The C*i*ME system
Version 2.02

Evelyne Contejean, Claude Marché, Benjamin Monate, and Xavier Urbain
LRI, CNRS UMR 8623
Bât. 490
Université Paris-Sud, Centre d'Orsay
91405 Orsay Cedex, France
Phone:+33 1 69 15 64 85
Fax: +33 1 69 41 65 86
Email: `{contejea,marche,monate,urbain}@lri.fr`

This research was supported in part by the EU project CCL, the HCM Network CONSOLE, and the "GDR de programmation du CNRS".

Contents

Part I

User manual

Chapter 1

Tutorial

This chapter is a small tutorial, to present the basic features in a progressive way.

1.1 The toplevel language

1.1.1 Expressions

The toplevel language is a simple language to allow the user to type-in expressions terminated by a semi-colon, and let CiME evaluate them. Simple expressions are made with integers, booleans and strings, and operations on them:

```
CiME> 12;
- : int = 12
CiME> "hello world";
- : string = "hello world"
CiME> true;
- : bool = true
CiME> 1+2;
- : int = 3
CiME> 3*4-1;
- : int = 11
CiME> -4*(3-2)+1;
- : int = -3
CiME> 4*4 < 5 or 6 <> 3+1;
- : bool = true
CiME> true and not false or true;
- : bool = true
```

Notice that for each expression, CiME first compute its type. In case of typing error, an error message is displayed and no evaluation is performed:

```
CiME> 1+true;
Typing error: bad type argument in application
```

1.1.2 Definitions with `let`

It is possible to give a name to the result of an evaluation, for further use in next expressions:

```
CiME> let x = 4*5;
x : int = 20
CiME> x*x-1;
- : int = 399
```

```
CiME> x > 10;
- : bool = true
```

An attempt to use an undefined name results in a typing error:

```
CiME> y+1;
Typing error: undefined identifier y
```

Notice that it is allowed to redefine a name:

```
CiME> let x=1;
x : int = 1
CiME> x+3;
- : int = 4
CiME> let x=2;
x : int = 2
CiME> x+3;
- : int = 5
```

notice that the second `let` does not behave like an assignment like in traditional imperative programming languages, but it is a new definition hiding to previous one, like in functional languages. (More precisely, it follows the static binding semantics, see Section 2.1.2.)

1.1.3 Definitions of functions

The core language allows the definition of functions, with a `let fun` construct. Application is denoted by a juxtaposition of the function and its arguments, as in LISP or other functional languages based on lambda-calculus.

```
CiME> let fun succ x = x+1;
succ : int -> int = <fun>
CiME> (succ 4);
- : int = 5
CiME> (succ (succ 6));
- : int = 8
```

Notice that putting parentheses around an application is recommended, by not mandatory:

```
CiME> succ 4;
- : int = 5
CiME> succ (succ 6);
- : int = 8
```

Functions with several arguments are defined analogously:

```
CiME> let fun norm x y = x*x+y*y;
norm : int -> int -> int = <fun>
CiME> norm 3 4;
- : int = 25
```

Functions may be defined recursively:

```
CiME> let fun fact n = if n <= 1 then 1 else n * (fact (n-1));
fact : int -> int = <fun>
CiME> fact 7;
- : int = 5040
CiME> fact 100;
- : int =
933262154439441526816992388562667004907159682643816214685929638952175 9
999322991560894146397615651828625369792082722375825118521091686400000 0
00000000000000000
```

Notice on that last example that integers may have arbitrary size.

1.1.4 Higher-order functions

The core language is fully higher-order polymorphic functional, so that you can use partial application, functions as arguments, polymorphic arguments:

```
CiME> let fun f x y = x*x+y*y;
f : int -> int -> int = <fun>
CiME> let g = f 2;
g : int -> int = <fun>
CiME> g 5;
- : int = 29
CiME> let fun eval_at_2 f = f 2;
eval_at_2 : (int -> 'a) -> 'a = <fun>
CiME> eval_at_2 g;
- : int = 8
CiME> eval_at_2 (fun x -> x+1);
- : int = 3
CiME> let fun compose f g x = f (g x);
compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

1.2 More toplevel user interaction

1.2.1 Loading expressions from files

For easy use, it is possible to write expressions in a file and tell CiME to evaluate them as if they were typed directly. It is done by the directive `#load` followed by the file name.

Suppose the file `test.cim` contains

```
let fun fib n = if n <= 1 then 1 else (fib (n-1)) + (fib (n-2));
fib 3;
fib 4;
fib 15;
fib 25;
```

then one may do

```
CiME> #load "test.cim";
fib : int -> int = <fun>
- : int = 3
- : int = 5
- : int = 987
- : int = 121393
```

or alternatively, you may run cime directly with file names as arguments: the toplevel will first load and execute the files, and then let the user type more expressions:

```
unix prompt> cime2 test.cim
fib : int -> int = <fun>
- : int = 3
- : int = 5
- : int = 987
- : int = 121393
CiME>
```

1.2.2 Interrupting evaluation

If you try to performed a evaluation that takes a large amount of time, and then want to stop the evaluation, you may type Control-C and you return back to the toplevel loop:

```
CiME> fib 100;
^CCommand aborted.
CiME>
```

1.2.3 Time spent in computation

You may display the CPU time spent in evaluation of each expression by typing `#time on;`:

```
CiME> #time on;
time is now on
CiME> fib 15;
Execution time: 0.2 sec.
- : int = 987
```

To turn off this feature, type `#time off;`. `#time;` alone toggles between `on` and `off`.

1.2.4 Quitting CiME

You may quit CiME by typing `#quit;`, or simply hitting Control-D.

1.3 Playing with the Diophantine constraints solver

CiME provides a general Diophantine constraints solver. Such constraints are equalities or inequalities between polynomials over an arbitrary number of variables, and a CiME predefined function may be used to try to solve such constraints for variables belonging to an interval $[0,\cdots,N]$ for a given N.

The following example shows how to solve the system

$$\begin{aligned} xy &= 6 \\ x+y &= 5 \end{aligned}$$

you must first define your constraints, and then call the solver, for a given bound N, say 100 here:

```
CiME> let constr = dioph_constraint "x.y = 6; x+y = 5" ;
constr : dioph_constraint = { 1*x*y + -6 = 0;
                              1*y + 1*x + -5 = 0;
                              }
(2 inequalitie(s) over 2 variable(s).)

CiME> dioph_solve 100 constr;
Solution :
x = 3
y = 2
- : unit = ()
```

Notice first the constraints displayed is not the same as the input, since C*i*ME performed some internal transformations.

One may also give inequalities:

```
CiME> let constr = dioph_constraint
  "x^2 + y^2 = z^2; x > 0; y > 0; z > 0";
constr : dioph_constraint = { -1*z^2 + 1*y^2 + 1*x^2 = 0;
                              1*x + -1 >= 0;
```

```
                              1*y + -1 >= 0;
                              1*z + -1 >= 0;
                              }
(4 inequalitie(s) over 3 variable(s).)

CiME> dioph_solve 100 constr;
Solution :
x = 4
y = 3
z = 5
- : unit = ()
```

Notice that only the first solution found is displayed, even if there are others.

Here is a quite larger example, the famous problem SEND+MORE=MONEY. Notice that no negation is allowed, so in this problem the fact that two variables x and y must be distinct is encoded as $(x-y)^2 > 0$. This example is found in the bunch of examples available:

```
CiME> #load "examples/diophantine_solver/send+more=money.cim2";
c : dioph_constraint = { -10*r1 + -1*y + 1*e + 1*d = 0;
                         -10*r2 + 1*r + 1*n + 1*r1 + -1*e = 0;
                         -10*r3 + 1*o + 1*r2 + -1*n + 1*e = 0;
                         -9*m + 1*s + 1*r3 + -1*o = 0;
                         1*s + -1 >= 0;
                         1*m + -1 >= 0;
                         1*s^2 + -2*e*s + 1*e^2 + -1 >= 0;
                         1*s^2 + -2*n*s + 1*n^2 + -1 >= 0;
                         1*s^2 + -2*d*s + 1*d^2 + -1 >= 0;
                         1*m^2 + -2*s*m + 1*s^2 + -1 >= 0;
                         1*s^2 + -2*o*s + 1*o^2 + -1 >= 0;
                         1*s^2 + -2*r*s + 1*r^2 + -1 >= 0;
                         1*s^2 + -2*y*s + 1*y^2 + -1 >= 0;
                         1*n^2 + -2*e*n + 1*e^2 + -1 >= 0;
                         1*e^2 + -2*d*e + 1*d^2 + -1 >= 0;
                         1*m^2 + -2*e*m + 1*e^2 + -1 >= 0;
                         1*o^2 + -2*e*o + 1*e^2 + -1 >= 0;
                         1*r^2 + -2*e*r + 1*e^2 + -1 >= 0;
                         1*y^2 + -2*e*y + 1*e^2 + -1 >= 0;
                         1*n^2 + -2*d*n + 1*d^2 + -1 >= 0;
                         1*m^2 + -2*n*m + 1*n^2 + -1 >= 0;
                         1*o^2 + -2*n*o + 1*n^2 + -1 >= 0;
                         1*r^2 + -2*n*r + 1*n^2 + -1 >= 0;
                         1*n^2 + -2*y*n + 1*y^2 + -1 >= 0;
                         1*m^2 + -2*d*m + 1*d^2 + -1 >= 0;
                         1*o^2 + -2*d*o + 1*d^2 + -1 >= 0;
                         1*r^2 + -2*d*r + 1*d^2 + -1 >= 0;
                         1*y^2 + -2*d*y + 1*d^2 + -1 >= 0;
                         1*m^2 + -2*o*m + 1*o^2 + -1 >= 0;
                         1*m^2 + -2*r*m + 1*r^2 + -1 >= 0;
                         1*m^2 + -2*y*m + 1*y^2 + -1 >= 0;
                         1*o^2 + -2*r*o + 1*r^2 + -1 >= 0;
                         1*o^2 + -2*y*o + 1*y^2 + -1 >= 0;
                         1*r^2 + -2*y*r + 1*y^2 + -1 >= 0;
                         }
(34 inequalitie(s) over 11 variable(s).)
```

```
time is now on
Solution :
d = 7
e = 5
y = 2
r1 = 1
n = 6
r = 8
r2 = 1
o = 0
r3 = 0
s = 9
m = 1
Execution time: 0.49 sec.
- : unit = ()
```

1.4 String rewriting

TODO

1.5 Parameterized string rewriting system

TODO

1.6 Definition of first-order signatures

```
CiME> let F_peano = signature "
0 : constant;
s : unary;
+,* : infix binary;
";
F_peano : signature = <signature>

CiME> let X = vars "x y z";
X : variable_set = <variable set>
```

1.7 Definition of terms

Once you have defined a signature and a set of variables, you may define terms, using the predefined function `term` which takes as arguments a signature, a set of variables, and a string denoting a term. Example:

```
CiME> let t = term F_peano X "s(s(s(0)))*(s(0)+s(s(0)))";
t : (F_peano,X) term = s(s(s(0))) * (s(0) + s(s(0)))
CiME> let u = term F_peano X "s(x)+0+x*s(0)";
u : (F_peano,X) term = ((s(x) + 0) + x) * s(0)
```

Beware! As you may see on this last example, C*i*ME does not make any assumptions on the priorities between binary operators of your defined signature. By default, it always associate to the left. To enforce a priority, you have to use parentheses, like below.

```
CiME> let u = term F_peano X "s(x)+0+(x*s(0))";
u : (F_peano,X) term = (s(x) + 0) + (x * s(0))
```

1.8 Rewriting

One may define a rewrite system with the ternary function `TRS`, which takes a signature as first argument, and a set of variables as second argument, a finally a string that contains a natural definition of rules :

```
CiME> let R_peano = TRS F_peano X "
x+0 -> x;
x+s(y) -> s(x+y);
x*0 -> 0;
x*s(y) -> (x*y)+x;
";
R_peano : (F_peano,X) TRS = { x + 0 -> x,
                              x + s(y) -> s(x + y),
                              x * 0 -> 0,
                              x * s(y) -> (x * y) + x } (4 rules)
```

Given a term and a TRS, you can normalize the term with the TRS, by use of the predefined function `normalize`:

```
CiME> normalize R_peano t;
- : (F_peano,X) term = s(s(s(s(s(s(s(s(0))))))))
CiME> normalize R_peano u;
- : (F_peano,X) term = s(x) + (0 + x)
```

1.9 Checking termination of rewrite systems

```
CiME> termination R_peano;
Entering the termination expert. Verbose level = 0
[0] = 1;
[s](X0) = 1*X0 + 1;
[+](X0,X1) = 2*X1 + 1*X0;
[*](X0,X1) = 2*X0*X1 + 2*X1 + 1*X0;
- : unit = ()
```

1.9.1 Search for incremental and modular termination proof

Termination proofs may be found in a much more efficient way using hierarchies of rewriting modules (HTRS) instead of TRS.

Hierarchical TRS

A hierarchical term rewriting system `R` is defined as follow:

`let R = HTRS {R0;...;Ri;...;Rk} Sig X "` *list of rules* `";`

where `Ri` are HTRS, `Sig` consists of some new symbols (possibly AC) and `X` is a set of variables. The given *list of rules* define some symbols in `Sig` such that the union of rules of `R` and all `Ri` is a TRS over the union of all `Sig`.

All symbols defined in *list of rules* must be in signature `Sig`.

Semantics of such a declaration is that `R` is an hierarchical extension of `Ri` with new symbols of `Sig` and rules from *list of rules*.

An extension of the empty HTRS is denoted:

`let R = HTRS {} Sig X "` *list of rules* `";`

Proving termination with HTRS

Termination proofs of HTRS are performed incrementally/modularly on modules constituting the extension. In particular $\mathcal{C}_{\mathcal{E}}$-termination of a `Ri` is proven only once, and never again in a termination proof of any extension of `Ri`.

The function for searching an incremental/modular termination proof of HTRS `R` is:

```
h_termination : ('A,'B) HTRS -> unit
```

it tries to find a termination proof of the rewrite system given as argument by providing for each sub-hierarchy (w.r.t. topological sorting) a suitable ordering.

The result of that search is displayed on standard output.

Note that, by definition, `h_termination` always makes use of dependency pairs of modules, even if `termcrit "standard";` was selected.

Optimizations such as `marks` and `graph` may be used as they affect both C*i*ME incremental/modular and classical termination experts.

In order to make the termination expert more efficient, it may prove useful to restrict to nonexpensive criteria on most of the proof. Such tuning may be done with help of function:

```
h_termination_with : ((string,int) set) -> ('A,'B) HTRS -> unit
```

which takes as first argument a list of pairs *criterion, bound*. They denote polynomials (as with `polyinterpkind`) and bounds (as with `termpolybound`) to be tried successively on a module when a search fails.

Finally, it is possible to check if a given `Ri` has been proven terminating with:

```
check_ce : ('A,'B) HTRS -> unit
```

which prints the result on standard output.

Splitting HTRS

A HTRS may be considered as a hierarchy of minimal modules by use of:

```
termcrit "minimal";
```

HTRS are then split up in minimal modules for termination proofs.

This is particularly useful when dealing with a huge bunch of rules in a single HTRS (see examples below).

1.9.2 Examples of modular expert sessions

Step by step

```
(* Binary arithmetics:
   Addition
   Multiplication
   Lists
*)

CiME> let X = vars "x y z l b";


(* Nonnegative rational integer in binary notation *)

CiME> let Fbin = signature "
    #   : constant ;
    0,1 : postfix unary ;
   ";
```

```
CiME> let Mbin = HTRS {} Fbin X "
  (#)0 -> #;
  ";


CiME> h_termination Mbin;

Entering the termination expert for modules. Verbose level = 1
checking each of the 0 strongly connected components :
Termination proof found.
Execution time: 0.000000 sec
- : unit = ()
CiME>

(* Addition *)

CiME> let Fadd = signature "
   + : infix binary
";

CiME> let Madd = HTRS {Mbin} Fadd X "
     # + x -> x;
     x + # -> x;
     (x)0 + (y)0 -> (x+y)0;
     (x)0 + (y)1 -> (x+y)1;
     (x)1 + (y)0 -> (x+y)1;
     (x)1 + (y)1 -> (x+y+(#)1)0;
";

CiME> h_termination Madd;

Entering the termination expert for modules. Verbose level = 0
CE-termination of
{ (#)0 -> # } (1 rules)
is already proven.

checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[#] = 1;
[+](X0,X1) = X1*X0;
[0](X0) = X0 + 1;
[1](X0) = 2*X0 + 1;
['+`](X0,X1) = X1*X0;

Termination proof found.
Execution time: 0.170000 sec
- : unit = ()
CiME>

(* Multiplication *)

CiME> let Fmult = signature "
  * : infix binary
";
```

```
CiME> let Mmult = HTRS {Madd} Fmult X "
     # * x -> #;
     x * # -> #;
     (x)0 * y -> (x*y)0;
     (x)1 * y -> (x*y)0 + y;
";

CiME> h_termination Mmult;

Entering the termination expert for modules. Verbose level = 0
CE-termination of
{ (#)0 -> #,
  # + V_0 -> V_0,
  V_0 + # -> V_0,
  (V_0)0 + (V_1)0 -> (V_0 + V_1)0,
  (V_0)0 + (V_1)1 -> (V_0 + V_1)1,
  (V_0)1 + (V_1)0 -> (V_0 + V_1)1,
  (V_0)1 + (V_1)1 -> ((V_0 + V_1) + (#)1)0 } (7 rules)
is already proven.

checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[#] = 0;
[+](X0,X1) = X1 + X0;
[*](X0,X1) = X1*X0 + X0;
[0](X0) = X0 + 1;
[1](X0) = X0 + 1;
['*`](X0,X1) = X0;

Termination proof found.
Execution time: 0.080000 sec
- : unit = ()
CiME>

(* Lists *)
CiME> let Flist = signature "
  nil : constant ;
  cons : binary";

CiME> let Mlist = HTRS {} Flist X "";


(* Sum in a liste *)
CiME> let Fsum = signature " sum : unary ";

CiME> let Msum = HTRS {Mlist ; Madd}
                Fsum X "
  sum(nil) -> (#)0;
  sum(cons(x,l)) -> x + sum(l);
";
```

Suitable interpretation are often simpler when using the incremental/modular expert. It is often judicious to search firstly for linear interpretations.

```
CiME> polyinterpkind "linear";

Now searching only linear polynomial interpretations
Execution time: 0.000000 sec
- : unit = ()

CiME> h_termination Msum;

Entering the termination expert for modules. Verbose level = 0
checking each of the 0 strongly connected components :
Termination proof found.

CE-termination of
{ (#)0 -> #,
  # + V_0 -> V_0,
  V_0 + # -> V_0,
  (V_0)0 + (V_1)0 -> (V_0 + V_1)0,
  (V_0)0 + (V_1)1 -> (V_0 + V_1)1,
  (V_0)1 + (V_1)0 -> (V_0 + V_1)1,
  (V_0)1 + (V_1)1 -> ((V_0 + V_1) + (#)1)0 } (7 rules)
is already proven.

checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[#] = 0;
[+](X0,X1) = X1 + X0;
[nil] = 0;
[sum](X0) = X0;
[0](X0) = 0;
[cons](X0,X1) = X1 + X0 + 1;
[1](X0) = 0;
['sum`](X0) = X0;

Termination proof found.
Execution time: 0.050000 sec
- : unit = ()
CiME>
```

The incremental/modular expert of C*i*ME is able to prove termination even if AC symbols are involved.

```
(* Addition, Multiplication
   and Multisets
*)

(* Multisets with an AC union operator *)
CiME> let Fbag = signature "
  empty : constant ;
  singl : unary ;
  U : AC";

CiME> let Mbag = HTRS {} Fbag X "
  empty U b -> b;
";

CiME> h_termination Mbag;
```

```
Entering the termination expert for modules. Verbose level = 0
checking each of the 0 strongly connected components :
Termination proof found.
Execution time: 0.000000 sec
- : unit = ()
CiME>

(* Sum in a multiset *)

CiME> let Msumbag = HTRS {Mbag ; Madd}
                   Fsum X
"
  sum(empty) -> (#)0;
  sum(singl(x)) -> x;
  sum(x U y) -> sum(x) + sum(y);
";

CiME> h_termination Msumbag;

Entering the termination expert for modules. Verbose level = 0
CE-termination of
{ empty U V_4 -> V_4 } (1 rules)
is already proven.

CE-termination of
{ (#)0 -> #,
  # + V_0 -> V_0,
  V_0 + # -> V_0,
  (V_0)0 + (V_1)0 -> (V_0 + V_1)0,
  (V_0)0 + (V_1)1 -> (V_0 + V_1)1,
  (V_0)1 + (V_1)0 -> (V_0 + V_1)1,
  (V_0)1 + (V_1)1 -> ((V_0 + V_1) + (#)1)0 } (7 rules)
is already proven.

checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[#] = 0;
[+](X0,X1) = X1 + X0;
[sum](X0) = X0;
[empty] = 0;
[0](X0) = 0;
[singl](X0) = X0;
[1](X0) = 0;
[U](X0,X1) = X1 + X0 + 1;
['sum`](X0) = X0;

Termination proof found.
Execution time: 0.050000 sec
- : unit = ()
CiME>
```

Proof on a hierarchy

A termination proof may be search for when all the relevant hierarchy has been defined.

```
CiME> let X = vars "x y z l b";

CiME> let Fbin = signature "
    #   : constant ;
    0,1 : postfix unary ;
   ";

CiME> let Mbin = HTRS {} Fbin X "
  (#)0 -> #;
  ";

CiME> let Fadd = signature "
  + : infix binary";

CiME> let Madd = HTRS {Mbin} Fadd X "
     # + x -> x;
     x + # -> x;
     (x)0 + (y)0 -> (x+y)0;
     (x)0 + (y)1 -> (x+y)1;
     (x)1 + (y)0 -> (x+y)1;
     (x)1 + (y)1 -> (x+y+(#)1)0;
";

CiME> let Fmult = signature "
  * : infix binary";

CiME> let Mmult = HTRS {Madd} Fmult X "
     # * x -> #;
     x * # -> #;
     (x)0 * y -> (x*y)0;
     (x)1 * y -> (x*y)0 + y;
";

CiME> let Flist = signature "
  nil : constant ;
  cons : binary";

CiME> let Mlist = HTRS {} Flist X "";

CiME> let Fsum = signature " sum : unary ";

CiME> let Msum = HTRS {Mlist ; Madd} Fsum X
"
  sum(nil) -> (#)0;
  sum(cons(x,l)) -> x + sum(l);
";

CiME> let Fbag = signature "
  empty : constant ;
  singl : unary ;
  U : AC";
```

```
CiME> let Mbag = HTRS {} Fbag X "
  empty U b -> b;
";

CiME> let Msumbag = HTRS {Mbag ; Madd}
                   Fsum X "
  sum(empty) -> (#)0;
  sum(singl(x)) -> x;
  sum(x U y) -> sum(x) + sum(y);
";

CiME> h_termination Msumbag;

Entering the termination expert for modules. Verbose level = 0
checking each of the 0 strongly connected components :
Termination proof found.

checking each of the 0 strongly connected components :
Termination proof found.

checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[#] = 1;
[+](X0,X1) = X1*X0;
[0](X0) = X0 + 1;
[1](X0) = 2*X0 + 1;
['+'](X0,X1) = X1*X0;

Termination proof found.

checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[#] = 1;
[+](X0,X1) = X1*X0;
[sum](X0) = X0;
[empty] = 1;
[0](X0) = X0;
[singl](X0) = X0;
[1](X0) = 0;
[U](X0,X1) = 2*X1*X0 + 2*X1 + 2*X0 + 1;
['sum'](X0) = X0;

Termination proof found.
Execution time: 0.240000 sec
- : unit = ()
CiME>
```

Note that nothing is done about the irrelevant multiplication.

A termination proof search regarding the hierarchy headed by `Mmult` does not need to be performed using simple polynomials everywhere since linear polynomials suffice for `Madd`. In order to avoid overheads one may use `h_termination_with`. For instance, termination of binary arithmetics (`Mbin`, `Madd` and `Mmult`) may be proven with:

```
CiME> h_termination_with {("linear",1);("simple",1)} Mmult;
```

Splitting up a hierarchy in minimal modules

Finally one can provide a single set of rules and ask C*i*ME to perform a minimal splitting up of it. For instance for sum in multiset:

```
CiME> let X = vars "x y z l b";
CiME> let F = signature "
    #   : constant ;
    0,1 : postfix unary ;
    + : infix binary ;
    empty : constant ;
    singl : unary ;
    U : AC ;
    sum : unary";

CiME> let R = HTRS {} F X "
  (#)0 -> #;
  # + x -> x;
  x + # -> x;
  (x)0 + (y)0 -> (x+y)0;
  (x)0 + (y)1 -> (x+y)1;
  (x)1 + (y)0 -> (x+y)1;
  (x)1 + (y)1 -> (x+y+(#)1)0;
  empty U b -> b;
  sum(empty) -> (#)0;
  sum(singl(x)) -> x;
  sum(x U y) -> sum(x) + sum(y);
";

CiME> termcrit "minimal";

Termination now uses minimal decomposition
- : unit = ()

CiME> polyinterpkind "linear";

Now searching only linear polynomial interpretations
- : unit = ()

CiME> h_termination R;

 Entering the termination expert for modules. Verbose level = 0
Checking module:
{}

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ empty U V_4 -> V_4 } (1 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
```

```
{}

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{}

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{}

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ (#)0 -> # } (1 rules)

checking each of the 0 strongly connected components :
Termination proof found.

Checking module:
{ # + V_0 -> V_0,
  V_0 + # -> V_0,
  (V_0)0 + (V_1)0 -> (V_0 + V_1)0,
  (V_0)0 + (V_1)1 -> (V_0 + V_1)1,
  (V_0)1 + (V_1)0 -> (V_0 + V_1)1,
  (V_0)1 + (V_1)1 -> ((V_0 + V_1) + (#)1)0 } (6 rules)

checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[#] = 0;
[0](X0) = X0 + 1;
[1](X0) = X0 + 1;
[+](X0,X1) = X1 + X0;
['+`](X0,X1) = X1 + X0;

Termination proof found.

Checking module:
{ sum(V_0 U V_1) -> sum(V_0) + sum(V_1),
  sum(singl(V_0)) -> V_0,
  sum(empty) -> (#)0 } (3 rules)

checking each of the 1 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[#] = 0;
[0](X0) = 0;
[1](X0) = 0;
[+](X0,X1) = X1 + X0;
[empty] = 0;
[singl](X0) = X0;
```

```
[U](X0,X1) = X1 + X0 + 1;
[sum](X0) = X0;
['sum`](X0) = X0;

Termination proof found.

- : unit = ()
```

Chapter 2

Reference manual

2.1 CiME toplevel core language

This section describes the toplevel core language. This language is a simple functional language that allows one to evaluate expressions, and possibly naming the results by a `let` construct.

2.1.1 Formal syntax

The whole language is described by the following grammar:

```
toplevel_phrase  ::=  definition | evaluation | directive
     definition  ::=  let id = expr ; | let fun id id+ = expr ;
     evaluation  ::=  expr ;
      directive  ::=  # id ; | # id string ; | # id id ;
           expr  ::=  id | int | string | ( expr ) | expr op expr | - expr |
                      expr expr | if expr then expr else expr | fun id -> expr
             op  ::=  + | - | * | = | <> | > | < | >= | <=
```

The identifiers are made with sequences of letters, digits, underline and quote characters; and should start by a letter. Beware that the interpreter makes a difference between uppercase and lowercase letters.

The priority between operators are, from lowest to highest: `or`, `and`, `not`, comparisons, `+` and `-`, `*`.

Comments may be put between `(*` and `*)`.

Apart from these very general constructs, all the toplevel language is made of predefined functions. These are described in the next section.

2.1.2 A note on static binding

The `let` construct is not like assignment, there is no possibility to change the value of a variable. Several `let`s for the same name define several variable, following the static binding semantics, which is illustrated by the following example:

```
CiME> let x=1;
x : int = 1
CiME> let fun f y = x*y+1;
f : int -> int = <fun>
CiME> f 3;
- : int = 4
CiME> let x=2;
x : int = 2
CiME> f 3;
- : int = 4
```

The second definition of `x` hides the first one, but in the definition of function `f`, `x` still refers to the first definition of `x`.

2.1.3 Directives

Some directives controls the interpreter. These directives are prefixed by `#`, and optionally take an argument. The available directives are the following.

- `#help`. Prints the list of predefined functions available.
- `#help` *name*. Prints some help on the predefined function *name*.
- `#load` *file*. Read commands from *file* as if they were typed directly. Notice that such a file may also contain directives, including the `#load` directive.
- `#time` [`on` or `off`]. When on, the time spent in executing a command is shown for each command. When no argument, toggles between on and off.
- `#quit`. Exits the interpreter.
- `#verbose` [*n*]. Sets verbose level to *n*, default 1. The verbose level controls for some functions how much verbose are they working. See details on each such function.
- `#quiet`. Equivalent to `#verbose 0`.
- `#mem` or `#memory`. Display amount of memory allocated so far.
- `#extern` *command*. Execute the *command* in a sub-shell.

2.2 C*i*ME toplevel predefined functions

This section described the predefined functions of the toplevel language. The list of each predefined function available may be obtain by directive `#help`, and a simple on-line help is obtained on a given function *name* by `#help "`*name*`"`.

2.2.1 Diophantine constraints

`dioph_constraint : string -> dioph_constraint`

(`dioph_constraint` s) returns a Diophantine constraint as read syntactically from the string s. The syntax of constraint is

Constraints	::=	(*Constraint*`;`)*
Constraint	::=	*Poly Op Poly*
Op	::=	`=` | `>` | `<` | `>=` | `<=`
Poly	::=	*Int* | *Var* | *Poly* `+` *Poly* | *Poly* `.` *Poly* | `-` *Poly* | `(` *Poly* `)` | *Poly*`^` *Int*
Ident	::=	$[\mathtt{a}-\mathtt{z}][\mathtt{0}-\mathtt{9}]^*$
Int	::=	$[\mathtt{0}-\mathtt{9}]^+$

`dioph_solve : int -> dioph_constraint -> unit`

(`dioph_solve` N c) solves the constraints c for variables between 0 and N. Only the first solution found is displayed.

2.2.2 Rewriting on words

This subsection describes functions for defining alphabets, words, word rewriting systems (SRSs[1]), precedences on alphabet symbols, orderings on words ; and functions for normalizing words, performing completion on SRSs.

[1]we use the standard abbreviation SRS for *string rewriting system*, but we always use *word* instead of *string* to avoid confusion with character strings

Definition of alphabet, words and SRSs

```
word_signature : string -> word_signature,
   word : ('A:word_signature) -> string -> 'A word,
   SRS : ('A:word_signature) -> string -> 'A SRS,
```

Normalization

`word_normalize : 'A SRS -> 'A word -> 'A word`.

Orderings on words

`word_precedence`, `length_lex`, `multi_lex`, `word_compare`.

Knuth-Bendix completion

`word_completion`.

2.2.3 Rewriting on first-order terms

Definition of signatures

the `signature` function takes a string as argument and return a signature:

```
signature : string -> signature
```

The string must constain syntactic definitions of operator symbols together with their arity, and possibly declare them as commutative or AC. These definitions must follow the following syntax.

$$
\begin{array}{rcl}
\mathit{Decl} & ::= & (\mathit{ArityDecl}\,\texttt{;})^* \\
\mathit{ArityDecl} & ::= & \mathit{OpList}\ \texttt{:}\ \mathit{Fix}\ \mathit{Arity} \\
\mathit{Arity} & ::= & \texttt{AC} \mid \texttt{commutative} \mid \texttt{constant} \mid \texttt{unary} \mid \texttt{binary} \mid \mathit{Int} \\
\mathit{Fix} & ::= & \texttt{infix} \mid \texttt{prefix} \mid \texttt{postfix} \mid \varepsilon \\
\mathit{OpList} & ::= & \mathit{Ident}\ (\texttt{,}\ \mathit{Ident})^* \\
\mathit{Ident} & ::= & [\texttt{a}-\texttt{z},\texttt{A}-\texttt{Z},\texttt{0}-\texttt{9},\texttt{\_},\texttt{'}]^+ \\
\mathit{Ident} & ::= & [\texttt{\^{}},\texttt{+},\texttt{.},\texttt{\&},\texttt{*},\texttt{-},\texttt{/},\texttt{?},\texttt{!},\texttt{@},\texttt{\~{}},\texttt{\#},|,\texttt{:},\texttt{\%},\texttt{\$},\texttt{<},\texttt{=},\texttt{>}]^+ \\
\mathit{Int} & ::= & [\texttt{0}-\texttt{9}]^+
\end{array}
$$

Arities of free symbols have to be given either by an number or by the abbreviations `constant` for 0, `unary` for 1 or `binary` for 2. Commutative or associative-commutative symbols are simply declared by the special keywords `commutative` and `AC`.

Valid identifiers are any word on $\{\texttt{a},\ldots,\texttt{z},\texttt{A},\ldots,\texttt{Z},\texttt{0},\ldots,\texttt{9},\texttt{\_},\texttt{'}\}$ and any word over characters $\{\texttt{\^{}},\texttt{+},\texttt{.},\texttt{\&},\texttt{*},\texttt{-},\texttt{/},\texttt{?},\texttt{!},\texttt{@},\texttt{\~{}},\texttt{\#},|,\texttt{:},\texttt{\%},\texttt{\$},\texttt{<},\texttt{=},\texttt{>}\}$, accept the following reserved words

```
constant unary binary infix prefix postfix commutative AC
: -> < > = <= >= <>
```

The optional fix status allows one to enforce the way a symbol will be syntactically used in the following: `prefix` for an operator placed before its arguments, `infix` for a binary operator placed between its arguments, and `postfix` for an operator placed after its arguments. By default, commutative and AC operators are infix and the others are prefix.

For example:

```
a,b,c,0,1 : constant ;
f,g       : unary ;
!         : postfix unary ;
.         : infix binary ;
p         : 3 ;
+         : AC ;
eq        : commutative ;
```

declares five constants a, b, c, 0 and 1; two unary functions f and g; a unary function ! to be used in postfix notation (may be for the factorial function...); a binary function . that will be used in infix notation even if not AC; a ternary function p, an AC symbol +; acommutative symbol eq and three variables x, y and z.

Notice that the last semicolon is optional.

Declaring variable names

the `vars` function takes a string as argument and return a set variable names:

```
vars : string -> variable set
```

Definition of terms

```
term : ('A:signature) -> ('B:variable_set) -> string -> ('A,'B) term
```

`term F X s` builds the term denoted by the string `s`, using signature `F` and variables of `X`. A error message is issued and no term is built of the string `s` is not well-formed with respect to the grammar below:

$$\begin{array}{lcl} \textit{Term} & ::= & \textit{Var} \mid \textit{ConstantSymb} \mid \textit{PrefixSymb}\ \textit{Term} \mid \textit{PrefixSymb}\ \texttt{(}\ \textit{Term}\ (\texttt{,}\,\textit{Term})^{*}\ \texttt{)} \\ \textit{Term} & ::= & \textit{Term}\ \textit{PostfixSymb} \mid \texttt{(}\ \textit{Term}\ (\texttt{,}\,\textit{Term})^{*}\ \texttt{)}\ \textit{PostfixSymb} \mid \textit{Term}\ \textit{InfixSymb}\ \textit{Term} \end{array}$$

Ambiguities are resolved as follows: prefix symbols have priority over postfix symbols, which themselves have priority over infix symbols. Infix symbols associate to the left. For example, for the signature

```
  # : prefix unary ;
  ! : postfix unary ;
  +,* : infix binary ;
```

`#x!` is parsed as `(#x)!`, `#x+y` is parsed as `(#x)+y`, `x+y!` is parsed as `x+(y!)`, `x+y*z` is parsed as `(x+y)*z`. You must use parenthesis to enforce other priorities.

Definition of TRS

```
TRS : ('A:signature) -> ('B:variable_set) -> string -> ('A,'B) TRS
```

Reduction of terms

```
normalize : ('A,'B) TRS -> ('A,'B) term -> ('A,'B) term
```

`normalize R t` returns a normal form of the term `t` with respect to the rewrite system `R`. This function call may run forever if `R` is not terminating on `t`. The strategy is innermost.

Notice that the type of this function implies that the rewrite system and the term have to be defined on exactly the same signature.

2.2.4 Defining term orderings

```
precedence : ('A:signature) -> string -> 'A precedence
```

`precedence S s` builds the precedence on signature `S` given by string `s`.

```
MPO : 'A precedence -> 'A term_ordering
```

`MPO p` is the multiset path ordering on terms defined the precedence `p`.

```
ACRPO : 'A precedence -> 'A term_ordering
```

`ACRPO p` is the AC path ordering [**?**] on terms defined the precedence `p`.

```
term_compare : 'A term_ordering -> ('A,'B) term -> ('A,'B) term ->
string
```

`term_compare o t1 t2` compares the terms `t1` and `t2` w.r.t. the term ordering `o`.

2.2.5 Checking termination

Note: checking termination of SRSs is not yet available.

2.2.6 Finding termination proofs

The main function for searching for a termination proof is

```
termination : ('A,'B) TRS -> unit
```

it tries to find a termination proof of the rewrite system given as argument. The result of that search is displayed on standard output.

There are several way of controlling the search, explained below.

Search for polynomial interpretations

```
polyinterpkind : (string * int) set -> unit
```

Sets the kind of interpretations to search for, in order. Valid values are `"linear"`, `"simple"`, and `simple-mixed"` [**?**, **?**]. Sets the search bound for polynomial coefficients. Valid values are any positive integer. Default is `{ ("linear",2) ; ("simple-mixed",6) }`.

Linear interpretations are of the form

$$P_f(x_1,\dots,x_n) = a_1x_1 + \cdots + a_nx_n + c$$

simple interpretations are of the form

$$P_f(x_1,\dots,x_n) = \sum_{i_1,\dots,i_n \in \{0,1\}} a_{i_1,\dots,i_n} x_1^{i_1} \cdots x_n^{i_n}$$

and simple-mixed are the same as simple except for unary symbols, which are of the form

$$P_f(x) = ax^2 + bx + c$$

Index

Part II

Implementation manual

Chapter 3

Library `Compat`

This library provides complements OCaml standard library, and various all-purposes data-structures.

3.1 Module *Int_utils*

power, *pgcd* = greatest common divisor, *ppcm* = least common multiple.

full_pgcd x y returns $d, k1, k2$ such that d=gcd(x,y), x=k1*d and y=k2*d.

val *power* : $int \rightarrow int \rightarrow int$
val *pgcd* : $int \rightarrow int \rightarrow int$
val *full_pgcd* : $int \rightarrow int \rightarrow int \times int \times int$
val *ppcm* : $int \rightarrow int \rightarrow int$

3.2 Module *Arrayutils*

val *index* : $'a \rightarrow$ $'a$ array $\rightarrow int$

val *find* : $('a \rightarrow bool) \rightarrow$ $'a$ array $\rightarrow int$

val *filter* : $('a \rightarrow bool) \rightarrow$ $'a$ array $\rightarrow$ $'a\ list$

val *fold_lefti* : $('a \rightarrow int \rightarrow {}'b \rightarrow {}'a) \rightarrow {}'a \rightarrow {}'b$ array $\rightarrow {}'a$

val *fold_righti* : $(int \rightarrow {}'a \rightarrow {}'b \rightarrow {}'b) \rightarrow {}'a$ array $\rightarrow {}'b \rightarrow {}'b$

val *filter_indices* : $('a \rightarrow bool) \rightarrow {}'a$ array $\rightarrow int\ list$

3.3 Module *Listutils*

This module provides some basic functions on lists, mainly functions that used to exist in earlier version of CAML, but do not exist anymore.

Advice: the functions that operate on lists as if they were sets should not be used anymore, the module Set should be used instead.

val *power* : $'a\ list \rightarrow int \rightarrow {}'a\ list$

val *intersect* : $'a\ list \rightarrow\ 'a\ list \rightarrow\ 'a\ list$
val *add* : $'a \rightarrow\ 'a\ list \rightarrow\ 'a\ list$
val *union* : $'a\ list \rightarrow\ 'a\ list \rightarrow\ 'a\ list$
val *index* : $'a \rightarrow\ 'a\ list \rightarrow\ int$
val *except* : $'a \rightarrow\ 'a\ list \rightarrow\ 'a\ list$
val *remove* : $'a \rightarrow (\ 'a \times\ 'b)\ list \rightarrow (\ 'a \times\ 'b)\ list$
val *flat_map* : $(\ 'a \rightarrow\ 'b\ list) \rightarrow\ 'a\ list \rightarrow\ 'b\ list$
val *subtract* : $'a\ list \rightarrow\ 'a\ list \rightarrow\ 'a\ list$
val *do_list_combine* : $(\ 'a \times\ 'b \rightarrow\ 'c) \rightarrow\ 'a\ list \times\ 'b\ list \rightarrow\ unit$
val *map_filter* : $(\ 'a \rightarrow\ 'b) \rightarrow (\ 'a \rightarrow\ bool) \rightarrow\ 'a\ list \rightarrow\ 'b\ list$
val *map_with_exception* : $exn \rightarrow (\ 'a \rightarrow\ 'b) \rightarrow\ 'a\ list \rightarrow\ 'b\ list$
val *split* : $string \rightarrow\ string\ list$

val *mapi* : $(int \rightarrow\ 'a \rightarrow\ 'b) \rightarrow\ 'a\ list \rightarrow\ 'b\ list$
val *flat_mapi* : $(int \rightarrow\ 'a \rightarrow\ 'b\ list) \rightarrow\ 'a\ list \rightarrow\ 'b\ list$
val *foldi* : $(int \rightarrow\ 'a \rightarrow\ 'b \rightarrow\ 'b) \rightarrow\ 'a\ list \rightarrow\ 'b \rightarrow\ 'b$

fold_right_env f env $[l1;..;lk]$ is equivalent to

```
let env1,e1 = f env l1 in
let env2,e2 = f env1 l2 in
...
let envk,ek = f env{k-1} lk in
envk,[e1;..,ek]
```

val *fold_right_env* :
$(\ 'a \rightarrow\ 'b \rightarrow\ 'a \times\ 'c) \rightarrow\ 'a \rightarrow\ 'b\ list \rightarrow (\ 'a \times\ 'c\ list)$

val *assoc_right* : $'a \rightarrow (\ 'b \times\ 'a)\ list \rightarrow\ 'b$
val *flatten_left* : $(\ 'a\ list \times\ 'b)\ list \rightarrow (\ 'a \times\ 'b)\ list$

3.4 Module *Balanced_trees*

data type of balanced tree with one info of type *'a*

type $'a\ t\ =\ Empty \mid Node$ of $'a\ t \times\ 'a \times\ 'a\ t \times\ int$

val *height* : $'a\ t \rightarrow\ int$

Creates a new node with left son l, value x and right son r. l and r must be balanced and | height l - height r | <= 2. Inline expansion of height for better speed.

val *create* : $'a\ t \rightarrow\ 'a \rightarrow\ 'a\ t \rightarrow\ 'a\ t$

Same as create, but performs one step of rebalancing if necessary. Assumes l and r balanced. Inline expansion of create for better speed in the most frequent case where no rebalancing is required.

val *bal* : $'a\ t \rightarrow\ 'a \rightarrow\ 'a\ t \rightarrow\ 'a\ t$

Same as bal, but repeat rebalancing until the final result is balanced.

val *join* : $'a\ t \rightarrow\ 'a \rightarrow\ 'a\ t \rightarrow\ 'a\ t$

Merge two trees l and r into one. All elements of l must precede the elements of r. Assumes | height l - height r | <= 2.

val *merge* : $'a\ t \rightarrow\ 'a\ t \rightarrow\ 'a\ t$

Same as merge, but does not assume anything about l and r.

val *concat* : $'a\ t \rightarrow\ 'a\ t \rightarrow\ 'a\ t$

3.5 Module *Mapord*

$Id: mapord.mli, v1.2 1998/04/28 13:35:00 marche Exp$

```
module type OrderedType =
  sig
    type t
    val compare : t → t → int
  end

module type OrderedMap =
  sig
    type key
    type 'a t
    val empty : 'a t
    val add : key → 'a → 'a t → 'a t
    val find : key → 'a t → 'a
    val remove : key → 'a t → 'a t
    val iter : (key → 'a → unit) → 'a t → unit
    val map : ('a → 'b) → 'a t → 'b t
    val fold : (key → 'a → 'b → 'b) → 'a t → 'b → 'b
    val max : 'a t → key × 'a
    val min : 'a t → key × 'a
    val elements_increasing_order : 'a t → (key × 'a) list
  end

module Make(Ord : OrderedType) : OrderedMap with type key = Ord.t
```

3.6 Module *Balanced_trees2*

```
type ('key,'data) node_info =
    {
      key : 'key;
      data : 'data;
      height : int;
      left : ('key,'data) t;
      right : ('key,'data) t
    }

and ('key,'data) t =
    Empty
  | Node of ('key,'data) node_info

val empty : ('key, 'data) t

val height : ('key, 'data) t → int

val create :
  ('key, 'data) t → 'key → 'data → ('key, 'data) t → ('key, 'data) t

val bal :
  ('key, 'data) t → 'key → 'data → ('key, 'data) t → ('key, 'data) t

val merge : ('key, 'data) t → ('key, 'data) t → ('key, 'data) t

val iter : ('key → 'data → 'a) → ('key, 'data) t → unit

val map : ('data1 → 'data2) → ('key, 'data1) t → ('key, 'data2) t

val mapi : ('key → 'data1 → 'data2) → ('key, 'data1) t → ('key, 'data2) t

val fold : ('key → 'data → 'a → 'a) → ('key, 'data) t → 'a → 'a
```

3.7 Module *Ordered_maps*

module type *OrderedMap* =
  sig
    type *'a key*
    type *('a,'b) t*
    val *empty* : $('a,'b)\ t$
    val *is_empty* : $('a,'b)\ t \rightarrow bool$
    val *add* : $'a$ key $\rightarrow\ 'b \rightarrow ('a,'b)\ t \rightarrow ('a,'b)\ t$
    val *find* : $'a$ key $\rightarrow ('a,'b)\ t \rightarrow\ 'b$
    val *remove* : $'a$ key $\rightarrow ('a,'b)\ t \rightarrow ('a,'b)\ t$
    val *mem* : $'a$ key $\rightarrow ('a,'b)\ t \rightarrow bool$
    val *iter* : $('a$ key $\rightarrow\ 'b \rightarrow unit) \rightarrow ('a,'b)\ t \rightarrow unit$
    val *map* : $('b \rightarrow\ 'c) \rightarrow ('a,'b)\ t \rightarrow ('a,'c)\ t$
    val *mapi* : $('a$ key $\rightarrow\ 'b \rightarrow\ 'c) \rightarrow ('a,'b)\ t \rightarrow ('a,'c)\ t$
    val *fold* : $('a$ key $\rightarrow\ 'b \rightarrow\ 'c \rightarrow\ 'c) \rightarrow ('a,'b)\ t \rightarrow\ 'c \rightarrow\ 'c$
    val *fold_from_min* : $('a$ key $\rightarrow\ 'b \rightarrow\ 'c \rightarrow\ 'c) \rightarrow ('a,'b)\ t \rightarrow\ 'c \rightarrow\ 'c$
    val *max* : $('a,'b)\ t \rightarrow\ 'a$ key $\times\ 'b$
    val *min* : $('a,'b)\ t \rightarrow\ 'a$ key $\times\ 'b$
    val *elements_increasing_order* : $('a,'b)\ t \rightarrow ('a$ key $\times\ 'b)$ list
    val *exists_one* : $('a$ key $\rightarrow\ 'b \rightarrow bool) \rightarrow ('a,'b)\ t \rightarrow bool$
    val *find_one* : $('a$ key $\rightarrow\ 'b \rightarrow bool) \rightarrow ('a,'b)\ t \rightarrow\ 'b$
    val *find_key* : $('a$ key $\rightarrow\ 'b \rightarrow bool) \rightarrow ('a,'b)\ t \rightarrow\ 'a$ key
    val *size* : $('a,'b)\ t \rightarrow int$
    val *union* : $('a,'b)\ t \rightarrow ('a,'b)\ t \rightarrow ('a,'b)\ t$
  end

module *Make*(*Ord* : *Ordered_types.OrderedType*) :
  OrderedMap with type *'a key* = *Ord.t*

module *MakePoly*(*Ord* : *Ordered_types.OrderedPolyType*) :
  OrderedMap with type *'a key* = *'a Ord.t*

3.8 Module *Bit_field*

Obsolete header:
  CiME Project - Démons research team - LRI - Université Paris XI
  $Id : bit_field.mli, v1.42001/04/2013 : 42 : 09 marcheExp$
  ****

module type *S* =
sig
  type *t*

all_zero n returns a bit filed of size n filled with 0. *all_one n* returns a bit filed of size n filled with 1.

  val *all_zero* : $int \rightarrow t$
  val *all_one* : $int \rightarrow t$

size of a bit field

  val *bit_length* : $t \rightarrow int$

boolean operations. bit fields have to be the same length.

  val *bit_and* : $t \rightarrow t \rightarrow t$
  val *bit_or* : $t \rightarrow t \rightarrow t$
  val *bit_not* : $t \rightarrow t$

bit_nth n l returns a *bit_field* encoding a vector of bits of length 31*l where all the bits are equal to 0, except at position *n* which is a 1.

```
(0,...,0,1,0,...,0)
 ^       ^       ^
 |       |       |
 0       n    (31*l) -1
```

bit_nth_first n l returs a *bit_field* encoding a vector of bits of length 31*l where the first nth bits are equal to 1, the others being equal to 0.

```
(1,...,1,1,0,...,0)
 ^       ^       ^
 |       |       |
 0       n    (31*l) -1
```

val *bit_nth* : *int* → *int* → *t*
val *bit_nth_first* : *int* → *int* → *t*

is_zero b returns true if the bit field *b* encodes the integer 0 in base 2.
atmost_one_one b returns true if the bit field *b* encodes a power of 2 in base 2.

val *is_zero* : *t* → *bool*

val *atmost_one_one* : *t* → *bool*

bit_field_to_vect_of_bits l bf returns the vector of bits of length l encoded by the *bit_field bf* *vect_of_bits_to_bit_field v* returns the *bit_field* encoding the vector of bits *v*

val *bit_field_to_vect_of_bits* : *int* → *t* → *int array*
val *vect_of_bits_to_bit_field* : *int array* → *t*

pretty-prints a bit field in the current formating box

val *print_bit_field* : *int* → *t* → *unit*

end

module *Small_bit_field* : *S* with type *t* = *int*

module *Large_bit_field* : *S* with type *t* = *int array*

3.9 Module *Numbers*

the type of numbers (integer or rational)

type *t*

val *hash* : *t* → *int*

basic constants and conversion from machine ints

val *zero* : *t*

val *one* : *t*

val *two* : *t*
val *minus_one* : *t*
val *of_int* : *int* → *t*

raises *Invalid_argument* `"Numbers.to_int"` if too big

val *to_int* : *t* → *int*

basic operations overs rational numbers

val $is\_zero\ :\ t\ \rightarrow\ bool$

val $denominator\ :\ t\ \rightarrow\ t$

val $add\ :\ t\ \rightarrow\ t\ \rightarrow\ t$

val $sub\ :\ t\ \rightarrow\ t\ \rightarrow\ t$

val $minus\ :\ t\ \rightarrow\ t$

val $abs\ :\ t\ \rightarrow\ t$

val $mult\ :\ t\ \rightarrow\ t\ \rightarrow\ t$

val $div\ :\ t\ \rightarrow\ t\ \rightarrow\ t$

val $power\_int\ :\ t\ \rightarrow\ int \rightarrow\ t$

comparators

val $compare\ :\ t\ \rightarrow\ t\ \rightarrow\ int$

val $eq\ :\ t\ \rightarrow\ t\ \rightarrow\ bool$

val $neq\ :\ t\ \rightarrow\ t\ \rightarrow\ bool$

val $ge\ :\ t\ \rightarrow\ t\ \rightarrow\ bool$

val $gt\ :\ t\ \rightarrow\ t\ \rightarrow\ bool$

val $le\ :\ t\ \rightarrow\ t\ \rightarrow\ bool$

val $lt\ :\ t\ \rightarrow\ t\ \rightarrow\ bool$

val $max\ :\ t\ \rightarrow\ t\ \rightarrow\ t$

val $min\ :\ t\ \rightarrow\ t\ \rightarrow\ t$

integer operations

val $succ\ :\ t\ \rightarrow\ t$

val $pred\ :\ t\ \rightarrow\ t$

val $quo\ :\ t\ \rightarrow\ t\ \rightarrow\ t$

val $modulo\ :\ t\ \rightarrow\ t\ \rightarrow\ t$

val $pgcd\ :\ t\ \rightarrow\ t\ \rightarrow\ t$
val $full\_pgcd\ :\ t\ \rightarrow\ t\ \rightarrow\ t\ \times\ t\ \times\ t$
val $ppcm\ :\ t\ \rightarrow\ t\ \rightarrow\ t$

val $div\_floor\ :\ t\ \rightarrow\ t\ \rightarrow\ t$
val $div\_ceil\ :\ t\ \rightarrow\ t\ \rightarrow\ t$
val $sqrt\_floor\ :\ t\ \rightarrow\ t$
val $sqrt\_ceil\ :\ t\ \rightarrow\ t$
val $root\_floor\ :\ int \rightarrow\ t\ \rightarrow\ t$
val $root\_ceil\ :\ int \rightarrow\ t\ \rightarrow\ t$

parsing and printing

val $from\_string\ :\ string \rightarrow\ t$

val $to\_string\ :\ t\ \rightarrow\ string$

3.10 Module *Ordered_sets*

module type *OrderedSet* =
sig
  type *'a elt*
  type *'a t* val *empty* : *'a t*
    (* The empty set. *)

val *is_empty* : *'a t* → *bool*
  (* Test whether a set is empty or not. *)

val *mem* : *'a elt* → *'a t* → *bool*
  (* *mem* x s tests whether x belongs to the set s. *)

val *add* : *'a elt* → *'a t* → *'a t*
  (* *add* x s returns a set containing all elements of s, plus x. If x was already in s, s is returned unchanged. *)

val *singleton* : *'a elt* → *'a t*
  (* *singleton* x returns a set containing x. *)

val *remove* : *'a elt* → *'a t* → *'a t*
  (* *remove* x s returns a set containing all elements of s, except x. If x was not in s, s is returned unchanged. *)

val *get_and_remove_min* : *'a t* → *'a elt* × *'a t*
  (* *get_and_remove_min* s returns a pair m, s' where m is the minimum element of s and s' is a set containing all elements of s, except m. If s is empty, the exception *Not_found* is raised. *)

val *union* : *'a t* → *'a t* → *'a t*
val *inter* : *'a t* → *'a t* → *'a t*
val *diff* : *'a t* → *'a t* → *'a t*
  (* Union, intersection and set difference. *)

val *compare* : *'a t* → *'a t* → *int*
  (* Total ordering between sets. Can be used as the ordering function for doing sets of sets. *)

val *equal* : *'a t* → *'a t* → *bool*
  (* *equal* *s1* *s2* tests whether the sets *s1* and *s2* are equal, that is, contain the same elements. *)

val *subset* : *'a t* → *'a t* → *bool*
  (* *subset* *s1* *s2* tests whether the set *s1* is a subset of the set *s2*. *)

val *iter* : (*'a elt* → *unit*) → *'a t* → *unit*
  (* *iter* f s applies f in turn to all elements of s. The order in which the elements of s are presented to f is unspecified. *)

val *fold* : (*'a elt* → *'b* → *'b*) → *'a t* → *'b* → *'b*
  (* *fold* f s a computes (f *xN* ... (f *x2* (f *x1* a))...), where *x1* ... *xN* are the elements of s. The order in which elements of s are presented to f is unspecified. *)

val *fold_from_min* : (*'a elt* → *'b* → *'b*) → *'a t* → *'b* → *'b*
  (* *fold* f s a computes (f *xN* ... (f *x2* (f *x1* a))...), where *x1* ... *xN* are the elements of s. The order in which elements of s are presented to f is increasing. *)

val *filter* : (*'a elt* → *bool*) → *'a t* → *'a t*
  (* *filter p s returns the set* of *all elements* in *s that satisfy predicate p*. ×)

val *cardinal* : *'a t* → *int*
  (* Return the number of elements of a set. *)

val *elements* : *'a t* → *'a elt list*
  (* Return the list of all elements of the given set. The elements appear in the list in some unspecified order. *)

val *choose* : *'a t* → *'a elt*
(∗ Return one element of the given set, or raise *Not_found* if the set is empty. Which element is chosen is unspecified, but equal elements will be chosen for equal sets. ∗)

val *min_elt* : *'a t* → *'a elt*
(∗ Return the smallest element of the given set, or raise *Not_found* if the set is empty. ∗)

val *find* : (*'a elt* → *bool*) → *'a t* → *'a elt*

val *find_and_apply* : (*'a elt* → *'c*) → *'a t* → *'c*
(∗ *find_and_apply f set* returns *f e* where *e* is an element of *set* such that *f e* does not raise *Not_found*. If there is no such element, *Not_found* is raised. ∗)

val *exists* : (*'a elt* → *bool*) → *'a t* → *bool*

val *for_all* : (*'a elt* → *bool*) → *'a t* → *bool*

val *half_set* : *'a t* → *'a t*
(∗ *half_set set* returns a subset of *set* such that its elements are minimal and its cardinal is roughly the half of the cardinal of set. ∗)

end

module *Make*(*Ord* : *Ordered_types.OrderedType*) :
(*OrderedSet* with type *'a elt* = *Ord.t*)
(∗ Functor building an implementation of the set structure given a totally ordered type. ∗)

module *MakePoly*(*Ord* : *Ordered_types.OrderedPolyType*) :
(*OrderedSet* with type *'a elt* = *'a Ord.t*)
(∗ Functor building an implementation of the set structure given a totally ordered polymorphic type. ∗)

3.11 Module *Cache*

This module provides some kind of maps with efficient access to elements. The size is bounded so when the cache is full, a free place is obtained by removing the element with the oldest access time.

3.11.1 Type of caches, and basic access functions

type (*'a*, *'b*) *cache*

(*create n*) returns a new cache of size n. n should be at least 1!

val *create* : *int* → (*'a*, *'b*) *cache*

(*add x y c*) adds the pair (x, y) in cache c. If there is no room left in c, the earliest accessed pair of c is removed.
If there was already a pair (x, z) for some z in c, it becomes hidden. It will never accessible again since there is no remove function on cache. It will eventually be removed after enough calls to add.

val *add* : *'a* → *'b* → (*'a*, *'b*) *cache* → *unit*

(*find x c*) returns y such that (x, y) belongs to c. This pair will be marked as the more recently accessed pair.
find raises exception *Not_found* if no such pair exists.

val *find* : *'a* → (*'a*, *'b*) *cache* → *'b*

int_int_find is a specialized version of *find* whenever the type of its first argument is *int* × *int*. In this case, this equality test should be more efficient.

val *int_int_find* : *int* × *int* → (*int* × *int*, *'b*) *cache* → *'b*

3.11.2 Iterator

$(iter\ f\ c)$ applies function f to each pair (x,y) of the cache c. the elements are processed from the newest accessed pair to the oldest

val *iter* : $('a \rightarrow\ 'b \rightarrow unit) \rightarrow ('a,'b)\ cache \rightarrow unit$

function1 n f is a function that calls f with caching of arguments, in a cache of size n

val *function1* : $int \rightarrow ('a \rightarrow\ 'b) \rightarrow ('a \rightarrow\ 'b)$

3.11.3 Statistics

$(collision\_rate\ c)$ returns a real value t which estimate the repartition of elements in the cache hash table. In fact, it is the average time to access an element in the cache.

$(max\_hash\_length\ c)$ is the maximal amount of time to access to an element.

$(list\_of\_cache\ c)$ returns the contents of the cache, useful for debugging purposes.

val *collision_rate* : $('a,'b)\ cache \rightarrow float$

val *max_hash_length* : $('a,'b)\ cache \rightarrow int$

val *list_of_cache* : $('a,'b)\ cache \rightarrow ('a \times\ 'b)\ list$

3.12 Module *Ordered_types*

```
module type OrderedType =
  sig
    type t
    val compare : t → t → int
  end
```

```
module type OrderedPolyType =
  sig
    type 'a t
    val compare : 'a t → 'a t → int
  end
```

3.13 Module *Class_maps*

```
class type ['a, 'b] map =
  object
    method add : 'a → 'b → unit
    method find : 'a → 'b
    method iter : ('a → 'b → unit) → unit
  end
```

val *empty_map* : $('a \rightarrow\ 'a \rightarrow int) \rightarrow ('a,'b)\ map$

3.14 Module *Polynomials*

This module allows to build polynomials over an arbitrary coefficient ring and an arbitrary set of variables.

3.14.1 Module for coefficient rings

Gives the type of coefficients, two constant 0 and 1, a test for nullity, and fonctions for addition, opposite, substraction and multiplication

```
module type RingType =
  sig
    type coef
    val zero : coef
    val one : coef
    val is_null : coef → bool
    val add : coef → coef → coef
    val minus : coef → coef
    val sub : coef → coef → coef
    val mult : coef → coef → coef
  end
```

3.14.2 Module for polynomials

```
module type PolynomialType =
  sig
```

Coefficients and variables

Types for coefficients and variables, and modules for sets and maps of variables.

```
    module Base_ring : RingType
    type coef = Base_ring.coef
    type variable
    module Var_set : Ordered_sets.OrderedSet
    module Var_map : Ordered_maps.OrderedMap
```

Monomials

(*make_monomial* c v) builds a monomial where c is the coefficient and v is a map giving to each variable its exponent (beware of giving only positive exponents and a non-zero coefficient)

(*monomial_coef* m) and (*monomials_vars*) are the corresponding acces functions.

(*monomial_degree*) returns the degree of m, that is the sum of its exponents.

```
    type monomial
    val make_monomial : coef → (variable,int) Var_map.t → monomial
    val monomial_coef : monomial → coef
    val monomial_vars : monomial → (variable,int) Var_map.t
    val monomial_degree : monomial → int
```

Type for polynomials, and basic polynomials

zero is the polynomial 0, *one* is 1

(*cte* a) is the constant polynomial a

$(var\ x)$ is the polynomial consisting in the variable x

type *poly*

val *zero* : *poly*
val *one* : *poly*
val *cte* : *coef* $\rightarrow$ *poly*
val *var* : *variable* $\rightarrow$ *poly*
val *poly_of_monomial* : *monomial* $\rightarrow$ *poly*

Basic functions over polynomials

$(is\_null\ p)$ is true if p is the polynomial 0
$(constant\_coef\ p)$ is the constant coefficient of p
$(total\_degree\ p)$ is the degree of p that is the maximum degree of its monomials
$(partial\_degre\ x\ p)$ is the partial degree of p in variable x, that is the maximum exponent of x in p
$(set\_of\_vars\ p)$ is the set of variables occurring in p

val *is_null* : *poly* $\rightarrow$ *bool*
val *constant_coef* : *poly* $\rightarrow$ *coef*
val *total_degree* : *poly* $\rightarrow$ *int*
val *partial_degree* : *variable* $\rightarrow$ *poly* $\rightarrow$ *int*
val *set_of_vars* : *poly* $\rightarrow$ *variable Var_set.t*

for debugging purposes

val *size* : *poly* $\rightarrow$ *int*

Printing function

$(print\_polynomial\ fc\ fv\ p)$ prints the polynomial p on standard output, where *fc* and *fv* are two printing functions for coefficients and variables. They must use the Format module for printing.

val *print_polynomial* :
$(coef \rightarrow unit) \rightarrow (variable \rightarrow unit) \rightarrow poly \rightarrow unit$

val *latex_print_polynomial* :
$out\_channel \rightarrow (coef \rightarrow unit) \rightarrow (variable \rightarrow unit) \rightarrow poly \rightarrow unit$
(*

Polynomial operations

$(add\ p1\ p2)$ is $[p1]+[p2]$
$(minus\ p)$ is $-[p]$
$(sub\ p1\ p2)$ is $[p1]-[p2]$
$(mult\ p1\ p2)$ is $[p1]\times[p2]$
$(power\ p\ n)$ is $[p]^{[n]}$
*)

val *add* : *poly* $\rightarrow$ *poly* $\rightarrow$ *poly*
val *minus* : *poly* $\rightarrow$ *poly*
val *sub* : *poly* $\rightarrow$ *poly* $\rightarrow$ *poly*
val *mult* : *poly* $\rightarrow$ *poly* $\rightarrow$ *poly*
val *power* : *poly* $\rightarrow$ *int* $\rightarrow$ *poly*

Polynomial evaluation

$(eval\ p\ m)$ evaluates p with variable values given by m, which is a map from variables to coefficients

val *eval* : $(variable \to coef) \to poly \to coef$

Polynomial substitutions

$(substitute\ m\ p)$ replaces in p each occurrence of variables by the corresponding polynomial in m, which is a map from variables to polynomials

val *substitute* : $(variable, poly)\ Var\_map.t \to poly \to poly$

Higher-order functions over polynomials

$(map\ f\ p)$ applies f to each monomials of p and builds the sum of the results. WARNING : this fonction assumes that the resulting set of monomials does have any two monomials with a common power product. If not the case, please use fold and add instead.

$(iter\ f\ p)$ applies f to each monomials of p.

$(fold\ f\ p\ a)$ computes $(f\ m1\ \ldots\ (f\ mN\ a)\ldots)$ where mi is the ith monomial of p, given in the lexicographic order of power products.

$(find\ f\ p)$ search for a monomial m in p such that (f m) does not raise *Not_found*. raises *Not_found* if no such monomial exists.

val *map* : $(monomial \to monomial) \to poly \to poly$
val *iter* : $(monomial \to unit) \to poly \to unit$
val *fold* : $(monomial \to 'a \to 'a) \to poly \to 'a \to 'a$
val *find* : $(monomial \to 'a) \to poly \to 'a$

end

3.14.3 The polynomial functor

Functor for building polynomial modules from a coefficient ring, an ordered set of variables, and two modules for set and map of variables.

module *Make*
(*Ring* : *RingType*)
(*Vars* : *Ordered_types.OrderedType*)
(*Var_set* : *Ordered_sets.OrderedSet* with type *'a elt* = *Vars.t*)
(*Var_map* : *Ordered_maps.OrderedMap* with type *'a key* = *Vars.t*) :

PolynomialType with
module *Base_ring* = *Ring*
and
type *variable* = *Vars.t*
and
module *Var_set* = *Var_set*
and
module *Var_map* = *Var_map*

3.15 Module *Classical_maps*

abstract data type of maps over some keys

type $('key, 'data)\ t$

empty comp returns an empty map, *comp* is a comparison function that is used for comparing keys.

val *empty* : $('key \to 'key \to int) \to ('key, 'data)\ t$

val *add* : $'key \to 'data \to ('key, 'data)\ t \to ('key, 'data)\ t$

val *find* : $'key \to ('key, 'data)\ t \to 'data$

val *remove* : $'key \to ('key, 'data)\ t \to ('key, 'data)\ t$

val *iter* : $('key \to 'data \to 'a) \to ('key, 'data)\ t \to unit$

val *map* : $('data \to 'data) \to ('key, 'data)\ t \to ('key, 'data)\ t$

val *fold* : $('key \to 'data \to 'a \to 'a) \to ('key, 'data)\ t \to 'a \to 'a$

3.16 Module *Labelled_graphs*

This module implements oriented graphs with labelled vertices (but no labels on arcs), in a functional way.

3.16.1 Vertices

abstract type for vertices and related modules for sets and maps of vertices

type *vertex*

module *VertexSet* : *Set.S* with type *elt* = *vertex*

module *VertexMap* : *Map.S* with type *key* = *vertex*

val *print_vertex_set* : *Format.formatter* → *VertexSet.t* → *unit*

3.16.2 Type for graphs

abstract type for graphs, parameterized by the type of vertex labels

type *'vertex_labels graph*

3.16.3 Constructor functions for graphs

empty_graph is the graph with no vertex

(*add_vertex g l*) returns a pair (v, g') where v is the new vertex having label l and g' is the graph obtained by adding v to g

(*add_arc g v1 v2*) returns the graph obtained by adding to g a new arc from vertex $v1$ to vertex $v2$. Raises Failure if the arc already exists.

val *empty_graph* : *'vertex_labels graph*

val *add_vertex* :
 'vertex_labels graph → *'vertex_labels*
 → *vertex* × *'vertex_labels graph*

val *add_arc* :
 'vertex_labels graph → *vertex* → *vertex* → *'vertex_labels graph*

3.16.4 Basic access functions

(*get_vertex_label g v*) returns the label of the vertex *v* in graph *g*

(*get_successors g v*) returns the set of vertices reachables from *v* in graph *g*, i.e. the set of *v'* suvh that there is an arc from *v* to *v'*.

val *get_vertex_label* : *'vertex_labels* graph → vertex → *'vertex_labels*

val *get_successors* : *'vertex_labels* graph → vertex → VertexSet.t

3.16.5 Iterator

(*fold_left_graph f a g*) returns *f* (... (*f* (*f a v1*) *v2*) ...) *vn* where *v1*,...,*vn* is the set of vertices of *g*, in a non-specified order.

val *fold_left_graph* :
  (*'a* → vertex → *'a*) → *'a* → *'vertex_labels* graph → *'a*

3.16.6 Extraction of subgraphs

Removing one vertex

The next two functions provide two ways of removing a vertex from a graph. (*remove_vertex_and_its_arcs g v*) returns the graph obtained from *g* by removing the vertex *v* and all the arcs going to *v* or comming from *v*. (*remove_vertex_and_link_its_arcs g v*) returns the graph obtained from *g* by removing *g* and "linking" the arcs going to *v* with arcs coming from *v*: if (v1,v) and (v,v2) are arcs in *g* then (v1,v2) is an arc in the new graph returned.

val *remove_vertex_and_its_arcs* :
  'vertex_labels graph → vertex → *'vertex_labels* graph

val *remove_vertex_and_link_its_arcs* :
  'vertex_labels graph → vertex → *'vertex_labels* graph

Extraction of of a whole subgraph

(*get_subgraph g e*) returns the subgraph of *g* whose vertices are in *e*, and arcs are the arcs of *g* that come from a vertex in *e* and go to a vertex in *e*.

val *get_subgraph* :
  'vertex_labels graph → VertexSet.t → *'vertex_labels* graph

3.16.7 Printing

(*print g f*) prints the graph *g*, using *f* as printing function for labels. Warning: this function *f* must use Format library!

(*compact_print g f*) does the same in a more compact format, using numbers to denote vertices.

(*dot_print g f*) does the same in *dot* format, see `http://www.research.att.com/sw/tools/graphviz/`.

val *print* : *'vertex_labels* graph → (*'vertex_labels* → *unit*) → *unit*

val *compact_print* : *'vertex_labels* graph → (*'vertex_labels* → *unit*) → *unit*

val *dot_print* : *'vertex_labels* graph → (*'vertex_labels* → *unit*) → *unit*

3.16.8 Cycles, connected components and such

Cycles

$(vertex\_has\_a\_1\_cycle\ g\ v)$ returns true if there is an arc from v to v in gq.

$(vertices\_on\_cycle\ g)$ returns the set of vertices of g that belong to a cycle

$(vertices\_not\_on\_cycle\ g)$ returns the set of vertices of g that do not belong to any cycle

$(split\_cycles\ g)$ returns a pair (a,b) of sets of vertices, where a is the set of vertices that lay on a cycle of g, b the set of vertices that do not belong to any cycle of g

val *vertex_has_a_1_cycle* :
  'vertex_labels graph → *vertex* → *bool*

val *split_cycles* :
  'vertex_labels graph → *VertexSet.t* × *VertexSet.t*

obsolete

```
val vertices_on_cycle :
  'vertex_labels graph -> VertexSet.t;;

val vertices_not_on_cycle :
  'vertex_labels graph -> VertexSet.t;;
```

Strongly connected components

$(compute\_strongly\_connected\_components\ even\_if\_no\_cycles\ g)$ returns the set of strongly connected components of the directed graph g. These are defined as follows: let $G = (V, E)$ be a directed graph, where V is the set of vertices and E the arcs, then let $G' = (V, E')$ where E' is defined by $v_1 \leftrightarrow_{E'} v_2$ iff $v_1 \rightarrow^{\star}_{E} v_2$ and $v_2 \rightarrow^{\star}_{E} v_1$. Then $\leftrightarrow_{E'}$ is an equivalence relation, and the strongly connected components of G are the equivalence classes of it.

if *even_if_no_cycles* is set to false, then the components containing only one vertex which has no arc to itself are not returned. Example : for the graph $a \rightarrow a, a \rightarrow b, b \rightarrow c, c \rightarrow c$, the component b will not be returned.

val *compute_strongly_connected_components* :
  bool → *'vertex_labels graph* → *VertexSet.t list*

Chapter 4

Library `regexp`

This is a regexp library, to be documented by Benjamin.

4.1 Module *Regular_expr*

This module defines the regular expressions, and provides some simple manipulation of them.

4.1.1 The regexp datatype and its constructors

The type of regular expressions is abstract. Regular expressions may be built from the following constructors :

- *empty* is the regexp that denotes no word at all.
- *epsilon* is the regexp that denotes the empty word.
- *char* c returns a regexp that denotes only the single-character word c.
- *chars* s returns a regexp that denotes any single-character word belonging to set of chars s.
- *string str* denotes the string *str* itself.
- *star* e where e is a regexp, denotes the Kleene iteration of e, that is all the words made of concatenation of zero, one or more words of e.
- *alt* $e1$ $e2$ returns a regexp for the union of languages of $e1$ and $e2$.
- *seq* $e1$ $e2$ returns a regexp for the concatenation of languages of $e1$ and $e2$.
- *opt* e returns a regexp for the set of words of e and the empty word.
- *some* e denotes all the words made of concatenation of one or more words of e.
- *compl* e denotes all the words not recognized by the regexp e.

type *'a regexp*

module *CharSet* : *Inttagset.IntTagSetModule* with type *'a elt* = *'a*

val *empty* : *'a regexp*

val *epsilon* : *'a regexp*

val *char* : *'a* → *'a regexp*

val chars : 'a CharSet.t -> 'a regexp;;
val string : string -> regexp;;

```
val star : 'a regexp → 'a regexp
val alt : 'a regexp → 'a regexp → 'a regexp
val seq : 'a regexp → 'a regexp → 'a regexp
val opt : 'a regexp → 'a regexp
val some : 'a regexp → 'a regexp
```

4.1.2 Simple regexp operations

The following three functions provide some simple operations on regular expressions:

- *nullable* r returns true if regexp r accepts the empty word.
- *residual* r c returns the regexp r' denoting the language of words w such that cw is in the language of r.
- *firstchars* r returns the set of characters that may occur at the beginning of words in the language of e.

```
val nullable : 'a regexp → bool
val residual : 'a regexp → 'a → 'a regexp
val firstchars : 'a regexp → 'a CharSet.t
```

4.1.3 Regexp matching by runtime interpretation of regexp

match_string r s returns true if the string s is in the language denoted by r. This function is by no means efficient, but it is enough if you just need to match once a simple regexp against a string.

If you have a complicated regexp, or if you're going to match the same regexp repeatedly against several strings, we recommend to use compilation of regexp provided by module *Automata*.

4.2 Implementation of module *Regular_expr*

```
module CharSet =
  Inttagset.MakePoly
    (struct type 'a t = 'a let tag = Hashtbl.hash end)

type 'a regexp =
  | Empty
  | Epsilon
  | Char of 'a CharSet.elt
  | Charset of 'a CharSet.t                    (* cardinal at least 2 *)
  | Star of 'a regexp
  | Alt of 'a regexp_set
  | Seq of 'a regexp × 'a regexp
and 'a regexp_set = 'a regexp list             (* naive implementation of sets *)

let add e l =
  if List.mem e l then l else e :: l

let rec union l1 l2 =
  match l1 with
    | [] → l2
    | e :: l → if List.mem e l2 then union l l2 else e :: union l l2
```

```
let empty = Empty

let epsilon = Epsilon

let char c = Char c

let star e =
  match e with
    | Empty | Epsilon -> Epsilon
    | Star _ -> e
    | _ -> Star e

let alt e1 e2 =
  match e1,e2 with
    | Empty,_ -> e2
    | _,Empty -> e1
    | Alt(l1),Alt(l2) -> Alt(union l1 l2)
    | Alt(l1),_ -> Alt(add e2 l1)
    | _,Alt(l2) -> Alt(add e1 l2)
    | _ -> if e1 = e2 then e1 else Alt([e1;e2])

let seq e1 e2 =
  match e1,e2 with
    | Empty,_ -> Empty
    | _,Empty -> Empty
    | Epsilon,_ -> e2
    | _,Epsilon -> e1
    | _ -> Seq(e1,e2)
```

4.2.1 extended regexp

```
let some e = (seq e (star e))

let opt e = (alt e epsilon)
```

4.2.2 Regexp match by run-time interpretation of regexp

```
let rec nullable r =
  match r with
    | Empty -> false
    | Epsilon -> true
    | Char _ -> false
    | Charset _ -> false
    | Star e -> true
    | Alt(l) -> List.exists nullable l
    | Seq(e1,e2) -> nullable e1 ∧ nullable e2
```

```
let rec residual r c =
  match r with
    | Empty → Empty
    | Epsilon → Empty
    | Char a → if a = c then Epsilon else Empty
    | Charset s → if CharSet.mem c s then Epsilon else Empty
    | Star e → seq (residual e c) r
    | Alt(l) →
        List.fold_right
          (fun e accu → alt (residual e c) accu)
          l
          Empty
    | Seq(e1,e2) →
        if nullable(e1)
        then alt (seq (residual e1 c) e2) (residual e2 c)
        else seq (residual e1 c) e2
```

firstchars r returns the set of characters that may start a word in the language of *r*

```
let rec firstchars r =
    match r with
      | Empty → CharSet.empty
      | Epsilon → CharSet.empty
      | Char a → CharSet.singleton a
      | Charset s → s
      | Star e → firstchars e
      | Alt(l) →
          List.fold_right
            (fun e accu →
                CharSet.union (firstchars e) accu)
            l
            CharSet.empty
      | Seq(e1,e2) →
          if nullable e1
          then CharSet.union (firstchars e1) (firstchars e2)
          else firstchars e1
```

4.3 Module *Automata*

```
module CharSet : Ordered_sets.OrderedSet with type 'a elt = 'a
```

Raised when one tries to count the number of words in an infinite language.

```
exception Infinite_language
```

count_nf sigma left_members compiled_srs returns the number of words not reducible by the *left_members* of the *compiled_srs*.

```
val count_nf :
  'a CharSet.t → 'a Words.word list → 'a String_rewriting.compiled_srs → int
```

4.4 Implementation of module *Automata*

```
module CharSet =
  Ordered_sets.MakePoly
    (struct type 'a t = 'a let compare = compare end)
```

4.4.1 The type of automata

Automata considered here are deterministic.

The states of these automata are always represented as natural numbers, the initial state always being 0.

An automaton is then made of a transition table containing at $i \times j$ the number of transitions from i to j.

```
type 'a full_automaton =
    {
      full_auto_trans : int array array;
      full_auto_accept : bool array;
      full_auto_init : bool array;
    }
```

Raised when one tries to count the number of words in an infinite language.

```
exception Infinite_language
```

```
let count_words {full_auto_accept = a ;
                 full_auto_trans = t;
                 full_auto_init = initials} =
  let n = Array.length t in
  let reaches_final = Array.create n false in
  let reaches_final_on_cycle = Array.create n false in
  let is_open = Array.create n false in
  let rec traverse i =
    if is_open.(i) ∨ reaches_final_on_cycle.(i) then begin
      reaches_final_on_cycle.(i) ← true
    end else begin
      reaches_final.(i) ← true;
      is_open.(i) ← true;
      for j = 0 to n − 1 do if t.(j).(i) > 0 then traverse j done;
      is_open.(i) ← false
    end
  in
    for i = 0 to n − 1 do if a.(i) then traverse i done;
  let count = Array.create n None in
  let rec count_from i = match count.(i) with
    | Some k →
        k
    | None →
        let k = ref 0 in
        for j = 0 to n − 1 do
          if t.(i).(j) > 0 ∧ reaches_final.(j) then begin
            if reaches_final_on_cycle.(j) then raise Infinite_language;
            k := !k + t.(i).(j) + count_from j
          end
        done;
        count.(i) ← Some !k;
        !k
  in
  let total = ref 1 in
  for i = 0 to n − 1 do if initials.(i) then total := !total + count_from i done;
  !total
```

```
let rec get_all_strict_sub w =
  if Array.length w ≤ 1 then CharSet.empty else
    begin
      let result = ref (CharSet.empty) in
```

```
      for i = 1 to (Array.length w − 1) do
        result := CharSet.add (Array.sub w i (Array.length w − i)) !result
      done;
      !result
    end

let does_overlap l r n =
  assert (n ≤ Array.length r);
  if n > Array.length l then false else
    let first_l = Array.length l − n in
    let first_r = 0 in
      try
        for i = 0 to n−1 do
          if l.(first_l+i) ≠ r.(first_r+i) then raise Exit
        done;
        true
      with Exit → false

exception Found of int

let longest_right_overlap left_array right =
try
  for n = Array.length right downto 1 do
    Array.iteri (fun i l → if i = 0 then () else
                          if does_overlap l right n then raise (Found i))
      left_array
  done;
  assert false
with
    Found n → n

let count_nf sigma word_list compiled_srs =
  let word_array = List.map Array.of_list word_list in
  let sigma_list = CharSet.elements sigma in
  let states_from_constructors = CharSet.fold
                                   (fun x acc → CharSet.add [|x|] acc)
                                   sigma CharSet.empty
  in
  let states_from_sub_terms =
    List.fold_left
      (fun acc x → CharSet.union acc (get_all_strict_sub x))
      states_from_constructors word_array
  in
  let state_number = 1 + CharSet.cardinal states_from_sub_terms in
  let _ = Format.printf "Number␣of␣states␣of␣the␣normal␣forms␣automaton:␣%d\n" state_number in
  let states_array = Array.create state_number [||] in
  let counter = ref 1 in
  let _ = CharSet.iter
            (fun x → states_array.(!counter) ← x; incr counter)
            states_from_sub_terms
  in
  let automaton =
    { full_auto_trans = Array.make_matrix state_number state_number 0;
      full_auto_accept = Array.init state_number (fun x → true) ;
      full_auto_init = Array.init state_number (function x → x = 0) ;
    }
  in
```

```
for i = 0 to state_number - 1 do
  List.iter
    (function letter ->
       let new_word = Array.append [|letter|] states_array.(i) in
         if String_rewriting.is_nf_compiled compiled_srs
           (Array.to_list new_word)
         then
           (
             let best = longest_right_overlap states_array new_word in
               automaton.full_auto_trans.(i).(best) <-
               automaton.full_auto_trans.(i).(best) + 1
           )
    )
    sigma_list
done;
count_words automaton
```

Chapter 5

Library Orderings

This library is for general definitions of orderings.

5.1 Module *Finite_orderings*

This module allows one to define ordering on finite sets of objects. The construction of such an ordering on a set E is obtained by starting from the "smallest" ordering on E (that is the relation $=$) and adding comparison pairs $x > y$ or $x = y$, performing the transitive closure at each step.

Formally speaking, this allows to build ordering on infinite sets, but only if they differ from identity on finitely many pairs.

exception *Incompatible*

type *'a finite_ordering*

val *compare* : *'a finite_ordering* → *'a Orderings_generalities.ordering*

val *add_equiv* : *'a finite_ordering* → *'a* → *'a* → *'a finite_ordering*

val *add_greater* : *'a finite_ordering* → *'a* → *'a* → *'a finite_ordering*

(*identity_ordering comp*) builds an identity ordering object on a given data type t. *comp* is any total comparison function on t: *comp* x y is 0 if x and y are equal, negative if $x < y$, and positive if $x > y$

val *identity_ordering* : (*'a* → *'a* → *int*) → *'a finite_ordering*

Building finite orderings from lists of comparisons.

(*add_list* o p) adds all comparison in list p to o. p is a concrete type that represents a list of comparison. For example $x > y = z < t$ is represented as $Gt(x, Eq(y, Lt(z, One(t))))$. On that example, it will do *add_greater* x y, *add_equiv* y z, and *add_greater* t z, of course performing transitive closure at each step.

type *'a precedence* =
One of *'a*
| *Gt* of *'a* × *'a precedence*
| *Lt* of *'a* × *'a precedence*
| *Eq* of *'a* × *'a precedence*

val *add_list* : *'a finite_ordering* → *'a precedence* → *'a finite_ordering*

val *map_prec* : (*'a* → *'b*) → *'a precedence* → *'b precedence*

5.2 Module *Orderings_generalities*

Comparison results

First, we define a type for given all different possible results of a comparison. Usually, a comparaison may result in either $>$, $<$ or $=$; but here the situation is a more complicated.

First, we are interested in what is called pre-orderings, or quasi-orderings, that is two different objects may be *equivalent* w.r.t. an ordering. Moreover, we are also interested in partial orderings, that is two objects may be *uncomparable* w.r.t to an ordering (neither $>$ nor $<$). That's why here an ordering function over a type t will be any function of type $t \to t \to result$ where *result* is either *equivalent*, *greater_than*, *less_than* or *uncomparable*. Such a ordering function defines in fact both a (partial) quasi-ordering $\succeq$ and an associated (partial) strict ordering $\succ$:

- $x \succeq y$ if $order(x,y) =$ ' $>$ '*or*' $=$ ' ;
- $x \succ y$ if $order(x,y) =$ ' $>$ '.

The ordering function *order* must be design in such a way that :

- $\succeq$ is reflexive and transitive : for all x, $x \succeq x$; and for all x, y and z, $x \succeq y$ and $y \succeq z$ imply $x \succeq z$;
- $\succ$ is transitive : for all x, y and z, $x \succ y$ and $y \succ z$ imply $x \succ z$;
- if $x \succeq y$ then $y \preceq x$;
- if $x \succ y$ then $y \prec x$;
- if $x \succ y$ then $x \succeq y$;
- ...

Moreover, we will be interested mainly an *term orderings*, that is orderings defined over sets of terms with variables. In that case, one is interested in having *stable* orderings : if $s \succeq t$ then $s\sigma \succeq t\sigma$ for all substitutions σ (and the same for $\succ$). In that situation, more complicated things may happen, for example let us assume that a is the smallest constant of a signature, then one would like to have $x \succeq a$ for any variable x, but in fact for the substitution $\sigma = \{x \mapsto a\}$ we would get $x\sigma \simeq a$ and for any other $x\sigma \succ a$. So what should return the function *order* on (x,a) ? ' $>$ ' or ' $=$ ' would be wrong so the only correct answer would be 'uncomparable'. These is not satisfactory at all, so we introduce two additional possible answers for an ordering function : *greater or equivalent* and *less or equivalent*.

Finally, the type of comparison is given by the following

```
type comparison_result =
  | Equivalent
  | Greater_than
  | Less_than
  | Greater_or_equivalent
  | Less_or_equivalent
  | Uncomparable
```

(*string_of_comparison_result r*) returns a string for displaying *r*

```
val string_of_comparison_result : comparison_result → string
```

```
type 'a ordering = 'a → 'a → comparison_result
```

```
val is_greater_or_equal : comparison_result → bool
```

```
val is_less_or_equal : comparison_result → bool
```

(*greater_than_all o x l*) returns true if for all *y* in list *l*, *x* is greater than *y* for *o*

```
val greater_than_all :
  'a ordering → 'a → 'a list → bool
```

(*exists_greater_or_equal o l y*) returns true if there exists *x* in list *l* such that *x* is greater than or equal to *y* for *o*

```
val exists_greater_or_equal :
  'a ordering → 'a list → 'a → bool
```

val *forall_exists_greater* :
 'a ordering $\rightarrow$ *'a list* $\rightarrow$ *'a list* $\rightarrow$ *bool*

val *remove_equivalent_elements* :
 'a ordering $\rightarrow$ *'a list* $\rightarrow$ *'a list* $\rightarrow$ (*'a list* $\times$ *'a list*)

General ordering functionals

Products of orderings
(*lexicographic_extension o*) where o is an ordering function over a type t, is the ordering function o' over lists of t by lexicographic use of o, that is $(x_1,\ldots,x_n) \succ' (y_1,\ldots,y_n)$ if there is a k s.t. $x_1 \succeq y_1, \ldots x_{k-1} \succeq y_{k-1}$ and $x_k \succ y_k$. Raises *Invalid_argument* of the two lists do not have the same length.

(*lexicographic_extension_of_orderings l*) where l is a list of ordering functions $o_1,\ldots,o_n$ all over the same type t, is the ordering function over t given by $x \succ y$ if there is a k s.t. $x \succeq_1 y, \ldots x \succeq_{k-1} y$ and $x \succ_k y$

val *lexicographic_extension* : (*'a ordering*) $\rightarrow$ (*'a list ordering*)

val *lexicographic_extension_of_orderings* : (*'a ordering*) *list* $\rightarrow$ (*'a ordering*)

(*multiset_extension o*) where o is an ordering function over a type t, is the ordering function o' over lists of t by multiset use of o.

val *multiset_extension* : (*'a ordering*) $\rightarrow$ (*'a list ordering*)

Chapter 6

Library `dioph`

This is a dioph library, to be documented by Evelyne.

Chapter 7

Library `Integer_solver`

This library provides methods for solving integer constraints : decision of validity/satisfiability of arbitrary first-order linear formulas, solving arbitray quantifer-free formulas over finite domains, etc.

7.1 Module *I_solve*

i_solve prend comme arguments

- un vecteur d'entiers $[|v0; v0+v1; ...; v0+v1+...+vk|]$ où $v0$ représente le nombre de variables non instanciées, et vi, $i > 0$, le nombre de variables instanciées dans la theorie Ti,
- un systeme d'equations ligne par ligne ;
- une liste qui correspond a $pe.edge$

et retourne un couple de vecteurs de solutions tel que: le premier element contient les solutions homogenes, i.e. qui valent 0 sur les variables instanciees, le second les autres solutions, sachant que les composantes associees a des variables instanciees ne peuvent prendre que les valeurs 0 et 1, et que de plus, si une composante associee a une variable instanciee dans la theorie Ti vaut 1, alors toutes les composantes associees a des variables instanciees dans une theorie Tj , i <> j, valent 0, Ti et Tj etant toutes deux des theories REGULIERES, COLLAPSE-FREE.

On va essayer autant que faire se peut de ne pas instancier des variables par une valeur contenant une variable marquee interdite.

Si des identifications de variables instanciees sont necessaires, on aura *AVEC_IDENT*, sinon *SANS_IDENT*.

i_solve_modulo fonctionne comme *i_solve*, avec un premier argument supplementaire, n, qui est l'entier modulo lequel on travaille. Au niveau de l'unification, ca correspond a l'unification ACUN(n), ou N(n) est la nilpotence d'ordre n: $x^n = 1$.

type *identifications_for_unification* =
    Without_identifications of (*int array array*) × (*int array array*)
  | *With_identifications* of (*int array array*) × (*int array array*)
  | *No_sol*

Careful extension of the functions / and mod, quotient and reminder of the integer division over the negative and positive integers in such a way that
ocwbegindcode $p = (p \backslash mathop\{zquo\}\ q) \times q + (p \backslash mathop\{zmod\}\ q), \backslash$ where $0 <= (p\,zmod\,q) < abs(q)$.

val *zquo* : $int \to int \to int$
val *zmod* : $int \to int \to int$

$(divides\ c\ n\ v)$ checks whether the integer c divides the nth first elements of the array of integers v.

val *divides* : $int \to int \to int\ array \to bool$

val *i_solve* :
  $int\ array \to int\ array\ array \to (int \times int)\ list \to$
    identifications_for_unification

7.2 Module *I_solve_modulo*

open *I_solve*

i_solve_modulo prend comme arguments

- un entier n modulo lequel on travaille Au niveau de l'unification, ca correspond a l'unification ACUN(n), ou N(n) est la nilpotence d'ordre n: $x^n = 1$.
- un vecteur d'entiers $[|v0; v0+v1; ...; v0+v1+...+vk|]$ où $v0$ represente le nombre de variables non instanciees, et vi, i>0, le nombre de variables instanciees dans la theorie Ti,
- un systeme d'equations ligne par ligne
- une liste qui correspond a $pe.edge$

et retourne un couple de vecteurs de solutions tel que: le premier element contient les solutions homogenes, i.e. qui valent 0 sur les variables instanciees, le second les autres solutions, sachant que les composantes associees a des variables instanciees ne peuvent prendre que les valeurs 0 et 1, et que de plus, si une composante associee a une variable instanciee dans la theorie Ti vaut 1, alors toutes les composantes associees a des variables instanciees dans une theorie Tj , i <> j, valent 0, Ti et Tj etant toutes deux des theories REGULIERES, COLLAPSE-FREE.

On va essayer autant que faire se peut de ne pas instancier des variables par une valeur contenant une variable marquee interdite.

Si des identifications de variables instanciees sont necessaires, on aura *AVEC_IDENT*, sinon *SANS_IDENT*.

val *i_solve_modulo* :
  int → *int array* → *int array array* → (*int* × *int*) *list* →
    identifications_for_unification

7.3 Module *Abstract_constraint*

type *expr* =
  | *Cte* of *Numbers.t*
  | *Var* of *string*
  | *Plus* of *expr* × *expr*
  | *Mult* of *expr* × *expr*
  | *Sub* of *expr* × *expr*
  | *Minus* of *expr*
  | *Quotient* of *expr* × *expr*

val *plus* : *expr* → *expr* → *expr*
val *mult* : *expr* → *expr* → *expr*
val *minus* : *expr* → *expr*
val *power* : *expr* → *int* → *expr*

type *formula* =
  | *True*
  | *False*
  | *Comp* of *expr* × *string* × *expr* (∗ <,>,=,>=,<= or <> or | ∗)
  | *And* of *formula list* (∗ length at least 2 ∗)
  | *Or* of *formula list* (∗ length at least 2 ∗)
  | *Neg* of *formula*
  | *Implies* of *formula* × *formula*
  | *Equiv* of *formula* × *formula*
  | *Exists* of *string list* × *formula* (∗ length at least 1 ∗)
  | *Forall* of *string list* × *formula* (∗ length at least 1 ∗)

val *divisible* : *expr* → *expr* → *formula* val *neg* : *formula* → *formula*

val *exists* : *string list* → *formula* → *formula*

val *forall* : *string list* → *formula* → *formula*

val *conj* : *formula* → *formula* → *formula*

val *conj_all* : *formula list* → *formula*

val *disj* : *formula* → *formula* → *formula*

val *disj_all* : *formula list* → *formula*

val *free_vars* : *formula* → *string list*

val *print_formula* : *formula* → *unit*

val *print_expr* : *expr* → *unit*

Rename all variables in a formula.

val *rename_formula* : (*string*×*string*) *list* → *formula* → *formula*

Rename all variables in an expression.

val *rename_expr* : (*string*×*string*) *list* → *expr* → *expr*

build_renaming l builds an association list from elements of *l* to fresh strings, that is never used before for renaming strings.

val *build_renaming* : *string list* → (*string*×*string*) *list*

7.4 Module *Linear_constraints*

7.4.1 Abstract data type for variables

variables are abstract, may only be built with *make_var*.

type *var_id*

val *make_var* : *string* → *var_id*

Same as *make_var* but the generated name is prepended with unique number.

val *make_var_uniq_name* : *string* → *var_id*

val *var_name* : *var_id* → *string*

7.4.2 Abstract data type for expressions

zero is 0, *one* is 1, (*cte n*) is the constant n, (*var v*) is the variable v, (*add e1 e2*) is e_1+e_2, (*sub e1 e2*) is e_1-e_2, (*minus e*) is $-e$, (*times n e*) is ne.

type *expr*

val *zero* : *expr*

val *one* : *expr*

val *minus_one* : *expr*

val *cte* : *Numbers.t* → *expr*

val *var* : *var_id* → *expr*

val *add_cte* : *expr* → *Numbers.t* → *expr* val *add* : *expr* → *expr* → *expr*

val *sub* : *expr* → *expr* → *expr*

val *minus* : *expr* → *expr*

val *times* : *Numbers.t* → *expr* → *expr*

val *div* : *expr* → *Numbers.t* → *expr*

val *is_cte* : *expr* → *bool*

val *val_of_cte* : *expr* → *Numbers.t*

val *get_coef* : *expr* → *var_id* → *Numbers.t*

val *remove_coef* : *expr* → *var_id* → *expr*

common_denominator e returns the least common multiple of denominators in e.

val *common_denominator* : *expr* → *Numbers.t*

module *VarMap* : *Inttagmap.IntTagMapModule*
with type *'a key* = *var_id*

type *substitution* = (*unit*,*expr*) *VarMap.t*

7.4.3 Abstract data type for formulas

Formulas are hashconsed for efficiency. All formulas should be build using given constructor below.

The internal structure is exported for pattern-matching purpose only.

type *formula* = *formula_struct Hcons.hash_consed*

and *formula_struct* =
| *True*
| *False*
| *Null* of *expr*
| *PositiveOrNull* of *expr*
| *Divisible* of *expr* × *Numbers.t*
| *And* of *substitution* × *formula_struct Ptset.t*
| *Or* of *formula_struct Ptset.t*
| *Not* of *formula*
| *Implies* of *formula* × *formula*
| *Equiv* of *formula* × *formula*
| *Exists* of *var_id list* × *formula*
| *Forall* of *var_id list* × *formula*

7.4.4 Atomic formulas

(*eq e1 e2*) is $e_1 = e_2$, (*ne e1 e2*) is $e_1 \neq e_2$, (*ge e1 e2*) is $e_1 \geq e_2$, (*gt e1 e2*) is $e_1 > e_2$, (*le e1 e2*) is $e_1 \leq e_2$, (*lt e1 e2*) is $e_1 < e_2$.

(*divides n e*) is n divides e, where n should be non null.

val *true_formula* : *formula*

val *false_formula* : *formula*

val *null* : *expr* → *formula*

val *positive* : *expr* → *formula*

val *positive_or_null* : *expr* → *formula*

val *negative* : *expr* → *formula*

val *eq* : *expr* → *expr* → *formula*

val *ne* : *expr* → *expr* → *formula*

val *ge* : *expr* → *expr* → *formula*

val *gt* : *expr* → *expr* → *formula*

val *le* : *expr* → *expr* → *formula*

val *lt* : *expr* → *expr* → *formula*

val *divides* : *Numbers.t* → *expr* → *formula*

val *is_atom* : *formula* → *bool*

7.4.5 Propositional connectors

hash_disj s (resp. *hash_conj*) builds the disjunction (resp conjunction) of formulas in s, assuming they are already hash consed.

these functions take in account the basic cases where s is empty or has only one elements.

val *hash_disj* : *formula_struct Ptset.t* → *formula*

val *hash_conj* : *substitution* → *formula_struct Ptset.t* → *formula*

neg f builds $\neg f$, *conj f1 f2* builds $f1 \wedge f2$, *disj f1 f2* builds $f1 \vee f2$. *conj_list l* builds the conjunct of all formulas in list *l*, *disj_list l* builds the disjunct of all formulas in list *l*. *implies f1 f2* builds *f1**rightarrow f2*, *equiv f1 f2* builds $f1 \leftrightarrow f2$.

val *neg* : *formula* → *formula*

val *conj* : *formula* → *formula* → *formula*

val *disj* : *formula* → *formula* → *formula*

val *conj_list* : *formula list* → *formula*

val *disj_list* : *formula list* → *formula*

val *implies* : *formula* → *formula* → *formula*

val *equiv* : *formula* → *formula* → *formula*

7.4.6 Iterators for conjunction and disjunction

map_conj_subst f sigma s computes

$$\bigwedge_{x=e\in[sigma]} f(x=e) \wedge \bigwedge_{h\in[s]} fs$$

map_conj_no_subst f sigma s computes

$$\bigwedge_{x=e\in[sigma]} x=e \wedge \bigwedge_{h\in[s]} fs$$

map_disj_set f s computes

$$\bigvee_{h\in[s]} fs$$

val *map_conj_subst* :
  (*formula* → *formula*) → *substitution* → *formula_struct Ptset.t* → *formula*

val *map_conj_no_subst* :
  (*formula* → *formula*) → *substitution* → *formula_struct Ptset.t* → *formula*

val *map_disj_set* :
  (*formula* → *formula*) → *formula_struct Ptset.t* → *formula*

7.4.7 Quantifiers

exists x f builds formula $\exists x.f$. *forall* x f builds formula $\forall x.f$.

val *exists* : *var_id* → *formula* → *formula*

val *forall* : *var_id* → *formula* → *formula*

val *exists_s* : *var_id list* → *formula* → *formula*

val *forall_s* : *var_id list* → *formula* → *formula*

7.4.8 Translation from abstract formulas

from_abstract_formula f builds the linear formula from the abstract formula f. returns also first the environment of free variables. Raises *Not_linear* of f is not linear.

exception *Not_linear*

type *var_env* = (*string* × *var_id*) *list*

val *from_abstract_expr* :
  var_env → *Abstract_constraint.expr* → *var_env* × *expr*

val *from_abstract_formula* :
  var_env → *Abstract_constraint.formula* → *var_env* × *formula*

7.4.9 Free vars of formula

free_vars f returns the list of free variables of f
  free_vars_env s f returns the union of s and the list of free variables of f

val *free_vars* : *formula* → *var_id list*

val *free_vars_env* : *var_id list* → *formula* → *var_id list*

val *var_occurs* : *var_id* → *formula* → *bool*

7.4.10 Instanciation, substitution

inst f x n returns the formula obtained by replace the variable x by number n in f.
  subst f x e returns the formula obtained by substituting the variable x by expression e in f.

type *instantiation* = (*unit*,*Numbers.t*) *VarMap.t*

val *inst1* : *var_id* → *Numbers.t* → *formula* → *formula*

val *subst1* : *var_id* → *expr* → *formula* → *formula* val *inst* : *instantiation* → *formula* → *formula*

val *subst* : *substitution* → *formula* → *formula*

7.4.11 Renaming

Type of the renaming tables

type *var_renaming* = (*unit*,*var_id*) *VarMap.t*

Rename all variables in a formula.

val *rename_formula* : *var_renaming* → *formula* → *formula*

Rename all variables in an expression.

val *rename_expr* : *var_renaming* → *expr* → *expr*

build_renaming l builds a fresh renaming for the variables of *l*.

val *build_renaming* : *var_id list* → *var_renaming*

7.4.12 Printing

(*print f*) prints the formula *f* on *Format.std_formatter* channel.

val *fprint_expr* : *Format.formatter* → *expr* → *unit*

val *print_expr* : *expr* → *unit*

val *fprint* : *Format.formatter* → *formula* → *unit*

val *print* : *formula* → *unit*

val *remove_denominators* : *formula* → *formula*

7.5 Module *Presburger*

open *Linear_constraints*

simplify f returns a formula *g* equivalent to *f* if interpreted in integers, performing geometric deductions over linear constraints.

For example :
$x > 0 and x < 0$ simplifies to false.
$x > 0 and x > 2$ simplifies to $x > 2$.
$x <= 0 or x >= 0$ simplifies to true.
$x <= 0 or x >= 1$ simplifies to true (since x is an integer).
assumes no denominators
val simplify : formula -> formula;;
assumes no denominators

val *eliminate_quantifiers* : *formula* → *formula*

is_satisfiable f returns true whenever *f* is satisfiable, i.e. is true for some integer values of its free variables. *is_valid f* returns true whenever *f* is valid, i.e. is true for each integer values of its free variables. *has_finitely_many_solutions f* is true whenever f is true for finitely many number of values of its free variables.

val *is_satisfiable* : *formula* → *bool*

val *is_valid* : *formula* → *bool*

val *has_finitely_many_solutions* : *formula* → *bool*

get_all_solutions f returns all solutions of f. Raises *Infinite* if there are infinitely many of them.

exception *Infinite*

val *get_all_solutions* : *formula* → (*var_id* × *Numbers.t*) *list list*

7.6 Module *Finite_domains*

This module defines finite domains constraints over integers, and allows to solve such constraints for given intervals to search for variables.

7.6.1 Finite domain variables

val *verbose* : *int ref*

type *fd_var_id*

val *string_of_fd_var_id* : *fd_var_id* → *string*

val *print_fd_var_id* : *fd_var_id* → *unit*

val *new_fd_var_id* : *unit* → *fd_var_id*

val *fd_var_id_of_string* : *string* → *fd_var_id*

```
module Fd_var_ord :
  sig
    type t = fd_var_id
    val compare : fd_var_id → fd_var_id → int
  end
```

module *Fd_var_map* : *Ordered_maps.OrderedMap* with type *'a key* = *fd_var_id*

module *Fd_var_set* : *Ordered_sets.OrderedSet* with type *'a elt* = *fd_var_id*

7.6.2 The domains expressions

Domain expressions are made from classical arithmetical operations, and min(x) and max(x) for any domain variable x.

There are two version of division depending whether the result is rounded to the floor or the ceiling. The same for square root and arbitrary nth root.

```
type domain_operation =
    Add | Sub | Mult | Div_floor | Div_ceil
```

```
type domain_expression =
    Const of Numbers.t
  | Min of fd_var_id
  | Max of fd_var_id
  | Oper of domain_expression × domain_operation × domain_expression
  | Sqrt_floor of domain_expression
  | Sqrt_ceil of domain_expression
  | Power of domain_expression × int
  | Root_floor of int × domain_expression
  | Root_ceil of int × domain_expression
```

7.6.3 Finite domain constraints

a finite domain constraint is always of the form $x \in [min..max]$ where *min* and *max* are two domain expressions

```
type fd_constraint =
  {
    var : fd_var_id;
    min : domain_expression;
    max : domain_expression;
  }

val print_fd_constraint : fd_constraint → unit
```

7.6.4 Solving constraints

(*finite_domain_solve* M C) returns a solution of the constraints C in the domain $0..M(x)$ for each variable x. Raises exception *Finite_domains.No_solution* if no solution exists.

```
exception No_solution

exception Time_out

val time_out : bool ref

val time_out_signal_handler : 'a → unit

val finite_domain_solve :
  (fd_var_id, Numbers.t) Fd_var_map.t → fd_constraint list
    → (fd_var_id, Numbers.t) Fd_var_map.t

val print_fd_solution : (fd_var_id, Numbers.t) Fd_var_map.t → unit
```

7.7 Module *Non_linear_solving*

```
val verbose : int ref

type constraint_op = Nul | Pos_or_nul

module NumRing : Polynomials.RingType with type coef = Numbers.t

module Fd_polynomials :
  Polynomials.PolynomialType with
                                  module Base_ring = NumRing
                                and
                                  type variable = Finite_domains.fd_var_id
                                and
                                  module Var_map = Finite_domains.Fd_var_map

val print_polynomial : Fd_polynomials.poly → unit

type non_linear_constraint =
    {
      expr : Fd_polynomials.poly;
      cond : constraint_op;
    }

val print_constraint : non_linear_constraint → unit

val print_non_linear_constraints : non_linear_constraint list → unit
```

val *from_abstract_formula* :
Abstract_constraint.formula $\rightarrow$ *non_linear_constraint list*

(*non_linear_solve* M C) returns a solution of the constraints C in the domain $0..M$ for each variable. Raises exception *Not_found* if no solution exists.

val *time_limit* : *float ref*

val *non_linear_solve* :
Numbers.t $\rightarrow$ *non_linear_constraint list*
$\rightarrow$ (*Finite_domains.fd_var_id*, *Numbers.t*) *Finite_domains.Fd_var_map.t*

temporaire

val *translate_constraints* :
(*Finite_domains.fd_var_id*, *Numbers.t*) *Finite_domains.Fd_var_map.t* $\rightarrow$
non_linear_constraint list $\rightarrow$
Finite_domains.fd_constraint list $\times$
(*Finite_domains.fd_var_id*, *Numbers.t*) *Finite_domains.Fd_var_map.t*

7.8 Module *Poly_lexer*

exception *Invalid_char* of *string*

val *token* : *Lexing.lexbuf* $\rightarrow$ *Poly_parser.token*

7.9 Lexer *Poly_lexer*

```
{
  open Poly_parser
  exception Invalid_char of string
}
rule token = parse
  | [' ' '\t' '\n'] { token lexbuf }
  | "and" | "/\\" { AND }
  | "or" | "\\/" { OR }
  | "not" | "~" { NOT }
  | "exists" { EXISTS }
  | "forall" { FORALL }
  | "true" { TRUE }
  | "false" { FALSE }
  | "implies" | "->" { IMPLIES }
  | "equiv" | "<->" { EQUIV }
  | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_' '\'' '@' '~']*
                                  { IDENT(Lexing.lexeme lexbuf) }
  | ';' { SEMICOLON }
  | ',' { COMMA }
  | '+' { PLUS }
  | '/' { DIV }
  | '|' { VERTICALBAR }
  | '-' { MINUS }
  | '.' { MULT }
  | '^' { EXP }
  | "="|">"|">="|"<"|"<="|"<>"
```

{ COMP(Lexing.lexeme lexbuf) }
| ['0'–'9']$^+$ { INT (Numbers.from_string (Lexing.lexeme lexbuf)) }
| '(' { PARGAUCHE }
| ')' { PARDROITE }
| _ { raise (Invalid_char (Lexing.lexeme lexbuf)) }
| eof { EOF }

7.10 Parser *Poly_parser*

Header

open *Abstract_constraint*

Token declarations

%token <*string*> *IDENT*
%token *PARGAUCHE PARDROITE SEMICOLON COMMA EOF*
%token *TRUE FALSE AND OR NOT IMPLIES EQUIV EXISTS FORALL*
%token *PLUS MINUS EXP MULT DIV VERTICALBAR*
%token <*string*> *COMP*
%token <*Numbers.t*> *INT*

%start *constraint_entry*
%type <*Abstract_constraint.formula*> *constraint_entry*

%start *expr*
%type <*Abstract_constraint.expr*> *expr*

%nonassoc *EXISTS FORALL*
%right *IMPLIES EQUIV*
%left *OR*
%left *AND*
%nonassoc *NOT*
%left *PLUS MINUS*
%left *MULT DIV*
%nonassoc *UMINUS*
%right *EXP*

Grammar rules

constraint_entry ::=
| *formula EOF* { $1 }

formula ::=
| *formula AND formula*
    { *conj* \$1 \$3 }
| *formula OR formula*
    { *disj* \$1 \$3 }
| *NOT formula*
    { *Neg*(\$2) }
| *formula IMPLIES formula*
    { *Implies*(\$1,\$3) }
| *formula EQUIV formula*
    { *Equiv*(\$1,\$3) }
| *EXISTS id_list COMMA formula*
    { *Exists*(\$2,\$4) }
| *FORALL id_list COMMA formula*
    { *Forall*(\$2,\$4) }
| *expr COMP expr*
    { *Comp*(\$1,\$2,\$3) }
| *expr VERTICALBAR expr*
    { *Comp*(\$1,"|",\$3) }
| *expr COMP expr COMP expr*
    { *conj* (*Comp*(\$1,\$2,\$3)) (*Comp*(\$3,\$4,\$5)) }
| *expr COMP expr COMP expr COMP expr*
    { *conj* (*conj* (*Comp*(\$1,\$2,\$3)) (*Comp*(\$3,\$4,\$5))) (*Comp*(\$5,\$6,\$7)) }
| *PARGAUCHE formula PARDROITE*
    { \$2 }
| *TRUE*
    { *True* }
| *FALSE*
    { *False* }

id_list ::=
| *IDENT*
    { [\$1] }
| *IDENT id_list*
    { \$1::\$2 }

expr ::=
| *IDENT*
    { *Var*(\$1) }
| *INT*
    { *Cte*(\$1) }
| *PARGAUCHE expr PARDROITE*
    { \$2 }
| *expr PLUS expr*
    { *Plus*(\$1,\$3) }
| *expr MINUS expr*
    { *Sub*(\$1,\$3) }
| *MINUS expr* %prec *UMINUS*
    { *Minus*(\$2) }
| *expr MULT expr*
    { *Mult*(\$1,\$3) }
| *expr DIV expr*
    { *Quotient*(\$1,\$3) }
| *expr EXP INT*

```
{ try
    power $1 (Numbers.to_int $3)
  with
    Invalid_argument("Numbers.to_int") →
      failwith "Exponent␣too␣large"
}
```

7.11 Module *Poly_syntax*

exception *Syntax_error* of *string*

val *constraint_of_string* : *string* → *Abstract_constraint.formula*

val *expr_of_string* : *string* → *Abstract_constraint.expr*

Chapter 8

Library `words`

This library provides all definitions and syntactic methods for string rewriting.

8.1 Module *String_rewriting*

This module defines functions for performing string rewriting of words.

8.1.1 String rewriting systems

A string rewriting system is defined by a set of pairs of words (left-hand side, right-hand side), represented by a list.

type *'symbol* rewrite_rule = (*'symbol* Words.word × *'symbol* Words.word)

type *'symbol* srs = (*'symbol* Words.word × *'symbol* Words.word) *list*

(normalize S w) returns the normal form of w w.r.t the SRS S by the rightmost strategy. Warning! If the righmost reduction of w is infinite, this function loops.

val *normalize* :
  'symbol srs → *'symbol* Words.word → *'symbol* Words.word

8.1.2 Efficient normalization

The complexity of the former normalization function increases linearly in the number of rules. The latter increases linearly in the maximal length of the left-hand sides of rules.

compiled_srs is an abstract data type that allows efficient matching (discrimination net)

type *'symbol* compiled_srs

(compiled_normalize S w) returns the normal form of w w.r.t the SRS S by the rightmost strategy. Warning! If the righmost reduction of w is infinite, this function loops.

val *compiled_normalize* :
  'symbol compiled_srs → *'symbol* Words.word → *'symbol* Words.word

(compile_srs S) returns a discrimination net equivalent to S, the be used in the previous function.

val *compile_srs* :
  'symbol srs → *'symbol* compiled_srs

val *is_nf_compiled* :
  'symbol compiled_srs → *'symbol* Words.word → *bool*

8.1.3 Printing SRSs

(*print sigma srs*) prints the SRS *srs*, its letters being printed according to *sigma#string_of_symbol*. The letters are separated by spaces. Beware that the printing is done using printing functions of the module *Format*.

(*pretty_print sigma srs*) does the same, but factorizing consecutive occurences of the same letter.

val *print* :
  'symbol #String_signatures.word_signature → *'symbol* srs → *unit*

val *pretty_print* :
  'symbol #String_signatures.word_signature → *'symbol* srs → *unit*

8.2 Module *Local_confluence*

Tests if the given string rewriting system is locally confluent. Note that the result is not correct if the *srs* is not inter-reduced.

val *is_localy_confluent* :
  'symbol String_rewriting.srs → *bool*

8.3 Module *String_signatures*

This module defines the class of signatures for words

8.3.1 The parameterized signature type

class type [*'a*] *word_signature* =
  object
    method *string_of_symbol* : *'a* → *string*
  end

(*from_list l*) builds the finite string signature made from symbols in *l*

type *symbol_id*

class type *user_word_signature* =
  object
    inherit [*symbol_id*] *word_signature*
    method *string_of_symbol* : *symbol_id* → *string*

(*symbol_of_string s*) returns the symbol represented by *s*, raises *Not_found* if no such symbol exists in the signature

    method *symbol_of_string* : *string* → *symbol_id*
  end

val *from_list* : *string list* → *user_word_signature*

val *from_string* : *string* → *user_word_signature*

8.4 Module *String_unification*

Computes the solutions of *l1 w1* = *w2 l2* with *w1* and *w2* not empty

val *superpose* :
  (*'symbol* Words.word × *'symbol* Words.word)
  → (*'symbol* Words.word × *'symbol* Words.word)
    → (*'symbol* Words.word × *'symbol* Words.word) *list*

```
val unify :
  'symbol Words.word → 'symbol Words.word →
    (('symbol Words.word × 'symbol Words.word) × ('symbol Words.word × 'symbol Words.word)) list
```

8.5 Module *User_words*

(*word_of_string sigma s*) returns the term over the signature *sigma*, read in the string *s*

```
val word_of_string :
  String_signatures.user_word_signature → string
    → String_signatures.symbol_id Words.word
```

(*srs_of_string sigma s*) returns the string rewriting system over the signature *sigma*, read in the string *s*

```
val srs_of_string :
  String_signatures.user_word_signature → string
    → String_signatures.symbol_id String_rewriting.srs
```

(*word_prec_of_string sigma s*) returns the precedence over the signature *sigma*, read in the string *s*

```
val word_prec_of_string :
  String_signatures.user_word_signature → string
    → String_signatures.symbol_id Orderings_generalities.ordering
```

8.6 Module *Word_lexer*

```
val token : Lexing.lexbuf → Word_parser.token
```

8.7 Lexer *Word_lexer*

```
{

  open Word_parser
  open Word_syntax
  open String_signatures

  let ident s =
    try
      IDENT(!current_signature#symbol_of_string s)
    with Not_found →
      try
        INT(int_of_string s)
      with
          _ →
            raise (Syntax_error ("Undeclared symbol '" ^ s ^ "'"))


}
```

```
rule token = parse
    [' ''\t''\n'] { token lexbuf }
  | "(*" { comment lexbuf; token lexbuf }
  | ';' { SEMICOLON }
  | ',' { COMMA }
  | "->" { ARROW }
  | '^' { POWER }
  | '(' { LPAR }
  | ')' { RPAR }
  | '<' { LT }
  | '>' { GT }
  | '=' { EQ }
  | ['a'-'z''A'-'Z''0'-'9']['a'-'z''A'-'Z''0'-'9''_''\'']*
  | ['+''.''&''*''-''/''!''?''@''~''#'] ('_'(['a'-'z''A'-'Z''0'-'9']+))*
                            { ident(Lexing.lexeme lexbuf) }
  | _ { raise (Syntax_error ("invalid char '" ^ (Lexing.lexeme lexbuf) ^ "'")) }
  | eof { EOF }

and comment = parse
    "*)" { () }
  | "(*" { comment lexbuf; comment lexbuf }
  | _ { comment lexbuf }
  | eof { raise (Syntax_error "unterminated comment") }
```

8.8 Parser *Word_parser*

Header

```
open Words
open Word_syntax
open String_signatures
open Finite_orderings

let word_power = Listutils.power
```

Token declarations

```
%token <String_signatures.symbol_id> IDENT
%token SEMICOLON ARROW LPAR RPAR COMMA
%token POWER EQ GT LT
%token <int> INT
%token EOF

%start word_eof
%type <String_signatures.symbol_id Words.word> word_eof

%start rule_set_eof
%type <String_signatures.symbol_id String_rewriting.srs> rule_set_eof

%start precedence_eof
%type <String_signatures.symbol_id Orderings_generalities.ordering> precedence_eof
```

Grammar rules

```
word_eof ::=
  word EOF { $1 }


word ::=
  /* epsilon */
    { [] }
| factor word
    { $1 @ $2 }


factor ::=
  IDENT
    { [$1] }
| factor POWER expo
    { word_power $1 $3 }
| LPAR word RPAR
    { $2 }


expo ::=
  INT { $1 }
| IDENT
      {
        try
          int_of_string(!current_signature#string_of_symbol $1)
        with
            _ →
              raise (Syntax_error "invalid␣exponent")
      }


rule_set_eof ::= rule_set EOF { $1 }

rule_set ::=
  /* epsilon */
    { [] }
| rule
    { [$1] }
| rule SEMICOLON rule_set
    { $1::$3 }


rule ::=
  word ARROW word { ($1,$3) }


precedence_eof ::= precedence EOF
  {
    let order = identity_ordering (Pervasives.compare : symbol_id → symbol_id → int)
    in
    let order = List.fold_left add_list order $1
    in compare order
  }
```

precedence ::=
  /∗ epsilon ∗/ { [] }
| *ordered_list* { [$1] }
| *ordered_list COMMA precedence* { $1::$3 }

ordered_list ::=
  IDENT { *One*($1) }
| *IDENT EQ ordered_list* { *Eq*($1,$3) }
| *IDENT GT ordered_list* { *Gt*($1,$3) }
| *IDENT LT ordered_list* { *Lt*($1,$3) }

8.9 Module *Word_orderings*

open *Orderings_generalities*

val *length_lex* : *'symbol* ordering → *'symbol* Words.word ordering

val *multi_lex* : *'symbol* ordering → *'symbol* Words.word ordering

8.10 Module *Word_syntax*

This module provides the environment for words and string rewrite rules syntactic analysis.
exception raised when a syntax error occurs

exception *Syntax_error* of *string*

current_signature is the signature to be used for parsing symbols

val *current_signature* : *String_signatures.user_word_signature ref*

8.11 Module *Srs_completion*

Completion of string-rewriting systems

val *verbose* : *int ref*

open *String_signatures*

val *complete_srs_std_strategy* :
  'symbol #word_signature →
  'symbol Words.word Orderings_generalities.ordering →
  'symbol String_rewriting.srs →
  'symbol String_rewriting.srs

val *complete_srs_alphabetic_order_std_strategy* :
  symbol_id #word_signature →
  symbol_id String_rewriting.srs →
  symbol_id String_rewriting.srs

8.12 Module *Words*

This module defines the strings (or words) over a string signature

8.12.1 type for words

A word over an alphabet in explicitly represented by a list of elements of this alphabet.

type *'symbol word* = *'symbol list*

val *length* : *'symbol word* → *int*

8.12.2 Printing words

(*print sigma w*) prints the word *w*, its letters being printed according to *sigma#string_of_symbol*. The letters are separated by spaces. Beware that the printing is done using printing functions of the module *Format*.

(*pretty_print sigma w*) does the same, but factorizing consecutive occurences of the same letter.

val *print* :
'symbol #String_signatures.word_signature → *'symbol word* → *unit*

val *pretty_print* :
'symbol #String_signatures.word_signature → *'symbol word* → *unit*

8.13 Module *Parameterized_signatures*

This module defines the class of parameters.

8.13.1 The parameters type

val *verbose* : *int ref*

val *debug* : ?*f* : (*string* → *unit*) → *int* → *string* → *unit*

class type *parameter_c* =
object
method *parameters* : *Linear_constraints.var_env*
method *print* : *unit* → *unit*
end

This module defines the class of parameterized signatures for parameterized words.

8.13.2 The parameterized signature type

The type of elements in the signature. For example $a_{i+n}|_i i \leq n$ is represented by {*sig_symbol* = `"a"`; *sig_index* = [`"i + n"`]; *sig_constr* = `"i <= n"`}

type *element* =
{ *sig_symbol* : *string*;
sig_index : *Linear_constraints.expr list*;
sig_constr : *Linear_constraints.formula*;
sig_env : *Linear_constraints.var_env*
}

The general signature class.

class type *parameterized_signature* =
object
val *psig* : (*string*,*element*) *Hashtbl.t*
val *parameters* : *parameter_c*

decoration_of_symbol f returns the index of the full description of the symbol *f* and its associated constraint.

```
    method decoration_of_symbol : string → element
    method to_list : unit → element list
    method parameters : parameter_c
    method print : unit → unit
    method print_element : element → unit
  end
```

8.14 Module *User_parameterized_signatures*

This module defines the class of parameterized signatures for parameterized words

8.14.1 The user parameterized signature type

```
class type user_parameterized_signature =
  object
    inherit Parameterized_signatures.parameterized_signature
  end
```

Instantiates a parameterized signature into a word signature.

```
val instantiate_signature :
  user_parameterized_signature →
    (string → Numbers.t) →
      String_signatures.user_word_signature
```

from_string p s returns the signature with parameters p and definition s.

```
val from_string :
  Parameterized_signatures.parameter_c →
    string →
      user_parameterized_signature
```

```
val parameters_from_string :
  string →
    Parameterized_signatures.parameter_c
```

8.15 Parser *Parameterized_signatures_parser*

Header

```
  open Parameterized_signatures
  open Parameterized_signatures_syntax
```

Token declarations

```
%token <string> FORMULA
%token SEMICOLON
%token PIPE
%token <string> IDENT
%token EOF
%token POWER FP LPAR RPAR ARROW
```

%start *signature_eof*
%type <*Parameterized_signatures_syntax.signature*> *signature_eof*

%start *cword_eof*
%type <*Parameterized_signatures_syntax.constrained_word*> *cword_eof*

%start *rules_eof*
%type <*Parameterized_signatures_syntax.rules*> *rules_eof*

Grammar rules

This part deals with signatures

signature_eof ::=
  | *EOF* {[]}
  | *signature EOF* {$1}

signature ::=
  | *elt* {[$1]}
  | *elt SEMICOLON* {[$1]}
  | *elt SEMICOLON signature* {$1::$3}

elt ::=
  | *IDENT expr_l*
    {$1,$2,*Abstract_constraint.True*}
  | *IDENT expr_l PIPE constr*
    {$1,$2,$4}

This part deals with words.

cword_eof ::=
  | *cword EOF* {$1}

cword ::=
  | *word*
    { ($1, *Abstract_constraint.True*) }
  | *word PIPE constr*
    { ($1, $3) }

word ::=
  | /* epsilon */
    { [] }
  | *factor word*
    { $1::$2 }
  | *simple_word word_no_simple*
    { *Simple*($1) :: $2 }

word_no_simple ::=
  | /* epsilon */
    { [] }
  | *factor word*
    { $1::$2 }

factor ::=
  | *letter POWER expr*
    { $Exp([\$1],\$3)$ }
  | *LPAR simple_word RPAR POWER expr*
    { $Exp(\$2,\$5)$ }
  | *FP FORMULA expr expr LPAR simple_word RPAR*
    { $Product(\$2,\$3,\$4,\$6)$ }
  | *FP FORMULA expr expr letter*
    { $Product(\$2,\$3,\$4,[\$5])$ }

simple_word ::=
  | *letter*
    { [$1] }
  | *simple_word letter*
    { $1@[$2] }

letter ::=
  | *IDENT expr_l*
    { $(\$1,\$2)$ }

This part deals with rules.

rules_eof ::=
  | *rules EOF*
    { $1 }

rules ::=
  | /* epsilon */
    { [] }
  | *rule*
    { [$1] }
  | *rule SEMICOLON rules*
    { $1 :: $3 }

rule ::=
  | *word ARROW word PIPE constr*
    { $(\$1,\$3,\$5)$ }
  | *word ARROW word*
    { $(\$1,\$3,Abstract\_constraint.True)$ }

This part deals with formulae and expressions.

expr_l ::=
  | /* epsilon */
    { [] }
  | *expr expr_l*
    { $1::$2 }

expr ::=
  | *FORMULA* {*Poly_syntax.expr_of_string* $1}

```
constr ::=
  | FORMULA {Poly_syntax.constraint_of_string $1}
```

8.16 Lexer *Parameterized_signatures_lexer*

```
{
  open Parameterized_signatures_parser
  open Parameterized_signatures_syntax
  let current_formula = Buffer.create 17
}
rule token = parse
    [' ' '\t' '\n' '\r'] { token lexbuf }
  | "(*" { comment lexbuf; token lexbuf }
  | ';' { SEMICOLON }
  | '|' { PIPE }
  | '^' { POWER }
  | "fp" { FP }
  | '(' { LPAR }
  | ')' { RPAR }
  | "->" { ARROW }
  | '{' { Buffer.clear current_formula;
                                  FORMULA (formula lexbuf) }
  | '}' {   raise (Syntax_error "unbalanced symbol \"}\"")}
  | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_' '\'' '@' '~']*
                                  { IDENT (Lexing.lexeme lexbuf) }
  | _ { raise (Syntax_error ("invalid char '" ^ (Lexing.lexeme lexbuf) ^ "'")) }
  | eof { EOF }
and comment = parse
    "*)" { () }
  | "(*" { comment lexbuf; comment lexbuf }
  | _ { comment lexbuf }
  | eof { raise (Syntax_error "unterminated comment") }
and formula = parse
  | "}" { Buffer.contents current_formula }
  | _ { Buffer.add_string
                                    current_formula (Lexing.lexeme lexbuf) ;
                                  formula lexbuf }
  | eof {   raise (Syntax_error "formulae must end with }") }
```

8.17 Module *Parameterized_signatures_syntax*

exception raised when a syntax error occurs.

```
exception Syntax_error of string

type expr = Abstract_constraint.expr

type expr_l = expr list

type letter = string × expr_l

type signature_element = string × expr_l × Abstract_constraint.formula
```

type *signature* $=$ *signature_element list*

type *simple_word* $=$ *letter list*

type *factor* $=$
| *Simple* of *simple_word*
| *Exp* of *simple_word* $\times$ *expr*
| *Product* of *string* $\times$ *expr* $\times$ *expr* $\times$ *simple_word*

type *word* $=$ *factor list*

type *constrained_word* $=$ *word* $\times$ *Abstract_constraint.formula*

type *rule* $=$ *word* $\times$ *word* $\times$ *Abstract_constraint.formula*

type *rules* $=$ *rule list*

Chapter 9

Library `matching`

This is a matching library, to be documented by Evelyne.

9.1 Module *Lazy_list*

type *'a frozen* =
  | *Freeze* of (*unit* → *'a*)
  | *Val* of *'a*

and *'a llist* =
  | *Nil*
  | *Cons* of *'a cell*

and *'a cell* =
    {
      head : *'a*;
      mutable *tail* : *'a llist frozen*;
    }

val *hdl* : *'a llist* → *'a*

val *mapl* : (*'a* → *'b*) → *'a llist* → *'b llist*

val *map* : (*'a* → *'b*) → *'a list* → *'b llist*

val *map_without_repetition* : (*'a* → *'b*) → *'a list* → *'b llist*
val *map2_without_repetition* : (*'a* → *'b* × *'b*) → *'a list* → *'b llist*
val *map12_without_repetition* : (*'a* → *'b* × (*'b* option)) → *'a list* → *'b llist*

val *lazy_append* : *'a llist* → *'a llist frozen* → *'a llist*

val *from_list* : *'a list* → *'a llist*

val *to_list* : *'a llist* → *'a list*

9.2 Module *Lazy_controle*

exception *No_solution*
exception *Not_appliable*

type *'pb disjunction* = *'pb* Lazy_list.llist

val *orelse* :
  (*'pb* → *'pb* disjunction) → (*'pb* → *'pb* disjunction) → *'pb* → *'pb* disjunction

val *repeat* :
  (*'pb* → *'pb* disjunction) → *'pb* → *'pb* disjunction

Chapter 10

Library `terms`

This library provides all definitions and syntactic methods for symbols, signatures, terms and rewrite rules.

10.1 Module *Signatures*

This module defines the class of first-order signatures, allowing commutative or associative-commutative symbols.

10.1.1 The signature class type

The signature class type is a very general definition of a signature: it is a arbitrary set (even infinite) equipped with an arity function and some others. The set is modelized here by a type parameter.

($arity$ f) returns the arity of the symbol f.

(is_ac f) returns true if f is an associative-commutative symbol.

($is\_commutative$ f) returns true if f is a commutative (but not associative) symbol.

(is_free f) returns true if f is neither commutative nor associative-commutative.

contains_ac_symbols is true there is at least one AC symbol an that signature.

contains_only_free_symbols is true there all symbols are free.

($string\_of\_symbol$ f) must return a printable representation of the symbol f.

($symbol\_fix$ f) returns a concrete value that tells if f must printed, in a term, as a prefix symbol, an infix symbol (like $+$ in $x+y$, or a postfix symbol (like ! in n!). The default fix value is infix for AC symbols are prefix for others.

```
type symbol_fix = Prefix | Infix | Postfix | Default
```

```
class type ['a] signature =
    object
      method arity : 'a → int
      method is_ac : 'a → bool
      method is_commutative : 'a → bool
      method is_free : 'a → bool
      method contains_ac_symbols : bool
      method contains_only_free_symbols : bool
      method string_of_symbol : 'a → string
      method symbol_fix : 'a → symbol_fix
    end
```

($symbol\_of\_string$ s) returns the symbol whose name is s. Raises exception *Not_found* is no symbol corresponds.

```
class type ['a] parseable_signature =
    object
      inherit ['a] signature
      method symbol_of_string : string → 'a
    end
```

For example, one may define the infinite signature made of the natural numbers seen as constant by saying : class *nat_signature* : object inherit [*int*] *signature* method *arity* $f = 0$ method *is_ac* f = false method *is_commutative* f = false method *is_free* f = true method *contains_ac_symbols* = false method *contains_only_free_symbols* = true method *string_of_symbol* f = *string_of_int* f method *symbol_fix* f = *Prefix* end

10.1.2 Finite symbol sets and maps

polymorphic set and map module for symbols, compared by the CAML polymorphic comparaison function.

module *SymbolOrd* : *Ordered_types.OrderedPolyType*
  with type *'a t* = *'a*

module *SymbolSet* : *Ordered_sets.OrderedSet*
  with type *'a elt* = *'a*

module *SymbolMap* : *Ordered_maps.OrderedMap*
  with type *'a key* = *'a*

class [*'a*] *default* : [*'a*] *parseable_signature*

10.2 Module *Signature_syntax*

type *symbol_theory* = *Free* | *Ac* | *Commutative*

exception *Syntax_error* of *string*

10.3 Module *Substitution*

open *Signatures*
open *Variables*
open *Gen_terms*

type *'symbol t* = (unit,*'symbol* term) *VarMap.t*

val *print* :
  'symbol #signature → *Variables.user_variables*
    → *'symbol t* → unit

(*apply_term* h t σ) computes $t\sigma$ and uses the hashconsing table of terms h

val *apply_term* : *'symbol* term → *'symbol t* → *'symbol* term

val *apply_sorted_term* :
  'symbol #signature → *'symbol* term →
    'symbol t → *'symbol* term

(*merge_subst* *subst1* *subst2*) merges substitutions *subst1* and *subst2*, that is put them together verifying that if *subst1* binds a variable x to a term t_1 and *subst2* binds the same variable to t_2, then $t_1 = t_2$

Raises Conflict if not.

This is standard substitution merging: equality of t_1 and t_2 is performed assuming that all symbols are free.

exception *Conflict*

val *merge_substs* : *'symbol t* → *'symbol t* → *'symbol t*

val *substitute* : *'symbol* #signature → *'symbol t* → *'symbol* term → *'symbol* term

val *apply_subst_to_eqs* :
  'symbol #signature → *'symbol t* →
    (*'symbol* term × *'symbol* term) *list* → (*'symbol* term × *'symbol* term) *list*

val *canonical_renaming* : *'symbol* term *list* → *'symbol t*

10.4 Module *Rewrite_rules*

```
open Signatures
open Gen_terms
```

Rewrite rules are used mainly in two different contexts:

- in the termination tool, for finding an ordering which ensures the termination of a system of rewrite rules,
- and for rewriting terms.

t is the type of rewriting rules used in the termination tool, defined by the lefthand side, the righthand side, and optionally the lhs and the rhs of the AC-extension.

```
type 'symbol t =
    {
      lhs : 'symbol term;
      rhs : 'symbol term;
      ext : ('symbol term × 'symbol term) option;
    }
```

```
exception Rule_with_a_var_lhs
```

(*make_rule sigma l r*) builds a rewrite rule over signature *sigma* with lefthand side *l* and righthand side *r*. Determines whether the rule needs to be AC-extended; in this case the hashconsing table *h* is used to build the terms of the extension.

May raise the exception *Rule_with_a_var_lhs* when this is the case!

```
val make_rule :
  'symbol #signature → 'symbol term → 'symbol term → 'symbol t
```

```
val print_rewrite_rule :
  'symbol #Signatures.signature → Variables.user_variables → 'symbol t → unit
```

```
val print_rewrite_rule_set :
  'symbol #Signatures.signature → Variables.user_variables → 'symbol t list → unit
```

```
val latex_print_rewrite_rule_set :
  out_channel → 'symbol #Signatures.signature → Variables.user_variables → 'symbol t list → unit
```

10.5 Module *Term_lexer*

```
exception Invalid_char of string
```

```
exception Unterminated_comment
```

```
val token : Lexing.lexbuf → Term_parser.token
```

10.6 Module *Signature_lexer*

```
exception Invalid_char of string
```

```
val token : Lexing.lexbuf → Signature_parser.token
```

10.7 Lexer *Term_lexer*

```
{

  open Term_parser
  open Term_syntax
  open Signatures
```

```
  exception Invalid_char of string
  exception Unterminated_comment

  let ident s =
    try
      match !parse_id s with
        | Postfix → POSTFIX_IDENT s
        | Infix → INFIX_IDENT s
        | Prefix → PREFIX_IDENT s
        | Default → PREFIX_IDENT s
    with
        Not_found →
          try
            let x = !current_variables#var_of_string s
            in VAR_IDENT x
          with
              Not_found →
                raise (Syntax_error ("undefined␣identifier␣"^s))

}

let letters = ['a'-'z''A'-'Z''0'-'9''_''\'']+
let symbols = ['^''+''.''&''*''-''/''!''?''@''~''#''|'':''%''$''<''=''>']+
rule token = parse
  | [' ''\r''\t''\n'] { token lexbuf }
  | ',' { COMMA }
  | ';' { SEMICOLON }
  | "->" { ARROW }
  | '(' { OPENPAR }
  | ')' { CLOSEPAR }
  | '<' { LT }
  | '>' { GT }
  | '=' { EQ }
  | "<=" { LE }
  | ">=" { GE }
  | "<>" { NE }
  | "mul" { MUL }
  | "lr_lex" { LR_LEX }
  | "rl_lex" { RL_LEX }
  | (letters | symbols) ( '_' (letters | symbols))*
                                  { ident(Lexing.lexeme lexbuf) }
  | "(*" { comment lexbuf; token lexbuf }
  | _ { raise (Invalid_char (Lexing.lexeme lexbuf)) }
  | eof { EOF }

and comment = parse
  | "*)" { () }
  | "(*" { comment lexbuf; comment lexbuf }
  | _ { comment lexbuf }
  | eof { raise Unterminated_comment }
```

10.8 Lexer *Signature_lexer*

```
{
  open Signature_parser
```

```
exception Invalid_char of string
```

hash table of keywords any number is appropriate in place of 17, a prime number is better

```
let keywords_table = let kt = ((Hashtbl.create 17) : (string,token) Hashtbl.t)
in List.iter
      (function (s,k) -> Hashtbl.add kt s k)
      [
        ("constant", KW_CONSTANT);
        ("unary", KW_UNARY);
        ("binary", KW_BINARY);
        ("infix", KW_INFIX);
        ("prefix", KW_PREFIX);
        ("postfix", KW_POSTFIX);
        ("commutative", KW_C);
        ("AC", KW_AC)
      ];
  kt

let keyword_or_ident s =
  try (Hashtbl.find keywords_table s)
  with Not_found -> IDENT s

}

let letters = ['a'-'z''A'-'Z''0'-'9''_''\'']+
let symbols = ['^''+''.''&''*''-''/''!''?''@''~''#''|'':''%''$''<''=''>']+
rule token = parse
    [' ''\r''\t''\n'] { token lexbuf }
  | ',' { COMMA }
  | ':' { COLON }
  | ';' { SEMICOLON }
  | ['0'-'9']+ { INT(Lexing.lexeme lexbuf) }
  | ['a'-'z''A'-'Z']+ { keyword_or_ident(Lexing.lexeme lexbuf) }
  | (letters | symbols) ( '_' (letters | symbols))*
                         { IDENT(Lexing.lexeme lexbuf) }
  | "(*" { comment lexbuf; token lexbuf }
  | _ { raise (Invalid_char (Lexing.lexeme lexbuf)) }
  | eof { EOF }

and comment = parse
  | "*)" { () }
  | "(*" { comment lexbuf; comment lexbuf }
  | _ { comment lexbuf }
```

10.9 Module *Term_syntax*

```
exception Syntax_error of string
```

parse_id is the function to use to parse identifier, returning their fix type.
current_variables is the current set of variables

```
val parse_id : (string -> Signatures.symbol_fix) ref

val check_make_term : (string -> string Gen_terms.term list -> string Gen_terms.term) ref

val current_variables : Variables.user_variables ref
```

10.10 Module *Hierarchical_signatures*

type *hsymbol*

class type *hierarchical_signature* =
    object
      inherit [*hsymbol*] *Signatures.parseable_signature*
      method *is_defined_here* : *hsymbol* → *bool*
    end

val *extend_signature* :
  hierarchical_signature list →
    #User_signatures.user_signature →
    hierarchical_signature

10.11 Module *User_terms*

(*term_of_string sigma t*) returns the term over the signature *sigma*, read in the string *s*

val *term_of_string* :
  'a #Signatures.parseable_signature → *Variables.user_variables* →
    string → *'a Gen_terms.term*

val *equation_of_string* :
  'a #Signatures.parseable_signature → *Variables.user_variables* →
      string → (*'a Gen_terms.term* × *'a Gen_terms.term*)

(*equation_set_of_string sigma t*) returns the set of equations over the signature *sigma*, read in the string *s*

val *equation_set_of_string* :
  'a #Signatures.parseable_signature → *Variables.user_variables* →
      string → (*'a Gen_terms.term* × *'a Gen_terms.term*) *list*

(*rule_set_of_string sigma t*) returns the set of rewrite rules over the signature *sigma*, read in the string *s*

val *rule_set_of_string* :
  'a #Signatures.parseable_signature → *Variables.user_variables* →
      string → (*'a Gen_terms.term* × *'a Gen_terms.term*) *list*

(*prec_of_string sigma s*) returns the precedence over the signature *sigma*, read in the string *s*

val *prec_of_string* :
  'a #Signatures.parseable_signature → *string*
    → *'a Orderings_generalities.ordering*

val *order_constraint_of_string* :
  'a #Signatures.parseable_signature → *Variables.user_variables* →
    string → *'a Order_constraints.formula*

(*status_of_string sigma s*) returns the function status (for RPO-like term orderings) over the signature *sigma*, read in the string *s* (raises the exception *Invalid_status* whenever an AC function symbol is not given a multiset status).

exception *Invalid_status* of *string*

val *status_of_string* :
  'a #Signatures.parseable_signature → *string*
    → *'a Rpo.status_function*

10.12 Module *Hierarchical_trs*

```
type htrs =
  {
    htrs_sig : Hierarchical_signatures.hierarchical_signature;
    htrs_new_rules :
      Hierarchical_signatures.hsymbol Rewrite_rules.t list;
    htrs_imported : htrs list;
    htrs_all_rules :
    Hierarchical_signatures.hsymbol Rewrite_rules.t list;
    proved_c_e_terminating : bool ref;
  }
```

extend imported sigma vars rules makes a new HTRS from the list *imported* of known HTRS, by adding the new signature *sigma* and the new rules syntactically given in the string *rules*, those being defined on *sigma* and the signatures of the *imported* htrs.

If a defined symbol in new rules *rules* is not defined in *sigma* then an exception *Overriden* is thrown.

```
val extend :
  htrs list → #User_signatures.user_signature →
    Variables.user_variables → string → htrs
```

```
val assume_ce :
  htrs → unit
```

```
val clear_ce :
  htrs → unit
```

```
exception Overriden of string
```

10.13 Module *Sorted_signatures*

```
open Sorts
open User_signatures
open Signatures
```

profile_of_symbol f returns (l,r) where l is the list of sorts of arguments of f and r is its return sort.

```
class type ['a] sorted_signature =
  object
    inherit ['a] signature
    method profile_of_symbol : 'a → sort_id list × sort_id
  end
```

```
class type user_sorted_signature =
  object
    inherit [symbol_id] sorted_signature
    inherit user_signature
  end
```

```
class virtual from_string : #sort → string → user_sorted_signature
```

10.14 Module *Variables*

This module provides an abstract data type for first-order variables. In an abstract point of view, a set of variables is any set equipped with total order (an equality is enough in theory, but we require a total order because we want to provide efficient implementation of finite sets of variables and finite maps indexed by variables. Finally a set of variables have to be infinite: it is always possible to find a variable that do not belong to a given finite set.

Variables have no readable representation by default. If you need a set of variables where some variables have a “name”, for user interaction purpose, you can use the class *user_variables*.
var_id is the abstract type for a variable.

string_of_var_id x displays a variable identifier in a raw way. The result is not supposed to be parse again. Mainly for debugging purpose. See *user_variables* class below for a better way of printing variable ids.

type *var_id*

val *string_of_var_id* : *var_id* → *string*

val *compare_var* : *var_id* → *var_id* → *int*

val *leftify_var* : *var_id* → *var_id*

val *rightify_var* : *var_id* → *var_id*

VarOrd is a module that provides a total order on variables.

module *VarOrd* : *Ordered_types.OrderedType* with type t = *var_id*

VarSet is a module that provides finite sets of variables.

module *VarSet* : *Ordered_sets.OrderedSet* with type *'a elt* = *var_id*

VarMap is a module that provides finite maps indexed by variables.

module *VarMap* : *Inttagmap.IntTagMapModule* with type *'a key* = *var_id*

(*fresh_variables* n) returns a list of n variables. Here, the word “fresh” does not means anything else that these are n distinct variables. Use the next function if you need to obtain a variable distinct from some others.

val *fresh_variables* : *int* → *var_id list*

(*var_outside_set* s) returns a variable that do not belong to the set s.

val *var_outside_set* : *unit VarSet.t* → *var_id*

val *shift_variable* : *var_id* → *var_id* → *var_id*

val *max_variable* : *var_id* → *var_id* → *var_id*

val *max_var_of_set* : *unit VarSet.t* → *var_id*

val *min_var* : *var_id*

(*init_for_unif* s) initializes the set of variables for unification. The intended meaning of s is the set of variables occurring in the initial unification problem.

val *init_for_unif* : *unit VarSet.t* → *unit*

fresh_var_for_unif () generates a variable at each call. In a session starting by a call to (*init_for_unif* s), these variables are pairwise distinct, and do not occur in s.

val *fresh_var_for_unif* : *unit* → *var_id*

val *print_unif_var* : *var_id* → *unit*

(*user_variables l*) returns an object that provides two functions for converting a variable into a readable name and vice-versa. *l* is a list of strings that give a list of name that you would like to use. Example :

```
let my_vars = new Variables.from_string "x y z"
```

exception *Syntax_error* of *string*

(*var_of_string* s) returns the variable whose name is s. Raises exception *Not_found* is no variable corresponds.

class type *user_variables* =
object
  method *string_of_var* : *var_id* → *string*
  method *var_of_string* : *string* → *var_id*
    end

val *split* : *string* → *string list*

val *from_list* : *string list* → *user_variables*

val *from_string* : *string* → *user_variables*

val *default* : *user_variables*

val *hash* : *var_id* → *int*

module *Default* :
  sig
    val *string_of_var* : *var_id* → *string*
    val *var_of_string* : *string* → *var_id*
  end

10.15 Module *Sorts*

type *sort_id*
type *sort_table*

class type *sort* =
    object
      val *sort_table* : *sort_table*
      method *string_of_sort* : *sort_id* → *string*
    end

new *from_list l* returns a class sort whose sort names are the strings in the list *l*. Example : new *from_list* ["nat";"int"]

class *from_list* : *string list* → *sort*

new *from_string s* returns a class sort obtained by the syntactical analysis of string s. Example : new *from_string* "nat␣int"

class *from_string* : *string* → *sort*

Chapter 11

Library `Term Orderings`

This library is for all various definitions of term orderings: polynomial interpretations, path orderings ; and combination of orderings to build orderings from others : lexicographic combinations, recursive program schemes.

11.1 Module *Poly_interp*

open *Orderings_generalities*
open *Signatures*
open *Variables*
open *Gen_terms*

Polynomials used in polynomial orderings are multivariate polynomials with arbitrary size integers as coefficients

module *NumRing* : *Polynomials.RingType* with type *coef* = *Numbers.t*

module *IntOrd* :
  Ordered_types.OrderedType
  with
    type *t* = *int*

module *IntSet* :
  Ordered_sets.OrderedSet with type *'a elt* = *int*

module *IntMap* : *Ordered_maps.OrderedMap* with type *'a key* = *int*

module *IntPolynomials* :
  Polynomials.PolynomialType
  with
    module *Base_ring* = *NumRing*
  and
    module *Var_set* = *IntSet*
  and
    module *Var_map* = *IntMap*
  and
    type *variable* = *int*

val *current_signature* : *User_signatures.user_signature ref*

current_signature is a reference onto the signature used for parsing the polynomial interpretations.

val *current_poly_vars* : *string list ref*

current_poly_vars is a reference onto the list of variables occurring in a polynomial interpretation (used in parsing).

type *'a t* = (*'a*, *IntPolynomials.poly*) *SymbolMap.t*

val *print_int_polynomial* : *IntPolynomials.poly* $\rightarrow$ *unit*
val *print* : *'a #signature* $\rightarrow$ *'a t* $\rightarrow$ *unit*

val *latex_print* :
  out_channel $\rightarrow$ *'a #Signatures.signature* $\rightarrow$ *'a t* $\rightarrow$ *unit*

(*build_var_interp_of_term t*) builds a one-to-one map from the (term) variables of t to integers

(*complete_var_interp_of_term old_interp t*) builds a one-to-one map from the (term) variables of t to integers variables by completing the old interpretations *old_interp*

val *complete_var_interp_of_term* :
  (*var_id*,*int*) *VarMap.t* $\rightarrow$ *int* $\rightarrow$ *'symbol term* $\rightarrow$ (*var_id*,*int*) *VarMap.t* $\times$ *int*

val *build_var_interp_of_term* : *'symbol term* $\rightarrow$ (*var_id*,*int*) *VarMap.t* $\times$ *int*

(*check_poly_interp sigma f p*) checks whether the polynomial p is a valid interpretation for the symbol f. Raise a *Failure* exception if not.

Verifications are :

- the dimension of p must be the arity of f ;
- p must be positive or null ;
- p must be increasing in each variable, that is each derivatives must be positive ;
- if f is commutative, p has to be symmetric ;
- if f is AC, f has to be of the form $axy+b(x+y)+c$ where $b^2 = b+ac$.

val *check_poly_interp* :
  'symbol #signature $\rightarrow$ *'symbol* $\rightarrow$ *IntPolynomials.poly* $\rightarrow$ *unit*

(*mu_translate mu P*) returns the translation of $P(X_1,...,X_n)$ by μ that is $P(X_1+\mu,..,X_n+\mu)-\mu$.

val *mu_translate* : *int* $\rightarrow$ *IntPolynomials.poly* $\rightarrow$ *IntPolynomials.poly*

(*poly_ord sigma P*) is the ordering on *sigma*-term generated by P, that is:
ocwbegindcode $s > t$ \mbox{ *iff* } $I(s) > I(t)$ \ where
ocwbegindcode $I(f(t\_1,...,t\_n)) = P(f)(I(t\_1),...,I(t\_n))$ \

val *poly_ord* :
  'symbol #signature
  $\rightarrow$ (*'symbol* $\rightarrow$ *IntPolynomials.poly*) $\rightarrow$ *'symbol term ordering*

(*poly_times* ()) returns the pair (user time, system time) spent in poly, in seconds

val *poly_times* : *unit* $\rightarrow$ *float* $\times$ *float*

module type S =
sig
  type *symbol*
  type *term*
  val *o* : *term ordering*
end

module *Make* (T : *Term_algebra.TermAlgebra*)
  (P : sig val *symb_interp* : *T.symbol* $\rightarrow$ *IntPolynomials.poly* end) :
  (S with type *symbol* = *T.symbol*
    and type *term* = *T.term*)

11.2 Module *Term_orderings*

```
open Orderings_generalities
open User_signatures
open Gen_terms
open Poly_interp

type 'symbol t =
  | KBO of ('symbol → int) × 'symbol ordering
  | RPO of ('symbol ordering × 'symbol Rpo.status_function)
  | POLY of ('symbol → IntPolynomials.poly)
  | LEX of 'symbol t × 'symbol t
  | Equivalent

module Make (T : Term_algebra.TermAlgebra) :
  sig
    val build : T.symbol t → T.term ordering
  end

val term_compare :
  symbol_entry array → symbol_id t → symbol_id term ordering
```

11.3 Module *Poly_ordering*

```
open Orderings
open Poly
```

ordre d'evaluation sur les polynomes

(*compare_poly P Q*) retourne

Equivalent si pour tt x1.. xn P(x1..xn)=Q(x1..xn) *Greater* si pour tt x1.. xn P(x1..xn)>Q(x1..xn) *Less_than* si pour tt x1.. xn P(x1..xn)<Q(x1..xn) *Greater_or_equivalent* si pour tt x1.. xn P(x1..xn)>=Q(x1..xn) *Less_or_equivalent* si pour tt x1.. xn P(x1..xn)>=Q(x1..xn) *Uncomparable* sinon

bien sur c'est imcomplet : en fait la specification est :

si (*compare_poly P Q*) retourne

Equivalent alors pour tt x1.. xn P(x1..xn)=Q(x1..xn) *Greater* alors pour tt x1.. xn P(x1..xn)>Q(x1..xn) *Less_than* alors pour tt x1.. xn P(x1..xn)<Q(x1..xn) *Greater_or_equivalent* alors pour tt x1.. xn P(x1..xn)>=Q(x1..xn) *Less_or_equivalent* alors pour tt x1.. xn P(x1..xn)>=Q(x1..xn) *Uncomparable* alors on ne sait rien

```
val compare_poly : polynome → polynome → comparison_result
```

11.4 Module *Rpo*

```
open Orderings_generalities
open Gen_terms

module type S =
sig
  type term
  val o : term ordering
end

type rpo_status =
  | Multiset
  | Lr_lexico
  | Rl_lexico
```

```
type 'symbol status_function = 'symbol → rpo_status
```

The functor *Make* (T) (PS) builds a module S which contains the ordering o over the terms of the term algebra module T as the RPO built with the precedence *prec* and status *stat* provided by the module PS.

It is assumed that all the symbols of the module T.*Signature* are free. Otherwise, use *Acrpo*.*Make* instead.

Warning! It is assumed that the status are valid with respect to precedence, that is whenever f and g are equivalent w.r.t. *prec* then they have the same status, and if it is a lexicographic status, then f and g have the same arity. Generally speaking, instead of using RPO with equivalent symbols in the precedence, it is better to apply first a RPS to replace symbols that are supposed to be equivalent by a representant of them. Moreover this allows to replace a symbol by a symbol with smaller arity.

However, a partial check is made when this function is called by the toplevel.

```
module Make
  (T : Term_algebra.TermAlgebra)
  (PS :
    sig
      type symbol = T.symbol
      val prec : symbol ordering
      val status : symbol status_function
    end) :
  (S with type term = T.term)
```

11.5 Module *Acrpo*

```
open Orderings_generalities
open Rpo
```

```
val rpo_time : float ref
```

The functor *Make* (T) (PS) builds a module S which contains the ordering o over the terms of the module $T\}$ as *the ACRPO built* with *the precedence* [*prec*] and *status* [*stat*] *provided by the* module [*PS*].

```
module Make
  (T : Term_algebra.TermAlgebra)
  (PS :
    sig
      type symbol = T.symbol
      val prec : symbol ordering
      val status : symbol status_function
    end) :
  (Rpo.S with type term = T.term)
```

Chapter 12

Library theories

This library provides ...

12.1 Module *Axioms*

```
module type S =
sig

  type symbol
  type term
  type side = Left | Right | Middle

  type axiom =
      Associativity of symbol
    | Unit of side × symbol × symbol
    | Inverse of side × symbol × symbol × symbol
    | Absorb of side × symbol × symbol
    | Nilpotency of symbol × symbol × int × int
    | Idempotency of symbol × int × int
    | Distributivity of side × symbol × symbol
    | Add_multiply of symbol × symbol
    | Pseudo_associativity of side × symbol
    | Associator of symbol × symbol × symbol × symbol
    | Commutator4 of symbol × symbol × symbol × symbol
    | Commutator3 of symbol × symbol × symbol
    | Ternary_associativity of symbol
    | Ternary_projection of int × int × int × symbol
    | Ternary_inverse of side × symbol × symbol
    | Ternary_single_axiom of symbol × symbol
    | Pixley_def of symbol × symbol × symbol × symbol
    | Pixley_1 of symbol × symbol × symbol
    | Pixley_2 of symbol × symbol × symbol
    | Pixley_3 of symbol × symbol × symbol
    | BOO_l1 of symbol × symbol
    | BOO_l3 of symbol × symbol
    | BOO_inverse of symbol × symbol × symbol
    | Majority_1 of symbol × symbol
    | Majority_2 of symbol × symbol
    | Majority_3 of symbol × symbol
    | Self_dual_distributivity of symbol × symbol
    | CL_def_axiom of symbol × symbol
```

```
    | B_axiom of symbol × symbol
    | C_axiom of symbol × symbol
    | K_axiom of symbol × symbol
    | H_axiom of symbol × symbol
    | O_axiom of symbol × symbol
    | N_axiom of symbol × symbol
    | N1_axiom of symbol × symbol
    | S_axiom of symbol × symbol
    | Q_axiom of symbol × symbol
    | Q1_axiom of symbol × symbol
    | T_axiom of symbol × symbol
    | W_axiom of symbol × symbol
    | W1_axiom of symbol × symbol
    | W2_axiom of symbol × symbol
    | SB_property of symbol × symbol × symbol × symbol
    | Abstraction of symbol × symbol × symbol
    | Waj1 of symbol × symbol
    | Waj2 of symbol × symbol
    | Waj3 of symbol
    | Waj4 of symbol × symbol × symbol
    | Robbins of symbol × symbol
    | Join_compl of symbol × symbol
    | Meet_compl of symbol × symbol × symbol
    | Cn_axiom of int × symbol

  val print_two_symbols : symbol → symbol → unit
  val print_three_symbols : symbol → symbol → symbol → unit
  val print_four_symbols : symbol → symbol → symbol → symbol → unit
  val print_five_symbols : symbol → symbol → symbol → symbol → symbol → unit
  val print_list_of_symbols : symbol list → unit

  val print_axiom : axiom → unit

  module AxiomSet : (Ordered_sets.OrderedSet with type 'a elt = axiom)

  val recognize_axiom : term → term → axiom
end

module Make :
  functor (T : Term_algebra.TermAlgebra) →
    (S with type symbol = T.symbol
       and type term = T.term)
```

12.2 Module *Standard_matching*

open *Gen_terms*

(*matching pattern subject*) returns the most general filter of *subject* over *pattern*. Raises *No_match* if no match is found.

This is standard matching : all symbols are assumed to be free.

exception *No_match*

val *matching* : *'symbol* term → *'symbol* term → *'symbol* Substitution.t

12.3 Module *Unif_index*

open *Term_algebra*

```
module type S =
sig
  type term
  type rule
  type t
  val compile_for_unification : rule list → t
  val retrieve : t → term → rule list
end

module Make (T : TermAlgebra) :
  (S with type term = T.term
     and type rule = T.rule)
```

12.4 Module *Controle*

```
exception No_solution
exception Not_appliable

type 'problem disjunction = 'problem list

val orelse :
  ('problem → 'problem_disjunction) → ('problem → 'problem_disjunction)
    → 'problem → 'problem_disjunction

val repeat :
  ('problem → 'problem disjunction) → 'problem → 'problem disjunction
```

12.5 Module *Theory*

There 3 kinds of unifications in C*i*ME:

- the *PLAIN* unification is the usual unification modulo;
- the *AC_COMPLETE* unification provides a representation of the unifiers modulo C and AC by a set of unifiers modulo all the current theories (see [?] for more details);
- the *AC* unification provides a complete set of unifiers modulo C and AC, the others axioms of the theory being ignored.

```
type unif_kind = PLAIN | AC_COMPLETE | AC_ONLY

type 'symbol elem_theory =
  | Empty of 'symbol option
  | C of 'symbol
  | AC of 'symbol
  | ACU of 'symbol × 'symbol
  | ACI of 'symbol
  | AG of 'symbol × 'symbol × 'symbol
  | ACUN of 'symbol × 'symbol × int
  | BR of 'symbol × 'symbol × 'symbol × 'symbol

  exception Syntax_error of string

module type S =
sig
  type symbol

  val unif_type : unif_kind ref
```

```
  type elem_theory =
    | Empty of symbol option
    | C of symbol
    | AC of symbol
    | ACU of symbol × symbol
    | ACI of symbol
    | AG of symbol × symbol × symbol
    | ACUN of symbol × symbol × int
    | BR of symbol × symbol × symbol × symbol

  exception No_theory

  type unif_elem_theory = symbol option × elem_theory

  module TheorySet :
    Ordered_sets.OrderedSet with type 'a elt = elem_theory

  module UnifTheorySet :
    Ordered_sets.OrderedSet with type 'a elt = unif_elem_theory

  val additive_symbol_of_theory : elem_theory → symbol
  val unit_symbol_of_theory : elem_theory → symbol
  val minus_symbol_of_theory : elem_theory → symbol

  val elem_theory_from_unif_elem_theory : unif_elem_theory → elem_theory
  val string_of_elem_theory : elem_theory → string
  val string_of_unif_elem_theory : unif_elem_theory → string

  val print_theory : 'a TheorySet.t → unit

  val theory_check : elem_theory → elem_theory
end

module Make :
  functor (T : Term_algebra.TermAlgebra) →
    (S with type symbol = T.symbol)

module String_theory : (S with type symbol = string)
```

12.6 Module *Variable_abstraction*

```
open Theory

module type S =
sig
  type symbol
  type unif_elem_theory
  type term
  val purify_list_of_equations :
    unif_kind → (symbol → unif_elem_theory) → (term × term) list
      → (term × term) list
end

module Make
  (T : Term_algebra.TermAlgebra)
  (Th : Theory.S with type symbol = T.symbol) :
  (S with type symbol = T.symbol
     and type unif_elem_theory = Th.unif_elem_theory
     and type term = T.term)
```

12.7 Module *Problems*

```
open Variables
open Theory

module type S =
sig
  type symbol
  type elem_theory
  type unif_elem_theory
  type term
  module UnifElemThMap :
    (Ordered_maps.OrderedMap with type 'a key = unif_elem_theory)

  type status = Unsolved | Merged | Marked | Solved
  type mark = No_mark | Erasable | Permanent of elem_theory

  type elem_pb =
      {
        key : unif_elem_theory;
        status : status;
        size : int option;
        elem_th : elem_theory;
        inst_variables : (unit, term) VarMap.t;
        marked_variables : (unit, mark) VarMap.t;
        edges : (var_id × var_id) list;
        equations : (term × term) list
      }

  type problem = {
    unif_kind : unif_kind;
    global_status : status;
    find_th : symbol → unif_elem_theory;
    vars_for_eqe : unit VarSet.t;
    first_vars : unit VarSet.t;
    var_var : (unit, var_id) VarMap.t;
    elem_pbs : (unit, elem_pb) UnifElemThMap.t;
    solved_part : (term × term) list;
  }

  val print_elem_pb : elem_pb → unit

  val print_problem : problem → unit

  val replace_a_var_by_a_var_in_an_eq :
    var_id → term → term × term → term × term

  val add_an_equation_between_variables :
    problem → var_id → var_id → problem
```

Initialisation of the unification problem : (*init th l*) builds the data structucture for solving *l* modulo the equational theory *th*, where *l* is a list of equations between terms.

```
  val init :
    unif_kind → (symbol → unif_elem_theory) → (term × term) list → problem

  val insert_solved_elem_pbs :
    problem → unif_elem_theory → (term × term) list list → problem list

  val existential_quantifiers_elimination : problem → (term × term) list
end
```

```
module type Solve =
sig
  type term
  type elem_pb
```

solve elem_pb computes a unifier for *elem_pb* modulo an elementary unification theory, that is, solves the equations of the problem and checks that the constraints of marks are satisfied.

```
  val solve :
    unif_kind → unit VarSet.t → elem_pb → (term × term) list list
end

module Make
  (T : Term_algebra.TermAlgebra)
  (Th : Theory.S with type symbol = T.symbol)
  (V : Variable_abstraction.S with type symbol = T.symbol
                              and type unif_elem_theory = Th.unif_elem_theory
                              and type term = T.term) :
  (S with type symbol = T.symbol
     and type elem_theory = Th.elem_theory
     and type unif_elem_theory = Th.unif_elem_theory
     and type term = T.term)
```

12.8 Module *Theory_syntax*

```
val is_defined : (string → User_signatures.symbol_id) ref
val current_signature : User_signatures.user_signature ref
```

12.9 Parser *Theory_parser*

Header

```
  open User_signatures
  open Term_algebra
```

Token declarations

```
%token <User_signatures.symbol_id> IDENT
%token <int> INT
%token KW_ACU KW_ACI KW_AG KW_ACUN KW_BR
%token COMMA SEMICOLON OPENPAR CLOSEPAR
%token EOF

%start theory
%type <User_signatures.symbol_id Theory.elem_theory list> theory
```

Grammar rules

```
theory ::=
      EOF
          { [] }
    | decl
          {
             [$1]
          }
    | decl SEMICOLON theory
          {
             $1 :: $3
          }


decl ::=
    acu {  $1 }
  | aci {  $1 }
  | ag {  $1 }
  | acun {  $1 }
  | br {  $1 }


acu ::=
KW_ACU OPENPAR IDENT COMMA IDENT CLOSEPAR
    {
       Theory.ACU($3,$5)
    }


aci ::=
KW_ACI OPENPAR IDENT CLOSEPAR
    {
       Theory.ACI($3)
    }


ag ::=
KW_AG OPENPAR IDENT COMMA IDENT COMMA IDENT CLOSEPAR
    {
       Theory.AG($3,$5,$7)
    }


acun ::=
KW_ACUN OPENPAR IDENT COMMA IDENT COMMA INT CLOSEPAR
    {
       Theory.ACUN($3,$5,$7)
    }


br ::=
KW_BR OPENPAR IDENT COMMA IDENT COMMA IDENT COMMA IDENT CLOSEPAR
    {
       Theory.BR($3,$5,$7,$9)
    }
```

12.10 Lexer *Theory_lexer*

```
{
  open Theory_parser
  open Theory_syntax
  open Signatures

  exception Invalid_char of string

  let ident s =
    try
      let f = !is_defined s in IDENT f
    with
        Not_found →
          try
            INT(int_of_string s)
          with _ → raise (Theory.Syntax_error ("undefined identifier "^s))
}

rule token = parse
    [' ''\t''\n'] { token lexbuf }
  | ',' { COMMA }
  | ';' { SEMICOLON }
  | '(' { OPENPAR }
  | ')' { CLOSEPAR }
  | "ACU" { KW_ACU }
  | "ACI" { KW_ACI }
  | "AG" { KW_AG }
  | "ACUN" { KW_ACUN }
  | "BR" { KW_BR }
  | ['a'-'z''A'-'Z''0'-'9']['a'-'z''A'-'Z''0'-'9''_''\'']*
  | ['^''+''.''&''*''-''/''!''?''@''~''#'] ('_'(['a'-'z''A'-'Z''0'-'9']+))*
                                { ident(Lexing.lexeme lexbuf) }
  | _ { raise (Invalid_char (Lexing.lexeme lexbuf)) }
  | eof { EOF }
```

12.11 Module *User_theory*

```
val from_string :
  User_signatures.user_signature → string →
    User_signatures.symbol_id Theory.elem_theory list
```

12.12 Module *Oc*

```
open Variables

module type S =
sig
  type term

  type cycle =
      Cycle of var_id list
    | No_cycle of var_id list
```

(*occur_check_without_var_var list_of_equations*) check that there is no cycle in the occurrence graph generated by the *list_of_equations*, that is returns

- *No_cycle list_of_vars* when there is no cycle in the graph; in this case, the (total) ordering of the variables in the *list_of_vars* is compatible with the (partial) ordering induced by the graph,
- *Cycle list_of_vars* when there is a cycle in the graph going through the nodes *list_of_vars*.

It is assumed that *list_of_equations* does not contains any equation between variables.

val *occur_check_without_var_var* : $(term \times term)$ *list* $\rightarrow$ *cycle*

A call to (*instanciate_when_no_cycle list_of_vars list_of_equations*) assumes that

- *list_of_equations* is a list of pair of terms
- there is no cycle in the occur-check graph generated by the *list_of_equations*,
- and that *list_of_vars* provides a total ordering compatible with the occur-check graph.

This function takes *list_of_equations* as a DAG-solved form, in particular all the equations are of the form *variable* $=$ *term*, and it returns an equivalent solved form.

val *instanciate_when_no_cycle* :
  var_id list $\rightarrow$ $(term \times term)$ *list* $\rightarrow$ $(term \times term)$ *list*

A call to (*occur_check sign hct list_of_eqs_var_var list_of_equations*) assumes that

- *list_of_equations* is a list of pair of terms built over the signature *sign* thanks to the hashconsing table *hct*,
- *list_of_eqs_var_var* contains only equations between variables,
- *list_of_equations* contains only equations between a variable and a non-variable term,
- and the **Coalesce** rule does not apply on the union of these two sets of equations.

It returns either a failure when there is a cycle in the occur-check graph or a list of equations which is a solved form for the union of *list_of_eqs_var_var* and *list_of_equations*.

val *occur_check* :
  (*unit*, *var_id*) *VarMap.t* $\rightarrow$ $(term \times term)$ *list* $\rightarrow$
    $(term \times term)$ *list*
end

module *Make* :
  functor (*T* : *Term_algebra.TermAlgebra*) $\rightarrow$
    (*S* with type *term* = *T.term*)

12.13 Module *Unif_free*

open *Variables*

module type *S* =
sig

  type *term*
  type *elem_pb*
  type *pure_elem_pb* =
    {
      map_of_var_var : (*unit*, *var_id*) *VarMap.t*;
      scanned_equations : $(term \times term)$ *list*;
      other_equations : $(term \times term)$ *list*
    }

Deletion of trivial equations.

val *delete* : *pure_elem_pb* $\rightarrow$ *pure_elem_pb*

Coalesce (Replacement of a variable by a variable).

val *coalesce* : *pure_elem_pb* $\rightarrow$ *pure_elem_pb*

Merge for two equations with the same variable left-hand side.

val *merge* : *pure_elem_pb* $\rightarrow$ *pure_elem_pb*

solve_without_marks sign list_of_equations computes a unifier for *list_of_equations* modulo the empty equational theory.

val *solve_without_marks* : $($*term* $\times$ *term*$)$ *list* $\rightarrow$ $($*term* $\times$ *term*$)$ *list*

solve elem_pb computes a unifier for *elem_pb* modulo the empty equational theory, that is, solves the equations of the problem and checks that the constraints of marks are satisfied.

val *solve* : *elem_pb* $\rightarrow$ $($*term* $\times$ *term*$)$ *list list*

end

module *Make*
  $(T$: *Term_algebra.TermAlgebra*$)$
  $(P$: *Problems.S* with type *term* $=$ *T.term*$)$
  $(O$: *Oc.S* with type *term* $=$ *T.term*$)$:
  $(S$ with type *term* $=$ *T.term*
    and type *elem_pb* $=$ *P.elem_pb*$)$

12.14 Module *Unif_commutative*

module type S =
sig
  type *term*
  type *elem_pb*

solve elem_pb computes a unifier for *elem_pb* modulo the commutativity, that is, solves the equations of the problem and checks that the constraints of marks are satisfied.

val *solve* : *elem_pb* $\rightarrow$ $($*term* $\times$ *term*$)$ *list list*

end

module *Make*
  $(T$: *Term_algebra.TermAlgebra*$)$
  $(P$: *Problems.S* with type *term* $=$ *T.term*$)$
  $(O$: *Oc.S* with type *term* $=$ *T.term*$)$
  $(F$: *Unif_free.S* with type *term* $=$ *T.term*
      and type *elem_pb* $=$ *P.elem_pb*$)$:
  $(S$ with type *term* $=$ *T.term*
    and type *elem_pb* $=$ *P.elem_pb*$)$

12.15 Module *Unif_to_arith*

open *Variables*

This module provides some functions in order to translate a unification modulo AC (resp. ACU, resp. AG) into a system of linear equations over the non-negative integers (resp. non-negative integers, resp. integers). The variables of the problem are sorted according to the theory (other than the current one) where they are instanciated:

```
 x ....... x     x    .... x     x    ... x   .....
  1         k0    k0+1      k1    k1+1     k2
 \____  ____/   \_____  _____/   \____  ____/
      \/             \/              \/
non inst. vars    vars inst.       vars int.
                   in Th1           in Th2
```

The integer solver returns only solutions of the form

```
 n ... n     0 .... 0  ... 0 .... 0   (0 or 1) ... (0 or 1) 0 .... 0 ...
  1     k0
\___  ___/  \__  __/     \__  __/   \________  ________/
    \/          \/           \/              \/

 non inst.  vars inst.    vars int.        vars int.      vars int.
   vars       in Th1      in Th2           in Th(l-1)      in Thl
```

module type S =
sig
  type *term*
  type *elem_theory*
  type *elem_pb*

  val *add_term* :
    elem_theory $\rightarrow$ (*unit*, *int*) *VarMap.t* $\rightarrow$ *int array* $\rightarrow$ *term* $\rightarrow$ *unit*

  unif_to_arith_without_matrix elem_pb returns a pair made of

- a map giving the indices corresponding to the variables,
- an array of variables corresponding to the inverse map of the above map
- an array $[|k1; k2; ...|]$ of indices as described above.

Remark : this function should be called only on problems with AC-like theories, that is AC, ACU, ACI, ACUN, and AG.

  val *unif_to_arith_without_matrix* :
    elem_pb $\rightarrow$ (*unit*, *int*) *VarMap.t* $\times$ *var_id array* $\times$ *int array*

  unif_to_arith elem_pb returns a quadruple made of

- a map giving the indices corresponding to the variables,
- an array of variables corresponding to the inverse map of the above map
- an array $[|k1; k2; ...|]$ of indices as described above.
- a matrix of non-negative integers corresponding to the translation of the equations of the elementary problem.

Remark : this function should be called only on problems with AC-like theories, that is AC, ACU, ACI, ACUN, and AG.

  val *unif_to_arith* :
    elem_pb $\rightarrow$ (*unit*, *int*) *VarMap.t* $\times$ *var_id array* $\times$ *int array* $\times$ *int array array*
end

module *Make*
  (*T* : *Term_algebra.TermAlgebra*)
  (*Th* : *Theory.S* with type *symbol* = *T.symbol*)
  (*P* : *Problems.S* with type *term* = *T.term*
        and type *elem_theory* = *Th.elem_theory*) :
  (*S* with type *term* = *T.term*
    and type *elem_theory* = *Th.elem_theory*
    and type *elem_pb* = *P.elem_pb*)

12.16 Module *Hullot_bin_trees*

module type *BINARY_TREE* =
  sig
    type *t*
    val *arbre_binaire* : $int \rightarrow (t \rightarrow bool) \rightarrow (t \rightarrow bool) \rightarrow t\ list$
  end

module *Small_binary_tree* :
  BINARY_TREE with type $t = int$

module *Large_binary_tree* :
  BINARY_TREE with type $t = int\ array$

12.17 Module *Arith_to_unif*

open *Bit_field*
open *Variables*
open *Theory*

module type *S* =
sig

  type *elem_theory*
  type *term*
  type *elem_pb*

cache est une fonction qui prend un vecteur de vecteurs d'entiers positifs (solutions diophantiennes d'un systeme), et qui retourne un vecteur de naturels codant des vecteurs de bits, tel que chaque 1 correspond a un entier non-nul de l'entree, et chaque 0 correspond a un 0, le tout transpose, pour avoir directement acces aux colonnes associees a chaque variable.

  val *pcache* : $int\ array\ array \rightarrow int\ array$
  val *psmall_enough* : $int \rightarrow int \rightarrow int\ array \rightarrow int \rightarrow bool$
  val *pgreat_enough* : $int \rightarrow int \rightarrow int\ array \rightarrow int \rightarrow bool$

  val *gcache* : $int\ array\ array \rightarrow Large\_bit\_field.t\ array$
  val *gsmall_enough* :
    $int \rightarrow int \rightarrow Large\_bit\_field.t\ array \rightarrow Large\_bit\_field.t \rightarrow bool$
  val *ggreat_enough* :
    $int \rightarrow int \rightarrow Large\_bit\_field.t\ array \rightarrow Large\_bit\_field.t \rightarrow bool$

combinaison_lineaire + *vect_var vect_sols vect_nouv_var vect_car* retourne une liste d'equations de la forme var (de *vect_var*) = un terme de symbole de tete +, dont les variables sont celles de *vect_nouv_var* avec les coefficients apparaissant dans *vect_sols*.

  val *linear_combination* :
    $unif\_kind \rightarrow elem\_theory \rightarrow unit\ VarSet.t \rightarrow$
      $(unit, int)\ VarMap.t \rightarrow int\ array\ array \rightarrow term\ array \rightarrow int\ array \rightarrow$
        $(term \times term)\ list$

nettoyer pe_edge vect_var vect_sols enleve de *vect_sols* les solutions qui vont provoquer un echec pour OC, grace a *pe_edge*.

  val *clean_solutions* :
    $elem\_pb \rightarrow (unit, int)\ VarMap.t \rightarrow int\ array\ array \rightarrow$
      $int\ array\ array$

(*classify elem_pb array_of_vars map_var_int v_type vect_sols*) returns a pair (*vect_homogeneous_sols*, *vect_heterogeneous_sols*) where

- *vect_homogeneous_sols* contains the solutions from the array *vect_sols* which are equal to 0 over the marked variables,

- *vect_heterogeneous_sols* contains the other solutions of *vect_sols*

Remark : The solutions of *vect_sols* creating a cycle of size 2 are removed in *sorted_vect_sols*.

val *classify* :
  elem_pb → *var_id array* → (*unit*, *int*) *VarMap.t* → *int array* →
    int array array → (*int array array*) × (*int array array*)

(*sum_of_columns matrix*) returns an array containing the sum of the columns of the argument *matrix*.

val *sum_of_columns* : *int array array* → *int array*

(*associated_var_with_sol sum_of_sols array_of_vars v_type sol*) returns

- *Some c* when the value of the solution *sol* is equal to 1 for the component corresponding to the marked variable *c* (when there are several such marked variables, the function returns the first one encountered).

- *Some x* when the value of the solution *sol* is equal to 1 for the component corresponding to the variable *x*, and there is no other such variable.

- *None* otherwise.

val *associated_var_with_sol* :
  int array → *var_id array* → *int array* → *int array* → *var_id option*

(*associated_marked_var_with_sol array_of_vars v_type sol*) returns

- *Some c* when the value of the solution *sol* is equal to 1 for the component corresponding to the marked variable *c* (when there are several such marked variables, the function returns the first one encountered).

- *None* otherwise.

val *associated_marked_var_with_sol* :
  var_id array → *int array* → *int array* → *var_id option*

(*generate_vect_char_cst nb_var nb_true_var vect_sols_cst*) generates the list of subsets of non-homogeneous Diophantine solutions (encoded as 0-1 vectors) which are usefull in order to build unifiers for "arithmetic" theories which possess a unit, that is ACU, ACUN and AG.

val *generate_vect_char_cst* : *int* → *int* → *int array array* → *int array list*
end

module *Make* :
  functor (*T* : *Term_algebra.TermAlgebra*) →
    functor (*Th* : *Theory.S* with type *symbol* = *T.symbol*) →
      functor (*P* : *Problems.S* with type *term* = *T.term*
                and type *elem_theory* = *Th.elem_theory*) →
        (*S* with type *elem_theory* = *Th.elem_theory*
          and type *term* = *T.term*
          and type *elem_pb* = *P.elem_pb*)

12.18 Module *Unif_ac*

```
module Make
  (T : Term_algebra.TermAlgebra)
  (Th : Theory.S with type symbol = T.symbol)
  (P : Problems.S with type term = T.term
                  and type elem_theory = Th.elem_theory)
  (U : Unif_to_arith.S with type term = T.term
                       and type elem_theory = Th.elem_theory
                       and type elem_pb = P.elem_pb)
  (A : Arith_to_unif.S with type elem_theory = Th.elem_theory
                       and type term = T.term
                       and type elem_pb = P.elem_pb) :
  (Problems.Solve with type term = T.term
                  and type elem_pb = P.elem_pb)
```

12.19 Module *Unif_acu*

```
open Variables
open Theory

module type S =
sig
  type term
  type elem_pb

  val generate_vect_char :
    unif_kind → int → int → (unit, int) VarMap.t → int array array →
      int array array → int array list

  val filter_non_unit_vars :
    unit VarSet.t → int → (unit, int) VarMap.t → (unit, int) VarMap.t

  val solve :
    unif_kind → unit VarSet.t → elem_pb → (term × term) list list

  val is_ac_unifiable : unit VarSet.t → elem_pb → bool
end

module Make :
  functor (T : Term_algebra.TermAlgebra) →
    functor (Th : Theory.S with type symbol = T.symbol) →
      functor (P : Problems.S with type term = T.term
                              and type elem_theory = Th.elem_theory) →
        functor (U : Unif_to_arith.S with type term = T.term
                                     and type elem_theory = Th.elem_theory
                                     and type elem_pb = P.elem_pb) →
          functor (A : Arith_to_unif.S with type elem_theory = Th.elem_theory
                                       and type term = T.term
                                       and type elem_pb = P.elem_pb) →
            (S with type term = T.term
              and type elem_pb = P.elem_pb)
```

12.20 Module *Mark_acu*

module *Make* :
  functor (T : *Term_algebra.TermAlgebra*) $\rightarrow$
    functor (Th : *Theory.S* with type *symbol* = *T.symbol*) $\rightarrow$
      functor (P : *Problems.S* with type *term* = *T.term*
                and type *elem_theory* = *Th.elem_theory*) $\rightarrow$
        (*Problems.Solve* with type *term* = *T.term*
              and type *elem_pb* = *P.elem_pb*)

12.21 Module *Merge_acu*

module *Make* :
  functor (T : *Term_algebra.TermAlgebra*) $\rightarrow$
    functor (Th : *Theory.S* with type *symbol* = *T.symbol*) $\rightarrow$
      functor (P : *Problems.S* with type *term* = *T.term*
                and type *elem_theory* = *Th.elem_theory*) $\rightarrow$
        functor (U : *Unif_to_arith.S* with type *term* = *T.term*
                  and type *elem_theory* = *Th.elem_theory*
                  and type *elem_pb* = *P.elem_pb*) $\rightarrow$
          functor (A : *Arith_to_unif.S* with type *elem_theory* = *Th.elem_theory*
                    and type *term* = *T.term*
                    and type *elem_pb* = *P.elem_pb*) $\rightarrow$
            functor (Acu : *Unif_acu.S* with type *term* = *T.term*
                      and type *elem_pb* = *P.elem_pb*) $\rightarrow$
              (*Problems.Solve* with type *term* = *T.term*
                    and type *elem_pb* = *P.elem_pb*)

12.22 Module *Unif_aci*

module *Make* :
  functor (T : *Term_algebra.TermAlgebra*) $\rightarrow$
    functor (Th : *Theory.S* with type *symbol* = *T.symbol*) $\rightarrow$
      functor (P : *Problems.S* with type *term* = *T.term*
                and type *elem_theory* = *Th.elem_theory*) $\rightarrow$
        functor (U : *Unif_to_arith.S* with type *term* = *T.term*
                  and type *elem_theory* = *Th.elem_theory*
                  and type *elem_pb* = *P.elem_pb*) $\rightarrow$
          functor (A : *Arith_to_unif.S* with type *elem_theory* = *Th.elem_theory*
                    and type *term* = *T.term*
                    and type *elem_pb* = *P.elem_pb*) $\rightarrow$
            (*Problems.Solve* with type *term* = *T.term*
                  and type *elem_pb* = *P.elem_pb*)

12.23 Module *Unif_ag_acun*

module *Make*
  (T : *Term_algebra.TermAlgebra*)
  (Th : *Theory.S* with type *symbol* = *T.symbol*)
  (P : *Problems.S* with type *term* = *T.term*
      and type *elem_theory* = *Th.elem_theory*)
  (U : *Unif_to_arith.S* with type *term* = *T.term*

```
                              and type elem_theory = Th.elem_theory
                              and type elem_pb = P.elem_pb)
  (A : Arith_to_unif.S with type elem_theory = Th.elem_theory
                             and type term = T.term
                             and type elem_pb = P.elem_pb)
  (Acu : Unif_acu.S with type term = T.term
                         and type elem_pb = P.elem_pb) :
  (Problems.Solve with type term = T.term
                       and type elem_pb = P.elem_pb)
```

12.24 Module *Unif_bool*

```
open Variables

module type S =
sig
  type term
  type elem_pb

  val solve : elem_pb → (term × term) list list
end

module Make
  (T : Term_algebra.TermAlgebra)
  (Th : Theory.S with type symbol = T.symbol)
  (P : Problems.S with type term = T.term
                       and type elem_theory = Th.elem_theory) :
  (S with type term = T.term
      and type elem_pb = P.elem_pb)
```

12.25 Module *Mark*

```
open Variables

module type S =
sig
  type unif_elem_theory
  type mark
  type problem

  val add_a_mark :
    var_id → mark → unif_elem_theory → problem → problem
```

(*mark pb*) applies the **Mark** rule on the unification problem *pb* and returns

- either the exception *Not_appliable*
- either the exception *No_solution*
- or a list of marked unification problems.

```
  val mark : problem → problem list

  val lazy_mark : problem → problem Lazy_list.llist

end
```

```
module Make :
  functor (T : Term_algebra.TermAlgebra) →
    functor (Th : Theory.S with type symbol = T.symbol) →
      functor (P : Problems.S with type term = T.term
                                  and type elem_theory = Th.elem_theory
                                  and type unif_elem_theory = Th.unif_elem_theory) →
        (S with type unif_elem_theory = Th.unif_elem_theory
           and type mark = P.mark
           and type problem = P.problem)
```

12.26 Module *Cycle*

```
module type S =
sig
  type problem

  val cycle : problem → problem list
  val lazy_cycle : problem → problem Lazy_list.llist
end

module Make :
  functor (T : Term_algebra.TermAlgebra) →
    functor (Th : Theory.S with type symbol = T.symbol) →
      functor (P : Problems.S with type term = T.term
                                  and type elem_theory = Th.elem_theory
                                  and type unif_elem_theory = Th.unif_elem_theory) →
        functor (M : Mark.S with type unif_elem_theory = Th.unif_elem_theory
                                and type mark = P.mark
                                and type problem = P.problem) →
          functor (O : Oc.S with type term = T.term) →
            (S with type problem = P.problem)
```

12.27 Module *E_res*

```
module type S =
sig
  type problem
  val general_E_resolution : problem → problem list
  val lazy_general_E_resolution : problem → problem Lazy_list.llist
end

module Make :
  functor (T : Term_algebra.TermAlgebra) →
    functor (Th : Theory.S with type symbol = T.symbol) →
      functor (P : Problems.S with type term = T.term
                                  and type elem_theory = Th.elem_theory
                                  and type unif_elem_theory = Th.unif_elem_theory) →
        functor (M : Mark.S with type unif_elem_theory = Th.unif_elem_theory
                                and type mark = P.mark
                                and type problem = P.problem) →
          functor (O : Oc.S with type term = T.term) →
            (S with type problem = P.problem)
```

12.28 Module *Eqe*

12.29 Module *Unification*

```
open Variables
open Theory

val verbose : int ref
val unification_time : float ref
```

set_of_unifiers unif_k E term1 term2 returns a complete set of unifiers of *term1* and *term2* modulo the equational theory E. E may be any combination of some elementary theories among

- the free theory
- C
- AC
- ACU
- ACI
- AG
- ACUN
- BR

provided that they are pairwise signature-disjoint.

```
module type S =
sig
  type symbol
  type term
  type substitution
  type elem_theory
  type unif_elem_theory

  module TheorySet :
    Ordered_sets.OrderedSet with type 'a elt = elem_theory

  val verbose : int ref
  val th_from_user_th :
    User_signatures.user_signature → User_signatures.symbol_id Theory.elem_theory →
      elem_theory
  val set_of_solutions :
    unif_kind → (symbol → unif_elem_theory) → term → term → substitution list
  val plain_set_of_solutions : unit TheorySet.t → term → term → substitution list
  val display_plain_set_of_solutions : unit TheorySet.t → term → term → unit

  val free_unification : term → term → substitution
  val ac_unification : term → term → substitution list
  val first_ac_solution : (term × term) list → (term × term) list
  val has_an_ac_solution : (term × term) list → bool
  val free_solve_constraints : (term × term) list → (term × term) list
end

module Make
  (T : Term_algebra.TermAlgebra) :
  (S with type symbol = T.symbol
     and type term = T.term
     and type substitution = T.substitution)
```

12.30 Module *Ac_unification*

set_of_solutions ...

Chapter 13

Library `rewriting`

Functions to rewrite a term at top by a rule or a set of rules.

13.1 Module *Full_dnet*

module *Data* : (*Inttagset.IntTagSetModule* with type ('a,'b) *elt* = *int* × 'a)

module type *S* =
sig

type *term*
type *rule*
type *substitution*
type 'a *dnet*

val *all_data* : 'a *dnet* → (*int* × 'a) *list*
val *compile* : (*term* × 'a) *list* → 'a *dnet*
val *compile_rules* : *rule list* → *rule dnet*

add_a_data sign (*term*, *data*) *dnet* builds a new dnet corresponding to the old one *dnet* where one has added the data *data* corresponding with the term *term*.

val *add_a_data* : *term* × 'a → 'a *dnet* → 'a *dnet*

remove_a_data sign (*term*, *data*) *dnet* builds a new dnet corresponding to the old one *dnet* where one has removed the data *data* corresponding with the term *term*. When *data* is not indexed by *term* in *dnet*, *dnet* is returned unchanged.

val *remove_a_data* : *term* × 'a → 'a *dnet* → 'a *dnet*

replace_a_data sign old_data (*term*, *new_data*) *dnet* builds a new dnet corresponding to the old one *dnet* where one has replaced the data *old_data* corresponding with the term *term* by the data *new_data*. When *old_data* is not indexed by *term* in *dnet*, *dnet* is returned unchanged.

val *replace_a_data* : 'a → *term* × 'a → 'a *dnet* → 'a *dnet*

type *partial_match* =
| *Partial* of *substitution*
| *Total* of *substitution*

val *discriminate* :
'a *dnet* → *term* → (*partial_match* × 'a, *unit*) *Data.t*

val *is_encompassed_by* : 'a *dnet* → *term* → *bool*

type *position* = *int list*

module *PosData* : (*Ordered_maps.OrderedMap* with type *'a key* = *int* × *position*)

type *'a full_dnet*

val *full_compile* : (*term* × *'data*) *list* → *int* × *'data full_dnet*

val *full_compile_rules* : *rule list* → *int* × *rule full_dnet*

val *full_discriminate* :
  int → *'data full_dnet* → *term* →
    (*int* × *position*, (*partial_match* × *'data*, *unit*) *Data.t*) *PosData.t*

end

module *Make*
  (*T* : *Term_algebra.TermAlgebra*) :
  (*S* with type *term* = *T.term*
    and type *rule* = *T.rule*
   and type *substitution* = *T.substitution*)

13.2 Module *Standard_innermost*

open *Gen_terms*

(*innermost_normalize red t*) returns the innermost normal form of t w.r.t to the reduction relation *red*.

red is a function which, given a term t, returns a pair $(u, sigma)$ such that t rewrites to u $sigma$, or raises *Irreducible* if t is irreducible.

The substitution will be applied during the normalization process.

val *innermost_normalize* :
  'symbol #Signatures.signature →
    Variables.user_variables →
      (*'symbol term* → *'symbol term* × *'symbol Substitution.t*) →
        'symbol term → *'symbol term*

(*force_innermost_normalize red t*) does the same as (*innermost_normalize red t*), but raises *Irreducible* if no reduction at all is possible.

exception *Irreducible*

val *force_innermost_normalize* :
  'symbol #Signatures.signature → *Variables.user_variables* →
    (*'symbol term* → *'symbol term* × *'symbol Substitution.t*) →
      'symbol term → *'symbol term*

safe_innermost_normalize n red t does the same but where n is a bound for the number of rewrite steps to apply. If this bound is reached, then this function raises the exception *Unnormalized u* where u is the reduct of t obtained so far

exception *Unnormalized*

val *safe_innermost_normalize* :
  'symbol #Signatures.signature →
    Variables.user_variables → *int* →
      (*'symbol term* → *'symbol term* × *'symbol Substitution.t*) →
        'symbol term → *'symbol term*

val *safe_force_innermost_normalize* :
  'symbol #Signatures.signature →
    Variables.user_variables → *int* →
      (*'symbol term* → *'symbol term* × *'symbol Substitution.t*) →
        'symbol term → *'symbol term*

13.3 Module *Innermost*

module type S =
sig
  type *term*
  type *substitution*

(*innermost_normalize find memo red t*) returns the innermost normal form of t w.r.t to the reduction relation *red*.

red is a function which, given a term t, returns a pair $(u, sigma)$ such that t rewrites to u *sigma*, or raises *Irreducible* if t is irreducible.

The substitution will be applied during the normalization process. *find* and *memo* are memoization functions.

  val *innermost_normalize* :
    $(term \rightarrow term) \rightarrow (term \rightarrow term \rightarrow unit) \rightarrow$
      $(term \rightarrow term \times substitution) \rightarrow term \rightarrow term$

(*force_innermost_normalize find memo red t*) does the same as (*innermost_normalize find memo red t*), but raises *Irreducible* if no reduction at all is possible. *find* and *memo* are memoization functions.

  exception *Irreducible*

  val *force_innermost_normalize* :
    $(term \rightarrow term) \rightarrow (term \rightarrow term \rightarrow unit) \rightarrow$
      $(term \rightarrow term \times substitution) \rightarrow term \rightarrow term$

safe_innermost_normalize find memo n red t does the same but where n is a bound for the number of rewrite steps to apply. If this bound is reached, then this function raises the exception *Unnormalized* u where u is the reduct of t obtained so far

  exception *Unnormalized*

  val *safe_innermost_normalize* :
    $(term \rightarrow term) \rightarrow (term \rightarrow term \rightarrow unit) \rightarrow int \rightarrow$
      $(term \rightarrow term \times substitution) \rightarrow term \rightarrow term$

  val *safe_force_innermost_normalize* :
    $(term \rightarrow term) \rightarrow (term \rightarrow term \rightarrow unit) \rightarrow int \rightarrow$
      $(term \rightarrow term \times substitution) \rightarrow term \rightarrow term$

end

module *Make* (T : *Term_algebra.TermAlgebra*) :
  (S with type *term* = *T.term*
    and type *substitution* = *T.substitution*)

open *Gen_terms*
open *User_signatures*

val *normalize* :
  symbol_entry array $\rightarrow$ (*symbol_id term* $\times$ *symbol_id term*) *list* $\rightarrow$
    symbol_id term $\rightarrow$ *symbol_id term*

Chapter 14

Library `completion`

This is a completion library, to be documented.

14.1 Module *Standard_critical_pairs*

module type S =
sig
  type *term*
  type *rule*

  val *critical_pairs* :
    $(term \rightarrow bool) \rightarrow (int \rightarrow term \rightarrow bool) \rightarrow rule \rightarrow rule \rightarrow (term \times term)\ list$

  val *self_critical_pairs* :
    $(term \rightarrow bool) \rightarrow (int \rightarrow term \rightarrow bool) \rightarrow rule \rightarrow (term \times term)\ list$

end

module *Make* (T : *Term_algebra.TermAlgebra*) :
  (S with type *term* = *T.term*
    and type *rule* = *T.rule*)

14.2 Module *Ac_critical_pairs*

module type S =
sig
  type *term*
  type *substitution*
  type *rule*

  val *critical_pairs* :
    $(term \times term \rightarrow substitution\ list) \rightarrow (term \times term \rightarrow substitution\ list \rightarrow unit) \rightarrow$
      $(term \rightarrow bool) \rightarrow (int \rightarrow term \rightarrow bool) \rightarrow rule \rightarrow rule \rightarrow (term \times term)\ list$

  val *self_critical_pairs* :
    $(term \times term \rightarrow substitution\ list) \rightarrow (term \times term \rightarrow substitution\ list \rightarrow unit) \rightarrow$
      $(term \rightarrow bool) \rightarrow (int \rightarrow term \rightarrow bool) \rightarrow rule \rightarrow (term \times term)\ list$

end

module *Make* (T : *Term_algebra.TermAlgebra*) :
  (S with type *term* = *T.term*
    and type *substitution* = *T.substitution*
    and type *rule* = *T.rule*)

14.3 Module *Abstract_rewriting*

****

This module provides parameterized functions to complete a rewriting system. They are intended to apply as well on terms and on words.

CiME Project - Démons research team - LRI - Université Paris XI

$Id : abstract_rewriting.mli, v1.33 2003/09/12 16:07:05 contejea Exp$

****

open *Variables*
open *Orderings_generalities*

module type S =
  sig
    type *t*
    type *rule*
    type *compiled_rules*
    type *compiled_pairs*
    type *'a compiled_data*
    type *index_for_unif*
    module *Tset* : (*Ordered_sets.OrderedSet* with type *'a elt* = *t*)
    module *Ttable* : (*Hashtbl.S* with type *key* = *t*)
    val *my_o* : *t ordering*
    val *penalty_of_non_oriented_eq* : *int*
    val *smallest_constant* : *t option*
    val *a_variable_of_t* : *t* → *t option*
    val *variables_of_t* : *t* → *unit VarSet.t*
    val *equals* : *t* → *t* → *bool*
    val *make_rule* : *int* → *t option* → *t* → *t* → *rule*
    val *lhs_of_rule* : *rule* → *t*
    val *rhs_of_rule* : *rule* → *t*
    val *alt_rhs_of_rule* : *rule* → *t option*
    val *is_oriented* : *rule* → *bool*
    val *is_encompassed_by_a_t* : *unit compiled_data* → *t* → *bool*
    val *compile* : *rule list* → *compiled_rules*

(∗ *update_compiled_rules sign compiled_rules to_add to_remove to_replace unchanged* compiles the rules corresponding to the list of rules *to_add*, *unchanged*, and the second components of *to_replace*, provided that *compiled_rules* corresponds to the list of rules *unchanged*, *to_remove* and the first components of *to_replace*. ∗)

    val *update_compiled_rules* :
      compiled_rules → *rule* → *rule list* → (*rule* × *rule*) *list* →
        rule list → *compiled_rules*
    val *compile_pairs* : *rule list* → *compiled_pairs*
    val *compile_data* : (*t* × *'a*) *list* → *'a compiled_data*
    val *compile_for_unification* : *rule list* → *index_for_unif*

normalize find memo r t returns the normal form of *t*. *force_normalize* does the same but raises *Irreducible* if already in normal form.

    val *normalize* :
      (*t* → *t*) → (*t* → *t* → *unit*) → *compiled_rules* → *t* → *t*

    exception *Irreducible*

    val *force_normalize* :
      (*t* → *t*) → (*t* → *t* → *unit*) → *compiled_rules* → *t* → *t*

self_critical_pairs find cr already_computed_cp r returns the list of critical pairs of *r* into itself. The standard elimination criterion is used, the unifier of the left hand sides has to be irreducible wrt the rules *cr*. *find* is a function which tries to find the normal form of a term wrt *cr*.

val *self_critical_pairs* :
  $(t \to t) \to$ *compiled_rules* $\to$ (*int* $\to$ *int* $\to$ *bool*) $\to$ *rule* $\to$ $(t \times t)$ *list*

critical_pairs find cr already_computed_cp r1 r2 returns the list of critical pairs between *r1* and *r2*. *r1* and *r2* are supposed to be different, use *self_critical_pairs* for computing critical pairs of a rule into itself. The standard elimination criterion is used, the unifier of the left hand sides has to be irreducible wrt the rules *cr*. *find* is a function which tries to find the normal form of a term wrt *cr*.

val *critical_pairs* :
  $(t \to t) \to$ *compiled_rules* $\to$ (*int* $\to$ *int* $\to$ *bool*) $\to$ *rule* $\to$ *rule* $\to$ $(t \times t)$ *list*

val *init_narrow* : $t \to t \to ((t \times t) \times (t \times t) \times (t \times t))$

val *is_a_nar_eq* : $t \to t \to$ *bool*

val *pair_size* : $t \to t \to$ *int* $\times$ *int*

val *is_an_instance* : $(t \times t) \to (t \times t) \to$ *bool*

is_encompassed_by_a_pair p1 p2 t1 t2 returns true whenever there exists a context $C[]$, and a substitution sigma such that $t1 = C[p1\ sigma]$ and $t2 = C[p2\ sigma]$.

val *is_encompassed_by_a_pair* : *compiled_pairs* $\to t \to t \to$ *bool*

regular_pair c t1 t2 returns the term *t2'* such that all variables in *t2* which do not occur in the term *t1* have been substituted by the constant term *c*. It raises *Not_found* whenever *t2'* is identical to *t2*.

val *regular_pair* : $t \to t \to t \to t$

canonize_pairs l returns the list *l* where all pairs of terms have been put in a kind of canonical form with respect to name of variables. This is only for having a better printing of rules, it can be the identity function.

val *canonize_pairs* : $(t \times t)$ *list* $\to$ $(t \times t)$ *list*
val *print_t* : $t \to$ *unit*
val *print_equation_set* : $(t \times t)$ *list* $\to$ *unit*
val *print_rewrite_rule* : *rule* $\to$ *unit*

val *is_a_c_equality* : $t \to t \to$ *bool*
val *is_an_ac_equality* : $t \to t \to$ *bool*
val *nb_shared_symbols* : $t \to t \to$ *int*
val *print_stats* : *unit* $\to$ *unit*
end

module *MakeWordRewriting* :
  functor
    (*SWords* :
      sig
        val *my_sign* : *String_signatures.symbol_id String_signatures.word_signature*
        val *my_o* : *String_signatures.symbol_id Words.word ordering*
      end) $\to$
      (*S* with type *t* = *String_signatures.symbol_id Words.word*
        and type *rule* = *String_signatures.symbol_id String_rewriting.rewrite_rule*)

module *MakePWordRewriting* :
  functor
    (*PWords* :
      sig
        val *my_sign* : *Parameterized_signatures.parameterized_signature*
        val *my_o* : *Parameterized_words.word ordering*
      end) $\to$
      (*S* with type *t* = *Parameterized_words.word*
        and type *rule* = *Parameterized_rewriting.rewrite_rule*)

14.4 Module *Confluence*

```
open User_signatures
open Variables
open Gen_terms

module type S =
sig
  type rule
  val is_confluent : rule list → bool
  val print_all_critical_pairs : rule list → unit
end

module StandardConfluence
  (T : Term_algebra.TermAlgebra) :
  (S with type rule = T.rule)

module ACConfluence
  (T : Term_algebra.TermAlgebra) :
  (S with type rule = T.rule)

val is_confluent :
  symbol_entry array → (symbol_id term × symbol_id term) list → bool

val print_all_critical_pairs :
  symbol_entry array → (symbol_id term × symbol_id term) list → unit
```

14.5 Module *Kb*

```
open Gen_terms
open User_signatures
open Orderings_generalities

val coeff_size : int ref
val coeff_age : int ref
val verbose : int ref

type builtin_th

module type KB =
  sig
    type t
    type simple_rule
    exception KB_th_detected of builtin_th × t × ((t × t) list) × ((t × t) list)
    val verbose : int ref
    val o : t ordering
    val complete : (t × t) list → (t × t) list → simple_rule list
    val prove_conj_by_comp : (t × t) list → (t × t) list → (t × t) list → bool
  end

module Make (R : Abstract_rewriting.S) (Sco : Best_pair.Score) :
  (KB with type t = R.t
      and type simple_rule = R.rule)

module StandardKBCompletion
  (T : Term_algebra.TermAlgebra)
  (O : sig val my_o : T.term ordering end) :
  (KB with type t = T.term
       and type simple_rule = T.rule)
```

```
module ACCompletion
  (T : Term_algebra.TermAlgebra)
  (O : sig val my_o : T.term ordering end) :
  (KB with type t = T.term
        and type simple_rule = T.rule)

module OrderedKBCompletion
  (T : Term_algebra.TermAlgebra)
  (O : sig val my_o : T.term ordering end)
  (Sco : Best_pair.Score) :
  (KB with type t = T.term
        and type simple_rule = T.rule)

module OrderedACCompletion
  (T : Term_algebra.TermAlgebra)
  (O : sig val my_o : T.term ordering end)
  (Sco : Best_pair.Score) :
  (KB with type t = T.term
        and type simple_rule = T.rule)

module WordCompletion
  (SWords :
    sig
      val my_sign : String_signatures.symbol_id String_signatures.word_signature
      val my_o : String_signatures.symbol_id Words.word ordering
    end) :
  (KB with type t = String_signatures.symbol_id Words.word
      and type simple_rule =
        String_signatures.symbol_id String_rewriting.rewrite_rule)

module PWordCompletion
  (PWords :
    sig
      val my_sign : Parameterized_signatures.parameterized_signature
      val my_o : Parameterized_words.word ordering
    end) :
  (KB with type t = Parameterized_words.word
      and type simple_rule = Parameterized_rewriting.rewrite_rule)

val complete :
  symbol_entry array → symbol_id Term_orderings.t →
    (symbol_id term × symbol_id term) list → (symbol_id term × symbol_id term) list

val prove_conjectures :
  symbol_entry array → Variables.user_variables →
    (symbol_id term × symbol_id term) list → (symbol_id term × symbol_id term) list → bool
```

Chapter 15

Library `eq_proof`

This library provides ...

15.1 Module *Proof*

module type P $=$
  sig
    type t
    val $\mathit{prove\_conj\_without\_strategy}$: $(t \times t)$ $\mathit{list} \rightarrow$ $(t \times t)$ $\mathit{list} \rightarrow$ bool
  end

Chapter 16

Library `coq_interface`

This library provides . . .

16.1 Module *Trace*

This module defines the trace used when normalizing a term: it basically consists in remembering which and where rules are applied to reduce the term.

module type S =
sig
  type *rule*
  class t : object
    method *get* : $(int \times int\ list)\ list$
    method *add* : $rule \rightarrow Positions.position \rightarrow unit$
    method *to_string* : $string\ array \rightarrow string$
    method *print* : $unit \rightarrow unit$
  end
end

module *Make* (T : *Term_algebra.TermAlgebra*) :
  (S with type *rule* = *T.rule*)

16.2 Module *Traced_rewriting*

module type S =
sig

  type *term*
  type *substitution*
  type *rule*
  type *dnet*
  type *trace*
  (∗ *new_compiled_rewrite_at_top* is the same as *Standard_rewriting.compiled_rewrite_at_top*, except it returns the whole rewrite rule applied for the reduction instead of just the right member. ∗)
  val *new_compiled_rewrite_at_top* : $dnet \rightarrow term \rightarrow substitution \times rule$

traced_innermost_normalize sigma red t, with *red* returning the whole (numbered) rewrite rule, is the same as *Innermost.innermost_normalize red' t*, except that it returns a trace of the normalization as well.

  val *traced_innermost_normalize* :
    $(term \rightarrow substitution \times rule) \rightarrow term \rightarrow term \times trace$

end

module *Make*
  $(T$: *Term_algebra.TermAlgebra*$)$
  $(D$: *Full_dnet.S* with type *term* = *T.term*
  and type *rule* = *T.rule*
  and type *substitution* = *T.substitution*$)$
  $(Tr$: *Trace.S* with type *rule* = *T.rule*$)$:
  $(S$ with type *term* = *T.term*
  and type *substitution* = *T.substitution*
  and type *dnet* = *T.rule D.dnet*
  and type *trace* = *Tr.t*$)$

16.3 Module *Coq_syntax*

Chapter 17

Library **Termination**

This library contains the needed stuff for either checking the termination of a rewrite system, or even trying to find a termination proof of a rewrite system.

17.1 Module *Basic_criterion*

$($*all_decreasing* $>$ $R)$ returns true if all rules of R are strictly decreasing w.r.t to $\rangle]$. *If* $\neg$, *prints the first rule found that does* $\neg$ *decrease* and *returns* [false].

```
val all_decreasing :
  Terms.term Orderings.ordering -> Equations.regle list -> bool;;
```

$($*all_decreasing_constraints* $R)$ returns the list of ordering constraints that should be satisfied for that each rule of R is strictly decreasing.

val *all_decreasing_constraints* :
  'symbol Rewrite_rules.t *list* $\rightarrow$
  'symbol Termination_constraints.conj_termination_constraints

17.2 Module *Dependency_pairs_criteria*

17.2.1 Simple dependency pairs criterion

$($*simple_dependency_pair_criterion* $R)$ returns the set of ordering existence constraints given by the simple dependency pairs termination criterion on R, which is the following :

1. let D be the set of *defined* symbols of R, that is the set of symbols occuring at root of a lefthand side of a rule of R ;
2. a *dependency pair* is any pair of terms (s,t) such that s is the lefthand side of a rule $s \rightarrow r$ in R and t is any subterm of r whose root symbol is defined.
3. a condition for R terminating is that there is a weakly monotonic ordering $\geq$ such that $l \geq r$ for all rules of R and $s > t$ for all dependency pair of R ;
4. $\geq$ must be AC-compatible.

val *compute_defined_symbols* :
  $($*'symbol* Rewrite_rules.t$)$ *list* $\rightarrow$
    'symbol Signatures.SymbolSet.t

val *all_rules_weakly_decreasing* :
  (*'symbol* Rewrite_rules.t) *list* →
    'symbol Termination_constraints.AtomicTerminationConstraintSet.t

val *all_dp_strictly_decreasing* :
  (*'symbol* Gen_terms.term × *'symbol* Gen_terms.term) *list* →
    'symbol Termination_constraints.AtomicTerminationConstraintSet.t

val *compute_dependency_pairs* :
  'symbol Signatures.SymbolSet.t →
    (*'symbol* Rewrite_rules.t) *list* →
      (*'symbol* Gen_terms.term × *'symbol* Gen_terms.term) *list*

val *simple_dependency_pair_criterion* :
  'symbol #Signatures.signature →
    'symbol Rewrite_rules.t *list* →
      'symbol Termination_constraints.conj_termination_constraints

17.3 Module *Generic_polynomials*

type *polyinterptype* = *Linear* | *Simple* | *Simple_Mixed* | *Quadratic*

val *use_interp* : *polyinterptype ref*

module *GenericPolynomial* defines polynomials over the coefficients ring : polynomials of finite domain variables ; and variables indexed by integers.

module *Fd_polynomials_ring* : *Polynomials.RingType*
  with
    type *coef* = *Non_linear_solving.Fd_polynomials.poly*

module *GenericPolynomials* : *Polynomials.PolynomialType*
  with
    module *Base_ring* = *Fd_polynomials_ring*
  and
    type *variable* = *int*

(*print_generic_polynomial p*) outputs *p* to standard output

val *print_generic_polynomial* : *GenericPolynomials.poly* → *unit*

(*print_interp i*) outputs the polynomial interpretation *i* (that is a map which associates polynomials to symbols) to standard output

val *print_interp* :
  'a #Signatures.signature
  → (*'a*, *GenericPolynomials.poly*) *Signatures.SymbolMap.t* → *unit*

(*polynomial_constraints_of_termination_constraints sigma i c*) returns a pair (*interp*, *c1*, *c2*, *c3*) where *interp* is a set of generic polynomial interpretations of symbols in *c* and *c1*, *c2* and *c3* are lists of diophantine constraints over variables of *interp* such that :

- if *c1* then interp is AC-compatible
- if *c2* then interp is strictly monotonic
- if *c3* then *c*

sigma is the signature on which constraints are defined, and *i* is a 'predefined" interpretation for some symbols, that is the function search only for an interpretation *interp* that extends *i*. Of course *i* may be empty.

val *polynomial_constraints_of_termination_constraints* :
  'a #Signatures.signature
  → (*'a*, *GenericPolynomials.poly*) *Signatures.SymbolMap.t*
    → *'a Termination_constraints.conj_termination_constraints*
      →
        (*'a*, *GenericPolynomials.poly*) *Signatures.SymbolMap.t*
        × *Non_linear_solving.non_linear_constraint list*
        × *Non_linear_solving.non_linear_constraint list*
        × *Non_linear_solving.non_linear_constraint list*

(*specialize_polynomial p s*) returns the polynomial obtained by instantiation of finite domain variables in coefficients of *p* by their value given by the map *s*

val *specialize_polynomial* :
  GenericPolynomials.poly
  → (*Finite_domains.fd_var_id*, *Numbers.t*) *Finite_domains.Fd_var_map.t*
    → *GenericPolynomials.poly*

val *instantiate_interp* :
  (*'symbol*, *GenericPolynomials.poly*) *Signatures.SymbolMap.t* →
    (*Finite_domains.fd_var_id*, *Numbers.t*) *Finite_domains.Fd_var_map.t* →
      (*'symbol*, *Poly_interp.IntPolynomials.poly*) *Signatures.SymbolMap.t*

(*solve_constraint w sigma c*) tries to solve the constraint *c* on signature *sigma*, using each kind of interpretations in *param_list* in order. If *w* is true, searches for strictly monotonic ordering only.

Returns the interpretation found, or raises *No_proof_found*.

param_list is the list of kinds of polynomial interpretation to look for, in order. Default `"linear"`,1;`"linear"`,2;`"simple"`,2

val *param_list* : (*polyinterptype* × *int*) *list ref*

val *solve_constraint* :
  bool → *'a #Signatures.signature*
    → *'a Termination_constraints.conj_termination_constraints*
      → *'a Poly_interp.t*

17.4 Module *Genpoly_lexer*

val *token* : *Lexing.lexbuf* → *Genpoly_parser.token*

17.5 Lexer *Genpoly_lexer*

{

open *Genpoly_syntax*

open *Genpoly_parser*

open *Marked_dp_criteria*

exception *Syntax_error* of *string*

```
let interp s =
  try
    let is_marked =
      String.get s 0 = '\'' ∧ String.get s (pred (String.length s)) = '`'
    in
    let s' =
      if is_marked
      then String.sub s 1 (pred (pred (String.length s)))
      else s
    in
    let f = !current_signature#symbol_of_string s
    in
    let g =
      if is_marked
      then Marked f
      else Original f
    in INTERP(g)
  with
      Not_found →
        raise (Syntax_error ("undefined␣identifier␣"^s))

}

rule token = parse
    [' ''\t''\n']+ { token lexbuf }
  |['a'-'z''A'-'Z']['0'-'9']*
                              { VAR(Lexing.lexeme lexbuf) }
  |'[' { symbol lexbuf }
  |';' { SEMICOLON }
  |'+' { PLUS }
  |'-' { MINUS }
  |'.' { MULT }
  |'^' { EXP }
  |['0'-'9']+ { INT (Num.num_of_string (Lexing.lexeme lexbuf)) }
  |'(' { PARGAUCHE }
  |')' { PARDROITE }
  |eof { EOF }

and symbol = parse
    [' ''\t''\n']+ { symbol lexbuf }
  |[^']'' ''\t''\n']+ { let f = interp (Lexing.lexeme lexbuf)
                                in skip_bracket lexbuf; f }

and skip_bracket = parse
    [' ''\t''\n']+ { skip_bracket lexbuf }
  |']' { () }
```

17.6 Parser *Genpoly_parser*

Header

```
open Signatures

open Generic_polynomials
```

```
open Genpoly_syntax

open Non_linear_solving

open Finite_domains

open Poly_interp

exception Syntax_error of string

let gen_var s =
  try
    let n = Listutils.index s !current_poly_vars
    in GenericPolynomials.var n
  with
      Not_found →
        GenericPolynomials.cte
          (Fd_polynomials.var (fd_var_id_of_string s))

let var s =
  try
    let n = Listutils.index s !current_poly_vars
    in IntPolynomials.var n
  with
      Not_found → raise (Syntax_error ("undefined␣variable␣"^s))
```

Token declarations

```
%token <string> VAR
%token <User_signatures.symbol_id Marked_dp_criteria.dupl> INTERP
%token PARGAUCHE PARDROITE SEMICOLON EQUAL COMMA EOF
%token PLUS MINUS EXP MULT
%token <Num.num> INT

%start gen_poly_entry
%type <Generic_polynomials.GenericPolynomials.poly> gen_poly_entry

%start poly_interp_entry
%type <(User_signatures.symbol_id Marked_dp_criteria.dupl, Poly_interp.IntPolynomials.poly) Signatures.SymbolMap.t > poly_inte

%left PLUS MINUS
%left MULT
%nonassoc UMINUS
%right EXP
```

Grammar rules

```
gen_poly_entry ::=
  gen_poly EOF { $1 }
```

```
gen_poly ::=
   VAR { gen_var $1 }
| INT { GenericPolynomials.cte (Fd_polynomials.cte $1) }
| PARGAUCHE gen_poly PARDROITE { $2 }
| gen_poly PLUS gen_poly { GenericPolynomials.add $1 $3 }
| gen_poly MINUS gen_poly { GenericPolynomials.sub $1 $3 }
| MINUS gen_poly %prec UMINUS { GenericPolynomials.minus $2 }
| gen_poly MULT gen_poly { GenericPolynomials.mult $1 $3 }
| gen_poly EXP INT
      { try
           GenericPolynomials.power $1 (Num.int_of_num $3)
        with
           Failure("int_of_big_int") →
             failwith "Exponent too large"
      }

poly_interp_entry ::=
   interp EOF { $1 }

interp ::=
   /* epsilon */ { SymbolMap.empty }
| symbol_interp EQUAL poly SEMICOLON interp
                                          { SymbolMap.add $1 $3 $5 }

symbol_interp ::=
   INTERP { current_poly_vars := []; $1 }
| INTERP PARGAUCHE vars { current_poly_vars := $3; $1 }

vars ::=
   VAR PARDROITE { [$1] }
| VAR COMMA vars { $1::$3 }

poly ::=
   VAR { var $1 }
| INT { IntPolynomials.cte $1 }
| PARGAUCHE poly PARDROITE { $2 }
| poly PLUS poly { IntPolynomials.add $1 $3 }
| poly MINUS poly { IntPolynomials.sub $1 $3 }
| MINUS poly %prec UMINUS { IntPolynomials.minus $2 }
| poly MULT poly { IntPolynomials.mult $1 $3 }
| poly EXP INT
      { try
           IntPolynomials.power $1 (Num.int_of_num $3)
        with
           Failure("int_of_big_int") →
             failwith "Exponent too large"
      }
```

17.7 Module *Genpoly_syntax*

current_signature is the signature to be used for parsing function symbols

```
val current_signature : User_signatures.user_signature ref
```

current_poly_vars is the current list of polynomial variables in parsing generic polynomials

```
val current_poly_vars : string list ref
```

17.8 Module *Marked_dp_criteria*

```
open Gen_terms
open Orderings_generalities
```

(*var_multiplicities t*) returns an association table from variables of t to their multiplicity, i.e., the number of their occurrences at depth 1 in t.

```
val var_multiplicities :
  'symbol term → (Variables.var_id,int) Variables.VarMap.t

type 'symbol dupl =
  | Original of 'symbol
  | Marked of 'symbol

val mark_root_symbol : 'symbol term → ('symbol dupl) term

class ['a] marked_signature : ('a #Signatures.signature) →
  object
    method arity : 'a dupl → int
    method is_ac : 'a dupl → bool
    method is_commutative : 'a dupl → bool
    method is_free : 'a dupl → bool
    method contains_ac_symbols : bool
    method contains_only_free_symbols : bool
    method string_of_symbol : 'a dupl → string
    method symbol_fix : 'a dupl → Signatures.symbol_fix
  end

val signature_with_marks :
  'symbol #Signatures.signature → 'symbol marked_signature

val ac_rhs_rules :
    'a #Signatures.signature →
      'a dupl #Signatures.signature → 'a Signatures.SymbolSet.t →
        'a Rewrite_rules.t list → 'a dupl Rewrite_rules.t list

val copy_rule :
  'a Rewrite_rules.t → 'a dupl Rewrite_rules.t

val remove_markings_rules :
  'a #Signatures.signature →
    'a Signatures.SymbolSet.t → 'a dupl Rewrite_rules.t list

val compute_dependency_pairs_with_markings :
  bool →
  ('a #Signatures.signature) →
      'a Signatures.SymbolSet.t →
        'a Rewrite_rules.t list →
          ('a dupl term × 'a dupl term) list

val dependency_pair_criterion_with_markings :
  bool →
  ('symbol #Signatures.signature) →
      'symbol Rewrite_rules.t list →
        ('symbol dupl) Termination_constraints.conj_termination_constraints
```

17.9 Module *Old_basic_criterion*

(*all_decreasing* > *R*) returns true if all rules of *R* are strictly decreasing w.r.t to ⟩]. *If* ¬, *prints the first rule found that does* ¬ *decrease* and *returns* [false].

```
val all_decreasing :
  Terms.term Orderings.ordering → Equations.regle list → bool
```

17.10 Lexer *Poly_lexer*

```
{
  open Poly_parser
  open Poly_syntax
  open Generic_polynomials
  open Signatures

  exception Invalid_char of string

  let ident s =
    try
      let is_marked =
        String.get s 0 = '\'' ∧ String.get s (pred (String.length s)) = '`'
      in
      let s' =
        if is_marked
        then String.sub s 1 (pred (pred (String.length s)))
        else s
      in
      let f = !current_signature#symbol_of_string s
      in
        if is_marked
        then Marked f
        else Original f
    with
        Not_found →
          Errors.semantical_error ("undefined␣identifier␣"^s)
}

rule token = parse
    [' ''\t''\n'] { token lexbuf }
  | ',' { COMMA }
  | ';' { SEMICOLON }
  | "->" { ARROW }
  | '(' { OPENPAR }
  | ')' { CLOSEPAR }
  | ['a'-'z''A'-'Z''0'-'9']['a'-'z''A'-'Z''0'-'9''_''\'']*
  | ['^''+''.''&''*''-''/''!''?''@''~''#'] ('_'(['a'-'z''A'-'Z''0'-'9']+))*
                                  { ident(Lexing.lexeme lexbuf) }
  | _ { raise (Invalid_char (Lexing.lexeme lexbuf)) }
  | eof { EOF }
```

17.11 Module *Termination_constraints*

An atomic termination constraint is either $s = t$, $s > t$ or $s >= t$ for two terms s and t

type *constraint_op_type* = *Equal* | *Greater* | *Greater_or_equal*

type *'symbol* atomic_termination_constraint =
{
left : *'symbol* term;
right : *'symbol* term;
constraint_op : *constraint_op_type*;
}

a termination constraint is a disjunction of conjonction of atomic constraints.

module *AtomicTerminationConstraintSet* : *Ordered_sets.OrderedSet*
with type *'symbol elt* = *'symbol* atomic_termination_constraint

type *'symbol* conj_termination_constraints =
'symbol AtomicTerminationConstraintSet.t

val *print_atomic_termination_constraint* :
'symbol #Signatures.signature →
Variables.user_variables → *'symbol* atomic_termination_constraint
→ *unit*

val *print_conj_termination_constraints* :
'symbol #Signatures.signature →
Variables.user_variables → *'symbol* conj_termination_constraints
→ *unit*

(*set_of_symbols* c) returns the set of symbols occuring in the list of constraints c

val *set_of_symbols* :
'symbol conj_termination_constraints → *'symbol* SymbolSet.t

val *conj_of_order_constraints* :
'symbol Order_constraints.formula →
'symbol conj_termination_constraints

17.11.1 Checking constraints

val *check_conj_constraints* :
'symbol #Signatures.signature →
Variables.user_variables →
'symbol conj_termination_constraints →
'symbol Gen_terms.term Orderings_generalities.ordering → *unit*

val *display_conj_termination_constraints* :
(*'symbol* #Signatures.signature) →
Variables.user_variables →
'symbol conj_termination_constraints
→ *unit*

17.12 Module *Minimal_split*

A module of graphs of symbols with a special function *scc* which returns strongly connected components of its argument (see module Graph).

```
module Symgraph :
  sig
    type 'symbol elt = 'symbol Signatures.SymbolOrd.t
    and 'symbol t = 'symbol Graph.Make(Signatures.SymbolOrd).t
    exception NodeNotInGraph
    exception NodeAlreadyInGraph
    val empty : 'symbol t
    val add_node : 'symbol t → 'symbol elt → 'symbol t
    val is_node : 'symbol t → 'symbol elt → bool
    val add_edge : 'symbol t → 'symbol elt → 'symbol elt → 'symbol t
    val is_edge : 'symbol t → 'symbol elt → 'symbol elt → bool
    val del_node : 'symbol t → 'symbol elt → 'symbol t
    val del_edge : 'symbol t → 'symbol elt → 'symbol elt → 'symbol t
    val node_list : 'symbol t → 'symbol elt list
    val edge_list : 'symbol t → ('symbol elt × 'symbol elt) list
    val neighbours_list : 'symbol t → 'symbol elt → 'symbol elt list
    val iter_node : 'symbol t → ('symbol elt → unit) → unit
    val iter_edge : 'symbol t → ('symbol elt → 'symbol elt → unit) → unit
    val fold : ('symbol elt → 'b → 'b) → 'b → 'symbol t → 'b
    val transpose : 'symbol t → 'symbol t
    val scc : 'symbol t → 'symbol elt list list
  end
```

```
val graph_from_a_list :
  'symbol Rewrite_rules.t list →
    'symbol Symgraph.elt Symgraph.t
```

```
val packs_from_a_list :
  'symbol Rewrite_rules.t list →
    'symbol Symgraph.elt Symgraph.elt list list
```

```
val packs_and_rules_from_packs :
  'symbol Rewrite_rules.t list →
    'symbol list list → ('symbol list × 'symbol Rewrite_rules.t list) list
```

minimal_split sigma vars r returns a list of minimal modules in htrs *r*.

```
val minimal_split :
  'symbol #Signatures.signature → Hierarchical_trs.htrs → Hierarchical_trs.htrs list
```

17.13 Module *Relative_dp*

```
open Rewrite_rules
```

```
open Marked_dp_criteria
```

```
val mdp_of_rules :
  ('symbol Rewrite_rules.t) list →
    ('symbol Gen_terms.term × 'symbol Gen_terms.term ) list
```

```
val mdp_of_rules_marks :
  bool →
  'symbol #Signatures.signature → ('symbol Rewrite_rules.t) list →
      (('symbol dupl) Gen_terms.term × ('symbol dupl) Gen_terms.term ) list
```

```
val modular_dp_criterion :
  Hierarchical_trs.htrs →
    (Hierarchical_signatures.hsymbol) Termination_constraints.conj_termination_constraints
```

val *modular_dp_criterion_marks* :
  bool →
    Hierarchical_trs.htrs →
      Hierarchical_signatures.hsymbol Marked_dp_criteria.dupl #Signatures.signature →
        Hierarchical_signatures.hsymbol dupl Termination_constraints.conj_termination_constraints

17.14 Module *Termination_expert*

This module is the termination expert. The main function looks for a termination proof for a given rewrite system.

17.14.1 Global configuration

the following global variables controls which criterion the termination expert uses.

When *standard_criterion* is true, the termination criterion is the standard one: each must decrease with respect to a monotonic well-founded ordering. When it is false, the dependency pair criteria are used. See module *Dependency_pair_criteria* 17.2. default true

val *standard_criterion* : *bool ref*

When *dp_criterion_uses_markings* is true, DP criteria with marks are used. default true

val *dp_criterion_uses_markings* : *bool ref*

When *dp_criterion_uses_markings_ac* is true, AC symbols are marked in DP criteria with marks. default false

val *dp_criterion_uses_markings_ac* : *bool ref*

When *dp_criterion_uses_graph* is true, DP criteria with graph are used. default true

val *dp_criterion_uses_graph* : *bool ref*

When *enable_split* is true, relative dp criteria are used. default false

val *enable_split* : *bool ref*

The verbose level.

1. the termination constraints to solve are displayed.

2. ...

Default is 0

val *verbose* : *int ref*

17.14.2 Computing depencency pairs

show_pairs F H MH X R computes the dependency pairs of the (F,X)-TRS R, and displays them. If the marked criterion is currently selected, then marked pairs are computed. If the graph criterion is currently selected, also displays the approximated dependency graph. H is an hashconsing table for standard terms and MH is an hashconsing table for marked terms.

val *show_pairs* :
  (*'symbol #Signatures.signature*) →
      Variables.user_variables →
        (*'symbol Rewrite_rules.t*) *list* → *unit*

17.14.3 Computing termination constraints

```
val compute_termination_constraints_without_marks :
  ('symbol #Signatures.signature) →
    Variables.user_variables →
      ('symbol Rewrite_rules.t) list →
        'symbol Termination_constraints.conj_termination_constraints
```

```
val compute_termination_constraints_with_marks :
  ('symbol #Signatures.signature) → 'symbol Marked_dp_criteria.dupl #Signatures.signature →
    Variables.user_variables →
      ('symbol Rewrite_rules.t) list →
        'symbol Marked_dp_criteria.dupl
          Termination_constraints.conj_termination_constraints
```

17.14.4 The main function

expert F X R looks for a termination proof of the (F,X)-TRS R. Displays the results.

```
type 'a graph_termination_proof =
     (('a Gen_terms.term × 'a Gen_terms.term) Labelled_graphs.graph
      × 'a termination_proof_component) list
```

```
and 'a termination_proof_component =
  | Graph_simple_criterion of 'a Poly_interp.t
  | Graph_complex_criterion of
        'a Poly_interp.t × 'a graph_termination_proof
```

```
type 'a termination_proof =
  | Failed
  | Standard of 'a Signatures.signature × 'a Poly_interp.t
  | Dp_graph_nomark of 'a Signatures.signature × 'a graph_termination_proof
  | Dp_nograph_nomark of 'a Signatures.signature × 'a Poly_interp.t
  | Dp_graph_mark of
        ('a Marked_dp_criteria.dupl) Signatures.signature ×
        ('a Marked_dp_criteria.dupl) graph_termination_proof
  | Dp_nograph_mark of
        ('a Marked_dp_criteria.dupl) Signatures.signature ×
        ('a Marked_dp_criteria.dupl) Poly_interp.t
```

```
val print_proof : 'a termination_proof → unit
```

```
val latex_print_proof : string → 'a termination_proof → unit
```

```
exception No_proof_found
```

```
val expert :
     ('symbol #Signatures.signature) →
       Variables.user_variables →
         ('symbol Rewrite_rules.t) list →
           'symbol termination_proof
```

type *modular_termination_proof* =
  | *Modular_nograph_nomark* of
      Hierarchical_signatures.hsymbol Signatures.signature ×
      Hierarchical_signatures.hsymbol Poly_interp.t
  | *Modular_nograph_mark* of
      (*Hierarchical_signatures.hsymbol Marked_dp_criteria.dupl*)
        Signatures.signature ×
      (*Hierarchical_signatures.hsymbol Marked_dp_criteria.dupl*) *Poly_interp.t*
  | *Modular_graph_nomark* of
      Hierarchical_signatures.hsymbol Signatures.signature ×
      Hierarchical_signatures.hsymbol graph_termination_proof
  | *Modular_graph_mark* of
      (*Hierarchical_signatures.hsymbol Marked_dp_criteria.dupl*)
        Signatures.signature ×
      (*Hierarchical_signatures.hsymbol Marked_dp_criteria.dupl*) *graph_termination_proof*

val *print_module_termination_proof* : *modular_termination_proof* → *unit*

val *print_modular_proof* :
  (*Hierarchical_trs.htrs* × *modular_termination_proof*) *list* → *unit*

val *latex_print_modular_proof* :
  string → (*Hierarchical_trs.htrs* × *modular_termination_proof*) *list* → *unit*

val *modular_expert* :
      Variables.user_variables →
        Hierarchical_trs.htrs →
              (*Hierarchical_trs.htrs* × *modular_termination_proof*) *list*

17.15 Module *Automaton*

Obsolete header:
CiME Project - Démons research team - LRI - Université Paris XI
$Id : automaton.mli, v1.22001/04/2013 : 42 : 09marcheExp$
****
Les automates ascendants d'arbres avec condition sur les sous-termes immediats pour des termes sans symboles AC. Les conditions sont de la forme : un entier = le numero de l'argument (de 0 a (arite f)-1 la comparaison a effectuer entre les sous-termes un entier = le numero de l'autre sous-terme a comparer ex : (2,=,3) verifie que les 2eme et 3eme arguments sont egaux

type *t* = *int*

open *Types*

type *var_id* = *int*

exception *VARIABLE* of *var_id* (∗state∗)

type (*'a*, *'b*) *u*

type *'SYMBOL internal_term* =
    Var of *var_id*
  | *App* of (*'SYMBOL* × (*'SYMBOL internal_term list*))

type *'SYMBOL formula* = *OR* of *'SYMBOL formula* × *'SYMBOL formula*
    | *AND* of *'SYMBOL formula* × *'SYMBOL formula*
    | *EQ* of ((*'SYMBOL internal_term*) *list* × (*'SYMBOL internal_term*) *list*)
    | *EXIST* of ((*'SYMBOL internal_term*) *list* × *'SYMBOL formula*)
    | *TRUE* | *FALSE*

type (*'STATE*) *epsilon_rule* = { *initial_state* : *'STATE* ;
final_state : *'STATE*; }

type (*'SYMBOL*, *'TERM*, *'STATE*) *conditional_rewrite_rule* = {*tete* : *'SYMBOL*;
etats_initiateurs : *'STATE list* ;
conditions : (*int* × (*'TERM* → *'TERM* → *bool*) × *int*) *list* ;
etat_final : *'STATE* }

type (*'SYMBOL*, *'TERM*, *'TYPE*, *'STATE*) *automata* = { *alphabet* : *'SYMBOL list* ;
typage : (*'SYMBOL* → *'TYPE*) ;
tete_de_terme : *'TERM* → *'SYMBOL* ;
liste_des_sous_termes : *'TERM* → (*'TERM list*);
etats : *'STATE list* ;
etats_finaux : *'STATE list* ;
etat_poubelle : *'STATE* ;
epsilon : (*'STATE*) *epsilon_rule list* ;
regles : ((*'SYMBOL*, *'TERM*, *'STATE*) *conditional_rewrite_rule*) *list* ;
internal_names : ((*'SYMBOL internal_term*) *list*) ;
internal_formula : ((*'SYMBOL formula*) *list*)
}

val *make_automata* : *'SYMBOL list* → (*'SYMBOL* → *'TYPE*) → (*'TERM* → *'SYMBOL*) → (*'TERM* → (*'TERM list*)) → *'STATE list* → *'STATE list* → *'STATE* → (*'STATE*) *epsilon_rule list* → ((*'SYMBOL*, *'TERM*, *'STATE*) *conditional_rewrite_rule*) *list* → (*'SYMBOL*, *'TERM*, *'TYPE*, *'STATE*) *automata*

val *make_rule* : *'SYMBOL* → *'STATE list* → ((*int* × (*'TERM* → *'TERM* → *bool*) × *int*) *list*) → *'STATE* → (*'SYMBOL*, *'TERM*, *'STATE*) *conditional_rewrite_rule*

val *get_alphabet* : (*'SYMBOL*, *'TERM*, *'TYPE*, *'STATE*) *automata* → *'SYMBOL list*

val *get_states* : (*'SYMBOL*, *'TERM*, *'TYPE*, *'STATE*) *automata* → *'STATE list*

val *get_final_states* : (*'SYMBOL*, *'TERM*, *'TYPE*, *'STATE*) *automata* → *'STATE list*

val *get_rules* : (*'SYMBOL*, *'TERM*, *'TYPE*, *'STATE*) *automata* → (((*'SYMBOL*, *'TERM*, *'STATE*) *conditional_rewrite_rule*) *list*)

val *add_rule* : (*'SYMBOL*, *'TERM*, *'TYPE*, *'STATE*) *automata* → (*'SYMBOL*, *'TERM*, *'STATE*) *conditional_rewrite_rule* → (*'SYMBOL*, *'TERM*, *'TYPE*, *'STATE*) *automata*

val *add_epsilon_rule* : (*'SYMBOL*, *'TERM*, *'TYPE*, *'STATE*) *automata* → *'STATE epsilon_rule* → (*'SYMBOL*, *'TERM*, *'TYPE*, *'STATE*) *automata*

val *get_epsilon_rules* : (*'SYMBOL*, *'TERM*, *'TYPE*, *'STATE*) *automata* → ((*'STATE*) *epsilon_rule list*)

val *get_head_of_rule* : ((*'SYMBOL*, *'TERM*, *'STATE*) *conditional_rewrite_rule*) → *'SYMBOL*

val *get_constraints_of_rule* : ((*'SYMBOL*, *'TERM*, *'STATE*) *conditional_rewrite_rule*)
→ ((*int*× (*'TERM* → *'TERM* → *bool*) × *int*) *list*)

val *get_initiator_of_rule* : ((*'SYMBOL*, *'TERM*, *'STATE*) *conditional_rewrite_rule*)
→ *'STATE list*

val *get_final_state_of_rule* : ((*'SYMBOL*, *'TERM*, *'STATE*) *conditional_rewrite_rule*) → *'STATE*

Deterministe et sans epsilon-transitions termes clos

val *run_automata* : (*'SYMBOL*, *'TERM*, *'TYPE*, *'STATE*) *automata* → *'TERM* → *'STATE*

val *test_acceptance* : (*'SYMBOL*, *'TERM*, *'TYPE*, *'STATE*) *automata* → *'TERM* → (*'STATE* × *bool*)

Non Deterministe et avec epsilon-transitions termes clos val *nd_run_automata* : ('*SYMBOL*, '*TERM*, '*TYPE*, '*STATE*) *automata* → '*TERM* → '*STATE list*

val *nd_test_acceptance* : ('*SYMBOL*, '*TERM*, '*TYPE*, '*STATE*) *automata* → '*TERM* → ('*STATE list* × *bool*)

nd et epsilon terme avec variable

val *nd_test_var_acceptance* : (*var_id* × *sorte_de_base*) *list* → ('*SYMBOL*, '*TERM*, '*TYPE*, *Types.state*) *automata* → '*TERM* → (*Types.state list* × *bool*)

val *nd_test_complement* : ('*SYMBOL*, '*TERM*, '*TYPE*, '*STATE*) *automata* → '*TERM* → ('*STATE list* × *bool*)

val *nd_test_intersect* : ('*SYMBOL*, '*TERM*, '*TYPE*, '*STATE*) *automata* → ('*SYMBOL*, '*TERM*, '*TYPE*, '*STATE1*) *automata* → '*TERM* → *bool*

Functions that should be done in the future

```
val nd_test_final_state : ('a, term, 'b, state) automata -> term ->
state list * bool ;;

val nd_test_final_state : ('a, 'b, 'c, 'd) automata -> 'b -> 'd list *
bool;;

val remove_epsilons : ('SYMBOL, 'TERM, 'TYPE, 'STATE) automata ->
('SYMBOL, 'TERM, 'TYPE, 'STATE) automata ;;

 val determinize_automata : ('SYMBOL, 'TERM, 'TYPE, 'STATE) automata
 -> ('SYMBOL, 'TERM, 'TYPE, 'STATE) automata ;;
```

val *union_of_automata* : ('*SYMBOL*, '*TERM*, '*TYPE*, '*STATE*) *automata* → ('*SYMBOL*, '*TERM*, '*TYPE1*, '*STATE1*) *automata* → ('*SYMBOL*, '*TERM*, '*TYPE*, ('*STATE*,'*STATE1*) *u*) *automata*

Donne un etiquetage generique a l'automate sans oublier le premier etiquetage

val *internalize_automata* : ('*SYMBOL*, '*TERM*, '*TYPE*, '*STATE*) *automata* → ('*SYMBOL*, '*TERM*, '*TYPE*, '*STATE*) *automata*

Ne s'applique que si l'automate est internalize !!

val *create_internal_formula* : ('*SYMBOL*, '*TERM*, '*TYPE*, '*STATE*) *automata* → ('*SYMBOL*, '*TERM*, '*TYPE*, '*STATE*) *automata*

val *triv_leads_to* : ('*SYMBOL*, '*TERM*, '*TYPE*, '*STATE*) *automata* → '*STATE* → '*STATE* → *bool*

val *clean_automata* : ('*a*, '*b*, '*c*, '*d*) *automata* × '*d epsilon_rule list* → ('*a*, '*b*, '*c*, '*d*) *automata* × '*d epsilon_rule list*

17.16 Module *Specif*

open *Tree_automata*
open *Symbols*
open *Terms*
open *Equations*
open *Types*

val *make_signature_automata* :
  unit → (*symbol_id*, *term*, *sorte list*, *state*) *automata*

val *create_current_env* :
  term → (*Symbols.var_id* × *Symbols.sorte_de_base*) *list*

val *build_nf_automata* :
$\mathit{regle\ list} \rightarrow (\mathit{symbol\_id},\ \mathit{term},\ \mathit{sorte\ list},\ \mathit{state})\ \mathit{automata} \times \mathit{state\ epsilon\_rule\ list}$

val *print_delta_automata* :
$(\mathit{symbol\_id},\ \mathit{term},\ \mathit{sorte\ list},\ \mathit{state})\ \mathit{automata} \times \mathit{state\ epsilon\_rule\ list} \rightarrow \mathit{unit}$

val *print_clean_automata* :
$(\mathit{symbol\_id},\ \mathit{term},\ \mathit{sorte\ list},\ \mathit{state})\ \mathit{automata} \times \mathit{state\ epsilon\_rule\ list} \rightarrow \mathit{unit}$

val *print_state* : $\mathit{state} \rightarrow \mathit{unit}$

17.17 Module *Types*

type *sorte_de_base* = *int*

type *atomic_state* = *FS* of (*User_signatures.symbol_id* $\times$ (*atomic_state list*))
| *VS* of *sorte_de_base*

type *state* = *State* of (*atomic_state list*)

Chapter 18

Library toplevel

This library defines the toplevel language and its interpreter.

18.1 Module *Abstract_syntax*

type *abstract_expr* =
  | *Integer* of *Numbers.t*
  | *String* of *string*
  | *Bool* of *bool*
  | *Var* of *string*
  | *Fun* of *string* × *abstract_expr*
  | *Apply* of *abstract_expr* × *abstract_expr*
  | *If* of *abstract_expr* × *abstract_expr* × *abstract_expr*
  | *Tuple* of *abstract_expr list*
  | *Set* of *abstract_expr list*

type *abstract_command* =
  | *Def* of *string* × *abstract_expr*
  | *Deftuple* of *string list* × *abstract_expr*
  | *Deffun* of *string* × *string list* × *abstract_expr*
  | *Eval* of *abstract_expr*
  | *Directive* of *string* × *string*

val *print_expr* : *abstract_expr* → *unit*

val *print_named_expr* : *string* → *abstract_expr* → *unit*

18.2 Parser *Toplevel_parser*

Header

open *Abstract_syntax*

Token declarations

%token <*string*> *IDENT*
%token <*Numbers.t*> *INTEGER*
%token <*string*> *STRING*
%token *LET FUN ARROW*
%token *IF THEN ELSE*
%token *AND OR NOT TRUE FALSE*
%token *PLUS STAR MINUS*
%token *CONCAT*
%token *GE GT LE LT NEQ*
%token *LEFTPAR RIGHTPAR SEMICOLON COMMA LEFTBRACE RIGHTBRACE*
%token *EQUAL*
%token *DIRECTIVE*
%token *EOF*

%start *command*
%type <*Abstract_syntax.abstract_command*> *command*

%nonassoc *ARROW IF*
%left *OR*
%left *AND*
%left *NOT*
%left *GE GT LE LT NEQ EQUAL*
%left *PLUS MINUS*
%left *STAR*
%nonassoc *UMINUS*
%nonassoc *IDENT INTEGER STRING FUN LEFTPAR LEFTBRACE CONCAT TRUE FALSE*
%left *APPLY*

Grammar rules

command ::=
  EOF { raise End_of_file }
| *command_aux SEMICOLON* { $1 }

command_aux ::=
| *LET IDENT EQUAL expr* { *Def*($2,$4) }
| *LET LEFTPAR identlist RIGHTPAR EQUAL expr* { *Deftuple*($3,$6) }
| *LET FUN IDENT args EQUAL expr* { *Deffun*($3,$4,$6) }
| *expr* { *Eval*($1) }
| *DIRECTIVE IDENT* { *Directive*($2,"") }
| *DIRECTIVE IDENT STRING* { *Directive*($2,$3) }
| *DIRECTIVE IDENT IDENT* { *Directive*($2,$3) }
| *DIRECTIVE IDENT INTEGER* { *Directive*($2,*Numbers.to_string* $3) }

identlist ::=
| *IDENT* { [$1] }
| *IDENT COMMA identlist* { $1::$3 }

```
args ::=
   IDENT {  [$1] }
| IDENT args {  $1::$2 }


expr ::=
| IF expr THEN expr ELSE expr %prec IF
      {  If($2,$4,$6) }
| expr simple_expr %prec APPLY
      {  Apply($1,$2) }
| FUN IDENT ARROW expr
      {  Fun($2,$4) }
| expr PLUS expr
      {  Apply(Apply(Var("+"),$1),$3) }
| expr MINUS expr
      {  Apply(Apply(Var("-"),$1),$3) }
| MINUS expr %prec UMINUS
      {  Apply(Var("_minus"),$2) }
| expr STAR expr
      {  Apply(Apply(Var("*"),$1),$3) }
| expr EQUAL expr
      {  Apply(Apply(Var("="),$1),$3) }
| expr NEQ expr {  Apply(Apply(Var("<>"),$1),$3) }
| expr GE expr {  Apply(Apply(Var(">="),$1),$3) }
| expr GT expr {  Apply(Apply(Var(">"),$1),$3) }
| expr LE expr {  Apply(Apply(Var("<="),$1),$3) }
| expr LT expr {  Apply(Apply(Var("<"),$1),$3) }
| expr AND expr {  Apply(Apply(Var("and"),$1),$3) }
| expr OR expr {  Apply(Apply(Var("or"),$1),$3) }
| expr CONCAT expr {  Apply(Apply(Var("^"),$1),$3) }
| NOT expr {  Apply(Var("not"),$2) }
| simple_expr
         {  $1 }


simple_expr ::=
| IDENT
      {  Var($1) }
| INTEGER
      {  Integer($1) }
| TRUE
      {  Bool(true) }
| FALSE
      {  Bool(false) }
| STRING
      {  String($1) }
| LEFTPAR expr RIGHTPAR
      {  $2 }
| LEFTPAR expr COMMA commalist RIGHTPAR
         {  Tuple($2 :: $4) }
| LEFTBRACE RIGHTBRACE
               {  Set([]) }
| LEFTBRACE semicolonlist RIGHTBRACE
               {  Set($2) }
```

commalist ::=
| *expr*
{ [$1] }
| *expr COMMA commalist*
{ $1::$3 }

semicolonlist ::=
| *expr*
{ [$1] }
| *expr SEMICOLON*
{ [$1] }
| *expr SEMICOLON semicolonlist*
{ $1::$3 }

18.3 Module *Eval*

open *Abstract_syntax*

open *Values*

val *eval* : *env_type* → *abstract_expr* → *values*

18.4 Module *Typing*

open *Abstract_syntax*

exception raised in case of typing error, with an explicit message as argument.

exception *Type_error* of *string*

the type system in the toplevel

A type is either

- a base type : int, bool, string, etc ;
- a dependent type : (Σ, X) term, etc ;
- a type variable (without explicit name) ;
- an *arrow type* $t_1 \rightarrow t_2$;
- a *dependent arrow type* $(x : t_1)t_2$;

Base types are particular cases of dependent types where the list of dependent vars is empty.

An arrow type $t_1 \rightarrow t_2$ is a particular case of the dependent arrow type $(x : t_1)t_2$ when x does not occur in t_2, which is made explicit by putting x as "anonymous".

type *depend* = *Depend_var* | *Anonymous* | *Depend_id* of *string*

type *depend_name* = *depend ref*

type *types* =
| *Base_type* of *depend_name list* × *types list* × *string*
| *Dep_arrow* of *depend_name* × *types* × (*depend_name* → *types*)
| *Type_variable* of *types option ref*

and *type_scheme* =
Generalized of (*depend_name* → *type_scheme*)
| *Non_generalized* of *types*

and $typing\_env\_type \ = \ (string \times type\_scheme) \ list$

val $generalize \ : \ types \ \rightarrow \ type\_scheme$

val $print\_type \ : \ types \ \rightarrow \ unit$

val $unify\_types \ : \ types \ \rightarrow \ types \ \rightarrow \ bool$

val $type\_expr \ : \ typing\_env\_type \ \rightarrow \ typing\_env\_type \ \rightarrow \ abstract\_expr \ \rightarrow \ types$

val $unit\_type \ : \ types$

val $int\_type \ : \ types$

val $string\_type \ : \ types$

val $bool\_type \ : \ types$

val $set\_type \ : \ types \ \rightarrow \ types$

val $pair\_type \ : \ types \ \rightarrow \ types \ \rightarrow \ types$

val $dioph\_constraint\_type \ : \ types$

val $signature\_type \ : \ types$

val $word\_signature\_type \ : \ types$

val $theo\_type \ : \ depend\_name \ \rightarrow \ types$

val $variable\_set\_type \ : \ types$

val $word\_type \ : \ depend\_name \ \rightarrow \ types$
val $word\_precedence\_type \ : \ depend\_name \ \rightarrow \ types$
val $precedence\_type \ : \ depend\_name \ \rightarrow \ types$
val $status\_type \ : \ depend\_name \ \rightarrow \ types$
val $word\_ordering\_type \ : \ depend\_name \ \rightarrow \ types$
val $term\_ordering\_type \ : \ depend\_name \ \rightarrow \ types$
val $srs\_type \ : \ depend\_name \ \rightarrow \ types$
val $position\_type \ : \ types$
val $term\_type \ : \ depend\_name \ \rightarrow \ depend\_name \ \rightarrow \ types$
val $conjecture\_type \ : \ depend\_name \ \rightarrow \ depend\_name \ \rightarrow \ types$
val $equations\_type \ : \ depend\_name \ \rightarrow \ depend\_name \ \rightarrow \ types$
val $trs\_type \ : \ depend\_name \ \rightarrow \ depend\_name \ \rightarrow \ types$
val $htrs\_type \ : \ types$
val $termination\_constraint\_type \ : \ depend\_name \ \rightarrow \ depend\_name \ \rightarrow \ types$
val $order\_constraint\_type \ : \ depend\_name \ \rightarrow \ depend\_name \ \rightarrow \ types$

val $parameter\_set\_type \ : \ types$
val $pword\_signature\_type \ : \ types$
val $pword\_type \ : \ depend\_name \ \rightarrow \ types$
val $psrs\_type \ : \ depend\_name \ \rightarrow \ types$
val $psubst\_type \ : \ types$

val $arrow\_type \ : \ types \ \rightarrow \ types \ \rightarrow \ types$

val $bin\_arith\_type \ : \ type\_scheme$

val $bin\_comp\_type \ : \ type\_scheme$

val $bin\_bool\_type \ : \ type\_scheme$

18.5 Module *Predef*

open *Abstract_syntax*

open *Values*

val *initial_env* : *unit* → *env_type*

val *initial_env_type* : *unit* → *Typing.typing_env_type*

val *get_code* : *string* → (*int* × (*values list* → *values*))

Returns the list of the predefined category names.

val *get_predef_category_list* : *unit* → *string list*

Returns the list of the predefined command names in the given category.

val *get_predef_command_list* : *string* → *string list*

Returns the category and the help string associated to a command name.

val *get_help_of_command* : *string* → *string* × *string*

Returns the type of the function associated to a command name.

val *get_type_of_command* : *string* → *Typing.type_scheme*

18.6 Module *Values*

open *Abstract_syntax*

```
type named_sig =
{
  mutable sig_name : string ;
  sig_val : User_signatures.user_signature;
  sig_table : User_signatures.symbol_entry array;
}
```

```
type named_hsig =
{
  mutable hsig_name : string ;
  hsig_val : Hierarchical_signatures.hierarchical_signature
}
```

```
type named_word_sig =
{
  mutable word_sig_name : string ;
  word_sig_val : String_signatures.user_word_signature
}
```

```
type named_parameters =
{
  mutable parameters_name : string ;
  parameters_val : Parameterized_signatures.parameter_c ;
}
```

```
type named_pword_sig =
{
  mutable pword_sig_name : string ;
  pword_sig_val : User_parameterized_signatures.user_parameterized_signature
}
```

```
type named_psubst =
{
  mutable psubst_name : string ;
  psubst_val : string → Numbers.t
}

type named_vars =
{
  mutable vars_name : string ;
  vars_val : Variables.user_variables;
  vars_table : string array;
}

type values =
  | Unit_value
  | Int_value of Numbers.t
  | Bool_value of bool
  | String_value of string
  | Fun_value of string × env_type × abstract_expr
  | Predef of string × values list

  | Set_value of values list
  | Prod_value of values list

  | Sig_value of named_sig
  | Word_sig_value of named_word_sig

  | Parameters_value of named_parameters
  | Pword_sig_value of named_pword_sig
  | Pword_value of
      named_pword_sig × Parameterized_words.word
  | Psrs_value of
      named_pword_sig × Parameterized_rewriting.srs

  | Psubst_value of named_psubst

  | Theo_value of named_sig × User_signatures.symbol_id Theory.elem_theory list

  | Var_value of named_vars
  | Word_value of
      named_word_sig × String_signatures.symbol_id Words.word
  | Word_precedence_value of
      named_word_sig
      × String_signatures.symbol_id Orderings_generalities.ordering
  | Precedence_value of
      named_sig
      × User_signatures.symbol_id Orderings_generalities.ordering
  | Status_value of
      named_sig
      × User_signatures.symbol_id Rpo.status_function
  | Word_ordering_value of
      named_word_sig
      × String_signatures.symbol_id Words.word Orderings_generalities.ordering
  | Term_ordering_value of named_sig × User_signatures.symbol_id Term_orderings.t
  | SRS_value of
      named_word_sig ×
      String_signatures.symbol_id String_rewriting.srs ×
      (String_signatures.symbol_id String_rewriting.compiled_srs) Lazy.t
  | Position_value of Positions.position
```

```
  | Term_value of
      named_sig × named_vars ×
      User_signatures.symbol_id Gen_terms.term
  | Conjecture_value of
      named_sig × named_vars ×
      (User_signatures.symbol_id Gen_terms.term × User_signatures.symbol_id Gen_terms.term)
  | Equations_value of
      named_sig × named_vars ×
      (User_signatures.symbol_id Gen_terms.term × User_signatures.symbol_id Gen_terms.term) list
  | TRS_value of
      named_sig × named_vars ×
      (User_signatures.symbol_id Gen_terms.term × User_signatures.symbol_id Gen_terms.term) list ×
      (User_signatures.symbol_id Gen_terms.term → User_signatures.symbol_id Gen_terms.term)
  | HTRS_value of
      named_vars × Hierarchical_trs.htrs
  | Dioph_constraint of Abstract_constraint.formula
  | Linear_constraint of Linear_constraints.formula
  | Termination_constraint of
      named_sig × named_vars ×
      User_signatures.symbol_id Termination_constraints.conj_termination_constraints
  | Order_constraint of
      named_sig × named_vars ×
      User_signatures.symbol_id Order_constraints.formula

and env_data =
    {
      (* typ : types *)
      mutable value : values
    }

and env_type = (string × env_data) list

val print_value : values → unit

exception Eval_error of string
```

18.7 Module *Toplevel_lexer*

```
exception Lexical_error of string

val token : Lexing.lexbuf → Toplevel_parser.token
```

18.8 Module *Version*

```
val version : string

val date : string
```

18.9 Lexer *Toplevel_lexer*

```
{
  open Toplevel_parser

  exception Lexical_error of string

}
```

```
rule token = parse
     [' ''\r''\t''\n'] { token lexbuf }
   | "let" { LET }
   | "fun" { FUN }
   | "if" { IF }
   | "then" { THEN }
   | "else" { ELSE }
   | "and" { AND }
   | "or" { OR }
   | "not" { NOT }
   | "true" { TRUE }
   | "false" { FALSE }
   | "#" { DIRECTIVE }
   | ['A'-'Z''a'-'z']['A'-'Z''a'-'z''0'-'9''_''\'']*
                                   { IDENT(Lexing.lexeme lexbuf) }
   | ';' { SEMICOLON }
   | ',' { COMMA }
   | '+' { PLUS }
   | '-' { MINUS }
   | '*' { STAR }
   | '^' { CONCAT }
   | "=" { EQUAL }
   | "->" { ARROW }
   | ">" { GT }
   | ">=" { GE }
   | "<" { LT }
   | "<=" { LE }
   | "<>" { NEQ }
   | ['0'-'9']+ { INTEGER (Numbers.from_string (Lexing.lexeme lexbuf)) }
   | '(' { LEFTPAR }
   | ')' { RIGHTPAR }
   | '{' { LEFTBRACE }
   | '}' { RIGHTBRACE }
   | '"' { STRING(string lexbuf) }
   | "(*" { comment lexbuf }
   | _ { raise (Lexical_error (Lexing.lexeme lexbuf)) }
   | eof { EOF }

and string = parse
     '"' { "" }
   | '\\' _ { let s = Lexing.lexeme lexbuf in s^(string lexbuf) }
   | [^'\\''"']+ { let s = Lexing.lexeme lexbuf in s^(string lexbuf) }
   | eof { raise (Lexical_error "string not terminated") }

and comment = parse
     "*)" { token lexbuf }
   | [^'*']+ { comment lexbuf }
   | _ { comment lexbuf }
   | eof { raise (Lexical_error "comment not terminated") }
```

Index of Identifiers