SPECIFICATION LANGUAGE

J.R. Abrial S.A. Schuman B. Meyer
Consultant Intermetrics E.D.F

"You have a quarrel on hand, I see," said I, "with some of the algebraists of Paris; but proceed."

Edgar Allan Poe, The Purloined Letter

1. INTRODUCTION

The concept of specification language is now widely spread; formalising a problem is well recognised as a necessary step preceding any programming. The formalisation technique, however, is still the purpose for intensive research: the number of proposals in this field is a sufficient account of this fact. But a few basic principles seem to emerge and be generally agreed upon:

using a strict formalism inherited from mathematical practice

recognising the set theory as a sound basis for the formalisation

necessity of strong structuring of the formal text.

The proposed language takes its inspiration from these principles; it is especially indebted to the effort made within the last fifty years to present mathematical works in a satisfactory way.

The formal specification of a problem is provided by a strict statement of its contents written in a non natural language, in such a way that any future reader might have the same understanding of it. This necessitates that the given definition be exhaustive and unambiguous, in contrast with most of the non-formal, natural language specifications.

Experience shows that practical (industrial) problems seldom pose major theoretical difficulties; their complexity lies rather in the large number of intricate details that hide their in-depth nature and thus impede the discovery of clear solutions. Consequently, such problems raise the following question: how to emphasize the main points without sacrificing the details? The answer is of great importance, as the emergence of the "true" problem makes it possible to discover its decomposition into possibly known sub-problems. By doing this, the formalisation is no longer a lonely activity. It now belongs to a larger work performed by the same person, or better, a whole community: the specification language thus becomes a communication medium. One recognises a process that has been at work for more than two thousand years among mathematicians; returning to this source therefore seems to be an especially adequate step.

2. THE MATHEMATICAL TEXT

What is the organisation of the mathematical text? This is certainly a leading question for the beginning "formaliser." To illustrate, let us then open a book and analyse its contents.

The most obvious structure is shown by the decomposition of the book into chapters, sections, paragraphs, and so forth, each of them with a title and a tree structured number. The reason for this is quite obvious: it allows nonlinear reading of the text by using references, tables of contents, or other indications (sometimes a graph); in other words, everything that provides fruitful use of the book. These first elements constitute the so called utilisation text; it is by now almost standard.

The second structure encountered in the mathematical text is the one given by the various definitions, axioms, or theorems of a chapter. As above, all these elements have a name allowing further references. The content of definitions and the statement of axioms or theorems constitute the so called statement text; it is partially formalised, or even almost completely so, as in algebra, for example.

The third category is the proof text containing, as its name indicates, the proofs of the theorems. It is also only partially formalised.

Finally, in the midst of these texts, one may find all sorts of remarks, comments and the like, forming the explanation text.

It is interesting to note that, very often, these various texts are distinguished besides their content by the character set used to print them. Frequently, the utilisation text is printed with boldface characters, the statement text with italic characters, the proof text with normal, and the explanation text with small characters.

At a deeper level, the mathematical text is characterised by two trends general enough to require attention. Most of the time, a mathematical statement takes a generic (polymorphic, schematic) form, e.g., the statement in question contains set identifiers that are free. The following definition, for example, is generic with respect to A and B:

> "Let f be a function from A to B. One says that f is an **injection if two distinct elements of A have distinct images** through f."

The second trend of the modern mathematical text lies in the intensive usage of the notation of structure, as the order structure, the topological structure, the group or ring structure, and so forth. The definition of a structure consists of two distinct elements: firstly, the "typification," giving the (generic) definitions of its components; and, secondly, the axiomatisation, pointing out the characteristic properties of the components.

To reason in terms of structure has obvious advantages: this allows us to give definitions and to prove theorems at an abstract level. Then, if in some problem, one encounters an instance (a special case) of a known structure, one may apply all previous accumulated knowledge. Modern mathematics is actually a vast construction of structures.

3. LANGUAGE PRINCIPLES

The previous analysis, superficial as it was, allowed us to define some useful terms for defining the basic building principles of the language. It should allow us to write utilisation and statement texts; proof and explanation texts will take the form of mere comments written in natural language and formal language as well.

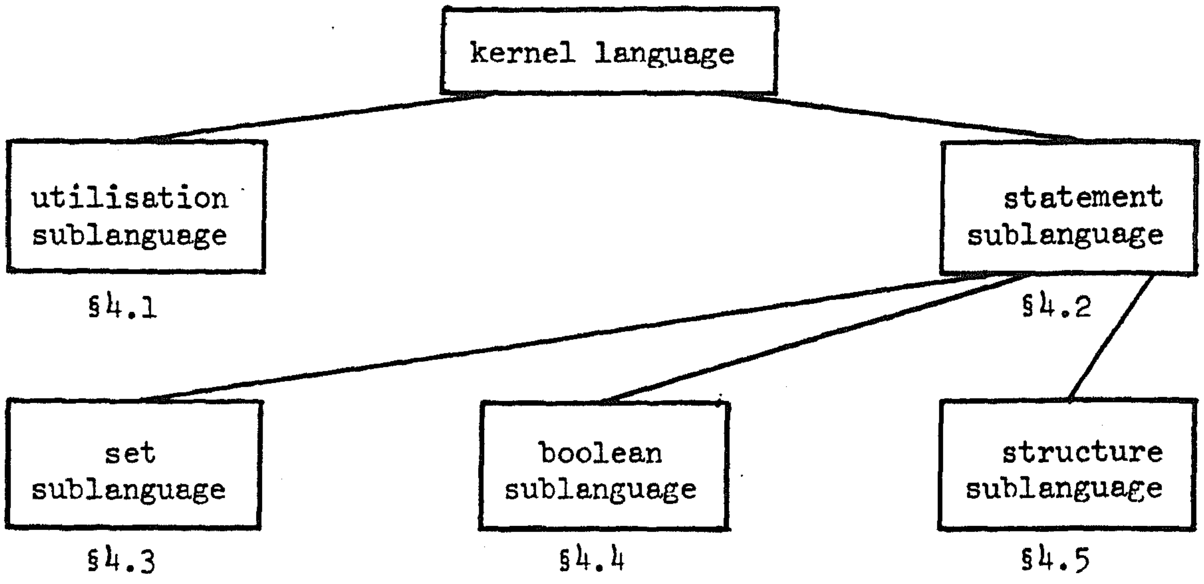
Thus a text will present itself as a set of named chapters, each of them with the names of locally used chapters. Each chapter is made of a list of possibly generic definitions or theorems. Definitions are given for sets or structures as previously encountered.

The specification of a problem is actually realised by writing a certain number of new chapters incrementing the set of old ones.

The language is now described by giving first the definition of a kernel language (§4), followed by the visibility rules (§5); some syntactic extensions are then given (§6) before a few last remarks on structures (§7).

4. THE KERNEL LANGUAGE

The kernel language may be decomposed as indicated by the following diagram:



The syntax is written in classical BNF with the following added conventions:

[ ] means an option

{ } means a zero or more times repetition

When the symbols {, }, [,], or | are used as linguistic symbols, they are quoted in order to avoid confusion with their metalinguistic usage.

The language definition does not contain any examples. The reader may skip at will to §9 and §10.

4.1 Utilisation sublanguage

The utilisation sublanguage describes the framework of a chapter.

Syntax

chapter ::= id = [use id_list] def body end id

id_list ::= id {, id}

The identifiers between use and def reference locally used chapters. The last identifier (after end) is the same as the first one (before =).

4.2 Statement sublanguage

The statement sublanguage describes the content of a chapter as a list of (possibly generic) definitions of sets, or structures

(here called classes), or theorems.

Syntax

```
body             ::= clause {; clause}
clause           ::= set_definition|theorem|class_definition
set_definition   ::= generic_name = set
theorem          ::= generic_name => bool
class_definition ::= generic_name = class
generic_name     ::= id ['['id_list']']
```

The identifiers possibly found in the list of a generic_name are formal generic parameters (set identifiers) of the corresponding definition or theorem.

4.3 Set sublanguage

The set sublanguage gives the various forms taken by a set expression.

Syntax

```
set    ::= set id_list for decl [where cond [given def]] end   |
           any id_list for decl [where cond [given def]] end   |
           subset (set)|'{'set{;set}'}'|null|set_id|
           object      |   set{,set}   |(set)

decl   ::= id_list : set {; id_list : set}

cond   ::= bool {; bool}

def    ::= id_list = set {; id_list = set}

set_id ::= [id.]id['['set{,set}']']
```

First, it is worth noting that the set sublanguage is "pure" in the sense that any distinction between sets and atoms does not exist: set elements, if any, always are sets.

The main set expressions correspond to the classical axioms of the theory: comprehension axiom, choice axiom, powerset axiom and extensive definition axiom.

The first form may be written:

```
set $id_1,\dots,id_n$ for
   $id_1$ : $set_1$;
   ...
   $id_n$ : $set_n$
[where
   cond
[given
   def]]
end
```

This derivation shows the existing constraints between the list of identifiers following the key-word set and the declarations; note that a declaration may be factorised as usual, but that all declarations must be independent of each other. If there are more than one identifier in the list, the defined subset is a subset of the Cartesian product of the various sets present in the declarations. The condition defines the characteristic predicates (bool) of the subset. One may provide some local definitions ("def") to lighten the writing of the predicates.

The second form, very close to the first, defines a "privileged" element of a non empty set. The declaration constraints are the same as above. Note that, axiomatically, the privileged elements provided by "two" equal and non empty sets are alike. Consequently, the operator any does not perform any random choice; such a set expression is therefore factorisable in a local definition.

The third form (key_word subset) corresponds to the mathematical $\mathcal{P}$ operator (the set of all subsets of a given set).

In the fourth form (extensive definition), the various set expressions must already denote elements belonging to the same set: it is not possible to construct any "heterogeneous" set.

In a set identifier:

[id_1].id_2['['$set_1,\dots,set_n$']']

"id_1" is a chapter identifier (if any ambiguity occurs on "id_2")

"id_2" is a set or a class identifier defined in the same chapter or in a locally used chapter

"set_1",...,"set_n" are actual generic parameters in equal number to the formal generic parameters (see §4.2) associated with the definition of "id_2".

Note that "set-id" may also be a simple identifier corresponding to a variable bound by a declaration or to a local definition.

The next set form corresponds to a class instance ("object". See §4.5).

4.4 Boolean sublanguage

The boolean sublanguage describes the boolean expressions.

Syntax

```
bool ::= not(bool)|bool or bool|
         set = set|set ε set   |
         finite (set) | (bool)
```

The third form (operator =) introduces the set equality and corresponds to the extensionality axiom of the theory (two sets are equal if they have the same elements).

In the fourth form (membership operator ε), when the left set corresponds to a list, then the right set is a subset of a Cartesian product constituted by as many sets as the left list has elements.

The fifth form (operator finite) corresponds to the axiom of infinity (there exists at least an infinite set).

4.5 Structure sublanguage

The structure sublanguage describes a class and how class instances may be constructed.

Syntax

```
class     ::= class [decl[where cond[given def]]] end  |
              subclass class_exp[class decl][where cond given def]]
                                                          end
class_exp ::= class-id {x class_id} | class_id{'|' class_id}
class_id  ::= set_id
object    ::= cons object_id [with def [given def]] end  |
              repl object_id with def [given def] end
object_id ::= set_id
```

A class definition may be derived as follows:

class
[id_1 : set_1; basic components
...
id_n : set_n
[where
cond axioms

[given
id'_1 = set'_1
... derived components
id'_m = set'_m]]]
end

A class definition is essentially an open definition: it is possible, from a given class, to define another one having more components and more axioms corresponding to the added basic components. The definition

subclass class_id class
class_body
end

implicitly contains all basic components, axioms and derived components of "class_id" as well as its own components and axioms. All definitions or theorems applicable to "class_id" may be used also, by extension, for the so defined subclass, but the converse is not possible.

A subclass may also be defined from several other classes (or subclasses) either in conjunction (operator x) or as alternatives (operator |).

The universal class

class end

has no components. Every class is therefore a subclass of the universal class.

It is worth noting the difference between subclass and subset. A subclass generally corresponds to a richer structure (more components, more axioms) than its constituent classes: in mathematics, for example, the topological group structure is richer than the topology or group structures alone. On the other hand, the notion of subset corresponds to a poorer construction than its "parent" set (it has less elements because of the added constraints); the notion of subset may be applied to classes as well: in mathematics, for example, the abelian groups are a (generic) subset of the groups.

In the same way, it is important to note the difference between the universal class (that has no component) and the empty class or set (that has no element).

To construct an object of a class, the value of each of its basic components is given. An object may be constructed globally (operator cons) or from a previous object (operator repl) by providing only some values that are supposed to replace some basic component values, the others remaining implicitly unchanged. In any case, the construction of an object corresponds as well to the statement of a theorem expressing the validity of the proposed values against the class axioms. In order to simplify the value expressions, one may use, in the object construction, any derived components already defined with the class or some local definitions (given def).

Note that the definition of a subclass without extra components or conditions

generic_name = subclass class_id end

builds a new subclass (of "class_id").

5. VISIBILITY RULES

New identifiers may be defined in the following conditions:

chapter identifier (at the beginning of a chapter. See §4.1).

set, theorem, or class identifier (heading a generic_name. See §4.2).

formal generic parameter (within a generic_name. See §4.2).

bound variable identifier (within a declaration. See §4.3).

basic or derived class component identifier. (See §4.5).

local definition identifier. (See §4.3 and §4.5).

Scope rules

The scope of an identifier defines where it may be used.

The scope of a chapter identifier is universal.

The scope of a set, theorem, or class identifier covers the chapter where it is defined as well as the chapters where this chapter is used (use).

The scope of a formal generic parameter covers the corresponding definition or theorem.

The scope of a bound variable identifier covers the corresponding construction (See §4.3).

The scope of a basic or derived class component identifier is the same as the one of this class identifier (basic components of a class must be independent of each other, however).

The scope of a local definition identifier covers the corresponding construct.

Non recovering rule

Except for set or class identifiers defined in different chapters, no identifier shall be ambiguous within its scope. A dot notation is used when ambiguities occur in using set or class identifiers (see "set_id" §4.3).

Non recursivity rule

No definition shall be directly or indirectly recursive.

6. KERNEL LANGUAGE EXTENSIONS

The whole language is obtained by extending the kernel language; some useful syntactic construct may be replaced by simpler ones. These equivalences are denoted by special syntactic equations

new syntactic construct ::=:: syntactic construct

where "::=::" may be read as "is syntactically equivalent to". A syntactic construct is represented by an incomplete derivation containing some non terminal symbols acting as metalinguistic variables that may be indexed or primed. Lists are denoted by "...". These extensions concern the set, boolean and structure sublanguages.

SPECIFICATION LANGUAGE

6.1 Boolean sublanguage extensions

Boolean sublanguage extensions introduce classical boolean operators as well as existential and universal quantifiers.

Syntactic extensions

```
bool_1 => bool_2     ::=:: not (bool_1) or bool_2
bool_1 and bool_2    ::=:: not (not (bool_1) or not (bool_2))
bool_1 <=> bool_2    ::=:: (bool_1 => bool_2) and (bool_2 => bool_1)
set_1 ≠ set_2        ::=:: not (set_1 = set_2)
set_1 ∉ set_2        ::=:: not (set_1 ε set_2)

exist id_list for  ::=:: (set id_list for
   spec                      spec
end                       end) ≠ null
existl id_list for ::=::  exist id for
   spec                      id : set
end                       where
                            (set id_list for
                               spec
                            end) = {id}
                          end
```

where "spec" is defined by:

```
   spec ::= decl [where cond [given def]]

forall id_list for ::=:: not (exist id_list for
   decl                         decl
[where                        where
   cond_1]                      [cond_1;]
then                            not cond_2
   cond_2                     [given
[given                          def]
   def]                       end)
end

bool_1;...;bool_n     ::=:: bool_1 and ... and bool_n
```

6.2 Set sublanguage extensions

Set sublanguage extensions introduce a simplified notation for the Cartesian product, a simplified notation for the "privileged" element of a set, a notation for the set of relations, total or partial functions from a set to another one and the classical functional and relational notations.

Syntactic extensions

set_1 x ... x set_n ::=:: set $\text{id}_1, \ldots, \text{id}_n$ for
id_1 : set_1;
...
id_n : set_n
end

any (set_1 x...x set_n) ::=:: any $\text{id}_1, \ldots, \text{id}_n$ for
id_1 : set_1;
...
id_n : set_n
end

set ↔ set' ::=:: subset (set x set')

The above notation denotes the set of binary relations from one set to another.

rel id_list ↔ id_list' for ::=:: set id_list, id_list' for
spec spec
end end

The above notation allows definition of a binary relation with a predicate. Note that, if "id_list" or "id_list'" have several identifiers, then several binary relations may be defined this way for the same right syntactic construction. This is due to the fact that a Cartesian product made of more than two sets may be "cut" in different ways.

rel_id (set ↔ set') ::=:: ((set).(set')) ϵ rel_id

In the above notation "rel_id" designates a relation identifier.

```
rel_id ($set_1$ x...x $set_n$) ::=:: set id' for
                                   id' : codom (rel_id)
                                 where
                                   exist $id_1$,...,$id_n$ for
                                     $id_1$ : $set_1$;
                                     ...
                                     $id_n$ : $set_n$
                                   where
                                     rel_id (($id_1$,...,$id_n$) ↔ id')
                                   end
                                 end
```

The above notation defines the image of a set through a given relation. The expressions "dom (rel_id)" and "codom (rel_id)" designate the domain and codomain of a relation (denoted by the identifier "rel_id").

```
rel_id ($set_1$,...,$set_n$) ::=:: rel_id ({($set_1$,...,$set_n$)})
{$set_1$ ↔ $set'_1$;...;     ::=:: {(($set_1$),($set'_1$));...;
 $set_n$ ↔ $set'_n$}               (($set_n$),($set'_n$))}
```

The above notation allows for the extensive definition of a binary relation.

Similar notations are now given for functions.

```
set → set' ::=:: set id'' for
                   id'' : set ↔ set'
                 where
                   forall id for
                     id : set
                   then
                     existl id' for
                       id' : set'
                     where
                       id'' (id ↔ id')
                     end
                   end
                 end
```

The above notation defines the set of total functions from one set to another.

```
set - set' ::=:: set id'' for
                   id'' : set ↔ set'
                 where
                   exist id for
                     id : subset (set)
                   where
                     id'' = id → set'
                   end
                 end
```

The above notation defines the set of partial functions from one set to another.

```
func id_1,...,id_n → id'_1,...,id'_m for ::=:: rel id_1,...,id_n ↔
                                                  id'_1,...,id'_m for
   id_1 : set_1;                                 id_1 : set_1;
   ...                                           ...
   id_n : set_n;                                 id_n : set_n;
   id'_1 : set'_1;                               id'_1 : set'_1;
   ...                                           ...
   id'_m : set'_m                                id'_m : set'_m
[where                                         where
   cond]                                        [cond;]
then                                             id'_1 = set''_1;
   id'_1 = set''_1;                              ...
   ...                                           id'_m = set''_m
   id'_m = set''_m                             [given
                                                 def]
[given
   def]                                        end
end
```

The above notation defines a function by one or several formulas. Note that "cond" shall not contain any of the bound variable

SPECIFICATION LANGUAGE

identifiers "id'_1",...,"id'_m".

```
func id_list → id_list' for ::=:: rel id_list ↔ id_list' for
    decl                              decl
[where                            where
    cond]                           [cond;]
when $bool_1$ then                   $bool_1$ => $bind_1$ or
    $bind_1$                          ...
...                                  $bool_n$ => $bind_n$ [or
                                     not ($bool_1$ and ... and $bool_n$)
                                                  => bind]

when $bool_n$ then
    $bind_n$                         [given
[else                                    def]
    bind]                            end
[given
    def]
end
```

The above notation defines a function by case. Note that the various predicates "$bool_1$",...,"$bool_n$" shall be exclusive (no non-determinism) and that, in the case of a missing "else", their disjunction shall be true. The non terminal symbol "bind" may be defined by

```
bind ::= id = set {; id = set}
```

```
func_id ($set_1$,...,$set_n$) ::=:: any id' for
                                  id' : codom (func_id)
                              where
                                ((set_1,...,$set_n$),id') ε func_id
                              end
```

The above equation introduces the usual functional notation ("func_id" is a function identifier)

```
func_id ($set_1$ x...x $set_n$) ::=:: set id' for
                                    id' : codom (func_id)
                                  where
                                    exist $id_1$,...,$id_n$ for
```

```
                              $id_1 : set_1$;
                              ...
                              $id_n : set_n$
                            where
                             (($id_1$,...,$id_n$),id') ε func_id
                            end
                        end
```

The above notation defines the image of a set through a given function.

```
{$set_1$ → $set'_1$;...;  ::=:: {$set_1$ ↔ $set'_1$;...;
 $set_n$ → $set'_n$              $set_n$ ↔ $set'_n$
```

The above notation defines a function extensively. Note that, obviously, the expressions "set_1",...,"set_n" must all have different values as well as the expressions "set'_1",...,"set'_n".

```
subst func_id with ::=:: func id → id' for
   $set_1$ → $set'_1$;          id : dom (func_id);
   ...                        id' : codom (func_id)
   $set_n$ → $set'_n$          when id = $set_1$ then
                                 id' =  $set_1$
end                         ...
                            when id = $set_n$ then
                                 id' = $set'_n$
                            else
                                 id' = func_id(id)
                            end
```

The above notation allows definition of a function by changing some of the values of a given function, leaving the others unchanged. Note that the expressions "set_1",...,"set_n" shall have different values and that the expressions "set'_1",...,"set'_n" shall be such that the result still is a function.

Some remarks

The general form of a function definition is, as seen previously:

```
id[$id''_1$,...,$id''_p$] = func $id_1$,...,$id_n$ → $id'_1$,...,$id'_m$ for
                            ...
                          end
```

The identifiers "id''_1",...,"id''_p" denote formal generic parameters whereas "id_1",...,"id_n" denote the formal parameters of the function.

An invocation of this function has the following form

$id(set_1,\ldots,set_n)$

where "set_1",...,"set_n" denote the actual parameter of the function. In this case, it is not necessary to provide some values for the actual generic parameters because they are implicitly defined within the expressions "set_1",...,"set_n".

Whenever $n = 2$, it is sometimes useful to denote a function invocation using an infixed form:

set_1 id set_2

In order to indicate this special usage, the identifier "id" is replaced in its definition by

op (id).

Any use of "id", out of an invocation, must be written

op (id) [$set''_1,\ldots,set''_p$]

where "set''_1",...,"set''_p" denote the actual generic parameters.

This special notation may be used for binary relations as well.

6.3 Structure sublanguage extension

The structure sublanguage extension allows us to define a discrete set composed of a certain number of explicitly denoted elements.

Syntactic extensions

```
$id = \{id_1;\ldots;id_n\}$ ::=:: $id_1$ = class end;
                   ...
                   $id_n$ = class end;
                   $id'$ = subclass $id_1$ |...| $id_n$ end;
                   id  = set $id''$ for
                           $id''$ : set ($id'$)
                         where
                           $id'' = id_1$ or
                           ...
                           $id'' = id_n$
                         end
```

7. SOME REMARKS ABOUT CLASSES

Recall that a class is defined by (See §4.5)

```
$id(id''_1,...,id''_\ell)$ = class
                            $id_1 : set_1$;
    formal                  ...                   basic components
    generic                 $id_n : set_n$
    parameters
                          where
                            cond                  axioms
                          given
                            $id'_1 = set'_1$;
                            ...                   derived components
                            $id'_m = set'_m$
                          end
```

An expression like

$$id[set''_1,...,set''_\ell]$$

denotes the set of objects of the class "id" for the values "set''_1,...,"set''_ℓ" of the generic parameters. Such a set may be used in a declaration

$$id''' : id[set''_1,...,set''_\ell]$$

where the identifier "id'''" denotes a bound variable (see §4.3), or even a component of yet another class (see §4.5). In order to reference a basic or derived component of the object "id'''", a <u>functional</u> notation is used, i.e.:

$$id_i(id''') \text{ or } id_j(id''')$$

Class component identifiers denote (generic) unary functions on the objects of the class. This notation may be applied for explicitly constructed objects as well (operator <u>cons</u> or <u>repl</u>).

Finally, note that within an explicit replacement construction (operator <u>repl</u>), or within a class definition, the usage of a component identifier alone is sufficient to refer to the component in question (this is a convention similar to the one used in PASCAL within a "<u>with</u>" construct).

8. SYMBOLS AND KEY WORDS

The symbols and key-words of the kernel language are the following:

= , ; => : . ε x | ( ) [ ] { }

any
class
cons
def
end
finite
for
given
not
null
or
repl
set
subclass
subset
use
where
with

The symbols and key-words of the extended language are the following

<=> ≠ ∉ ↔ → ↛

and
codom
dom
else
exist
existl
forall
func
op
rel
subst
when
with

9. BASIC CHAPTERS

We now present a few basic "chapters" that will be extensively used in later applications. The first of these chapters, named SET, defines the standard generic operators of elementary set theory. It is worth noting that these operators apply to binary relations or functions as well because they are themselves sets.

The next chapter, named REL, uses SET and defines the standard operators of binary relation theory, namely inversion, composition, functionality (to go possibly from a relation to a function), and products. These operators apply to functions as well, because they are special cases of binary relations.

A third chapter named FUNC uses SET and REL, and defines special kinds of functions, namely injections, surjections, and bijections. It also defines the restriction of a function and the constant function.

```
SET =
   def
      op(∪)[X] = func S1,S2 → S3 for
                    S1,S2, S3 : subset(X)
                 then
                    S3 = set x for x : X where
                            x ε S1 or x ε S2
                         end
                 end;
      op(∩)[X] = func S1,S2 → S3 for
                    S1,S2,S3 : subset(X)
                 then
                    S3 = set x for x : X where
                            x ε S1 and x ε S2
                         end
                 end;
      union[X] = func SS → S for
                    SS : subset(subset(X));
                    S  : subset(X)
                 then
                    S = set x for x : X where
                           exist S' for S' : SS where
                              x ε S'
                           end
                        end
                 end;
      inter[X] = func SS → S for
                    SS : subset(subset(X));
                    S  : subset(X)
                 then
                    S = set x for x : X where
                           forall S' for S' : SS then
                              x ε S'
                           end
```

```
                      end
               end;
    op(-)[X] = func S1,S2 → S3 for
                  S1,S2,S3 : subset(X)
               then
                  S3 = set x for x : X where
                          x ε S1 and x ⊄ S2
                       end
               end;
    op(⊂)[X] = rel S1 ↔ S2 for
                  S1,S2 : subset(X)
               where
                  forall x for x : S1 then
                     x ε S2
                  end
               end;
    op(⊄)[X] = rel S1 ↔ S2 for
                  S1,S2 : subset(X)
               where
                  not(S1⊂S2)
               end;
partition[X] = set SS for
                  SS : subset(subset(X))
               where
                  union(SS) = X;
                  forall S1,S2 for
                     S1,S2 : SS
                  where
                     S1 ≠ S2
                  then
                     S1∩S2 = null
                  end
               end;
```

```
     proj1[X,Y] = func x,y → x' for
                      x,x' : X;
                      y    : Y;
                  then
                      x' = x
                  end;

     proj2[X,Y] = func x,y → y' for
                      x    : X;
                      y,y' : Y
                  then
                      y' = y
                  end
end SET

REL =
    use SET def
        inv[X,X']    = func r → r' for r : X ↔ X'; r' : X' ↔ X then
                           r' = rel x' ↔ x for x' : X'; x : X where
                                    r(x ↔ x')
                                end
                       end;

    op(∘)[X,Y,Z]     = func r2,r1 → r3 for
                           r1 : X ↔ Y; r2 : Y ↔ Z; r3 : X ↔ Z
                       then
                           r3 = rel x ↔ z for x : X; z : Z where
                                    exist y for y : Y where
                                        r1(x ↔ y); r2(y ↔ z)
                                    end
                                end
                       end;

        ident[X]     = rel x ↔ x' for x,x' : X where x = x' end;
```

```
functional[X,Y]     = set r for r : X ↔ Y where
                          r'(Y) = X; (r ∘ r') ⊂ ident[Y]
                        given
                          r' = inv(r)
                        end;

  function[X,Y]     = func r → f for r : X ↔ Y; f : X → Y where
                          r ε functional[X,Y]
                        then
                          f = func x → y for x : X; y : Y then
                                y = any (r(x))
                              end
                        end;

op(prod)[A,B,C,D]   = func r1,r2 → r3 for
                          r1 : A ↔ B; r2 : C ↔ D;
                          r3 : A x B ↔ C x D
                        then
                          r3 = rel a,c ↔ b,d for
                                  a : A; b : B; c : C; d : D
                                where
                                  r1(a ↔ b); r2(c ↔ d)
                                end
                        end;

  op(&)[A,B,C]      = func r1,r2 → r3 for
                          r1 : A ↔ B; r2 : A ↔ C;
                          r3 : A ↔ B x C;
                        then
                          r3 = rel a ↔ b,c for
                                  a : A; b : B; c : C
                                where
                                  r1(a ↔ b); r2(a ↔ c)
                                end
                        end;

end REL
```

```
FUNC =
    use SET, REL def
        inj[X,Y]         = set f for f : X → Y where
                               inv(f) ∘ f = ident[X]
                           end;

        surj[X,Y]        = set f for f : X → Y where
                               f ∘ inv(f) = ident[Y]
                           end;

        bij[X,Y]         = inj[X,Y] ∩ surj[X,Y];

        inverse[X,Y]     = func f → f' for f : bij[X,Y]; f' : Y → X then
                               f' = function(inv (f))
                           end;

restriction[X,Y]         = func f,S → f' for
                               f : X → Y; S : subset (X); f' : X ↛ Y
                           then
                               f' = func x → y for x : S; y : Y then
                                      y = f(x)
                                    end
                           end;

        const[X,Y]       = func S, y → f for
                               S : subset (X); y : Y; f : X ↛ Y
                           then
                               f = func x → y' for x : S; y' : Y then
                                     y' = y
                                   end
                           end;

end FUNC
```

The next two chapters define the natural numbers and the sequences. NAT, the first of them, starts by introducing generically the cardinal of a set S as the set of set S' equinumerous with S; then the set of natural numbers is the set of finite cardinals. The classical relations "≤" and '<' and the operation successor are then defined before the iterate of a function. This allows us to give the definition of the basic arithmetic operations.

The chapter SEQ generically defines the sequences as the set of functions whose domains are segments of the natural numbers; i.e. {0,1,...,n}. It is then easy to define the concatenation (operator *), "first" and "tail" operators. The chapter ends with definitions of a sorted sequence of natural numbers and the set of sub-sequences of a given sequence.

```
NAT =
    use SET, REL, FUNC def
       equinumerous[X] = rel S ↔ S' for S,S' : subset (X) where
                            bij [S,S'] ≠ null
                         end;

       card[X]         = func S → SS for S : subset (X); SS : subset
                                                  (subset(X)) then
                            SS = equinumerous(S)
                         end;

       NAT[X]          = set n for n : subset (subset(X)) where
                            not (finite(X));
                            exist S for S : subset(X) where
                               card(S) = n; finite(S)
                            end
                         end;

       0[X]            = card(null);

       th1[X]         => 0[X] = null;

       th2[X]         => 0[X] ε NAT[X];

       op(≤)[X]        = rel n1 ↔ n2 for n1,n2 : NAT[X] where
                            exist S1,S2 for S1,S2 : subset(X) where
                               card(S1) = n1; card(S2) = n2;
                                              inj[S1,S2] ≠ null
                            end
                         end;

       op(<)[X]        = rel n1 ↔ n2 for n1,n2 : NAT[X] where
                            n1 ≤ n2; n1 ≠ n2
                         end;
```

```
    succ[X]          = func n1 → n2 for n1,n2 : NAT[X] then
                           n2 = card (S∪{x})
                       given
                           S = any (n1); x = any (X - S)
                       end;

    relpred[X]       = inv(succ[X]);

    th3[X]          => relpred[X] ε (NAT[X] - {0[X]}) → NAT[X];

    pred[X]          = function(relpred[X]);

-- from now on the generic parameter X is omitted

 induction_theorem => forall S for S : subset(NAT) where
                           0 ε S;
                           forall n for n : S then
                              succ(n) ε S
                           end
                       then
                           S = NAT
                       end;

    recursion[X]     = function(rel y,g ↔ f for
                                     y : Y; g : Y → Y; f : NAT → Y
                                 where
                                     f(0) = y;
                                     f ∘ succ = g ∘ f
                                 end);
```

$$\begin{tikzcd} n \arrow[r, "f"] \arrow[d, "succ"'] & y \arrow[d, "g"] \\ n+1 \arrow[r, "f"'] & y' \end{tikzcd}$$

```
    iter[Z]          = func h → f for
                           h : Z → Z; f : NAT → (Z → Z)
                       then
                           f = recursion(y,g)
                       given
                           y = ident[Z];
                           g = func h1 → h2 for h1,h2 : Z → Z then
                                  h2 = h ∘ h1
                               end
                       end;
```

```
-- f(0) = ident[Z]
-- f(n+1) = g(f(n)) = h ∘ f(n) = .... = h^(n+1)
      op(+)          = func n1,n2 → n3 for n1,n2,n3 = NAT then
                           n3 = iter(succ)(n1)(n2)
                       end;
      op(x)          = func n1,n2 → n3 for n1,n2,n3 : NAT then
                           n3 = iter(op(+))(n1)(n2)
                       end;
      op(exp)        = func n1,n2 → n3 for n1,n2,n3; NAT then
                           n3 = iter (op(x))(n1)(n2)
                       end;
      op(-)          = function(rel n1,n2 ↔ n3 for n1,n2,n3 :
                                                   NAT where
                                     n2≤n1; n1 = n2+n3
                                 end);
      div_mode       = function(rel a,b ↔ q,r for a,b,q,r :
                                                   NAT where
                                     b≠0; a = (b x q) + r; r<b
                                 end);
      op(div)        = proj1[NAT,NAT] ∘ div_mod;
      op(mod)        = proj2[NAT,NAT] ∘ div_mod;
      1 = succ(0); 2 = succ(1); 3 = succ(2); 4 = succ(3);
      5 = succ(4); 6 = succ(5); 7 = succ(6); 8 = succ(7);
      9 = succ(8)
end NAT
SEQ=
    use SET, REL, NAT def
      segment        = func n → S for n : NAT; S : subset(NAT) then
                           S = set i for i : NAT where i<n end
                       end;
```

```
  seq[X]             = set s for s : NAT ⇸ X where
                          dom (s) ε segment(NAT)
                       end;

  length[X]          = func s → n for s : seq[X]; n : NAT then
                          n = card(dom(s))
                       end;

  op(*)[X]           = func s1,s2 → s3 for s1,s2,s3 : seq[X] then
                          s3 = s1∪(s2 ∘ f)
                       given
                          f = iter(pred)(length(s1))
                       end;

  first[X]           = func s → x for s : seq[X]; x : X where
                          s ≠ null
                       then
                          x = s(0)
                       end;

  cat[X]             = func x, s1 → s2 for s1,s2 : seq[X];
                                              x : X then
                          s2 = {0 → x}  * s1
                       end;

  tail[X]            = func s1 → s2 for s1,s2 : seq[X] where
                          s1 ≠ null
                       then
                          s2 = (s1 - {0 → first(s1)}) ∘ succ
                       end;

  th[X] => cat[X] ∘ (first[X] & tail[X]) = ident[seq[X]];

associated_rel[X]    = func s → r for
                          s : seq[X];
                          r : X ↔ X
                       then
                          r = s ∘ succ • inv(s)
                       end;
```

```
    sorted          = set s for s : seq[NAT] where
                         associated_rel(s)⊂op(≤)
                      end;

    sub_seq[X]      = rel s1 ↔ s2 for s1,s2 : seq[X] where
                         exist s3 for s3 : sorted where
                            s1 = s2 • s3
                         end
                      end

end SEQ
```

As a last syntactic extension, an explicit sequence is denoted by

$<x_1;x_2;\ldots;x_n>$

The chapter MON defines the monoids as a sub-class of a sub-group (a binary commutative operation). Classical examples of monoids are then given, followed by the definition of the extensions of binary operations (with neutral element) to sequences.

```
MON =
    use SET, REL, FUNC, NAT, SEQ def
       subgroup[S]      = class
                             oper : S x S → S
                          where
                             oper • (oper prod ident[S]) = oper •
                                                  (ident[S] prod oper)
                          end;

       monoid[S]        = subclass subgroup[S] class
                             u : S
                          where
                             oper • (const(S,u) & ident[S]) = ident[S];
                             oper • (ident[S] & const(S,u)) = ident[S]
                          end;

       example1[X]      = cons monoid[subset(X)] with
                             oper = op (U)[X]; u = null
                          end;
```

```
    example2[X]      = cons monoid[subset(X)] with
                          oper = op(∩)[X]; u = X
                       end;

    example3[X]      = cons monoid[X → X] with
                          oper = op (∘)[X,X,X]; u = ident[X]
                       end;

    example4         = cons monoid[NAT] with
                          oper = op(+); u = 0
                       end;

    example5         = cons monoid[NAT] with
                          oper = op(x); u = 1
                       end;

    example6[X]      = cons monoid[seq[X]] with
                          oper = op(*)[X]; u = null
                       end;

    extension[X]     = function(rel m ↔ f for
                                     m : monoid[X];
                                     f : seq[X] → X
                                  where
                                     f = f1 U f2
                                  given
                                     f1 = const({null}, u(m));
                                     f2 = oper(m) • f3;
                                     f3 = first[X] & (f • tail[X])
                                  end);

    sigma = extension (example4);
    pi    = extension (example5);
    comp[X] = extension (example3[X]);
    conc[X] = extension (example6[X])

end MON
```

Finally, the chapter RELATIONS gives the classical definitions of the transitive closure of a binary relation, of symmetry, transitivity, reflexivity, and so forth, as well as preorder, equivalence, order, and so forth, for binary relations.

```
RELATIONS =
    use SET, REL, NAT def
        rel_iter[Z]     = func r → f for
                               r : Z ↔ Z; f : NAT → (Z ↔ Z)
                           then
                               f = recursion(y,g)
                           given
                               y = ident[Z];
                               g = func r1 → r2 for r1,r2 : Z ↔ Z then
                                       r2 = r • r1
                                   end
                           end;

-- f(0) = ident[Z]
-- f(n+1) = g(f(n))     = r • f(n) = ... = r^(n+1)

       closure[Z]       = func r → r' for r,r': Z ↔ Z then
                               r' = union(rel_iter(r)(NAT))
                           end;

-- closure(r)           = ident[Z] U r U r^2 U ... U r^n U ...

        th[Z]           = forall r for r : Z ↔ Z then
                               ident[Z] ⊂ closure(r);
                               r • closure(r) ⊂ closure(r)
                           end;

        sym[X]          = set r for r : X ↔ X where r = inv(r) end;

        trans[X]        = set r for r : X ↔ X where (r • r) ⊂ r
                                                     end;

        reflex[X]       = set r for r : X ↔ X where ident[X] ⊂ r
                                                     end;

        asym[X]         = set r for r : X ↔ X where(r ∩ inv(r)) =
                                                     null end;
```

```
    antisym[X]       = set r for r : X ↔ X where (r ∩ inv(r)) =
                                                ident[X] end;

    irreflex[X]      = set r for r : X ↔ X where (r ∩ ident[X])=
                                                null end;

    total[X]         = set r for r : X ↔ X where (r ∪ inv(r)) =
                                                X x X end;

    preorder[X]      = trans[X] ∩ reflex[X];

    equiv[X]         = preorder[X] ∩ sym[X];

    order[X]         = preorder[X] ∩ antisym[X];

    strict_order[X] = trans[X] ∩ irreflex[X];

    total_order[X]  = order[X] ∩ total[X];

    th1[X]          => op (≤) ε total_order (NAT);

    th2[X]          => op (<) ε strict_order (NAT);

    th3[X]          => op (⊂)[X] ε order(subset(X));

    th4[X,Y,Z]      => forall r1,r2,r3 for
                           r1 : X ↔ Y; r2 : Y ↔ Z; r3 : X ↔ Z
                       then
                        ((r2 ∘ r1) ∩ r3 = null) => (inv(r3) • r2)
                                                ∩ inv(r1) = null)

                       end;

    th5[X]          => strict_order[X] => asym[X]

end RELATIONS
```

10. EXAMPLES

The preceding "chapters" were extensions of the language in order to constitute an elementary mathematical background. This section attacks more "realistic" problems in various areas of programming: an editing problem (§10.1) represents "classical" programming, a system problem (§10.2), and a garbage collector specification (§10.3) cover the "system" programming field, and finally a very simple algebraic language definition (§10.4) goes towards the language design area.

Note : All "basic chapters" are implicitly used in the examples.

10.1 An editing problem

The first problem that we try to specify is a simple editing problem. It may be informally stated as follows: to transform a string of characters by replacing all its substrings of consecutive blank characters by a single blank character. This problem is interesting for several reasons:

- it is simple enough so that anyone may understand it immediately
- it is a practical and classical problem illustrating a large class of editing problems
- the corresponding program is not very difficult to write although its complete proof is not that trivial.

Before attacking the problem we need to write a small "theoretical" chapter defining a few concepts of the fixed point theory. These concepts may be informally defined as follows:

Let f be a function from X to X; if, for all x, there exists a natural number n such that

$$f^{n+1}(x) = f^{n}(x)$$

then any sequence

$$x,\ f(x),\dots,f^{i}(x),\dots$$

is stationary after a certain number n depending upon x, i.e., all further elements of the sequence are the same and said to be *the* stationary element of x through f. The corresponding function is called the *limit* of f. Note that not all functions from X to X have such a limit.

In order to ensure that a function f has a limit, it is sufficient to find a *variant*, i.e., a function g from X to the natural numbers such that

$$\text{if } f(x) = x \text{ then } g(x) = 0$$

$$\text{if } f(x) \neq x \text{ then } g(x) \neq 0 \text{ and } g(f(x)) < g(x)$$

A binary relation R is said to be consistent with respect to function f from X to X, if, for any x, the following holds:

$$x \mathrel{R} f(x)$$

A very useful theorem finally states that if R is consistent with respect to f, then R* (the transitive closure of R) is consistent with respect to the limit of f (if any).

```
MINI_FIXED_POINT_THEORY =
    def
       limit[X]          = func f → f' for f,f' : X → X where
                              forall x for x : X then
                                 r(x) ≠ null
                              end
                           then
                              f' = func x → x' for x,x' : X then
                                      x' = iter(f)(i)(x)
                                   given
                                      i = any(r(x))
                                   end
                           given
                              r = rel x ↔ n for x : X; n : NAT where
                                     iter(f)(n+1)(x) = iter(f)(n)(x)
                                  end
                           end;

       variant[X]        = rel f ↔ g for f : X → X; g : X → NAT where
                              forall x for x : X then
                                 (f(x) = x) => (g(x) = 0);
                                 (f(x) ≠ x) => (g(x) > 0 and
                                                 g(f(x)) < g(x))
                              end
                           end;

   variant_theorem[X]  => forall f for f : X → X where
                              variant(f) ≠ null
                           then
                              f ε dom(limit[X])
                           end;
```

```
invariant_theorem[X]  => forall f,r for
                            f : dom(limit[X]);
                            r : X ↔ X
                         where
                            f ⊂ r
                         then
                            limit(f) ⊂ closure(r)
                         end

     end MINI_FIXED_POINT_THEORY
```

The specification of the editing problem constitutes another chapter using MINI_FIXED_POINT_THEORY. A class "state" is first defined as that containing three components: "b" (for blank), and "in" and "out", that are a sequence of characters. The purpose of the specification is to define the properties of "out" with regard to "in", i.e., "out" shall not contain two consecutive blank characters (this is specified in "spec1", a subclass of state) and shall be "equivalent" to "in" (this is described in "spec2", a subclass of "spec1"): two sequences of characters are said to be equivalent if they only differ by the (non null) length of their subsequences of consecutive blank characters.

A function "one-step" is then given that is proven (i) to leave "spec1" invariant, (ii) to have a limit, (iii) to be such that "equivalent" is consistent with respect to it (remember that the concepts of limit and consistency have been defined in the previous "chapter"). As a consequence, the limit of "one-step" is proven to fulfil the specification of the problem.

In order to construct a real program, the function "one-step" is then decomposed into two other functions, namely "step0", handling null "out" sequences, and "step1", handling non null "out" sequences.

The PASCAL program is then written as a final step of the specification and construction process.

```
EDITING_PROBLEM =
     use MINI_FIXED_POINT_THEORY def
        state[C]         = class
                              b : C;
                              in,out : seq[C]
                           end;
```

```
spec1[C]       = subclass state[C] where
                    out ε no_two_consecutive_blanks
                 given
                    no_two_consecutive_blanks =
                       set s for s : seq[C] where
                          not(associated_rel(s)(b ↔ b))
                       end
                 end;

spec2[C]       = subclass spec1[C] where
                    equivalent_string(in ↔ out)
                 given
                    equivalent_string = closure(r);
                    r = rel s1 ↔ s2 for s1,s2 : seq[C]
                                           where
                          exist x,y,b1,b2 for
                             x,y : seq[C];
                             b1,b2 : seq[{b}] - null
                          where
                             s1 = x*b1*y;
                             s2 = x*b2*y
                          end
                        end
                 end;

one_step[C]     = func s → s' for
                     s,s' : spec1[C]
                  when in(s) = null then
                     s' = s
                  when in(s) ≠ null and
                  out(s)*<first(in(s)) > ε no_consecutive_
                                          blanks(s) then
                  s' = repl s'1 with
                          out = out*<first(in)>
                       end
```

```
                         else
                         s' = s'1
                         given
                         s'1 = repl s with
                                     in = tail(in)
                             end
                         end;

  var[C]               = func s → n for s : specl[C]; n : NAT then
                           n = length(in(s))
                         end;

  thl[C]              => var[C] ε variant(one_step)

  th2[C]              => one_step[C] ε dom(limit[specl[C]])

-- after thl and variant_theorem

  equivalent_state[C] =
     rel s ↔ s' for s,s' : specl[C] where
        equivalent_string(s)(out(s)*in(s) ↔ out(s')*in(s'))
     end;

  th3[C] => one_step[C] ⊂ equivalent_state[C];

-- after definition of one_step[C]

  th4[C] => limit(one_step[C]) ⊂ equivalent_state[C];

-- after th3[C] and invariant_theorem.  Note that

-- equivalent_state[C] = closure(equivalent_state[C])

  normalise[C]         = func s → s' for
                           s,s' = specl[C]
                         where
                           out(s) = null
                         then
                           s' = limit(one_step[C])(s)
                         end;
```

```
th5[C]          => forall s for s : spec1[C] where
                      out(s) = null
                   then
                      s' ε spec2[C]
                   given
                      s' = repl s with
                              out = out(normalise(s))
                           end
                   end;

-- after th4[C] and definition of normalise.  Towards a Pascal
   program

spec1'[C]         = subclass spec1[C] class
                       ch : C
                    end;

step0[C]          = func s → s' for s,s' = spec1'[C] where
                       out(s) = null
                    when in(s) = null then
                       s' = s
                    else
                       s' = repl s'1 with
                               out = out * <ch1>; -- write (ch1)
                               ch = ch1
                            end
                    given    -- read (ch1)
                       s'1 = repl s with
                                in = tail(in)
                             end;
                       ch1 = first(ch)
                    end;

step1[C]          = func s → s' for s,s' : spec1'[C] where
                       out(s) ≠ null;
                       ch(s) = last(out(s))
                    when in(s) = null then
                       s' = s
```

```
                         when in(s) ≠ null and(ch(s) ≠ b or chl ≠ b)
                            then
                            s' = repl s'l with
                                    out = out * <chl>; -- write(chl)
                                    ch = chl
                                 end
                         else
                            s' = s'l
                         given -- read (chl)
                            s'l = repl s with
                                     in = tail(in)
                                  end;
                            chl = first(in(s))
                         end;

       normalise'[C]   = limit(step1[C]) • step0[C]

end EDITING_PROBLEM
```

The corresponding PASCAL program is the following

```
program normalise(input,output);

const b = ' ';

var ch,chl : char;

begin ------------------------------------------------------------
    if not eof then
    begin
       read(chl);
       write(chl);                                       step0[C]
       ch := chl
    end;
       -----------------------------------------------------------
    while not eof do
    begin
       read(chl);
       if chl ≠ b or ch ≠ b then
```

```
    begin
       write(ch1);                                    limit(step1[C])
       ch := ch1
    end
  end
end.
```

10.2 A "system" problem

The behaviour of a disk handler is now specified as a system programming example. In order to prove that this system has some "good" properties, a first "theoretic" chapter introduces a simple model for a non-deterministic system. This model is a graph in which the nodes and edges, respectively, represent the states and possible transitions of a dynamic system. Predefined "initial" and "final" states indicate where the system should start and possibly stop. These components constitute a structure (a class) whose axioms state that a final state has no successors and an initial state either is a final state or has successors. Four special cases are then introduced, namely:

- loop_free_systems, whose graphs have no loop
- deadlock_free_system, where those nodes that are reachable from the initial nodes are final or have successors
- finite systems where the set of nodes that are reachable from an initial node is finite
- well_halting_systems that contain all the previous properties.

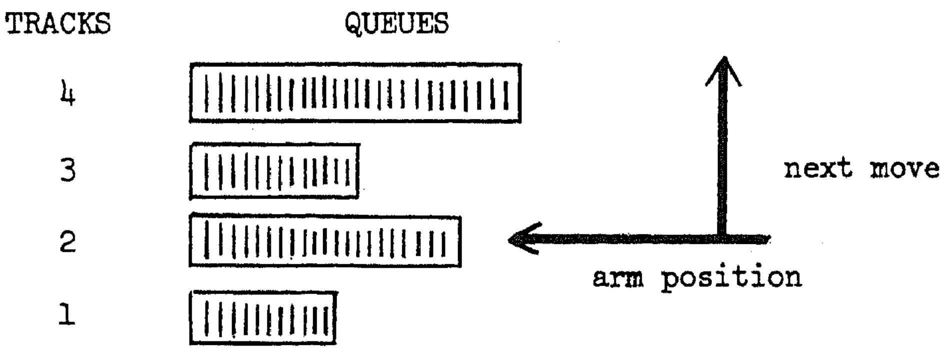
```
NON_DETERMINISTIC_SYSTEM =
   def
      system[X]        = class
                            reachable : X ↔ X;
                            initial, final : subset(X)
                         where
                            reachable ε trans[X];
                            initial ⊂ (final U inv(reachable)(X));
                            reachable(final) = null
                         end;
```

SPECIFICATION LANGUAGE

```
    loop_free_system[X]= subclass system[X] where
                            reachable ε irreflex[X]
                         end;
dead_lock_free_system[X]=subclass system[X] where
                            reachable(initial) ⊂ (final U inv(reach-
                                                        able)(X))
                         end;
    finite_system[X]    = subclass system[X] where
                            forall x for x : initial then
                               finite(reachable(x))
                            end
                         end;
    halting_system[X]   = loop_free_system[X] ∩
                         dead_lock_free_system[X] ∩
                         finite_system[X]
end NON_DETERMINISTIC_SYSTEM
```

An informal description of the disk handler is now given. A disk is made up of a finite number of concentric tracks. In order to optimise the arm movement, one organises the disk scheduling in such a way that the arm goes regularly from the exterior to the interior and back (this is the "lift" algorithm): the queries are therefore not served according to a FIFO strategy but rather by taking into account the current arm position and its next intended move. With each track is associated a queue of recognised queries that have not yet been served.

Example:



After serving the queries for track No 2, the arm moves to track No 3, serves its waiting queries, does the same for track No 4, then turns around to serve successively lower tracks, and so on.

The various tracks of the disk constitute, as stated above, a finite well order. It is then necessary, before entering the main definitions, to write yet another "chapter" introducing the concept and properties of well ordering. We first define the minimal elements of a set through an order relation; a well order relation is simply an order relation where the minimal element of all non null sets is unique: it is called the minimum of the given set through the relation. A finite_well_order is a well order relation whose domain is finite. The inverse of a finite_well_order is also a finite_well_order: this allows us to define the maximum of a set through a finite_well_order.

```
WELL_ORDER =
    def
       minimum[X]          = rel S,r ↔ x for
                                  S : subset(X) - {null};
                                  r : order[X];
                                  x : X
                              where
                                  inv(r)(x) ∩ S = {x}
                              end;

       well_order[X]       = set r for r : order[X] where
                                  minimum ε functional(Y,X)
                              given
                                  Y = (subset(X) - {null}) x order[X]
                              end;

       min[X]              = func S,r → x for
                                  S : subset(X) - {null};
                                  r : well_order[X];
                                  x : X
                              then
                                  x : function(minimum)(S,r)
                              end;

finite_well_order[X]       = set r for r : well_order[X] where
                                  finite(X)
                              end;
```

```
    th[X]          => forall r for r : finite_well_order[X] then
                         inv(r) ε finite_well_order[X]
                      end;

    max[X]          = func S,r → x for
                         S : subset(X) - {null};
                         r : finite_well_order[X];
                         x : X
                      then
                         x = min(S,inv(r))
                      end

end WELL_ORDER
```

The final "chapter", named LIFT_SYSTEM, contains the specification of the disk handler. It starts with the definition of the "hardware", a class defining the finite well ordering of the tracks and the content of the disk. This class is generic with respect to both TRACK and VALUE sets, the latter representing, without further details, the possible data stored on a disk track. The "hardware" class is then extended (subclass "static_state") by adding two new components, the first giving the maximum queue size (this is a "software" parameter), the second defining the initial input as a function from QUERY (another generic parameter) to TRACK. Finally, "static_state" is also extended (subclass "state"), thus defining the complete dynamic state of this system. This last subclass contains four new components, namely:

- input : a partial function from QUERY to TRACK representing the not yet entered queries (future queries)
- wait : a partial function from QUERY to TRACK representing the entered but not yet served queries (those that are waiting in the internal queues)
- output : a partial function from QUERY to VALUE representing the past queries (already served)
- current: giving the current track position of the arm.

These components, of course, obey some predicates in order to constitute an acceptable state; no query shall be simultaneously in the "input", "wait" or "output" domains; the number of queries waiting within the internal queues shall not exceed the maximum size of such queues; the disk value corresponding to each query shall not be changed throughout the dynamic evolution of the system (no updating). Four partial functions from "state" to "state" describe the transitions

- "ask" enters a query into the internal queues
- "serve" removes a query from the "current_queue" after serving it
- "change_move" changes the direction of the disk head movement
- "search" looks for the next track to become the current track.

The union of these partial functions defines a binary relation "next_state" between states. It is now possible to construct an instance of a "system" (the general model described in the chapter NON_DETERMINISTIC_SYSTEM) and to prove that the proposed "lift_system" is indeed a "halting_system".

```
LIFT_SYSTEM =
    use NON_DETERMINISTIC_SYSTEM, WELL_ORDER def
       hardware[TRACK,VALUE] = class
                                  track_order : finite_well_order
                                                           [TRACK];
                                  disk : TRACK → VALUE
                               end;

static_state[TRACK,VALUE,QUERY] = subclass hardware[TRACK,VALUE]
                                                           class
                                     max_queue_size : NAT;
                                     initial_input  : QUERY → TRACK
                                  where
                                     finite(QUERY)
                                  end;

    state[T,V,Q]        = subclass static_state[T,V,Q] class
                              input,wait : Q ⇸ T;
                              output     : Q ⇸ V;
                              current    : T
                          where
                              {dom(input); dom(wait); dom(output)} ε
                                                  partition[Q];
                              card(queue(T)) ≤ max_queue_size;
                              (disk • (input U wait)) U output = disk
                                                  • initial_input
```

```
                            given
                               queue           = inv(wait);
                               waiting_queries = queue(T);
                               candidate       = track_order(current)
                                                   ∩ wait(Q);
                               current_queue   = queue(current)
                            end;

-- now the state transition functions

    ask[T,V,Q]          = func s → s' for s,s' : state[T,V,Q] where
                            input(s) ≠ null;
                            card(waiting_queries(s)) < max_queue_
                                                    size(s)
                         then
                            s' = repl s with
                                    input = input - {q → input(q)};
                                    wait  = wait  U {q → input(q)}
                                 given
                                    q = any(dom(input))
                                 end
                         end;

    serve[T,V,Q]        = func s → s' for s,s' : state[T,V,Q] where
                            current_queue(s) ≠  null
                         then
                            s' = repl s with
                                    wait = wait - {q → wait(q)};
                                 given
                                    q = any(current_queue)
                                 end
                         end;

change_move[T,V,Q]      = func s → s' for s,s' : state[T,V,Q] where
                            waiting_queries(s) ≠ null;
                            candidate(s) = null;
                            current_queue(s) = null
```

```
                         then
                            s' = repl s with
                                   track_order = inv(track_order)
                                 end
                         end;

  search[T,V,Q]        = func s → s' for s,s' : state[T,V,Q] where
                            candidate(s) ≠ null;
                            current_queue(s) = null
                         then
                            s' = repl s with
                                   current = min(candidate,
                                                 track_order)
                                 end
                         end;

next_state[T,V,Q]      = rel s ↔ s' for s,s' : state[T,V,Q] where
                            s' = ask(s) or s' = serve(s) or
                            s' = change_move(s) or s' = search(s)
                         end;

-- now the final instantiation

initial_state[T,V,Q]   = subclass state[T,V,Q] where
                            input = initial_input
                         end;

final_state[T,V,Q]     = subclass state[T,V,Q] where
                            output = disk • initial_input
                         end;

   lift[T,V,Q]         = cons system[state[T,V,Q]] with
                            reachable = closure(next_state[T,V,Q])
                                              -ident[state[T,V,Q]];
                            initial = initial_state[T,V,Q];
                            final = final_state[T,V,Q]
                         end;

   th[T,V,Q]          => lift[T,V,Q] ε halting_system[state[T,V,Q]]
end LIFT_SYSTEM
```

10.3 Garbage collectors

A classical example is now proposed. It has already been described in several papers, particularly the one by Dijkstra et al. (*).

The informal description of this system will be given together with the formal text; however, a previous knowledge of the problem is probably necessary to comprehend fully the proposed development.

```
GARBAGE_COLLECTORS =

def

/* A first class, called "state0[N]", describes the basic data
structure of this system. It is generic with respect to N (for
Node) */

state0[N] = class
                next : N ↔ N;
                free,root : subset(N)
            given
                reachable = closure(next)(root)
            end;

/* An acceptable state is one where free nodes are not reachable
and have no successors */

state1[N] = subclass state0[N] where
                free ∩ reachable = null;
                next(free) = null
            end;

/* One now describes three functions, together called the "mutator".
They stand for the basic primitives at a user's disposal*/

/* The first primitive allows a user to extend the reachable nodes
by connecting an already reachable node with one that is free.
This node will, of course, lose this property */
```

(*) On_the_Fly Garbage Collection: An Exercise in Cooperation, E.W. Dijkstra et al., CACM, Vol 21, No 11, Nov. 1978.

```
extendl[N] = func n,s → s' for
                n : N; s,s' : statel[N]
             where
                n ε reachable(s); free(s) ≠ null
             then
                s' = repl s with
                        next = next U {n ↔ n'};
                        free = free - {n'}
                     given
                        n' = any(free)
                     end
             end;

/* The second primitive allows a user to connect two already reach-
able nodes.  This primitive requires that the set of free  nodes be
not empty, although this is not strictly necessary */

insertl[N] = func n,n',s → s' for
                n,n' : N; s,s' : statel[N]
             where
                n ε reachable(s);
                n' ε reachable(s);
                free(s) ≠ null
             then
                s' = repl s with
                        next = next U {n ↔ n'}
                     end
             end;

/* The third primitive disconnects two reachable nodes (if they were
already connected).  A non empty free node set is also required */

removel[N] = func n,n',s → s' for
                n,n' : N; s,s' : statel[N]
             where
                n ε reachable(s);
                n' ε reachable(s);
                free(s) ≠ null
```

```
             then
                s' = repl s with
                        next = next - {n ↔ n'}
                     end
             end;

/* Whenever the free set is empty any previous "mutator" activity
ceases and another function, called the "collector", appends the
non-reachable nodes (called "garbage") to the free set */

collector1[N] = func s → s' for
                   s,s' : state1[N]
                where
                   free(s) = null
                then
                   s' = repl s with
                           free = free U garbage;
                           next = next - (garbage x next(garbage))
                        given
                           garbage = node - reachable
                        end
                end;

/* Note that the "mutator" and "collector" activities exclude each
other.  Note also that

    next = next - (garbage x next(garbage))

ensures that the invariant of "state1[N]"

    next(free) = null

always holds */

/* The "collector" activity will now be decomposed into two phases

- a marking phase where reachable nodes are marked

- an appending phase where non marked nodes are appended to the
  free nodes.

In order to do this one extends "state1[N]" to introduce marked
nodes */
```

```
state2[N] = subclass state1(N) class
                marked : subset(N)
            where
                marked ⊂ reachable;
                root   ⊂ marked
            end;

/* The "mutator" primitives do not change.  The first "collector"
primitive marks the nodes */

mark2[N] = func s → s' for s,s' : state2[N] where
               next(s)(marked(s)) ⊄ marked(s);
               free(s) = null
           then
               s' = repl s with
                        marked = marked U next(marked)
                    end
           end;

/* The second "collector" primitive appends the non marked nodes to
the free set */

append2[N] = func s → s' for s,s' : state2[N] where
                 next(s)(marked(s)) ⊂ marked(s);
                 free(s) = null
             then
                 s' = repl s with
                          free = free U non_marked;
                          next = next - (non_marked x next(non_marked));
                          marked = root
                      given
                          non_marked = node - marked
                      end
             end;

/* Note that both "collector" primitives  exclude each other (and
still exclude the "mutator" activities), and that  the marking phase
is usually performed by several invocations of the "mark" function.
It is important to prove that this new "collector" does the same
```

thing as the previous one. In other words, we have to prove the following theorem */

```
th2[N] => forall s for s : dom(append2[N]) then
              marked(s) = reachable(s)
          end;
```

/* We have to prove

$$\text{marked} = \text{closure(next)(root)}$$

under the following hypothesis

$$\text{H1} : \text{root} \subset \text{marked} \subset \text{closure(next)(root)}$$

coming from the definition of "state2[N]"

$$\text{H2} : \text{next(marked)} \subset \text{marked}$$

coming from the definition of "dom(append2[N])".

It is therefore sufficient to prove

$$\text{closure(next)(root)} \subset \text{marked}$$

This is done by induction.

$$\begin{array}{lll} \text{step 0 :} & \text{root} \subset \text{marked} & \text{(from H1)} \\ \text{step n :} & \text{next}^{n}\text{(root)} \subset \text{marked} & \text{(induction Hyp)} \\ & \text{next}^{n+1}\text{(root)} \subset \text{next(marked)} & \\ & \text{next}^{n+1}\text{(root)} \subset \text{marked} & \text{(from H2)} \end{array}$$

Q.E.D. */

/* One now removes the constraint that "mutator" and "collector" activities exclude each other. In other words, we allow the "mutator" activities to be possibly performed between two invocations of the "mark" function. In order to do this, "state1[N]" is extended by another component, a set of "pre_marked" nodes, and extra axioms */

```
state3[N] = subclass state1[N] class
                marked,pre_marked : subset(N)
            where
                marked ∩ pre_marked = null;
                next(marked) ∩ non_marked = null;
                root ∩ non_marked = null
```

```
          given
             non_marked = node - (marked U pre_marked)
          end;

/* The "mutator" functions are, of course, different.  In particular,
the "insert" and "remove" functions do no longer require that the
free set be non_empty */

extend3[N] = func n,s → s' for
                n : N; x,x' : state3[N]
             where
                n ε reachable(s); free(s) ≠ null
             then
                repl s with
                   next = next U {n ↔ n'};
                   free = free - {n'};
                   pre_marked = pre_marked U ({n'} ∩ non_marked)
                given
                   n' = any(free)
                end
             end;

/* Note that

    pre_marked = pre_marked U ({n'} ∩ non_marked)

ensures the conservation  of

    next(marked) ∩ non_marked = null

which is an axiom of "state3[N]" whose importance will be clear
later.  (See the proof of th3[N]) */

insert3[N] = func n,n',s → s' for
                n,n' : N; s,s' : state3[N]
             where
                n ε reachable(s); n' ε reachable(s)
             then
                s' = repl s with
                        next = next U {n ↔ n'};
                        pre_marked = pre_marked U ({n'}∩non_marked)
                     end
             end;
```

```
/* The last "mutator" primitive "remove3[N]" is the same as
"remove1[N]".

Next are the "collector" primitives */

mark3[N] = func s → s' for s,s' : state3[N] where
              pre_marked ≠ null
           then
              s' = repl s with
                    marked = marked U pre_marked;
                    pre_marked = next(pre_marked) ∩ non_marked
                 end
           end;

append3[N] = func s → s' for s,s' : state3[N] where
                pre_marked = null
             then
                s' = repl s with
                      free = free U non_marked;
                      next = next - (non_marked x next(non_marked));
                      marked = null;
                      pre_marked = root
                    end
             end;

/* Note first that the "collector" activities still exclude each
other.  It is now necessary to prove that this third "collector"
does the same thing as the previous one.  This is  not actually
true: this new "collector" only collects part of the garbage as
stated by the following theorem */

th3[N] => forall s for s : dom(append3[N]) then
             reachable(s) ⊂ marked(s)
          end;

/* By comparison with "th2[N]" above, one may figure out that when
"append3[N]" is invoked, there exist some "marked" nodes that are
no longer reachable.  We have to prove that

    closure(next)(root) ⊂ marked

under the following hypothesis
```

H1 : root ⊂ (marked U pre_marked)

H2 : next(marked) ⊂ (marked U pre_marked)

both coming from the definition of "state3[N]"

H3 : pre_marked = null

coming from the definition of "dom(append3[N])".

Proof:

(1) root ⊂ marked (by H1 and H3)

(2) next(marked) ⊂ marked (by H2 and H3)

(3) closure(next)(marked) = marked (by (2))

(4) closure(next)(root) ⊂ closure(next)(marked) (by (1))

(5) closure(next)(root) ⊂ marked (by (3) and (4))

Q.E.D.

Unfortunately, this theorem does not prove that this actual "collector" indeed collects anything. In other words, the set of "non_marked" nodes might very well be empty when "append3[N]" is invoked. Let "old_garbage" be the set of nodes that are reachable but still marked when "append3[N]" is invoked. One now proves that this "old_garbage" will indeed be appended to the free set upon the next invocation of "append 3[N]". To do this, the following extension of "state3[N]" is performed */

```
state4[N] = subclass state3[N] class
                old_garbage : subset(N)
            where
                old_garbage ∩ free = null;
                old_garbage ∩ reachable = null;
                old_garbage ⊂ non_marked
            end
```

/* The function "append3[N]" is accordingly changed into */

```
append 4[N] = func s → s' for s,s' : state4[N] where
                  pre_marked = null
              then
                  s' = repl s with
```

```
                              free = free U non_marked;
                              next = next - (non_marked x next(non_marked))
                              marked = null;
                              pre_marked = root;
                              old_garbage = marked - reachable
                         end
              end;
```

/* As "old_garbage" is neither in the free set nor reachable, it so remains through the "mutator" activities, and neither does the interfering marking phase "paint" it. Therefore, the following theorem holds */

```
th4[N] => forall s for s : dom(append4[N]) then
              old_garbage(s) ⊂ free(s')
          given
              s' = append4(s)
          end;
```

/* Note that "old_garbage" is an "auxiliary variable" that has nothing to do with the system itself: it is only defined for the purpose of proving "th4[N]" */

/* One now proceeds by decomposing the "mutator" activities one step further, thereby allowing more interferences to occur with the marking phase. Remember that the following was performed by "extend3[N]" and "insert3[N]"

```
    pre_marked = pre_marked U ({n'} ∩ non_marked)
```

By doing this, we possibly "shade" n'. This shading might be performed in a non-exclusive way. To do this, a new component called "param" is added, the purpose of which is to "store" the value of n' while other activities occur before its shading */

/* In order to prove that th3[N] still holds, we introduce yet another auxiliary variable (*) named "old_next" that retain the value of "next" just before the possible invocation of the shading primitive */

```
state5[N] = subclass state1[N] class
                 marked, pre_marked,param : subset(N);
                 old_next : N ↔ N
            where
```

```
            marked ∩ pre_marked = null;
            next(marked) ⊂ (marked U pre_marked U param);
            root ⊂ (marked U pre_marked);
            free ⊂ marked;
            param ⊂ old_reachable;
            old_next(marked) ⊂ (marked U pre_marked)
        given
            non_marked = node - (marked U pre_marked);
            old_reachable = closure(old_next)(root)
        end;
```

/* It is interesting to note the difference from the axioms of "state3[N]":

- "next(marked)" is no longer always "non_marked" as "param" may be "non_marked".
- We require that the free set be "marked".

The new "mutator" functions are the following */

```
extend5[N] = func n,s → s' for
                 n : N ; s,s' : state5[N]
             where
                 n ε reachable(s);
                 free(s) ≠ null;
                 param(s) = null
             then
                 s' = repl s with
                          next = next U {n ↔ n'};
                          free = free - {n'};
                          old_next = next U {n ↔ n'}
                      given
                          n' = any(free)
                      end
             end;
```

(*) Auxiliary variable technique was first introduced by S. Owicki in her thesis: Axiomatic Proof Technique For Parallel Programs, Dept. C.S., Cornell University, TR.251 (1975).

```
/* Note that n' need not be shaded as it is already "marked" because
it belongs to "free" */

insert5[N] = func n,n',s → s' for
                n,n' : N; s,s' : state5[N]
            where
                n ε reachable(s);
                n' ε reachable(s);
                param(s) = null
            then
                s' = repl s with
                        next = next U {n ↔ n'};
                        param = {n'};
                        old_next = next
                     end
            end;

/* Note that after the invocation of insert5[N] the following still
holds

    param ⊂ old_reachable
    old_next(marked) ⊂ (marked U pre_marked)

because n' was an element of "reachable" and "param" was empty
before the invocation */

shade5[N] = func s → s' for
                s,s' : state5[N]
            where
                param(s) ≠ null
            then
                s' = repl s with
                        pre_marked = pre_marked U (param ∩
                                                    non_marked);
                        param = null;
                        old_next = next
                     end
            end;
```

```
/* Note that  after the invocation of "shade5[N]", the invariant

    next(marked) ⊂ (marked U pre_marked U param)

still holds, since "pre_marked" was possibly extended if "param"
was "non_marked".  Note that this would not have been the case if
"shade5[N]" had been performed before "insert5[N]" */

remove5[N] = func n,n',s → s' for
                 n,n' : N;
                 s,s' : state[N]
             where
                 n ε reachable(s);
                 n' ε reachable(s);
                 param(s) = null
             then
                 s' = repl s with
                         next = next - {n ↔ n'};
                         old_next =  next - {n ↔ n'};
                      end
             end;

/* Now the "collector" */

mark5[N]   = func s → s' for
                 s,s' : state5[N]
             where
                 pre_marked ≠ null
             then
                 s' = repl s with
                         marked = marked U pre_marked;
                         pre_marked = next(marked) ∩ non_marked
                      end
             end;

append5[N] = func s → s' for
                 s,s' : state5[N]
             where
                 pre_marked = null
```

```
            then
               s' = repl s with
                        free = free U non_marked;
                        next = next - (non_marked x next(non_marked));
                        marked = non_marked;
                        pre_marked = root;
                        old_next = old_next - (non_marked x next
                                                    (non_marked))
                    end
            end;
```

/* Of course, it is now important to prove that th3[N] still holds, namely:

```
    forall s for s : dom(append5[N]) then
       reachable(s) ⊂ marked(s)
    end
```

The only hypothesis of th3[N] that changes is H2, that was

```
    H2 : next(marked) ⊂ (marked U pre_marked)
```

which now becomes

```
    H2 : next(marked) ⊂ (marked U pre_marked U  param)
```

One proves that H2 indeed holds because of the following theorem */

```
th5[N] => forall s for s : dom(append5[N]) then
             param(s) ⊂ marked(s)
          end
```

/* One has to prove

```
    param ⊂ marked
```

under the following hypotheses

```
    H1 : param ⊂ closure(old_next)(root)
    H2 : root ⊂ (marked U  pre_marked)
    H3 : old_next(marked) ⊂ (marked U pre_marked)
```

all three coming from the definition of "state5[N]"

H4 : pre_marked = null

coming from the definition of "dom(append5)[N]"

Proof:

(1) root ⊂ marked (by H2 and H4)

(2) old_next(marked ⊂ marked (by H3 and H4)

(3) closure(old_next)(marked) = marked (by (2))

(4) closure(old_next)(root) ⊂ closure(old_next)(marked) (by (1))

(5) closure(old_next)(root) ⊂ marked (by (3) and (4))

(6) param ⊂ marked (by H1 and (5))

Q.E.D. */

end GARBAGE_COLLECTORS

The reader is invited to pursue further decompositions of "extend", "mark", and "append". Note that the previous formalisation does not contain any proof that the marking phase ever terminates.

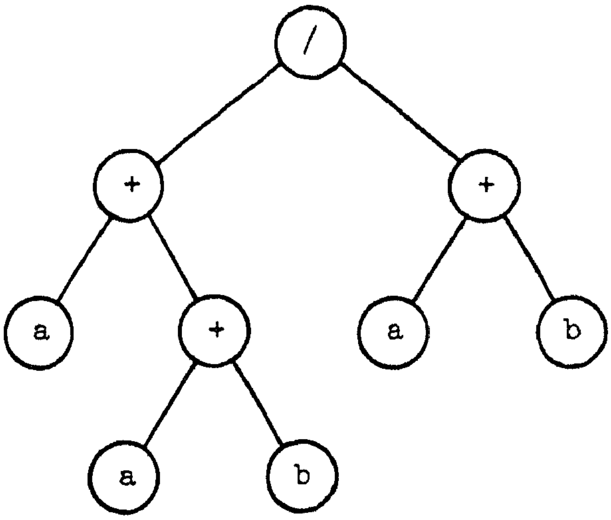
10.4 Algebraic language

Our last example is an attempt to specify a formal language (!) by defining its abstract syntax and semantics. We have chosen algebraic languages because they are simple enough and also because they are part of any programming language containing (boolean, arithmetic and so forth) "expressions". The specification is given at a general enough level so that any instantiation might be performed for a particular case of algebraic language: in this example, a boolean algebra.

An algebraic expression, as is well known, may be represented by a tree structure. For example, the following expression

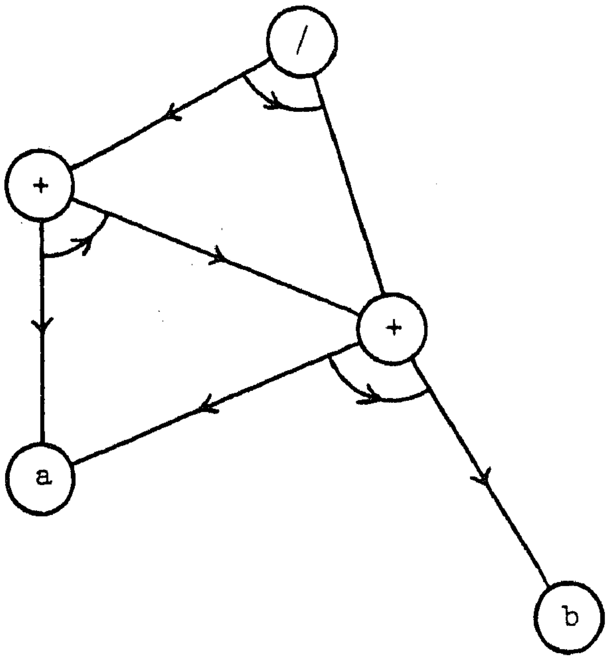
(a + (a+b)) / (a + b)

is pictured by



In this drawing, each node represents a sub-expression made of an operator and a set of ordered edges leading to other sub-expressions. The number of out-going edges depends upon a property of the corresponding operator called its arity, i.e., "/" and "+" have an arity of two, whereas "a" and "b" are considered as operators of 0 arity.

Algebraic expressions may contain several common subexpressions: therefore a new structure, here called a double order, is best suited to represent them. The above expression, for example, might be represented by



Such a structure is a "double" order because (i) it is a strict order relation, (ii) the out-going edges are also ordered. Before entering into the main definitions, a small "chapter" describes double orders.

```
DOUBLE_ORDER =
    def
        double_order[X] = set f for
                              f : X → seq[X]
                          where
                              (closure(r) - ident[X]) ε strict-order[X]
                          given
                              r = rel x ↔ x' for
                                      x,x' : X
                                  where
                                      x' ε f(x)(NAT)
                                  end
                          end;

/* One now defines the evaluation of a function through a double
order */

eval_rel[X,Y] = rel g,f ↔ h for
                    g : X x seq[Y] → Y;
                    f : X → seq[X];
                    h : X → Y
                where
                    f ε double_order[X];
                    forall x for x : X then
                       h(x) = g(x,h ∘ f(x))
                    end
                end;

th[X,Y] => eval_rel[X,Y] ε functional(((X x seq[Y] → Y)
                                          x (X → seq[X]),X → Y))

eval[X,Y] = function(eval_rel[X,Y])

end DOUBLE_ORDER
```

The chapter "ALGEBRAIC_LANGUAGE" first defines a "program" as a class whose components are the operator and arguments of each sub-expression. Another class defines an "algebra" as a class defining the application of an operator to a sequence of data as well as the arity of each operator. The semantics of a program with respect to an algebra defines the value of a sub-expression as the evaluation of its arguments through the double order represented by the program. An instantiation of these classes finally defines a boolean expression "program" interpreted by a boolean algebra.

```
ALGEBRAIC_LANGUAGE =
    use DOUBLE_ORDER def
       program[EXP,OP] = class
                            operator : EXP → OP;
                            argument : EXP → seq[EXP]
                         where
                            argument ε double_order[EXP]
                         end;

       algebra[DATA,OP] = class
                             value : OP x seq[DATA] ↛ DATA;
                             arity : OP → NAT
                          where
                             dom(value) = set op,s for
                                             op : OP;
                                             s  : seq[DATA]
                                          where
                                             arity(op) = length(s)
                                          end
                          end;

       semantics[EXP,OP,DATA] = func p,a,e → v for
                                   p : program[EXP,OP];
                                   a : algebra[DATA,OP];
                                   e : EXP;
                                   v : DATA
                                where       -- static check
                                   arity(a) • operator(p) =
                                      length[EXP] • argument(p)
                                then
                                   v = eval(value(a),argument(p))(e)
                                end;

-- now an example
```

```
BOOL_OP = {∧; ∨;¬; true; false};
BOOL    = {true; false};

boolean_algebra =
    cons algebra(BOOL,BOOL_OP) with
       value = {∧, <true; true>   → true;
                ∧, <true; false>  → false;
                ∧, <false; true>  → false;
                ∧, <false; false> → false;
                ∨, <true; true>   → true;
                ∨, <true; false>  → true;
                ∨, <false; true>  → true;
                ∨, <false; false> → false;
                ¬, <true> → false;
                ¬, <false> → true;
                true, null → true;
                false, null → false};

       arity = {∨     → 2;
                ∧     → 2;
                ¬     → 1;
                true  → 0;
                false → 0 }

    end;

    EXP     = {e1; e2; e3; e4; e5; e6};
    example = cons program[EXP,BOOL_OP] with
                 operator = {e1 →¬;
                             e2 → ∨;
                             e3 → ∨;
                             e4 → ∧;
                             e5 → true;
                             e6 → false};

                 argument = {e1 → <e2>;
                             e2 → <e3;e4>;
                             e3 → <e5;e4>;
```

```
                                   e4 → <e5;e6>;
                                   e5 → null;
                                   e6 → null}
                        end;

   th => semantics(example,boolean_algebra,el) = false

end ALGEBRAIC_LANGUAGE
```

APPENDIX: SUMMARY OF THE LANGUAGE

Utilisation sublanguage

```
chapter ::= id = [use id_list] def body end id

id_list ::= id{,id}
```

Statement sublanguage

```
body    ::= clause{;clause}

clause ::= generic_name = set     |

           generic_name => bool   |

           generic_name = class

generic_name ::= id['['id_list']']
```

Set sublanguage

```
set     ::= set id_list for spec end |

            any id_list for spec end |

            subset(set) |

            '{'set{;set}'}' |

            null |

            set_id |

            object |

            set{,set} |

            (set)
```

```
spec    ::= decl[where cond[given def]]

decl    ::= id_list : set{;id_list : set}

cond    ::= bool{;bool}

def     ::= id_list = set{;id_list = set}

set_id ::= [id.]id['['set{,set}']']
```

Boolean sublanguage

```
bool    ::= not(bool) |

            bool or bool |

            set = set |

            set ε set |

            finite(set) |

            (bool)
```

Class sublanguage

```
class      ::= class[spec] end |

               subclass class_exp[class decl][where cond[given def]]
                                                   end

class_exp ::= class_id{x class_id} |

               class_id{'|' class_id}

class_id   ::= set_id

object     ::= cons object_id[with def[given def]] end |

               repl object_id with def[given def] end

object_id ::= set_id
```

Statement sublanguage extensions

```
clause ::= operator_generic_name = set

operator_generic_name ::= op(id)['['id_list']']
```

SPECIFICATION LANGUAGE

Boolean sublanguage extensions

```
bool    ::= bool => bool |

            bool and bool |

            bool <=> bool |

            set ≠ set |

            set ⊄ set |

            exist id_list for spec end |

            existl id_list for spec end |

            forall id_list for decl[where cond] then cond
                                             [given def] end |

            bool{; bool}
```

Set sublanguage extensions

```
set     ::= set{x set} |

            any(set) |

            set ↔ set |

            set → set |

            set ↛ set |

            dom(id) |

            codom(id) |

            rel id_list ↔ id_list for spec end |

            func id_list → id_list for decl[where cond]bind
                                             [given def] end |

            operator_id |

            id(set ↔ set) |

            id(set) |

            '{'set ↔ set{; set ↔ set}'}' |

            '{'set → set {; set → set}'}' |

            subst id with set → set{; set → set}[given def] end
```

```
bind   ::= then def |

           when bool then def{when bool then def}[else def]

operator_id ::= [id.] op(id)['['set{,set}']']
```

Class sublanguage extension

```
class  ::= '{'id_list'}'
```

ACKNOWLEDGEMENTS

We are particularly indebted to M. Demuynck, A. Guillon, P. Moulin and G. Terrine for their fruitful comments, and to B. Keller for her beautiful typing of the original version of this text.