ELAN

USER MANUAL

November 29, 2006
ELAN V3.7

Peter Borovanský
Horatiu Cirstea
Eric Deplagne
Hubert Dubois
Claude Kirchner
Hélène Kirchner
Pierre-Etienne Moreau
Quang-Huy Nguyen
Christophe Ringeissen
Marian Vittek

ELAN: USER MANUAL

Authors: Peter Borovanský, Horatiu Cirstea, Eric Deplagne, Hubert Dubois, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, Quang-Huy Nguyen, Christophe Ringeissen, Marian Vittek

LORIA: CNRS, INPL, INRIA, UHP, Université de Nancy 2
Campus scientifique
615, rue du Jardin Botanique
BP101
54602 Villers-lès-Nancy CEDEX
FRANCE

---

elan n. **1**. Enthusiastic vigor and liveness. **2**. Style; flair. [Fr < OFr. *eslan*, rush < eslancer, to throw out: es-, out (< Lat. ex-) + *lancer*, to throw (< LLat. *lanceare*, to throw a lance < Lat. *lancea*, lance).]

The American Heritage Dictionary

---

1. ÉLAN. *n.m.* (vers 1420 selon Bloch ; de *élancer*.
‖ **1** Mouvement pour s'élancer. *Calculer son élan* (Cf. Calculer, cit. 7). ...
‖ **2** *Par anal.* Mouvement par lequel la voix reprend, s'élance. ...
‖ **3** *Fig.* Mouvement ardent, subit qu'un vif sentiment inspire. V. **Mouvement; ardent, impulsion, poussée.** *Brusque élan. Élan impétueux. Élans de jeunesse, de passion.* ...

2. ÉLAN. *n.m.* (XVII^{e} s.; *hellent* au XV^{e} s., *ellend* au XVI^{e}; haut allem. *elend*, du baltique *elnis*). Mammifère artiodactyle (*Cervidés*), des pays du Nord. *Élan du Canada.* V. **Orignal.**

LE ROBERT, extraits.

Contents

Foreword

This manual presents the version V3.7 of the ELAN language and of its environment.

This is an update of version V3.6 with the new builtin integers with overflow control. V3.6 was an update of version V3.4 with bug corrections with several new features offered by the compiler such that the Input/Output facilities, the new builtins (array, hashing tables) and the possibility to trace the normalisation process in ELAN programs. The ELAN compiler compiles now all possible patterns that contain AC symbols. No program transformation is required for the user. Furthermore, from V3.4 we distribute the sources of the system. The compilation/installation process follows the same procedure as other GNU softwares.

This is the user manual of ELAN version V3.7
We will be pleased to know any inaccuracy, error or typos.
Your comments on any part of the language, the system or this manual are welcome
on the elan-user mailing-list at `elan-user@loria.fr`.

Chapter 1

A short introduction to ELAN

This chapter presents in a top-down approach the ELAN main features. If you are beginning to use ELAN, you should certainly have a look at this chapter first. Complementarily Chapter 3 presents a bottom-up complete description of the language.

For a gently introduction to ELAN we recommend to read [BKK+98b] or to access to `http://elan.loria.fr`.

1.1 What could you do in ELAN?

Relying on the premiss that inferences rules can be quite conveniently described by rewrite rules, we started in the early nineties to design and implement a language in which inference systems can be represented in a natural way, and executed reasonably efficiently.

This leads us quickly to formalise such a language using the rewriting logic introduced by J. Meseguer [Mes92] and to see ELAN as a logical framework where the frame logic is rewriting logic extended with the fundamental notion of strategies [Vit94, KKV95, BKKM02]. A rewrite theory and an associated strategy is called a *computational system*.

In ELAN, a logic can be expressed by specifying its syntax, its inference rules and a description of how these rules should be applied. The syntax of the logic can be described using mixfix operators as in the OBJ language [GKK+87] or SDF [HHKR89]. The inference rules of the logic are described by conditional rewrite rules with `where` assignments allowing to introduce variables local to a rule.

From this description of the logic and a theory written in this logic, we infer automatically a computational system consisting of a rewriting theory plus strategies.

Since we wanted ELAN to be modular, with a syntactic level well-adapted to the user's needs, the language is designed in such a way that three programming levels are possible.

- First the design of a logic is done by the so-called *super-user*, with the help of a powerful preprocessor that expands specific macros into ELAN constructions.
- Such a logic can be used by the (standard) *user* in order to write a specification.
- Finally, the *end-user* can evaluate queries in the specification, following the semantics described by the logic.

This corresponds to the general diagram given in Figure 1.1. The query is interactively given by the end-user at the ELAN prompt level.

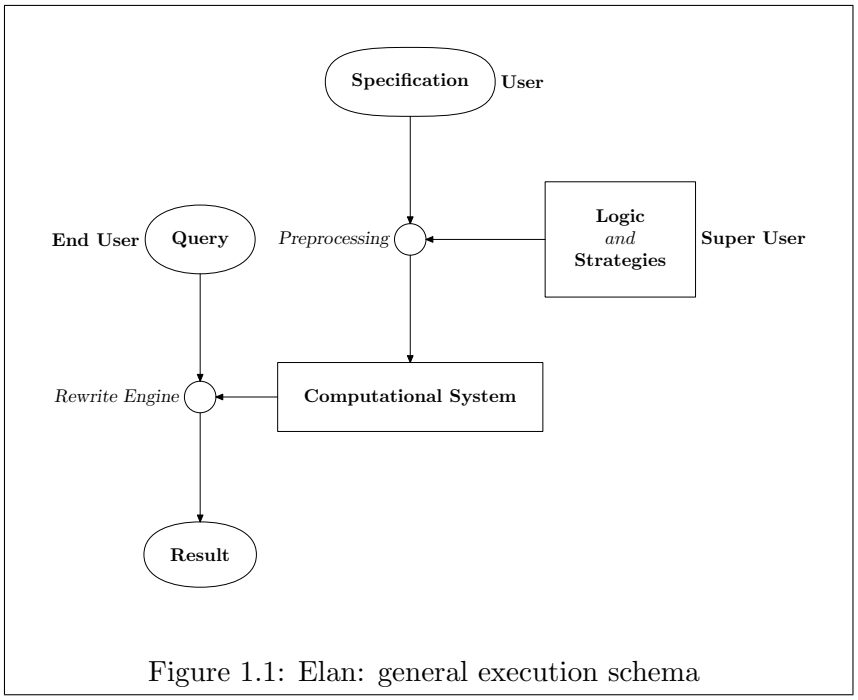

Figure 1.1: Elan: general execution schema

1.2 A very simple example

Let us fully detail a complete example. The next module illustrates what the programmer has to write in order to describe the derivative operation on simple polynomials. The first part contains:

- the module name: `poly1`
- modules you want to import: `int` (the integer library module)
- sort declaration: `variable` and `poly`
- operators definitions: constructors and functions declarations

The second part contains the computation definition. Given a query, ELAN repeatedly normalises the term using unlabelled rules. This is done in order to perform functional evaluation and thus it is recommended for the user to provide a confluent and terminating unlabelled rewrite system to ensure termination and unicity of the result. This normalisation process is built in the evaluation mechanism and consists in a leftmost innermost normalisation. This yields always a *single* result.

You can control the ELAN construction of terms by giving parsing annotations like associativity or priority to operators.

ElanExamples/poly1.eln

The top level of the logic description given in the following module describes the way to run the system. The logic declaration just introduces the sorts of interest (query and result) and defines `.eln` modules to be imported: this is the "top level".

ElanExamples/poly1.lgi

Once defined, we can use the two previous modules by calling the interpreter as detailed in section 2.1.1.

```
% elan poly1.lgi
enter query term finished by the key word 'end':
  deriv(X) end

[] start with term: deriv(X)
[] result term:       1

enter query term finished by the key word 'end':
  deriv(3*X*X + 2*X + 7) end

[] start with term: deriv(3*X*X+2*X+7)
[] result term:       0*X*X+3*1*X+X*1+0*X+2*1+0
```

1.3 A more generic example

The next example, introduces more ELAN features. It consists of the specification of elementary polynomials built over a finite set of variables, and the derivative functions to compute the derivated form with respect to a given variable. Tasks are divided as follows:

1. the super-user describes in a generic way the derivative and factorisation inferences, i.e. a logic for polynomial differentiation,
2. the user gives the specification of an algebra in which (s)he wants to derivate polynomials; in this case, this is quite simple since it amounts to specify the variables of the considered polynomial algebra,
3. the end-user gives a differentiation problem.

The diagram in Figure 1.1 thus instantiates as described in Figure 1.2.

The description of the logic and of the specification is done in the ELAN syntax described in the Chapter 3 and it can be further extended by the super-user. These descriptions are assumed to be done in files with specific extensions:

1. `.lgi` for the top level logic description, this file is written by the ELAN super-user,
2. `.eln` for a module used in a logic description, this file is also written by the ELAN super-user,
3. `.spc` for a specification, i.e. a program written in the defined logic by an ELAN user.

The `.eln` module of our example is the following:

ElanExamples/poly2.eln

Then, the logic declaration describes the way to parse the specification file that contains a finite list of variables.

ElanExamples/poly2.lgi

To instantiate the *Logic* and build the *Computational System*, you have to give a specification. An example of such specification is:

ElanExamples/someVariables.spc

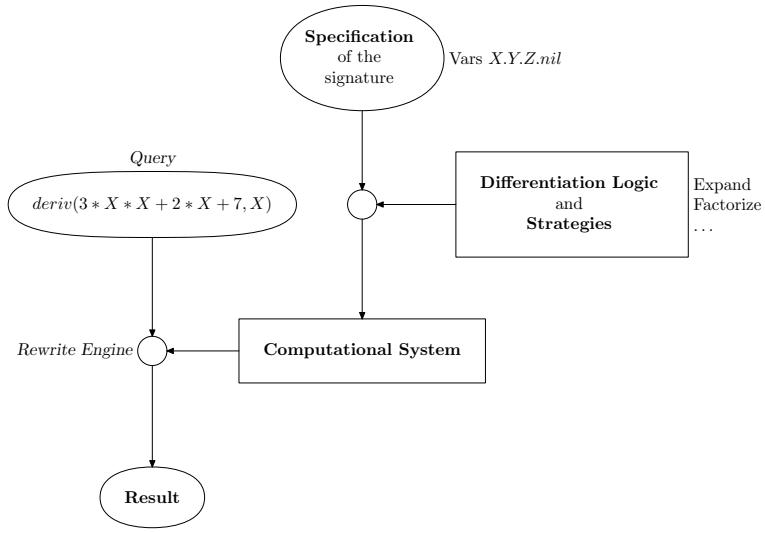

Figure 1.2: Elan: how to implement polynomial differentiation

It contains a list of variables that is read and transformed into a rewrite rule: `Vars => X.Y.Z.nil`

The `FOR EACH` preprocessor construction used in the `poly2.eln` module performs a textual replacement that extracts identifiers `X`, `Y`, `Z` from the list `X.Y.Z.nil` and declare them of sort `variable`.

You can notice that operators `*` and `+` are now declared as associative and commutative, which is helpful to implement some simplification rules. One rule contains a conditional part.

Now let us call ELAN with the specification `someVariables.spc`:

```
% elan poly2 someVariables.spc
enter query term finished by the key word 'end':
  deriv(3*X*X + 2*X + 7 , X) end

[] start with term: deriv(3*X*X+2*X+1,X)
[] result term:       X+X*3+2
```

The result has been simplified in a better way, but it is still difficult to read due to the lack of parenthesis.

1.4 An extended example

This last example shows how to play with labelled rules and strategies.

Unlabelled rules are applied repeatedly, at any position, to normalize the term. A labelled rule is applied only when a strategy calls it. In this case, the rule is applied at top position on the term. In the next module, you can recognize labelled rules: names are given between brackets.

You can define strategies in the same way you define rules and use them in local affectation parts introduced by the key word `where`. When a rule is applied, before constructing the right-hand-side, conditional parts are evaluated and variables involved in local affectation parts are

instantiated by the result of the application of a strategy on a term.

ElanExamples/poly3.eln

And now we can use it:

```
% elan poly3 someVariables.spc
enter query term finished by the key word 'end':
  deriv(3*X*X + 2*X + 7 , X) end

[] start with term: deriv(3*X*X+2*X+1,X)
[] result term:       ((X*6)+2)
[] end
```

Chapter 2

How to use ELAN

2.1 Using ELAN

The language can be used either through an interpreter or a compiler as described below in section 2.1.1 and 2.1.3 respectively.

2.1.1 How to run the ELAN interpreter

You can now execute ELAN by just typing the `elan` command with the following usage:

```
ELAN version 3.7a

Copyright (C) 1994-2006  LORIA (CNRS, INPL, INRIA, UHP, U-Nancy 2)
                         Nancy, France.

Usage:
        elan [options] lgi_file [spc_file]
Options:
        --dump, -d              : dump of signature, strategies and rules
        --trace, -t [num]       : tracing of execution (default max)
        --statistic, -s/-S      : statistics: short/long
        --warningsoff, -w       : suppress warning messages of the parser
        --quiet, -q             : quiet regime of execution
        --batch, -b             : batch regime (no messages at all)
        --elanlib               : elanlib
        --secondlib, -l lib     : second elan library (by default ..)
        --command, -C           : command language
        --export, --cexport     : export to a .ref file
        --import                : import from a .ref file
```

Both the logic description file (`lgi_file`) and the specification file (`spc_file`) may be written without their suffixes. Default suffixes for ELAN files are:

`.lgi` – for the top level logic description file,

`.eln` – for modules used in the top level logic description file,

`.spc` – for the specification file,

`.ref` – for the file in REF format.

The system ELAN searches these files (i.e. `.eln`, `.lgi`, `.spc`, `.ref`) in the following directories with the decreasing priorities:

- in the current directory,
- in the directory described by the value of a system variable `SECONDELANLIB`,
- in the directories described as `ELANLIB/commons`, `ELANLIB/noquote`, `ELANLIB/strategy` and `ELANLIB/ref`, in this order.

The default value of the variable `ELANLIB` is set to the subdirectory `elan_library/elanlib` of the source directory but it can be locally replaced by the switch `--elanlib`. The default value of the variable `SECONDELANLIB` is set to the parent directory, however, it can be also locally replaced by the switch `--secondlib` (or, `-l`) (for details, see below).

In Chapter 1, we have shown, how to simply run the ELAN interpreter with a logic description. We now illustrate on several examples the usage of the different switches.

`-d`, (or, `--dump`) dumps out the signature of the described logic (i.e. all sorts and function symbols with their profiles), definitions of all strategies and rewrite rules. For the example `poly3` from Chapter 1:

```
> elan -d poly3 someVariables
```

we obtain a list of rewrite rules of the form:

```
local rule expand:poly/poly3[Vars] [1/3/fsym=233/vars=9]
 ( VAR(0)+(( VAR(1)+ VAR(2))*( VAR(3)+ VAR(4))))         =>
                     ( VAR(0)+( VAR(5)+( VAR(6)+( VAR(7)+ VAR(8)))))
     where VAR(8) := (simplify:poly/poly3[Vars]) ( VAR(2)* VAR(4))
     where VAR(7) := (simplify:poly/poly3[Vars]) ( VAR(2)* VAR(3))
     where VAR(6) := (simplify:poly/poly3[Vars]) ( VAR(1)* VAR(4))
     where VAR(5) := (simplify:poly/poly3[Vars]) ( VAR(1)* VAR(3))

local rule expand:poly/poly3[Vars] [2/4/fsym=233/vars=6]
 ( VAR(0)+(( VAR(1)+ VAR(2))* VAR(3)))   => ( VAR(0)+( VAR(4)+ VAR(5)))
     where VAR(5) := (simplify:poly/poly3[Vars]) ( VAR(2)* VAR(3))
     where VAR(4) := (simplify:poly/poly3[Vars]) ( VAR(1)* VAR(3))
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . etc
```

Names of rewrite rules (e.g. `expand`) are displayed together with their sorts (e.g. `poly`), over which these rules work, and with a module name, where they have been defined (e.g. `poly3[Vars]`). An internal code of the top-most function symbol of its left-hand side and the number of variables of each rule are also displayed as an additional information. Variables appearing in rewrite rules are displayed in a uniform way: `VAR(i)`, where i is an internal index.

The print-out continues with a list of strategies in the form:

```
strategy global simplify:poly/poly3[Vars]
 repeat* (
   dc one(expand:poly )
 );
 repeat* (
   dc one(factorize:poly )
 )
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . etc
```

The label of the strategy is composed of the name of the strategy, its sort and the name of the module where it has been defined (as for labelled rules).

The specification of the signature of the described logic is composed of a set of defined (and internal) sorts:

```
Sorts
intern int, ident, identifier, string, list[identifier],
intern ident, int, variable, intern string, bool,
poly,
```

and of a list of function symbols with their profiles in the following form:

```
Function symbols
 @                    : (variable)poly        pri 0 code 231;
 @                    : (int)poly             pri 0 code 232;
 @ '+' @              : (poly poly)poly       assocRight      pri 1 code 233;
'(' @ '+' @ ')'       : (poly poly)poly       pri 0 code 233;
 @ '*' @              : (poly poly)poly       assocRight      pri 2 code 234;
'(' @ '*' @ ')'       : (poly poly)poly       pri 0 code 234;
'deriv' '(' @ ',' @ ')': (poly variable)poly   pri 0 code 235;
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . etc
```

-t (or, --trace) allows tracing ELAN programs. The maximal depth of tracing can be specified, like `-t 9`, however, by default (i.e. `-t`), it traces all levels. The number specified as the trace level counts matchings of a left-hand side, verifications of the conditional part of a rule, and evaluations of strategies applied over terms in local affectations. Thus, if the user wants to trace his/her rewrite derivation up to the 5-th level, the trace argument should be specified as `-t 15`.

```
 > elan -t poly3 someVariables
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . etc
enter query term finished by the key word 'end':
deriv(X*X,X) end

[] start with term :
    deriv((X*X),X)

       [reduce] start:
       [0]  deriv((X*X),X)
                  [reduce] start:
                  [0] (X*deriv(X,X))
                  [1] (X*1)
                  [2]  X
                  [reduce] stop :
              applying strategy 'simplify:poly/poly3[Vars]' on  X
                trying  'expand:poly' on        X
                fail of 'expand:poly'
                trying  'expand:poly' on        X
                fail of 'expand:poly'
                trying  'factorize:poly' on     X
                fail of 'factorize:poly'
                trying  'factorize:poly' on     X
                fail of 'factorize:poly'
                trying  'factorize:poly' on     X
                fail of 'factorize:poly'
            setting VAR(4) on  X
                  [reduce] start:
                  [0] (deriv(X,X)*X)
                  [1] (1*X)
                  [2]  X
                  [reduce] stop :
              applying strategy 'simplify:poly/poly3[Vars]' on  X
                trying  'expand:poly' on        X
                fail of 'expand:poly'
                trying  'expand:poly' on        X
```

```
                  fail of 'expand:poly'
                  trying  'factorize:poly' on     X
                  fail of 'factorize:poly'
                  trying  'factorize:poly' on     X
                  fail of 'factorize:poly'
                  trying  'factorize:poly' on     X
                  fail of 'factorize:poly'
               setting VAR(5) on  X
                     [reduce] start:
                     [0] (X+X)
                     [reduce] stop :
                 applying strategy 'simplify:poly/poly3[Vars]' on (X+X)
                  trying  'expand:poly' on       (X+X)
                  fail of 'expand:poly'
                  trying  'expand:poly' on       (X+X)
                  fail of 'expand:poly'
                  trying  'factorize:poly' on    (X+X)
                  fail of 'factorize:poly'
                  trying  'factorize:poly' on    (X+X)
                  fail of 'factorize:poly'
                  trying  'factorize:poly' on    (X+X)
                  fail of 'factorize:poly'
               setting VAR(3) on (X+X)
         [1] (X+X)
         [reduce] stop :
      trying  'expand:poly' on   (X+X)
      fail of 'expand:poly'
      trying  'expand:poly' on   (X+X)
      fail of 'expand:poly'
      trying  'factorize:poly' on        (X+X)
      fail of 'factorize:poly'
      trying  'factorize:poly' on        (X+X)
      fail of 'factorize:poly'
      trying  'factorize:poly' on        (X+X)
      fail of 'factorize:poly'

   [] result term:
      (X+X)

   [] end
```

-s (-S, or --statistics) displays a brief (-s) or a complete (-S) version of statistics. These statistics contain the information about the running time of the last query, average speed in rewrites per second, the statistics of labelled and unlabelled rules, which were tried and applied.

```
> elan -S poly3 someVariables
 . . . . . . . . . . . . . . . . . . . . . . . . . . . . etc
 enter query term finished by the key word 'end':
 deriv(X*X,X) end

[] start with term :
    deriv((X*X),X)

[] result term:
   (X+X)

[] end

 Statistics:
```

```
total time     (0.007+0.000)=0.007 sec (main+subprocesses)

average speed  714 inf/sec
               5 nonamed rules applied,    23 tried
               0   named rules applied,    20 tried
named rules
          applied   tried    rule for symbol
                0       8    expand:poly
                0      12    factorize:poly
nonamed rules
          applied   tried    rule for symbol
                0       4    ( poly + poly )
                2      12    ( poly * poly )
                3       7    deriv( poly , variable )
end of statistics
```

The complete statistics show a detailed list of applications and tries for labelled and unlabelled rules. The brief statistics do not precise the last paragraph about unlabelled rules.

`-w` **(or,** `--warningsoff`**)** eliminates all ambiguity warnings produced during parsing of the ELAN description of a logic.

`-q` **(or,** `--quiet`**)** suppresses all error messages and warnings during the execution of the ELAN program.

`-b` **(or,** `--batch`**)** suppresses all messages, thus this mode is useful when the ELAN system is executed from a script file in the batch mode.

`--elanlib dir` locally redefines the value of the system variable `ELANLIB` by the string `dir`.

`--secondlib dir` **(or,** `--secondlib dir`**)** does the same with the variable `SECONDELANLIB`.

`--export fname`, `--cexport fname` exports the input specification into the REF format stored in the file `fname`. The switch `cexport` is used only when the produced REF file is compiled by the ELAN compiler.

`--import fname` imports a specification in the REF format from the REF file `fname` produced, in general, by the switch `--export`.

```
> elan --export poly3.ref poly3 someVariables.spc

ELAN version 3.7a

Copyright (C) 1994-2006  LORIA (CNRS, INPL, INRIA, UHP, U-Nancy 2)
                         Nancy, France.
 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . etc

Export file poly3.ref has been created

> elan --import poly3.ref
ELAN version 3.7a

Copyright (C) 1994-2006  LORIA (CNRS, INPL, INRIA, UHP, U-Nancy 2)
                         Nancy, France.

Import form file poly3
```

```
Importing Identifiers
Importing Sorts
Importing Modules
Importing RuleNames
Importing StrategyNames
Import rule extractrule1:identifier/list[identifier]!LO form module list[identifier]
Import rule extractrule2:identifier/list[identifier]!LO form module list[identifier]
Import rule expand:poly/poly3[Vars]!LO form module poly3[Vars]
Import rule expand:poly/poly3[Vars]!LO form module poly3[Vars]
Import rule factorize:poly/poly3[Vars]!LO form module poly3[Vars]
Import rule factorize:poly/poly3[Vars]!LO form module poly3[Vars]
Import rule factorize:poly/poly3[Vars]!LO form module poly3[Vars]
Import strategy listExtract:identifier/list[identifier]
Import strategy simplify:poly/poly3[Vars]
Import strategy last_simplify:poly/poly3[Vars]

enter query term finished by the key word 'end':
```

`-C` runs the ELAN interpreter with a simple command language interpreted on the top-most level. This mode is described in Section 2.1.2.

After loading all modules specified in the logic description file, ELAN requires to enter a query term, which has to be finished by the word 'end' or by character `^D` (control-D). Its evaluation can be interrupted by typing `^C`, the interpreter then proposes a simple run-time menu:

- `ExecutionAbort` - A - aborts the current execution and allows to enter another query term,
- `Continue` - C - continues the interrupted execution,
- `Dump` - D - dumps the signature, rules and strategies of the currently loaded logic,
- `Exit` - E - quits the ELAN system,
- `Statistics` - S, s - writes the actual statistics corresponding to the last evaluated query,
- `ChangeTrace` - T - changes trace level of the current derivation such that the interpreter asks for a new trace level,
- `ChangeQuiet` - Q - switches between two displaying modes: quiet – unquiet,

2.1.2 Top level interpreter

The command language of the ELAN interpreter improves the user's interface by offering to the user several commands, which allow to:

- change several parameters of the system,
- debug programs using break-points,
- keep the history of queries and their results,
- runs the current logic with different strategies, different entries, etc.

When the ELAN interpreter is started with the option `-C`, instead of asking:
`enter query term finished by the key word 'end'`
ELAN prompts the user:
`enter command finished by ';'`
We show a brief description of all commands of the top level command language which will be later illustrated on several examples.

`quit` terminates the session and leaves the ELAN interpreter.

`load mod` loads (i.e. imports) an additional module, where the identifier `mod` describes its name (possibly with its arguments),

`batch mod` calls a script file with the name `mod` (an identifier), which contains a stream of commands of the command language. It tries to evaluate all commands of the script file, and it returns the control to the interactive mode. The script files can be also concatenated.

`qs` shows the queue of input queries (i.e. the history up to the last ten queries). The terms in this queue of queries `qs` can be referenced by the identifiers `Q`, `Q1`, ..., `Q9` respecting their sorts. These sorted identifiers could be used in constructions of further queries.

`rs` shows the queue of the latest results (i.e. the history up to the last ten results). Terms in this queue of results `rs` are referenced by identifiers `R`, `R1`, ..., `R9`. In the case of non-deterministic computations, there is no correspondence between query terms `Qi` and result terms `Ri`.

`startwith (str)term` redefines the 'start with' pattern originally defined in the logic description file `.lgi`. The pattern `term` may contain also symbols `query` standing as a place-holder of an input term entered by the user using the command `run`, or symbols `Q`, ..., `Qi` or `R`, ..., `Ri` standing for values of previously put queries or results.

`checkwith term` redefines the 'check with' pattern originally defined in the logic description file. The boolean pattern `term` may contain any symbol amongst `query`, `Q`, ..., `Qi` or `R`, ..., `Ri`.

`printwith term` sets-up the 'print-with' pattern. The 'print-with' pattern may contain the symbol `result`, which is a place-holder for result terms. Intuitively, the meaning of the 'print-with' term P is that any result r of the computation is substituted for the place-holder `result` in the pattern P. The substituted term $P[result/r]$ is normalized and printed out. This command can be used for the automatic transformation of results into different formats, e.g. a transformation of very large result terms into more readable or incomplete notation, or a conversion into a LaTeX notation. The default value of the 'print-with' pattern is `result`, i.e. the identity.

`run term` evaluates the term `term` as a query w.r.t. rewrite rules of the current logic and eventually returns results. The entry term `term` may contain identifiers `Q`, `Q1`, ..., `Q9`, (resp. `R`, `R1`, ..., `R9`) referring to values of preceding queries or results, e.g. items of the queue of queries `qs` (resp. results `rs`).

`sorts type type type` sets-up sorts of the symbols `query`, `result` and of the 'print-with' pattern.

`stat` prints the statistics,

`dump` prints rules, strategies and the signature,

`dump name` dumps only labelled rules with the name `name`, strategies labelled by `name` or unlabelled rules with the head symbol `name`.

`dump n` dumps only unlabelled rules of a function symbol with the internal code `n`.

`display n` if `n` is not zero, all printed terms are in the internal form. The internal form of a term attaches to a symbol name its internal code, which allows to identify overloaded function symbols. `display 0` switches all print-outs into traditional (thus, overloaded) format.

`trace n` sets-up the trace level to the value `n`.

`break name` (resp. `unbreak name`) sets (resp. removes) break-points to all labelled rules with the name `name`, to all strategies labelled by `name` and to all unlabelled rewrite rules with the head symbol `name`.

`break n` (resp. `unbreak n`) sets (resp. removes) break-points to unlabelled rules with the head symbol with the internal code `n`.

`breaks` shows a list of all breakpoints.

We illustrate the command language on a small ELAN session with the example `poly3` from the previous Chapter. When the switch `-C` is specified, the ELAN interpreter loads the module `Query` describing the syntax of commands.

```
> elan -q -C poly3 someVariables

ELAN version 3.7a

Copyright (C) 1994-2006  LORIA (CNRS, INPL, INRIA, UHP, U-Nancy 2)
                         Nancy, France.
   handling module identifier
     handling module ident
       handling module bool
       end of bool
     end of ident
     handling module eq[identifier]
       handling module cmp[identifier]
       end of cmp[identifier]
     end of eq[identifier]
   end of identifier
   handling module list[identifier]
     handling module int
       handling module builtinInt
       end of builtinInt
     end of int
   end of list[identifier]
   handling module poly3[Vars]
     handling module eq[variable]
       handling module cmp[variable]
       end of cmp[variable]
     end of eq[variable]
   end of poly3[Vars]
 handling module Query[poly,poly,poly]
 end of Query[poly,poly,poly]
```

We can run simple queries using the command `run`:

```
enter command finished by ';':
run deriv(X*X,X);
[] start with term :
    deriv((X*X),X)
[] result term:
   (X+X)
[] end

enter command finished by ';':
run deriv(X*X*X+X*X+X+1,X);
[] start with term :
    deriv(((X*(X*X))+((X*X)+(X+1))),X)
[] result term:
   (1+(X*(((X*2)+2)+X)))
[] end
```

The queries and their results are stored in two queues, which can be listed by commands `qs`, `rs`:

```
enter command finished by ';':
qs;
Queries ... 2 : elements
Q1      poly     deriv((X*X),X)
Q       poly     deriv(((X*(X*X))+((X*X)+(X+1))),X)

enter command finished by ';':
rs;
Results ... 2 : elements
R1      poly    (X+X)
R       poly    (1+(X*(((X*2)+2)+X)))
```

The previous queries and their results can be referenced in the construction of the new ones:

```
enter command finished by ';':
run deriv(R,X) ;
[] start with term :
    deriv((1+(X*(((X*2)+2)+X))),X)
[] result term:
   ((X*6)+2)
[] end

enter command finished by ';':
run deriv(Q*Q,X) ;
[] start with term :
    deriv((deriv((1+(X*(((X*2)+2)+X))),X)*deriv((1+(X*(((X*2)+2)+X))),X)),X)
[] result term:
   (24+(X*72))
[] end
```

Function symbols defined by unlabelled rules can be debugged, for example, if we know their internal code. In the following example, the command `dump` displays the symbol with the internal code `216`, and then, we can put a break-point on this function.

```
dump 235;
function dump
'deriv' '('  @ ','  @ ')'        : (poly variable)poly   pri 0 code 235;

enter command finished by ';':
break 235;
function dump
'deriv' '('  @ ','  @ ')'        : (poly variable)poly   pri 0 code 235;
```

Having break-points set, the execution shows all entry and exit points of traced functions, or strategies:

```
enter command finished by ';':
run deriv(X*X,X);

[] start with term :
    deriv((X*X),X)
 [0]  deriv((X*X),X)
       [0]  deriv(X,X)
       [1]  1
       [0]  deriv(X,X)
       [1]  1
 [1] (X+X)
[] result term:
   (X+X)
[] end
```

We can switch all outputs into the internal format by the command `display 1`:

```
enter command finished by ';':
display 1;

enter command finished by ';':
run deriv(X*X,X);
[] start with term :
    deriv_235((X_228*X_228),X_228)
 [0]  deriv_235((X_228*X_228),X_228)
       [0]  deriv_235(X_228,X_228)
       [1]  1
       [0]  deriv_235(X_228,X_228)
       [1]  1
 [1] (X_228+X_228)
[] result term:
   (X_228+X_228)
[] end
```

We can change the 'start-with' pattern to `(last_simplify)deriv(query,X)`, and the command `stat` informs us about the current setting and the performance of the last executed query.

```
enter command finished by ';':
startwith (last_simplify)deriv(query,X);

enter command finished by ';':
stat;
Input type : poly
Output type: poly
Print type : poly
Strategy   : last_simplify
Start_with :  deriv_235( VAR(0),X_228)
Check_with :  true_1
Print_with :  VAR(0)
------------

 Statistics:
 total time     (0.013+0.010)=0.023 sec (main+subprocesses)

 average speed  384 inf/sec
                5 nonamed rules applied,    23 tried
                0   named rules applied,    20 tried
 named rules
           applied   tried    rule for symbol
                 0       8    expand:poly
                 0      12    factorize:poly
 nonamed rules
```

```
          applied   tried    rule for symbol
                0       4    ( poly + poly )
                2      12    ( poly * poly )
                3       7    deriv( poly , variable )
 end of statistics
```

2.1.3 How to run the ELAN compiler

The ELAN compiler transforms a logic description into an executable binary file. The compiler (for detailed description, see [Vit96, MK98]) produces an efficient `C` code, which is later compiled by the `cc` or the `gnu gcc` compiler. The executable binary code is dependent on the particular architecture, because a small part of the ELAN library performing the basic non-deterministic operations is written in assembly language [Mor98]. Up to now, this is the only reason that makes the compiler available only on the architectures DEC-ALPHA, SUN4 and Intel-PC. This release of the ELAN compiler does not compile the whole ELAN language: the main restriction is that the ELAN compiler does not treat input/output and reflective capabilities of the ELAN language. There is also a restriction on the class of rules containing AC-symbols that can be compiled. Thus, any logic description containing AC-symbols should be carefully studied before being compiled (see Section 4.5 for more details).

To run the ELAN compiler, a Java Virtual Machine (JAVA 2 SDK, aka jdk 1.2, or newer), a C compiler (gcc recommended) and a C-shell (tcsh recommended) have to be installed on your system. To complete the installation, the script `PWD/bin/elanc` has to be modified: `CLASSPATH` and `JAVA` variables have to be set according to your system. Then, you can now execute the ELAN compiler by just typing the `elanc` command with the following usage:

```
elanc [[elan options][compiler options]] file[.lgi] [file.[spc]]
Options:
        -output <name>
        -verbose
        -nocode
        -nosplit
        -quiet
        -debug
        -noOptimiseChoicePoint [default]
        -optimiseChoicePoint
        -proofterm
```

By default, the compiler produces a lot of `C` files and a `lgiName.makefile` file, which are later compiled.

The switch `-output nqueens` produces the executable file with the name `nqueens`.

The switch `-nosplit` generates only two files: `lgiName.c` and `lgiName.core.c`. This switch is useful to compile small specifications.

The switch `-nocode` directly builds an executable file and deletes all generated `C` files.

The switch `-quiet` suppresses all printed messages during compilation phase.

The switch `-debug` activates the debugging mode: more messages are printed during compilation and the generated code becomes less efficient.

The switch `-noOptimiseChoicePoint [default]` avoids the deterministic analysis phase.

The switch `-optimiseChoicePoint` reduces the number of set choice-points: this switch reduces the runtime needed memory and improve the efficiency. It is recommended to not activate this switch if your specification contains AC symbols.

The switch `-proofterm` generates the complete trace of the normalisation process with respect to a set of labelled rules by the strategy **normin** described in Section 3.

We illustrate the compiler on a simple example of a distributed file. The first example shows the compilation of the program `nqueens` with the query `queens(8)` (given in the `.lgi` file):

```
> elanc -output queens -nosplit -nocode -optimiseChoicePoint nqueens
elan --cexport nqueens.ref ./nqueens.lgi

ELAN version 3.7a

Copyright (C) 1994-2006  LORIA (CNRS, INPL, INRIA, UHP, U-Nancy 2)
                         Nancy, France.
   handling module nqueens
     handling module int
       handling module builtinInt
         handling module bool
         end of bool
       end of builtinInt
     end of int
   end of nqueens

Export file nqueens.ref has been created

java -classpath classes.zip:elanlib/Compiler/classes REM nqueens.ref
-output queens -nosplit -nocode -optimiseChoicePoint
Reduce ELAN Code Machine. Reading from file nqueens.ref . . .
Parsing..........
Compiling nonamed rules.................
Compiling strategies..
Building queens...

> queens
result = .([](4),.([](2),.([](7),.([](3),.([](6),.([](8),.([](5),.([](1),nil))))))))
result = .([](5),.([](2),.([](4),.([](7),.([](3),.([](8),.([](6),.([](1),nil))))))))
result =
.([](3),.([](5),.([](2),.([](8),.([](6),.([](4),.([](7),.([](1),nil))))))))
...
after 91 solutions:
result = .([](5),.([](7),.([](2),.([](6),.([](3),.([](1),.([](4),.([](8),nil))))))))

rewrite_step = 227356
```

It is also possible to get the whole trace of the execution. In order to produce this information the executable code has to be compiled in the debugging mode and the switch `-trace` has to be used:

```
> elanc -debug -nosplit nqueens
 ...
Reduce ELAN Code Machine. Reading from file nqueens2.ref . . .
Parsing
Compiling nonamed rules.................
Compiling strategies..
execute the following line to build your program
        gmake -f nqueens.make
> gmake -f nqueens.make
gcc -pipe -g3 -DDEBUG -DPDEBUG -DBITSET32 ...
...
```

```
> a.out -trace
|start with: fun_221_queens()_([](8))
|rewrite[1] queens(,,)([](8),[](8),nil)
|start with: str_176(queens(,,)([](8),[](8),nil))
| start with: fun_211_greater_int(,)_([](8),[](0))
| rewrite[2] true
| start with: str_278([](8))
| rewrite(str_278)[3] [](1)
| start with: fun_222_noattack(,,)_([](1),[](1),nil)
| rewrite[4] true
| start with: fun_202_minus(,)_([](8),[](1))
| rewrite[5] [](7)
| start with: str_176(queens(,,)([](7),[](8),.([](1),nil)))
|  start with: fun_211_greater_int(,)_([](7),[](0))
|  rewrite[6] true
|  start with: str_278([](8))
|  rewrite(str_278)[7] [](1)
|  start with: fun_222_noattack(,,)_([](1),[](1),.([](1),nil))
|   start with: fun_210_neq_int(,)_([](1),[](1))
|   rewrite[8] false
|  rewrite[8] noattack(,,)([](1),[](1),.([](1),nil))
...
```

A more detailed description of the compiler can be found in Section 4.5.
▷ Another way to call the ELAN compiler (also known as *Reduced ELAN Machine*) consists in running Java as follows:
`java -classpath ${ELANLIB}/Compiler/classes REM` <*REF file name*> <*options*>
The <*REF file name*> must be created by the option `--cexport` <*REF file name*> of the ELAN interpreter simply used here to parse the <*LGI file name*> .

2.2 Keep in touch

In case you have any problems, questions or comments about ELAN, or just only to be keeping in touch with the ELAN team, please post on the elan-users mailing-list:

`elan-users@loria.fr`

More informations on ELAN can be found on the web page:

`http://elan.loria.fr`.

ELAN is issued from the PROTHEO research team, information about this research project can be found at the following address:

`http://www.loria.fr/equipes/protheo`.

Chapter 3

ELAN: the language

This chapter presents a full bottom-up description of the language features that, contrary to Chapter 1, uses a construction of the language only when it has been introduced before.

The ELAN syntax is given in the Extended Backus-Naur Form (EBNF). We assume that the reader is familiar with this well-known notation, where for instance { X } $^+$ (resp. { X } *) denotes a non-empty (resp. a possibly empty) repetition of X, and [X] means that X is optional, and may be omitted.

All examples given below are running under the current implementation of the ELAN version V3.7.

3.1 Lexicographical conventions

3.1.1 Separators

All ASCII characters whose code are less or equal than 32 are considered by the parser as *separators*. Thus spaces, tabulations and newlines are separators.

3.1.2 Lexical unities

A *lexical unity* in ELAN is either an identifier, a natural number or a special character.

Identifiers are composed by concatenation of letters, numerals and the character "_", and they should begin with a letter.

<*letter*> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
| a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

<*numeral*> ::= 0|1|2|3|4|5|6|7|8|9

<*identifier*> ::= <*letter*> { <*letter*> | <*numeral*> | _ } *

<*quoted identifier*> ::= ' { <*letter*> | <*qchar*> | <*numeral*> | _ } $^+$ '

<*elan identifier*> ::= <*identifier*> except ELAN's keywords

All characters different from the previously introduced ones (letters, numerals, separators) and the characters '{', '}', '~', are considered as special characters and are referenced as <*char*>.

<*nchar*> ::= <*char*> except '@' and ':'

<qchar> ::= *<char>* except ‘’’

A *number* is the concatenation of numerals.

<number> ::= { *<numeral>* } +

3.1.3 Comments

All strings enclosed between the strings "`/*`" and "`*/`" or between the string "`//`" and the end of line, are considered by the parser as *comments*.

3.2 Element modules

Element modules are the basic unit blocks of the language. They allow to define computational systems i.e. to define a rewrite theory together with a strategy. For modularity reasons, importations are made possible in ELAN modules, under the condition that no cycle exists in the importation relation. This is described in the modularity section 3.7.

The syntax is the following:

<module> ::= **module** *<formal module name>*
[*<imports>*]
[*<sort definition>*]
[*<operator definition>*]
[*<stratop definition>*]
{ *<family of rules>* } *
{ *<family of strategies>* } *
end

The module of name `ModuleName` should be declared in the file `ModuleName.eln`. Only one, and exactly one module can be described in a given file.

Importations can be made local or global:

<imports> ::= **import** { *<instantiated module name>* } + **; end**
| **import** [*<global imports>*] [*<local imports>*] **end**

<global imports> ::= **global** { *<instantiated module name>* } + **;**

<local imports> ::= **local** { *<instantiated module name>* } + **;**

An instantiated module name like `List[List[Int]]` is described by:

<instantiated module name> ::= *<identifier>*
| *<identifier>* [*<instantiated module name>*
{ **,** *<instantiated module name>* } *]

Sorts are always global:

<sort definition> ::= **sort** { *<sort name>* } + **; end**

but operators can be exported, and thus declared global, or just local in which case they can be used only in the module where they are defined.

<operator definition> ::= **operators**
[*<global operator definition>*] [*<local operator definition>*]
end

<global operator definition> ::= **global** { *<operator declaration>* } + **;**

<local operator definition> ::= **local** { *<operator declaration>* } + **;**

And similarly for strategy operator definition:

<stratop definition> ::= **stratop**
[*<global stratop definition>*] [*<local stratop definition>*]
end

<global stratop definition> ::= **global** { *<strategy declaration>* } + **;**

<local stratop definition> ::= **local** { *<strategy declaration>* } + **;**

<family of rules> is defined in Section 3.4. *<family of strategies>* is defined in Section 3.5.

3.3 Definition of signatures

ELAN is a multi-sorted language, and signatures are composed by the definitions of the sorts and operators symbols. The operator syntax is given in a mix-fix form which allows the super-user to define very conveniently its own syntax.

3.3.1 Sort declarations

A *sort declaration* is either a sort name (an identifier) or a sort name followed by the list of sort names it depends on:

<sort name> ::= *<identifier>*
| *<identifier>* [*<sort name>* { , *<sort name>* } *]

Example 3.1 bool, int, list[bool], pair[int,list[bool]], list[X] *are sort names in ELAN.*

Remark: A sort name is always *global* in ELAN. Two sort declarations using the same sort name, will result in the creation of only *one* sort.

Built-in sorts For efficiency, sorts could be defined as built-ins. See section 5.2 for details.

3.3.2 Operator declarations

A *function symbol*, also called *operator*, is given either by defining a new symbol together with its rank or by aliasing an existing operator. Optional properties can be specified:

<operator declaration> ::= *<new operator declaration>*
| *<operator alias>*

<new operator declaration> ::= *<operator name>* **:** *<rank>* [{ *<operator option>* } +]

<operator alias> ::= *<new operator declaration>* **alias** *<old operator name>* **:**

In case of *aliasing*, *<new operator declaration>* is only another name for the function symbol *<old name>* . The *two* names are then accessible and refer to the *same* object.
▷ The last alias defined is considered by the parser as having the highest priority with respect to the previously defined ones.

<old operator name> ::= *<operator name>*

<operator name> ::= *<name>*

The name of a function symbol contains information that will be used by the mixfix parser to read a term:

<name> ::= { *<symbol>* } +

<symbol> ::= *<single lexem>*
| **@**
| *<quoted single lexem>*
| **'** *<number>* **'**

<single lexem> ::= *<elan identifier>*
| *<number>*
| *<nchar>*

<quoted single lexem> ::= *<quoted identifier>*

where, as defined on page 27, *<elan identifier>* is any identifier except keywords. Notice that indeed keywords can be used without their special meaning when placed between quotes.

The characters '{', '}', '~' are restricted to the pre-processor use. Thus they cannot be used in any module, but they can be used in the specification file or in the description of the query, since both are not pre-processed. The ':' character indicates the end of the name and the character '@' is the place-holder for an operator '@'rgument. These two last characters can be used with another semantics than their built-in one, when placed between quotes.

A quoted number after the character '\' represents the definition of a character by its ascii code. This facility can be used in order to add to the syntactic form of a term non-visible characters or the characters '{', '}', '~'.

Finally an operator declaration is achieved by defining its *rank*, which is just a list of sort names with optional description of *selectors*.

<rank> ::= *<sort name>*
| ({ *<sort name>* } +) *<sort name>*
| ({ *<identifier>* : *<sort name>* } +) *<sort name>*

Example 3.2 *With the two declarations:*

```
@ + @              : ( int int ) int
+(@,@)             : ( int int ) int                    alias @+@:
```

the strings "$\mathtt{x}+\mathtt{y}$*" and "*$+(\mathtt{x},\mathtt{y})$*" represent the same term.*

When defining a constructor operator, it is possible to define the related selectors and modifiers, as outlined in the next example.

Example 3.3 *In the following module example, the selector* `first` *and* `second` *are specified and the system will automatically generate the corresponding selector and modifier operators and the corresponding rewrite rules.*

ElanExamples/simplePairSelectors.eln

This is similar to the following module:

ElanExamples/simplePair.eln

where the following rewrite rules are in fact automatically generated:

```
(x,y).first => x
(x,y).second => y
(x,y)[.first<-z] => (z,y)
(x,y)[.second<-w] => (x,w)
```

3.3.3 Function declaration options

As in languages like OBJ or ASF+SDF, an operator declaration carries many informations about the syntax and the semantics of the operator. We also need to specify complementary information like the precedence of the operator for the parser, or some of its semantic properties like associativity. This is done in ELAN using the following syntax:

<operator option> ::= **assocLeft**
| **assocRight**
| **pri** *<number>*
| (**AC**)
| **code** *<number>*
| **builtin** *<number>*

Parsing Associative operators The first two options are useful for defining syntactic associativity of symbols:

Example 3.4 *The declarations:*

```
@ * @              : ( int int ) int      assocLeft
*(@,@)             : ( int int ) int                    alias @*@:
```

allow to parse the term $\mathtt{x}*\mathtt{y}*\mathtt{z}$ *as* $*(*(\mathtt{x},\mathtt{y}),\mathtt{z})$.

Priorities The *priority* parsing problems can be solved using the priority option:

Example 3.5 *The declarations:*

```
@ + @              : ( int int ) int      assocLeft pri 10
@ * @              : ( int int ) int      assocLeft pri 20
```

allow to parse an expression like `x + y * z` *as usual in* `x + (y * z)`.

Coercions can be defined and even hidden as in the following example:

Example 3.6 *Assume that equations are constructed with the equality symbol:*

```
@ = @              : ( term term ) equation;
```

and equational systems as conjunctions of equational systems:

```
@ & @              : ( eqSystem eqSystem ) eqSystem;
```

An equation should be considered as an equational system, which corresponds to consider the sort `equation` *as a subsort of* `eqSystem`*. This is done by the declaration of an invisible operator:*

```
@                  : ( equation ) eqSystem;
```

This (invisible) coercion allows parsing the expression `(P & e)` *where* `P:eqSystem; e:equation`.

Semantic properties Semantic properties can be used. Currently only associativity and commutativity (abbreviated *AC*) is allowed and only for binary operators. The user should be aware that in interpreted mode, AC matching can be quite inefficient. It is strongly recommended to use the compiled mode in presence of AC operators.

Example 3.7 *One can define a union operator which is associative and commutative in the following way:*

```
@ ∪ @              : ( set set ) set      (AC)
```

User built-in operators In order to improve efficiency of the evaluation, built-in operators can be specified using the **code** and **builtin** options. The natural number specified after the **code** and **builtin** keywords specifies the index of this symbol in the symbol table of ELAN. A **builtin** option is for built-in operators without attached code defining its semantics, while the **code** option is for built-in operators with an attached procedure.

Example 3.8 *The symbol* `true` *is defined in the* `bool` *module as the first symbol of the symbol table:*

```
true               : bool                 builtin 1
```

Section 5.2 summarizes the codes already defined for ELAN's standard built-ins.
▷ Adding built-in operators is of course a delicate matter since the ELAN source code has to modified.

3.3.4 Strategy declaration

A *strategy operator* is similar to a function symbol as defined previously except that at least one of the sorts involved in the strategy rank is a strategy sort.

<strategy declaration> ::= *<new strategy declaration>*
| *<strategy alias>*

<new strategy declaration> ::= *<strategy name>* **:** *<strategy rank>* [*<strategy option>*]

<strategy alias> ::= *<new strategy declaration>* **alias** *<old strategy name>* **:**

In case of *aliasing*, *<new strategy declaration>* is only another name for the strategy operator *<old strategy name>* . The *two* names are then accessible and refer to the *same* object.
▷ The last alias defined is considered by the parser as having the highest priority with respect to the previously defined ones.

<old strategy name> ::= *<strategy name>*

<strategy name> ::= *<name>*

<strategy rank> ::= *<strategy sort name>*
| (*<strategy args>*) *<sort name>*
| ({ *<strategy arg>* } +) *<strategy sort name>*

<strategy arg> ::= *<sort name>*
| *<strategy sort name>*

<strategy args> ::= *<sort name>* *<strategy args>*
| *<strategy sort name>* { *<strategy arg>* } *

<strategy sort name> ::= < *<strategy sort name>* -> *<strategy sort name>* >
| < *<sort name>* -> *<sort name>* >
| < *<sort name>* -> *<strategy sort name>* >
| < *<strategy sort name>* -> *<sort name>* >

<strategy option> ::= *<elementary strategy option>*
| *<defined strategy option>*

<elementary strategy option> ::= **bs**

<defined strategy option> ::= **assocLeft**
| **assocRight**
| **pri** *<number>*
| (**AC**)

The **bs** attribute (for **b**asic **s**trategy) indicates that the definition of the strategy is given by a rewrite rule whose right-hand side only uses the constructions of the elementary strategies language defined in Section 3.5.1.

3.4 Definition of rules

Rewriting is the basic concept underlying the design of ELAN. The language allows defining labelled conditional rules with local assignments. It is important to realize the difference between:

– **labelled rules** whose evaluation is fully controled by the user defined strategies and,

– **unlabelled rules** which are intended to perform functional evaluation. They are applied using a built-in left-most inner-most strategy.

A *condition* controling the firing of a rule is a term of sort `bool` introduced by the **if** keyword. A *local assignment* is introduced by the keyword **where**. Its main purpose is to assign to local variables the result of the application of a strategy on a term.

3.4.1 Rule syntax

Rules are introduced in families, possibly restricted to a single element.

<family of rules> ::= **rules for** *<sort name>*
[{ *<variable declare>* } +]
[**global** { *<labelled or unlabelled rule>* } +]
[**local** { *<labelled rule>* } +]
end

<labelled or unlabelled rule> ::= *<labelled rule>*
| *<unlabelled rule>*

<labelled rule> ::= [*<rule label>*] *<rule body>* **end**

<rule label> ::= *<identifier>* [(variable { , *<variable>* } *)]

<unlabelled rule> ::= [ ] *<rule body>* **end**

<rule body> ::= *<term>* => *<term>* [{ *<if-where-choose>* } *]

<if-where-choose> ::= **if** *<boolean term>*
| **where** *<variable name>* **:=** ([*<Estrategy term>*]) *<term>*
| **where** *<variable name>* **:=** [*<Dstrategy term>*] *<term>*
| **where** (*<sort name>*) *<term>* **:=** ([*<Estrategy term>*]) *<term>*
| **where** (*<sort name>*) *<term>* **:=** [*<Dstrategy term>*] *<term>*
| **choose**
{ **try** *<if-where-choose>* } +
end

<variable declare> ::= [{ *<variable name>* , } * *<variable name>*] **:** *<sort name>* **;**

<variable name> ::= *<elan identifier>*

<Estrategy term> defined in Section 3.5.1 is a constant strategy name declared with the attribute **bs**. Note that it is optional.

<Dstrategy term> defined in Section 3.5.2 is a term built on the signature of strategy operators.

Remark: The syntax for terms is not defined here since it is user defined and induced from the rank of the operators. Because of the very liberal way of defining operator ranks, the term syntax in *mixfix* in the tradition of algebraic specification languages like ASF+SDF or OBJ. The term parser is based on Earley's context free parser [Ear70] where the grammar is extracted from the operator rank definitions.

— Release Note: *Since `V.3.0`, the choose-try construction, the patterns in where facility and the user defined strategies call by "`[ ]`" have been introduced.*

The rule labels accept now arguments that should be variables used in the rule.

In the **where** *construction with a term, the sort is just used to declare the sort of the term used as pattern. This is useful for the parser which then just checks that this is also the sort of the result on the right hand side of "***:=***". Note that the sort declarations is useless for the* **where** *construction using only variables, since the sort of the variable is always attached to it.*

There are two ways for calling strategies, according to the strategy language used by the programmer:

▷ A strategy call of the form "( )" can only be done using a strategy that is declared as **bs**.

▷ A strategy call of the form "[ ]" can only be done using a strategy that is *not* declared as **bs**.

Example 3.9 *Here is one way for defining the idempotency rules for booleans. Notice that the two rules are local to the module so that their label can only be used for defining strategies in this module. Note also that the two rules have, on purpose, the same label:*

ElanExamples/rule-ex.eln

Since several rules may have the same label, the resulting ambiguities are handled by the strategy evaluation as explained later. The rule label is optional and rules may have no name. This last case is designed in such a way that the intended semantics is functional, and it is currently the responsability of the rewrite rule designer to check that the rewrite system composed of all unlabelled rules is confluent and terminating. Unlabelled rules are handled by a built-in strategy (left-most inner-most) as fully described in Section 3.6.

3.4.2 The where construction

The **where** construction has several purposes. The first one is to modularize the computations by computing auxiliary results affecting them to local variables, thus providing computation sharing. The second is to direct the evaluation mechanism using strategies and the built-in left-most inner-most evaluation reduction. Let us review the possibilities offered by ELAN.

The **where** statement is first useful when one wants to call a strategy on subterms.

Example 3.10 *As a first introducing example, let us consider the module:*

ElanExamples/ruleWithWhere.eln

Using the rule `r0`*, the term* `a` *reduces in one step to* `b`*. Using the rule* `r1`*, the term* `b` *is first normalized into* `b'` *using unlabelled rules and then* `a` *is replaced by* `b'`*. Using the rule* `r2`*, the term* `b` *is first normalized into* `b'` *using unlabelled rules and then the strategy* `strat`*, defined elsewhere by the super-user, is applied on* `b'` *and the result is substituted to* `a`*.*

One can also perform more complex affectations of the local variables by using *patterns*.

The current syntax of the **where** statement with patterns can be rephrased as follows:

$$\textbf{where}\ (sort)p\ ::= ()term \mid (s)term \mid [s]term$$

where the pattern p should be of sort *sort*. AC-operators as well as non-linearity are allowed for patterns.

This pattern in where capability is exemplified in the next subsection.

3.4.3 Choose-Try

The construction **Choose try** allows factorizing common parts of rules with the same left-hand side. For instance the two rules

$$\begin{array}{llll} [\ell] & l & \Rightarrow & r \quad \textbf{where } y_1 := ()u_1 \\ & & & \quad \textbf{if } c_2'(l, y_1) \\ & & & \quad \textbf{where } y_3 := ()u_3 \\ [\ell] & l & \Rightarrow & r \quad \textbf{where } y_1 := ()u_1 \\ & & & \quad \textbf{if } c_2''(l, y_1) \\ & & & \quad \textbf{where } y_3 := ()u_3 \end{array}$$

are factorized into one rule:

$$\begin{array}{llll} [\ell] & l & \Rightarrow & r \quad \textbf{where } y_1 := ()u_1 \\ & & & \quad \textbf{choose} \\ & & & \quad\quad \textbf{try if } c_2'(l, y_1) \\ & & & \quad\quad \textbf{try if } c_2''(l, y_1) \\ & & & \quad \textbf{end} \\ & & & \quad \textbf{where } y_3 := ()u_3 \end{array}$$

In this factorized form, the term u_1 is normalized only once and the assignment to y_1 is performed also only once.

Example 3.11 *This example shows the use of patterns as well as the* **Choose try** *construction.*

ElanExamples/quick.eln

3.4.4 Conditions

The boolean term c given in a condition is reduced using the strategy that results from normalization using the non labelled rules and by the user defined strategies. If the computation stops and the resulting term is the boolean **true** then the condition is considered to be satisfied and the rule can be applied. If the computation stops and the resulting term is not **true** then the condition is considered as false.

3.4.5 Labels visibility

The **global** and **local** attributes make sense only for labelled rules. When a labelled rule is:

– **local,** then its label can only be used in the module in which the rule is declared,

– **global,** then its label can be used in all the modules importing the module defining the rule, with the visibility described in the section on modularity (see section 3.7).

▷ According to their semantics, unlabelled rules are always global.

— Release Note: *Since version V.3.0, the notion of single rule has been removed and the notion of rule locality has been introduced, so that rule labels can now be declared to be local to a module or, on the contrary, globally accessible. See section 3.9 for details concerning these changes.*
Remark: We will see next on page 49 how rule families can be constructed automatically in ELAN using the pre-processor construction "FOR EACH".

3.5 Definition of strategies

The notion of strategy is one of the main originalities of ELAN. In practice, a strategy is a way to describe which rewrite computations the user is explicitly interested in. It specifies when a given rule should be applied in the term to be reduced. From a theoretical point of view, a strategy can be viewed either as a function or as a subset of all proof terms defined by the current rewrite theory. The application of a strategy to a term results in the (possibly empty) collection of all terms that can be derived from the starting term using this strategy [KKV95, Vit94]. When a strategy returns an empty set of terms, we say that it *fails*.
▷ A strategy enumerates all the terms it describes, should this collection be finite or not. Consequently the user should note that in case (s)he writes a strategy that enumerates an infinity of terms, then the evaluation process will of course not terminate.

Two strategy languages are offered in ELAN. The language of *elementary strategies* is mainly based on regular expressions built from rule labels, while the language of *defined strategies* allows the user to define his own elaborated strategies, possibly recursively. Although the language of defined strategies embeds all the constructions of the language of elementary strategies, the latter is maintained since it is based on a more efficient implementation.

Families of strategy rules are intended to give a semantics to the strategy operators. The syntax of strategy rule families is the following:

```
<family of strategies> ::= strategies for <sort name>
                           [ { <strategy variable declare> } + ]
                         { implicit { <elementary strategy rule> } +
                         | implicit { <defined strategy rule> } +
                         | explicit { <defined strategy rule> } + } +
                           end

<strategy variable declare> ::= [ { <variable name> , } * <variable name> ]
                                :
                                <sort or strategy sort name> ;

<sort or strategy sort name> ::= <sort name>
                               | <strategy sort name>
```

The keyword **explicit** is only used with defined strategies and introduces a strategy rule that contains explicitly in its left-hand side the application operator [@]@ that will be introduced in Section 3.5.2. Otherwise, the strategy rule is introduced by the keyword **implicit**.

3.5.1 Elementary strategies syntax

.

The general syntax of elementary strategy rules is the following:

```
<elementary strategy rule> ::= [] <Estrategy name> => <Estrategy term> end
```

An elementary strategy rule has no label. Thus it is always global.

```
<Estrategy term> ::= <Estrategy name>
                   |  <choosing>
                   |  <concatenation>
                   |  <iterator>
                   |  <normalize>
                   |  <normalize with trace>
                   |  fail
                   |  id

<choosing> ::= dc ( <Estrategy term> { , <Estrategy term> } * )
             |  dk ( <Estrategy term> { , <Estrategy term> } * )
             |  first ( <Estrategy term> { , <Estrategy term> } * )
             |  dc one ( <Estrategy term> { , <Estrategy term> } * )
             |  first one ( <Estrategy term> { , <Estrategy term> } * )

<concatenation> ::= <Estrategy term> ; <Estrategy term>

<iterator> ::= iterate* ( <Estrategy term> )
             |  iterate+ ( <Estrategy term> )
             |  repeat* ( <Estrategy term> )
             |  repeat+ ( <Estrategy term> )

<normalize> ::= normalize ( <Estrategy term> )
              |  normalise ( <Estrategy term> )

<normalize with trace> ::= normin ( <Estrategy name> { , <Estrategy name> } * )
```

<Estrategy name> is a constant strategy name declared with the attribute **bs**.

Labelled rules

A labelled rule is the most elementary strategy and is called a *primal strategy*.

The application of a rewrite rule in ELAN yields a set of results. There are in general several ways to apply a given conditional rule with local assignments. This is first due to equational matching (e.g. AC-matching) and second to the **where** assignment, since it may itself recursively return several possible assignments for variables when using for example non-deterministic strategies.

Note that there may be several rules with the same label. If no rule labelled ℓ applies on the term t, the set of results is empty and we say that the rule ℓ fails.

A labelled rule ℓ can be considered as the simplest form of a strategy which returns all results of the rule application.

Thus the language provides basic constructions to handle this non-determinism through choice strategy operators.

Choice strategy operators

Let us first concentrate on expressing choices among the set of results on a primal strategy ℓ. By default the strategy ℓ returns all results of this primal strategy. If at most one result is needed, ℓ has to be encapsulated by the strategy operator **dc one** that returns a non-deterministically chosen result, or **first one** that returns the first result, when it exists.

Example 3.12 *Let us consider the following module (incompletely described here):*

ElanExamples/indeterminism.eln

The application of the rule `extractrule` *on the term* `empty U a U b` *can be done in two ways, since the AC-matching algorithm returns a complete set of matches consisting in the two substitutions:* { `S` ↦ `empty U a, e` ↦ `b`} *and* {`S` ↦ `empty U b, e` ↦ `a`}.

Using the strategy `extractrule` *on this term results in the set consisting of two terms* `a` *and* `b`. **dc one**(`extractrule`) *returns either* `a` *or* `b` *depending of the implementation of the non deterministic choice.* **first one**(`extractrule`) *returns either* `a` *or* `b` *depending of the implementation of the AC-matcher.*

More choice operators are defined and apply in general on a list of arguments.

- The **dk** operator, with a variable arity, is an abbreviation of *dont know choose*. $\mathbf{dk}(S_1, \ldots, S_n)$ takes all strategies given as arguments, and returns, for each of them the set of all its results. $\mathbf{dk}(S_1, \ldots, S_n)$ fails if all strategies $S_1, \ldots, S_n$ fail.

- The **dc** operator, with a variable arity, is an abbreviation of *dont care choose*. $\mathbf{dc}(S_1, \ldots, S_n)$ selects only one strategy that does not fail among its arguments, say S_i, and returns all its results. $\mathbf{dc}(S_1, \ldots, S_n)$ fails if all strategies $S_1, \ldots, S_n$ fail. How to choose S_i is not specified.

- A specific way to choose an S_i is provided by the **first** operator that selects the first strategy that does not fail among its arguments, and returns all its results. So if S_i is selected, this means that all strategies $S_1, \ldots, S_{i-1}$ have failed. Again $\mathbf{first}(S_1, \ldots, S_n)$ fails if all strategies $S_1, \ldots, S_n$ fail.

- If only one result is wanted, one can use the operators **first one** or **dc one** that select a non-failing strategy among their arguments (either the first or anyone respectively), and return a non-deterministically chosen result of the selected strategy.

Example 3.13 *Consider the following module:*

ElanExamples/testStrat.eln

Then the strategy `strat` *applied to* `a` *results in* `c`*, and the same strategy applied to* `b` *results in* {`e`,`f`}.

Note that the same rule label can be used for naming different rules. In fact this is equivalent to design rules with different labels and a choice operator to link them. For instance in Example 3.13, we could have the label `r3` again instead of `r4` for the fourth rule. The strategy could then be defined as `dc(dk(r1), dk(r2), dk(r3))`.

Strategies concatenation

Two strategies are concatenated by applying the second strategy on all the results of the first one.

The concatenation operator denoted "`;`" builds the sequential composition of two strategies S_1 and S_2. The strategy $S_1; S_2$ fails if S_1 fails, otherwise it returns all results (maybe none) of S_2 applied to the results of S_1;

Example 3.14 *In the same context as in the previous example (Example 3.13), the strategy:*

ElanExamples/testConc.eln

when applied on the term `a` *results in the application of first* `r1`*, then* `r5`*, and finally* `r3` *and* `r4` *in all possible ways, yielding the results* {`e,f`}*.*

Strategy iterators

In order to allow for the automatic concatenation of the same strategy, ELAN offers four powerful iterators:

The strategy **iterate*** corresponds to applying zero, then one, then two, ... n times the strategy to the starting term, until the strategy fails. Thus (**iterate***(`s`))`t` returns $\bigcup_{n=0}^{\infty}(s^n)t$. Notice that **iterate*** returns the results one by one, even when the iteration of the strategy never fails. This strategy cannot fail since it returns at least the initial term.

Its variant **iterate+** does not return the initial term but returns $\bigcup_{n=1}^{\infty}(s^n)t$. It can fail if s fails on the initial term.

The strategy **repeat** iterates the strategy until it fails and returns just the terms resulting from the last unfailing call of the strategy. The two variants are thus defined as:
(**repeat***(`s`))`t` = (s^n)`t` where (s^{n+1})`t` fails, $n \geq 0$ and $s^0 = id$,
(**repeat+**(`s`))`t` = (s^n)`t` where (s^{n+1})`t` fails and $n > 0$.

Example 3.15 *To illustrate the basic behaviour of the elementary strategies, let us consider the following module:*

ElanExamples/stratFail.eln

When applying the strategies on the terms `a` *or* `b`*, we are obtaining the following results:*

strategy	a	b
tryIt	{b}	∅
repeatS	{b}	{b}
repeatP	{b}	∅

In particular, the difference between the **repeat*** *and* **repeat+** *is due to to fact that when* `dc(a2b)` *is applied zero time, this is by definition the identity strategy, returning the initial term.*

Example 3.16 *The extraction of the elements of a list can be realized in the following way:*

ElanExamples/iterRepeat.eln

We then obtain the following results for the different strategies applied on the terms `1` *or* `element(1.2.3)` *respectively:*

strategy	1	element(1.2.3)
all0	∅	{1, element(2.3)}
allRepS	1	{1, 2, 3}
allRepP	∅	{1, 2, 3}
allIter	1	{element(1.2.3), element(2.3), element(3), 1, 2, 3}

The normalize strategy

Since labelled rules are applied only at the top of the terms, it is useful to have at hand a way to apply a family of labelled rules everywhere in a term to get it normalized. This is provided in ELAN by the **normalize** strategy. This strategy takes a strategy and use it in order to normalize the term on which it is applied.
▷ There is no assumption of the way the rules are applied, e.g. neither innermost nor outermost.

Example 3.17 *A typical example of the use of* **normalize** *is the normalization process of the typed lambda calculus, whose main module is described as follows:*

ElanExamples/normLambda.eln

The normalize strategy with trace

This new strategy (**normin**) normalizes terms with respect to a set of labelled rules in leftmost-innermost fashion. Moreover, a complete trace is generated (see option `-proofterm` of the compiler). This trace contains all informations about each rewrite step (*i.e.* the context, the rule and the substitution) and allows the normalisation process to be exported to external systems. The strategy **normin** is used, for example in an ELAN based tactic for (simple or AC) rewriting in the Coq proof assistant available at [Ngu].

Identity and Failure

Two more constructions are available:

- **id** is the identity strategy that does nothing, and never fails.
- **fail** always fails and returns an empty set of results.

3.5.2 Defined strategies

The defined strategy language extends the elementary one by the possibility to define recursive strategies. The application of a defined strategy is itself performed by an ELAN program that defines the interpretation of the new constructions introduced by the user. All the modules defining the appropriate syntax and semantics are provided in the ELAN library.

The syntax of the defined strategy language is described in an ELAN module called `strsig[X,Y]` when strategies ranks contain sorts of the form `<X->Y>` (strategies are non sort-preserving), and the module `strat[X]` when the strategies concern sorts of the form `<X->X>` (strategies are sort-preserving). To use non sort-preserving strategies, one has to import the module `tcstrat[X,Y]` where `X` and `Y` are different sorts. These modules implement the interpreter of the defined strategy language using a set of labelled rewrite rules and a basic strategy `eval`. The syntax of defined strategies is given below:

<defined strategy rule> ::=
| [ ] *<strategy body>* **end**
| [.] *<strategy body>* **end**
| [*<rule label>*] *<strategy body>* **end**

<*strategy body*> ::= <*Dstrategy term*> ==> <*Dstrategy term*> [{ <*if-where-choose*> } +]

<*if-where-choose*> is defined in Section 3.4.1.
<*Dstrategy term*> is a term built on the signature of strategy operators. It has a strategy sort. The different labels [], [.] and [ℓ] for strategy rules correspond to different evaluation modes explained in Section 3.6.2.

Example 3.18 *An example of the use of these features is the following:*

ElanExamples/map.eln

Example 3.19 *The following module shows how to program loops with the defined strategy language.*

ElanExamples/loop.eln

3.6 Evaluation mechanism

This section does not introduce any new syntactic material, but gives informal explanations on the operational semantics of the language.

3.6.1 Rewrite rules on terms

As we have seen, there are two kinds of rules: labelled and unlabelled. They define two classes of rules that are applied on the term to be evaluated in the following way:

– **Step 1** The current term is normalized using the unlabelled rules. This is done in order to perform functional evaluation and thus it is recommended to the user to provide a confluent and terminating unlabelled rewrite system to ensure termination and unicity of the result. This normalization process is built-in in the evaluation mechanism and consists in a leftmost innermost normalization. This yields always a single result.

– **Step 2** Then one tries to apply on the normalized term a labelled rule following the strategy described in the logic description. This leads to a (possibly empty) collection of terms. If this set is empty, then the evaluation backtracks to the last choice point; if it is not empty, then the evaluation goes on by setting a new choice point and evaluating one of the returned terms by going to **step 1**.

In a slightly more formal way, a rule

$$[\ell] \quad l \to r \quad s_1 \dots s_n$$

where the s_i are either **where** or **if** or **choose try** expressions, is applied on a term t by:

1. Matching l against t. This computes a multiset of substitutions (because of equational matching). If this set contains more than two elements, one is chosen and the other ones are stored for possible future backtracking. Let σ be the chosen substitution.

2. The evaluation goes on by evaluating the expressions $s_1, \dots, s_n$, one by one and in this *order* (i.e. from 1 to n).

3. If s_i is of the form **where** $x_i := (strat_i)t_i$, then one of the results (call it t'_i) of the application of the strategy $strat_i$ on the term t_i is chosen, and the substitution σ is extended by $x_i \mapsto t'_i$. The other results are stored for possible backtracking, and the evaluation goes on with s_{i+1}. If the strategy $strat_i$ fails on t_i, then we backtrack to the previous choice point.

4. If s_i is of the form **where** $p_i := (strat_i)t_i$, then one of the results (call it t'_i) of the application of the strategy $strat_i$ on the term t_i is chosen, and the substitution σ is extended by the matching substitution σ_i from p_i to t'_i. The other results are stored for possible backtracking, and the evaluation goes on with s_{i+1}. If the strategy $strat_i$ fails on t_i, then we backtrack to the previous choice point. Note that when p_i contains AC symbols AC-matching is called and there is an additional backtracking mechanism to find an adequate AC-match σ_i.

5. If s_i is of the form **if** c_i, then the term c_i is evaluated following the normalization strategy. If the result is the `bool` constant `true`, then one evaluates the next expression s_{i+1}, otherwise one backtracks to s_{i-1}.

6. If s_i is of the form

$$\begin{array}{l}\textbf{choose}\\ \qquad\textbf{try } s_1^1 \dots s_m^1\\ \qquad\dots\\ \qquad\textbf{try } s_1^j \dots s_k^j\\ \qquad\dots\\ \qquad\textbf{try } s_1^p \dots s_l^p\\ \textbf{end}\end{array}$$

   then we can mimic the evaluation mechanism by replacing all occurrences of ℓ in strategy expressions by p rules $\ell_1, \dots, \ell_p$ (in this order) defined as follows for $j = 1, \dots, p$:

$$[\ell_j] \quad l \to r \quad s_1 \dots s_{i-1}\ s_1^j \dots s_k^j\ s_{i+1} \dots s_n$$

   Since **choose try** is a recursive construction, this unfolding process is performed by a leftmost-innermost mechanism, until we get rule expressions with no occurrence of **choose try**.

▷ One should note that the term to be evaluated is *first* normalized by ELAN using the unlabelled rules and the resulting term is then reduced using the given strategy. The following example illustrates this behaviour.

Example 3.20

ElanExamples/normalizeFirst.eln

Applying the strategy `s1` *on the term* `f(a)` *gives no result since:*

1. `f(a)` *is normalized by the unlabelled rule into the term* `a`*,*

2. *then the normalization process tries to apply the labelled rule* `r1` *on* `a` *and fails. So* no *strategy applies! The set of results in empty.*

When applying the strategy `s2` *on* `f(a)`*, for the same reason the reduction process fails to apply the rule* `r1` *but the strategy* `identity` *can then be applied and the result is then* `a` *since the term is firstly normalized.*

Example 3.21 *Let us consider a brute force specification of the eight queens problem: How can one put eight queens on one chess-board in such a way that they do not interfere? In this example* $p1$ *refers to the position of the queen in the first column,* $p2$ *to the position of the second queen which should be in the second column and so on up to* $p8$*.*

ElanExamples/queens.eln

Note that the strategy above returns only one solution. If it is changed to a **dk***, we get all possibilities.*

Note also the way all the numbers between 1 and 8 are enumerated in ELAN using a **dk** *strategy (see* `tryrule`*).*

3.6.2 Rewrite rules on strategies

Elementary strategies are defined by rules of the form `[] c => strat` where `c` is a constant of sort `<s -> s'>` and `strat` a term built on elementary strategy constructors. Such rules are always unlabelled. Application of such a strategy `c` on a term `t`, denoted `(c)t` is performed by a C function generated by ELAN.

Defined strategies are like ordinary terms except that they have a functional sort of the form `<s -> s'>`. To understand how rules are applied on these strategy terms, it is helpful to distinguish two kinds of rule purposes: either the rule defines the result of the application of a strategy to a term (and involves implicitly or explicitly the application operator `[@]@`), or the rule defines a computation on strategy terms.

- Consider first the case of a rule which defines the result of the application of a strategy to a term.
  - When the rule is labelled by `[.]`, it is used by the strategy interpreter (i.e. the ELAN program given in the module `strat[X]` for sort preserving strategies or `strat[X,Y]` for others). For that, the rule is automatically transformed into a rule with the label `DSTR` and is applied with the default strategy `eval` of the strategy interpretor.
  - When the rule is labelled by $[\ell]$, it is used by the ELAN interpreter as any other labelled rule governed by a strategy. So the user has to provide a strategy involving ℓ.
- Rules that define a computation on strategy terms are evaluated exactly like rules on (ordinary) terms:
  - If the rule is unlabelled, the leftmost innermost predefined strategy of the interpreter is applied.
  - If it is labelled by $[\ell]$, the evaluation process needs a user-defined strategy involving ℓ.

Example 3.22 *We define here a small strategy library composed of several defined strategies that provide primitives for depth-first search in a computation tree. This strategy library is used to build a strategy that finds out a path (or, all paths) towards an exit of a labyrinth.*

The strategy library supports four primitives for depth-first search. The current situation of the partially discovered search tree is represented by a path from the initial state (the root of the tree) to the current state (the left-most leaf). It also remembers all possible alternatives (or, choice points) met along this path. Each step of the path is represented by a list of states, where its head is the currently chosen state, and the rest represents all non-explored alternative states. Each step of the path is a list of states, thus, the whole path could be naturally represented as list[list[state]]. This sort also describes the current situation during the depth-first search in this tree.

Four primitive strategies are defined over the sort list[list[state]].

- `call(S)` *applies the strategy S on the current state, where S is of sort* $\langle state \rightarrow state \rangle$. *All possible results of this application, if there are any, are grouped together into a new step (level) attached to the current path.*
- `next` *throws away the current state and continues searching with the next possibility of the previous step (the last choice-point).*
- `exit` *leaves the current state ignoring all alternatives and it returns the control to the previous step of the current path.*
- `cut` *eliminates all alternative states of the last step.*

ElanExamples/space.eln

The variable state represents the current state, the variable level represents all alternatives of the current state, i.e. state.level : *list*[*state*] *is one step of the path, where the others steps are recorded in the variable space.*
The only non-trivial strategy among the four primitives is the strategy call(S), *which uses the function symbol set_of to collect all results of an application of a strategy S.*

In order to design a strategy helping to find out an exit from a labyrinth, we split the problem into a part dependent on the labyrinth and an independent one. The dependent part contains a specification of a state, which is a pair of coordinates, and a definition of four basic moves in the labyrinth. They are realized as state transforming strategies left, right, up, down of sort $\langle state \rightarrow state \rangle$ *such that these strategies fail, if some movement in a given state is not possible (because of a wall). All exits of the labyrinth are specified by a strategy exitable, which fails, if the current state has 'no exit doors'. Otherwise, it is equivalent to the identity strategy.*

ElanExamples/robot.eln

The independent part of the strategy consists of a definition of the strategy `search` *of sort* $\langle list[list[state]] \mapsto list[list[state]] \rangle$ *and an auxiliary strategy* `moves` *of sort* $\langle state \mapsto state \rangle$. *The strategy* `search` *succeeds on an exitable state (a room with exit doors). Otherwise, if the current path contains a loop, it backtracks and takes the next possible state to go on. If it does not contain a loop, it either moves to the neighbour state and goes on searching, or, if no exit has been found, it ignores the current choice, and backtracks to the next alternative.*

3.7 Modules

The modularity constructions in the current version of ELAN are as simple as possible and the semantics of parametrisation as well as importation is textual expansion.

The top-level information that should be provided by the super-user is the description of the logic (s)he wants to make available. This is achieved in ELAN by using two kinds of modules. The

first one describes the top level of the logic and are presented in files having the '.lgi' extension, the second ones contain all subsequently needed informations. They are called element modules and they are presented in files having the extension '.eln'. Element modules have been described in Section 3.2.

3.7.1 Visibility rules

The visibility rules in ELAN are the following:

- Only sorts are global. This means that they can be used in any module without any importation command. As a consequence, their name is unique in the ELAN universe built for a run.
- Operators, rewrite rules and strategies can on the contrary be local or global. A local operator will be known only in its own module (i.e. the module where it is defined). If an operator is declared to be global, then it is known in all modules importing the module where it is declared. This can be slightly modified, when qualifying the importation command by local or global. By default an importation without local or global specification is assumed to be local.
- A special case is for aliases. The visibility rules for them are the same as for operators. The only interesting case is for a module that locally imports a signature but exports the alias definition. In this case, only the alias name of the operator will be known outside and not its original name: this is quite useful for renaming purposes.

— Release Note: *The main change concerning the visibility rules is that now only sorts are always global. See section 3.9 for a summary of all syntax modifications.*

3.7.2 Built-in modules

For convenience and efficiency several modules are provided as built-ins. See section 5.2 for a full description of the current available ones.

3.7.3 Parameterised modules

Parameterised modules can be defined in ELAN as follows:

<formal module name> `::=` *<identifier>*
`|` *<identifier>* `[` *<identifier>* { `,` *<identifier>* } * `]`

The module instantiation by its actual parameters is currently done by the syntactical replacement of the formal arguments by the actual ones.

Example 3.23 *The classical example is the one of parameterised lists:*

ElanExamples/simpleList.eln

and we get list of integers by instantiating `X` *by* `int` *and importing* `int` *as in:*

ElanExamples/simpleList.lgi

3.7.4 LGI modules

The top level description of the logic consists of:

1. the description of the syntax that the user is allowed to use in order to give his specifications.
2. the description of a term, called the query, that will be reduced in the rewrite logic defined by the (super) user.

The syntax is the following:

<lpl description> ::= **LPL** *<identifier>* **description**
[*<specification description>*]
query
of sort *<sort name>*
result of sort *<sort name>*
import { *<instantiated module name>* } +
[**check with** *<boolean term>*]
start with *<start term>*
end

<specification description> ::= **specification description**
{ *<specification part description>* } +
end

<specification part description> ::= **part** *<identifier>*
of sort *<sort name>*
import { *<instantiated module name>* } +
[**check with** *<boolean term>*]

<start term> ::= ([*<Estrategy name>*]) *<query term>*

The syntax of of *<query term>* is built over the user-defined signature enriched by the constant **query**. The sort of **query** is given by the query declaration above.

Example 3.24 *If no specification is needed, the specification description part is skipped, as in the following logic description used for running Example 3.13:*

ElanExamples/testStrat.lgi

Example 3.25 *Here is a simple example of the top level description of how unification can be implemented:*

ElanExamples/unification.lgi

In this unification logic, the user should provide the specification information as follows:

ElanExamples/simplesig.spc

The user should use the (super-user defined) keywords `Vars` *and* `Ops` *to define respectively the variables and the operators, namely* `a` *of arity 0,* `f` *of arity 2, etc...*

Semantics

When parsing a specification, ELAN generates new modules that complete the definition of the rewrite theory to be used during the evaluation of the query. For instance, in the case of the previous example, these new modules are:

ElanExamples/moduleVarUnif.eln

and

ElanExamples/moduleOpsUnif.eln

When a user provides a specification to an ELAN logic, this simply provides more information about the context in which the user wants to work, like the name of function symbols or basic axioms to use.

3.8 Pre-processing

Another original feature proposed by ELAN is the use of a pre-processing phase that allows to describe the logic to be encoded in an easier way.

As described in Figure 1.1, the pre-processor is performing textual replacements starting from informations given by the super-user in the modules, and by the user in the specification file. The pre-processing phase in ELAN can just be thought of as a macro expansion mechanism which extends the parameterization process described before.

In order to express the syntax of the pre-processor construction, we need the notion of *constant expression* and of *text* defined as follows:

<constant expression> `::=` *<number>*
`|` `(` *<subexpression>* `)`

<subexpression> `::=` *<number>*
`|` *<subexpression>* `+` *<subexpression>*
`|` *<subexpression>* `-` *<subexpression>*
`|` *<subexpression>* `*` *<subexpression>*
`|` *<subexpression>* `/` *<subexpression>*
`|` *<subexpression>* `%` *<subexpression>*
`|` `(` *<subexpression>* `)`

<lexem> `::=` *<identifier>*
`|` *<number>*
`|` *<char>*

<text> `::=` `{` *<lexem>* `}` $^{+}$

3.8.1 Simple duplication

A first feature is to allow simple duplications of a part of text. The syntax is:

<simple repetition> `::=` *<lexem>* $\tilde{}$ *<constant expression>*
`|` `{` *<text>* `}`$\tilde{}$ *<constant expression>*

Example 3.26 *The text "*`a{,b}~3`*" is processed into "*`a, b, b, b`*".*

3.8.2 Duplication with argument

It is often necessary to allow description of objects like $f(t_1, \ldots, t_5)$. This is possible using the syntax:

<arg.repetition> ::= **{** *<text>* **}_** *<identifier>*
=
<constant expression> **...** *<constant expression>*

Example 3.27 *The text "*`P { & s_I=t_I }_I=1...3`*" is processed into "*`P & s_1=t_1 & s_2=t_2 & s_3=t_3`*".*

A special form of this duplication with arguments is the explicit construction of a list of indexed identifiers allowed using the syntax:

<arg.repetition> ::= *<identifier>* _ *<number>* *<char>* **...** *<char>* *<identifier>* _ *<number>*

Example 3.28 *The text "*`t_1,...,t_5`*" is processed into "*`t_1 , t_2 , t_3 , t_4 , t_5`*".*

3.8.3 Enumeration using FOR EACH

This construction allows to make the link between the specification given by the user and the logic described by the super-user. A natural motivation for this construction is given by the "standard" way inference rules are used. For example when describing how unification works, the following transformation rule is given:

Decompose $P \wedge f(s_1, \ldots, s_n) =^? f(t_1, \ldots, t_n) \rightarrow P \wedge s_1 =^? t_1 \wedge \ldots \wedge s_n =^? t_n$

It is generic in the sense that the operator f is unspecified. It can be a + of arity 2, or a $if@then@else@$ of arity 3, or just a constant. We do not want, when specifying how the logic works, to give only the specific cases, we need to be as generic as possible.

ELAN provides via the FOR EACH construction of the pre-processor, a way to be generic. The syntax is the following:

<for each const.> ::= **FOR EACH** *<vardecl>* **SUCH THAT** *<varaffect>* **: {**
<text>
}

<vardecl> ::= *<varname>* { **,** *<varname>* } * **:** *<sort name>*
| *<vardecl>* **;** *<vardecl>*

<varaffect> ::= *<varname>* **:=** **(** `[` *<strategy name>* `]` **)** *<term>*
| *<varaffect>* **AND** *<varaffect>*
| *<varaffect>* **ANDIF** *<boolean term>*

▷ Since the symbols { , }, ~ are reserved for pre-processor use, it is not possible to use them for any other purpose, even quoted.

▷ The pre-processor uses the character '_' (underline) for assembling identifiers. For example, the text 'a f _1_ x b" is pre-processed into the sequence of three identifiers a, f_1_x and b. The character '_' is thus part of an identifier even if there is a space between it and the rest of the string. It is thus better for the user not to use the _ symbol, except for building identifiers of course.

Example 3.29 *The rule* **Decompose** *that we mentioned at the beginning of this section can be expressed in the following way:*

ElanExamples/decomposition.eln

If the specification given by the user consists in the operator symbols `f` *and* `g` *of respective arities 2 and 1, then the pre-processor expands the previous* **FOR EACH** *construction into:*

ElanExamples/decompositionExpanded.eln

3.9 Differences with previous version of the language

The language has evolved a lot between ELAN version 1.17 and ELAN version 3.0. The changes concern the syntax of:

- modules,
- rules and
- strategies.

The main changes are emphasized in the corresponding sections of the language description, and the main concern has been to uniformize the language constructions.

The transformations needed to change from the syntax of version 1.17 to the current one (i.e. later than 2.00) are summarized below.

Concerning the declaration parts:

- `import ...` becomes `import ... end`
- `sort ...` becomes `sort ... end`
- `op ... endop` becomes `operators ... end`

Now, there is only one construction to define rules, you have to replace:

- `rule for ...` by `rules for ...`
- `rule <name> for ...` by `rules for ... [name] ...`
- suppress `declare` keyword
- `body` or `bodies` by `global` or `local`
- `end of rule`, `end of rules` by `end`

The rule labels accept now arguments that should be variables involved in the rule.

Definition of strategies looks like rules definitions, you have to replace:

- `strategy ...` by `strategies for ... global|local [name] ...`
- `iterate ... enditerate` by `iterate*(...)`
- `while ... endwhile` by `repeat*(...)`
- `repeat ... endrepeat` by `repeat*(...)`

- `dont know choose(...)` by `dk(...)`
- `dont care choose(...)` by `dc(...)`
- `end of strategy` by `end`
- `end of module` by `end`

Local affectations and conditions are now evaluated from top to bottom. The order of presentation has been reversed, so that the *old* syntax:

```
[] l => r
        if c2
        where v2:=(...) ...
        where v1:=(...) ...
        if c1
```

becomes in the *new* syntax:

```
[] l => r
        if c1
        where v1:=(...) ...
        where v2:=(...) ...
        if c2
```

Chapter 4

ELAN: the system

This section is devoted to the user environment of the ELAN language, and is partly borrowed from [BKK+98b].

4.1 Global architecture

The ELAN environment consists of several components, as depicted in Figure 4.1. The preprocessor expands a few concise constructions allowed in the language. The parser checks the syntax of programs and verifies that terms are syntactically well-formed. The interpreter is an interactive tool allowing the user to check that the results he expects are indeed obtained. Initially, the preprocessor and the parser were integrated in the interpreter and so it was not possible to call them as stand-alone processes. The compiler transforms specifications into independent executable C code (Section 4.5). It is a stand-alone tool without any preprocessing and parsing facilities. The preprocessing and parsing phases are provided by the interpreter, which communicates the result of these processes to the compiler via a data exchange format called REF. Section 4.6 describes some tools to translate ELAN to REF and vice versa. Especially, the translation from ELAN to REF performed by the `query2ref` tool relies mainly on reusing the parser initially integrated in the interpreter.

4.2 The preprocessor

The ELAN syntax provides a few fancy constructions to perform textual replacements. So the preprocessor may be used to automatically generate parts of specifications used to analyse the rest of a program. It should be emphasised that there is strong interaction between the parser, the preprocessor and the interpreter. See Section 3.8 for a description of the preprocessor functionality.

4.3 The parser

The ELAN parser is based on Earley's one [Ear70] and is using sort informations in order to determine as soon as possible in the analysis the right choice. This explains why in the current version, some construction of the language require to give the sort information together with the term.

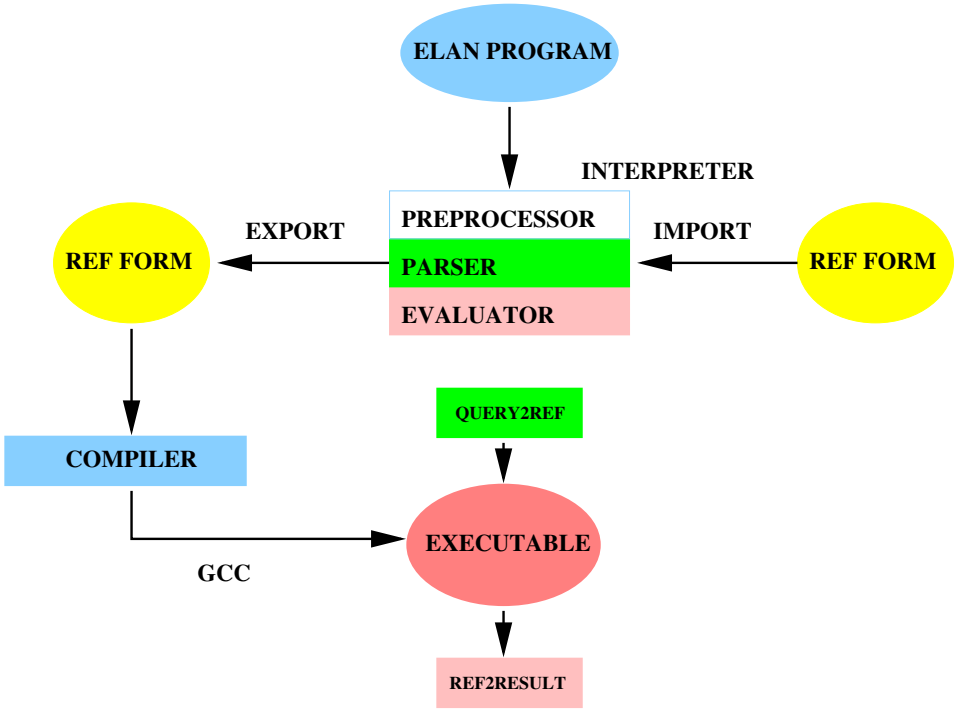

Figure 4.1: Global architecture of the ELAN environment

4.4 The interpreter

The interpreter takes a well-formed program and a well-formed query (both checked by the parser) and applies the rules and strategies defined in the program to the query. In order to find which rules can apply, the selection is guided by the top symbol of the rules: only those rules whose left-hand side has the same top symbol as the term to be reduced are selected. They are then tried in the order given in the program. Another kind of choices arbitrarily made by the interpreter is for the strategy $\mathbf{dc}(S_1, ..., S_n)$ that should select randomly a non-failing strategy among $S_1, ..., S_n$. In practice, the interpreter selects the first one, so implements **dc** and **first** in the same way. Note however that there exists a version of ELAN which concurrently executes the n strategies and selects the first one which terminates without failure [BC98].

Once a set of rules is selected, a many-to-one matching algorithm is applied. When associative and commutative (AC for short) operators are involved, an external one-to-one AC-matching algorithm described in [Eke95] is called. This algorithm is not fully integrated in the interpreter, so data structure conversions are required and lower the efficiency of the AC-matching, already quite complex. Once a match is found, local evaluations are performed and if all succeed, the result term is built, taking advantage of the right-hand side of the rule and of term sharing.

4.5 The compiler

The ELAN compiler transforms a logic description into an executable binary file. The compiler (for detailed description, see [Vit96, MK98]) produces an efficient `C` code, which is later compiled by the `cc` or the `gnu gcc` compiler. The executable binary code is dependent on the particular architecture, because a small part of the ELAN library performing the basic non-deterministic operations is written in assembly language [Mor98]. Up to now, this is the only reason that

makes the compiler available only on the architectures DEC-ALPHA, SUN4 and Intel-PC.

4.5.1 Non-linear rules

The ELAN compiler does not support non-left-linear rules (a rule is called non-left-linear if a variable occurs more than once in the left-hand side). The many-to-one matching algorithm only compiles linear rules. The ELAN parser automatically transforms non-left-linear rules into left-linear conditional rules by renaming variables that occurs more than once and adding some conditions to ensure their equality.

For example, the rule `f(x,x) => g(x)` is transformed into `f(x,y) => g(x) if x==y`.

4.5.2 Built-ins

To be more efficient, builtin data type such as *builtinInt* and *identifier* are compiled in a particular way.

The main drawback of this approach is that only completely defined functions can be defined on builtin data type.

Let `f(@) : (builtinInt) builtinInt` be an operator. For each integer n, $f(n)$ must be reducible to an integer. Suppose defined the rule `[] f(n) => n-1 if n > 1`, the term `f(0)` can not be reduced to a builtin integer. In this case, behaviour is undefined. This explains why *builtinInt* are never used directly: it is recommended to use the sort *int* which is no longer a builtin data type.

4.5.3 Input/Output facilities

Input/Output facilities are now fully implemented in the compiler. Their syntax is described in the module **IO.eln** of the standard library.

As in the version V3.4, it is possible, in the generated program, to enter a query term in the ELAN syntax, just like in the interpreter. In the same way, result terms computed by the generated program are now pretty-printed in the ELAN syntax. Therefore, it is no more necessary to give a fixed ground term in the `.lgi` file, and `.lgi` files involving the `query` keyword can now be compiled. In the current version, the query is parsed dynamically in the generated program by reusing a subpart of the parser integrated to the interpreter.

The generated program can be executed with several switches used to specify the different possible formats of input (query) and output (result) terms:

Input switches

- `-REFInput`: the query term must be in the REF format (see Section 4.6).
- `-noInput`: the query ground term is supposed to be located in the `.lgi` file, where it replaces the `query` keyword.

Output switches

- `-REFOutput`: result terms will be given in the REF format (see Section 4.6).
- `-noOutput`: results are computed but not written out. The interest of this switch is rather limited, but it can be useful for people who only want the statistics of the computation, like the number of rewrite steps per seconds.

- `-internalOutput`: result terms are given by using a prefix notation. This is the original syntax used in the first version of the compiler.

The generated program, called by default *a.out*, can be executed with some other switches not necessarily related to Input/Ouput, like `a.out -quiet` or `a.out -debug`.

At this point, it is still possible to change the content of the `start with` instruction declared in the `.lgi` file. The switch `-strategy` <*strategy name*> `:` <*sort name*> will apply the strategy <*strategy name*> on a query term of sort <*sort name*> .

Similarly, the switch `-sort` <*sort name*> will apply the empty strategy on a query term <*sort name*> . Note that `-strategy` and `-sort` are mutually exclusive.

▷ The list of available sorts (resp. strategy names) can be found in the `.ref` file.

4.6 The REF format

The Reduced ELAN Format, REF for short, is a term-like representation of ELAN programs introduced both for the interconnection of software components involved in the ELAN system, and for combining ELAN programs. This format is also of greatest interest for the implementation of reflection techniques in ELAN. A detailed description of its applications can be found in [BKK+98b].

The REF format has been initially developed to reuse the powerful Earley's parser integrated in the interpreter. By loading with the interpreter an ELAN program possibly in mixfix notation, we can get a REF program which can be viewed as the external representation of the interpreter working memory. REF is easier to parse than ELAN since the REF grammar is simply LALR. The ELAN interpreter generates a REF file *prog.ref* thanks to the option `--export`:

```
elan --export prog.ref prog.lgi [prog.spc]
```

The REF program contained in *prog.ref* can be directly executed by the interpreter as follows:

```
elan --import prog.ref
```

Then, the behavior of the interpreter is exactly the one expected with the command:

```
elan prog.lgi [prog.spc]
```

but without the time-consuming preprocessing and parsing phases.

The core of the ELAN compiler, called REM for "Reduced ELAN Machine" and implemented in `Java`, only knows the REF syntax. This explains why we need a script `elanc` to first call the interpreter in order to obtain a REF file, and second to execute REM with this REF file as input.

Then, a compiled `C` program, say `a.out`, is generated by REM. This program can be executed with different options like `a.out -REFInput`, `a.out -REFOutput` or even better `a.out -REFInput -REFOutput`:

- The option `-REFInput` reads a new query in REF syntax to replace the ground query occurring initially in *prog.lgi*.
- The option `-REFOutput` writes all result terms computed by *a.out* in REF syntax, instead of the ELAN syntax used by default.

So, we still have to convert an ELAN query to a REF query, and conversely to retrieve the ELAN syntax of result terms written in REF syntax.

For this purpose, one can use another tool to read a query and to pretty-print result terms of the *a.out* program, both in ELAN syntax, similarly to the interpreter. This tool is called `prettyIO` and may be used as follows:

```
prettyIO a.out prog.ref
```

Note that *prog.ref* is needed to find the ELAN syntax of functions implemented by *a.out*, and so to encode a query term provided by the user in ELAN syntax, and to decode terms written in REF syntax by `a.out -REFInput -REFOutput`. The executable `prettyIO` is implemented as a script calling some basic tools:

- `query2ref prog.ref` reads repeatedly (from `stdin`) a sort, and parses a term of this sort written in ELAN syntax. The result (written to `stdout`) is the corresponding term in REF syntax. By using the option `--query` as follows:

  ```
  query2ref prog.ref --query
  ```

  we only parse terms of the sort of the query, as indicated in *prog.lgi* (this information also occurs in *prog.ref*).

- `ref2result prog.ref` reads repeatedly (from `stdin`) terms encoded in REF syntax and writes (to `stdout`) the related terms in ELAN syntax, by using the signature occurring in *prog.ref*.

The script `prettyIO` is located in the directory `bin/` like `elanc`, and the compiled `C/C++` programs `query2ref` and `ref2result` are located in the subdirectory of `bin/` related to the current system architecture, like the interpreter `elan`.
▷ The user should be aware that `prettyIO` is much less interesting in the current version, since it is now possible to parse a query term and to pretty-print all results by simply using the generated program *a.out* itself.

Chapter 5

The standard library and the built-ins

This chapter presents the main features of the ELAN standard library. Remember that the path to this library is specified in the system variable ELANLIB whose default value is set to the subdirectory `elan_library/elanlib` of the source directory. ELAN can be used without any reference to this library, *except* for what concerns the use of the built-in objects which are documented in the second part of this chapter.

This library has been designed to be small and as efficient as possible. In particular *no* AC operators is used. The resulting code is more efficient, at the price of sometimes heavier descriptions.

5.1 The ELAN standard library

The standard library is fully documented in [BCD+98] to which we refer for details. We give below the list of files defined in the standard library. Some of them define built-in sorts and operators and in this case, they also appear in the next section (section 5.2).

5.1.1 common

In this category, the most common abstract data types are defined.

anyIdentifier.eln
anyInteger.eln
array.eln
arrayLIN.eln
arrayLOG.eln
bool.eln
builtinArray.eln
builtinHashTree.eln
builtinInt.eln
builtinIO.eln
builtinStdio.eln
builtinString.eln
builtinSyntacticMatchin.eln
cmp.eln
eq.eln
identity.eln
int.eln
integerConstants.eln
IO.eln
list.eln
occur.eln
pair.eln
prompt.eln
Query.eln
replace.eln
stdio.eln
string.eln
strlist.eln
tuple.eln

5.1.2 ref

All the modules needed to deal with the Reduced ELAN Format.
Meta_Apply.eln ref_modules.eln ref_terms.eln
REF.eln ref_read.eln ref_unify.eln
ref_rules.eln ref_unify_builtin.eln
ref2string.eln ref_sorts.eln ref_write.eln
ref2term.eln ref_strategies.eln refstring2string.eln
ref_idents.eln ref_tables.eln string2refterm.eln

5.1.3 noquote

These modules describe a possible syntax for the user specifications. More complicated syntax (e.g. mixfix) can also be defined.
applySubstOn.eln hornClauseSyntax.eln syntacticUnification.eln
atom.eln ident.eln termCommons.eln
atomP.eln identifier.eln sigSyntax.eln
eqSystem.eln substitution.eln

5.1.4 strategy

These modules are used for the definition and evaluation of defined strategies.
Any.eln booleg.eln meta.eln strsig.eln
any.eln Meta_apply.eln strapply.eln symbol.eln
Meta_capply.eln commit.eln strat.eln tcstrat.eln
Meta_strat.eln strcapply.eln loops.eln strconc.eln

5.2 The built-ins

There are two types of built-in symbols:

- **The built-in constructors**, which are non-reducible symbols, built into the ELAN system. Terms built over them are either constructed, or interpreted by the ELAN system, e.g. the symbol `Error` defined in the module .../common/builtinIO.eln as follows:

```
Error(@) : (builtinInt) X   code 122;
```

  which is used to report various errors of the I/O sub-system written in C++.

- **The built-in functions**, which represent symbols with pre-defined semantics, in general, not [easily] specifiable in ELAN itself. The symbol `open` in the module .../common/builtinStdio.eln defined as follows:

```
open_builtinStdio(@,@)     : (builtinString builtinString) builtinInt code 116;
```

  can be taken as an example. In this case, the declaration (profile) of the symbol `open` represents an interface between the ELAN specification and its built-in semantics. In this case, the code "116" identifies its semantics.

If there is an overloaded built-in symbol whose exact semantics depends on its profile, a procedure implementing the built-in symbol should know, which instance of the built-in symbol is executed. The overloaded symbol `read_builtinIO` is an example of this problem:

```
read_builtinIO(@,@)      : (builtinInt X) X   pri 200 code 120;
```

When the built-in symbol `read_builtinIO` is used for several sorts `X`, there exist several built-in symbols `read_builtinIO` in the user's specification. All of them are linked with the same semantic function (with the code 120) parameterized by the concrete profile of `read_builtinIO`. Thus, the sort of a read term is known in the semantic procedure due to the profile of actually applied symbol `read_builtinIO`. The difference between two kinds of *built-in function* symbols is expressed by the sign of code numbers.

5.2.1 Booleans

At the beginning there is nothing, so ELAN provides the true and false values and introduces the bool module. These two values are built-in and are deeply connected to the implementation of conditions in rewrite rules.

/elanlib/common/bool.eln

Builtins	Code	Signature	Comments
`false`	0	`() bool`	The logical `false`.
`true`	1	`() bool`	The logical `true`.
`@ and @`	21	`(bool bool) bool`	The logical conjonction.
`@ or @`	22	`(bool bool) bool`	The logical disjonction.
`@ xor @`	23	`(bool bool) bool`	The logical exclusive disjonction.
`not(@)`	24	`(bool) bool`	The logical negation.

/elanlib/common/cmp.eln

To enrich the booleans, *polymorphic* equality, disequality and inequalities are defined and are also built-in:

Builtins	Code	Signature	Comments
@ == @	18	`(X X) bool`	Tests the equality between two elements of the same sort `X`.
@ != @	19	`(X X) bool`	Tests the inequality between two elements of the same sort `X`.

5.2.2 Numbers

Numbers can of course be created "by hand", but we choose in ELAN to provide built-in integers.

/elanlib/common/builtinInt.eln

Builtins	Code	Signature	Comments
`@ + @`	310	`(builtinInt builtinInt) builtinInt`	The addition on integers, failing in case of a signed overflow.
`@ - @`	311	`(builtinInt builtinInt) builtinInt`	The substraction on integers, failing in case of a signed overflow.
`@ * @`	312	`(builtinInt builtinInt) builtinInt`	The multiplication on integers, failing in case of a signed overflow.
`@ / @`	327	`(builtinInt builtinInt) builtinInt`	The division on integers, of course failing in case of a division by 0, but also in case of a signed overflow.
`@ % @`	27	`(builtinInt builtinInt) builtinInt`	The modulo operation, of course failing in case of a division by 0.
`- @`	313	`(builtinInt) builtinInt`	The opposite of an integer, failing in case of a signed overflow.
`uplus(@,@)`	323	`(builtinInt builtinInt) builtinInt`	The addition on unsigned integers having the same binary representation, failing in case of an unsigned overflow.
`uminus(@,@)`	324	`(builtinInt builtinInt) builtinInt`	The substraction on unsigned integers having the same binary representation, failing in case of an unsigned overflow.
`utime(@,@)`	314	`(builtinInt builtinInt) builtinInt`	The multiplication on unsigned integers having the same binary representation, failing in case of an unsigned overflow.

Builtins	Code	Signature	Comments
`udiv(@,@)`	321	`(builtinInt builtinInt) builtinInt`	The division on unsigned integers having the same binary representation, of course failing in case of a division by 0.
`umod(@,@)`	322	`(builtinInt builtinInt) builtinInt`	The modulo operation on unsigned integers having the same binary representation, of course failing in case of a division by 0.
`unfailing(@ + @)`	3	`(builtinInt builtinInt) builtinInt`	The addition on integers, not failing in case of an overflow.
`unfailing(@ - @)`	4	`(builtinInt builtinInt) builtinInt`	The substraction on integers, not failing in case of an overflow.
`unfailing(@ * @)`	5	`(builtinInt builtinInt) builtinInt`	The multiplication on integers, not failing in case of a signed overflow.
`unfailing(@ / @)`	6	`(builtinInt builtinInt) builtinInt`	The division on integers, of course failing in case of a division by 0, but not in case of an overflow.
`unfailing(- @)`	20	`(builtinInt) builtinInt`	The opposite of an integer, not failing in case of an overflow.
`unfailing(utime(@,@))`	325	`(builtinInt builtinInt) builtinInt`	The multiplication on unsigned integers having the same binary representation.
`overflow(@ + @)`	315	`(builtinInt builtinInt) builtinInt`	The unsigned "carry-out" overflow of addition on integers.
`overflow(@ + @)`	316	`(builtinInt builtinInt) bool`	Indicates if there is a signed overflow for addition on integers.
`overflow(@ - @)`	317	`(builtinInt builtinInt) builtinInt`	The unsigned "carry-out" overflow of substraction on integers.
`overflow(@ - @)`	318	`(builtinInt builtinInt) bool`	Indicates if there is a signed overflow for substraction on integers.

Builtins	Code	Signature	Comments
`overflow(@ * @)`	319	`(builtinInt builtinInt) builtinInt`	The overflow value (most significant part) of multiplication on integers.
`overflow(@ * @)`	329	`(builtinInt builtinInt) bool`	Indicates if there is a signed overflow for multiplication on integers.
`overflow(@ / @)`	328	`(builtinInt builtinInt) bool`	Indicates if there is an overflow of division on integers.
`overflow(- @)`	320	`(builtinInt) builtinInt`	Carry-out overflow of the opposite of an integer.
`overflow(- @)`	335	`(builtinInt) bool`	Indicates if there is a signed overflow for the opposite of an integer.
`overflow(utime(@,@))`	326	`(builtinInt builtinInt) builtinInt`	The overflow value (most significant part) of multiplication on unsigned integers having the same binary representation.
`@ & @`	28	`(builtinInt builtinInt) builtinInt`	bitwise conjunction.
`@ | @`	29	`(builtinInt builtinInt) builtinInt`	bitwise disjunction.
`@ ∧ @`	330	`(builtinInt builtinInt) builtinInt`	bitwise exclusive disjunction.
`neg(@)`	331	`(builtinInt) builtinInt`	bitwise negation.
`shiftl(@)`	332	`(builtinInt) builtinInt`	bit shift to the left.
`shiftr(@)`	333	`(builtinInt) builtinInt`	bit shift to the right, filling with sign bit.
`ushiftr(@)`	334	`(builtinInt) builtinInt`	bit shift to the right, filling with 0.
`@ == @`	8	`(builtinInt builtinInt) bool`	The equality test.
`@ != @`	9	`(builtinInt builtinInt) bool`	The inequality test.
`@ < @`	10	`(builtinInt builtinInt) bool`	The strict ordering test between integers.
`@ <= @`	11	`(builtinInt builtinInt) bool`	The ordering test between integers.
`@ > @`	12	`(builtinInt builtinInt) bool`	The strict ordering test between integers.
`@ >= @`	13	`(builtinInt builtinInt) bool`	The ordering test between integers.
`eq_builtinInt(@,@)`	8	`(builtinInt builtinInt) bool`	The equality test.

Builtins	Code	Signature	Comments
`neq_builtinInt(@,@)`	9	`(builtinInt builtinInt) bool`	The inequality test.
`less_builtinInt(@,@)`	10	`(builtinInt builtinInt) bool`	The strict ordering test between integers.
`lesseq_builtinInt(@,@)`	11	`(builtinInt builtinInt) bool`	The ordering test between integers.
`greater_builtinInt(@,@)`	12	`(builtinInt builtinInt) bool`	The ordering test between integers.
`greatereq_ builtinInt(@,@)`	13	`(builtinInt builtinInt) bool`	The ordering test between integers.
`btoi(@)`	25	`(bool) builtinInt`	The conversion from a boolean to an integer ($false$ becomes 0 and $true$ becomes 1).
`itob(@)`	26	`(builtinInt) bool`	The conversion from any integer to a boolean (0 becomes $false$ and any other $true$).

5.2.3 Identifiers

/elanlib/noquote/ident.eln

Builtins	Code	Signature	Comments
@ == @	14	`(ident ident) bool`	Tests the equality between two identifiers.
@ != @	15	`(ident ident) bool`	Tests the inequality between two identifiers.

/elanlib/common/builtinString.eln

Builtins	Code	Signature	Comments
`strlen(@)`	150	`(string) builtinInt`	Computes the length of a string.
`strcat(@,@)`	151	`(string string) string`	Concatenates two strings.
`@[@]`	152	`(string builtinInt)`	`s[i]` selects the `i`th character of the string `s`.
`@[@<-@]`	153	`(string builtinInt builtinInt) string`	`s[i<-c]` replaces the `i`th character of the string `s` by the character `c`.
`substr(@,@,@)`	154	`(string builtinInt builtinInt) string`	`substr(s,i,n)` extracts from the string `s` a substring of length `n` from the `i`th position.

Builtins	Code	Signature	Comments
`strspn(@,@)`	156	`(string string) builtinInt`	`strspn(s1,s2)` returns the length of the first part of `s1` entirely consituted by characters of `s2`.
`strcmp(@,@)`	157	`(string string) builtinInt`	`strcmp(s1,s2)` compares two strings character by character and returns an integer (a negative one if `s1` is less than `s2`, `0` if they are equal and otherwise a positive integer).
`string(@)`	158	`(builtinInt) string`	Conversion from an integer (that is, the associated character) to a string.
`ident2string(@)`	177	`(ident) string`	Conversion from an identifier to a string.

5.2.4 Elementary term computations

Since it is of primarily use in symbolic computation on terms (and remember that everything in ELAN is a term except the built-ins), the occurrence relation and the replacement operation are provided as built-ins.

/elanlib/common/occur.eln

The module `occur[X,Y]` is parameterized by two sorts `X,Y` that must be non built-in.

Builtins	Code	Signature	Comments
`occurs @ in @`	17	`(X Y) bool`	`occurs s in t` checks if `s` occurs in `t`.

/elanlib/common/replace.eln

The module `replace[X,Y]` is parameterized by two sorts `X,Y` that must be non built-in.

Builtins	Code	Signature	Comments
`replace @ by @ in @`	16	`(X X Y) Y`	`replace s by u in t` replaces all occurrences of `s` by the `u` in `t`.

/elanlib/common/builtinSyntacticMatching.eln

Matching and unification operations are provided as built-ins.

Builtins	Code	Signature	Comments
builtinSyntactic Matching(@,@,@,@)	191	(X X Y Y) Y	builtinSyntacticMatching (t1,t2,vars,failure) tries to match terms t1 to t2 (i.e. t1 << t2), where vars:Y represents list of variables occurred in t1:X and t2:X and failure:Y is a value returned in a non-successful case. Otherwise, a matching substitution is returned in the form of a list of terms associated to the list of variables vars. The sort Y is supposed to be list[X]. It works only for empty theories.
builtinSyntactic Unification(@,@,@,@)	175	(X X Y Y) Y	the same as the previous built-in function, except that a most general unifier is searched.

5.2.5 Input/Output

/elanlib/common/Query.eln

The module Query[I,O,P] defines the parameterized sort Query[I,O,P] and the sort ide (embedding the sort ident of identifiers). All symbols defined in this module are built-in constructors w.r.t. the classification mentioned before.

Builtins	Code	Signature	Comments
quit	90	() Query[I,O,P]	quit just quit the command interpreter
help	112	() Query[I,O,P]	help prints the help menu (in the file elanlib/help.txt)
load @	91	(ide) Query[I,O,P]	load f loads the file f.eln and extends the current specification
batch @	107	(ide) Query[I,O,P]	batch f runs commands of the batch file f.eln
run @	94	(I) Query[I,O,P]	run q runs the current specification with the query q
sorts @ @ @	92	(ide ide ide) [I,O,P]	sorts q r s redefines sorts of query, results and print-outs
startwith (@)@	93	(ident O) Query[I,O,P]	startwith (s)t redefines the starting term (s)t of the .lgi file, where s is a strategy and t is a pattern of the input query

Builtins	Code	Signature	Comments
`checkwith @`	102	`(bool) Query[I,O,P]`	`checkwith b` redefines the checking term of the `.lgi` file, where `b` is a pattern of a boolean condition tested before any input query
`printwith @`	98	`(P) Query[I,O,P]`	`printwith t` redefines the printing term used to pretty-print any result
`qs`	100	`Query[I,O,P]`	dump the stack of queries
`rs`	101	`Query[I,O,P]`	dump the stack of results
`stat`	96	`Query[I,O,P]`	print statistics
`dump`	95	`Query[I,O,P]`	dump all rules
`dump @`	108	`(ident) Query[I,O,P]`	`dump(l)` dumps rules or strategies with the label/name `l`
`dump @`	109	`(builtinInt) Query[I,O,P]`	`dump(s)` gives all information about the symbol `s` given by its code
`display @`	103	`(builtinInt) Query[I,O,P]`	`display m` changes the printing mode of symbols: `display 1` shows terms in internal form, while `display 0` uses the traditional representation
`trace @`	97	`(builtinInt) Query[I,O,P]`	`trace n` changes the level of debugging to level `n`
`break @`	104	`(ident) Query[I,O,P]`	`break f` sets a break-point on rules or strategies with the label/name `f`
`break @`	110	`(builtinInt) Query[I,O,P]`	`break s` sets a break-point on the symbol `s` given by its code
`unbreak @`	105	`(ident) Query[I,O,P]`	`unbreak f` deletes the break-point on rules or strategies with the label/name `f`
`unbreak @`	111	`(builtinInt) Query[I,O,P]`	`unbreak s` deletes the break-point on the symbol `s`
`breaks`	106	`() Query[I,O,P]`	lists all set break points

/elanlib/common/IO.eln

The module `IO[X]` defines the Input/Output primitives which can only be used with the compiler. This module is parameterized by the sort `X` of the terms used in input/output operations. Currently, only the following sorts are available in the input/output operations: `string`, `int`, `bool`, `term`. For example, to write out or read in strings, the user only need to import `IO[string]` in his or her program.

Builtins	Code	Signature	Comments
`write(@,@)`	-119	`(File X) X`	Writes on output `File` the argument of sort `X`.
`writeln(@,@)`	-119	`(File X) X`	Writes on output `File` the argument of sort `X` and goes to the next line.
`read(@)`	-120	`(File) X`	Reads from input `File` and return an element of sort `X`.
`Error(@)`	123	`(builtinInt) X`	Reports various errors of the I/O sub-system written in C++.

/elanlib/common/builtinStdio.eln

The module `builtinStdio.eln` provides several input/output primitives. But except for `open` and `close`, the others can and should be replaced by the primitives in `IO`. Notice that the module `builtinStdio.eln` is already imported by `IO` and do not need to be explicitly imported in ELAN programs. Three constants of sort `File` `stdin`, `stdout` and `stderr` provide respectively the standard input, the standard output and the standard error output.

Builtins	Code	Signature	Comments
`getc(@)`	113	`(builtinInt) builtinInt`	Gets the next character from an input file or pipe.
`putc(@)`	114	`(builtinInt builtinInt) builtinInt`	Puts a character to an output file or pipe.
`create(@)`	115	`(builtinString) builtinInt`	Creates a process with a given name. The created process is linked with the parent process via two blocking pipes.
`create_noblock(@)`	117	`(builtinString) builtinInt`	Creates a process with a given name such that two communication pipes are in a non blocking mode.
`open(@,@)`	116	`(builtinString builtinString) File`	Opens a file for reading or writing. The first argument is the name of the file, and the second argument is either `"w"` (write or create mode), `"r"` (read mode) or `"a"` (append mode).
`close(@)`	118	`(File) builtinInt`	Closes a file or kill a process denoted by the pipe number of sort `File`.

5.2.6 Strategies

/elanlib/strategy/Meta_strat.eln

The module `Meta_strat[X]` parameterized by a sort `X` can only by used in interpreted mode. This module exports the sort `Strategy[X]`, which specifies an external form of basic strategies of sort `X`. The signature of basic strategies is defined by several built-in constructors used to convert strategy terms of sort `Strategy[X]` into basic strategies over the sort `X`. This conversion is used for the implementation of `meta_apply` symbols of the previous module.

Builtins	Code	Signature	Comments
`@`	131	`(Strategy[X]) Strategies[X]`	Embeds a basic strategy in a list of basic strategies.
`@ , @`	132	`(Strategy[X] Strategies[X]) Strategies[X]`	Concatenates a basic strategy to a list of basic strategies.
`@`	133	`(string) Labels[X]`	Embeds a string in a list of rule labels.
`@ , @`	134	`(string Labels[X]) Labels[X]`	Concatenates a string to a list of rule labels.
`dc(@)`	135	`(Labels[X]) Strateg[X]`	`dc(l)` is a basic strategy that chooses one label in the list of labels `l` and returns all results of the corresponding rule.
`dk(@)`	136	`(Labels[X]) Strateg[X]`	`dk(l)` is a basic strategy that chooses successively all labels in the list of labels `l` and returns all results of the corresponding rules.
`dc(@)`	137	`(Strategies[X]) Strateg[X]`	`dc(ls)` is a basic strategy that chooses one basic strategy in the list `ls` and returns all its results.
`dk(@)`	138	`(Strategies[X]) Strateg[X]`	`dk(ls)` is a basic strategy that chooses successively all basic strategies in the list `ls` and returns all their results.
`repeat*(@)`	139	`(Strategy[X]) Strateg[X]`	`repeat*(s)` is a basic strategy that repeats the application of the basic strategy `s` until it fails.
`iterate*(@)`	140	`(Strategy[X]) Strateg[X]`	`iterate*(s)` is a basic strategy that repeats the application of the basic strategy `s` until it fails and returns intermediate results.
`@`	141	`(Strateg[X]) Strategy[X]`	Embeds the previous constructions into the sort `Strategy[X]`.
`@ ; @`	142	`(Strateg[X] Strategy[X]) Strategy[X]`	Concatenates any previous construction to a strategy.
`id`	143	`() Strateg[X]`	`id` is a basic strategy.
`call @ : X`	144	`(string) Strateg[X]`	`call s:X` represents an application of a basic strategy named `s` of sort `X`.

/elanlib/strategy/Meta_apply.eln

The module `Meta_apply[X]` parameterized by a sort `X` can only by used in interpreted mode (cf. the module `Meta_capply[X]` for the compiled mode).

The construction `where y:=(META)meta_apply(s,t)` exported from this module is semantically equivalent to `where y:=(s)t` where `s` is a basic strategy, i.e. it gives all results of application `s` on `t`. There exists also a sightly modified construction `where y:=(META)meta_apply(s,t,n)` giving only the `n`-th solution of the application of `(s)t`, if it exists.

The ELAN built-in strategy `META` used here just indicates that `(META)meta_apply(s,t)` returns in general several results, which would not be the case with `()meta_apply(s,t)`. The built-in strategy `META` is applicable only to terms of the form `meta_apply(s,t)` or `meta_apply(s,t,n)`.

The module `Meta_apply.eln` exports several variants of the symbol `set_of(s,t,m,n)` returning a list of `m` results of the application `(s)t` from the `n`-th result. The symbols `set_of(s,t)`, `set_of(s,t,m)` and `set_of(s,t,m,n)` are implemented using a result-collecting built-in symbol `meta_apply(s,ts,m,n):(Strategy[X] list[X] builtinInt builtinInt) list[X]` defined in this module.

Builtins	Code	Signature	Comments
`meta_apply(@,@)`	-129	`(Strategy[X] X) X`	`meta_apply(s,t)` returns all results of application of the basic strategy `s` on theterm `t`.
`meta_apply(@,@,@)`	-130	`(Strategy[X] X builtinInt) X`	`meta_apply(s,t,n)` returns the `n`-th result of application of the basic strategy `s` on the term `t`, if it exists.
`meta_apply(@,@,@,@)`	-128	`(Strategy[X] list[X] builtinInt builtinInt) list[X]`	The local built-in symbol `meta_apply(s,t.nil,m,n)` returns `n` results starting from the `m`-th of application of the basic strategy `s` on the term `t`, if they exist.

/elanlib/strategy/Meta_capply.eln

The module `Meta_capply[X]` parameterized by a sort `X` can only by used in compiled mode. It represents a simplified form of the module `Meta_apply[X]`, where a construction of a strategy of sort `Strategy[X]` is restricted to a reference of an existing strategy (i.e. a strategy already occurring in the compiled program). Thus, in a `meta_apply` call, only a name (i.e. a string) of a referenced strategy is used instead of a strategy term of sort `Strategy[X]`.

Builtins	Code	Signature	Comments
`meta_apply(@,@)`	-129	`(string X) X`	`meta_apply(s,t)` returns all results of application of the strategy named `s` on the term `t`.
`meta_apply(@,@,@)`	-130	`(string X builtinInt) X`	`meta_apply(s,t,n)` returns the `n`-th result of application of the strategy named `s` on the term `t`, if it exists.

/elanlib/strategy/strsig.eln

The module `strsig[X,Y]` parameterized by the two sorts `X,Y` defines the signature of the defined strategies. It introduces several built-in constructors used to construct strategy terms of the strategy language. The module `strsig.eln` contains strategy constructors used both for type-preserving and type-changing strategies.

Builtins	Code	Signature	Comments
`[@]@`	-180	`(<X->Y> X) Y`	`[S]t` applies the strategy `S` to the term `t`.
`dk(@)`	-182	`(Strlist[X,Y]) <X->Y>`	`dk(l)` is the defined strategy that chooses successively all strategies in the list `l` and returns all their results.
`dc(@)`	-181	`(Strlist[X,Y]) <X->Y>`	`dc(l)` is the defined strategy that chooses one strategy in the list `l` and returns all its results.
`dc one(@)`	-190	`(Strlist[X,Y]) <X->Y>`	`dc one (l)` is the defined strategy that chooses one strategy in the list `l` and returns one result.
`fail`	-183	`() <X->Y>`	The defined strategy that always gives an empty set of results.
`@,@`	-185	`(<X->Y> Strlist[X,Y]) Strlist[X,Y]`	Concatenates a defined strategy to a list of strategies.
`Epsilon`	-189	`() Strlist[X,Y]`	The empty list of strategies.
`if @ then @ else @ fi`	-184	`(bool <X->Y> <X->Y>) <X->Y>`	`if b then s1 else s2 fi` is the defined strategy that chooses either `s1` if `b` is true otherwise `s2`.
`@`	146	`(Assignment) AssignmentList`	Embeds an assignment in a list of assignments.
`@ , @`	147	`(AssignmentList Assignment) AssignmentList`	Add an assignment to the end of a list of assignments.
`if @`	145	`(bool) IfApplication`	A condition if an IfApplication.
`@`	146	`(IfApplication) AssignmentList`	Embeds an IfApplication in a list of assignments.
`@ , @`	147	`(AssignmentList IfApplication) AssignmentList`	Add an IfApplication to the end of a list of assignments.

/elanlib/strategy/strconc.eln

The module `strconc[X,Y,Z]` parameterized by three sorts `X,Y,Z` introduces a concatenation symbol ; of two strategies of sorts `<X->Y>` and `<Y->Z>`.

Builtins	Code	Signature	Comments
`@ ; @`	-188	`(<X->Y> <Y->Z>) <X->Z>`	`s1 ; s2` concatenates the strategies `s1:<X->Y>` and `s2:<Y->Z>` to get a strategy of sort `<X->Z>`.

/elanlib/strategy/strat.eln

The module `strat[X]` parameterized by a sort `X` has to be imported for working with type-preserving strategies of sort `<X->X>`. It defines the `eval` strategy of the interpreter of the language of defined strategies, and several specific strategy constructors, which are not present in the case of type-changing strategies. The rest of strategy constructors is defined in the module `strategy/strsig.eln`.

Builtins	Code	Signature	Comments
`id`	-186	`() <X->X>`	Identity strategy.
`if @ then @ orelse @ fi`	-187	`(<X->X> <X->X> <X->X>) <X->X>`	Orelse strategy.

5.2.7 REF Format

/elanlib/ref/Meta_Apply.eln

The module `ref/Meta_Apply.eln` provides several variants of the function `meta_apply`, which apply a strategy on a term in an environment defined by a program. These three components are either encoded in the REF-format or passed as strings. Results are produced equally in the REF-format, or as strings representing terms.

Builtins	Code	Signature	Comments
`meta_apply(@,@,@,@,@)`	170	`(string string string string builtinInt) string`	`meta_apply(s,t,file,spec,n)` gives the `n`-th result of the application of the strategy `s` on the term `t` w.r.t. the program stored in files named `file.lgi` and `spec.spc`.
`meta_apply(@,@,@,@,@)`	171	`(string list[string] string string builtinInt) string`	`meta_apply(s,t.nil,file,spec,n)` gives the `n`-th result of the application of the strategy `s` on the term `t` w.r.t. the program in files named `file.lgi` and `spec.spc`.
`meta_apply(@,@,@,@)`	172	`(string string string builtinInt) string`	`meta_apply(s,t,file,n)` gives the `n`-th result of the application of the strategy `s` on the term `t` w.r.t. the program in the REF-format stored in file named `file.ref`.
`meta_apply(@,@,@,@,@)`	173	`(string list[string] string string builtinInt) string`	`meta_apply(s,t.nil,file,n)` gives the `n`-th result of the application of the strategy `s` on the term `t` w.r.t. the program in the REF-format stored in file named `file.ref`.

/elanlib/ref/ref2string.eln

The module `ref/ref2string.eln` provides two conversion functions between strings of REF-terms and terms of the user's defined signature. It also exports a pretty-printing and a parsing function parameterized by a signature in the REF-format.

Builtins	Code	Signature	Comments
refstring2term(@)	-168	(string) X	refstring2term(s) builds a term of sort X from the string s.
term2refstring(@)	-167	(X) string	term2refstring(t) transforms the term t of sort X into a string.
term2string(@)	-169	(X) string	term2string(t) pretty-prints the term t of sort X.

/elanlib/ref/refstring2string.eln

The module ref/refstring2string.eln provides built-in pretty-printing and parsing functions converting strings of terms into strings of REF-terms, and vice-versa. This module defines also two symbols spec2ref and ThisProgram constructing a REF-representation of a program.

Builtins	Code	Signature	Comments
spec2ref(@,@)	162	(string string) string	spec2ref(file1,file2) builds a REF-file from the specification described by two file file1.spc and file2.lgi, and it returns its name file.ref.
ThisProgram	165	() Program	Produces the REF-format of the running program.
refstring2string(@,@)	160	(string string) string	refstring2string(s,file.ref) pretty-prints s according to the grammar defined in the file file.ref.
string2refstring(@,@,@)	161	(string string string) string	string2refstring(st:so,file.ref) parses the string st of a term of sort so according to the grammar defined in the file file.ref.

Chapter 6

Contributed works

6.1 Description of the ELAN contributions

Many computational processes in Automated Deduction and Constraint Programming can be expressed as instances of a general approach that consists of applying transformation rules on formulas with some strategy, until reaching specific normal forms. Such processes are naturally modelled in ELAN, and can be classified according to the area of interest, namely programming, proving and solving.

Programming: One of the first application was to prototype the fundamental mechanisms of logic and functional programming languages like first-order resolution and λ-calculus. The general framework of Constraint Logic Programming can be easily designed in the ELAN framework [KR98], since its operational semantics is clearly formalised as rewrite rules, although the application strategy is often defined in an informal way. Some implementations [Bor95] related to a calculus of explicit substitutions (the first-order rewrite system $\lambda\sigma$ that mimics λ-calculus) open the way of implementing higher-order logic programming languages via a first-order setting. Another calculus of explicit substitutions based on the π-calculus is used to provide a formal specification of Input/Output for ELAN [Vir96]. An extension of ELAN to deal with object oriented features has been designed in [DK00, Dub01]. Modeling chemical reactions is a challenging application of associative-commutative rewrite systems and should be controled by strategies. This is done in ELAN in a system called `Gazel` [BCC+03a, BCC+03b]. XML is now a well established meta-notation which indeed allows to make use of a symbolic term representation. A semantics for this notation can be conveniently defined by rewriting as done (in a quite pecular way) in XSLT. ELAN has been used to describe such elaborated XML transformations [SvdB03].

Proving: ELAN was used in order to implement a predicate prover based on the rules proposed by J.-R. Abrial, and implemented in the B-tools [CK97]. We developed also a propositional sequent calculus, completion procedures for rewrite systems [KM95], sufficient conditions for the termination problem [GG97]. A library for automata construction and manipulation has been designed. Approximation automata are used to check conditions for reachability, sufficient completeness, absence of conflicts in systems described by non-conditional rewrite rules [Gen98b, Gen98a]. Proving termination of rewriting under strategies, and therefore of ELAN program themself is described in [FGK03c, FGK01, FGK02, FGK03d, FGK03b, Fis03]. An implementation called `Cariboo` is itself done in ELAN, see[FGK03a]. The generation of proof terms for normalization derivation and the connection of ELAN to `Coq` to check equational as well as some inductive proof terms has been done in [AN00, NKK02, Ngu02, DKKN03].

Checking: A very special case of proof is just exhaustive check of all the possibilities. Using in particular "dont know" strategies, it was efficiently rediscovered by exhaustive search that the

well known Needam and Schroeder authentification protocol is unsafe [Cir99]. In a similar way, exhaustive search can be used to model-check timed automaton like in [BBKK01].

Solving: The notion of rewriting controlled by strategies is used in [Cas98a, KR98] to describe in a unified way the constraint solving mechanism as well as the meta-language needed to manipulate the constraints [Cas97a, CK98a]. This provides programs that are very close to the proof theoretical setting used now to describe constraint manipulations like unification or numerical constraint solving. ELAN offers a constraint programming environment where the formal description of a constraint solver is directly executable. ELAN has been tested on several examples of constraint solvers for various computation domains and combinations like abstract domains [KR98, Rin97] (term algebras) and more concrete ones (finite domains, integers). In [Cas96a, Cas96b, Cas97b, Cas98c], it is shown how to use computational systems as a general framework for handling Constraint Satisfaction Problems (CSP for short). The approach leads to the design in ELAN of COLETTE, a solver for constraints over integers and finite domains [Cas98b]. We also combine rules, constraints and strategies in order to deal with problems like planning and scheduling [DK99].

6.2 A short annotated bibliography about ELAN

[BKK+96b, BKK97, BKK+98b] General presentations of the different ELAN system releases.

[BKK96a, BK97, Bor98b] A description of the operational as well as denotational semantics of the new ELAN strategies, available since version 2.0. This allows the user to define its own strategies as a computational system.

[BKK98a, BKKR01] Towards a functional operational semantics of ELAN.

[BJMR98] Some useful applications related to the REF format.

[BC98] The facilities provided in ELAN for the cooperation of constraint solvers.

[Bor98a] A full description of the ELAN strategy language. In french.

[Bor95] An ELAN implementation of higher-order unification by using a calculus of explicit substitutions.

[Cas96b, Cas96a, Cas97b] The use of ELAN to specify various constraint manipulation algorithms for CSP's.

[Cas98a, KR98, Cas97b, Cas98b, Cas98c] The rule-based approach to implement Constraint Programming and Constraint Solving Techniques.

[CK98b, CK99a, CK99b, CK01b, CK01a, Cir00] The introduction and presentation of the rewriting calculus that provides a full semantics to ELAN. See also `http://www.loria.fr/~faure/TheRhoCalculusHomePage`.

[CK97] The specification of Abrial's B predicate prover.

[DK98] The strategy language is used to build plans.

[DK99] A combination of rewriting rules, constraints and strategy language for planning problems.

[**KKV95**] The first presentation of the general ideas developed in ELAN, with the definition of computational systems (including the definition of strategies) and the application of ELAN to design and to prototype constraint programming languages.

[**KM96**] An illustration of the reflective power of ELAN and rewriting logic.

[**KM95**] The ELAN implementation of some rule-based completion procedures.

[**Kir97**] An ELAN tutorial.

[**MK97, MK98, Mor99**] A description of the new compilation techniques developed for ELAN, in particular for associativity and commutativity

[**Rin97**] The use of ELAN to interface various unification algorithms and to build combination algorithms for unification.

[**Vir96**] An implementation of input/output in ELAN via an explicit substitution calculus for the π-calculus.

[**Vit96**] A description of the first ELAN compiler.

[**Vit94**] The first main version of the ELAN system in full details, from design to implementation. In French.

6.3 Going further

The work on the design and implementation of the ELAN language and system leaded to new concepts and languages. The main two directions are:

- The rewriting calculus (`http://www.loria.fr/~faure/TheRhoCalculusHomePage`),
- The TOM pattern matching programming language (`http://tom.loria.fr`).

Acknowledgements

We are very grateful to Mark van den Brand whose comments, enlighting questions and always pertinent suggestions allowed us to come up with a much better state of this manual. Many thanks to Carlos Castro, Thomas Genet and Christelle Scharff for many fruitful discussions and constructive criticisms about ELAN, for their careful reading of the manual and checking of examples. Many thanks also to Jounaidi Benhassen for his feedback and to Florent Garnier and Olivier Fissore for their readings of the 2004 version. Any remaining errors are of course entirely ours.

Specials thanks are going to José Meseguer for the many discussions we had on rewriting logic and rewrite based programming languages and to Steven Eker for his participation to a early stage of this project and for always useful interactions.

Many thanks also to all ELAN's users for their feedback on the language and the system.

Bibliography

[AN00] C. Alvarado and Q.-H. Nguyen. ELAN for equational reasoning in Coq. In J. Despeyroux, editor, *Proc. of LFM'00*. INRIA, June 2000. 75

[BBKK01] E. Beffara, O. Bournez, H. Kacem, and C. Kirchner. Verification of timed automata using rewrite rules and strategies. In N. Dershowitz and A. Frank, editors, *Proceedings BISFAI 2001, Seventh Biennial Bar-Ilan International Symposium on the Foundations of Artificial Intelligence*, Ramat-Gan, Israel, June 25–27, 2001. 76

[BC98] P. Borovanský and C. Castro. Cooperation of Constraint Solvers: Using the New Process Control Facilities of ELAN. In C. Kirchner and H. Kirchner, editors, *Proc. Second Intl. Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science. Elsevier, September 1998. 54, 76

[BCC+03a] O. Bournez, G.-M. Côme, V. Conraud, H. Kirchner, and L. Ibanescu. Automated generation of kinetic chemical mechanisms using rewriting. In P. Sloot, D. Abramson, A. Bogdanov, J. Dongarra, A. Zomaya, and Y. Gorbachev, editors, *International Conference on Computational Science - ICCS 2003, Melbourne, June 2-4, 2003, Proceedings, Part III*, volume 2659 of *Lecture Notes in Computer Science*, pages 367–376. Springer, 2003. 75

[BCC+03b] O. Bournez, G.-M. Côme, V. Conraud, H. Kirchner, and L. Ibanescu. A rule-based approach for automated generation of kinetic chemical mechanisms. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings*, volume 2706 of *Lecture Notes in Computer Science*, pages 30–45. Springer, 2003. 75

[BCD+98] P. Borovanský, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen, and M. Vittek. *User Manual of the* ELAN *Standard Library*. LORIA, Nancy (France), first edition, December 1998. 59

[BJMR98] P. Borovanský, S. Jamoussi, P.-E. Moreau, and C. Ringeissen. Handling ELAN Rewrite Programs via an Exchange Format. In C. Kirchner and H. Kirchner, editors, *Proc. Second Intl. Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science, Pont-à-Mousson (France), September 1998. Elsevier. 76

[BK97] P. Borovanský and H. Kirchner. Strategies of ELAN: meta-interpretation and partial evaluation. In *Proceedings of International Workshop on Theory and Practice of Algebraic Specifications ASF+SDF 97*, Workshops in Computing, Amsterdam, September 1997. Springer-Verlag. 76

[BKK96a] P. Borovanský, C. Kirchner, and H. Kirchner. Controlling Rewriting by Rewriting. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar (California), September 1996. Elsevier. 76

[BKK+96b] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar (California), September 1996. Elsevier. 76

[BKK97] P. Borovanský, C. Kirchner, and H. Kirchner. Rewriting as a Unified Specification Tool for Logic and Control:The ELAN Language. In *Proceedings of International Workshop on Theory and Practice of Algebraic Specifications ASF+SDF 97*, Workshops in Computing, Amsterdam, September 1997. Springer-Verlag. 76

[BKK98a] P. Borovanský, C. Kirchner, and H. Kirchner. A functional view of rewriting and strategies for a semantics of ELAN. In M. Sato and Y. Toyama, editors, *The Third Fuji International Symposium on Functional and Logic Programming*, pages 143–167, Kyoto, April 1998. World Scientific. 76

[BKK+98b] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An Overview of ELAN. In C. Kirchner and H. Kirchner, editors, *Proc. Second Intl. Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science, Pont-à-Mousson (France), September 1998. Elsevier. 7, 53, 56, 76

[BKKM02] P. Borovansky, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285(2):155–185, July 2002. 7

[BKKR01] P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: a functional semantics. *International Journal of Foundations of Computer Science*, 12(1):69–98, February 2001. 76

[Bor95] P. Borovanský. Implementation of higher-order unification based on calculus of explicit substitutions. In *Proceedings of SOFSEM'95*, Lecture Notes in Computer Science, pages 363–368. Springer-Verlag, 1995. 75, 76

[Bor98a] P. Borovanský. *Le contrôle de la réécriture : étude et implantation d'un formalisme de stratégies.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy 1, octobre 1998. 76

[Bor98b] P. Borovanský. The Control of Rewriting: Study and Implementation of a Strategy Formalism. In C. Kirchner and H. Kirchner, editors, *Proc. Second Intl. Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science, Pont-à-Mousson (France), September 1998. Elsevier. 76

[Cas96a] C. Castro. Binary CSP Solving as an Inference Process. In *Proceedings of the Eighth International Conference on Tools in Artificial Intelligence, ICTAI'96, Toulouse, France*, pages 462–463, November 1996. 76

[Cas96b] C. Castro. Solving Binary CSP using Computational Systems. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar (California), September 1996. Elsevier. 76

[Cas97a] C. Castro. Constraint Manipulation using Rewrite Rules and Strategies. In A. Drewery, G.-J. M. Kruijff, and R. Zuber, editors, *Proceedings of 2nd ESSLLI Student Session, 9th European Summer School in Logic, Language and Information, ESSLLI'97*, pages 45–56, Aix-en-Provence (France), August 1997. 76

[Cas97b] C. Castro. Constraint Manipulation using Rewrite Rules and Strategies. In A. Drewery, G.-J. M. Kruijff, and R. Zuber, editors, Proceedings of the Second ESSLLI Student Session, 9th European Summer School in Logic, Language and Information, ESSLLI'97, pages 45–56, Aix-en-Provence, France, August 1997. 76

[Cas98a] C. Castro. Building Constraint Satisfaction Problem Solvers Using Rewrite Rules and Strategies. *Fundamenta Informaticae*, 34(3):263–293, September 1998. 76

[Cas98b] C. Castro. COLETTE, Prototyping CSP Solvers Using a Rule-Based Language. In *Proceedings of The Fourth International Conference on Artificial Intelligence and Symbolic Computation, Theory, Implementations and Applications, AISC'98*, volume 1476 of *Lecture Notes in Artificial Intelligence*, Plattsburgh, NY, USA, September 1998. 76

[Cas98c] C. Castro. *Une approche déductive de la résolution de problèmes de satisfaction de contraintes*. Thèse de doctorat d'université, Université Henri Poincaré – Nancy 1, 1998. Also available as Technical Report 98-T-315, LORIA, Nancy (France). 76

[Cir99] H. Cirstea. Specifying authentication protocols using ELAN. In *Workshop on Modelling and Verification*, Besancon, France, December 1999. 76

[Cir00] H. Cirstea. *Calcul de réécriture : fondements et applications*. Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy I, 2000. 76

[CK97] H. Cirstea and C. Kirchner. Theorem proving Using Computational Systems: The case of the B Predicate Prover. Available at `http://www.loria.fr/~cirstea/Papers/TheoremProver.ps`, 1997. 75, 76

[CK98a] C. Castro and C. Kirchner. Constraint Rewriting. In *Proceedings of The Workshop on Applications of Rewriting, 9th International Conference on Rewriting Techniques and Applications, RTA'98, Tsukuba, Japan*, March 1998. 76

[CK98b] H. Cirstea and C. Kirchner. The Rwriting Calculus as a Semantics of ELAN. In J. Hsiang and A. Ohori, editors, *4th Asian Computing Science Conference*, volume 1538 of *Lecture Notes in Computer Science*, Manila, The Philippines, December 1998. Springer-Verlag. 76

[CK99a] H. Cirstea and C. Kirchner. Combining higher-order and first-order computation using ρ-calculus: Towards a semantics of ELAN. In D. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, Research Studies, ISBN 0863802524, pages 95–120. Wiley, 1999. 76

[CK99b] H. Cirstea and C. Kirchner. An introduction to the rewriting calculus. Research Report RR-3818, INRIA, December 1999. 76

[CK01a] H. Cirstea and C. Kirchner. Rewriting and Multisets in Rho-calculus and ELAN. *Romanian Journal of Information, Science and Technology*, 4(1-2):33–48, 2001. ISSN: 1453-8245. 76

[CK01b] H. Cirstea and C. Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, May 2001. 76

[DK98] H. Dubois and H. Kirchner. Actions and plans in ELAN. In *Proceedings of the Workshop on Strategies in Automated Deduction - CADE-15, Lindau, Germany*, pages 35–45, 1998. 76

[DK99] H. Dubois and H. Kirchner. Rule based programming with constraints and strategies. Technical Report 99-R-084, LORIA, Nancy, France, November 1999. ERCIM workshop on Constraints, Paphos (Cyprus). 76

[DK00] H. Dubois and H. Kirchner. Rule Based Programming with Constraints and Strategies. In K. Apt, A. A. C. Kakas, E. Monfroy, and F. Rossi, editors, *New Trends in Constraints, Papers from the Joint ERCIM/Compulog-Net Workshop, Cyprus, October 25-27, 1999*, volume 1865 of *Lecture Notes in Artificial Intelligence*, pages 274–297. Springer-Verlag, 2000. 75

[DKKN03] E. Deplagne, C. Kirchner, H. Kirchner, and Q.-H. Nguyen. Proof search and proof check for equational and inductive theorems. In *Proc. of 19th CADE*, Lecture Notes in Artificial Intelligence, pages 297–316. Springer-Verlag, 2003. 75

[Dub01] H. Dubois. *Systèmes de Règles de Production et Calcul de Réécriture*. Thèse de doctorat d'université, Université Henri Poincaré – Nancy 1, septembre 2001. Also available as Technical Report A01-T-200, LORIA, Nancy (France). 75

[Ear70] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970. 34, 53

[Eke95] S. Eker. Associative-commutative matching via bipartite graph matching. *Computer Journal*, 38(5):381–399, 1995. 54

[FGK01] O. Fissore, I. Gnaedig, and H. Kirchner. Termination of rewriting with local strategies. In M. P. Bonacina and B. Gramlich, editors, *Electronic Notes in Theoretical Computer Science*, volume 58. Elsevier Science Publishers, 2001. Also available as Technical Report A01-R-177, LORIA, Nancy (France). 75

[FGK02] O. Fissore, I. Gnaedig, and H. Kirchner. Outermost ground termination. *Electronic Notes in Theoretical Computer Science*, 71(A02-R-554), Sep 2002. 75

[FGK03a] O. Fissore, I. Gnaedig, and H. Kirchner. CARIBOO: A multi-strategy termination proof tool based on induction. In *6th International Workshop on Termination 2003 - WST'03, Valencia, Spain*, pages 77–79. Albert Rubio, Jun 2003. 75

[FGK03b] O. Fissore, I. Gnaedig, and H. Kirchner. Proving weak termination also provides the right way to terminate - extended version -. Rapport technique A03-R-361, LORIA, Nancy, France, Oct 2003. 75

[FGK03c] O. Fissore, I. Gnaedig, and H. Kirchner. Simplification and termination of strategies in rule-based languages. In *Proceedings of the Fifth International Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 124–135, Uppsala, Sweden, August 2003. ACM Press. 75

[FGK03d] O. Fissore, I. Gnaedig, and H. Kirchner. Termination of elan strategies by simplification - extended version -. Rapport technique A03-R-360, LORIA, Nancy, France, Aug 2003. 75

[Fis03] O. Fissore. *Terminaison de la réécriture sous stratégies.* Thèse d'université, UHP Nancy I, Dec 2003. 75

[Gen98a] T. Genet. *Contraintes d'ordre et automates d'arbres pour les preuves de terminaison.* Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, 1998. 75

[Gen98b] T. Genet. Decidable Approximations of Sets of Descendants and Sets of Normal Forms. In *Proceedings 9th Conference on Rewriting Techniques and Applications, Tsukuba (Japan)*, volume 1379 of *Lecture Notes in Computer Science*, pages 151–165. Springer-Verlag, 1998. 75

[GG97] T. Genet and I. Gnaedig. Termination Proofs using gpo Ordering Constraints. In M. Bidoit and M. Dauchet, editors, *Proceedings 22nd International Colloquium on Trees in Algebra and Programming, Lille (France)*, volume 1214 of *Lecture Notes in Computer Science*, pages 249–260. Springer-Verlag, 1997. 75

[GKK+87] J. A. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ-3. In J.-P. Jouannaud and S. Kaplan, editors, *Proceedings 1st International Workshop on Conditional Term Rewriting Systems, Orsay (France)*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, July 1987. Also as internal report CRIN: 88-R-001. 7

[HHKR89] J. Heering, P. Hendriks, P. Klint, and J. Rekers. The syntax definition formalisme SDF - reference manual. *ACM SIGPLAN Notices*, 24(11):43–75, 1989. 7

[Kir97] H. Kirchner. ELAN (tutoriel invité). In F. Fages, editor, *JFPLC'99 Journées Francophones de Programmation Logique et programmation par Contraintes*, pages 241–248. Hermes Science Publications, 1997. Report LORIA 99-R-129. 77

[KKV95] C. Kirchner, H. Kirchner, and M. Vittek. Designing Constraint Logic Programming Languages using Computational Systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, pages 131–158. MIT press, 1995. 7, 37, 77

[KM95] H. Kirchner and P.-E. Moreau. Prototyping completion with constraints using computational systems. In J. Hsiang, editor, *Proceedings 6th Conference on Rewriting Techniques and Applications, Kaiserslautern (Germany)*, volume 914 of *Lecture Notes in Computer Science*, pages 438–443. Springer-Verlag, 1995. 75, 77

[KM96] H. Kirchner and P.-E. Moreau. A reflective extension of ELAN. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar (California), September 1996. Elsevier. 77

[KR98] C. Kirchner and C. Ringeissen. Rule-Based Constraint Programming. *Fundamenta Informaticae*, 34(3):225–262, September 1998. 75, 76

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992. 7

[MK97] P.-E. Moreau and H. Kirchner. Compilation Techniques for Associative-Commutative Normalisation. In *Proceedings of International Workshop on Theory and Practice of Algebraic Specifications ASF+SDF 97*, Workshops in Computing, Amsterdam, September 1997. Springer-Verlag. 77

[MK98] P. Moreau and H. Kirchner. A compiler for rewrite programs in associative-commutative theories. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Principles of Declarative Programming*, number 1490 in Lecture Notes in Computer Science, pages 230–249. Springer-Verlag, September 1998. 23, 54, 77

[Mor98] P.-E. Moreau. A choice-point library for backtrack programming. JICSLP'98 Post-Conference Workshop on Implementation Technologies for Programming Languages based on Logic, June 1998. 23, 54

[Mor99] P.-E. Moreau. *Compilation de règles de réécriture et de stratégies non-deterministes*. Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, France, June 1999. also TR LORIA 98-T-326. 77

[Ngu] Q.-H. Nguyen. ELAN for AC rewriting in Coq. `http://www.loria.fr/~nguyenqh/coqelan/ACrew.html`. 41

[Ngu02] Q.-H. Nguyen. *Calcul de rcriture et automatisation du raisonnement dans les assistants de preuve*. Thèse de Doctorat d'Université, Université Henri Poincar - Nancy 1, Octobre 2002. 75

[NKK02] Q.-H. Nguyen, C. Kirchner, and H. Kirchner. External rewriting for skeptical proof assistants. *Journal of Automated Reasoning*, 29(3-4):309–336, 2002. 75

[Rin97] C. Ringeissen. Prototyping Combination of Unification Algorithms with the ELAN Rule-Based Programming Language. In *Proceedings 8th Conference on Rewriting Techniques and Applications, Sitges (Spain)*, volume 1232 of *Lecture Notes in Computer Science*, pages 323–326. Springer-Verlag, 1997. 76, 77

[SvdB03] J. Stuber and M. van den Brand. Extracting mathematical semantics from latex documents. In *Workshop on Principles and Practice of Semantic Web Reasoning - PPSWR'2003, Mumbai, India*, Dec 2003. 75

[Vir96] P. Viry. Input/Output for ELAN. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar (California), September 1996. Elsevier. 75, 77

[Vit94] M. Vittek. *ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes*. Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy 1, octobre 1994. 7, 37, 77

[Vit96] M. Vittek. A Compiler for Nondeterministic Term Rewriting Systems. In H. Ganzinger, editor, *Proceedings 7th Conference on Rewriting Techniques and Applications, New Brunswick (New Jersey, USA)*, volume 1103 of *Lecture Notes in Computer Science*, pages 154–168. Springer-Verlag, July 1996. 23, 54, 77