N.W.P. van Diepen

# Algebraic specification of a language with goto-statements

Computer Science/Department of Software Technology Note CS-N8601 February

69D21, 69D31, 69F31, 69F32

# Algebraic Specification of a Language with Goto-Statements

N.W.P. van Diepen
*Centre for Mathematics and Computer Science*
*P.O. Box 4079, 1009 AB Amsterdam, The Netherlands*

The algebraic specification of the semantics of SMALL - a programming language designed to demonstrate specifications in denotational semantics - is given. Focus of attention are the specification of the semantics of goto-statements and the modular build-up of a language specification.

*1980 Mathematics Subject Classification:* 68Bxx [software]
*1983-84 CR Categories:* D.2.1 SOFTWARE ENGINEERING: Requirements, Specifications; D.3.1 PROGRAMMING LANGUAGES: Formal Definitions and Theory; F.3.1 LOGICS AND MEANINGS OF PROGRAMS: Specifying and Verifying and Reasoning about Programs; F.3.2 LOGICS AND MEANINGS OF PROGRAMS: Semantics of Programming Languages - Algebraic Approaches to Semantics.

*Note 1:* Partial support has been received from the European Communities under ESPRIT project no. 348 (Generation of Interactive Programming Environments).
*Note 2:* This paper will be presented at the fourth *NGI-SION Voorjaarssymposium*, Utrecht, april 2 and 3, 1986.

## 1. INTRODUCTION

Automatic generation of a programming environment for a programming language requires the description of that language in a formal way. Progress has been made in the field of *algebraic specification* of programming languages. Bergstra, Heering & Klint [1] have described the toy programming language PICO (the language of **while**-programs) in detail. Their specification gives a complete parser, type-checker and interpreter for PICO-programs. PICO's small supply of language constructs leaves room for investigation in the specification of the semantics of more involved statement types.

The language to be specified in this paper is SMALL, designed by Gordon [5] as an example language to show specifications in *denotational semantics*. SMALL is built in layers to allow one to concentrate on the difficulties in specification of a certain language construct while others are excluded. The already available specification by Gordon in denotational semantics may lead to a comparison between this specification formalism and an algebraic specification. In particular we are interested in the way **goto**'s are defined in both formalisms: the denotational definition uses continuations (i.e. higher order functions) for this purpose while our algebraic formalism is restricted to first order functions. The freedom allowed by **goto**-statements makes it one of the most difficult classical programming primitives to specify. Its a-structural semantics turned it already into a controversial construct [6]. We are also interested in the question how to capture the various layers of SMALL in one, modular, definition.

The next section describes the syntax and (informally) the semantics of the SMALL kernel language (SMALL proper), followed by the syntax and semantics of an extension with **goto**-statements. An algebraic specification of the semantics of the kernel language is given in section 3, both to give an idea of algebraic specifications of languages and to provide a basis for section 4, a specification of the extension with **goto**'s.

Note CS-N8601
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

An algebraic specification consists of *sorts*, *functions* on these sorts (a constant is a 0-ary function), and *equations* (which may contain *variables* over the sorts) describing the relations between functions. In the specification formalism used the choice is made for a modular approach. Hence some mechanisms are available to formalize inclusion and parameterization of modules. A module can have an *export* section, containing all sorts, functions and constants available to the outside and a section with only *locally* visible definitions. It can also have an *import* section. All exports from the imported module are available in the importing module. Lastly a module can have *parameters* to which objects can be bound upon import.

The algebraic specification formalism used is described in detail in [1]. For more detail on algebraic specifications see [2].

## 2. Syntax and informal semantics of Small

### *2.1. Syntax*

The syntax of Small is given below in standard BNF-notation. Note that the primitive notions `<basic-value>`, `<identifier>` and `<binary-operator>` are left unspecified.

In this concrete syntax it is ambiguous which commands belong to the body of higher-level commands (e.g. where the body of a **while**-loop ends) since no delimiters are given. In the abstract syntax tree (which will be our starting point) this ambiguity is resolved and the unspecified notions will turn up as primitive nodes.

```
<program>     ::= 'program' <command> .
<command>     ::= <expression> ':=' <expression> |
                  'output' <expression> |
                  'if' <expression> 'then' <command> 'else' <command> |
                  'while' <expression> 'do' <command> |
                  'begin' <declaration> ';' <command> 'end' |
                  <command> ';' <command> .
<expression>  ::= <basic-value> | 'true' | 'false' |
                  'read' | <identifier> |
                  <expression> '(' <expression> ')' |
                  'if' <expression> 'then' <expression> 'else' <expression> |
                  <expression> <binary-operator> <expression> .
<declaration> ::= 'const' <identifier> '=' <expression> |
                  'var' <identifier> '=' <expression> |
                  'proc' <identifier> '(' <identifier> ');' <command> |
                  'fun' <identifier> '(' <identifier> ');' <expression> |
                  <declaration> ';' <declaration> .
```

Small will be augmented with **goto**-constructs in section 4. The syntax will be enlarged as follows:

```
<command>     ::= ...
                  'goto' <identifier> |
                  <identifier> ':' <command> .
<declaration> ::= ...
                  'label' <identifier> .
```

### 2.2. *Informal semantics and Abstract Syntax Trees*

*2.2.1. Basic values, identifiers and operators.* Some primitive notions are needed to give a basis to the operations of a programming language. Firstly the module `Booleans` with `true`, `false` and a few functions is needed. Further notions are treated abstractly and are grouped into one module: `SMALL-Primitives`, containing the sorts `BASICVAL`, `IDNT` and `BINOP`, together with an equality function on `IDNT` yielding a boolean (hence `Booleans` has to be imported).

*2.2.2. Abstract syntax and informal semantics of the kernel language.* The constructor functions for the abstract syntax are combined in module `SMALL-Abs-Synt`. The sorts `DECL`, `DECLS`, `EXPR`, `CMND`, `CMNDS` and `PROGRAM` are defined here and the modules `Booleans` and `SMALL-Primitives` are imported.

The following constructor functions are defined:

`<program>` constructor:

`program: CMNDS -> PROGRAM`
: turning a series of commands into a SMALL program.

`<command>` constructors:

`abs-assign: EXPR # EXPR -> CMND`
: The first expression should yield an identifier, to which the value of the second is assigned.

`abs-output: EXPR -> CMND`
: The value of the expression will be added to the output. Output and input are treated abstractly.

`abs-proccall: EXPR # EXPR -> CMND`
: The first expression gives the identifier of the procedure, the second one gives the value of the (single) parameter. Every procedure and function in SMALL has exactly one parameter.

`abs-if: EXPR # CMNDS # CMNDS -> CMND`
: The expression should yield a Boolean value and the series of commands the then- and else-branches.

`abs-while: EXPR # CMNDS -> CMND`
: The expression should yield a Boolean value and the commands the loop-body.

`abs-block: DECLS # CMNDS -> CMND`
: A block consists of a list of declarations and a list of commands.

`abs-ser: CMND # CMNDS -> CMNDS`

`abs-skip: -> CMNDS`
: Sequential composition of commands with terminal constant.

`<expression>` constructors:

`absexp-basicval: BASICVAL -> EXPR`
: Expression consisting of a basic value.

`absexp-read: -> EXPR`
: To read a value.

`absexp-ident: IDNT -> EXPR`
: Expression consisting of a single identifier.

`absexp-funcall: EXPR # EXPR -> EXPR`
: The first expression yields the name of the function, the second the parameter value.

`absexp-ifexp: EXPR # EXPR # EXPR -> EXPR`
: The first expression yields a Boolean, the second and third form the then- and else-clause respectively.

`absexp-binop: BINOP # EXPR # EXPR -> EXPR`
: The expressions yield the left- and righthand operand of the binary operator.

`<declaration>` constructors:

`absdecl-const: IDNT # EXPR -> DECL`

The identifier yields the new name and the expression the value.

`absdecl-var: IDNT # EXPR -> DECL`
: The initialization value will be yielded by the expression.

`absdecl-proc: IDNT # IDNT # CMNDS -> DECL`
: The first identifier yields the name of the procedure, the second identifier the parameter, and the series of commands the body.

`absdecl-fun: IDNT # IDNT # EXPR -> DECL`
: As above with an expression for the body.

`absdecl-ser: DECL # DECL -> DECL`

`absdecl-skip: -> DECL`
: The constructor to combine declarations and the corresponding terminal constant.

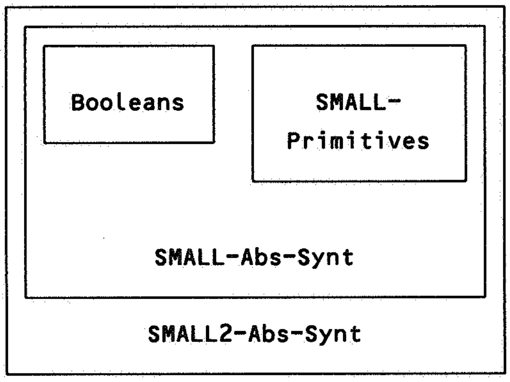
*2.2.3. Abstract syntax and informal semantics of the extension with goto's.* To enrich the SMALL abstract syntax with **goto**'s a module `SMALL2-Abs-Synt` is layered around `SMALL-Abs-Synt`. It contains three additional constructor functions:

`abs-goto: IDNT -> CMND`
: The jump is to the *last* label with this name in the block in which the label is declared. Jumps into an inner block or a procedure are illegal, jumps out of a procedure or an inner block are allowed. Jumps into the body of loops continue with the rest of the body followed by the whole loop and the rest of the program.

`abs-labldcmnd: IDNT # CMND -> CMND`
: The identifier is the label of the command.

`absdecl-label: IDNT -> DECL`
: The identifier gives the name of a new label.

The following structure diagram (see [1]) shows the import relationship between the modules discussed above.



## 3. ALGEBRAIC SEMANTICS OF THE SMALL KERNEL

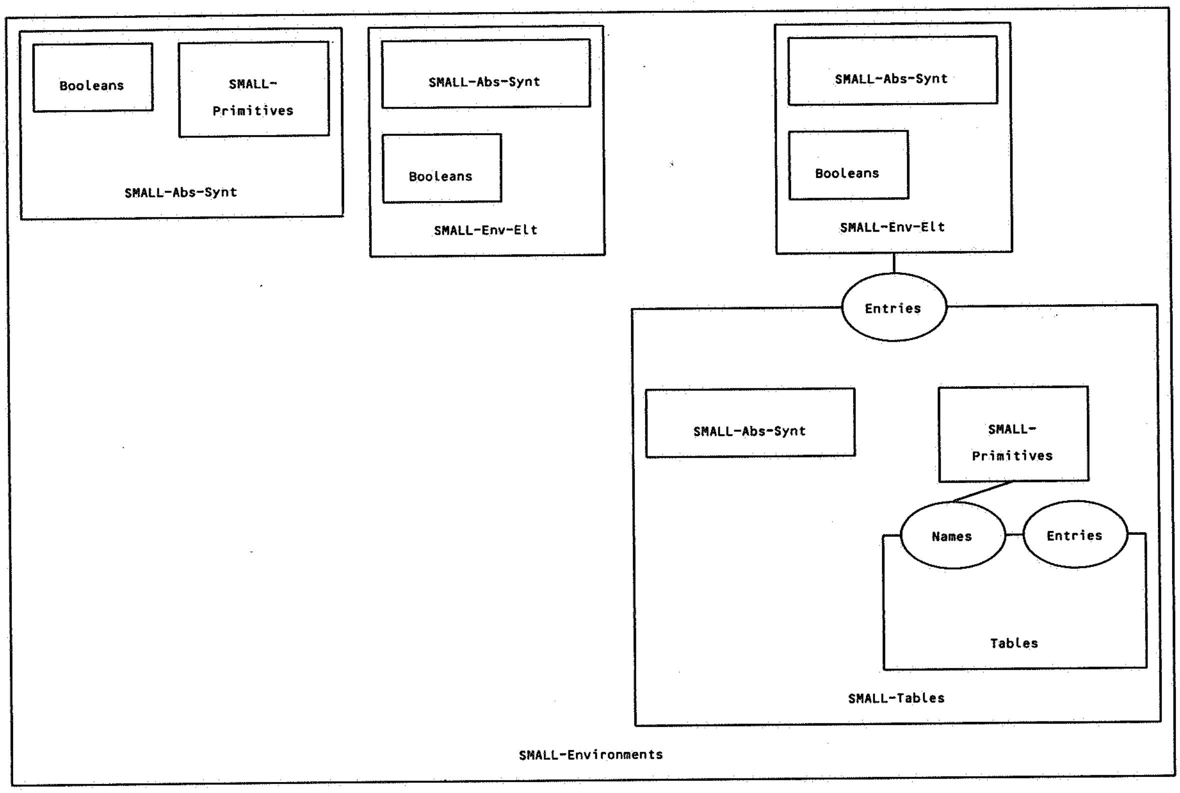
### *3.1. Environment specification*

To manipulate entities necessary to describe the behaviour of a SMALL program an abstract storage mechanism is needed. The basis for this storage mechanism is the sort `TABLE`, essentially a stack-like structure with two parameters: `Names` and `Entries`. The functions `nulltable` (generates an empty table), `tableadd` (puts a fresh *name-entry* combination in a table), `tablech` (changes an entry corresponding to a given name) and `lookup` (returns `true` and the found entry or `false` and the `error-entry` for a given name and table) are given. Of course an equality check must be given on the names. These sorts and functions are bundled in module `Tables`.

SMALL has a block structure (as in e.g. Pascal). The elementary storage mechanism provided by

Tables does not provide sufficient power to capture this structure in an easy way. Hence a new module SMALL-Tables is tailored for this task around Tables. A constant blockmark is introduced to separate blocks on a table. This constant is of a new sort, TABLEMARK. Of course a function removeblock is defined. Finally NAME is bound to IDNT (from SMALL-Primitives) since we are focusing on SMALL anyway.

Next the objects we want to put into the table have to be bound to entries from Entries. Since this comprises objects of various sorts (e.g. declarations and basic values) an intermediary module SMALL-Env-Elt is constructed to provide a common sort, called ENVELT (environment-element), and injection functions into this sort.

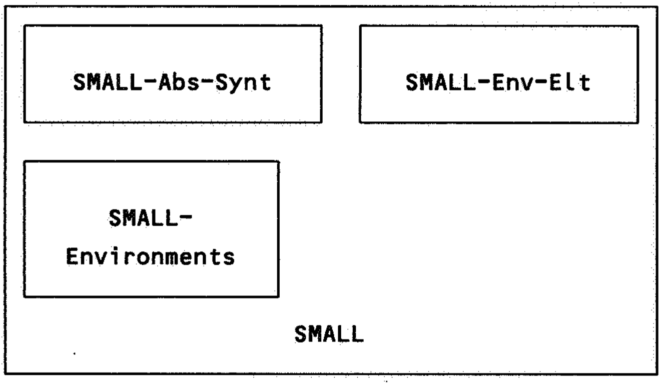


These modules are combined in module SMALL-Environments. It imports SMALL-Tables - with Entries bound by SMALL-Env-Elt and TABLE renamed to SENV - and SMALL-Abs-Synt and SMALL-Env-Elt. The structure diagram gives a schematic impression of the import relationship. The ellipses indicate parameters and lines drawn from them indicate the binding of these parameters to modules. The parameter Entries is not bound in SMALL-Tables, so it becomes a parameter of this module.

## *3.2. Semantical specification*

The algebraic specification of the semantics of the SMALL kernel language is quite straightforward. See the accompanying structure diagram below.

In this specification the work is mainly carried out by evaluation functions for the elementary language constructs. eval is given either a program and input or a series of commands and an environment. evalexpr operates on an expression and an environment and evaldecl on a declaration or a series of declarations and an environment. The environment resulting from a correct evaluation contains the output and the (possibly exhausted) input. Giving more detail on I/O handling is not of interest to the topics treated here.



An auxiliary constant `abs-blockend` is introduced to mark the end of the series of commands forming a block in the series of commands to be executed. The auxiliary function `cat` is necessary to join series of commands.

Note that in equation 4 a block is constructed around a procedure containing the initialization of the parameter. A similar construction is used in equation 14 to store the parameter of a function call. In equation 8 a block is created in the environment, and in equation 9 it is removed again.

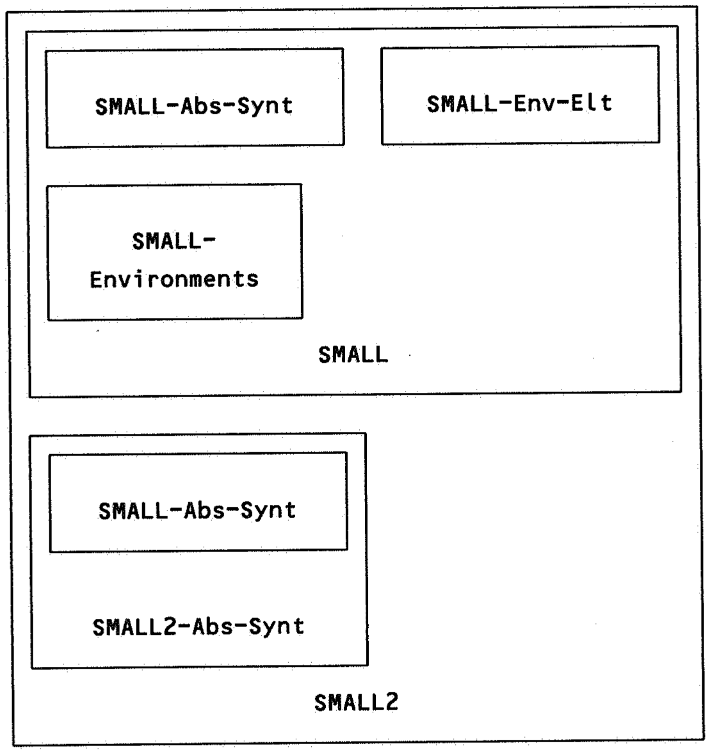
```
module SMALL
begin
exports
    begin
    functions
        eval       : PROGRAM # ENVELT             ->  SENV
        eval       : CMNDS # SENV                 ->  SENV
        evaldecl   : DECL # SENV                  ->  SENV
        evaldecl   : DECLS # SENV                 ->  SENV
        evalexpr   : EXPR # SENV                  -> (BASICVAL # SENV)

        applyfun   : IDNT # BASICVAL # SENV       -> (BASICVAL # SENV)
        applybinop : BINOP # BASICVAL # BASICVAL ->  BASICVAL

        abs-blockend :                -> CMND
        cat          : CMNDS # CMNDS -> CMNDS

        in  : -> IDNT
        out : -> IDNT
    end
imports SMALL-Abs-Synt, SMALL-Env-Elt, SMALL-Environments

variables dcl                       : -> DECL
          dcls                      : -> DECLS
          exp, exp1, exp2           : -> EXPR
          cmd                       : -> CMND
          cmds, cmds1, cmds2        : -> CMNDS
          senv, senv1, senv2        : -> SENV
          bval, bval1, bval2        : -> BASICVAL
          idnt, idnt1, name, param : -> IDNT
        oper                    : -> BINOP
        entry, input            : -> ENVELT
        bool                    : -> BOOL

equations
   [1] eval(program(cmds),input)
             = eval(cmds, tableadd(out,error-value,
                          tableadd(in,input,null-senv)))
   [2] eval(abs-ser(abs-assign(exp1,exp2),cmds),senv)
             = eval(cmds,tablech(idnt,envelt(bval),senv2))
               when <bval,senv1>           = evalexpr(exp2,senv),
                    <basicval(idnt),senv2> = evalexpr(exp1,senv1)
   [3] eval(abs-ser(abs-output(exp),cmds),senv)
             = eval(cmds,tablech(out,cat(entry,bval),senv1))
               when <true,entry> = lookup(out,senv1),
                    <bval,senv1> = evalexpr(exp,senv)
   [4] eval(abs-ser(abs-proccall(exp1,exp2),cmds),senv)
             = eval(abs-ser(abs-block(
                               absdecl-ser(
                                  absdecl-const(param,absexp-basicval(bval)),
                                  absdecl-skip),
                               cmds1),
                            cmds),
                    senv1)
               when <true,envelt(absdecl-proc(name,param,cmds1))>
                                          = lookup(name,senv1),
                    <basicval(name),senv1> = evalexpr(exp1,senv),
                    <bval,senv2>           = evalexpr(exp2,senv1)
   [5] eval(abs-ser(abs-if(exp,cmds1,cmds2),cmds),senv)
             = if(bool,
                  eval(cat(cmds1,cmds),senv1),
                  eval(cat(cmds2,cmds),senv1))
               when <basicval(bool),senv1> = evalexpr(exp,senv)
   [6] eval(abs-ser(abs-while(exp,cmds1),cmds),senv)
             = if(bool,
                  eval(cat(cmds1,abs-ser(abs-while(exp,cmds1),cmds)),senv1),
                  eval(cmds,senv1))
               when <basicval(bool),senv1> = evalexpr(exp,senv)
   [7] eval(abs-ser(abs-block(dcls,cmds1),cmds),senv)
             = eval(cat(cmds1,abs-ser(abs-blockend,cmds)),
                    evaldecl(dcls,tableadd(blockmark,senv)))
   [8] eval(abs-ser(abs-blockend,cmds),senv) = eval(cmds,removeblock(senv))
   [9] eval(abs-skip,senv) = senv

   [10] evalexpr(absexp-basicval(bval),senv) = <bval,senv>
   [11] evalexpr(absexp-read,senv)
             = <bval,tablech(in,pop(entry),senv)>
               when <true,entry> = lookup(in,senv),
                    bval         = top(entry)
   [12] evalexpr(absexp-ident(idnt),senv) = <bval,senv>
               when <true,envelt(bval)> = lookup(idnt,senv)
   [13] evalexpr(absexp-funcall(exp1,exp2),senv)
                  = applyfun(name,bval2,senv2)
                    when <bval2,senv2>          = evalexpr(exp2,senv1),
                         <basicval(name),senv1> = evalexpr(exp1,senv)
   [14] applyfun(name,bval,senv)
                  = <bval1,removeblock(senv1)>
                    when <bval1,senv1>
                             = evalexpr(exp,tableadd(param,envelt(bval),
                                           tableadd(blockmark,senv))),
                         <true,envelt(absdecl-fun(name,param,exp))>
                             = lookup(name,senv)
   [15] evalexpr(absexp-ifexp(exp,exp1,exp2),senv)
                  = if(bool, evalexpr(exp1,senv1), evalexpr(exp2,senv1))
                    when <basicval(bool),senv1> = evalexpr(exp,senv)
   [16] evalexpr(absexp-binop(oper,exp1,exp2),senv)
                  = <applybinop(oper,bval1,bval2),senv2>
                    when <bval2,senv2> = evalexpr(exp2,senv1),
                         <bval1,senv1> = evalexpr(exp1,senv)

   [17] evaldecl(absdecl-const(idnt,exp),senv)
                  = tableadd(idnt,envelt(bval),senv1)
                    when <bval,senv1> = evalexpr(exp,senv)
   [18] evaldecl(absdecl-var(idnt,exp),senv)
                  = tableadd(idnt,envelt(bval),senv1)
                    when <bval,senv1> = evalexpr(exp,senv)
   [19] evaldecl(absdecl-proc(name,param,cmds),senv)
                  = tableadd(name,envelt(absdecl-proc(name,param,cmds)),
                             senv)
   [20] evaldecl(absdecl-fun(name,param,exp),senv)
                  = tableadd(name,envelt(absdecl-fun(name,param,exp)),
                             senv)
   [21] evaldecl(absdecl-ser(dcl,dcls),senv)
                  = evaldecl(dcls,evaldecl(dcl,senv))
   [22] evaldecl(absdecl-skip,senv) = senv

   [23] cat(abs-ser(cmd,cmds1),cmds2) = abs-ser(cmd,cat(cmds1,cmds2))
   [24] cat(abs-skip,cmds) = cmds
end SMALL
```

## 4. Small with goto's

A new module `SMALL2` is created. The kernel of this module is formed by module `SMALL`. The new abstract syntax tree constructors from `SMALL2-Abs-Synt` are added and the evaluation functions and where appropriate the auxiliary functions are augmented to cope with these new functions. The structure diagram below gives the relationship.

Function `evaldecl` will need information about the program when declarations of labels are encountered. Hence the evaluation of a block has to be adapted. In equation 27 the body of the block and the rest of the program are temporarily stored in the environment. These program fragments can be retrieved by function `lookupprogram`.

This equation and equation 7 from module `SMALL` both describe the evaluation of a block. When a block contains label-declarations equation 7 will not provide an answer, while equation 27 will.



When a block does not contain label-declarations, both equations together imply the equivalence of the two applications of `evaldecl` for such a block.

Some auxiliary functions will be used to catch the behaviour of the **goto**-construct. `jmpcont` selects the corresponding rest of the program by a given identifier from the environment. This function uses `adjust-nesting`, which deletes the part of the environment corresponding to inner blocks.

The most important functions are `continuation` and its auxiliary `search-cont` which look for a continuation corresponding to a label at the moment it is declared. The first function selects the body of the block and the rest of the program from the environment, and starts up the search for a continuation in the blockbody. When a continuation is found the rest of the program is attached to this series of commands, preceded by an `abs-blockend`-marker.

The scan of the blockbody is the task of function `search-cont`. Many statements are simply skipped (equations 37, 38, 39, 42 and 43). Equation 42 describes that it is impossible to jump into an inner block.

The scan of **if**-statements is shown in equation 40. First (in the conditional clause) a continuation is searched in the **else**-branch and the rest of the block. If no continuation has been found the **then**-branch is searched.

The **while**-construct is treated in equation 41. When a label is encountered in the body of a **while**-loop, the whole loop has to be concatenated with the tail of the loop. A search is made of the rest of the blockbody for the label. When a continuation is found this is passed on. Otherwise the loopbody is scanned.

Equation 44 deals with labeled commands. If the label is found a check is made on the rest of the blockbody to find out if it is the last occurrence of this label. In that case the continuation after the last occurrence is returned. Otherwise the label is compared with the label looked for, and the value of this comparison and the rest of the blockbody are returned.

```
module SMALL2
begin
exports begin
        functions
             absdecl-lbldcmnd: CMNDS                 ->  DECL
             jmpcont         : IDNT # SENV           -> (CMNDS # SENV)
             continuation    : IDNT # SENV           -> (BOOL # CMNDS)
             search-cont     : IDNT # CMNDS          -> (BOOL # CMNDS)
             adjust-nesting  : IDNT # SENV # SENV    ->  SENV
             saveprogram     : CMNDS # CMNDS # SENV  ->  SENV
             lookupprogram   : SENV                  -> (CMNDS # CMNDS)
             deleteprogram   : SENV                  ->  SENV
             blockbody       :                       ->  IDNT
             progrest        :                       ->  IDNT
        end
imports SMALL, SMALL2-Abs-Synt

variables dcls: -> DECLS
          exp, exp1, exp2         : -> EXPR
          cmd                     : -> CMND
          cmds, cmds1, cmds2, cmds3: -> CMNDS
          senv, senv1             : -> SENV
          idnt, idnt1, lbl        : -> IDNT
          bool, found, found2     : -> BOOL
          envlt                   : -> ENVELT

equations
   [25] eval(abs-ser(abs-labldcmnd(lbl,cmd),cmds),senv)
                = eval(abs-ser(cmd,cmds),senv)
   [26] eval(abs-ser(abs-goto(lbl),cmds),senv)
                = eval(cmds1,senv1)
                  when <cmds1,senv1> = jmpcont(lbl,senv)
   [27] eval(abs-ser(abs-block(dcls,cmds1),cmds),senv)
                = eval(cat(cmds1,abs-ser(abs-blockend,cmds)),
                       deleteprogram(evaldecl(dcls,saveprogram(cmds1,cmds,senv))))

   [28] saveprogram(cmds1,cmds,senv)
                = tableadd(blockmark,
                  tableadd(blockbody,envelt(absdecl-lbldcmnd(cmds1)),
                  tableadd(progrest,envelt(absdecl-lbldcmnd(cmds)),
                  tableadd(blockmark,
                  senv))))
   [29] lookupprogram(senv) = <cmds1,cmds>
                  when <true,envelt(absdecl-lbldcmnd(cmds1))>
                            = lookup(blockbody,senv),
                       <true,envelt(absdecl-lbldcmnd(cmds))>
                            = lookup(progrest,senv)
   [30] deleteprogram(tableadd(idnt,envlt,senv))
                = tableadd(idnt,envlt,deleteprogram(senv))
   [31] deleteprogram(tableadd(blockmark,senv))
                = tableadd(blockmark,removeblock(senv))
```

```
[32] jmpcont(lbl,senv) = <cmds,senv1>
              when <true,envelt(absdecl-lbldcmnd(cmds))>
                        = lookup(lbl,senv),
                   senv1 = adjust-nesting(lbl,senv,senv)

[33] adjust-nesting(idnt,senv,tableadd(idnt1,envlt,senv1))
            = if(eq(idnt,idnt1),senv,adjust-nesting(idnt,senv,senv1))
[34] adjust-nesting(idnt,senv,tableadd(blockmark,senv1))
            = adjust-nesting(idnt,senv1,senv1)

[35] evaldecl(absdecl-label(lbl),senv)
            = tableadd(lbl,envelt(absdecl-lbldcmnd(cmds)),senv)
              when <true,cmds> = continuation(lbl,senv)

[36] continuation(lbl,senv) = <bool,cat(cmds2,cmds)>
              when <cmds1,cmds> = lookupprogram(senv),
                   <bool,cmds2> = search-cont(lbl,cmds1)
[37] search-cont(lbl,abs-ser(abs-assign(exp1,exp2),cmds))
            = search-cont(lbl,cmds)
[38] search-cont(lbl,abs-ser(abs-output(exp),cmds))
            = search-cont(lbl,cmds)
[39] search-cont(lbl,abs-ser(abs-proccall(exp1,exp2),cmds))
            = search-cont(lbl,cmds)
[40] search-cont(lbl,abs-ser(abs-if(exp,cmds1,cmds2),cmds))
            = if(found,
                 <found,cmds3>,
                 search-cont(lbl,cat(cmds1,cmds)))
              when <found,cmds3> = search-cont(lbl,cat(cmds2,cmds))
[41] search-cont(lbl,abs-ser(abs-while(exp,cmds1),cmds))
            = if(found,
                 <found,cmds3>,
                 <found2,cat(cmds2,abs-ser(abs-while(exp,cmds1),cmds))>)
              when <found2,cmds2> = search-cont(lbl,cmds1),
                   <found,cmds3>  = search-cont(lbl,cmds)
[42] search-cont(lbl,abs-ser(abs-block(dcls,cmds1),cmds))
            = search-cont(lbl,cmds)
[43] search-cont(lbl,abs-ser(abs-goto(idnt),cmds))
            = search-cont(lbl,cmds)
[44] search-cont(lbl,abs-ser(abs-labldcmnd(idnt,cmd),cmds))
            = if(found,
                 <found,cmds1>,
                 <eq(lbl,idnt),abs-ser(cmd,cmds)>)
              when <found,cmds1> = search-cont(lbl,cmds)
[45] search-cont(lbl,abs-skip) = <false,abs-skip>

end SMALL2
```

## 5. Conclusions and further research

The prime question to be answered in this paper is whether an elegant algebraic specification can be given of the most unstructured of the classical program features: the jump. In my opinion, this question can be answered positively. The present specification is somewhat longer than the specification in *denotational semantics* by Gordon [5]. It is felt, however, that the algebraic specification is at least as legible as the denotational specification.

The modularization of the definition of Small posed various interesting problems. It was for instance a challenge to make a specification of Small with functions `eval` and `evaldecl` that could be reused in the specification of Small2. At first, it seems to be possible to eliminate the auxiliary command `abs-blockend` from the specification of module `SMALL` through a change in the equations for the evaluation function for series of commands like this:

```
eval(abs-ser(cmd,cmds),senv) = eval(cmds,evalcmd(cmd,senv))
```

However, this approach does not allow one to force a break in the flow of control of the program in a natural way. From a model-theoretic point of view such a specification and the specification of `SMALL` in the paper have the same initial algebra (intuitively the language Small). A specification containing the above equation proves to be stronger than the specification in the paper, hence there are fewer models satisfying it, and Small2 is not among these.

Another problem encountered is the wish to hide auxiliary functions like `cat` in module `SMALL`. This function is really an internal construct, not a feature of Small, so the user of module `SMALL` exclusively should not be bothered by its existence. However, it is needed in `SMALL2`, hence it must be exported. In this paper the problem is ignored by simply exporting everything. Further research on this topic is clearly needed.

## Acknowledgement

Discussions with J. Heering, P. Hendriks, P. Klint, A. Verhoog and H. Walters markedly improved earlier versions of this specification. Referee C. Koster, and J. Heering and P. Klint made useful comments on the paper in general.

## References

1. J.A. Bergstra, J. Heering, and P. Klint (1985). *Algebraic definition of a simple programming language,* Report CS-R8504, CWI, Amsterdam.
2. H. Ehrig and B. Mahr (1985). *Fundamentals of Algebraic Specifications 1,* EATCS Monographs on Theoretical Computer Science, 6, Springer-Verlag, Berlin.
3. H. Ganzinger and R. Giegerich (1985). *An experiment in logic specification of compilers and interpreters,* EATCS Monographs on Theoretical Computer Science, Universität Dortmund, Draft to be presented at the workshop "Programs & data objects" Copenhagen.
4. J.A. Goguen and K. Parsaye-Ghomi (1981). Algebraic denotational semantics using parameterized abstract modules.. J. Diaz & I. Ramos (eds.). *Formalizing programming concepts*, Springer Lecture Notes in Computer Science, 292-309.
5. M.J.C. Gordon (1979). *The denotational description of programming languages*, Springer Lecture Notes in Computer Science, Springer-Verlag, New York.
6. D.E. Knuth (1974). *Structured programming with goto statements,* Report STAN-CS-74-416, Stanford.