CHAPTER 6

ALGOL 60

ALGOL 60 has provided a reference point for many facets of computer science. It has been formally defined a number of times (e.g. [Lauer 68a, Allen 72a, Mosses 74a]). As such, the task of writing a definition of the language provides a convenient test of a definition method. The ALGOL 60 definition in this chapter provides the reader with a first example of a VDM definition of a real (as opposed to a contrived) problem. The text discusses the models of those language concepts not covered in chapter 4. One of the outcomes of constructing a definition is a list of "reservations" about the object under study: that is, comments on how the same effect could be achieved more simply. The only major problem left in the "Modified Language" ([de Morgan 76a]) is the incomplete type checking.

This definition shows that a powerful language can have a concise definition providing the language is well thought out. (This chapter is a revised version of [Henhapl 78a].) Other complete language definitions using VDM ideas are Pascal in chapter 7, PL/1 in [Bekić 74a], Ada in [Bjørner 80f] (see also [Ada 80c]), and CHILL [CCITT 80a].

CONTENTS

(*) Sect. 6.3.4 is treated in sect. 6.2.2; sect. 6.4.2 in sect. 6.2.1.

INTRODUCTION

For many years the official description of ALGOL 60 has been the "Revised Report" [Naur 63a]. Not only the language, but also its extremely precise description have been seen as a reference point. There were, however, a number of known unresolved problems and most of these have been eliminated by the recent modifications given in [de Morgan 76a]. A number of formal definitions exist for the language of the revised report: this paper presents a denotational defnition of the language as understood from the Modified ALGOL 60 Report (MAR).

Before making some introductory remarks on the definition, three points will be made about the language itself (as in MAR). Firstly the modifications have followed the earlier *"ECMA subset"* by making the language (almost) statically typed. Although all parameters must now be specified, there is still no way of fixing the dimensions of array parameters nor the required parameter types of procedure or function parameters (cf. ALGOL 68). In connection with this, it could be observed that the parameter matching rules of section 4.7.5.5 of MAR are somewhat difficult. In particular the definition given below assumes that, for *"by name"* passing of arithmetic expressions, the types must match exactly.

The third observation is one of surprise. The decision to restrict the control variable of a *<for statement>* to be a *<variable identifier>* (i.e. not a subscripted variable) may or may not be wise: but the argument that *<for statement>* can now be defined by expansion within ALGOL is surely dangerous. The definition given here would have had no difficulty treating the more general case because the concept of location has anyway to be introduced for other purposes.

Two of the major points resolved by the modifications are the meaning of *"own"* variables and the provision of a basic set of input-output functions: particular attention has been given to these points in the formal definition below. In fact, the treatment of own given here is more detailed than that for PL/1 static variables in [ULD69c]. Rather than perform name changes and generate dummy declarations in the outermost block, an extra environment component is used here to retain a mapping from (additional) unique names to their locations. This *own-env* is used in generating the denotations for "own" variables for insertion in the local *enc*. The input-output functions are defined to change the "Channel" components of the *State*.

As discussed elsewhere in this volume, the definition of arbitrary order of evaluation has not been addressed: had it been, one would, for example, have to show that the elements of an expression can be evaluated in any order.

Neither the concrete syntax nor the translation to the abstract form are given in this definition.

6.0 BASIC DOMAINS

6.0.1 Static Environment

In order to define the context conditions (*WF*) a static environment is created. This same object facilitates the determination of the types of expressions etc.(*TP*). *Specifier* is defined in section 6.2.2 (see 6.7.2 for an Index).

$$STATICENV = Id \xrightarrow{m} Specifier$$

Note that a list of abbreviations is given in section 6.7.1.

Certain obvious steps have been taken to shorten the *WF* functions given below, for example if:

$$\theta :: \theta_1\ \theta_2\ \ldots\ \theta_n$$

then a rule (or part thereof) of the form:

$$WF[mk\text{-}\theta(\theta_1,\theta_2,\ldots,\theta_n)](env) \triangleq WF[\theta_1](env) \wedge WF[\theta_2](env) \wedge \ldots \wedge WF[\theta_n](env)$$

are omitted.

6.0.2 State

Central to the definition of the semantics is the *State* which contains the values for the (scalar) locations and for the input-output channels.

STATE :: STR :*STORE*
CHANS :*CHANNELS*
STORE = *SCALARLOC* $\overrightarrow{m}$ [*SCALARVAL*]
SCALARLOC Infinite set
SCALARVAL = *Int* | *Real* | *Bool*
CHANNELS = *Int* $\overrightarrow{m}$ *Char**

6.0.3 Environment

The denotations of (local) identifiers are contained in *env* which is a parameter to the meaning function (*M*).

$$ENV = Id \overrightarrow{m} DEN$$

$$DEN = TYPEDEN \mid ARRAYDEN \mid PROCDEN \mid ATVFNDEN \mid LABELDEN \mid SWITCHDEN \mid BYNAMEDEN \mid String$$

The denotation of (typed) scalar variables is a (scalar) location whose value is found in the *Store* part of the *State*.

$$TYPEDEN = SCALARLOC$$

Array variables denote maps from a dense (cf. *rect*) set of indices to scalar locations. Since each index denotes a separate location the mapping is one-one. As discussed in chapter 4, this mapping is part of the environment because elements of arrays can be passed as parameters "by location".

$$ARRAYDEN = Int^+ \overleftrightarrow{m} SCALARLOC$$
$$\underline{\text{constraint}}\ (\exists ipl \in (Int \times Int)^+)(\underline{domaloc} = rect(ipl))$$

Procedure denotations are functions which yield transformations.

These transformations are of the abnormal, or *exit*, type. For function procedures, a scalar value may also be returned. The obvious parameter of a procedure denotation is the list of denotations for its actual parameters. A procedure denotation also depends on a set of activation identifiers. The use of such identifiers to uniquely identify labels is discussed in Chapter 4. The current set must be passed along the (dynamic) call chain in order to ensure that new names are chosen.

$$PROCDEN = (ACTPARMDEN^* \times AID\text{-}set) \rightarrow (STATE \xrightarrow{\sim} (STATE \times [LABELDEN] \times [SCALARVAL]))$$

The denotations of actual parameters include type information because, as pointed out above, the checking of actual formal parameter matching must be performed dynamically in ALGOL.

$$ACTPARMDEN :: s\text{-}v:DEN \quad s\text{-}tp:Specifier$$

A function procedure returns a value by assignment to a location associated with the name of the function. The denotation for an "activated" function procedure must therefore include such a scalar location.

$$ATVFNDEN :: s\text{-}loc:SCALARLOC \quad s\text{-}fn:PROCDEN$$

Label denotations include an activation identifier in order to ensure uniqueness.

$$LABELDEN :: Id \quad AID$$

AID Infinite set of *Tokens*

Switch denotations are very like those for procedures. It is important to realize that switches in ALGOL 60 do not damage the "stack property" of the language (that is no goto or call could ever refer to a completed block) - they do not provide the generality of PL/1 label variables.

$$SWITCHDEN = (Int \times AID\text{-}set) \rightarrow (STATE \xrightarrow{\sim} STATE \times [LABELDEN] \times LABELDEN)$$

ALGOL 60's "by name" parameter passing is very general and is modelled by something which can be seen to be like a parameterless procedure. It is necessary to distinguish below between those actual parameters which can be evaluated to a location since the corresponding formal parameter can then be used on the left of an assignment in parameter passing by location.

$$BYNAMEDEN = BYNAMELOCDEN \mid BYNAMEEXPRDEN$$

$$BYNAMELOCDEN :: AID\text{-}set \rightarrow (STATE \xrightarrow{\sim} (STATE \times LOC))$$

$$BYNAMEEXPRDEN :: AID\text{-}set \rightarrow (STATE \xrightarrow{\sim} (STATE \times SCALARVAL))$$

6.0.4 Auxiliary Semantic Objects

A special environment is created at the program level for own variables.

$$OWNENV = Ownid \xrightarrow{m} (TYPEDEN \mid ARRAYDEN)$$

Composite environments are used in the semantic functions (M) for statements and expressions.

$$STMTENV = OWNENV \times ENV \times AID\text{-}set$$
$$EXPRENV = ENV \times AID\text{-}set$$

Auxiliary objects are used in same type clauses

$$LOC = SCALARLOC \mid ARRAYDEN$$
$$VAL = SCALARVAL \mid LABELDEN$$

The *exit* transformations used in this definition are:

$$TR = STATE \xrightarrow{\sim} (STATE \times [LABELDEN])$$

The following type clause abbreviations (~) are used:

$$M: D \Rightarrow \quad \sim \quad M: D \to TR$$
$$M: D \Rightarrow R \quad \sim \quad M: D \to STATE \xrightarrow{\sim} (STATE \times [LABELDEN] \times R)$$

6.1 PROGRAM LEVEL

6.1.1 Programs

Program :: *Programblock*
Programblock :: *Block*

Comment The Dummy *Programblock* can be thought of as defining the lifetime of the "own" declarations. It also contains the standard functions and procedures. Section 6.1.3 provides some meta-language definitions for standard functions which cannot be written in ALGOL.

Standard-proc-names = *Real-funct-names* | *Int-funct-names* | *Proc-names*

```
Real-Funct-names = "abs"    | "sqrt"    | "ln"      | "exp" |
                   "sin"    | "cos"     | "arctan"  |
                   "maxreal"| "minreal" | "epsilon"

Int-funct-names  = "iabs"   | "sign"    | "entier"  |
                   "length" | "maxint"

Proc-names       = "inchar" | "outchar" | "outstring" |
                   "stop"   | "fault"   |
                   "inreal" | "outreal" |
                   "ininteger" | "outinteger" | "outterminator"
```

Comment The quotes around the standard-procedure-names indicate the translated version of the identifiers.

```
WF[mk-Program(mk-Programblock(b))] ≜
  is-uniqueoids(b) ∧ is-constownbds(b) ∧
  (let senv0 = [n ↦ mk-Typeproc(INT)  | n∈Int-funct-names] ∪
               [n ↦ mk-Typeproc(REAL) | n∈Real-funct-names] ∪
               [n ↦ PROC | n∈Proc-names] in
   WF[b](senv0))
type Program → Bool
```

Comment $senv_0$ contains information about the language defined functions and procedures;

```
M[mk-Program(pb)](chanv) ≜
   let σ0 = mk-STATE([],chanv) in
   CHANS(M[pb](σ0))
type Program → (CHANNELS ~→ CHANNELS)

M[mk-Programblock(b)] ≜
  let owns = {d | is-within(d,b) ∧ d∈(Owntypedecl∪Ownarraydecl)} in
  def ownenv : [s-oid(d) ↦ M[d] | d∈owns];
  (fixe [ RET → I]
   in M[b](ownenv,[],{}));
   epilogue({s-id(d) | d∈owns})(ownenv)
type Programblock =>
```

6.1.2 Own Declarations

```
Owntypedecl  :: s-id:Id   s-oid:Ownid   s-desc:Type
Ownarraydecl :: s-id:Id   s-oid:Ownid   s-tp:Type   s-bdl:Boundpair+

WF[mk-Ownarraydecl(id,oid,tp,bdl)](senv) ≜
   (∀i∈indsbdl)(TP[s-bdl(bdl[i])](senv)∈Arithm ∧
                TP[s-ubd(bdl[i])](senv)∈Arithm)

M[mk-Owntypedecl(id,oid,tp)] ≜
  def loc: M[mk-Typedecl(id,oid,tp)];
  if tp=BOOL then assign(FALSE,loc)
  else assign(0,loc);
  return(loc)
type Owntypedecl => TYPEDEN

M[mk-Ownarraydecl(id,oid,tp,bdl)] ≜
  def loc: M[mk-Arraydecl(id,oid,tp,bdl)]([ ],{ });
  if tp=BOOL then for all scl∈rngloc do assign(FALSE,scl)
  else for all scl∈rng loc do assign(0,scl);
  return(loc)
type Ownarraydecl => ARRAYDEN
Comment All own variables have a defined initial value.
```

6.1.3 Standard Functions and Transput

It is assumed that the translation of the standard functions and procedures are contained in the ("fictitious") outer block. The interpretation of their *proc-decl* follows the normal interpretation rules except in the cases where the body cannot be expressed in Algol. In these cases the state transition of the non-Algol part is explicitly listed below.

Note: Referencing the translated identifiers we use quotes. For the translation of the identifier inreal we thus get: *"inreal"*

(1) In procedure *stop*:

"goto Omega" → *exit*(RET)

(2) In procedure *inchar*, *s-body* contains:

```
def chv   : contents(env("channel"));
let str   = env("str")in
def int   : M[mk-Simplevarbn("int")](exenv);
def chans : dom c CHANS;
if chv∈chans
then (def chan : (c CHANS)(chv);
      if chan=<> then error
      else (let char = hdchan in
            let ind  = (if (∃i∈indsstr)(str[i]=char)
                        then (Λi∈indsstr)(str[i]=char∧
                                          (∀k∈{1:i-1})(str[k]≠char)))
                        else 0) in
            CHANS := c CHANS + [chv ↦ tlchan];
            assign(ind,int)))
else error
```

(3) In procedure *outchar*:

```
def chv   : contents(env("channel"));
let str   = env("str") in
def int   : contents(env("int"));
let char  = str(int)   in
def chans : dom c CHANS;
if chv∈chans then CHANS := c CHANS + [chv ↦ (c CHANS)(chv)^<char>]
else error
```

(4) In procedure *outterminator*: *s-body* contains:

```
def chv   : contents(env("channel"));
def chans : dom c CHANS;
if chv∈chans then CHANS := c CHANS + [chv ↦ (c CHANS)(chv) ^
                                        <implementation defined
                                         symbol, depending on the
                                         state of the channel>]
else error
```

Procedures *maxint*, *minreal*, *maxreal* and *epsilon* have bodies which return the appropriate implementation defined constants.

6.2 SCOPE DEFINITION

6.2.1 Blocks

```
Block :: Decl-set   Stmt*

WF[mk-Block(dcls,stl)](senv) ≜
   let labl = contndll(stl) in
   is-uniquel(labl) ∧
   is-disjointl(<elemslabl, {s-id(d) | d∈dcls}>) ∧
   (let lenv = [s-id(d) ↦ SPEC(d) | d∈dcls] ∪
               [lab ↦ LABEL | lab∈elemslabl] in
    let renv = senv \ domlenv in
    let nenv = senv + lenv    in
    (∀d∈dcls)((d∈Arraydecl ⊃ WF[d](renv)) ∧
              (d∈Proc      ⊃ WF[d](nenv)) ∧
              (d∈Switch    ⊃ WF[d](nenv))) ∧
    (∀st∈elemsstl)(WF[st](nenv)))
type Block → STATICENV → Bool
```

Comment It is important to notice the different (static) environments used to check parts of the *Block*. In particular, a reduced environment (*renv*) is used for *Arraydecl*s because any contained expressions should not contain references to local variables of the *Block* (to interpret such references in *senv* would be counter to the scope concept).

```
M[mk-Block(dcls,stl)](ownenv,env,cas) ≜
  let aid ∈(AID - cas) in
  def nenv : env +
          ([s-id(d) ↦ ownenv(s-oid(d))
                      | d∈dcls ∧ d∈(OwntypedeclUOwnarraydecl)]   U
           [s-id(d) ↦ M[d]                    | d∈dcls ∧ d∈Typedecl]   U
           [s-id(d) ↦ M[d](env,cas)           | d∈dcls ∧ d∈Arraydecl]  U
           [s-id(d) ↦ M[d](nenv)              | d∈dcls ∧ d∈Switch]     U
           [s-id(d) ↦ M[d](ownenv,nenv)       | d∈dcls ∧ d∈Proc]       U
           [lab     ↦ mk-LABELDEN(lab,aid)    | lab∈contndls(stl)]);
  let stenv = (ownenv,nenv,casU{aid}) in
  always epilogue({s-id(d) | d∈dcls ∧ d∈(Typedecl ∪ Arraydecl)})(nenv)
  in (tixe [mk-LABELDEN(tlab,aid) ↦ Mcue[tlab,stl](stenv)
                                  | tlab∈contndls(stl)] in
      for i=1 to lenstl do M[s-sp(stl[i])](stenv))
  type Block → STMTENV =>
```

6.2.2 Procedures

```
Proc      :: s-id :Id                    s-tp  :(Type | PROC)
             s-fpl:Id*                   s-vids:Id-set
             s-spm:Id ~m~> Specifier     s-body:(Block | Code)

Specifier = Type | Typearray | Typeproc |PROC|LABEL|STRING|SWITCH
Typearray :: Type
Typeproc  :: Type
Code          Implementation defined

WF[mk-Proc(id,tp,fpl,vids,spm,b)](senv) ≜
   is-uniquel(fpl) ∧ ¬(id∈elemsfpl)                          ∧
   vids⊆elemsfpl ∧ domspm=elemsfpl                           ∧
   (∀id∈vids)(spm(id)∈(Type∪Typearray∪{LABEL})) ∧
   WF[b](senv + spm)
type Proc → STATICENV → Bool
```

Comment the "Revised" version of ALGOL requires specification of all formals.

```
M[mk-Proc(id,tp,fpl,vids,spm,b)](oenv,env) ≜
  let f(denl,cas) =
    (if lendenl≠lenfpl ∨
        (∃i∈indsfpl)(¬is-parmmatch(denl[i],spm(fpl[i]),fpl[i]∈vids))
     then error
     else (def nenv : env +
                ([fpl[i] ↦ s-v(denl[i]) | i∈indsfpl ∧ ¬(fpl[i]∈vids)] ∪
                 [fpl[i] ↦ M[spm(fpl[i])](s-v(denl[i]),cas)
                         | i∈indsfpl ∧ fpl[i]∈vids]);
           def nfenv : if tp=PROC then nenv
                       else (def rloc : genscden();
                             nenv + [id ↦ mk-ATVFNDEN(rloc,env(id))]);
           M[b](oenv,nfenv,cas);
           epilogue({fpl[i] | i∈indsfpl ∧ fpl[i]∈vids ∧
                              spm(fpl[i])∈(Type∪Typearray)});
           def rv : if tp=PROC then nil
                    else (let mk-ATVFNDEN(rloc,) = nfenv(id) in
                          contents(rloc);
                          STR := c STR - {rloc});
           return(rv))) in f
type: Proc → (OWNENV × ENV) → PROCDEN
```

Comment note that the set of *AID*s used in the evaluation of the body is passed from the (dynamic) call.

Comment for a function procedure, *rloc* is the location into which assignments to the function name (for return) are made.

```
is-parmmatch(mk-Actparmden(den,spa),spf,bv) ≜
  if bv then
   (spa = spf)                                          ∨
   (spf∈Arithm∧spa∈Arithm)                              ∨
   (spf∈Typearray∧spa∈Typearray∧s-type(spf)∈Arithm∧s-type(spa)∈Arithm)
  else
   (spa=spf)                                            ∨
   (spf=PROC∧spa∈Typeproc)                              ∨
   (spf∈Typeproc∧spa∈Typeproc∧s-type(spa)∈Arithm∧s-type(spf)∈Arithm)
type  Actparmden × Specifier × Bool → Bool
```

Comment This check has to be dynamic because of the incomplete specification of arrays and procedures.

Comment The third parameter signifies whether the parameter passing is "by value" (*true*) or "by name" (*false*).

```
M : Specifier → (DEN × AID-set) => DEN
```

Comment Obtains value for "by value" actuals:

```
M[mk-Typearray(tp)](den,cas) ≜
    def aloc: genarrayden(domden);
    for all esscl∈domden do (def v: contents(den(esscl));
                             assign(v,aloc(esscl)));
    return (aloc)

M[tp](den,cas) ≜
    def v : if den∈BYNAMEEXPRDEN then den(cas)
            else (def l : den(cas); contents(l));
    let vc = conv(v,tp) in
    def l  : genscden();
    assign(vc,l);
    return(l)
```

```
M[LABEL](den,cas) ≜ den(cas)

M : Code → STMTENV =>
M[c](senv) ≜    Implementation defined
```

6.3 DECLARATIONS

```
Decl  =  Typedecl | Arraydecl   | Switchdecl  |
         Proc     | Owntypedecl | Ownarraydecl

Spec: Decl → Specifier  - obvious function
```

6.3.1 Type Declarations

```
Typedecl :: s-id:Id   s-desc:Type
Type     =  Arithm | BOOL
Arithm   =  INT | REAL

M[mk-Typedecl(id,tp)] ≜ genscden()
type Typedecl => TYPEDEN

genscden() ≜
   def ulocs: dom c STR;
   let l ∈ (SCALARLOC - ulocs) in
   STR := c STR ∪ [l ↦ NIL];
   return(l)
type => SCALARLOC

epilogue(ids)(env) ≜
   let scLocs = {env(id) |  id∈ids ∧ env(id)∈TYPEDEN} ∪
                union{rng(env(id)) | id∈ids ∧ env(id)∈ARRAYDEN} in
   STR := c STR\scLocs
type Id-set → ENV =>
```

6.3.2 Array Declarations

```
Arraydecl :: s-id:Id   s-tp:Type   s-bdl:Boundpair+
Boundpair :: s-lbd:Expr   s-ubd:Expr
```

The use of one <*bound pair list*> to define several <*array identifiers*>

is expanded by the translator. Notice that this can not be justified from *MAR* and, with side-effect producing function references in the bound pair list, is strictly wrong.

```
WF[mk-Arraydecl(,,bdl)](senv) Δ
 (∀bdp∈elemsbdl)
    (TP[s-lbd(bdp)](senv)∈Arithm ∧ TP[s-ubd(bdp)](senv)∈Arithm)

M[mk-Arraydecl(,tp,bdl)](exenv) Δ
   def ebds  : <M[bdl[i]](exenv) | 1≤i<lenbdl>;
   let indes = rect(ebds)  in
   def aloc  : genarrayden(indes);
   return(aloc)
type: Arraydecl → EXPRENV => ARRAYDEN

M[mk-Boundpair(lbd,ubd)](exenv) Δ
   def lbdv  : M[lbd](exenv);
   def ubdv  : M[ubd](exenv);
   let lbdvc : conv(lbdv,INT) in
   let ubdvc : conv(ubdv,INT) in
   if ubdvs<lbdvc then error else return(lbdvc,ubdvc)
type: Boundpair → EXPRENV => (Int × Int)
```

Comment The generation of an array with no elements (for a dimension) is defined here to be in error. This shows the error prior to reference;

```
genarrayden(inds) Δ
   def aloc: [indl ↦ genscden() | indl∈inds];
   return(aloc)
type: (Int+)-set => ARRAYDEN
```

Comment Assumes index set is dense.

Comment One-one property of denotation ensured.

```
rect: (Int × Int)+ → (Int+)-set
```

pre: No lower bound exceeds the corresponding upperbound.

Comment Generates the (dense) set of valid index lists.

6.3.3 Switch Declarations

```
Switchdecl :: s-id:Id   s-exl:Expr+

WF[mk-Switchdecl(,exl)](senv) ≜
    (∀ex∈elemsexl)(TP[ex](senv)=LABEL)
type: Switchdecl → STATICENV → Bool

M[mk-Switchdecl(,exl)](env) ≜
   let f(ind,cas) = (if 1≤ind≤lenexl
                     then M[exl[ind]](env,cas) else error)
   in f
type: Switchdecl → ENV → SWITCHDEN
```

Comment Notice that the expressions of the *Switchdecl* are evaluated when referenced (dynamically). The static environment (that of declaration) is used.

6.3.4 Procedure Declarations -- already treated in sect. 6.2.2

6.4 STATEMENTS

```
Stmt      :: s-lp:Id-set   s-sp:Unlabstmt
Unlabstmt = Compstmt | Block    | Assign | Goto    |
            Dummy    | Condstmt | For    | Procstmt
```

"Standard" types:

```
WF: Unlabstmt → STATICENV → Bool
M:  Unlabstmt → STMTENV   =>
```

6.4.1 Compound Statements

```
Compstmt :: Stmt*

M[mk-Compstmt(stl)](stenv) ≜
    for i=1 to lenstl do M[s-sp(stl[i])](stenv)
```

6.4.2 Blocks -- already treated in sect. 6.2.1

6.4.3 Assignment Statements

```
Assign   :: s-lp:Destin+   s-rp:Expr
Destin   :: s-tg:Leftpart   s-tp:Type
Leftpart  = Var | Atvfnid
Atvfnid  :: Id
```

Comment The return of a value from a function procedure is achieved by an assignment to a destination with the name of the function. Such names are called activated function identifiers (*atvfnid*).

```
WF[mk-Assign(dl,e)](senv) ≜
    let etp = TP[e](senv) in
    (∀i∈indsdl)(WF[dl[i],etp](senv))

WF[mk-Destin(lp,tp),etp](senv) ≜
    compattps(tp,etp) ∧
    (lp∈Var ∧ is-scalar(lp,senv) ∧ tp=TP[lp](senv) ∨
     lp∈Atvfnid ∧ mk-Typeproc(tp)=senv(s-id(lp)))
type: Destin × Type → STATICENV → Bool

TP[mk-Atvfnid(id)](senv) ≜ let mk-Typeproc(tp)=senv(id) in tp

M[mk-Assign(dl,e)](,env,cas) ≜
    def edl : <M[dl[i]](env,cas) | 1≤i≤lendl>;
    def v   : M[e](env,cas);
    for i=1 to lenedl do (let vc = conv(v,s-tp(dl[i])) in
                          assign(vc,edl[i]))
```

Comment Order of evaluation defined to resolve non-determinism.

```
M[mk-Destin(lp,tp)](exenv) ≜ M[lp](exenv)
type: Destin → EXPRENV => SCALARLOC
```

Comment The context conditions ensure that the location corresponding to a variable will be a member of *SCALARLOC*.

```
M[mk-Atvfnid(id)](env,cas) ≜ let mk-ATVFNDEN(loc,)=env(id) in loc
type: Atvfnid → EXPRENV → SCALARLOC
```

6.4.4 Goto Statements

```
Goto :: Expr

WF[mk-Goto(e)](senv)      ≜ TP[e](senv)=LABEL

M[mk-Goto(e)](,env,cas) ≜ def ld : M[e](env,cas); exit(ld)
```

The possibility of branching (by goto) into phrase structures in reflected by providing appropriate *Mcue* functions. These functions deliver denotations (of *TR*) which correspond to the execution of a phrase structure from some label to the end of (or abnormal exit from) the phrase. Notice that *tixe* is only used at the *Block* level in this definition.

```
Mcue[lab,mk-Stmt(labs,sp)](stenv) ≜
  if lab∈labs then M[sp](stenv) else Mcue[lab,sp](stenv)
type: Id × Stmt → STMTENV =>
pre:  lab ∈ contndls(mk-Stmt(labs,sp))

Mcue[lab,mk-Compstmt(stl)](stenv) ≜ Mcue[lab,stl](stenv)
type: Id × Compstmt → STMTENV =>

Mcue[lab,stl](stenv) ≜
  let i = index(lab,stl) in
  Mcue[lab,stl[i]](stenv);
  for j=i+1 to lenstl do M[s-sp(stl[j])](stenv)
type: Id × Stmt* → STMTENV =>
pre:  lab ∈ contndls(stl)

Mcue[lab,mk-Condstmt(,th,el)](stenv) ≜
  if lab∈contndls(th) then Mcue[lab,th](stenv)
  else Mcue[lab,el](stenv)
type: Id × Condstmt → STMTENV =>
pre:  lab ∈ (contndls(th) ∪ contndls(el))
```

6.4.5 Dummy Statements

```
Dummy :: DUMMY

M[mk-Dummy(DUMMY)](stmtenv) ≜ I_STATE
```

6.4.6 Conditional Statements

```
Condstmt :: s-test:Expr   s-th:Stmt   s-el:Stmt
```

The else statement is always present, if necessary the translator inserts a null statement.

```
WF[mk-Condstmt(e,th,el)](senv) ≜ TP[e](senv)=BOOL

M[mk-Condstmt(e,th,el)](stmtenv) ≜
    let (,env,cas) = stmtenv in
    def b          : M[e](env,cas);
    if b then M[s-sp(th)](stmtenv) else M[s-sp(el)](stmtenv)
```

6.4.7 For Statements

```
For          :: s-cv:Simplevar s-cvtp:Arithm s-fl:Forelem+ s-b:Block
Forelem      =  Exprelem | Whileelem | Stepuntilelem
Exprelem     :: Expr
Whileelem    :: s-in:Expr   s-wh:Expr
Stepuntilelem :: s-in:Expr  s-st:Expr   s-un:Expr
```

The body of the abstract form of a for statement is always a block; if not present in the concrete form it is generated by the translator.

```
WF[mk-For(cv,cvtp,fl,b)](senv) ≜
   is-scalar(cv,senv) ∧ cvtp=TP[cv](senv)

WF[mk-Exprelem(e)](senv) ≜ TP[e](senv)∈Arithm

WF[mk-Whileelem(in,wh)](senv) ≜
   TP[in](senv)∈Arithm ∧ TP[wh](senv)=BOOL

WF[mk-Stepuntilelem(in,st,un)](senv) ≜
   TP[in](senv)∈Arithm ∧ TP[st](senv)∈Arithm ∧ TP[un](senv)∈Arithm

M[mk-For(cv,cvtp,flel,b)](stenv) ≜
   for i=1 to lenflel do M[flel[i],cv,cvtp,b](stenv)
```

Types for remainder of this sub-section:

```
M: Forelem × Var × Type × Block → STMTENV =>
```

```
M[mk-Exprelem(e),cv,cvtp,b](stmtenv) Δ
    let (,env,cas) = stenv          in
    def v          : M[e](env,cas);
    let vc         = conv(v,cvtp)   in
    def l          : M[cv](env,cas);
    assign(vc,l);
    M[b](stmtenv)

M[mk-Whileelem(in,wh),cv,cvtp,bd](stmtenv) Δ
    let (,env,cas) = stmtenv        in
    def v          : M[in](env,cas);
    let vc         = conv(v,cvtp)   in
    def l          : M[cv](env,cas);
    assign(vc,l);
    def b          : M[wh](env,cas);
    if b then (M[bd](stmtenv);
               M[mk-Whileelem(in,wh),cv,cvtp,bd)(stmtenv))
    else I_STATE
```

Comment Re-evaluation of initial expression (*in*) required by language.

```
M[mk-Stepuntilelem(in,st,un),cv,cvtp,bd](stenv) Δ
    let (,env,cas) = stenv          in
    let exenv      = (env,cas)      in
    def vin        : M[in](exenv);
    let vinc       = conv(vin,cvtp) in
    def l          : M[cv](exenv);
    assign(vinc,l);
    step(st,un,cv,cvtp,bd)(stenv)

step(st,un,cv,cvtp,b)(stenv) Δ
    let (,env,cas) = stenv          in
    let exenv      = (env,cas)      in
    def vst        : M[st](exenv);
    def b          : (M["cv-un"](exenv)*sign(vst))≤0;
    if b then (M[b](stenv);
               def l      : M[cv](exenv);
               def vcur   : contents(l)+vst;
               let vcurv  = conv(vcur,cvtp) in
               assign(vcurv,l);
               step(st,un,cv,cvtp,b)(stenv))
    else I_STATE
```

6.4.8 Procedure Statements

```
Procstmt    :: (Procdes | Functdes)
Procdes     :: s-pn:Id   s-app:Actparm*
Actparm     :: s-v:Actparmval   s-tp:Specifier
Actparmval  =  Parmexpr | Arrayname | Switchname | Procname | String
Parmexpr    :: Expr
Arrayname   :: Id
Switchname  :: Id
Procname    :: Id
String      :: Char*
Char           Implementation defined set
```

```
WF[mk-Procdes(id,apl)](senv) ≜ env(id)=PROC
type: Procdes → STATICENV → Bool
```

```
WF[mk-Actparm(v,sp)](senv) ≜
   TP[v](senv) = sp                                  ∧
   (sp∈Type                    ⊃ v∈Parmexpr)         ∧
   (sp∈Arraytype               ⊃ v∈Arrayname)        ∧
   (sp∈(Typeproc ∪ {PROC}      ⊃ v∈Procname)         ∧
   (sp=LABEL                   ⊃ v∈Parmexpr)         ∧
   (sp=STRING                  ⊃ v∈String)           ∧
   (sp=SWITCH                  ⊃ v∈Switchname)
```

TP: *Actparmval* → *STATICENV* → (*Type* | LABEL) Similar to *TP* of *Expr*

```
M[mk-Procstmt(des)](,env,cas) ≜
   if des∈Procdes then M[des](env,cas)
   else (def v : M[des](env,cas); I_STATE)
```

Comment When a function procedure is invoked by a *Procstmt* the returned value is discarded

```
M[mk-Procdes(id,apl)](env,cas) ≜
   def denl : <M[apl[i]](env) | i<i<lenapl>;
   let f    = env(id) in
   f(denl,cas)
type: Procdes → EXPRENV => VAL
```

M[mk-Actparm(v,tp)](env) $\triangleq$
let d = M[v](env) in mk-Actparmden(d,tp)
type: Actparm → ENV → DEN

The rest of the functions in this section are of type:

M: Actparmval → ENV → DEN

M[mk-Parmexpr(e)](env) $\triangleq$
if e∈Varref then (let f(cas)=M[e](env,cas) in mk-BYNAMELOCDEN(f))
else (let f(cas) = M[e](env,cas) in mk-BYNAMEEXPRDEN(f))

Comment The calling information does not determine whether "byname" or "byvalue" mode is to be used so the "byname" parameters are genrated and evaluated when necessary within the called procedure. The distinction between whether evaluation can be used to determine a location is, however, made here.

M[mk-Arrayname(id)](env) $\triangleq$ *env(id)*

M[mk-Switchname(id)](env) $\triangleq$ *env(id)*

M[mk-Procname(id)](env) $\triangleq$ *env(id)*

M[mk-String(s)](env) $\triangleq$ *s*

6.5 EXPRESSIONS

Expr = Typeconst | Varref | Labelconst | Switchdes |
Functdes | Prefixexpr | Infixexpr | Condexpr

Comment The order of evaluation, which is governed in the concrete syntax by operator priority and parenthesis, is captured by the structure of abstract expressions.

Comment The various forms of expressions have been brought together in one abstract syntax. Context conditions are defined which, for example, limit the form of label expressions to use conditionals (there are no infix or prefix operators on *LABELDEN*).

Within this section (unless otherwise indicated):

WF: Expr → STATICENV → Bool
TP: Expr → STATICENV → (Type | LABEL)
M: Expr → EXPRENV => VAL

6.5.1 Arithmetic Constants

Typeconst :: Boolconst | Arithmconst
Boolconst :: Bool
Arithmconst = Realconst | Intconst
Realconst :: Real
Intconst :: Int

Comment The abstract syntax contains the semantic objects rather than their representations.

TP[mk-Boolconst(bv)](senv) = BOOL
TP[mk-Realconst(rv)](senv) = REAL
TP[mk-Intconst(iv)](senv) = INT

M[mk-Boolconst(bv)](exenv) $\triangleq$ *bv*
M[mk-Realconst(rv)](exenv) $\triangleq$ *represent(rv)*
M[mk-Intconst(iv)](exenv) $\triangleq$ *test(iv)*

6.5.2 Variable References

Varref :: Var
Var = Simplevar | Subscrvar
Simplevar = Simplevarbn | Simplevarv
Simplevarbn :: s-nm:Id
Simplevarv :: s-nm:Id
Subscrvar = Subscrvarbn | Subscrvarv
Subscrvarbn :: s-nm:Id s-sscl:Subscrs
Subscrvarv :: s-nm:Id s-sscl:Subscrs
Subscrs :: Expr$^+$

Comment The distinction is made between references to "byname" (*bn*) formal parameters and values (*v*). The latter class includes "byvalue" formal parameters and normal variables.

WF[mk-Simplevarbn(id)](senv) $\triangleq$ *senv(id)*$\in$*Type*
WF[mk-Simplevarv(id)](senv) $\triangleq$ *senv(id)*$\in$*Type*

```
WF[mk-Subscrvarbn(id,sscl)](senv) Δ
    senv(id)∈Typearray ∧ (∀i∈indssscl)(TP[sscl[i]](senv)∈Arithm)

WF[mk-Subscrvarv(id,sscl)](senv) Δ
    senv(id)∈Typearray ∧ (∀i∈indssscl)(TP[sscl[i]](senv)∈Arithm)
```

Comment The check that "byname" references are used exactly for "by-name" parameters is not shown formally. To do so would require a minor extension to the *STATICENV*.

```
TP[mk-Simplevarbn(id)](senv) Δ senv(id)
TP[mk-Simplevarv(id)](senv)  Δ senv(id)

M[mk-Varref(var)](env,cas) Δ
    if var∈(SimplevarvUSubscrvarv) ∨ env(s-nm(var))∈BYNAMELOCDEN
       then (def l    : M[var](env,cas);  contents(l))
       else (let bned = env(s-nm(var)) in bned(cas))
```

The remainder of the functions in this sub-section are of type:

```
M: Var → EXPRENV => LOC

M[mk-Simplevarbn(id)](env,cas) Δ
    let bnd = env(id) in
    if bnd∈BYNAMELOCDEN then bnd(cas) else error

M[mk-Simplevarv(id)](env,) Δ env(id)

M[mk-Subscrvarbn(id,sscl)](env,cas) Δ
    def esscl : M[sscl](env,cas);
    let bnd   = env(id) in
    if bnd∈BYNAMELOCDEN
       then (def aloc : bnd(cas);
             if esscl∈domaloc then return(aloc(esscl)) else error)
       else error

M[mk-Subscrvarv(id,sscl)](env,cas) Δ
    def esscl : M[sscl](env,cas);
    let aloc  = env(id) in
    if esscl∈domaloc then return(aloc(esscl)) else error
```

```
M[mk-Subscrs(sscl)](exenv) Δ
   def esscl: <(def essc  : M[sscl[i]](exenv);
                let esscv = conv(essc,INT)  in
                esscv) | 1≤i≤lensscl>;
   return(esscl)
type: Subscrs → EXPRENV => Int+
```

Comment Order of evaluation defined to resolve non-determinacy

6.5.3 Label Constants

```
Labelconst :: Id

WF[mk-Labelconst(id)](senv)   Δ senv(id)=LABEL

TP[mk-Labelconst(id)](senv)   Δ LABEL

M[mk-Labelconst(id)](env,cas) Δ env(id)
```

6.5.4 Switch Designators

```
Switchdes :: s-id:Id   s-ssc:Expr

WF[mk-Switchdes(id,ssc)](senv) Δ
   senv(id)=SWITCH ∧ TP[ssc](senv)∈Arithm

TP[mk-Switchdes(id)](senv) Δ LABEL

M[mk-Switchdes(id,ssc)](env,cas) Δ
   def essc  : M[ssc](env,cas);
   let esscv = conv(essc,INT) in
   let swden = env(id)        in
   def ld    : swden(esscv,cas);
   return(ld)
```

6.5.5 Function Designators

```
Functdes :: s-nm:Id   s-app:Actparm*

WF[mk-Functdes(id,app)](senv) Δ senv(id)∈Typeproc
```

```
TP[mk-Functdes(id,app)](senv) Δ s-type(senv(id))

M[mk-Functdes(id,app)](env,cas) Δ
    def denl: <M[app[i]](env) | 1≤i≤lenapp>;
    let f   = env(id)   in
    def v   : f(denl,cas);
    return(v)
```

6.5.6 Prefix Expressions

```
Prefixexpr :: s-opr:Prefixopr   s-op:Expr
Prefixopr  = REALPLUS | REALMINUS | INTPLUS | INTMINUS | NOT
```

Comment Type information is inserted into the tree form of expressions, this permits the insertion of appropriate operators.

```
WF[mk-Prefixexpr(opr,expr)](senv) Δ
    if opr=NOT then TP[expr](senv)=BOOL
    else TP[expr](senv)∈Arithm

TP[mk-Prefixexpr(opr,expr)](senv) Δ
    if opr=NOT then BOOL
    else if opr∈{REALPLUS,REALMINUS} then REAL else INT

M[mk-Prefixexpr(opr,expr)](exenv) Δ
    def v: M[expr](exenv);
    M[opr](v)
```

using:

```
M: Prefixopr → SCALARVAL => SCALARVAL

M[NOT](v)       Δ ¬v
M[REALPLUS](v)  Δ v
M[INTPLUS](v)   Δ v
M[REALMINUS](v) Δ -v
M[INTMINUS](v)  Δ -v
```

6.5.7 Infix Expressions

```
Infixexpr :: s-op1:Expr   s-opr:Infixopr   s-op2:Expr
```

Infixopr = REALADD | REALSUB | REALMULT | REALDIV |
INTADD | INTSUB | INTMULT | INTDIV |
REALEXP | REALINTEXP | INTEXP |
LT | LE | EQ | NE | GE | GT |
IMPL | EQUIV | AND | OR

WF[mk-Infixexpr(e1,opr,e2)](senv) ≜
let tp1 = TP[e1](senv) in
let tp2 = TP[e2](senv) in
(opr∈{REALADD,REALSUB,REALMULT}∧ REAL∈{*tp1,tp2*}⊆*Arithm* ∨
opr=REALDIV∧{*tp1,tp2*}⊆*Arithm* ∨
opr=REALEXP∧*tp1*∈*Arithm*∧*tp2*=REAL ∨
opr=REALINTEXP∧*tp1*=REAL∧*tp2*=INT ∨
opr∈{INTADD,INTSUB,INTMULT,INTDIV,INTEXP}∧*tp1*=*tp2*=INT ∨
opr∈{LT,LE,EQ,NE,GE,GT}∧{*tp1,tp2*}⊆*Arithm* ∨
opr∈{IMPL,EQUIV,AND,OR}∧*tp1*=*tp2*=BOOL)

TP[mk-Infixexpr(e1,opr,e2)](senv) ≜
opr∈{INTADD,INTSUB,INTMULT,INTDIV,INTEXP} → INT,
opr∈{REALADD,REALSUB,REALMULT,REALDIV,REALEXP,REALINTEXP} → REAL,
T → BOOL

M[mk-Infixexpr(e1,opr,e2)](exenv) ≜
def v1: M[e1](exenv);
def v2: M[e2](exenv);
M[opr](v1,v2)

Comment Evaluation is shown here as being left to right in order to resolve the non-determinism.

Using:

M: Infixopr → (SCALARVAL × SCALARVAL) => SCALARVAL

*M[*REALADD*](v,w)* ≜ *represent(v+w)*

*M[*REALSUB*](v,w)* ≜ *represent(v-w)*

*M[*REALMULT*](v,w)* ≜ *represent(v*w)*

*M[*REALDIV*](v,w)* ≜ *if w=0 then fault1 else represent(v*represent(1/w))*

M[REALEXP](v,w) Δ *if* v>0 *then* value of the standard function *exp* applied on *represent*(v * value of the standard function *ln* applied on w)
else if v=0∧w>0 *then* 0 *else* *fault2*

M[REALINTEXP](v,w) Δ
if v=0 ∧ w=0 *then* *fault3*
else (*let* *expn*(n) = *if* n=0 *then* 1 *else* *represent*(*expn*(n-1)*v) *in*
if w>0 *then* *expn*(w) *else* *represent*(1/*expr*(-w)))

M[INTADD](v,w) Δ *test*(v+w)

M[INTSUB](v,w) Δ *test*(v-w)

M[INTMULT](v,w) Δ *test*(v*w)

M[INTDIV](v,w) Δ *if* w=0 *then* *fault1* *else* *test*(v/w)

M[INTEXP](v,w) Δ *if* w<0 ∨ v=0 ∧ w=0 *then* *fault4*
else (*let* *expi*(n) = *if* n=0 *then* 1
else *text*(*expi*(n-1)*v) *in*
return(*expi*(w)))

M[LT](v,w) Δ *return*(v < w)
M[LE](v,w) Δ *return*(v ≤ w)
M[EQ](v,w) Δ *return*(v = w)
M[NE](v,w) Δ *return*(v ≠ w)
M[GE](v,w) Δ *return*(v ≥ w)
M[GT](v,w) Δ *return*(v > w)
M[IMPL](v,w) Δ *return*(v ⊃ w)
M[EQUIV](v,w) Δ *return*(v = w)
M[AND](v,w) Δ *return*(v ∧ w)
M[OR](v,w) Δ *return*(v ∨ w)

Comment *fault1* corresponds to *fault*('division by zero',v)
fault2 corresponds to *fault*('expr undefined',v)
fault3 corresponds to *fault*('expn undefined',v)
fault4 corresponds to *fault*('expi undefined',v)

6.5.8 Conditional Expressions

Condexpr :: s-dec:Expr s-th:Expr s-el:Expr

WF[mk-Condexpr(dec,th,el)](senv) $\triangleq$
  TP[dec](senv)=BOOL $\wedge$ *is-compattps(TP[th](senv),TP[el](senv))*

is-compattps(tp1,tp2) $\triangleq$ *tp1=tp2* $\vee$ {*tp1,tp2*}$\subseteq$*Arithm*
type: (Type|LABEL*)* $\times$ *(Type*|LABEL*)* $\rightarrow$ *Bool*

TP[mk-Condexpr(dec,th,el)](senv) $\triangleq$
  let tp1 = TP[th](senv) in
  let tp2 = TP[el](senv) in
  if tp1=tp2 then tp1 else REAL

M[mk-Condexpr(dec,th,el)](exenv) $\triangleq$
  def decv : M[dec](exenv);
  if decv then M[th](exenv) else M[el](exenv)

6.6 AUXILIARY FUNCTIONS

is-uniqueoids: Block $\rightarrow$ *Bool*

checks that the *oid* components of any pair of *Owntypedecl* or *Ownarraydecl within* the block are distinct from each other.

is-constownbds: Block $\rightarrow$ *Bool*

checks that all expressions in the *s-bdl* component of *Ownarraydecl*'s *within* the block are integer constants.

is-within: Phrase $\times$ *Block* $\rightarrow$ *Bool*

tests whether a phrase (of any syntactic class) is contained (at any nesting depth) within a block.

is-scalar(v)(senv) $\triangleq$
  env(s-nm(v))$\in$*Type* $\vee$ *v*$\in$*Subscrvar* $\wedge$ *env(s-nm(v))*$\in$*Typearray*

contndls: Stmt → Id-set*
contndll: Stmt → Id**

Two obvious functions for gathering the labels of a list of statements (labels within nested blocks are not collected). The list version is required to preserve duplicates so that a test for redefinition can be made:

index: Id × Stmt ⇸ Nat*

For identifiers in the *contndls*, this function finds the index of that statement which contains the identifier as a label.

is-uniquel: Object → Bool* -- *true* iff no duplicates

is-disjointl: (Object-set) → Bool* -- *true* iff sets are pairwise disjoint

conv(v,tp) ≜ *if tp=*INT *then test(*"rounded value of v"*) else v*
type: Scalar × Type → SCALARVAL
*pre: (v∈Bool ⊃ tp=*BOOL*) ∧ (v∈Real ⊃ tp∈Arithm)*

contents(l) ≜ *def v:(c* STR*)(l); if v=nil then error else return(v)*
type: Scalar => SCALARVAL

assign(v,l) ≜ STR := *c* STR + [*l ↦ v*]
type: SCALARVAL × SCALARLOC =>

test(i) ≜ *if* -MAXINT<i<MAXINT *then return(i) else error*
type: Int => Int

represent(r) ≜ *if -MAXREAL<r<-MINREAL ∧ MINREAL<r<MAXREAL ∨ r=0*
then return("implementation defined approximation to *r"*)
else error
type: Real => Real

compattps: Type × Type → Bool, *true* if types compatible for assignment

6.7 SUPPORT INFORMATION

This section offers two aids to reading the ALGOL definition.

6.7.1 Abbreviations

Object names have been abbreviated systematically as shown below. (Local values within functions are further abbreviated.)

*Abn*ormal component
*act*ual
activation *id*entifier
*activ*ated
*arithm*etic
*assign*ment
*by-n*ame
*Bool*ean
*by-v*alue
*char*acter
*comp*ound
*cond*itional
*const*ant
*decl*aration
*den*onation
*des*ignator
*descr*iptor
*dest*ination
*elem*ent
*env*ironment
*expr*ession
*func*tion
*id*entifier
*int*eger
*oper*ator
*param*eter
*proc*edure
*ref*erence
*spec*ification
*stat*ement
*subscr*ipted
*trans*formation
*unlab*elled
*val*ue
*var*iable

6.7.2 Index

This index includes objects and functions. The functions *(M, TP, WF)* are defined by cases on their phrases (Split), and can be found by locating the phrase.