Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

N.W.P. van Diepen

A study in algebraic specification:
a language with goto-statements

Computer Science/Department of Software Technology Report CS-R8627 August

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

A Study in Algebraic Specification: 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 illustrate 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].
1986 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 - GIPE).
Note 2: This paper will be submitted for publication elsewhere.

1. INTRODUCTION

1.1. The specification of jump-statements

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. See for instance [BTh83], [GM84] and [GPG81]. Bergstra, Heering & Klint [BHK85] 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 [Gor79] as an example language to illustrate specifications in *denotational semantics*. SMALL is built in layers to allow one to concentrate on the difficulties of specifying a certain language construct while other constructs are excluded. 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 [Knu74]. Hence specification of this construct is a serious test for any formalism. We are also interested in the question how to capture the various layers of SMALL in a single, modular, definition.

The next section describes the abstract 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. In section 5 an alternative, more elegant, specification of SMALL without **goto**'s

Report CS-R8627
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

is given. This specification is not suited for addition of **goto**-statements. A specification circumventing this problem of the extended language is given. Section 6 provides a description of an "ad hoc" implementation of the specification of sections 3 and 4 and some notes on automatic implementation.

1.2. The specification formalism

An algebraic specification consists of *sorts*, *functions* on these sorts (a constant is a function without arguments), 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 and are again exported by it. Lastly a module can have *parameters* to which modules can be bound upon import.

The algebraic specification formalism used is described in detail in [BHK85]. It is similar to algebraic formalisms defined in [Kla83], [Wir83], [Gau84] and [Loe84]. For more detail on algebraic specification see [EM85].

2. SYNTAX AND INFORMAL SEMANTICS OF SMALL

2.1. Syntax

The syntax of SMALL is given below in regular BNF-notation. The primitive notions `<basic-value>`, `<identifier>` and `<binary-operator>` are left unspecified.

In the 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 the starting point in this paper) this ambiguity is resolved, hence it presents no problems here. The unspecified primitive notions will turn up as primitive nodes.

```
<program>     ::= 'program' <command> .
<command>     ::= <expression> ':=' <expression> |
                  'output' <expression> |
                  <expression> '(' <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> .
```

In the sequel the kernel language will be called SMALL1 and the extension SMALL2. These indications will be used in the names of the modules of the specification.

2.2. Abstract syntax and informal semantics

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: **SMALL1-Primitives**, containing the sorts **BASICVAL** (the basic values of SMALL1), **IDNT** (identifiers) and **BINOP** (binary operators), together with an equality function on **IDNT** yielding a boolean.

2.2.2. Abstract syntax and informal semantics of the kernel language. The constructor functions for the abstract syntax are combined in module **SMALL1-Abs-Synt**. The sorts **DECL** (declarations), **DECLS** (lists of declarations), **EXPR** (expressions), **CMND** (commands), **CMNDS** (lists of commands) and **PROGRAM** (SMALL1 programs) are defined here and the module **SMALL1-Primitives** is imported.

The following constructor functions are defined:

- the **<program>** constructor:

```
program: CMNDS -> PROGRAM
```

This function corresponds to the root of the abstract syntax tree of a SMALL1 program. It turns a series of commands into a program.

- **<command>** constructors:

```
abs-assign  : EXPR # EXPR          -> CMND
abs-output  : EXPR                 -> CMND
abs-proccall: EXPR # EXPR          -> CMND
abs-if      : EXPR # CMNDS # CMNDS -> CMND
abs-while   : EXPR # CMNDS         -> CMND
abs-block   : DECLS # CMNDS        -> CMND
abs-ser     : CMND # CMNDS         -> CMNDS
abs-skip    :                      -> CMNDS
```

SMALL1 has rather conventional commands. Unusual features include the left-hand side of an assignation command, which is an expression that yields an identifier. Similarly, the first expression of a procedure call gives its name, the second one gives the value of the (single) parameter. Every procedure and function has exactly one parameter.

A block consists of a list of declarations and a list of commands. Sequential composition of commands is modeled as a list with **abs-skip** as terminator.

- **<expression>** constructors:

```
absexp-basicval: BASICVAL            -> EXPR
absexp-read    :                     -> EXPR
absexp-ident   : IDNT                -> EXPR
absexp-funcall : EXPR # EXPR         -> EXPR
absexp-ifexp   : EXPR # EXPR # EXPR  -> EXPR
absexp-binop   : BINOP # EXPR # EXPR -> EXPR
```

A function call consists - like a procedure call - of an expression yielding its name, and again exactly one parameter. Since basic values are treated abstractly, no concrete binary operators have been defined and their appearance here is purely *pro forma.*

- `<declaration>` constructors:

```
absdecl-const: IDNT # EXPR          -> DECL
absdecl-var  : IDNT # EXPR          -> DECL
absdecl-proc : IDNT # IDNT # CMNDS -> DECL
absdecl-fun  : IDNT # IDNT # EXPR  -> DECL
absdecl-ser  : DECL # DECLS         -> DECLS
absdecl-skip :                      -> DECLS
```

In declarations of constants and variables, the first component yields the new name and the second component states the initialization value. The second identifier of function and procedure declarations is the name of the parameter. The structure of declarations is list-like.

2.2.3. Abstract syntax and informal semantics of the extension with goto's. To enrich the SMALL1 abstract syntax with **goto**'s a module `SMALL2-Abs-Synt` is built on top of `SMALL1-Abs-Synt`. It contains three additional constructor functions:

```
abs-goto     : IDNT          -> CMND
abs-labldcmnd: IDNT # CMND -> CMND
absdecl-label: IDNT          -> DECL
```

A **goto**-statement jumps to the *last* label with the name `IDNT` 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.

The following structure diagram (see [BHK85]) shows the import relationship between the modules discussed above.

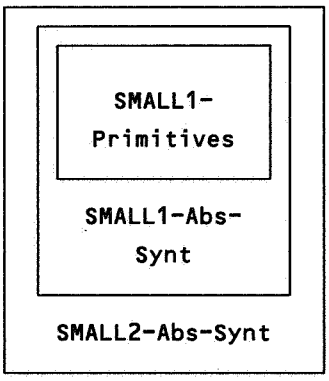

FIGURE 2.1. The structure of the abstract syntax modules.

3. ALGEBRAIC SEMANTICS OF THE SMALL KERNEL

3.1. Specification of the environment

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 `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 entry found or `false` and the `error-entry` for a given name and table) are given. An equality predicate must be defined on the names. These sorts and functions are bundled in module `Tables`.

```
module Tables
begin   parameters
        Names   begin   sorts NAME
                        functions eq: NAME # NAME -> BOOL
                end Names,
        Entries begin   sorts ENTRY
                        functions error-entry: -> ENTRY
                end Entries
exports begin
        sorts TABLE
        functions nulltable  :                        ->  TABLE
                  tableadd   : NAME # ENTRY # TABLE ->  TABLE
                  tablech    : NAME # ENTRY # TABLE ->  TABLE
                  lookup     : NAME # TABLE         -> (BOOL # ENTRY)
        end
imports Booleans

variables
    name, name1, name2    : -> NAME
    entry, entry1, entry2 : -> ENTRY
    table                 : -> TABLE

equations
```

```
  [  i] lookup(name, nulltable) = <false, error-entry>
  [ ii] lookup(name1, tableadd(name2,entry,table))
           = if(eq(name1,name2),
                <true,entry>,
                lookup(name1,table))
  [iii] tablech(name1,entry1, tableadd(name2,entry2,table))
           = if(eq(name1,name2),
                tableadd(name1,entry1,table),
                tableadd(name2,entry2,tablech(name1,entry1,table)))
end Tables
```

SMALL has 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 `SMALL1-Tables` is built for this task on top of `Tables`. A constant `blockmark` is introduced to separate blocks in a table. This constant is of a new sort, `TABLEMARK`. Of course a function `removeblock` is defined.

The parameter `NAME` is bound to `IDNT` (from `SMALL1-Primitives`). The objects we want to put into the table have to be bound to sort `ENTRY` from `Entries`. Since this comprises objects of various sorts (e.g. declarations and basic values) an intermediate module `SMALL1-Env-Elt` is constructed to provide a common sort, called `ENVELT` (environment-element), and injection functions into this sort. This intermediate sort is bound to `ENTRY`.

Finally sort `TABLE` is renamed to `SENV` (SMALL-environment).

```
module SMALL1-Tables
begin
exports begin
        sorts TABLEMARK
        functions blockmark  :                  -> TABLEMARK
                  tableadd   : TABLEMARK # SENV -> SENV
                  removeblock: SENV             -> SENV
        end
imports Tables  { renamed by [TABLE     -> SENV,
                              nulltable -> null-senv]
                  Names bound by [NAME -> IDNT,
                                  eq   -> eq]
                     to SMALL1-Primitives
                  Entries bound by [ENTRY       -> ENVELT,
                                    error-entry -> error-value]
                     to SMALL1-Env-Elt
                }

variables
    idt : -> IDNT
    elt : -> ENVELT
    tbl : -> SENV

equations
   [ iv] removeblock(tableadd(blockmark,tbl)) = tbl
   [  v] removeblock(tableadd(idt,elt,tbl))   = removeblock(tbl)
   [ vi] lookup(idt,tableadd(blockmark,tbl))  = lookup(idt,tbl)
```

```
   [vii] tablech(idt,elt,tableadd(blockmark,tbl))
              = tablech(idt,elt,tbl)
end SMALL1-Tables
```

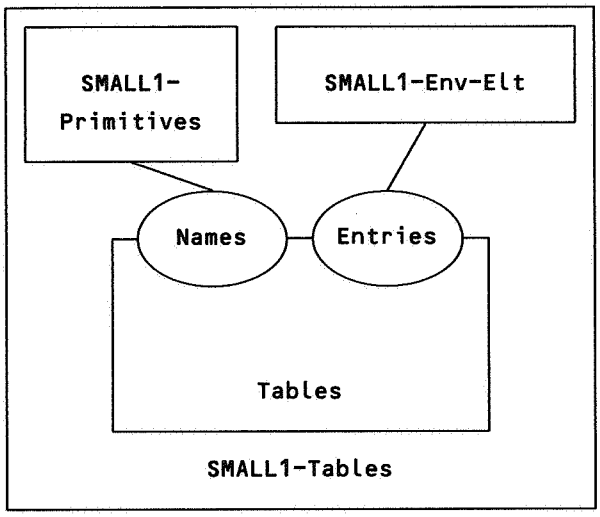

FIGURE 3.1. Environment structure.

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.

3.2. Specification of the semantics

The algebraic specification of the semantics of SMALL1 is quite straightforward. See the accompanying structure diagram below.

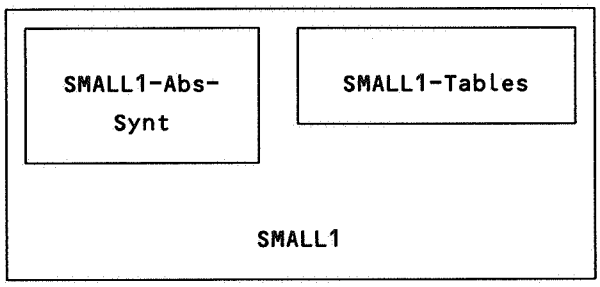

FIGURE 3.2. The structure of module SMALL1.

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.

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 parameter binding is described by constructing a new block consisting of

the declaration and initialization of the parameter followed by the procedure body itself. A similar construction is used in equation 14 to bind the parameter of a function call. The difference in treatment between procedure and function calls is a reflection of the asymmetry of the notions *command* and *expression* in SMALL1. Evaluation of an expression produces a value as result, while evaluation of a command only affects the environment.

In equation 8 a block is created in the environment, and in equation 9 it is removed again.

```
module SMALL1
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 SMALL1-Abs-Synt, SMALL1-Tables

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,emptylist,
                           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 SMALL1
```

4. SMALL WITH GOTO'S

Module **SMALL2** is defined by extending **SMALL1** with the abstract syntax tree constructors introduced in **SMALL2-Abs-Synt** and by augmenting the evaluation functions and where appropriate the auxiliary functions to cope with these new functions. The structure diagram below gives the relationship between the modules.

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 **SMALL1** 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. (If, accidentally, these equations produce different answers, the specification would be incorrect, perhaps even inconsistent.)

Some auxiliary functions will be used to describe the behaviour of the **goto**-construct. **jmpcont** selects from the environment the continuation of the program for a given label identifier. 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

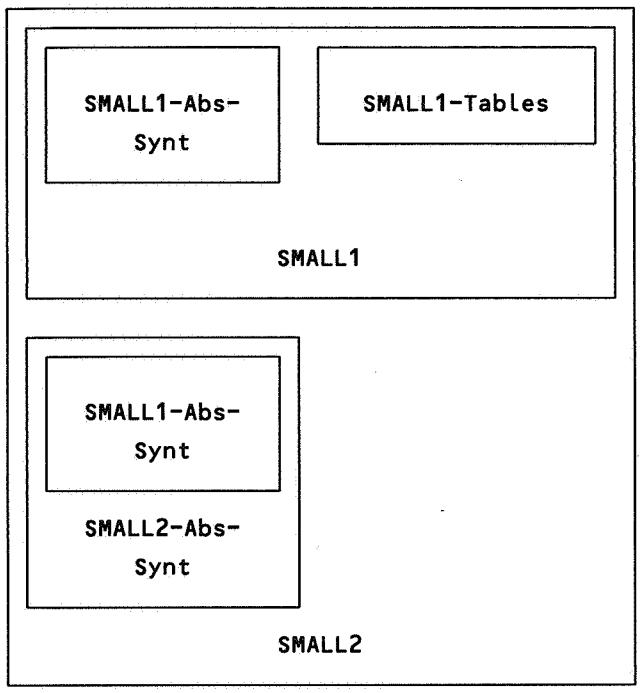

FIGURE 4.1. The structure of module SMALL2.

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 appended to the remainder of the loopbody after the label. A search is made in 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 SMALL1, 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. A NOTE ON MODULARITY

5.1. Auxiliary functions

A problem was encountered with the hiding of auxiliary functions like `cat` in module **SMALL1**. This function is a typical internal construct, needed to make use of an intermediate result in the specification, and in no way an essential feature of SMALL1. Hence the user of module **SMALL1** exclusively should not be bothered by its existence. However, it is needed in **SMALL2**, so it must be exported or redefined. In this paper the problem is ignored by simply exporting everything. The export facility of the algebraic specification formalism used is too weak to handle such (quite common) situations. Further research on this topic is clearly needed.

5.2. An alternative definition of SMALL1

It is possible to eliminate the auxiliary command `abs-blockend` from the specification of module **SMALL1** through a change in the equations for the evaluation function for series of commands like this:

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

wherein `eval` also operates on single commands. A specification in this form has another pleasing aesthetic aspect. The evaluation function is able to treat all constructors of commands as primitive operands, not as head or constructor of a list. Thus the specification is both shorter and more symmetric. This specification is given below.

```
module SMALL1
begin
exports
    begin
    functions
        eval       : PROGRAM # ENVELT          ->  SENV
        eval       : CMND # SENV               ->  SENV -- new
        eval       : CMNDS # SENV              ->  SENV
        .
        . as in the first specification without abs-blockend
        .
        out : -> IDNT
    end
imports SMALL1-Abs-Synt, SMALL1-Tables

variables identical to the first specification

equations
   [ 1] eval(program(cmds),input)
              = eval(cmds, tableadd(out,emptylist,
                           tableadd(in,input,null-senv)))
   [2a] eval(abs-assign(exp1,exp2),senv)
              = tablech(idnt,envelt(bval),senv2)
                when <bval,senv1>           = evalexpr(exp2,senv),
                     <basicval(idnt),senv2> = evalexpr(exp1,senv1)
```

```
[3a] eval(abs-output(exp),senv)
             = tablech(out,cat(entry,bval),senv1)
               when <true,entry> = lookup(out,senv1),
                    <bval,senv1> = evalexpr(exp,senv)
[4a] eval(abs-proccall(exp1,exp2),senv)
             = eval(abs-block(absdecl-ser(
                                absdecl-const(param,
                                        absexp-basicval(bval)),
                                absdecl-skip),
                              cmds1),
                    senv1)
               when <true,envelt(absdecl-proc(name,param,cmds1))>
                                        = lookup(name,senv1),
                    <basicval(name),senv1> = evalexpr(exp1,senv),
                    <bval,senv2>           = evalexpr(exp2,senv1)
[5a] eval(abs-if(exp,cmds1,cmds2),senv)
             = if(bool,
                  eval(cmds1,senv1),
                  eval(cmds2,senv1))
               when <basicval(bool),senv1> = evalexpr(exp,senv)
[6a] eval(abs-while(exp,cmds1),senv)
             = if(bool,
                  eval(cat(cmds1,
                           abs-ser(abs-while(exp,cmds1),abs-skip)),
                       senv1),
                  senv1)
               when <basicval(bool),senv1> = evalexpr(exp,senv)
[7a] eval(abs-block(dcls,cmds1),senv)
             = removeblock(
                       eval(cmds1,
                            evaldecl(dcls,tableadd(blockmark,senv))))
[8a] eval(abs-ser(cmd,cmds),senv)
             = eval(cmds,eval(cmd,senv))
[ 9] eval(abs-skip,senv) = senv
.
. identical to the first specification
.
[24] cat(abs-skip,cmds) = cmds

end SMALL1
```

The numbering of the old equations has been retained whenever possible. Unchanged equations retain their number, adapted equations have an "a" appended. Only equation 8a is really new. It replaces the equation describing `abs-blockend` in the first specification.

Other changes fall into two categories. The enclosing `abs-ser` with trailing tail of the program has disappeared everywhere. Secondly, in constructs enclosing an inner series of commands (**if**- and **while**-statements, blocks and with them procedures) the boundaries are delineated by a recursive application of function `eval`. For blocks this results in the superfluity of the marker `abs-blockend`.

5.3. A problem with extension the alternative SMALL1 *definition*

The modularization of the definition of SMALL2 posed an interesting problem. It was a challenge to make a specification of SMALL1 with functions `eval` and `evaldecl` that could be reused in the specification of SMALL2.

The approach of the preceding section does not allow one to describe non-structured flow of control constructs. An example will serve to illustrate this. Suppose we have a program that executes exactly one jump (to, say, label `lbl`) in some correct evaluation. Then the following equations would hold:

```
eval(abs-ser(abs-goto(lbl),original-tail),senv)                    =
eval(original-tail,eval(abs-goto(lbl),senv))                       =
eval(original-tail,eval(abs-ser(abs-lbldcmnd(lbl,...),...),senv)) =
eval(original-tail,result-senv)
```

In effect the jump-statement is evaluated correctly. However, the obsolete continuation dating back to the state of the evaluation just before the jump is not forgotten. After finishing the whole evaluation in correct order (it has been assumed that it did not contain other jumps), the resulting environment is treated as input for the old tail of the program, clearly an undesired action.

There is no way to avoid this problem when the second specification of **SMALL1** is extended to **SMALL2**. From a model-theoretic point of view the first and the second specification of module **SMALL1** have the same initial model (intuitively the language SMALL1). The specification containing equation 8a is stronger than the first specification, hence there are fewer models satisfying it, and the initial model of SMALL2 is not among these.

5.4 The alternative definition of SMALL2

Since the evaluation function from the second specification of SMALL1 cannot be used to specify SMALL2, the only way to specify an evaluation function for the latter language starting with this specification of SMALL1 is the introduction of another evaluation function. This function will be called `eval2`. The adapted specification of SMALL2 is given below. To facilitate comparison the numbering of previous definitions has been retained when possible.

```
module SMALL2
begin
exports begin
        functions
             eval2            : PROGRAM # ENVELT     ->  SENV -- new
             eval2            : CMNDS # SENV         ->  SENV -- new
             absdecl-lbldcmnd: CMNDS                 ->  DECL
             .
             . identical to the first specification
             .
             progrest         :                      ->  IDNT
             abs-blockend     :                      ->  CMND -- new
        end
imports SMALL1, SMALL2-Abs-Synt

variables identical to the first specification

equations
   [1b] eval2(program(cmds),input)
                = eval2(cmds,tableadd(out,emptylist,
                                tableadd(in,input,null-senv)))
```

```
[2b] eval2(abs-ser(abs-assign(exp1,exp2),cmds),senv)
            = eval2(cmds,senv1)
              when senv1 = eval(abs-assign(exp1,exp2),senv)
[3b] eval2(abs-ser(abs-output(exp),cmds),senv)
            = eval2(cmds,senv1)
              when senv1 = eval(abs-output(exp),senv)
[4b] eval2(abs-ser(abs-proccall(exp1,exp2),cmds),senv)
            = eval2(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)
[5b] eval2(abs-ser(abs-if(exp,cmds1,cmds2),cmds),senv)
            = if(bool,
                 eval2(cat(cmds1,cmds),senv1),
                 eval2(cat(cmds2,cmds),senv1))
              when <basicval(bool),senv1> = evalexpr(exp,senv)
[6b] eval2(abs-ser(abs-while(exp,cmds1),cmds),senv)
            = if(bool,
                 eval2(cat(cmds1,
                           abs-ser(abs-while(exp,cmds1),
                           cmds)),
                       senv1),
                 eval2(cmds,senv1))
              when <basicval(bool),senv1> = evalexpr(exp,senv)
[25a] eval2(abs-ser(abs-labldcmnd(lbl,cmd),cmds),senv)
            = eval2(abs-ser(cmd,cmds),senv)
[26a] eval2(abs-ser(abs-goto(lbl),cmds),senv)
            = eval2(cmds1,senv1)
              when <cmds1,senv1> = jmpcont(lbl,senv)
[27a] eval2(abs-ser(abs-block(dcls,cmds1),cmds),senv)
            = eval2(cat(cmds1,abs-ser(abs-blockend,cmds)),
                    deleteprogram(
                       evaldecl(dcls,
                          saveprogram(cmds1,cmds,senv))))
[8b] eval2(abs-ser(abs-blockend,cmds),senv)
            = eval2(cmds,removeblock(senv))
[9b] eval2(abs-skip,senv) = senv
```

```
[28] saveprogram(cmds1,cmds,senv)
.
. identical to the first specification
.
[45] search-cont(lbl,abs-skip) = <false,abs-skip>

end SMALL2
```

Unfortunately function `eval` is rarely reusable. Only commands into which and out of which one cannot jump - this is restricted to assignment-, output- and dummy-statements in SMALL1 - can use the semantics defined with the old function to define the semantics with `eval2`.

All other occurrences of `eval2` have to be defined from scratch, starting with the evaluation of programs, and ending with the reappearance of the marker `abs-blockend`. Of course this specification is similar to the old specification of `eval` in `SMALL1` and `SMALL2`.

The disadvantages of this approach are obvious. The specification of SMALL2 is longer and redoes definitions found in the specification of SMALL1. Also the triviality of the extension has been lost: it is not clear without proof that a SMALL1 program will behave the same when it is evaluated using the `SMALL1`- or the `SMALL2`-specification respectively. The relation between rules 7a and 27a only exists in the sense that they are designed to have the same meaning in specific circumstances, the evaluation of a SMALL1 program.

However there are also advantages to this approach. First of all, it is perhaps more realistic: the module `SMALL1` could come from a library of programming languages as a black box. Also the definition of SMALL1 is more elegant, so the chances of mistakes in this definition are smaller.

6. The implementation of algebraic specifications

Our ultimate goal is to generate an interpreter or compiler for a programming language, based on an algebraic specification of its semantics. Two attempts have been made to implement the specification presented in this paper. Both implementations have been done by hand, but with an open eye for the possibilities to generate them automatically. The scheme specifically designed to be mechanized operates on a class of specifications which is shown to be too restrictive to be of practical use for our purposes. Some comments will be given on the problems concerning automatic translation.

6.1. An "ad hoc" implementation

6.1.1. Term rewriting systems. An algebraic specification can be implemented if it can be turned into a *term rewriting system* ([BK82], [DE84]). This can be done by giving directions to the equations in the sense that

$$A = B$$

is replaced by

$$A \rightarrow B$$

(or $A \leftarrow B$; most algebraic specifications have an intuitive direction from left to right). $A \rightarrow B$ has the meaning that term A can be reduced (rewritten) to term B. For B to be a proper reduct of term A it should be closer to a so called normal form (an irreducible term), if any. Intuitively, a normal form is the standard, most simple, way to express a certain term.

Similarly, conditional equations of the form

$$A = B \textbf{ when } C_1 = D_1, \ldots, C_n = D_n$$

are replaced by

$$((C_1 \rightarrow D_1) \wedge \cdots \wedge (C_n \rightarrow D_n)) \rightarrow (A \rightarrow B)$$

The theory of term rewriting systems deals with properties like *termination* (every reduction is finite, i.e. after a finite number of steps a normal form is reached) and *confluency* (two divergent finite reduction sequences from the same term have to converge again). In general our algebraic specifications cannot be turned into term rewriting systems with these nice properties. Since we may specify a possibly infinite loop, termination cannot be assured, and treatment of error cases may result in more than one stop criterion. Usually, however, the writer of the specification has a good intuitive working model of his specification in mind, in which these "bugs" are simply ignored. For our purpose it is good enough if the writer of a specification follows the scheme above in an implementable way.

6.1.2. A method to represent equations in Prolog. The language Prolog lends itself relatively well to implementing an algebraic specification as a term rewriting system. The arrow in $A \rightarrow B$ can be read as "the analysis of A reduces to the analysis of B". This we can model with a relation `analyse` between terms and their normal forms.

Schematically $A \rightarrow B$ then translates into the Prolog clause:

```
analyse(A, Res) :- analyse(B, Res).
```

which reads: "the analysis of term A has result Res when the analysis of term B has result Res".

This crude scheme will need modification, however, to deal with evaluation of arguments of term A that have to be known first.

With the same provision for both term A and terms C_i, rules of the format $((C_1 \rightarrow D_1) \wedge \cdots \wedge$

$(C_n \rightarrow D_n)) \rightarrow (A \rightarrow B)$ translate to:

```
analyse(A, Res)
    :- analyse(C1 , Res1),
       ...,
       analyse(Cn , Resn),
       analyse(B, Res).
```

Since conditions $C_i \rightarrow D_i$ may interact in the sense that one defines an intermediate result for another, during translation their order may have to be changed to provide for the correct interdependency.

Sometimes no constructive translation of the **when**-part of the specification exists. This happens when the clause is used to simulate an existential quantifier. Hence the wish to produce implementable specifications automatically will pose constraints on the class of allowable specifications.

6.1.3. The "ad hoc" implementation. The specification of SMALL2 has been implemented along the lines indicated above. This posed only minor difficulties, though it indicated some possible problem areas.

The main trouble spot from the point of view of implementation is the **when**-clause. Existential quantifiers had to be eliminated. Most of them were just aliases for longer expressions. The specification contained some trivial cases of true quantification, for instance variable `bool` in equation 5 (section 3) is quantified over the sort `BOOL`, the equation has no meaning when evaluation of expression `exp` would yield something else. A close operational translation has been made in these cases, which posed no difficulties. A keyword reserved for abbreviation might allow enough flexibility while avoiding confusion. True existential quantification could then be ruled out without problem. More thought went into the correct order of the evaluation of the conditions. It looks feasible to let the order of specification be the order of evaluation. This follows closely the intuitively attractive bottom-up approach in writing conditions. Alternatively, the reverse order of specification, corresponding to the top-down approach, could be chosen.

The evaluation scheme so far cannot handle terms with terms as arguments. These terms fall into two categories per argument. The easiest and more frequent form is

```
f(X) = g(h(X))
```

which is equivalent to

```
f(X) = g(Y)
   when h(X) = Y
```

and can be evaluated as such. Sometimes, however, no intermediate result can be found, as in

```
if(Test,Then-part,Else-part) = Res
```

where for instance the `Then-part` might be ill-defined if the `Test` evaluates to `false`. These cases have been solved by splitting such equations as follows:

```
if(Test,Then-part,Else-part) = Then-part
   when Test = true
if(Test,Then-part,Else-part) = Else-part
   when Test = false
```

A more general discussion of techniques for delaying the evaluation of certain arguments follows in the sections below.

6.2. The automatic scheme of Drosten and Ehrich

An automatic translation method for algebraic specifications has been proposed by Drosten and Ehrich [DE84]. This method is presented here in a slightly modified version. It has three parts:

a. For every **function** (and constant) `f(I1,..., In)` in the specification, a rule

```
analyse(f(I1,..., In), T):-
            analyse(I1,K1),
            ...,
            analyse(In,Kn),
            normalize(f(K1,..., Kn), T).
```

is added to the system. (This function `analyse` has the same intrinsic meaning as the function of the same name above.)

b. Every **equation** `A = B` in the specification is given a rewriting direction `A` → `B` and is represented by the addition to the system of a fact

```
rule(A, B).
```

c. Additionally two rules are added to the system in the order given below:

```
normalize(X,Y) :- rule(X,Z), analyse(Z,Y).
normalize(X,X).
```

The rules and facts given under *a*, *b* and *c* represent the whole program.

In the description above the convention is adopted that variables start with a capital and functions with a small letter. Hence the specifications in our formalism need a trivial transformation.

Analysis of a term is done as follows:

- All parameters of the function are recursively reduced to normal form (rule *a*).
- An attempt is made to unify one rule with the function operating on normal form arguments.
  If the matching succeeds the result of the rule is the new starting point for the analysis (*c*, first rule).
  If the matching fails then no rule applies to the term, hence it is in normal form (the first rule of *c* fails and the second succeeds).

Since formally a term can also exist of a single variable without context Drosten and Ehrich added a fact

```
analyse(c,C)
```

for every variable `C` (note the capital) to represent it in a theoretically complete way. These facts are deleted in the description above since the evaluation of a variable on its own has no practical value.

EXAMPLE: the translation of the following specification is given (this example is essentially the same as the example on page 10 in [DE84]). For ease of presentation the functions are given with small letters and the variables with capitals.

```
module Natnumbers
begin
exports
  begin
    sorts
      nat
    functions
      zero:           -> nat
      succ: nat       -> nat
      add : nat # nat -> nat
      mult: nat # nat -> nat
 end

 variables
   M,N: nat

 equations
   add(zero, N)      = N
   add(succ(M), N)   = succ(add(M,N))
   mult(zero, N)     = zero
   mult(succ(M), N) = add(mult(M,N), N)

end Natnumbers
```

Step *a* of the implementation strategy yields:

```
analyse( zero, T) :-
            normalize( zero, T).
analyse( succ(I1), T) :-
            analyse(I1, K1),
            normalize( succ(K1), T).
analyse( add(I1,I2), T) :-
            analyse(I1, K1),
            analyse(I2, K2),
            normalize( add(K1,K2), T).
analyse( mult(I1,I2), T) :-
            analyse(I1, K1),
            analyse(I2, K2),
            normalize( mult(K1,K2), T).
```

Note that constant `zero` is treated as an ordinary function. By default all its arguments are analysed before the result is normalized.

The equations are treated as rewrite rules from left to right as follows:

```
rule( add(zero,N),      N ).
rule( add(succ(M),N),   succ(add(M,N)) ).
rule( mult(zero,N),     zero ).
rule( mult(succ(M),N),  add(mult(M,N), N) ).
```

Finally we add:

```
normalize(X,Y) :- rule(X,Z), analyse(Z,Y).
normalize(X,X).
```

The scheme of Drosten and Ehrich is an innermost evaluation scheme: first all arguments of a function are brought into normal form before the function as a whole is tackled. The check whether arguments are in normal form has two distinct disadvantages for implementation purposes.

- Normal forms may be checked over and over again, which decreases efficiency. This may be solved by creating a *cache* of known normal forms. However, in general such a *cache* will rapidly become very large. Looking up a normal form then becomes a bottleneck itself. Efficient ordering of the normal forms or the saving of the most frequently encountered normal forms only will improve the performance of the search algorithm. What ordering should be chosen is very much problem dependent, hence no generally suitable strategy can be devised. Saving a certain fixed number of the latest encountered normal forms might be good enough to implement the second alternative. However, it is not a trivial task to implement this strategy in Prolog.
- More serious is the problem that this scheme is not optimal with respect to the termination behaviour of the resulting rewrite system. For instance, this scheme cannot cope with non-strict functions (functions which can successfully be evaluated even though one or more of their arguments are still unknown). The prime example in this category is the function:

```
if(Test, Then-part, Else-part)
```

in which the `Else-part` can be disregarded if `Test` evaluates to `true`, and the `Then-part` if it is `false`. A function like this is needed to specify evaluation of loops in a language. In such a specification some infinite sequences of reductions (the reductions corresponding to endless loops) cannot be eliminated, since in general termination of a loop depends on the environment during execution.

For our purposes improvement is needed in the scheme of Drosten and Ehrich on efficiency (the number of reductions performed and the stack space used) and termination are still unknown. behaviour. The next section introduces a better reduction strategy principle.

6.3. Lazy evaluation

6.3.1 Outermost reduction strategies. The converse of innermost reduction strategies are outermost reduction strategies. These reduction strategies postpone evaluation and try to do as little work as possible. Hence they are alternatively described as lazy evaluation.

When applying an outermost reduction strategy one first tries to reduce a term as a whole. If this does not succeed, an attempt is made to perform at least one (outermost) reduction step of one of its arguments. This method is repeated until no further reductions can be performed.

Outermost reduction strategies differ in the number and order of arguments that are reduced when the outermost function cannot be reduced as it stands. Leftmost-outermost reduction e.g. only reduces the leftmost argument that can be reduced. This reduction strategy would be the most efficient strategy for the evaluation of the `if`-function in the preceding section. However, should the function be changed to `if(Then-part,Else-part,Test)` evaluation may never end again. (Viz. the `Then-part` contains an infinite loop when the `Test` reduces to `false`.) Since we cannot assume prior information about the order in which arguments should be evaluated another strategy is needed.

Optimal termination behaviour can be reached when the user indicates which arguments are essential for reduction to progress and which arguments can or must be delayed. Such an attitude transfers responsibility to the user for the choice of reduction strategies. In my view this is best introduced as an option to overrule a default strategy. An outermost reduction strategy with reasonably broad application fields is presented in the next section.

6.3.2. Parallel Outermost Reduction in Prolog[1]. The problems with the `if`-function introduced in the preceding section can be avoided through simultaneous outermost reduction of *all* arguments. Such a reduction is called *parallel outermost reduction.* When applying this strategy, one reduction step is made for all arguments and some non-terminating sequence of reductions for one argument can no longer interfere with a terminating sequence from another argument in this kind of non-strict function.

An implementation in Prolog is given below.

a. The `analyse`-function is given in the following order by

```
analyse(T,Res) :-
    step(T,I),
    test(T,I), !,
    analyse(I,Res).
analyse(T,T).
```

b. The test in step *a* is necessary to detect the existence of changes at the argument level. It fails when no change has been made.

1. This section reports on work together with L.C. van der Gaag and P.R.H. Hendriks (both CMCS).

```
test(X,X):- !, fail.
test(X,Y).
```

c. A reduction step is implemented as follows:

```
step(T,Res) :- rule(T,Res).
step(T,Res) :-
    T =.. [Func|Args],
    argsstep(Args,Args1),
    Res =.. [Func|Args1].
step(T,T).
argsstep([],[]).
argsstep([H|T],[H1|T1]) :-
    step(H,H1),
    argsstep(T,T1).
```

First of all this scheme tries to apply some rule. If no rule applies, term `T` is decomposed into a function name and a list of arguments. A step is made in all arguments and the term is reconstructed using this reduced argument list. Otherwise the term cannot be reduced, hence a zero step is made.
This evaluation is independent of the names of the functions and constants in the algebraic specification. A method to retain the dependence is given in the sequel.

d. Equations A = B are given a direction A → B and are represented (compare Drosten and Ehrich) as:

```
rule(A,B).
```

This parallel outermost reduction scheme has a better termination behaviour than the scheme of Drosten and Ehrich. It will perform at most the same number of one-step reductions as the innermost scheme. The translation will produce less text for any specification but the smallest: the static overhead of rules is a bit larger, but an analysis for every function separately is no longer necessary.

In certain cases the normal form of a proper subterm has to be found before a rule can be applied. If this reduction takes several steps the reduction of the term as a whole acts like a jo-jo: no rule can be applied to the term so the internal arguments are examined and one step is applied. The scheme calls for another test at top level (which fails) and the whole term has to be dissected again. Here the innermost scheme would be more efficient.

It might be preferable to have the names of functions available in the reduction scheme, for instance for typing purposes. This can be done by replacement of implementation step *c* by the following rule for every function f in the specification:

```
step(f(X1,...Xn),f(Y1,...Yn)) :-
    step(X1,Y1),
    ...
    step(Xn,Yn).
```

and by retaining the rules:

```
step(T,Res) :- rule(T,Res).
step(T,T).
```

Of course, the gain in the size of the translation has been lost then.

7. 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 [Gor79]. It is felt, however, that the algebraic specification is at least as legible as the denotational specification.

Progress is being made in the field of modularity of specifications. Recent work by Bergstra, Heering and Klint on *module algebra* [BHK86] provides formal tools to reason about import/export relationships. The problems encountered are largely circumvented in the present paper, with the notable exception of the more elegant definition of function `eval` in section 5.

The question of efficient implementation of algebraic specifications is still an open problem. The solutions suggested in section 6 are either to restrict the class of allowable specifications or to give the writer more responsibility for the termination behaviour of the term rewriting system derived from his specification. An optimal trade-off cannot be given yet.

Acknowledgements

Discussions with J. Heering, P. Hendriks, P. Klint, A. Verhoog and H. Walters markedly improved earlier versions of this specification. J. Heering, P. Klint and C. Koster made useful comments on the paper in general, while L. van der Gaag and P. Hendriks contributed to and commented on section 6.

References

[BHK85] J.A. Bergstra, J. Heering, and P. Klint (1985). *Algebraic definition of a simple programming language,* Report CS-R8504, CMCS, Amsterdam.

[BHK86] J.A. Bergstra, J. Heering, and P. Klint (1986). *Module algebra,* Report CS-R8617, CMCS, Amsterdam.

[BK82] J.A. Bergstra and J.W. Klop (1982). *Conditional rewrite rules: confluency and termination,* Report IW 198/82, MC, Amsterdam.

[BTh83] A.T. Berztiss and S. Thatte (1983). Specification and implementation of abstract data types, in *Advances in Computers,* Vol. 22, ed. M.C. Yovits, pp. 295-353, Pittsburgh, Pennsylvania.

[DE84] K. Drosten and H.-D. Ehrich (1984). *Translating algebraic specifications to Prolog programs,* Informatik-Bericht Nr. 84-08, Technische Universität Braunschweig.

[EM85] H. Ehrig and B. Mahr (1985). *Fundamentals of Algebraic Specifications 1,* EATCS Monographs on Theoretical Computer Science, Vol. 6, Springer-Verlag, Berlin.

[Gau84] M.-C. Gaudel (1984). *A first introduction to PLUSS,* Draft, Université de Paris-Sud, Orsay.

[GM84] J.A. Goguen and J. Meseguer (1984). *Equality, types, modules, and (why not?) generics for logic programming,* Journal of Logic Programming, Vol. 2, pp. 179-210.

[GPG81] J.A. Goguen and K. Parsaye-Ghomi (1981). Algebraic denotational semantics using parameterized abstract modules, in *Formalizing programming concepts,* pp. 292-309, ed. J.

Diaz & I. Ramos, Lecture Notes in Computer Science, Vol. 107, Springer-Verlag.

[Gor79] M.J.C. GORDON (1979). The denotational description of programming languages, Springer-Verlag, New York.

[Kla83] H.A. KLAEREN (1983). *Algebraische Spezifikationen: eine Einführung,* Springer-Verlag.

[Knu74] D.E. KNUTH (1974). *Structured programming with goto statements,* Report STAN-CS-74-416, Stanford.

[Loe84] J. LOECKX (1984). *Algorithmic specifications: a constructive method for abstract data types,* Report A84/03, Universität des Saarlandes.

[Wir83] M. WIRSING (1983). *A specification language,* Dissertation, Universität München.