Various Alma-0-statements create such *choice points* as a side effect of their execution. Choice points are stored on a stack. As long as this stack is not empty, backtracking is possible. The computation of a program succeeds if, with or without backtracking, the execution of all statements of a sequence succeeds. Otherwise, the computation fails.

Alma-0 is based on a subset of Modula-2. The following features of Modula-2 have *not* been included in the current version of Alma-0:

- The CARDINAL type, sets, variant records, open array parameters, procedure types, pointer types.
- The CASE, WITH, LOOP and EXIT statements.
- Nested procedures.
- Modules, and therefore the EXPORT and IMPORT declarations.

As will be explained in section 4, the language that is specified in this report is subjected to some additional restrictions.

Alma-0 offers nine extensions of Modula-2 that introduce a declarative approach to programming. They are briefly described in the following list.

1. **BES**: Add boolean expressions to statements.
   The expression `x > 2` is a valid statement. If the result of the expression is `true`, the execution of the program continues. If the expression evaluates to `false`, the execution fails and backtracking is tried. If the stack of choice points is empty, the execution of the complete program fails.

2. **SBE**: Add statement sequences to boolean expressions.
   In the statement `IF (FOR i:=1 TO N-1 DO A[i] < A[i+1] END) THEN ...` the FOR-statement is a valid boolean expression. The boolean expression evaluates to `true` if the computation of the statement succeeds. Otherwise, the result is `false`.

3. **EQ**: Generalize equality.
   The relational operator '`=`' can be used for testing the equality of two expressions that do not contain uninitialized variables. However, if one expression is a single uninitialized variable, the value of the other expression is assigned to that variable (and the equality test succeeds).

4. **KNOWN**: Test on initialization of a variable.
   The `KNOWN` relation tests whether a variable of a simple type is initialized.

5. **ORELSE**: Alternative statement sequences.
   The statement `EITHER S ORELSE T END` introduces two alternative statement sequences, `S` and `T`. On execution of the `ORELSE` statement the statement sequence `S` is executed. The statement sequence `T` is stored in a choice point for backtracking purposes.

6. **SOME**: Alternative statement sequences.
   The statement `SOME i:=`e_1 `TO` e_2 `DO S END` introduces $e_2 - e_1 + 1$ alternative statement sequences. First, `S` is executed with the variable `i` set to e_1. On backtracking, `S` is executed with `i` set to $e_1 + 1$, etc., until `i` equals e_2. If $e_2 < e_1$, then the whole statement is skipped.

7. **COMMIT**: Restricted backtracking.
   The statement `COMMIT S END` prohibits backtracking on statements of the sequence `S`, once the execution of `S` has succeeded. After the execution of `S`, all choice points created by `S` are deleted.

8. **FORALL**: Exhaustive backtracking.
   The statement `FORALL S DO T END` repeatedly executes the statement sequence `S;T` for all possible executions of `S`. So, after each execution of `S;T` backtracking on `S` is tried. This means that variable bindings due to a previous execution of `S` are undone. However, variable bindings due to a previous execution of `T` are never undone.

   If the computation of `S` fails, the FORALL statement is skipped. However, if for one the successful computations of `S` the computation of `T` fails, the computation of the FORALL statement fails.

9. **MIX**: New parameter mechanism.
   In a procedure declaration a MIX parameter behaves like a VAR parameter if the actual parameter is a variable, otherwise it behaves like a value parameter.

The following program fragments illustrate the use of the various Alma-0 constructs. More examples can be found in [ABPS97].

I. Pattern matching.
The following program counts the number of occurrences of a pattern `p` in a string `s`. Execution of the procedure `StringMatch` succeeds if there exists a match of `p` and a substring of `s`. With the `FORALL` statement, *all* occurrences of `p` in `s` are counted: after each execution of the statement `count := count+1` backtracking is tried on the procedure `StringMatch`. Within this procedure choice points are created by the `SOME` statement.

```
PROCEDURE StringMatch(p,s : String);
  VAR i,j : Integer;
  BEGIN
    SOME i:=0 to N-M DO
      FOR j:=0 TO M-1 DO
        s[i+j] = p[j]
      END
    END
  END StringMatch;

VAR count : Integer;

BEGIN
  count:=0;
  FORALL
    StringMatch(p,s)
  DO
    count := count+1
  END
END
```

Remark: backtracking fails when no more choice points are left. It is clear that the program only behaves correct when the `FORALL S DO T END` statement succeeds on a (final) failure of `S`.

II. Eight queens.
The well-known eight queens problem is solved in the following program.

```
PROCEDURE Queens(VAR x: ARRAY OF INTEGER);
  VAR i,column,row : INTEGER;
  BEGIN
    FOR column:=1 TO 8 DO
      SOME row:=1 TO 8 DO
```

```
        FOR i:=1 TO column-1 DO
          x[i] # row;
          x[i] # row+column-i;
          x[i] # row+i-column
        END;
        x[column] = row
    END
  END
 END Queens;

VAR b : ARRAY[1..8] OF INTEGER;

BEGIN
  Queens(b)
END
```

In the procedure `Queens` a value is assigned to an element of the array `x` using the Alma-0 BES and EQ extensions. This makes it possible to use the procedure in various settings:

1. An arbitrary solution can be found by calling the procedure `Queens` with an uninitialized variable `b`.
2. A particular solution can be checked. If, before calling the procedure `Queens`, the variable `b` is (partially) initialized with a particular solution, the procedure `Queens` can be used to complete the solution. So,

   ```
   b[1] = 4;
   b[4] = 8;
   Queens(b);
   ```

   produces a solution with two queens on a fixed position.
3. The number of possible solutions can be counted by the statement sequence

   ```
   count := 0;
   FORALL
     Queens(b)
   DO
     count:=count+1
   END;
   ```

   with `b` uninitialized.

III. Lexical next permutation.
The following program fragment computes the 'lexical next permutation' of a given sequence of numbers (or characters).

```
PROCEDURE Permutation(VAR in, out : Vector);
  VAR pi : Vector;
      i,j : Integer;
  BEGIN
    FOR i:=1 TO 4 DO
      SOME j:=1 TO 4 DO
         pi[j] = i
      END
```

```
    END;
    FOR i:=1 TO 4 DO
      out[i] = in[pi[i]]
    END
  END Permutation;

 PROCEDURE Lex(a,b : Vector);
  VAR i: Integer;
  BEGIN
    NOT (FOR i:=1 TO 4 DO a[i] = b[i] END);
    a[i] < b[i]
  END Lex;

 PROCEDURE Between(a,b : Vector);
  VAR c : Vector;
  BEGIN
    Permutation(a,c);
    Lex(a,c);
    Lex(c,b)
  END Between;

 PROCEDURE Next(VAR a,b : Vector);
  BEGIN
    Permutation(a,b);
    Lex(a,b);
    NOT (Between(a,b))
  END Next;

 BEGIN
   Next(x,y)
 END
```

The procedure `Permutation` computes a permutation of the sequence `in`. On backtracking, this procedure produces *all* permutations of the given sequence. The procedure `Lex` tests whether or not the first parameter lexically precedes the second parameter. If this is not the case, the execution of the procedure fails. The procedure `Between` searches for a permutation of a given sequence `a` with a lexical ordering between `a` and a given sequence `b`. The execution of this procedure fails when no such sequence can be found. Finally, the procedure `Next` searches for a permutation of a sequence `a` with a lexical ordering after `a`, such that no other permutations of `a` exist with an ordering between `a` and `b`. With `x` an initialized sequence of characters (numbers), the procedure call `Next(x,y)` produces the lexical next permutation of `x`, assigned to the variable `y`.

As in the previous example, this program fragment can be used in various settings. E.g., the procedure call `Next(y,x)` with `x` initialized and `y` not, produces in `y` the lexical *preceding* permutation of `x`.

2 Specification outline

The syntax and semantics of Alma-0 are specified in the executable algebraic ASF+SDF formalism of Klint ([Kli93]). ASF+SDF encourages a modular specification style. In this section we give an overview of the specification by presenting the modular structure. Furthermore, we discuss some general principles that have been followed while designing the specification.

The specification consists of fourteen modules. Five of them can be called 'basic' modules in the sense that they are non-specific for the Alma-0 specification. These modules deal with simple layout properties, boolean constants, integer constants and the usual arithmetic on this kind of constants. They are copied from Visser ([Vis96]). In section 3 we present the visible part of the signature of the modules `BoolCon`, `Booleans`, `IntCon` and `Integers`.

The syntax of Alma-0 is specified in four modules: `BasicSyntax`, `ExpressionSyntax`, `StatementSyntax` and `BlockSyntax`. In section 4 these modules are presented in full detail.

The semantics of Alma-0 is specified in three large modules: `ExpressionSemantics`, `StatementSemantics` and `BlockSemantics`. These modules are presented and explained in section 6.

In the semantics specification two constructs are needed: a program state (environment) with variable bindings and a stack of choice points. These constructs are specified in two modules, `Environment` and `ChoicePoints`, presented in section 5.

Figure 1 shows the import-graph of the specification.

While writing the specification some general principles have been applied.

1. Copy the specification of basic datatypes (integers, booleans) from elsewhere. As mentioned before, the modules concerned are reused from the work of Visser ([Vis96]).

2. No 'generic' specification of datatypes for a stack of choice points and the machine state (environment). As will be showed in section 5, these constructs are integrated in the Alma-0-specification, without generic functions like 'push' and 'pop', but with domain-specific functions that operate on e.g. the stack elements.

3. No error-handling is specified. Most notably, type errors are not handled. E.g., the evaluation of the expression `3+true` will not result in an error message, but will lead to a more or less incomprehensible term.

4. Efficiency has not been a primary goal while writing the specification. Comprehensibility has been. In order to make things clear, sometimes a specification style has been chosen that is known to be inefficient: equations with many conditions, functions on lists that apply pattern matching.

3 Basic modules

In order to make the modules in the other sections more comprehensible, in this section we give the exported signature of four basic modules. The module `Layout` is not shown at all.

3.1 BoolCon

imports Layout
exports
  sorts BoolCon
  context-free syntax
    "true" $\rightarrow$ BoolCon
    "false" $\rightarrow$ BoolCon
  variables
    "*bc*"[*0-9'*]$*$ $\rightarrow$ BoolCon

3.2 Booleans

imports BoolCon
exports
  sorts Bool

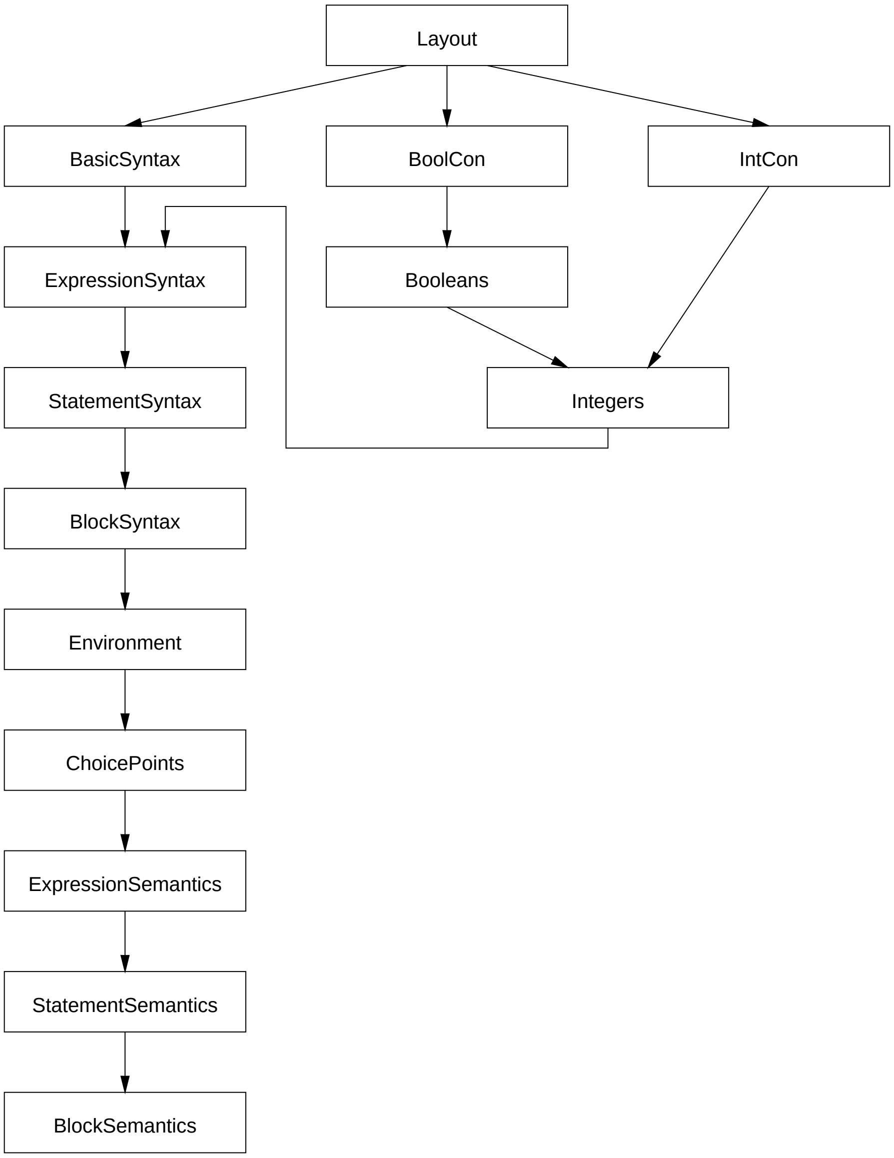


Figure 1: Specification import graph

context-free syntax
BoolCon → Bool
con(Bool) → BoolCon
"∼" Bool → Bool
Bool "∧" Bool → Bool {**assoc**}
Bool "∨" Bool → Bool {**assoc**}
"(" Bool ")" → Bool {**bracket**}
priorities
"∼"Bool → Bool > Bool "∧"Bool → Bool > Bool "∨"Bool → Bool
variables
[*b*][*0-9′*]∗ → Bool

3.3 IntCon

imports Layout
exports
sorts NatCon IntCon
lexical syntax
[0-9]+ → NatCon
context-free syntax
NatCon → IntCon
"+" NatCon → IntCon
"−" NatCon → IntCon
variables
[*n*][*0-9′*]∗ → NatCon
[*z*][*0-9′*]∗ → IntCon

3.4 Integers

imports IntCon Booleans
exports
sorts Int
context-free syntax
IntCon → Int
con(Int) → IntCon
abs(Int) → NatCon

Int "+" Int → Int {**left**}
Int "−" Int → Int {**left**}
Int "∗" Int → Int {**left**}
Int "/" Int → Int {**non-assoc**}
"(" Int ")" → Int {**bracket**}

Int ">" Int → Bool
Int "≥" Int → Bool
Int "<" Int → Bool
Int "≤" Int → Bool
priorities

{Int "∗"Int → Int, Int "/"Int → Int} > {**left**: Int "+"Int → Int, Int "−"Int → Int}

4 Syntax modules

The syntax of Alma-0 is based on the syntax of the programming language Modula-2. The syntax specification in this section is based on the BNF definition of the Modula-2 syntax, given by Wirth in [Wir85]. In order to keep this part of the specification convenient, the syntax specification has been spread over four modules, specifying the basic (lexical) language constructs, expressions, statements and 'blocks' (declarations followed by a sequence of program statements).

In section 1, some Modula-2 constructs have been listed that have not (yet) been included in Alma-0. Moreover, in the syntax specification presented in this report, the following constructs have been omitted.

- Real numbers (floating point constants). Only (decimal) integer numbers have been included. The presence or absence of real number arithmetics is not relevant to the Alma-0 features. Therefore we have decided to omit real number specification.
- The REPEAT statement. This statement can simply be expressed in a WHILE statement.
- Procedure return types and therefore the RETURN construct.

4.1 Basic Syntax

In this module the lexical syntax of Alma-0-identifiers and various operators is specified. The set of operators is split in three subsets: relational operators, arithmetical and logical operators with low priority (+, −, OR), arithmetical and logical operators with high priority (∗, /, AND).

Furthermore, variables of the various sorts are declared. In this report we will apply the convention that variable names are built from two or three (meaningful) lower case letters, possibly followed by one or more numbers or primes. A variable name that denotes a list has a postfix '+' for a list with one or more elements and a postfix '∗' for a list with zero or more elements.

module BasicSyntax
imports Layout
exports
  sorts Ident RelOp AddOp MulOp IdentList
  lexical syntax
    [a-zA-Z][a-zA-Z0-9]∗ → Ident
  context-free syntax
    "=" → RelOp
    "#" → RelOp
    "<" → RelOp
    ">" → RelOp
    "≤" → RelOp
    "≥" → RelOp
    "+" → AddOp
    "−" → AddOp
    "OR" → AddOp
    "∗" → MulOp
    "/" → MulOp
    "AND" → MulOp
    {Ident ","}∗ → IdentList
  variables
    id[*0-9′*]∗ → Ident
    id[*0-9′*]∗"+" → {Ident ","}+
    id[*0-9′*]∗"∗" → {Ident ","}∗

ro[0-9′]∗	→ RelOp
ao[0-9′]∗	→ AddOp
mo[0-9′]∗	→ MulOp

4.2 Expression Syntax

The syntax of expressions is specified straightforward: expressions are built from other expressions using the binary operators from the module `BasicSyntax` and some unary operators. In a priority section the relative priority of the various operators is defined. Basic expressions are an integer constant, a boolean constant and a 'designator' construct, denoting a variable (including an array variable and a record variable). In order to avoid a syntactical ambiguity (a parse conflict), the Modula-2 'qualident' construct (defined in BNF notation as `qualident = ident.{"."ident}`) is not included. The designator definition has been changed to `designator = ident{...}` instead of `designator = qualident{...}`. See also [Par97] for this issue. Furthermore, the syntactical definition of the Alma-0 **KNOWN** relation has been added.

module ExpressionSyntax
imports BasicSyntax Integers
exports
sorts Exp ExpList Designator DesCon
context-free syntax

Exp RelOp Exp	→ Exp	
Exp AddOp Exp	→ Exp	{**left**}
Exp MulOp Exp	→ Exp	{**left**}
"+" Exp	→ Exp	
"−" Exp	→ Exp	
"NOT" Exp	→ Exp	
IntCon	→ Exp	
BoolCon	→ Exp	
Designator	→ Exp	
"KNOWN" "(" Designator ")"	→ Exp	
{Exp ","}+	→ ExpList	
Ident DesCon∗	→ Designator	
"." Ident	→ DesCon	
"[" ExpList "]"	→ DesCon	
"(" Exp ")"	→ Exp	{**bracket**}

priorities
Exp MulOp Exp → Exp > Exp AddOp Exp → Exp > Exp RelOp Exp → Exp
variables

ex[0-9′]∗	→ Exp
ex[0-9′]∗"+"	→ {Exp ","}+
dr[0-9′]∗	→ Designator
dc[0-9′]∗	→ DesCon
dc[0-9′]∗"∗"	→ DesCon∗
dc[0-9′]∗"+"	→ DesCon+

4.3 Statement Syntax

In this module the syntax of the Alma-0 program statements is defined. First of all, an expression is defined as a statement, incorporating the Alma-0 **BES** feature. The syntax definition of the assignment statement is self-explaining. The IF statement and the WHILE statement now have a statement sequence as a condition. As an expression is a statement, this does not conflict with the traditional syntax of these statements. As mentioned before, the REPEAT statement is not included. The syntax definition of the Alma-0 statements (OR, SOME, COMMIT, FORALL) is straightforward. The last statement definitions deal with a procedure call, without or with parameters. Notice that 'functions' (procedures with a type) are not incorporated.

The IF statement and the ORELSE statement have an optional 'continuation' part, that can be repeated zero or more times. These continuation parts are specified separately, after the list of statement definitions. A statement sequence is defined as a sequence of zero or more statements, separated by a semicolon. Additional to the definition of expression factors in the previous module, a bracketed statement sequence is defined as a factor, implementing the **SBE** feature. Brackets are used to avoid ambiguities in statements like `x:=true AND y` or `x := true; false`. By requiring brackets around a statement in a factor, the statement `x:=true AND y` denotes an assignment, whereas the assignment statement in the expression `(x:=true) AND y` denotes a statement-as boolean expression. In [Par97], Partington chooses another solution for this problem: all statements, except the assignment statement, are defined as a factor. So, brackets are only needed around an assignment statement. However, we prefer a syntax specification in which statements are identified as such, and not as factors in an expression.

For each sort variables are defined. Deviant from the naming convention for variables (section 4.1), statement sequence variables are named S, T or $S+$. This makes it possible to denote statements like 'FORALL S DO T END' in the 'Alma-0 style', practiced in the literature.

module StatementSyntax
imports ExpressionSyntax
exports
  sorts Stat IfCon ElsIf OrElse OrElseList StSeq
  context-free syntax

Exp	→ Stat	
Designator ":=" Exp	→ Stat	
"`IF`" StSeq "`THEN`" StSeq IfCon "`END`"	→ Stat	
"`WHILE`" StSeq "`DO`" StSeq "`END`"	→ Stat	
"`FOR`" Ident ":=" Exp "`TO`" Exp "`DO`" StSeq "`END`"	→ Stat	
"`EITHER`" StSeq OrElseList "`END`"	→ Stat	
"`SOME`" Ident ":=" Exp "`TO`" Exp "`DO`" StSeq "`END`"	→ Stat	
"`COMMIT`" StSeq "`END`"	→ Stat	
"`FORALL`" StSeq "`DO`" StSeq "`END`"	→ Stat	
Ident	→ Stat	
Ident "(" ExpList ")"	→ Stat	
ElsIf*	→ IfCon	
ElsIf* "`ELSE`" StSeq	→ IfCon	
"`ELSIF`" StSeq "`THEN`" StSeq	→ ElsIf	
OrElse+	→ OrElseList	
"`ORELSE`" StSeq	→ OrElse	
{Stat ";"}*	→ StSeq	
"(" StSeq ")"	→ StSeq	{**bracket**}
"(" StSeq ")"	→ Exp	

variables

st[*0-9′*]∗	→ Stat
[*ST*][*0-9′*]∗	→ {Stat ";"}∗
S [*0-9′*]∗"+"	→ {Stat ";"}+
ic[*0-9′*]∗	→ IfCon
ei[*0-9′*]∗"∗"	→ ElsIf∗
oe[*0-9′*]∗	→ OrElse
oe[*0-9′*]∗"+"	→ OrElse+

4.4 Block Syntax

The last syntax module defines what is called a *block* in Modula-2: a list of zero or more declarations, followed by a a statement sequence surrounded by a `BEGIN-END` pair. A declaration may be a type declaration, a variable declaration or a procedure declaration. A constant declaration is not incorporated. Only simple types, array's and records are specified as valid types. Because of the absence of the 'qualident' construct (see section 4.2), in the definition of a simple type an 'Ident' is used instead. In the procedure declaration the Alma-0 feature of `MIX` parameters is added to the formal parameter section.

module BlockSyntax
imports StatementSyntax
exports
sorts Block Decl TypeDecl VarDecl Type SimpleType ArrayType RecordType ProcDecl ProcHeading FormalPar FPSection FormalType
context-free syntax

Decl∗ "`BEGIN`" StSeq "`END`"	→ Block
"`TYPE`" {TypeDecl ";"}+ ";"	→ Decl
"`VAR`" {VarDecl ";"}+ ";"	→ Decl
ProcDecl ";"	→ Decl
Ident "=" Type	→ TypeDecl
IdentList ":" Type	→ VarDecl
SimpleType	→ Type
ArrayType	→ Type
RecordType	→ Type
Ident	→ SimpleType
"(" IdentList ")"	→ SimpleType
"[" Exp ".." Exp "]"	→ SimpleType
"`ARRAY`" {SimpleType ","}+ "`OF`" Type	→ ArrayType
"`RECORD`" {VarDecl ";"}+ "`END`"	→ RecordType
ProcHeading ";" Block Ident	→ ProcDecl
"`PROCEDURE`" Ident	→ ProcHeading
"`PROCEDURE`" Ident FormalPar	→ ProcHeading
"(" {FPSection ";"}+ ")"	→ FormalPar
IdentList ":" FormalType	→ FPSection
"`VAR`" IdentList ":" FormalType	→ FPSection
"`MIX`" IdentList ":" FormalType	→ FPSection
Ident	→ FormalType
"`ARRAYOF`" Ident	→ FormalType

variables

bl[*0-9′*]∗	→ Block

$dl[0\text{-}9']*$ → Decl
$dl[0\text{-}9']*$"*" → Decl*
$td[0\text{-}9']*$"+" → {TypeDecl ";"}+
$vd[0\text{-}9']*$"+" → {VarDecl ";"}+
$tp[0\text{-}9']*$ → Type
$fp[0\text{-}9']*$ → FormalPar
$fs[0\text{-}9']*$ → FPSection
$fs[0\text{-}9']*$"+" → {FPSection ";"}+
$ft[0\text{-}9']*$ → FormalType

5 Supporting modules for the semantics specification

Before specifying the semantics of Alma-0, we present two supporting modules. The module Environment defines the 'machine state': a set of program variables bound to particular values. This module is an extension of a module written by Visser ([Vis96]). In the module ChoicePoints a choice point is defined, as well as some constructor functions for a stack of choice points.

5.1 Environment

In the module Environment the 'machine state' is defined as a term of the sort 'Env'. An environment is built from a sequence of atomic environments. An atomic environment consists of a mapping of a key to a value. A key can be a variable name, a value can be an integer constant. So, $x \mapsto 5$ is an atomic environment, denoting that the variable x has the integer value 5. For compound variables (array's, records) an identifier is bound not to a single integer or boolean value, but to an 'environment value'. In this environment elements of the compound variable are identified by separate keys. For an array element the key is supposed to be an integer constant, for a record field the key is supposed to be an identifier. So, an array A with three elements, $A[1] = 10$, $A[2] = 20$ and $A[3] = 30$ is stored as the atomic environment $A \mapsto [1 \mapsto 10\ 2 \mapsto 20\ 3 \mapsto 30]$. This explains why an integer constant is specified as a key and an environment is specified as a value.

During a procedure call, a global variable has to be 'occluded' when its name is identical to the name of a formal parameter or a local variable. In the environment specification a variable is occluded by placing the key between braces. We will come back to this in section 6.3.

During the computation of the semantics of a statement sequence a procedure is also stored in the environment. The procedure name serves as a key, a construct with the various elements of a procedure serves as a value. In the case of a procedure without parameters this construct consists of a procedure block (declaration of local variables and procedure statements), in case of a procedure with parameters the list with formal parameters is included in the construct. These two constructs are also declared as values.

In the module also a construct of the sort 'CEnv' is defined. In this construct not a single key, but a *list* of keys is bound to a value. This construct is used in a function that updates the value of a single element of a compound variable (now denoted by a list of keys) in the environment. The list of keys is generated by a function that is specified in the module StatementSemantics, see section 6.2.

The module defines a set of functions that operate on an environment:

- $\mathcal{E}(k^+)$: lookup the value of a variable (denoted by a list of keys k^+) in the environment $\mathcal{E}$. E.g., $[x \mapsto 5](x)$ produces the value 5.

- $\mathcal{E}_1 \rhd \mathcal{E}_2$: destructively update environment $\mathcal{E}_2$ with environment $\mathcal{E}_1$. The bindings in $\mathcal{E}_2$ for variables which have a binding in $\mathcal{E}_1$ are discarded by this operation, e.g., $([x \mapsto v] \rhd \mathcal{E})(x)$ is v and not the value of x in $\mathcal{E}$.

- The update function $\triangleright$ is also defined for a term of the sort CEnv.
- The function remove($\mathcal{E}, k$) removes a variable from an environment.
- The boolean function def($\mathcal{E}, k^+$) tests if a variable is present in an environment. The variable is denoted by a list of keys, so this test can also be used for a single element of a compound variable.
- The boolean function eq(v_1, v_2) tests if two values are equal or not.

Some additional comments are added in the equations section of the module.

module Environment
imports BlockSyntax
exports
  sorts Env AEnv Key Value KeyList CEnv
  context-free syntax

"[" AEnv∗ "]"	→ Env	
Key "↦" Value	→ AEnv	
Ident	→ Key	
IntCon	→ Key	
"{" Key "}"	→ Key	
IntCon	→ Value	
BoolCon	→ Value	
Env	→ Value	
Block	→ Value	
FormalPar "◇" Block	→ Value	
{Key ","}+	→ KeyList	
"(" KeyList "⟼" Value ")"	→ CEnv	
"(" Env ")"	→ Env	{**bracket**}
Env "(" KeyList ")"	→ Value	
Env "▷" Env	→ Env	{**non-assoc**}
CEnv "▷" Env	→ Env	
remove "(" Env "," Key ")"	→ Env	
def "(" Env "," KeyList ")"	→ Bool	
eq "(" Value "," Value ")"	→ Bool	

  variables

"$\mathcal{E}$"*[0-9']*∗	→ Env
"*ae*"*[0-9']*∗	→ AEnv
"*ae*∗"*[0-9']*∗	→ AEnv∗
"*ae*+"*[0-9']*∗	→ AEnv+
"*k*"*[0-9']*∗	→ Key
"*k*+"*[0-9']*∗	→ {Key ","}+
"*v*"*[0-9']*∗	→ Value

equations
Lookup function.

[1] $[k \mapsto v\ ae^*](k)\ = v$
[2] $[k \mapsto v\ ae^*](k') = [ae^*](k')$ **when** $k \neq k'$

In case of a compound variable the function is recursively called.

[3] $[k \mapsto \mathcal{E}\ \ ae^*](k,\, k^+)\ = \mathcal{E}(k^+)$

[4] $[k \mapsto v\ ae^*](k', k^+) = [ae^*](k', k^+)$ **when** $k \neq k'$

Destructive update of one environment by another.

[5] $[] \rhd \mathcal{E} \ = \ \mathcal{E}$

[6] $\mathcal{E} \rhd [] \ = \ \mathcal{E}$

[7] $[ae^+\ ae] \rhd \mathcal{E} \ = \ [ae^+] \rhd ([ae] \rhd \mathcal{E})$

Update of a single variable.

[8] $[k \mapsto v] \rhd [k \mapsto v'\ ae^*] \ = \ [k \mapsto v\ ae^*]$

Recursive equation if $k \neq k'$.

[9] $$\frac{k \neq k',\ [k \mapsto v] \rhd [ae^*] = [ae_2^*]}{[k \mapsto v] \rhd [k' \mapsto v'\ ae^*] \ = \ [k' \mapsto v'\ ae_2^*]}$$

Destructive update of an environment by a single (compound) variable update.
1. Atomic variable: apply the generic update function

[10] $(k \longmapsto v) \rhd \mathcal{E} = [k \mapsto v] \rhd \mathcal{E}$

2. Compound variable: recursively update the environment.
If the variable is already defined, its value (the environment $\mathcal{E}_1$) has to be updated.

[11] $$\frac{\mathrm{def}(\mathcal{E}, k) = \mathrm{true},\ \mathcal{E}(k) = \mathcal{E}_1,\ (k^+ \longmapsto v) \rhd \mathcal{E}_1 = v_1}{(k, k^+ \longmapsto v) \rhd \mathcal{E} \ = \ [k \mapsto v_1] \rhd \mathcal{E}}$$

If the variable is not yet defined, its value is created from an empty environment.

[12] $$\frac{\mathrm{def}(\mathcal{E}, k) = \mathrm{false},\ (k^+ \longmapsto v) \rhd [] = v_1}{(k, k^+ \longmapsto v) \rhd \mathcal{E} \ = \ [k \mapsto v_1] \rhd \mathcal{E}}$$

Remove an element from an environment.

[13] $\mathrm{remove}([ae_1^*\ k \mapsto v\ ae_2^*], k) = [ae_1^*\ ae_2^*]$
[14] $\mathrm{remove}(\mathcal{E}, k) = \mathcal{E}$ **otherwise**

Test if a variable is defined in an environment.

[15] $\mathrm{def}([ae_1^*\ k \mapsto v\ ae_2^*], k) = \mathrm{true}$

In case of a compound variable, the function is recursively called with the remaining keys.

[16] $$\frac{\mathrm{def}(\mathcal{E}, k) = \mathrm{true},\ \mathcal{E}(k) = \mathcal{E}_1}{\mathrm{def}(\mathcal{E}, k, k^+) \ = \ \mathrm{def}(\mathcal{E}_1, k^+)}$$

[17] $\mathrm{def}(\mathcal{E}, k^+) = \mathrm{false}$ **otherwise**

Test on equality of values.

[18] $\mathrm{eq}(v, v) = \mathrm{true}$
[19] $\mathrm{eq}(v_1, v_2) = \mathrm{false}$ **otherwise**

5.2 Choice Points

For the specification of the backtracking facilities of Alma-0 a stack of *choice points* is needed. A choice point consist of a tuple with a statement sequence and an environment. The statement sequence is the sequence that has to be executed when backtracking is initiated. The environment contains the variable bindings that correspond with the program state at the moment backtracking is initiated. Choice points are created by non-deterministic statements like the `ORELSE` statement or the `SOME` statement. During the computation of the semantics of a program choice points are stored on a stack. The module contains two constructor-functions for creating a stack of choice points: the constant '⊥', denoting the empty stack, and the function '⋈', denoting the constructor-function for a non-empty stack. Any other stack function (push, pop, top) is absent, stack operations will be specified directly on terms built with the constructor-functions.

In order to extract a statement sequence or an environment from a choice point tuple, the functions 'fst' and 'snd' are specified.

module ChoicePoints
imports Environment
exports
  sorts ChoicePoint Stack
  context-free syntax
    "≺" StSeq "," Env "≻" → ChoicePoint
    "⊥" → Stack
    ChoicePoint "⋈" Stack → Stack
    fst "(" ChoicePoint ")" → StSeq
    snd "(" ChoicePoint ")" → Env
  variables
    CP [*0-9'*]∗ → ChoicePoint
    [*C*][*0-9'*]∗ → Stack
equations
Selection functions for tuple elements.

[20] $\mathrm{fst}(\prec S, \mathcal{E} \succ) = S$
[21] $\mathrm{snd}(\prec S, \mathcal{E} \succ) = \mathcal{E}$

6 Semantics modules

The specification of the Alma-0 semantics is presented in three modules. These modules follow the same partitioning as the syntax specification. So, we have modules for the semantics of expressions, statements and a block.

6.1 Expression Semantics

The choice has been made to specify the evaluation of only two expression types: boolean and integer. With this selection it is possible to evaluate programs as presented in section 1. Floating point constants were already excluded in the syntax specification, so it is obvious that the evaluation of floating point expression is not included. Also, expressions with usually built-in functions on e.g. characters and strings are not evaluated.

The semantics of an expression is defined with the function 'eval'. This function has two arguments: the expression-to-be-evaluated and the environment in which the expression has to be evaluated. The result is given not in a single value, but in a tuple: besides the value of the expression, also an environment is returned. This is required because in the evaluation of an expression the environment can be modified. Both the EQ extension and the SBE extension of Alma-0 may result in a modification of the environment. E.g., the expression `x=4`, with `x` an uninstantiated variable, results in an update of the environment with the variable binding '$x \mapsto 4$'.

Two help-functions are defined. The boolean function 'known' returns the value false if its argument is an uninitialized variable. For any other expression the function returns the value true. This function is almost the same as the KNOWN relation of Alma-0. The difference is that the domain of this function is the set of expressions, whereas the the KNOWN-construct is defined for 'designators' only.

The index of an array-element can be an expression itself: `A[i]`, `B[2*j+1]`, etc. The help-function 'evaldr' evaluates expressions that are part of a designator. The result is a *list* of environment keys. This list is used in various environment functions, see section 5.1.

In the semantics modules the details concerning the definition of the various functions are explained in text that is interleaved with the equations.

module ExpressionSemantics
imports ChoicePoints
exports
sorts Env-Value
context-free syntax

eval "⟦" Exp "⟧" "(" Env ")"	→ Env-Value
"<" Env "," Value ">"	→ Env-Value
known "(" Exp "," Env ")"	→ Bool
evaldr "{" Designator "}" "(" Env ")"	→ KeyList

equations
Evaluation of a constant expression (integer or boolean): the result is the environment and the value.

[22] $\mathrm{eval}[\![z]\!](\mathcal{E}) \;=\; <\mathcal{E},\, z>$

[23] $\mathrm{eval}[\![bc]\!](\mathcal{E}) \;=\; <\mathcal{E},\, bc>$

Evaluation of a single variable expression: first evaluate the constituing parts of the variable (designator), then lookup its value in the environment.

[24] $$\frac{\mathrm{evaldr}\{id\ dc^*\}(\mathcal{E}) = k^+}{\mathrm{eval}[\![id\ dc^*]\!](\mathcal{E}) \;=\; <\mathcal{E},\, \mathcal{E}(k^+)>}$$

The evaluation of arithmetic expressions is straightforward. The equations follow the definition of the syntax of the arithmetic expressions.

[25] $$\frac{\mathrm{eval}[\![ex_1]\!](\mathcal{E}) = <\mathcal{E}_1,\, z_1>,\ \mathrm{eval}[\![ex_2]\!](\mathcal{E}_1) = <\mathcal{E}_2,\, z_2>}{\mathrm{eval}[\![ex_1 + ex_2]\!](\mathcal{E}) \;=\; <\mathcal{E}_2,\, \mathrm{con}(z_1 + z_2)>}$$

[26] $$\frac{\mathrm{eval}[\![ex_1]\!](\mathcal{E}) = <\mathcal{E}_1,\, z_1>,\ \mathrm{eval}[\![ex_2]\!](\mathcal{E}_1) = <\mathcal{E}_2,\, z_2>}{\mathrm{eval}[\![ex_1 - ex_2]\!](\mathcal{E}) \;=\; <\mathcal{E}_2,\, \mathrm{con}(z_1 - z_2)>}$$

[27] $$\frac{\mathrm{eval}[\![ex_1]\!](\mathcal{E}) = <\mathcal{E}_1,\, z_1>,\ \mathrm{eval}[\![ex_2]\!](\mathcal{E}_1) = <\mathcal{E}_2,\, z_2>}{\mathrm{eval}[\![ex_1 * ex_2]\!](\mathcal{E}) \;=\; <\mathcal{E}_2,\, \mathrm{con}(z_1 * z_2)>}$$

[28] $$\frac{\mathrm{eval}[\![ex_1]\!](\mathcal{E}) = <\mathcal{E}_1,\, z_1>,\ \mathrm{eval}[\![ex_2]\!](\mathcal{E}_1) = <\mathcal{E}_2,\, z_2>}{\mathrm{eval}[\![ex_1\ /\ ex_2]\!](\mathcal{E}) \;=\; <\mathcal{E}_2,\, \mathrm{con}(z_1\ /\ z_2)>}$$

$$[29] \quad \frac{\text{eval}[\![ex]\!](\mathcal{E}) = <\mathcal{E}_1,\, z>}{\text{eval}[\![+\ ex]\!](\mathcal{E}) \ = \ <\mathcal{E}_1, \text{con}(z)>}$$

$$[30] \quad \frac{\text{eval}[\![ex]\!](\mathcal{E}) = <\mathcal{E}_1,\, z>}{\text{eval}[\![-\ ex]\!](\mathcal{E}) \ = \ <\mathcal{E}_1, \text{con}(0 - z)>}$$

The definition of the relational operators.
First we give a definition of the EQ extension of Alma-0: depending on the status of the operands, the operator '=' denotes the conventional equality test or an assignment.
Case 1. Both se1 and se2 are known expressions (not uninitialized variables). We have the common equality test.

$$[31] \quad \frac{\begin{array}{c}\text{known}(ex_1, \mathcal{E}) = \text{true},\ \text{known}(ex_2, \mathcal{E}) = \text{true},\\ \text{eval}[\![ex_1]\!](\mathcal{E}) = <\mathcal{E}_1,\, v_1>,\ \text{eval}[\![ex_2]\!](\mathcal{E}_1) = <\mathcal{E}_2,\, v_2>\end{array}}{\text{eval}[\![ex_1 = ex_2]\!](\mathcal{E}) \ = \ <\mathcal{E}_2, \text{con}(\text{eq}(v_1,\, v_2))>}$$

Case 2. ex1 is an uninitialized variable. Assign the value of ex2 to it and return the value true.

$$[32] \quad \frac{\begin{array}{c}\text{known}(ex_1, \mathcal{E}) = \text{false},\ \ ex_1 = dr,\ \text{evaldr}\{dr\}(\mathcal{E}) = k^+,\\ \text{eval}[\![ex_2]\!](\mathcal{E}) = <\mathcal{E}_1,\, v>,\ \ (k^+ \longmapsto v) \rhd \mathcal{E} = \mathcal{E}_2\end{array}}{\text{eval}[\![ex_1 = ex_2]\!](\mathcal{E}) \ = \ <\mathcal{E}_2, \text{true}>}$$

Case 3. ex2 is an uninitialized variable. Assign the value of ex1 to it.
(We suppose that not both expressions are uninitialized variables.)

$$[33] \quad \frac{\text{known}(ex_2, \mathcal{E}) = \text{false}}{\text{eval}[\![ex_1 = ex_2]\!](\mathcal{E}) \ = \ \text{eval}[\![ex_2 = ex_1]\!](\mathcal{E})}$$

The definition of the other relational operators is straightforward.

$$[34] \quad \frac{\text{eval}[\![ex_1]\!](\mathcal{E}) = <\mathcal{E}_1,\, z_1>,\ \text{eval}[\![ex_2]\!](\mathcal{E}_1) = <\mathcal{E}_2,\, z_2>}{\text{eval}[\![ex_1 > ex_2]\!](\mathcal{E}) \ = \ <\mathcal{E}_2, \text{con}(z_1 > z_2)>}$$

$$[35] \quad \frac{\text{eval}[\![ex_1]\!](\mathcal{E}) = <\mathcal{E}_1,\, z_1>,\ \text{eval}[\![ex_2]\!](\mathcal{E}_1) = <\mathcal{E}_2,\, z_2>}{\text{eval}[\![ex_1 \geq ex_2]\!](\mathcal{E}) \ = \ <\mathcal{E}_2, \text{con}(z_1 \geq z_2)>}$$

$$[36] \quad \frac{\text{eval}[\![ex_1]\!](\mathcal{E}) = <\mathcal{E}_1,\, z_1>,\ \text{eval}[\![ex_2]\!](\mathcal{E}_1) = <\mathcal{E}_2,\, z_2>}{\text{eval}[\![ex_1 < ex_2]\!](\mathcal{E}) \ = \ <\mathcal{E}_2, \text{con}(z_1 < z_2)>}$$

$$[37] \quad \frac{\text{eval}[\![ex_1]\!](\mathcal{E}) = <\mathcal{E}_1,\, z_1>,\ \text{eval}[\![ex_2]\!](\mathcal{E}_1) = <\mathcal{E}_2,\, z_2>}{\text{eval}[\![ex_1 \leq ex_2]\!](\mathcal{E}) \ = \ <\mathcal{E}_2, \text{con}(z_1 \leq z_2)>}$$

$$[38] \quad \frac{\text{eval}[\![ex_1]\!](\mathcal{E}) = <\mathcal{E}_1,\, v_1>,\ \text{eval}[\![ex_2]\!](\mathcal{E}_1) = <\mathcal{E}_2,\, v_2>}{\text{eval}[\![ex_1 \# ex_2]\!](\mathcal{E}) \ = \ <\mathcal{E}_2, \text{con}(\sim \text{eq}(v_1,\, v_2))>}$$

The definition of the boolean operators NOT, OR and AND.

$$[39] \quad \frac{\text{eval}[\![ex]\!](\mathcal{E}) = <\mathcal{E}_1,\, bc>}{\text{eval}[\![\texttt{NOT}\ ex]\!](\mathcal{E}) \ = \ <\mathcal{E}_1, \text{con}(\sim bc)>}$$

$$[40] \quad \frac{\text{eval}[\![ex_1]\!](\mathcal{E}) = <\mathcal{E}_1,\, bc_1>,\ \text{eval}[\![ex_2]\!](\mathcal{E}_1) = <\mathcal{E}_2,\, bc_2>}{\text{eval}[\![ex_1\ \texttt{OR}\ ex_2]\!](\mathcal{E}) \ = \ <\mathcal{E}_2, \text{con}(bc_1 \vee bc_2)>}$$

[41] $$\frac{\text{eval}[\![ex_1]\!](\mathcal{E}) = <\mathcal{E}_1,\, bc_1>,\ \text{eval}[\![ex_2]\!](\mathcal{E}_1) = <\mathcal{E}_2,\, bc_2>}{\text{eval}[\![ex_1 \texttt{ AND } ex_2]\!](\mathcal{E}) \ = \ <\mathcal{E}_2, \text{con}(bc_1 \wedge bc_2)>}$$

The evaluation of the Alma-0 KNOWN language construct is restricted to a single variable (designator). The help-function 'known' is used.

[42] $$\frac{\text{known}(dr, \mathcal{E}) = \text{true}}{\text{eval}[\![\texttt{KNOWN}(dr)]\!](\mathcal{E}) \ = \ <\mathcal{E}, \text{true}>}$$

[43] $\text{eval}[\![\texttt{KNOWN}(dr)]\!](\mathcal{E}) = <\mathcal{E}, \text{false}>$ **otherwise**

The evaluation of a bracketed statement (sequence) as expression (the Alma-0 SBE extension) is defined in the module StatementSemantics.

The boolean function 'known' identifies an uninitialized variable in an environment.

[44] $$\frac{\text{evaldr}\{dr\}(\mathcal{E}) = k^+}{\text{known}(dr, \mathcal{E}) \ = \ \text{def}(\mathcal{E}, k^+)}$$

For every other expression the function returns by default the value true.

[45] $\text{known}(ex, \mathcal{E}) = \text{true}$ **otherwise**

The function 'evaldr' evaluates the constituing parts of a compound variable (designator) and places them in a list of environment keys. A designator is processed from right to left.

[46] $\text{evaldr}\{id\}(\mathcal{E}) = id$

[47] $$\frac{\text{evaldr}\{id\ dc^*\}(\mathcal{E}) = k^+}{\text{evaldr}\{id\ dc^*\ .\ id'\}(\mathcal{E}) \ = \ k^+,\, id}$$

[48] $$\frac{\text{eval}[\![ex]\!](\mathcal{E}) = <\mathcal{E}_1,\, z>,\ \text{evaldr}\{id\ dc^*\}(\mathcal{E}) = k^+}{\text{evaldr}\{id\ dc^*\ [ex]\}(\mathcal{E}) \ = \ k^+,\, z}$$

[49] $$\frac{\text{eval}[\![ex]\!](\mathcal{E}) = <\mathcal{E}_1,\, z>,\ \text{evaldr}\{id\ dc^*\ [ex^+]\}(\mathcal{E}) = k^+}{\text{evaldr}\{id\ dc^*\ [ex^+,\, ex]\}(\mathcal{E}) \ = \ k^+,\, z}$$

6.2 Statement Semantics

The basic construct in the specification of the semantics of statements is a *configuration* triple. The first triple element contains the statement sequence that has to be evaluated, the second element is an environment and the third element is a stack of choice points. The environment and the stack of choice points are the result of the evaluation of preceding statements. The function 'sem' takes a configuration as input argument and produces a new configuration, by evaluating the first statement of the sequence. This implies that the function 'sem' is recursively called –with an updated environment and stack of choice points– until the statement sequence is empty. For each statement the update of the environment and/or the stack of choice points is defined. The evaluation of a statement sequence succeeds if the statement sequence in the resulting configuration is empty. This is specified in the first equation for the function 'sem'. If the computation of a single statement fails, backtracking takes place by popping the top element of the choice point stack. If this stack is empty, the whole computation fails. In that case a configuration is returned with a non-empty statement sequence. The first statement of this sequence shows where the computation has failed.

The function 'sem' is defined recursively on a *sequence* of statements. Another approach would have been to define a semantics function for each statement separately, together with a composition function 'semC' that defines the result for a sequence of more than one statement:

$$\frac{\mathrm{sem}(\ll S, \mathcal{E}, C \gg) = \ll\ , \mathcal{E}_1, C_1 \gg}{\mathrm{semC}(\ll S; T, \mathcal{E}, C \gg) = \mathrm{semC}(\ll T, \mathcal{E}_1, C_1 \gg)}$$

However, for an update of the stack of choice points, the function 'sem' needs the sequence of *all* statements that follow the statement S. As 'sem' has only the statement S as input, this update has to be performed by the composition function 'semC'. We have chosen not to follow this approach.

In the module below two local help-functions are defined: 'semFA' and 'changes', both used in the evaluation of the FORALL statement.

In the equations section first two additional equations for the function 'eval' from the previous module are given. In these equations the SBE extension (statements as boolean expression) is defined. As these equations make use of the function 'sem', they are included in this module.

The details of the definition of the functions are commented in a text that is interleaved with the equations.

module StatementSemantics
imports ExpressionSemantics
exports
  sorts Config
  context-free syntax
    "≪" StSeq "," Env "," Stack "≫" → Config
    sem "(" Config ")" → Config
hiddens
  context-free syntax
    semFA "(" Config ")" → Config
    changes "(" Env "," Env ")" → Env
equations

The SBE extension: two additional equations for the 'eval' function. Depending on the result of the function 'sem' on the statement (sequence), the value true or false is returned. The success of a computation is indicated by an empty statement sequence in the first element of a configuration triple.

[50] $$\frac{\mathrm{sem}(\ll S,\, \mathcal{E},\, \perp \gg) = \ll\ ,\, \mathcal{E}_1,\, C \gg}{\mathrm{eval}[\![(S)]\!](\mathcal{E}) \;=\; <\mathcal{E}_1, \mathrm{true}>}$$

[51] $$\frac{\mathrm{sem}(\ll S,\, \mathcal{E},\, \perp \gg) = \ll S^+,\, \mathcal{E}_1,\, C \gg}{\mathrm{eval}[\![(S)]\!](\mathcal{E}) \;=\; <\mathcal{E}_1, \mathrm{false}>}$$

Semantics of program statements.

The 'empty program' (an empty first element of the configuration triple) identifies the base case for the recursive 'sem' function. In this case, simply the configuration is returned.

[52] $\mathrm{sem}(\ll\ ,\, \mathcal{E},\, C \gg) = \ll\ ,\, \mathcal{E},\, C \gg$

The assignment statement: the expression is evaluated and the environment is updated.

$$[53] \quad \frac{\text{evaldr}\{dr\}(\mathcal{E}) = k^{+},\ \text{eval}[\![ex]\!](\mathcal{E}) = <\mathcal{E}_1,\ v>,\ (k^{+} \longmapsto v) \rhd \mathcal{E}_1 = \mathcal{E}_2}{\text{sem}(\ll dr := ex;\ S,\ \mathcal{E},\ C \gg) \ = \ \text{sem}(\ll S,\ \mathcal{E}_2,\ C \gg)}$$

The IF statement: IF T THEN S_1 ... END.
The condition, that is the statement sequence T, is evaluated using the function 'sem'. If the computation succeeds (the first element of the resulting configuration triple is empty), the statement sequence in the THEN branch is evaluated. Otherwise, the statement sequence in the ELSE(IF) branch (if present) is evaluated.

$$[54] \quad \frac{\text{sem}(\ll T,\mathcal{E},\perp \gg) = \ll\ ,\mathcal{E}_1,\ C_1 \gg}{\text{sem}(\ll \texttt{IF}\ T\,\texttt{THEN}\ S_1\ ic\ \texttt{END};\ S_2,\mathcal{E},\ C \gg) \ = \ \text{sem}(\ll S_1;\ S_2,\mathcal{E}_1,\ C \gg)}$$

$$[55] \quad \frac{\text{sem}(\ll T,\mathcal{E},\perp \gg) = \ll S^{+},\mathcal{E}_1,\perp \gg}{\text{sem}(\ll \texttt{IF}\ T\,\texttt{THEN}\ S_1\ \ \texttt{END};\ S_2,\mathcal{E},\ C \gg) \ = \ \text{sem}(\ll S_2,\mathcal{E}_1,\ C \gg)}$$

$$[56] \quad \frac{\text{sem}(\ll T,\mathcal{E},\perp \gg) = \ll S^{+},\mathcal{E}_1,\perp \gg}{\text{sem}(\ll \texttt{IF}\ T\,\texttt{THEN}\ S_1\ \ \texttt{ELSE}\ S_2\ \texttt{END};\ S_3,\mathcal{E},\ C \gg) \ = \ \text{sem}(\ll S_2;\ S_3,\mathcal{E}_1,\ C \gg)}$$

Notice that in all cases that the computation of T/T_1 fails, the remaining statement sequences are executed in the new environment $\mathcal{E}_1$ generated by T/T_1, but with respect to the old stack of choice points. This semantical property is e.g. used in the procedure `Lex` of the third example in section 1.2
An IF-statement with an ELSIF construct is rewritten to a statement with ELSE IF.

[57] $\texttt{IF}\ T_1\ \texttt{THEN}\ S_1\ \texttt{ELSIF}\ T_2\ \texttt{THEN}\ S_2\ ei^{*}\ \texttt{END}$
$= \texttt{IF}\ T_1\ \texttt{THEN}\ S_1\ \ \texttt{ELSE IF}\ T_2\ \texttt{THEN}\ S_2\ ei^{*}\ \texttt{END END}$

[58] $\texttt{IF}\ T_1\ \texttt{THEN}\ S_1\ \texttt{ELSIF}\ T_2\ \texttt{THEN}\ S_2\ ei^{*}\ \texttt{ELSE}\ S_3\ \texttt{END}$
$= \texttt{IF}\ T_1\ \texttt{THEN}\ S_1\ \ \texttt{ELSE IF}\ T_2\ \texttt{THEN}\ S_2\ ei^{*}\ \texttt{ELSE}\ S_3\ \texttt{END END}$

The WHILE statement: WHILE T DO S END.
The condition (T) is evaluated. Depending on the result, the while loop is 'unrolled' one step or skipped.

$$[59] \quad \frac{\text{sem}(\ll T,\mathcal{E},\perp \gg) = \ll\ ,\mathcal{E}_1,\ C_1 \gg}{\begin{array}{l}\text{sem}(\ll \texttt{WHILE}\ T\,\texttt{DO}\ S\,\texttt{END};\ S_1,\mathcal{E},\ C \gg) \ = \\ \text{sem}(\ll S;\ \texttt{WHILE}\ T\,\texttt{DO}\ S\,\texttt{END};\ S_1,\mathcal{E}_1,\ C \gg)\end{array}}$$

$$[60] \quad \frac{\text{sem}(\ll T,\mathcal{E},\perp \gg) = \ll S^{+},\mathcal{E}_1,\perp \gg}{\text{sem}(\ll \texttt{WHILE}\ T\,\texttt{DO}\ S\,\texttt{END};\ S_1,\mathcal{E},\ C \gg) \ = \ \text{sem}(\ll S_1,\mathcal{E}_1,\ C \gg)}$$

The same remarks concerning the environment and the stack of choice points apply as in the case of the IF statement.

The FOR statement: FOR $id := ex_1$ TO ex_2 DO S END.
The statement is transformed into a WHILE statement. The loop control variable is (conditionally) decremented in order to obtain a correct exit value. E.g., the evaluation of the condition `(FOR i:=1 TO 5 DO i<4 END)` results in the value `false`, with the value of `i` equal to 3.

[61] $\text{sem}(\ll \texttt{FOR}\ id := ex_1\ \texttt{TO}\ ex_2\ \texttt{DO}\ S\,\texttt{END};\ S_1,\mathcal{E},\ C \gg)$

$$= \text{sem}(\ll id := ex_1;\ \texttt{WHILE}\ id \leq ex_2\ \texttt{DO}\ S;\ id := id + 1\ \texttt{END};\ \texttt{IF}\ ex_1 \leq ex_2\ \texttt{THEN}\ id := id - 1\ \texttt{END};\ S_1, \mathcal{E}, C \gg)$$

In the following part we define the semantics of the specific Alma-0 statements.

The BES extension (a boolean expression as a statement): evaluate the boolean expression with the function 'eval'. If the outcome is true, the computation continues in the new environment. Notice that $\mathcal{E}_1$ can differ from $\mathcal{E}$ because a statement like `x := 1` can be part of the Boolean expression.

$$[62]\quad \frac{\text{eval}[\![ex]\!](\mathcal{E}) = <\mathcal{E}_1, \text{true}>}{\text{sem}(\ll ex;\ S, \mathcal{E}, C \gg) \ = \ \text{sem}(\ll S, \mathcal{E}_1, C \gg)}$$

If the outcome is false, two cases have to be distinguished.
Case 1. The stack of choice points is empty. The computation fails, but changes in the environment are included. (This is necessary because the expression may be included in a condition.)

$$[63]\quad \frac{\text{eval}[\![ex]\!](\mathcal{E}) = <\mathcal{E}_1, \text{false}>}{\text{sem}(\ll ex;\ S, \mathcal{E}, \perp \gg) \ = \ \ll ex;\ S, \mathcal{E}_1, \perp \gg}$$

Case 2. The stack of choice points is not empty. Backtracking takes place. This implies that a statement sequence and an environment are extracted from the choice point at the top of the stack. The functions 'fst' and 'snd' have been specified in he module ChoicePoints.

$$[64]\quad \frac{\text{eval}[\![ex]\!](\mathcal{E}) = <\mathcal{E}_1, \text{false}>}{\text{sem}(\ll ex;\ S, \mathcal{E}, CP \bowtie C \gg) \ = \ \text{sem}(\ll \text{fst}(CP), \text{snd}(CP), C \gg)}$$

The OR statement: EITHER S_1 ORELSE S_2 ... END.
Evaluate the first branch and push the remaining alternative(s) on the stack of choice points. A tuple with a statement (sequence) and the current environment is placed on top of the stack. In case of more than one ORELSE branch, a new OR statement together with the remaining statements is pushed on the stack.

$$[65]\quad \text{sem}(\ll \texttt{EITHER}\ S_1\ \texttt{ORELSE}\ S_2\ \texttt{END};\ S_3, \mathcal{E}, C \gg) = \text{sem}(\ll S_1;\ S_3, \mathcal{E}, \prec S_2;\ S_3, \mathcal{E} \succ \bowtie C \gg)$$

$$[66]\quad \text{sem}(\ll \texttt{EITHER}\ S_1\ \texttt{ORELSE}\ S_2\ oe^+\ \texttt{END};\ S_3, \mathcal{E}, C \gg) = \text{sem}(\ll S_1;\ S_3, \mathcal{E}, \prec \texttt{EITHER}\ S_2\ oe^+\ \texttt{END};\ S_3, \mathcal{E} \succ \bowtie C \gg)$$

The SOME statement: SOME $id := ex_1 TO ex_2$ DO S END.
Case 1. The value of ex_1 is greater than the value of ex_2. The whole statement is skipped.

$$[67]\quad \frac{\text{eval}[\![ex_1 > ex_2]\!](\mathcal{E}) = <\mathcal{E}_1, \text{true}>}{\text{sem}(\ll \texttt{SOME}\ id := ex_1\ \texttt{TO}\ ex_2\ \texttt{DO}\ S\ \texttt{END};\ S_1, \mathcal{E}, C \gg) \ = \ \text{sem}(\ll S_1, \mathcal{E}_1, C \gg)}$$

Case 2. The value of both expressions is equal. The SOME statement reduces to a single assignment, followed by a statement sequence.

$$[68]\quad \frac{\text{eval}[\![ex_1 = ex_2]\!](\mathcal{E}) = <\mathcal{E}_1, \text{true}>}{\text{sem}(\ll \texttt{SOME}\ id := ex_1\ \texttt{TO}\ ex_2\ \texttt{DO}\ S\ \texttt{END};\ S_1, \mathcal{E}, C \gg) \ = \ \text{sem}(\ll id := ex_1;\ S;\ S_1, \mathcal{E}_1, C \gg)}$$

Case 3. The value of ex_1 is less than the value of ex_2. The SOME statement transforms into an OR statement with assignments as branches, followed by a statement sequence.

$$[69]\quad \frac{\text{eval}[\![ex_1 < ex_2]\!](\mathcal{E}) = <\mathcal{E}_1, \text{true}>,\ \text{eval}[\![ex_1]\!](\mathcal{E}) = <\mathcal{E}_2, z>}{\begin{array}{l}\text{sem}(\ll \texttt{SOME}\ id := ex_1\ \texttt{TO}\ ex_2\ \texttt{DO}\ S\ \texttt{END};\ S_1, \mathcal{E}, C \gg) = \\ \text{sem}(\ll \texttt{EITHER}\ id := ex_1\ \texttt{ORELSE}\ \texttt{SOME}\ id := z + 1\ \texttt{TO}\ ex_2\ \texttt{DO}\ \texttt{END}\ \texttt{END}; \\ \quad S;\ S_1, \mathcal{E}, C \gg)\end{array}}$$

The COMMIT statement: COMMIT S END.
Case 1. The computation of S succeeds. The computation continues with the initial stack.

$$[70] \quad \frac{\text{sem}(\ll S, \mathcal{E}, \perp \gg) = \ll\ , \mathcal{E}_1, C_1 \gg}{\text{sem}(\ll \texttt{COMMIT}\ S\ \texttt{END};\ S_1, \mathcal{E}, C \gg) \ = \ \text{sem}(\ll S_1, \mathcal{E}_1, C \gg)}$$

Case 2: the computation of S fails (the first element of the resulting configuration triple is not empty). Then backtracking takes place.

$$[71] \quad \frac{\text{sem}(\ll S, \mathcal{E}, \perp \gg) = \ll S^+, \mathcal{E}_1, \perp \gg}{\begin{array}{l}\text{sem}(\ll \texttt{COMMIT}\ S\ \texttt{END};\ S_1, \mathcal{E}, CP \bowtie C \gg) \ = \\ \text{sem}(\ll \text{fst}(CP), \text{snd}(CP), C \gg)\end{array}}$$

The FORALL statement: FORALL S DO T END.
The semantics of the FORALL statement is defined in a separate function 'semFA', specified below. A separate function is needed because evaluation of the FORALL statement requires a local stack of choice points, generated by the evaluation of the statement sequence S. Within the context of the 'semFA' function, the configuration stack is used for this local stack. We distinguish two cases.

Case 1. The computation of the FORALL statement succeeds. We continue with the new environment and the initial stack. So, choice points that are generated during the execution of T are ignored.

$$[72] \quad \frac{\text{semFA}(\ll \texttt{FORALL}\ S\ \texttt{DO}\ T\ \texttt{END}, \mathcal{E}, \perp \gg) = \ll\ , \mathcal{E}_1, \perp \gg}{\text{sem}(\ll \texttt{FORALL}\ S\ \texttt{DO}\ T\ \texttt{END};\ S_1, \mathcal{E}, C \gg) \ = \ \text{sem}(\ll S_1, \mathcal{E}_1, C \gg)}$$

Case 2. The computation of the FORALL statement fails. Then backtracking takes place.

$$[73] \quad \frac{\text{semFA}(\ll \texttt{FORALL}\ S\ \texttt{DO}\ T\ \texttt{END}, \mathcal{E}, \perp \gg) = \ll S^+, \mathcal{E}_1, \perp \gg}{\text{sem}(\ll \texttt{FORALL}\ S\ \texttt{DO}\ T\ \texttt{END};\ S_1, \mathcal{E}, CP \bowtie C \gg) \ = \ \text{sem}(\ll \text{fst}(CP), \text{snd}(CP), C \gg)}$$

A computation fails if none of the equations for the function 'sem' given above applies. A failure is indicated by a non-empty statement-sequence S^+ as a first element of the configuration-triple. Together with the environment (the second element of the triple), the first statement of the remaining sequence reveals the cause of the failure. The stack of choice points is empty, otherwise backtracking would have taken place.
This leads to the following default equation for the function 'sem'.

$$[74] \quad \text{sem}(\ll S^+, \mathcal{E}, \perp \gg) = \ll S^+, \mathcal{E}, \perp \gg \quad \textbf{otherwise}$$

The function 'semFA' specifies the semantics of an isolated FORALL statement. As mentioned before, the stack in the input configuration is now the stack of choice points created by the statement sequence S.
We distinguish three cases.
Case 1. The computation of S fails.
Then the FORALL statement is skipped. This means that the 'semFA' function returns a configuration with the empty statement sequence, the initial environment and an empty stack of choice points. In the computation of S the stack of choice points C, created by the computation of previous alternatives of S, is taken into account.

$$[75] \quad \frac{\text{sem}(\ll S, \mathcal{E}, C \gg) = \ll S^+, \mathcal{E}_1, \perp \gg}{\text{semFA}(\ll \texttt{FORALL}\ S\ \texttt{DO}\ T\ \texttt{END}, \mathcal{E}, C \gg) \ = \ \ll\ , \mathcal{E}, \perp \gg}$$

Case 2. The computation of S succeeds, but the computation of T fails.
Then the computation of the FORALL statement fails. A configuration with a non-empty statement sequence is returned.

$$[76]\quad \frac{\text{sem}(\ll S, \mathcal{E}, C \gg) = \ll\, , \mathcal{E}_1, C_1 \gg,\ \text{sem}(\ll T, \mathcal{E}_1, \bot \gg) = \ll S^+, \mathcal{E}_2, \bot \gg}{\text{semFA}(\ll \texttt{FORALL}\ S\ \texttt{DO}\ T\ \texttt{END}, \mathcal{E}, C \gg) \ = \ \ll S^+, \mathcal{E}_2, \bot \gg}$$

Case 3 The computation of both S and T succeed.
Subcase 3.1. After the computation of S no stack of choice points is left. The computation of the FORALL statement succeeds. The resulting environment is the initial environment $\mathcal{E}$, updated with the changes resulting from the computation of T. These changes are computed with the function 'changes', specified below.

$$[77]\quad \frac{\begin{array}{c}\text{sem}(\ll S, \mathcal{E}, C \gg) = \ll\, , \mathcal{E}_1, \bot \gg,\ \text{sem}(\ll T, \mathcal{E}_1, \bot \gg) = \ll\, , \mathcal{E}_2, C_2 \gg,\\ \text{changes}(\mathcal{E}_1, \mathcal{E}_2) = \mathcal{E}_3\end{array}}{\text{semFA}(\ll \texttt{FORALL}\ S\ \texttt{DO}\ T\ \texttt{END}, \mathcal{E}, C \gg) \ = \ \ll\, , \mathcal{E}_3 \rhd \mathcal{E}, \bot \gg}$$

Subcase 3.2. After the computation of S the stack of choice points is not empty. The function 'semFA' is recursively called with S taken from the top of the stack and $\mathcal{E}$ equal to the environment taken from the top of the stack, updated with the changes resulting from the computation of T. For the resulting environment both the changes resulting from the computation of T ($\mathcal{E}_3$) and the changes due to the recursive call of 'semFA' ($\mathcal{E}_5$) are used to update the initial environment $\mathcal{E}$.

$$[78]\quad \frac{\begin{array}{c}\text{sem}(\ll S, \mathcal{E}, C \gg) = \ll\, , \mathcal{E}_1, CP \bowtie C_1 \gg,\ \text{sem}(\ll T, \mathcal{E}_1, \bot \gg) = \ll\, , \mathcal{E}_2, C_2 \gg,\\ \text{changes}(\mathcal{E}_1, \mathcal{E}_2) = \mathcal{E}_3,\\ \text{semFA}(\ll \texttt{FORALL}\ \text{fst}(CP)\ \texttt{DO}\ T\ \texttt{END}, \mathcal{E}_3 \rhd \text{snd}(CP), C_1 \gg) = \ll\, , \mathcal{E}_4, \bot \gg,\\ \text{changes}(\mathcal{E}_3 \rhd \text{snd}(CP), \mathcal{E}_4) = \mathcal{E}_5\end{array}}{\text{semFA}(\ll \texttt{FORALL}\ S\ \texttt{DO}\ T\ \texttt{END}, \mathcal{E}, C \gg) \ = \ \ll\, , \mathcal{E}_5 \rhd (\mathcal{E}_3 \rhd \mathcal{E}), \bot \gg}$$

If the recursive call of 'semFA' fails, then the whole computation still fails. A configuration with a non-empty statement sequence is returned.

$$[79]\quad \frac{\begin{array}{c}\text{sem}(\ll S, \mathcal{E}, C \gg) = \ll\, , \mathcal{E}_1, CP \bowtie C_1 \gg,\\ \text{sem}(\ll T, \mathcal{E}_1, \bot \gg) = \ll\, , \mathcal{E}_2, C_2 \gg,\\ \text{changes}(\mathcal{E}_1, \mathcal{E}_2) \rhd \text{snd}(CP) = \mathcal{E}_3,\\ \text{semFA}(\ll \texttt{FORALL}\ \text{fst}(CP)\ \texttt{DO}\ T\ \texttt{END}, \mathcal{E}_3 \rhd \text{snd}(CP), C_1 \gg) = \ll S^+, \mathcal{E}_4, \bot \gg\end{array}}{\text{semFA}(\ll \texttt{FORALL}\ S\ \texttt{DO}\ T\ \texttt{END}, \mathcal{E}, C \gg) \ = \ \ll S^+, \mathcal{E}_4, \bot \gg}$$

The function 'changes' isolates the changes that have been made to the environment while executing the T-part of the FORALL statement (the 'permanent' changes). The first argument is the environment before the computation of T, the second argument is the environment after the computation of T.

The first equation isolates a variable binding the value of which has been changed in the computation of T.

$$[80]\quad \frac{\text{eq}(v_1, v_2) = \text{false}}{\begin{array}{l}\text{changes}([ae_1^*\ id \mapsto v_1\ ae_2^*], [ae_3^*\ id \mapsto v_2\ ae_4^*]) \ = \\ [id \mapsto v_2] \rhd \text{changes}([ae_1^*\ ae_2^*], [ae_3^*\ ae_4^*])\end{array}}$$

The second equation isolates the binding of a new variable, introduced in the computation of T.

$$[81]\quad \frac{\text{def}(\mathcal{E}, id) = \text{false}}{\text{changes}(\mathcal{E}, [ae_1^*\ id \mapsto v\ ae_2^*]) \ = \ [id \mapsto v] \rhd \text{changes}(\mathcal{E}, [ae_1^*\ ae_2^*])}$$

If none of these equations apply, the function results in an empty environment, as specified by the default equation.

[82] $\text{changes}(\mathcal{E}_1, \mathcal{E}_2) = []$ **otherwise**

6.3 Block Semantics

In order to have a formal definition of the semantics of Alma-0 programs that include procedures, the semantics of a *block* is specified. The syntax of a block has been given in section 4.4. In this section we define a function 'semBlock'. This function has a block as input argument and produces a configuration triple (see the preceding section) as output. In the definition of the function 'semBlock' a long list of local help-functions plays a role. Before going into the details of the definition, we first give an outline of the approach that has been followed.

In the definition of the semantics of a block two problems have to be solved:

1. The semantics of a procedure call has to be defined. Actual/formal parameters and local variables have to be handled. This problem is generic for any language that facilitates the use of procedures/functions.

   This problem is solved as follows. Before computing the semantics of a statement sequence, the procedures of a block are 'stored' in the environment, with the procedure name (an identifier) as a key, and a construct with the procedure block and, if present, a formal parameter list, as a value. In the specification below the function 'initEnv' performs this task. Furthermore, the definition of the function 'sem' is extended with four equations that define the computation of a procedure call. At the end the procedures are removed from the environment, so they are not present in the configuration that is returned by the function 'semBlock'.

2. Backtracking in a procedure body has to be specified. In Alma-0 programs a choice point can be generated in a procedure body, and hence, backtracking has to be possible up to a non-deterministic statement somewhere in a procedure.

   The solution for this problem is rather complex. After a successful procedure call, the stack of choice points has to be updated with the choice points that have been created by the procedure statements. These choice points cannot simply be placed on top of the stack, because this would ignore the context of the choice points. Before they are pushed on the stack, these choice points are 'packed' in a construct that contains all information that is needed to handle correctly a backtracking attempt on this choice point. Such a construct is called a *REPROC* (re-entering procedure) construct. By adding equations for the 'sem' function concerning the REPROC construct, backtracking in a procedure body is defined.

   Remark: this solution is definitely not 'implementation-directed'. In implementations of languages that support backtracking constructs are used that are addressed by pointers, like the *trail* in the Warren Abstract Machine ([H. 91]) or the *log* in the Alma Abstract Architecture of Partington ([Par97]). The absence of pointers (or, stated more generally, a 'global state') in an algebraic specification formalism like ASF+SDF makes the application of more or less artificial constructs necessary.

So, the module BlockSemantics contains additional equations for the 'sem' function. In these equations, almost all 'processing' is specified in a long list of conditions. These lists can be read as a sequence of computational steps that have to be performed in order to give a correct specification of the semantics of a procedure call or backtracking in a procedure body. As in the other semantics modules, equations are interleaved with explanatory text.

module BlockSemantics
imports StatementSemantics
exports
context-free syntax
semBlock "(" Block ")" → Config
hiddens
sorts ExpTuple
context-free syntax

initEnv "(" Decl∗ ")"	→ Env
removeProcDecl "(" Env ")"	→ Env
getLocalVars "(" Decl∗ ")"	→ IdentList
getFormalPars "(" FormalPar ")"	→ IdentList
removeProcIds "(" Env "," IdentList ")"	→ Env
hide "(" Env "," IdentList ")"	→ Env
unhide "(" Env ")"	→ Env
"REPROC" "(" IdentList "," StSeq ")"	→ Stat
"REPROC" "(" IdentList "," FormalPar "," ExpList "," StSeq ")"	→ Stat
stUpdate "(" IdentList "," StSeq "," Stack "," Stack ")"	→ Stack
stUpdate "(" IdentList "," FormalPar "," ExpList "," StSeq "," Stack "," Stack ")"	→ Stack
preParam "(" FormalPar "," ExpList "," Env ")"	→ Env
postParam "(" FormalPar "," ExpList "," Env ")"	→ Env
isVar "(" Exp ")"	→ Bool
"{" ExpList ";" ExpList "}"	→ ExpTuple
split "(" FPSection "," ExpList ")"	→ ExpTuple

equations
Block semantics: first an initial environment is constructed from the declarations section of a block. Then the sem-function is applied to the statement sequence in the block. At the end, the procedure declarations are removed from the environment.

[83] $$\frac{\begin{array}{c}\text{initEnv}(dl^*) = \mathcal{E},\\ \text{sem}(\ll S, \mathcal{E}, \perp \gg) = \ll S_1, \mathcal{E}_1, C \gg,\\ \text{removeProcDecl}(\mathcal{E}_1) = \mathcal{E}_2\end{array}}{\text{semBlock}(dl^* \;\texttt{BEGIN}\; S \;\texttt{END}) \;=\; \ll S_1, \mathcal{E}_2, C \gg}$$

An initial environment is derived from a declarations section. In this specification, only procedure declarations are added to the environment. As type-checking is not part of the specification, type declarations and variable declarations are ignored.

[84] $\text{initEnv}() = []$

Procedure declaration without formal parameters.

[85] $$\frac{\text{initEnv}(dl^*) = [ae^*]}{\text{initEnv}(\texttt{PROCEDURE}\; id;\; bl\; id;\; dl^*) \;=\; [id \mapsto bl\; ae^*]}$$

Procedure declaration with formal parameters.

[86] $$\frac{\text{initEnv}(dl^*) = [ae^*]}{\text{initEnv}(\texttt{PROCEDURE}\; id\; fp;\; bl\; id;\; dl^*) \;=\; [id \mapsto fp \diamond bl\; ae^*]}$$

Ignore other declarations.

[87] $\text{initEnv}(dl\ dl^*) = \text{initEnv}(dl^*)$ **otherwise**

Semantics of a procedure call.
Additional definition of the 'sem' function for a statement containing a procedure call.
A procedure call is recognized by the presence of a procedure 'declaration' in the environment. This is tested by retrieving a 'procedure pattern' from the environment. Regardless the presence or absence of formal parameters the following computations are made.

- A list of local variables is constructed with the function 'getLocalVars'.
- In the environment all variables that have a name identical to a local variable name are occluded by the function 'hide'.
- The semantics of the procedure statements is computed with the function 'sem'.
- All local variable bindings are removed from the resulting environment with the function 'removeProcIds'.
- In the environment the global variables that have been occluded are made visible again with the function 'unhide'.
- With the function 'stUpdate' the stack of choice points is updated with the choice points that have been generated by the procedure statements.

For a procedure with parameters some additional computations are required.

- Formal parameters are bound to actual parameters. With the function 'preParam' an environment is created in which formal parameters are bound to actual parameter values (if present).
- A list of formal parameter identifiers is created with the function 'getFormalPars'.
- All elements of this list are also input for the functions 'hide' and 'unhide'. So, all global variables with a name equal to a formal parameter are also temporary occluded.
- After the computation of the semantics of the procedure statements, with the function 'postParam' an environment is created in which actual parameters are bound to values derived from the procedure execution. In an environment update these bindings are added to the resulting environment.

If the computation of the semantics of the procedure statements fails, backtracking is tried with the initial stack of choice points.

1. Procedure without parameters.
1.1 The Procedure call succeeds.

[88]
$$\frac{\begin{array}{c} \mathcal{E}(id) = dl^*\ \texttt{BEGIN}\ T\ \texttt{END}, \\ \text{getLocalVars}(dl^*) = id^*, \\ \text{hide}(\mathcal{E},\ id^*) = \mathcal{E}_1, \\ \text{sem}(\ll T, \mathcal{E}_1, \perp \gg) = \ll\ , \mathcal{E}_2,\ C_1 \gg, \\ \text{removeProcIds}(\mathcal{E}_2,\ id^*) = \mathcal{E}_3, \\ \text{unhide}(\mathcal{E}_3) = \mathcal{E}_4, \\ \text{stUpdate}(id^*,\ S,\ C_1,\ C) = C_2 \end{array}}{\text{sem}(\ll id;\ S, \mathcal{E},\ C \gg)\ =\ \text{sem}(\ll S, \mathcal{E}_4,\ C_2 \gg)}$$

1.2 The procedure call fails. Backtracking takes place.

$$[89]\quad \frac{\begin{array}{c}\mathcal{E}(id) = dl^{*}\ \texttt{BEGIN}\ T\ \texttt{END},\\ \mathrm{getLocalVars}(dl^{*}) = id^{*},\\ \mathrm{hide}(\mathcal{E},\ id^{*}) = \mathcal{E}_1,\\ \mathrm{sem}(\ll T, \mathcal{E}_1, \perp \gg) = \ll S^{+}, \mathcal{E}_2, \perp \gg\end{array}}{\mathrm{sem}(\ll id;\ S, \mathcal{E},\ CP \bowtie C \gg)\ =\ \mathrm{sem}(\ll \mathrm{fst}(CP), \mathrm{snd}(CP),\ C \gg)}$$

2. Procedure with parameters.
2.1 The procedure call succeeds.

$$[90]\quad \frac{\begin{array}{c}\mathcal{E}(id) = fp \diamond dl^{*}\ \texttt{BEGIN}\ T\ \texttt{END},\\ \mathrm{preParam}(fp,\ ex^{+}, \mathcal{E}) = [ae^{*}],\\ \mathrm{getLocalVars}(dl^{*}) = id^{*},\\ \mathrm{getFormalPars}(fp) = id'^{*},\\ \mathrm{hide}(\mathcal{E},\ id^{*},\ id'^{*}) = [ae'^{*}],\\ \mathrm{sem}(\ll T, [ae^{*}\ ae'^{*}], \perp \gg) = \ll\ , \mathcal{E}_1,\ C_1 \gg,\\ \mathrm{postParam}(fp,\ ex^{+}, \mathcal{E}_1) = \mathcal{E}_2,\\ \mathrm{removeProcIds}(\mathcal{E}_1,\ id^{*},\ id'^{*}) = \mathcal{E}_3,\\ \mathrm{unhide}(\mathcal{E}_3) = \mathcal{E}_4,\\ \mathrm{stUpdate}(id^{*},\ fp,\ ex^{+},\ S,\ C_1,\ C) = C_2\end{array}}{\mathrm{sem}(\ll id(ex^{+});\ S, \mathcal{E},\ C \gg)\ =\ \mathrm{sem}(\ll S, \mathcal{E}_2 \triangleright \mathcal{E}_4,\ C_2 \gg)}$$

2.2 The procedure call fails. Backtracking takes place.

$$[91]\quad \frac{\begin{array}{c}\mathcal{E}(id) = fp \diamond dl^{*}\ \texttt{BEGIN}\ T\ \texttt{END},\\ \mathrm{preParam}(fp,\ ex^{+}, \mathcal{E}) = [ae^{*}],\\ \mathrm{getLocalVars}(dl^{*}) = id^{*},\\ \mathrm{getFormalPars}(fp) = id'^{*},\\ \mathrm{hide}(\mathcal{E},\ id^{*},\ id'^{*}) = [ae'^{*}],\\ \mathrm{sem}(\ll T, [ae^{*}\ ae'^{*}], \perp \gg) = \ll S^{+}, \mathcal{E}_1, \perp \gg\end{array}}{\mathrm{sem}(\ll id(ex^{+});\ S, \mathcal{E},\ CP \bowtie C \gg)\ =\ \mathrm{sem}(\ll \mathrm{fst}(CP), \mathrm{snd}(CP),\ C \gg)}$$

Semantics of the REPROC construct.
The construct contains all information that is needed for resuming program execution within the body of a procedure. In the equations below, following the computation of the semantics of the remaining procedure statements we find the same computations as in the equations above.

1. Procedure without parameters.
1.1 Backtracking succeeds, normal procedure exit.

$$[92]\quad \frac{\begin{array}{c}\mathrm{sem}(\ll T, \mathcal{E}, \perp \gg) = \ll\ , \mathcal{E}_1,\ C_1 \gg,\\ \mathrm{removeProcIds}(\mathcal{E}_1,\ id^{*}) = \mathcal{E}_2,\\ \mathrm{unhide}(\mathcal{E}_2) = \mathcal{E}_3,\\ \mathrm{stUpdate}(id^{*},\ S,\ C_1,\ C) = C_2\end{array}}{\mathrm{sem}(\ll \mathrm{REPROC}(id^{*},\ T);\ S, \mathcal{E},\ C \gg)\ =\ \mathrm{sem}(\ll S, \mathcal{E}_3,\ C_2 \gg)}$$

1.2 Backtracking with REPROC fails. Backtracking to the next choice point takes place.

$$[93]\quad \frac{\mathrm{sem}(\ll T, \mathcal{E}, \perp \gg) = \ll S^{+}, \mathcal{E}_1, \perp \gg}{\mathrm{sem}(\ll \mathrm{REPROC}(id^{*},\ T);\ S, \mathcal{E},\ CP \bowtie C \gg)\ =\ \mathrm{sem}(\ll \mathrm{fst}(CP), \mathrm{snd}(CP),\ C \gg)}$$

If the stack of choice points is empty a failure is encountered: the REPROC construct is replaced by the boolean constant `false`. The changes made in the environment are also returned. So, if the failure takes place while evaluating a condition, the environment changes can be used.

$$[94]\quad \frac{\begin{array}{c}\text{sem}(\ll T, \mathcal{E}, \perp \gg) = \ll S^+, \mathcal{E}_1, \perp \gg, \\ \text{removeProcIds}(\mathcal{E}_1, id^*) = \mathcal{E}_2, \\ \text{unhide}(\mathcal{E}_2) = \mathcal{E}_3\end{array}}{\text{sem}(\ll \text{REPROC}(id^*, T); S, \mathcal{E}, \perp \gg) \; = \; \ll \text{false}; S, \mathcal{E}_3, \perp \gg}$$

2 Procedure with parameters.
2.1 Backtracking succeeds, normal procedure exit.

$$[95]\quad \frac{\begin{array}{c}\text{sem}(\ll T, \mathcal{E}, \perp \gg) = \ll , \mathcal{E}_1, C_1 \gg, \\ \text{postParam}(fp, ex^+, \mathcal{E}_1) = \mathcal{E}_2, \\ \text{getFormalPars}(fp) = id'^*, \\ \text{removeProcIds}(\mathcal{E}_1, id^*, id'^*) = \mathcal{E}_3, \\ \text{unhide}(\mathcal{E}_3) = \mathcal{E}_4, \\ \text{stUpdate}(id^*, fp, ex^+, S, C_1, C) = C_2\end{array}}{\text{sem}(\ll \text{REPROC}(id^*, fp, ex^+, T); S, \mathcal{E}, C \gg) \; = \; \text{sem}(\ll S, \mathcal{E}_2 \triangleright \mathcal{E}_4, C_2 \gg)}$$

2.2 Backtracking with REPROC fails. Backtracking to the next choice point takes place.

$$[96]\quad \frac{\text{sem}(\ll T, \mathcal{E}, \perp \gg) = \ll S^+, \mathcal{E}_1, \perp \gg}{\text{sem}(\ll \text{REPROC}(id^*, fp, ex^+, T); S, \mathcal{E}, CP \bowtie C \gg) \; = \; \text{sem}(\ll \text{fst}(CP), \text{snd}(CP), C \gg)}$$

If backtracking is not possible the same holds as in the case of a REPROC construct for a procedure without parameters.

$$[97]\quad \frac{\begin{array}{c}\text{sem}(\ll T, \mathcal{E}, \perp \gg) = \ll S^+, \mathcal{E}_1, \perp \gg, \\ \text{postParam}(fp, ex^+, \mathcal{E}_1) = \mathcal{E}_2, \\ \text{getFormalPars}(fp) = id'^*, \\ \text{removeProcIds}(\mathcal{E}_1, id^*, id'^*) = \mathcal{E}_3, \\ \text{unhide}(\mathcal{E}_3) = \mathcal{E}_4\end{array}}{\text{sem}(\ll \text{REPROC}(id^*, fp, ex^+, T); S, \mathcal{E}, \perp \gg) \; = \; \ll \text{false}; S, \mathcal{E}_2 \triangleright \mathcal{E}_4, \perp \gg}$$

In the remaining part of the module the various help-functions are specified.

Remove procedure declarations from the final environment.

[98] $\text{removeProcDecl}([ae^* \; id \mapsto bl \; ae'^*]) = \text{removeProcDecl}([ae^* \; ae'^*])$
[99] $\text{removeProcDecl}([ae^* \; id \mapsto fp \diamond bl \; ae'^*]) = \text{removeProcDecl}([ae^* \; ae'^*])$
[100] $\text{removeProcDecl}(\mathcal{E}) = \mathcal{E}$ **otherwise**

Make a list of local variables from a list of local declarations.

[101] $\text{getLocalVars}() =$

1. Skip type declarations.

[102] $\text{getLocalVars}(\texttt{TYPE}\; td^+; dl^*) = \text{getLocalVars}(dl^*)$

2. Get identifiers from var declarations.

[103] $\text{getLocalVars}(\texttt{VAR}\; id^+ : tp;) = id^+$

[104] $$\frac{\text{getLocalVars}(\texttt{VAR}\ vd^{+};) = id^{*}}{\text{getLocalVars}(\texttt{VAR}\ id^{+} : tp;\ vd^{+};)\ =\ id^{+},\ id^{*}}$$

Make a list of formal parameter identifiers from a formal parameter section.

[105] $\text{getFormalPars}((id^{+} : ft)) = id^{+}$
[106] $\text{getFormalPars}((\texttt{VAR}\ id^{+} : ft)) = id^{+}$
[107] $\text{getFormalPars}((\texttt{MIX}\ id^{+} : ft)) = id^{+}$

One equation for more than one formal parameter section.

[108] $$\frac{\text{getFormalPars}((fs)) = id^{+},\ \text{getFormalPars}((fs^{+})) = id'^{+}}{\text{getFormalPars}((fs;\ fs^{+}))\ =\ id^{+},\ id'^{+}}$$

Remove local variables and formal parameters from the environment after procedure evaluation.

[109] $\text{removeProcIds}(\mathcal{E},\ ) = \mathcal{E}$

[110] $$\frac{\text{remove}(\mathcal{E},\ id) = \mathcal{E}_1}{\text{removeProcIds}(\mathcal{E},\ id,\ id^{*})\ =\ \text{removeProcIds}(\mathcal{E}_1,\ id^{*})}$$

Hide a (global) variable in the environment if there is a local variable or a formal parameter with the same name. Hiding is done by placing the identifier key between braces. Every identifier that is already placed between brackets gets another pair.

[111] $\text{hide}([],\ id^{*}) = []$
[112] $\text{hide}([id \mapsto v\ ae^{*}],\ id^{*},\ id,\ id'^{*}) = [\{id\} \mapsto v] \rhd \text{hide}([ae^{*}],\ id^{*},\ id'^{*})$
[113] $\text{hide}([\{k\} \mapsto v\ ae^{*}],\ id^{*}) = [\{\{k\}\} \mapsto v] \rhd \text{hide}([ae^{*}],\ id^{*})$

[114] $\text{hide}([id \mapsto v\ ae^{*}],\ id^{*})\ =\ [id \mapsto v] \rhd \text{hide}([ae^{*}],\ id^{*})$
otherwise

Unhide the global variables that have been occluded by removing the braces around an identifier key.

[115] $\text{unhide}([]) = []$
[116] $\text{unhide}([id \mapsto v\ ae^{*}]) = [id \mapsto v] \rhd \text{unhide}([ae^{*}])$
[117] $\text{unhide}([\{k\} \mapsto v\ ae^{*}]) = [k \mapsto v] \rhd \text{unhide}([ae^{*}])$

Update of the stack of choice points after a procedure call.
We have two slightly different functions, one for a procedure without parameters, the other for a procedure with parameters.

1. Procedure without parameters.
The function 'stUpdate' has the following list of arguments:

- a list of local variables,
- the sequence of statements following the procedure call.
- the stack of choice points generated during the procedure execution.
- the (global) stack of choice points before the procedure call.

If no choice points have been generated the global stack remains unchanged.

[118] $\text{stUpdate}(id^*, S, \bot, C) = C$

If a stack of choice points has been generated during the execution of a procedure, for each choice point a 'REPROC' construct is pushed on the global stack of choice points. The REPROC construct contains all the information that is required to resume the execution of a procedure at a choice point.

[119] $$\frac{\prec \text{REPROC}(id^*, T);\, S, \mathcal{E} \succ = CP}{\text{stUpdate}(id^*, S, \prec T, \mathcal{E} \succ \bowtie C, C_1) \;=\; CP \bowtie \text{stUpdate}(id^*, S, C, C_1)}$$

2. Procedure with parameters.
A list of formal parameters and a list of actual parameters are added to the REPROC construct.

[120] $\text{stUpdate}(id^*, fp, ex^+, S, \bot, C) = C$

[121] $$\frac{\prec \text{REPROC}(id^*, fp, ex^+, T);\, S, \mathcal{E} \succ = CP}{\text{stUpdate}(id^*, fp, ex^+, S, \prec T, \mathcal{E} \succ \bowtie C, C_1) \;=\; CP \bowtie \text{stUpdate}(id^*, fp, ex^+, S, C, C_1)}$$

Parameter processing for a procedure call.

A. Pre-processing.
Before the execution of the procedure statements an environment is created in which the formal parameters (as far as required) are bound to values that are derived from the corresponding actual parameters. The function 'preParam' has three arguments: the formal parameter sections of the procedure, a list with the actual parameters (expressions) and the environment at the moment the procedure is called.

1. Value parameter(s).
Extract an environment with the formal parameters bound to the actual parameter values.

[122] $$\frac{\text{eval}[\![ex]\!](\mathcal{E}) = <\mathcal{E}_1, v>}{\text{preParam}((id : ft)\ , ex, \mathcal{E}) \;=\; [id \mapsto v]}$$

[123] $$\frac{\text{eval}[\![ex]\!](\mathcal{E}) = <\mathcal{E}_1, v>}{\text{preParam}((id, id^+ : ft)\ , ex, ex^+, \mathcal{E}) \;=\; [id \mapsto v] \triangleright \text{preParam}((id^+ : ft)\ , ex^+, \mathcal{E}_1)}$$

2. VAR parameter(s).
2.1 The actual parameter (variable) is initialized. This is tested with the function 'known' from the module ExpressionSemantics. Its value is bound to the corresponding formal parameter.

[124] $$\frac{\text{known}(ex, \mathcal{E}) = \text{true},\ \ \text{eval}[\![ex]\!](\mathcal{E}) = <\mathcal{E}_1, v>}{\text{preParam}((\texttt{VAR}\ id : ft)\ , ex, \mathcal{E}) \;=\; [id \mapsto v]}$$

2.2 The actual parameter is not initialized. No binding is added to the environment.

[125] $$\frac{\text{known}(ex, \mathcal{E}) = \text{false}}{\text{preParam}((\texttt{VAR}\ id : ft)\ , ex, \mathcal{E}) \;=\; []}$$

The same holds for a list with more than one parameter.

[126] $$\frac{\text{known}(ex, \mathcal{E}) = \text{true},\ \ \text{eval}[\![ex]\!](\mathcal{E}) = <\mathcal{E}_1, v>}{\text{preParam}((\texttt{VAR}\ id, id^+ : ft)\ , ex, ex^+, \mathcal{E}) \;=\; [id \mapsto v] \triangleright \text{preParam}((\texttt{VAR}\ id^+ : ft)\ , ex^+, \mathcal{E}_1)}$$

$$[127] \quad \frac{\text{known}(ex, \mathcal{E}) = \text{false}}{\text{preParam}((\texttt{VAR}\ id,\ id^+ : ft)\ ,\ ex,\ ex^+, \mathcal{E})\ =\ \text{preParam}((\texttt{VAR}\ id^+ : ft)\ ,\ ex^+, \mathcal{E})}$$

3. MIX parameter(s).
Test if the actual parameter is a variable. If so, handle it like a VAR parameter.
The function 'isVar' is specified below.

$$[128] \quad \frac{\text{isVar}(ex) = \text{true}}{\text{preParam}((\texttt{MIX}\ id : ft)\ ,\ ex, \mathcal{E})\ =\ \text{preParam}((\texttt{VAR}\ id : ft)\ ,\ ex, \mathcal{E})}$$

$$[129] \quad \frac{\text{isVar}(ex) = \text{true},\ \ \text{preParam}((\texttt{VAR}\ id : ft)\ ,\ ex, \mathcal{E}) = \mathcal{E}_1}{\text{preParam}((\texttt{MIX}\ id,\ id^+ : ft)\ ,\ ex,\ ex^+, \mathcal{E})\ =\ \mathcal{E}_1 \triangleright \text{preParam}((\texttt{MIX}\ id^+ : ft)\ ,\ ex^+, \mathcal{E})}$$

Otherwise, handle the parameter like a value parameter.

$$[130] \quad \frac{\text{isVar}(ex) = \text{false}}{\text{preParam}((\texttt{MIX}\ id : ft)\ ,\ ex, \mathcal{E})\ =\ \text{preParam}((id : ft)\ ,\ ex, \mathcal{E})}$$

$$[131] \quad \frac{\text{isVar}(ex) = \text{false},\ \ \text{preParam}((id : ft)\ ,\ ex, \mathcal{E}) = \mathcal{E}_1}{\text{preParam}((\texttt{MIX}\ id,\ id^+ : ft)\ ,\ ex,\ ex^+, \mathcal{E})\ =\ \mathcal{E}_1 \triangleright \text{preParam}((\texttt{MIX}\ id^+ : ft)\ ,\ ex^+, \mathcal{E})}$$

In the case of more than one formal parameter section, the list of actual parameters is split by the function 'split' according to the number of formal parameters in the first section. The function 'preParam' is recursively applied to the remaining formal parameter sections and the remaining list of actual parameters.

$$[132] \quad \frac{\text{split}(fs,\ ex^+) = \{ex_1^+;\ ex_2^+\},\ \ \text{preParam}((fs)\ ,\ ex_1^+, \mathcal{E}) = \mathcal{E}_1}{\text{preParam}((fs;\ fs^+)\ ,\ ex^+, \mathcal{E})\ =\ \mathcal{E}_1 \triangleright \text{preParam}((fs^+)\ ,\ ex_2^+, \mathcal{E})}$$

B. Postprocessing procedure parameters.
The function 'postParam' produces an environment with the modified values of the VAR parameters. The function has three arguments: the formal parameter sections of the procedure, a list with the actual parameters (expressions) and the environment after the evaluation of the procedure statements.

1. Value parameters do not lead to modified values.

$$[133] \quad \text{postParam}((id^+ : ft)\ ,\ ex^+, \mathcal{E}) = []$$

2. VAR parameters: transfer the value of the formal parameters to the actual parameters.

$$[134] \quad \frac{\text{evaldr}\{dr\}(\mathcal{E}) = k^+,\ \ \mathcal{E} = [ae^*\ id \mapsto v\ ae'^*]}{\text{postParam}((\texttt{VAR}\ id : ft)\ ,\ dr, \mathcal{E})\ =\ (k^+ \longmapsto v) \triangleright []}$$

$$[135] \quad \frac{\text{evaldr}\{dr\}(\mathcal{E}) = k^+,\ \ \mathcal{E} = [ae^*\ id \mapsto v\ ae'^*]}{\begin{array}{l}\text{postParam}((\texttt{VAR}\ id,\ id^+ : ft)\ ,\ dr,\ ex^+, \mathcal{E})\ = \\ (k^+ \longmapsto v) \triangleright \text{postParam}((\texttt{VAR}\ id^+ : ft)\ ,\ ex^+, [ae^*\ ae'^*])\end{array}}$$

Nothing is transfered if the VAR parameter has no value (yet).

$$[136] \quad \frac{\text{def}(\mathcal{E},\ id) = \text{false}}{\text{postParam}((\texttt{VAR}\ id : ft)\ ,\ dr, \mathcal{E})\ =\ []}$$

[137] $$\frac{\mathrm{def}(\mathcal{E},\ id) = \mathrm{false}}{\mathrm{postParam}((\texttt{VAR}\ id,\ id^+ : ft)\ ,\ dr,\ ex^+,\ \mathcal{E})\ =\ \mathrm{postParam}((\texttt{VAR}\ id^+ : ft)\ ,\ ex^+,\ \mathcal{E})}$$

3. MIX parameters: distinguish between value or VAR

[138] $$\frac{\mathrm{isVar}(ex) = \mathrm{true}}{\mathrm{postParam}((\texttt{MIX}\ id : ft)\ ,\ ex,\ \mathcal{E})\ =\ \mathrm{postParam}((\texttt{VAR}\ id : ft)\ ,\ ex,\ \mathcal{E})}$$

[139] $$\frac{\mathrm{isVar}(ex) = \mathrm{true},\ \ \mathrm{postParam}((\texttt{VAR}\ id : ft)\ ,\ ex,\ \mathcal{E}) = \mathcal{E}_1}{\begin{array}{l}\mathrm{postParam}((\texttt{MIX}\ id,\ id^+ : ft)\ ,\ ex,\ ex^+,\ \mathcal{E})\ = \\ \mathcal{E}_1 \triangleright \mathrm{postParam}((\texttt{MIX}\ id^+ : ft)\ ,\ ex^+,\ \mathcal{E})\end{array}}$$

[140] $$\frac{\mathrm{isVar}(ex) = \mathrm{false}}{\mathrm{postParam}((\texttt{MIX}\ id : ft)\ ,\ ex,\ \mathcal{E})\ =\ []}$$

[141] $$\frac{\mathrm{isVar}(ex) = \mathrm{false}}{\mathrm{postParam}((\texttt{MIX}\ id,\ id^+ : ft)\ ,\ ex,\ ex^+,\ \mathcal{E})\ =\ \mathrm{postParam}((\texttt{MIX}\ id^+ : ft)\ ,\ ex^+,\ \mathcal{E})}$$

Again, one recursive equation for more than one parameter section.

[142] $$\frac{\mathrm{split}(fs,\ ex^+) = \{ex_1^+;\ ex_2^+\},\ \ \mathrm{postParam}((fs)\ ,\ ex_1^+,\ \mathcal{E}) = \mathcal{E}_1}{\mathrm{postParam}((fs;\ fs^+)\ ,\ ex^+,\ \mathcal{E})\ =\ \mathcal{E}_1 \triangleright \mathrm{postParam}((fs^+)\ ,\ ex_2^+,\ \mathcal{E})}$$

Test if the actual parameter is a variable.

[143] $\mathrm{isVar}(dr) = \mathrm{true}$
[144] $\mathrm{isVar}(ex) = \mathrm{false}$ **otherwise**

Split a list of actual parameters (expressions) in two parts. The first part has the same number of elements as the formal parameter list.

[145] $\mathrm{split}(id : ft,\ ex,\ ex^+) = \{ex;\ ex^+\}$

[146] $$\frac{\mathrm{split}(id^+ : ft,\ ex^+) = \{ex_1^+;\ ex_2^+\}}{\mathrm{split}(id,\ id^+ : ft,\ ex,\ ex^+)\ =\ \{ex,\ ex_1^+;\ ex_2^+\}}$$

Use the previous equations also for VAR and MIX parameter lists.

[147] $\mathrm{split}(\texttt{VAR}\ id^+ : ft,\ ex^+) = \mathrm{split}(id^+ : ft,\ ex^+)$
[148] $\mathrm{split}(\texttt{MIX}\ id^+ : ft,\ ex^+) = \mathrm{split}(id^+ : ft,\ ex^+)$

7 Concluding remarks

The specification presented in this report gives an executable operational semantics of a large subset of the programming language Alma-0. Parts of the specification are concerned with constructs that are present in many imperative languages and whose semantics are well-understood, like the usual selection and repetition statements. The specification of the operational semantics of these constructs is straightforward. However, the Alma-0 extensions introduced some peculiarities that required 1. more 'administration' (besides the environment also a stack of choice points had to be updated), 2. more complex handling of procedure calls, most because of backtracking within a procedure body. In absence of pointers (to environments and 'trails'), new concepts like the 'REPROC construct had to be developed.

The specification has been tested with a number of programs and program fragments, including the programs given in section 1. The results were as expected. The time performance could be better, but –as stated in section 2– efficiency has not been a primary goal while writing this specification.

Acknowledgements
Krzysztof Apt (CWI and University of Amsterdam) has put me on this job. His enthusiasm for the semantical aspects of the language Alma-0has been very stimulating and has been helpful to me in the sometimes hard job of getting everything on its place. The work of Eelco Visser (Oregon Graduate Institute, Portland) on language specification provided me with a very good starting point. Several parts of his work have been reused in the specification presented in this report. Both Krzysztof and Eelco are acknowledged for their comments on a draft version of this report.

References

[ABPS97] K.R. Apt, J. Brunekreef, V. Partington, and A. Schaerf. Alma-0: An imperative language that supports declarative programming. Technical Report PNA-R9713, Centrum voor Wiskunde en Informatica, Amsterdam, 1997. Submitted for publication elsewhere.

[AS97] K.R. Apt and A. Schaerf. Search and imperative programming. In *Proceedings of the 24th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 67–79. ACM Press, 1997.

[H. 91] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* The MIT Press, Cambridge, Massachusetts, 1991.

[Kli93] P. Klint. A meta–environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[Par97] V. Partington. Implementation of an imperative programming language with backtracking. Technical Report P9712, Programming Research Group, University of Amsterdam, 1997.

[Vis96] E. Visser. Executable Specification of Computer Languages. Unpublished lecture notes, 1996.

[Wir85] N. Wirth. *Programming in Modula-2.* Springer Verlag, New York, 1985.

Technical Reports of the Programming Research Group

Note: These reports can be obtained using the technical reports overview on our WWW site (`http://www.wins.uva.nl/research/prog/reports/`) or by anonymous ftp to `ftp.wins.uva.nl`, directory `pub/programming-research/reports/`.

[P9805] A. Sellink and C. Verhoef. *Development, Assessment, and Reengineering of Language Descriptions.*

[P9804] A. Sellink and C. Verhoef. *Native Patterns.*

[P9803] J.J. Brunekreef. *Annotated Algebraic Specification of the Syntax and Semantics of the Programming Language* Alma-0.

[P9802] W. Fokkink and C. Verhoef. *Conservative Extension in Positive/Negative Conditional Term Rewriting with Applications to Software Renovation Factories.*

[P9801] P. Klint and C. Verhoef. *Evolutionary Software Engineering: A Component-based Approach.*

[P9726] J.A. Bergstra and A. Ponse. *Grid Protocol Specifications.*

[P9725] J.A. Bergstra and A. Ponse. *Process Algebra Primitives for File Transfer.*

[P9724] J.A. Bergstra and A. Ponse. *Process Algebra with Four-Valued Logic.*

[P9723] J.A. Bergstra and A. Ponse. *Kleene's Three-Valued Logic and Process Algebra.*

[P9722] J.A. Bergstra and A. Ponse. *Bochvar-McCarthy Logic and Process Algebra.*

[P9721] M.P.A. Sellink and C. Verhoef. *Reflections on the Evolution of COBOL.*

[P9720] M. van der Graaf. *A Specification of Box to HTML in ASF+SDF.*

[P9719] M. van den Brand, M.P.A. Sellink, and C. Verhoef. *Current Parsing Techniques in Software Renovation Considered Harmful.*

[P9718] M. de Jonge. *Reuse of ASF+SDF Specifications by means of Renaming.*

[P9717] A. Sellink (Ed.). *Participants Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications.*

[P9717] E. Visser. *From Context-free Grammars with Priorities to Character Class Grammars.*

[P9716] H.M. Sneed. *Dealing with the Dual Crisis — Year 2000 and Euro — What Reverse Engineering can do to Help.*

[P9715] W. Fokkink and C. Verhoef. *An SOS Message: Conservative Extension in Higher Order Positive/Negative Conditional Term Rewriting.*

[P9714] M. van den Brand, M.P.A. Sellink, and C. Verhoef. *Control Flow Normalization for COBOL/CICS Legacy Systems.*

[P9713] B. Diertens. *Simulation and Animation of Process Algebra Specifications.*

[P9712] V. Partington. *Implementation of an Imperative Programming Language with Backtracking.*

[P9711] L. Moonen. *A Generic Architecture for Data Flow Analysis to Support Reverse Engineering.*

[P9710] B. Luttik and E. Visser. *Specification ofRewriting Strategies.*

[P9709] J.A. Bergstra and M.P.A. Sellink. *An Arithmetical Module for Rationals and Reals.*

[P9708] E. Visser. *Character Classes.*

[P9707] E. Visser. *Scannerless Generalized-LR Parsing.*

[P9706] E. Visser. *A Family of Syntax Definition Formalisms.*

[P9705] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. *Generation of Components for Software Renovation Factories from Context-free Grammars.*

[P9704] P.A. Olivier. *Debugging Distributed Applications Using a Coordination Architecture.*

[P9703] H.P. Korver and M.P.A. Sellink. *A Formal Axiomatization for Alphabet Reasoning with Parametrized Processes.*

[P9702] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. *Reengineering COBOL Software Implies Specification of the Underlying Dialects.*

[P9701] E. Visser. *Polymorphic Syntax Definition.*

[P9618] M.G.J. van den Brand, P. Klint, and C. verhoef. *Re-engineering needs Generic Programming Language Technology.*

[P9617] P.I. Manuel. *ANSI Cobol III in SDF + an ASF Definition of a Y2K Tool.*

[P9616] P.H. Rodenburg. *A Complete System of Four-valued Logic.*

[P9615] S.P. Luttik and P.H. Rodenburg. *Transformations of Reduction Systems.*

[P9614] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Core Technologies for System Renovation.*

[P9613] L. Moonen. *Data Flow Analysis for Reverse Engineering.*

[P9612] J.A. Hillebrand. *Transforming an ASF+SDF Specification into a ToolBus Application.*

[P9611] M.P.A. Sellink. *On the conservativity of Leibniz Equality.*

[P9610] T.B. Dinesh and S.M. Üsküdarlı. *Specifying input and output of visual languages.*

[P9609] T.B. Dinesh and S.M. Üsküdarlı. *The VAS formalism in VASE.*

[P9608] J.A. Hillebrand. *A small language for the specification of Grid Protocols.*

[P9607] J.J. Brunekreef. *A transformation tool for pure Prolog programs: the algebraic specification.*