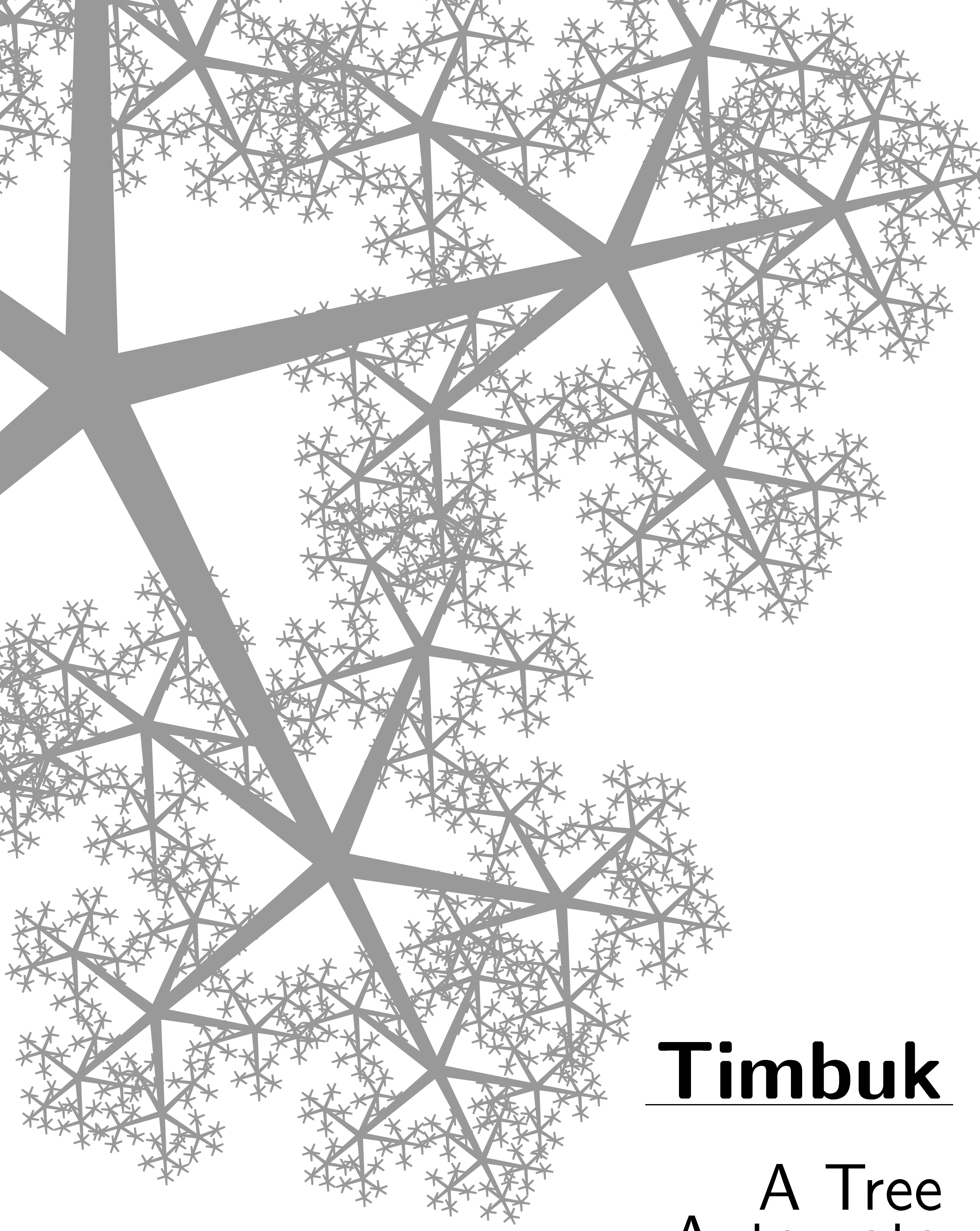

Timbuk

A Tree Automata Library

Thomas Genet & Valérie Viet Triem Tong
IRISA / Université de Rennes 1
`www.irisa.fr/lande/genet/timbuk`

1 What is Timbuk?

Timbuk is a Library of Objective Caml [5] functions designed to manipulate tree automata. The tree automata we use here are bottom-up non-deterministic finite tree automata (NFTA for short). See [1] for a survey on NFTA and on tree automata in general.

This Ocaml library offers basic functions on NFTA:

- boolean operations: intersection, union, inversion
- emptiness decision, inclusion decision
- cleaning, renaming
- determinisation
- normalization of transitions
- parsing, pretty printing
- reading and writing automata to disk
- and some more ...

This library also contains some basic functions on alphabets, terms and term rewriting systems, as well as specific functions implementing algorithms described in [3, 4]:

- approximation of the set of descendants and normal forms
- completion of a NFTA with regards to a Term Rewriting System (possibly non left-linear)
- matching in tree automata

Most of the functions of the library are implemented straightforwardly without objects and in a functional style. However, we tried to develop this library in a modular way such that optimization could be added afterward. Thus optimization's are welcome and can even be proposed to us for implementation.

License: Timbuk is freely available, under the terms of the GNU LIBRARY GENERAL PUBLIC LICENSE.

2 Documentation

Since this software is a modular library, we wanted to have a separated documentation for each module. That is why we chose to generate this documentation using ocamlweb [2]. In the following you will find one section for each important module and file: the tutorial files and their related specification files, and then the interface for modules automaton, term, term rewriting systems, ...

3 Using the library

This software may be used as (1) an Ocaml library i.e., functions can be called from some other Ocaml code, or (2) as a tree automata calculator through Ocaml toplevel for a teaching or an experimenting purpose.

1. a sample main file using the tree automata library is given in `main.ml`. It can be compiled as is by typing 'make' (it may be already up to date) and then by executing the 'main' Ocaml bytecode executable. This sample program simply reads a spec file name given as a parameter of the program on the command line and then starts a completion with:

   - the first term rewriting of the specification
   - the first tree automaton of the specification
   - the first approximation (if it exists) of the specification (or the approximation whose name is given with the option flag `--approx`

   and it permits to compute intersections with the completed automaton and all the other automata of the specification file with the `(i)` command of the completion mechanism. See the documentation of the file `tutorial_comp.mlml` in the following sections for more details about completion mechanism.

   Examples:

   ```
   main --approx Simple example_comp2.txt
   ```

   ```
   main example_comp2.txt
   ```

   ```
   main example_comp.txt
   ```

2. this library can also be used at toplevel of Ocaml as a tree automata simple calculator. The first thing to do is to compile the library by typing 'make' (it may be already up to date). Then, an example file is given in 'tutorial.mlml'. To see an example, simply type

   ```
   #use "tutorial.mlml";;
   ```

   at Ocaml toplevel (with the right path if necessary) to load the tutorial and then refer to the CALCULATOR MODE section of the documentation of the file `tutorial.mlml` (in the following sections). Then you can type (or copy and paste from the `tutorial.mlml file`) the given commands in order to compute intersection, determinise etc...

   NOTE: those example files only shows some of the functionalities offered by the library... please have a look to the interface file `automaton.mli` for other functions.

4 Bug report and information

Since we are not Ocaml wizards, do not hesitate to report any bug or awkwardness in the code... there should be some (a lot ?).

Please report comments and bugs to `Thomas.Genet@irisa.fr`.

See `http://www.irisa.fr/lande/genet/timbuk/` for information about Timbuk and related papers.

Acknowledgments

Many thanks to Thomas Colcombet for his numerous relevant advices on Ocaml programming.

References

[1] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. `http://l3ux02.univ-lille3.fr/tata/`, 1997.

[2] J.-C. Filiâtre and C. Marché. ocamlweb: a literate programming tool for Objective Caml. Institut National de Recherche en Informatique et Automatique, 2000. `http://www.lri.fr/~filliatr/ocamlweb/`.

[3] T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In *Proceedings 9th Conference on Rewriting Techniques and Applications, Tsukuba (Japan)*, volume 1379 of *Lecture Notes in Computer Science*, pages 151–165. Springer-Verlag, 1998.

[4] T. Genet and F. Klay. Rewriting for Cryptographic Protocol Verification. In *Proceedings 17th International Conference on Automated Deduction, Pittsburgh (Pen., USA)*, volume 1831 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2000.

[5] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system release 3.00 – Documentation and user's manual. Institut National de Recherche en Informatique et Automatique, 2000. `http://caml.inria.fr/ocaml/htmlman/`.

Module Tutorial.mlml

1. T.A. Basics: parse, print, intersect, determinise, save...

This is a short tutorial for achieving simple computations on tree automata. This file is to be loaded into the ocaml toplevel thanks to the ocaml command :

#use `"tutorial.mlml"`

This tutorial is to be used with the specification file *example.txt* of the distribution. This specification file is also in the next chapter.

The instructions of the tutorial can be found in the next section which is entitled 'CALCULATOR MODE' (to be found below). However, we first detail a bit the first lines of the file *tutorial.mlml* which are evaluated when the *#use* `"tutorial.mlml"` command is executed:

(1) the specification contained in the file *example.txt* is parsed and assigned to a variable *s*.

let *s* = *file_parse* `"example.txt"`

(2) automata and term rewriting systems are extracted from the specification: the first two tree automata named *Simple* and *SimpleAgain*, the term rewriting system labelled by *R1*, another tree automaton *AA* and finally the alphabet of the specification

let *a1* = *get_automaton* `"Simple"` *s*

let *a2* = *get_automaton* `"SimpleAgain"` *s*

let *r* = *get_trs* `"R1"` *s*

let *aa* = *get_automaton* `"AA"` *s*

let *alphabet* = *get_alphabet* *s*

2. CALCULATOR MODE. Once you have evaluated the command

#use `"tutorial.mlml"`

all preceding variable definitions have already been done. Now we give some expressions to evaluate at the toplevel in order to achieve interactively some simple computations on the tree automata we have in the variables *a1*, *a2* and *aa*.

3. To compute an intersection between the tree automata *a1* and *a2* and simplify the result (i.e. clean and rename), evaluate the following caml expression:

let *ainter* = *simplify* (*inter a1 a2*);;

4. To print the result, evaluate:

print_string (*to_string ainter*);;

5. or more briefly:

print ainter;;

6. To determinise *a2*, evaluate:

let $adet = \ determinise\ a2;;$

7. Then, print the result:

$print\ adet;;$

8. To compute the automaton recognising the terms in normal form with regard to term rewriting *R1* (stored in variable r), evaluate:

let $a\_nf = \ simplify(nf\_automaton\ alphabet\ r);;$

9. To prove that all terms recognized by *aa* are reducible, we can compute the intersection between *aa* and *a_nf* and verify that it is empty, evaluate:

let $ainter = \ simplify\ (inter\ aa\ a\_nf);;$

10. Print the result:

$print\ ainter;;$

11. Save the automaton *a_nf* under the name *toto* in a file named *test.txt*

save_automaton a_nf `"toto" "test.txt"`;;

Specification example.txt

```
(* Here is a typical specification (to be used with tutorial.mlml) where we find

   1) a unique (and compulsory) alphabet declaration for all the
      symbols on which terms are built. This declaration begins by the
      keyword "Ops" and is followed by every declared symbol with its
      arity (integer). A symbol should start with a letter and should
      not contain characters: '*', '+', ':', '"', '.', ';', '[', ']', '(',
      ')', nor strings "Ops", "Vars", "TRS", "->", "Automaton", "States",
      "Final", "Transitions", "Description", "--", "Prior".

   2) An optional variable declaration, beginning by the keyword "Vars"

   3) Any number of Tree Automata or TRSs declaration.

        - A TRS declaration begins by the keyword "TRS" followed by
          the name of the TRS (an identifier),
        - A Tree automaton declaration begin by the keyword
          "Automaton" followed by the name of the Tree automaton (an
          identifier)

TREE AUTOMATA should be normalized (have a normalized set of transitions) i.e.,
each transition should be of the form: f(q1, ..., qn) -> q where f is a symbol
of arity n and q1, ..., qn and q are states.

-------
REMARKS on the state declarations.
-------
      State labels can be either (1) new identifiers or (2) terms
      constructed using ** State operators ** transforming any term or
      configuration into a state label.

      (1) See in the tree automaton named "Simple" how to define
      state labels identifiers. For numbered state labels like "abc1,
      abc2, ..., abc123" it is possible to use the shorthand
      "abc[1--123]" to define them all at once.

      (2) See in the automaton named "bazounga", some particular
      states identifier followed by an arity. Those states are in fact
      state operators, like q:1. The type of the state operator q, can be
      viewed as:
```

```
        q: T(F U Q) -> Q, i.e.

      this operator takes a configuration (terms built on symbols of the
      alphabet F and the states of Q) and gives a state of Q.

      For the user, the state constructor q:1 permits to
      construct states in the usual subscript notation, for example thanks
      to q of arity 1, we can construct the state q(nil). Thus it is
      possible to have the transition nil -> q(nil) for example or
      cons(q1, q2) -> q(cons(q1, q2)) or even things more complicated...
      like combin(f(rev(q1)),q2)

      Finally, in automaton named "AA" there is an example of a
      description of states (this is also optional). This description
      permits to comment on the "intuition" behind a state i.e.,
      describe the set of objects it is supposed to recognize... for
      user information purpose only !
*)


Ops cons:2 a:0 b:0 c:0 d:0 nil:0 app:2 rev:1


Vars x y z


TRS R1
        app(nil, x) -> x
        app(cons(x, y), z) -> cons(x, app(y, z))

TRS R2
        rev(nil) -> nil
        rev(cons(x, y)) -> app(rev(y), cons(x, nil))


(* an automaton recognizing flat lists of 'a' *)

Automaton Simple

States q[1--2]
Final States q2
Transitions cons(q1, q2) -> q2
            nil -> q2
```

```
            a -> q1



(* an automaton recognizing non empty flat lists of 'a' followed by some 'b' *)

Automaton SimpleAgain

States qa qb qla qlb qfinal
Final States qfinal
Transitions
            cons(qa, qla) -> qfinal
            cons(qb, qlb) -> qfinal
            cons(qa, qlb) -> qfinal
            a -> qa
            b -> qb
            cons(qa, qla) -> qla
            cons(qa, qlb) -> qla
            nil -> qla
            cons(qb, qlb) -> qlb
            nil -> qlb



Automaton bazounga
States q[1--10] bob combin:2 f:1 q:1
Final States q1 q(nil) f(f(q2))
Transitions
        nil -> q(nil)
        cons(q1,q2) -> q(cons(q1,q2))
        cons(f(q1),f(f(q2))) -> combin(f(rev(q1)),q2)



Automaton AA

States q[0--3] q:1
Description q0 : "Non empty Flat lists of a"
            q1 : "Non empty Flat lists of b"
            q2 : "Flat lists of a (possibly empty)"
            q3 : "Flat lists of b (possibly empty)"
            q(a) : "Symbol a"
            q(b) : "Symbol b"
            q(app(q0,q1)) : "the app function applied to lists of a and lists of b"
```

```
Final States q(app(q0,q1))
Transitions
        a -> q(a)
        b -> q(b)
        cons(q(a), q2) -> q0
        cons(q(b), q3) -> q1
        nil -> q2
        cons(q(a), q2) -> q2
        nil -> q3
        cons(q(b), q3) -> q3
        app(q0,q1) -> q(app(q0,q1))
```

Module Tutorial_comp.mlml

12. Completion of T.A.

This is a tutorial for achieving completions on tree automata, in order to compute approximations of *sets of descendants*. This file is to be loaded into the ocaml toplevel thanks to the ocaml command:

$\#use$ `"tutorial_comp.mlml"`.

This tutorial is to be used with the specification file *example_comp.txt*.

The instructions of the tutorial can be found in the next section which is entitled 'CALCULATOR MODE' (to be found below). However, we first detail (a bit more than in the previous tutorial) the first lines of the file *tutorial_comp.mlml* which are evaluated when the previous $\#use$ `"tutorial_comp.mlml"` command is executed. In particular, whereas in previous tutorial some modules were opened in order to simplify notations, we chose here to use the unambiguous dotted notations for function calls (for example *Myautomaton.inter* instead of *inter*), in order to show to which module those function really belong.

(1) modules are constructed and assigned to local names

module $MyF = Alphabet(Symbol)$

module $MyX = Variable\_set(Variable)$

module $Myterm = Term(Symbol)(MyF)(Variable)(MyX)$

module $Mytrs = RewriteSystem(MyF)(MyX)(Myterm)$

module $MyQ = State\_set(Symbol)(MyF)(Myterm)(State\_content)$

module $Myautomaton = TreeAutomata(Symbol)(MyF)(Variable)(Myterm)$
$(State\_content)(Mytrs)(MyQ)$

module $MyGamma = Gamma(Symbol)(MyF)(MyX)(Myterm)(Mytrs)(MyQ)(Myautomaton)$

module $Myspec = Specification(MyF)(MyX)(Mytrs)(Myautomaton)(MyGamma)$

module $MyComp = Completion(MyF)(Myterm)(Mytrs)(MyQ)(Myautomaton)$
$(Myspec)(MyGamma)$

(2) the specification contained in the file *example_compl.txt* is parsed and assigned to variable s

let $s = Myspec.file\_parse$ `"example_comp.txt"`

(3) automata and term rewriting systems are extracted from the specification: we get the alphabet, the tree automata named $A0$ and B as well as the term rewriting system labelled R, and assign them to ocaml variables.

let $alpha = Myspec.get\_alphabet\ s$

let $a0 =$ *Myspec.get_automaton* "A0" s

let $b =$ *Myspec.get_automaton* "B" s

let $rapp =$ *Myspec.get_trs* "R" s

(3) we construct of an empty gamma approximation function.

let $gc =$ *MyGamma.empty*

13. CALCULATOR MODE. Now we give some expressions to type and evaluate at the toplevel in order to compute interactively an approximation automaton on a simple example.

14. We compute interactively the approximation automaton obtained by completion of automaton *a0* (corresponding to automaton *A0* of the file *example_compl.txt*) by TRS *rapp* with approximation initialized with *gc*, and a list of only one verification automaton: automaton b (the B automaton of the file *example_compl.txt*)

if completion is performed up to the message *Automaton is complete*, the result automaton recognise a super-set of $rapp^*(\mathcal{L}(a0))$ i.e., all terms reachable by applying TRS rapp on terms in the language recognised by *a0*. By typing:

let *aut_comp* = *MyComp.complet rapp a0 gc* ["Automaton with some 'b' before 'a'",b];;

you obtain the following menu on output:

```
STARTING COMPLETION...
-----------------------




Do you want to:

(c)omplete one step  (use Ctrl-C to interrupt if necessary)
complete (a)ll steps (use Ctrl-C to interrupt if necessary)
(m)erge some states
(s)ee current automaton
(i)ntersection with verif automata
(v)erify linearity condition on current automaton
(w)rite current automaton to disk
(u)ndo last step
(q)uit completion

(c/a/m/s/i/v/w/u/q)?
```

Type c in order to achieve one completion step.

```
Adding transition:

 cons(qa,app(qla,qlb)) -> q(app(qla,qlb))

Do you want to give by hand some rules to normalize the transition? (y/n)?
```

This means that the system have found an unresolved critical pair, and propose you to normalise it (since this transition is not normalised). You have two ways to normalise it: by giving explicitely new states for the normalisation, or by letting the system choosing new states. For example, for this first transition, we can choose to give by hand the new states: answer y to this first question... you obtain the output:

```
Use key word New States followed by the names of the new states ended
by a dot '.'(optional), then give a sequence of transitions ended by
a dot '.' Add a star '*' before transitions you want to add to the
prior set. The prior transitions should be normalized!!
```

Now we may choose to add a new state called $qapp$, for instance, and use it to normalise the transition in the following way: normalise $cons(qa, app(qla, qlb)) \to q(app(qla, qlb))$ into two transitions: $app(qla, qlb) \to qapp$ and transition $cons(qa, qapp) \to q(app(qla, qlb))$. This can be done by typing something like:

```
New States qapp.
app(qla, qlb) -> qapp.
```

This is enough to finish this first completion step since for the normalisation of other critical pairs user interaction is not needed.

```
Normalisation simplifies the transition into: cons(qa,qapp) -> q(app(qla,qlb))


Adding transition:

 cons(qa,qapp) -> q(app(qla,qlb))

 ... already normalised!


Adding transition:

 app(qla,qlb) -> qapp

 ... already normalised!


Adding transition:

 qlb -> q(app(qla,qlb))

 ... automatically normalized.
```

Then you come back to the menu. Let's type c in order to perform another step of completion

```
Adding transition:

 cons(qa,app(qla,qlb)) -> qapp

Do you want to give by hand some rules to normalize the transition? (y/n)?
```

This time, we want the system to automatically provide new states in order to normalise the transition, we thus answer n and we are asked if we want to use automatic normalisation to finish... answer: y (otherwise, the process starts again with hand rules to give, since the transition is still not normalised).

Answering yes, permits to end this new completion step and we find again the menu... Perform two other automatic completion steps, by typing c, n, y and then again c, n, y. Then, we find again the menu. Type s to see the current automaton, you should obtain something close to:

```
States qnew2:0 qnew1:0 qnew0:0 qapp:0 qa:0 qb:0 qla:0 qlb:0 q:1

Description
qla: "Flat lists of a"
qlb: "Flat lists of b"
qa: "Symbol a"
qb: "Symbol b"
q(app(qla,qlb)): "the app function applied to lists of a and lists of b"


Final States q(app(qla,qlb))

Transitions
a -> qa
b -> qb
cons(qb,qlb) -> qnew1
cons(qa,qnew2) -> qnew1
cons(qb,qlb) -> qnew0
cons(qa,qnew1) -> qnew0
cons(qb,qlb) -> qapp
cons(qa,qnew0) -> qapp
cons(qb,qlb) -> q(app(qla,qlb))
cons(qa,qapp) -> q(app(qla,qlb))
cons(qa,qla) -> qla
cons(qb,qlb) -> qlb
nil -> qnew1
nil -> qnew0
nil -> qapp
nil -> q(app(qla,qlb))
nil -> qla
nil -> qlb
app(qla,qlb) -> qnew2
app(qla,qlb) -> qnew1
app(qla,qlb) -> qnew0
app(qla,qlb) -> qapp
app(qla,qlb) -> q(app(qla,qlb))
```

From the last five lines, we can deduce that the completion process may go on forever since we may infinitely add new states qnew0, qnew1, ... all recognising app(qla, qlb)... In order to approximate, we can simply say to the system to identify qnew0, qnew1 and qnew2 (and thus fold an infinite branch) by the following command: *m* for merging states. Then type a rewrite rule on states for identifying qnew0, qnew1 and qnew2, for example:

```
Give a sequence of rewrite rules on state labels ended by a dot: '.'

qnew2 -> qnew1
qnew1 -> qnew0.
```

Type again *s* to see the current automaton where states have been merged. You can notice qnew2 and qnew1 have been identified to qnew0.

Now type *c* to continue completion and you obtain:

```
Automaton is complete!!
-----------------------
```

meaning that no other completion step can be performed and this automaton recognise a superset of reachable terms.

Since the TRS rapp used for this completion is the definition of the *app* function (concatenation of lists), and since the initial set of terms recognised by the tree automaton a0 is the set of terms app(x, y) where x is any flat list of *a* and y is any flat list of *b* we may check the simple property that *app* function applied on terms a0 may never produce any list where there is a *b* before an *a*. The automaton *B* in the *example_compl.txt* file recognise such 'ill-formed' lists. Now we can check the property by typing *i* in order to compute intersection between the completed automaton and the *B* automaton of the spec file. We obtain:

```
Intersection with Automaton with some 'b' before 'a' gives
(the empty automaton):

States

Final States

Transitions
```

meaning that the property " *app* function applied on terms a0 may never produce any list where there is a *b* before an *a*" is verified.

In order to see another tool for achieving approximation, go back to the menu and undo all completion done so far: type *u* (undo) until you get the message : Nothing to undo! Now we are back to the initial status of the completion Type *s* to verify that the current automaton is indeed the initial automaton (i.e. automaton "A" of the *example_compl.txt* file). Now type *c* to re-achieve the first completion step. Type *y* for giving rules by hand, define a new state qapp and a transition identical to the one we have given before, but type a '$*$' before, i.e. you should enter something like:

```
Use key word 'New States' followed by the names of the new states
ended by a dot '.' (optional) then give a sequence of transitions
ended by a dot '.' Add a star '*' before transitions you want to add
to the prior set. The prior transitions should be normalized!!

New States qapp.
*app(qla, qlb) -> qapp.
```

You just entered a *prior rule*... this is a particular transition of the automaton used to normalise every new critical pair before giving it to the user. By typing s you can see how the automaton has been modified, in particular transition $app(qla,\ qlb) \rightarrow qapp$ has been added to the *Prior* section of the transition table. Note that you can also directly add some prior transition into the automaton specification file. For example, you can look at automaton C in file *example_comp.txt*.

Then, you can end completion in purely automatic mode by typing: c, for a new completion step that is automatic and then c again which terminate the completion and the automaton is complete. Then type q to quit the completion function.

15. Now if you initially invoked the completion function with the command line:
let *aut_comp* = *MyComp.complet rapp a0 gc* [`"Automaton with some 'b' before 'a'"`,*b*];;
like it was proposed, then the variable *aut_comp* now contains the completed automaton.

16. Let's construct the automaton recognising terms in normal form w.r.t. TRS *rapp* by typing:
let *a_nf* = *Myautomaton.nf_automaton alpha rapp*;;

17. Now let's construct the automaton recognising *normal forms* reachable by application of TRS *rapp* on terms recognised by *a0*, i.e. intersection between reachable terms (*aut_comp*) and normal forms (*a_nf*).
let *aa* = *Myautomaton.inter a_nf aut_comp*;;

18. simplification of *aa*
let *a_simpl* = *Myautomaton.simplify aa*;;

19. printing the result... with the command
print_string (*Myautomaton.to_string a_simpl*);;
you should obtain:

```
States q7:0 q6:0 q5:0 q4:0 q3:0 q2:0 q1:0 q0:0

Final States q6 q7

Transitions
a -> q1
b -> q4
nil -> q0
nil -> q7
nil -> q3
cons(q4,q3) -> q2
cons(q1,q0) -> q2
cons(q1,q2) -> q2
cons(q4,q3) -> q6
cons(q1,q0) -> q6
cons(q4,q5) -> q2
cons(q1,q2) -> q6
cons(q4,q3) -> q5
cons(q4,q5) -> q5
cons(q4,q5) -> q6
```

this automaton recognize flat lists of a and b where a are all before b.

20. To write to disk this last automaton under the name *Reachable_normal_forms* into a file named *test2.txt*, simply type:
Myspec.save_automaton a_simpl `"Reachable_normal_forms" "test2.txt"`;;

Specification example_comp.txt

```
(* This is a specification file to be used with tutorial_comp.mlml *)

Ops cons:2 a:0 b:0 c:0 d:0 nil:0 app:2 rev:1


Vars x y z


TRS R
        app(cons(x, y), z) -> cons(x, app(y, z))
        app(nil, x) -> x




Automaton A0

States qa qb qla qlb q:1
Description qla : "Flat lists of a"
            qlb : "Flat lists of b"
            qa : "Symbol a"
            qb : "Symbol b"
            q(app(qla,qlb)) : "the app function applied to lists of a and lists of b"

Final States q(app(qla,qlb))
Transitions
        a -> qa
        b -> qb
        cons(qa, qla) -> qla
        nil -> qla
        cons(qb, qlb) -> qlb
        nil -> qlb
        app(qla,qlb) -> q(app(qla,qlb))




Automaton B
States qa qb qla qlba ql
Description qla : "Flat list with at least one a"
            ql : "Any flat list of a and b"
            qa : "Symbol a"
```

```
            qb : "Symbol b"
            qlba : "Flat list with at least one b followed by an a"

Final States qlba
Transitions a -> qa
            b -> qb
            cons(qb, qla) -> qlba
            cons(qa, qlba) -> qlba
            cons(qa, ql) -> qla
            cons(qb, qla) -> qla
            nil -> ql
            cons(qa, ql) -> ql
            cons(qb, ql) -> ql



Automaton C
States q[1--5]
Final States q5
Prior
        a -> q1
        b -> q2
        app(q3,q4) -> q5

Transitions
        cons(q1, q3) -> q3
        nil -> q3
        cons(q2, q4) -> q4
        nil -> q4
```

Module Tutorial_comp2.mlml

21. Completion of T.A.: more techniques!

This last tutorial should help some users needing more automation during completion computation. This file is to be loaded into the ocaml toplevel thanks to the ocaml command:
#use `"tutorial_comp2.mlml"`.

This tutorial is to be used with the specification file *example_comp2.txt*.

22. For computing approximation in more complex cases, you will often need some other functionnalities of the system.

- **Normalisation rules:** if your approximation can be statically described, you can give normalisation rules to describe it directly in the specification file
- **Strategy:** you can modify the completion strategy.
- **Interruption:** if the completion steps are getting longer, you may need to interrupt the computation without loosing the completed automaton

To illustrate those notions, let's parse another specification

let *s* = *Myspec.file_parse* `"example_comp2.txt"`

Then, extract the TRS, the automaton and the approximation:

let *rapp* = *Myspec.get_trs* `"R"` *s*

let *a0* = *Myspec.get_automaton* `"A0"` *s*

let *a* = *Myspec.get_approximation* `"Simple"` *s*

23. Normalisation rules. The approximation *Simple* is given into an additionnal (optional) field in the specification, an *Approximation* field:

```
Approximation Simple
States qnil
Rules
[cons(x, y) -> z] -> [y -> z]
[x -> y] -> [nil -> qnil]
```

In this approximation, called Simple, we declare a state *qnil* (which may or may not occur in other automata) and define an approximation thanks to normalisation rules, used to normalise new transition found during completion. A normalisation rule is of the form

$[l \to r] \to [\mathcal{R}]$ where $l \to r$ is called *the pattern* and $\mathcal{R}$ is a TRS. Given a new transition $c \to q$ to normalise, a normalisation rule $[l \to r] \to [\mathcal{R}]$ will apply on $c \to q$ if there exists two substitutions σ_l and σ_r such that $l\sigma_l = c, \quad r\sigma_r = q$, and $\sigma_l \circ \sigma_r$ is also a substitution (i.e. σ_l and σ_r do not map a same variable to distinct values). If the norm rule apply then the transition $c \to q$ is normalised thanks to $\mathcal{R}\sigma_l \circ \sigma_r$ i.e., c is rewritten by the TRS $\mathcal{R}$ where the variables of $\mathcal{R}$ have been substituted by $\sigma_l \circ \sigma_r$.

If we focus on the example of the *Simple* approximation given above, the first normalisation rule will match any transition $c \to q$ where c is of the form $cons(t_1, t_2)$ and will normalise c with the rule $t_2 \to q$. In the second normalisation rule, the pattern $x \to y$ matches every transition and simply rewrite any occurrence of the symbol *nil* into a state q_{nil} (similar to a prior rule).

Note that one may also define normalisation rules where some of the variables in the left-hand side of rules of $\mathcal{R}$ may be not instanciated by the pattern matching. However, right-hand side of rules of $\mathcal{R}$ have to be instanciated and necessarily by a state. For example, the following normalisation rules are valid:

```
[app(x, y) -> z] -> [u -> qnil]
[x -> y] -> [cons(u, v) -> y   app(u, v) -> qapp]
```

Assume that the transition to normalise is $app(app(cons(q_1, q_2), q_3), cons(q_4, q_5)) \to q$. The first normalisation rule would normalise the transition into the set of transitions:

$$\{cons(q_1, q_2) \to q_{nil}, cons(q_4, q_5) \to q_{nil}, app(q_{nil}, q_3) \to q_{nil}, app(q_{nil}, q_{nil}) \to q\}$$

And the second normalisation rule would normalise it into the set of transitions:

$$\{cons(q_1, q_2) \to q, cons(q_4, q_5) \to q, app(q, q_3) \to q_{app}, app(q_{app}, q) \to q\}$$

However, since rules are supposed to be ordered with regards to their priority, the first rule will apply first and normalise the transition. Thus, in that case, the second rule not will not apply since the transition would have already been fully normalised by the first rule.

Let's see how the *Simple* approximation leads to an automatic approximation. Simply type:

let *aut_comp* = *MyComp.complet rapp a0 a* [];;

where a is the *Simple* approximation given in the specification. Once you are proposed the menu, type a for doing all possible completion steps. Automaton is completed automatically thanks to the normalisation rules.

24. Strategy: so far we have only used the default completion strategy. Each approximation is assigned a specific strategy. To see which strategy is assigned to the *Simple* approximation, return to the top level of Ocaml and type:

MyGamma.get_strategy a;;

and you obtain a list of strings: "*prior*", "*norm_rules*", "*manual_conf*" and "*auto_conf*" corresponding to four different methods for normalising new transitions. In fact, in the *Gamma* module there exists two other methods "*manual*" and "*auto*" which are only variants of "*manual_conf*" and "*auto_conf*" respectively.

- "*prior*" corresponds to normalisation using prior rules
- "*norm_rules*" corresponds to normalisation using norm rules
- "*manual*" corresponds to the interactive normalisation of transitions
- "*manual_conf*" is similar but ask for a confirmation before starting the normalisation
- "*auto*" permits to normalise transition with new states
- "*auto_conf*" is similar but also ask for a confirmation before proceeding.

The order of the different methods in the strategy list corresponds to their priority. Hence in the default strategy, we first normalise the new transitions thanks to the prior rules, then with the norm rules, then by user interaction and finally with automatic normalisation with new states. However, note that each time that one of the method succeeds in normalising a transition or a subterm of the left-hand side, we continue the normalisation process restarting from the beginning of the strategy, i.e. prior rules, norm rules etc.

Now, let's change the strategy. Let's create a new approximation similar to a but with a strategy where "*auto*" is the unique applied method:

let *a2* = *MyGamma.set_strategy* ["auto"] *a*;;

25. Interruption. Of course, this strategy not terminating in general. We will see what happens on the last example with this specific strategy and the different ways to interrupt the completion. Start a new completion with this new strategy:

let *aut_comp* = *MyComp.complet rapp a0 a2* [];;

When looking at the menu, type a in order to achieve all possible completion step. The completion diverges fastly. Now to interrupt it type $Ctrl - C$ and you are proposed the following menu:

```
Do you want to:

(c)ontinue the completion
(s)ee the current automaton (on which critical pairs are computed)
(f)inish the current completion step and then go back to the menu
(q)uit the completion step NOW (thus loose current computation)
   and go back to the menu

(c/f/s/q)?
```

Now you can continue the completion by typing c, see the current automaton by typing s or interrupt the completion either by typing q and then the current completion step is given up (and lost) but the completion is halted immediately, or by typing f and then the current completion step is finished before interrupting the current completion step (and nothing is lost). Since this completion cannot terminate with this strategy, you have to type f or q.

Then, when proposed the main menu, you can either quit the completion or try to make it terminate by merging some states. For example, look at the automaton, find the new state with the highest natural label k such that there is a transition $app(qla,\ qlb) \ \rightarrow \ qnew(k)$ and merge the states $qnew(k)$ and $qnew(k-1)$ by giving the merging rule $qnew(k) \ \rightarrow \ qnew(k-1)$. Back the the main menu, finish the completion by typing a... the automaton is completed.

Specification example_comp2.txt

```
(* This is a specification file to be used with tutorial_comp2.mlml *)

Ops cons:2 a:0 b:0 c:0 d:0 nil:0 app:2 rev:1


Vars x y z u


TRS R
        app(cons(x, y), z) -> cons(x, app(y, z))
        app(nil, x) -> x


Automaton A0

States qa qb qla qlb q:1

Final States q(app(qla,qlb))
Transitions
        a -> qa
        b -> qb
        cons(qa, qla) -> qla
        nil -> qla
        cons(qb, qlb) -> qlb
        nil -> qlb
        app(qla,qlb) -> q(app(qla,qlb))


Approximation Empty
        States
Rules

Approximation Simple
        States qnil
        Rules
                [cons(x, y) -> z] -> [y -> z]
                [x -> y] -> [nil -> qnil]
```

Interface for module Automaton

26. This is the interface for bottom-up tree automata. A bottom-up tree automata is usually defined as a tuple: $\langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ where $\mathcal{F}$ is an alphabet of symbols, $\mathcal{Q}$ is a set of states, $\mathcal{Q}_f$ a set of final states and Δ is a set of transitions (also called a transition table). Here, the tree automata module is defined w.r.t.

- a symbol type
- an alphabet type (the type of $\mathcal{F}$) whose symbols are of symbol type
- a variable type. It is used for defining variables occuring in matching on tree automata
- a configuration type i.e. left-hand side of transitions
- a state content type which can be anything assigned to states: formulas, or simply text
- a transition type which is a term rewriting system and defined the type of Δ we use
- a state set type defining the type of $\mathcal{Q}$ and $\mathcal{Q}_f$ we use. In fact, in practice its major role is to assign state contents to states.

```
module TreeAutomata
  (Symbol_type : PRINTABLE_TYPE)
  (Alphabet_type : ALPHABET_TYPE with type symbol = Symbol_type.t)
  (Variable_type : PRINTABLE_TYPE)
  (Configuration_type : TERM_TYPE with type symbol = Symbol_type.t
                                  and type variable = Variable_type.t
                                  and type alphabet = Alphabet_type.t)
  (State_content : STATE_CONTENT_TYPE)
  (Transition_type : TRS_TYPE with type alphabet = Alphabet_type.t
                              and type term = Configuration_type.t)
  (State_set_type : STATE_SET_TYPE with type state = Configuration_type.t
                                   and type state_content = State_content.t
                                   and type alphabet = Alphabet_type.t
                                   and type symbol = Symbol_type.t) :
sig
  exception Not_a_state of string
  exception Not_in_folder
  exception Multiply_defined_symbol of string
  exception Linearity_problem of string
  exception Normalisation_problem of string
```

type $symbol =$ $Symbol\_type.t$
type $alphabet =$ $Alphabet\_type.t$
type $term =$ $Configuration\_type.t$
type $rule =$ $Transition\_type.rule$
type $substitution =$ $Configuration\_type.substitution$
type $state =$ $term$
type $state\_set =$ $State\_set\_type.t$

type $transition\_table$ $=$ $Transition\_type.t$
type $tree\_automata$

type t $=$ $tree\_automata$
type ('a, 'b) $folder$

27. Constructor of tree automata. The main difference with usual definitions of tree automata is that we here use an extended definition of states. States are terms (gasp!). States are terms constructed on a specific alphabet which is what we call *state operators*. This make no difference with usual definition of states and tree automata if you consider only state operators of arity 0 (i.e. constant state symbol) then if q, state123, q0, q1, etc... are state operators of arity 0, then q, state123, q0, q1, etc... are states. However, if you define a state operator q of arity 1, and qa of arity 0, then qa, q(qa), q(q(qa)), ... are states. In fact, you can even define more complicated states, since state operators can transform any term (constructed on the alphabet and on state operators) into a state. For example, assume that your alphabet $\mathcal{F}$ contains operators: f of arity 2 and b of arity 0, and your state operators contain at least q of arity 1 and qa of arity 0, then a, q(qa), q(q(qa)), q(b), q(f(b,b)), q(f(qa, b)), q(q(f(q(qa), b))), etc... are states.

In most cases, state operators of arity greater than 0 are not needed. Nevertheless, note that to define a simple tree automaton with state set $\mathcal{Q} = \{q0, q2, q3\}$ and final states $\mathcal{Q}_f = \{q2\}$, you will need to define state operators $q0, q2, q3$ of arity 0, and to give to the *make_automaton* function the state operators (of alphabet type), the state set corresponding to $\mathcal{Q}$ and then the set of final states representing $\mathcal{Q}_f$. However, it is much easier to use the parsing function of tree automata or, even simpler, the parsing function of the specification module, please have a look to the file *tutorial.mlml* for more details.

val $make\_automaton$:
  $main\_alphabet :$ $alphabet$ $\rightarrow$
    $state\_operators :$ $alphabet$ $\rightarrow$
      $state\_set$ $\rightarrow$
        $final\_states :$ $state\_set$ $\rightarrow$
          $transitions :$ $transition\_table$ $\rightarrow$
            $prior\_transitions :$ $transition\_table$ $\rightarrow$ t

28. accessors of automata

val $get\_alphabet$: t $\rightarrow$ $alphabet$
val get_state_ops : t $\rightarrow$ $alphabet$
val get_states : t $\rightarrow$ $state\_set$

val $get\_final\_states \ : \ t \ \rightarrow \ state\_set$
val $get\_transitions \ : \ t \ \rightarrow \ transition\_table$
val $get\_prior \ : \ t \ \rightarrow \ transition\_table$

29. Prettyprint of tree automata. The first thing to be able to do with an automaton is to display it.

val $print : \ t \ \rightarrow \ unit$

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

30. Now, we find the **boolean operations** on tree automata.
First of all, intersection of two tree automata. This function produces a tree automaton with structured states (states that are in fact products of states) and structured state sets (symbolic form of state set products). In order to obtain a full tree automaton with constructed state sets, apply accessibility cleaning (defined in the following) on it.

val $inter \ : \ t \ \rightarrow \ t \ \rightarrow \ t$

union of two tree automata (by renaming and union of transition tables, state set, final state sets etc...).

val $union \ : \ t \ \rightarrow \ t \ \rightarrow \ t$

The complement operation.

val $inverse \ : \ t \ \rightarrow \ t$

The automaton recognizing the subtraction of langages: subtract L(a2) to L(a1)

val $subtract \ : \ t \ \rightarrow \ t \ \rightarrow \ t$

Decision of inclusion between two langages: is L(a1) included in L(a2)?

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

Decision of the emptyness of a *language* recognized by a tree automaton.

val $is\_language\_empty \ : \ t \ \rightarrow \ bool$

31. Make a run of a tree automaton: verify if a term t rewrites into state q with regards to transitions of automaton a. This is not the usual definition of a run, but the usual one can easily be obtained from this one.

val $run \ : \ t : term \ \rightarrow \ q : state \ \rightarrow \ a : t \ \rightarrow \ bool$

32. The determinisation function: given a tree automaton it gives an equivalent deterministic one.

val $determinise \ : \ t \ \rightarrow \ t$

33. Completion of tree automaton... in a non-deterministic way i.e., the result is a non-deterministic tree automaton. If a deterministic one is needed, it needs to be determinised afterwards.

val $make\_complete \ : \ t \ \rightarrow \ t$

34. Construction of an automaton recognizing *reducible terms.* Starting from an alphabet a and a TRS r built on a, this function constructs the tree automaton recognizing terms reducible by r.

val $make\_red\_automaton$:
$a : alphabet \ \rightarrow \ r : Transition\_type.t \ \rightarrow \ t$

35. Construction of an automaton recognizing *irreducible terms.* Starting from an alphabet a and a TRS r built on a, this function constructs the tree automaton recognizing terms irreducible by r. The result is a deterministic complete tree automaton, it may be cleant afterwards with *simplify* if necessary.

This implements a standard algorithm that is usually not efficient at all. For a better efficiency, use the next function called nf_opt.

val $nf\_automaton$:
$a : alphabet \ \rightarrow \ r : transition\_table \ \rightarrow \ t$

This one is usually more efficient than the previous one in practice. However the result is also slightly different: the produced tree automaton is not necessarily deterministic nor complete!

val nf_opt :
$a : alphabet \ \rightarrow \ r : transition\_table \ \rightarrow \ t$

36. Cleaning of tree automata
Accessibility cleaning of tree automaton: retrieves all states that do not recognize any term.

val $accessibility\_cleaning \ : \ t \ \rightarrow \ t$

Utility cleaning: retrieves all dead states. For utility cleaning on an automaton with structured state sets (obtained for example by application of an intersection operation use *accessibility_cleaning* before this one.

val $utility\_cleaning \ : \ t \ \rightarrow \ t$

Accessibility cleaning followed by utility cleaning

val $clean \ : \ t \ \rightarrow \ t$

Simplification of tree automaton: a renumbering of the result of cleaning (accessibility + utility) of the tree automaton. Useful for deciding if the langage recognized by an automaton a is empty. If it is then $is\_emtpy(simplify\ a)$ is true

val $simplify \ : \ t \ \rightarrow \ t$

37. State Renumbering
This function rewrites state labels in a tree automaton a thanks to a term rewriting system r on states. Be careful! for state sets including states q1, q2 for example and if you use structured states labels like q(f(q1,q2)), if q1 and q2 are to be renamed into q3 and q4 respectively, then so is q(f(q1,q2)) which is renamed into q(f(q3,q4))!!

val *rewrite_state_labels* :
$a : t \rightarrow r : transition\_table \rightarrow t$

Automatic renumbering of a tree automaton. To apply this function on an automaton with structured state sets (obtained by intersection for example), use *accessibility_cleaning* before this one.

val *automatic_renum* : $t \rightarrow t$

38. For saving an automaton to disk, see function *save_automaton* in the module specification.

Low level functions

39. Emptyness of an *automaton*, i.e. emptyness of its transition table. For checking of the language apply simplify function before) i.e., a is a tree automaton recognizing an empty langage if and only if *is_empty*(*simplify a*) is true.

val *is_empty* : $t \rightarrow bool$

40. Modification of final state set.

val *modify_final* : $t \rightarrow state\_set \rightarrow t$

41. Construction of a state from a symbol with arity 0. Recall that a state is a term!

val *make_state* : $symbol \rightarrow state$

42. Construction of a state config from a state. A state config is a configuration (i.e. a lhs or a rhs of a transition) that is a state. For example: in $q1 \rightarrow q2$, $q1$ is a state configuration.

val *make_state_config* : $state \rightarrow term$

43. Construction of a new transition

val *new_trans* : $symbol \rightarrow state\ list \rightarrow state \rightarrow rule$

44. Is a configuration a state configuration? A state config is a configuration (i.e. a lhs or a rhs of a transition) that is a state. For example: in $q1 \rightarrow q2$, $q1$ is a state configuration.

val *is_state_config* : $term \rightarrow bool$

45. State label of a state in a state configuration.

val *state_label* : $term \rightarrow state$

val *lhs* : $rule \rightarrow term$
val *rhs* : $rule \rightarrow term$

46. Top symbol of a transition

val *top_symbol* :

$rule \ \rightarrow \ symbol$

47. Is a transition normalized? i.e. of the form $f(q1,\ldots,qn) \rightarrow q'$ where $q1,\ldots,qn$ are states.

val $is\_normalized \ : \ rule \ \rightarrow \ bool$

48. Construction of the list of states of the left hand side of a transition.

val $list\_state \ : \ rule \ \rightarrow \ state \ list$

49. Construction of the state set formed by the states of all the transition of the transition table.

val $states\_of\_transitions \ :$
$transition\_table \ \rightarrow \ state\_set$

50. Normalization of epsilon transitions of the form $q1 \rightarrow q2$ with regards to a given transition table *delta*.

val $normalize\_epsilon \ :$
$q1 : state \ \rightarrow$
$q2 : state \ \rightarrow$
$delta : transition\_table \ \rightarrow \ transition\_table$

51. Normalization of a transitions table *ltrans* with new states whose labels are *label^j* where j starts from i. It returns a triple with the new normalized transition table and the new state operator alphabet as well as the integer n+1 where n is the number of the last assigned new state.

val $normalize \ :$
$ltrans : transition\_table \ \rightarrow$
$delta : transition\_table \ \rightarrow$
$label : string \ \rightarrow \ i : int \ \rightarrow \ transition\_table \ \times \ int \times \ alphabet$

52. Matching of a term (ground or with variables) on a tree automaton configuration with regard to a transition list (here given as a folder of transitions sorted by top symbol and right-hand side (state).

val $matching \ :$
$the\_term : \ term \ \rightarrow$
$the\_config : \ term \ \rightarrow$
$(symbol, \ (state, \ rule \ list) \ folder) \ folder \ \rightarrow$
$substitution \ list$

53. Construct the disjointness constraint. This is used to check that there is no non-linear lhs of a rule (say f(x,x)) and no lhs of a transition (say f(q1,q2)) such that the language recognized by q1 and q2 are not disjoint. The non-linear lhs are given in a list of terms l, the transitions are given as a folder f of transitions sorted by top symbol and right-hand side (state), and the result is a list of list of states whose disjointness has to be checked.

val *disjointness_constraint* :
  l : *term list* →
    f : (*symbol*, (*state*, *rule list*) *folder*) *folder* →
      state list list

54. Is a term *t1* rewritten into a state *q* (special term) by transitions contained in the folder *f*.

val *is_recognized_into* :
  t1 : *term* →
    q : *state* →
      f : (*symbol*, (*state*, *rule list*) *folder*) *folder* → *bool*

55. similar to the *is_recognized_into* but in the particular case where the transition is an epsilon transition *q1*→*q2*, this consists in verifying that all the transitions going to *q1* are already going to *q2*. Transitions are given into a folder *f*.

val *is_covered* :
  q1 : *Configuration_type.t* →
    q2 : *Configuration_type.t* →
      f : (*symbol*, (*state*, *rule list*) *folder*) *folder* → *bool*

56. Parsing of a tree automaton with regards to an alphabet. For syntax, have a look to the *example.txt* file. See also the *file_parse* function of the module specification *specification.mli*.

val *parse* : *alphabet* → *Genlex.token Stream.t* → *t*

end

Interface for module Specification

57. This is the interface for specifications. What we call a specification is a collection of term rewriting systems and bottom-up tree automata all defined on a common alphabet. Consequenlty, this module is defined thanks to an alphabet type, a variable set type (used to define rewrite rules), a term rewriting system type and an automata type. Term rewriting system and tree automata are all assigned with a name (a string) in the specification. The simplest way to construct a specification is to write it in a file and parse it thanks to the *file_parse* function. For a **sample specification file**, please look at the file *example.txt* contained is the distribution.

module *Specification*
  (*Alphabet_type* : *ALPHABET_TYPE*)
  (*Variable_set_type* : *VARIABLE_SET_TYPE*)
  (*TRS_type* : *TRS_TYPE* with type *alphabet* = *Alphabet_type.t*
    and type *variable_set* = *Variable_set_type.t*)
  (*Automata_type* : *AUTOMATA_TYPE* with type *alphabet* = *Alphabet_type.t*)
  (*Gamma_type* : *GAMMA_TYPE* with type *variable_set* = *TRS_type.variable_set*
    and type *alphabet* = *TRS_type.alphabet*) :
sig
  type *alphabet* = *Alphabet_type.t*
  type *variable_set* = *Variable_set_type.t*
  type *trs* = *TRS_type.t*
  type *automaton* = *Automata_type.t*
  type *gamma_content* = *Gamma_type.gamma_content*
  type *spec*

  type *t* = *spec*
  exception *Name_used_twice* of *string*
  exception *No_TRS_of_that_name* of *string*
  exception *No_automaton_of_that_name* of *string*

58. Parsing of a specification in a file of name *file_name*.

  val *file_parse* : *file_name* : *string* → *spec*

59. Get the alphabet of a specification s.

  val *get_alphabet* : s : *spec* → *alphabet*

60. Get the set of variables of a specification s.

  val *get_variables* : s : *spec* → *variable_set*

61. Get the term rewriting system named *name* in the specification s.

val $get\_trs\ :\ name:\ string \rightarrow\ s:spec\ \rightarrow\ trs$

62. Get the list of named term rewriting systems of a specification s.

val $get\_list\_trs\ :\ s:spec\ \rightarrow\ (string \times trs)\ list$

63. Get the automaton named *name* in the specification s.

val $get\_automaton\ :\ name:\ string \rightarrow\ s:spec\ \rightarrow\ automaton$

64. Get the list of named automata of a specification s.

val $get\_list\_automata\ :\ s:spec\ \rightarrow\ (string \times automaton)\ list$

65. Get the approximation named *name* in the specification s.

val $get\_approximation:\ name:\ string \rightarrow\ s:spec\ \rightarrow\ gamma\_content$

66. Get the list of named approximation of a specification s.

val $get\_list\_approximation\ :\ s:spec\ \rightarrow\ (string \times gamma\_content)\ list$

67. Pretty print of a specification s.

val $to\_string\ :\ s:spec\ \rightarrow\ string$

68. Writing a specification s to a file named *file_name*.

val $write\_to\_disk\ :\ s:spec\ \rightarrow\ file\_name:\ string \rightarrow\ unit$

69. Saving an automaton a under the name *aut_name* in a specification file named *file_name*.

val $save\_automaton\ :\ a:automaton\ \rightarrow\ aut\_name:\ string \rightarrow\ file\_name\ :string \rightarrow\ unit$
end

Interface for module Term

70. This is the interface for terms of $\mathcal{T}(\mathcal{F}, \mathcal{X})$ constructed on an alphabet $\mathcal{F}$ and a set of variables $\mathcal{X}$

module *Term*
  (*Symbol_type* : *PRINTABLE_TYPE*)
  (*Alphabet_type* : *ALPHABET_TYPE* with type *symbol* = *Symbol_type.t*)
  (*Variable_type* : *PRINTABLE_TYPE*)
  (*Variable_set_type* : *VARIABLE_SET_TYPE* with type *variable* = *Variable_type.t*) :
sig

71. A term is either a variable, a constant, a functionnal symbol with a list of subterms, or a special term. A special term is build on a union of the alphabet and a special alphabet.

For example, let $\mathcal{F} = \{f : 2, g : 1, a : 0\}$ an alphabet and $\mathcal{F}' = \{prod : 2, q : 0, h : 2\}$ a special alphabet.

Then $f(g(a), g(prod(q, h(g(q), q))))$ is a term where $prod(q, h(g(q), q))$ is a special subterm. The *Special*() constructor is used in the implementation to separate the special subterms in a term.

type *symbol* = *Symbol_type.t*
type *variable* = *Variable_type.t*
type *alphabet* = *Alphabet_type.t*
type *variable_set* = *Variable_set_type.t*
type *term* = (*Symbol_type.t*, *Variable_type.t*) *term_const*
type *t* = *term*

type *substitution* = (*Variable_type.t* × *term*) *list*
exception *Terms_do_not_match* of *string* × *string*
exception *Badly_formed_term* of *string*
exception *Undefined_symbol* of *string*
exception *Bad_operation_on_special_term* of *string*
exception *Bad_operation_on_var* of *string*

val *equal* : *t* → *t* → *bool*

72. Depth of a term, where depth of Special terms, variables and constant is 0

val *depth* : *t* → *int*

73. Pretty printing of terms into strings

val *to_string* : *t* → *string*

val *top_symbol* : *t* → *Symbol_type.t*

74. the direct subterms of a term

val $list\_arguments\ :\ t\ \rightarrow\ t\ list$

75. is a term a variable?

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

76. is a term a constant?

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

77. is a term special: its top constructor is a Special constructor

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

78. get the term t from Special(t)

val $get\_special\ :\ t\ \rightarrow\ t$

79. mapping function f1 on every symbol of term t1 and f2 on every constant, variable or special term

val $map\ :\ f1 : (Symbol\_type.t\ \rightarrow\ Symbol\_type.t)\ \rightarrow\ f2 : (t\ \rightarrow\ t)\ \rightarrow\ t1 : t\ \rightarrow\ t$

80. get the list of the leaves of a term

val $list\_leaves\ :\ t\ \rightarrow\ t\ list$

81. get the list of variables of a term

val $list\_variables\ :\ t\ \rightarrow\ Variable\_type.t\ list$

82. is a term ground? i.e. with no variables. Note that a special term can be ground

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

83. is a term linear? i.e. there is only one occurence of each variable in the term

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

84. get the list of all terms t such that $Special(t)$ is a subterm of $t1$

val $list\_special\ :\ t1 : t\ \rightarrow\ t\ list$

85. Check the consistency of a term with regards to an alphabet. i.e. checks that for every subterm $f(s1, ..., sn)$ of $t1$, f has an arity n in the alphabet a. This function returns the term itself if it is correct, raise a *Badly_formed_term* exception if arity of the symbol does not correspond to its number of arguments, and raise a *Undefined_symbol* exception if the term contains a symbol that does not belong to the alphabet.

val $check\ :\ a : Alphabet\_type.t\ \rightarrow\ t1 : t\ \rightarrow\ t$

86. apply a substitution to a term (at every variable position in it) val *apply* : *substitution* → *t* → *t*

87. returns the list of terms (*s t1*) (substitution *s* applied to *t1*) for every substitution *s* of l

val *apply_several* : *l* : (*substitution list*) → *t1* : *t* → *t list*

88. returns the list of terms (*s t1*) (substitution *s* applied to *t1*) for every substitution *s* of l and every term t1 of lt

val *apply_substs_on_terms* : *l* : (*substitution list*) → *t1* : *t list* → *t list*

89. Parsing of terms w.r.t. an alphabet *a* and a set of variable *varset*

val *parse* :
  a : *Alphabet_type.t* →
    var_set : *Variable_set_type.t* → *Genlex.token Stream.t* → *t*

90. Parsing of ground terms w.r.t. an alphabet *a*

val *parse_ground* :
  a : *Alphabet_type.t* →
    Genlex.token Stream.t → *t*

91. Verify the coherence of a substitution: a variable must not be mapped to distinct terms. Otherwise a *Term_do_not_match* exception is raised

val *coherent* : *substitution* → *substitution*

92. matching of *term1* on *term2*, such that *term2* is ground or at least with variables disjoint from those of *term1*.

val *matching* : *term1* : *t* → *term2* : *t* → *substitution*

93. similar functions for special terms
Check the consistency of a term with regards to an alphabet *a* and a special alphabet *spa* i.e. checks that for every subterm $f(s1, ..., sn)$ of *t1*, *f* has an arity n in the alphabet if $f(s1, ..., sn)$ is a term or in *spa* if $f(s1, ..., sn)$ is below a Special constructor.This function returns the term itself if it is correct, raise a *Badly_formed_term* exception if arity of the symbol does not correspond to its number of arguments, and raise a *Undefined_symbol* exception if the term contains a symbol that does not belong to the alphabets.

val *check_special* : *a* : *Alphabet_type.t* → *spa* : *Alphabet_type.t* → *t1* : *t* → *t*

replacement in special terms: for every pair (*t1*, *t2*) of *l*, replace every *Special*(*t1*) by *Special*(*t2*) at every Special position in *t3*

val *replace_special* : *l* : ((*t* × *t*) *list*) → *t3* : *t* → *t*

Parsing of a term with special subterms w.r.t. alphabet *a* and special alphabet *spa*.

val *parse_special* :

a : $Alphabet\_type.t$ $\rightarrow$
    spa : $Alphabet\_type.t$ $\rightarrow$
        $Variable\_set\_type.t$ $\rightarrow$ $Genlex.token\ Stream.t$ $\rightarrow$ t

Parsing of ground special terms w.r.t. alphabet a and special alphabet spa.

val $parse\_ground\_special$:
  a : $Alphabet\_type.t$ $\rightarrow$
    spa : $Alphabet\_type.t$ $\rightarrow$
        $Genlex.token\ Stream.t$ $\rightarrow$ t

end

Interface for module Rewrite

94. This is the interface for rewrite rules and rewrite systems constructed on an alphabet $\mathcal{F}$, a set of variables $\mathcal{X}$ and a set of terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$

module $RewriteSystem$
  $(Alphabet\_type : \; ALPHABET\_TYPE)$
  $(Variable\_set\_type : \; VARIABLE\_SET\_TYPE)$
  $(Term\_type : \; TERM\_TYPE$ with type $variable\_set = \; Variable\_set\_type.t$
    and type $alphabet = \; Alphabet\_type.t) :$
sig
  type $alphabet = \; Alphabet\_type.t$
  type $variable\_set = \; Variable\_set\_type.t$
  type $term \; = \; Term\_type.t$
  type $ruleSystem$
  type $t \; = \; ruleSystem$
  type $rule$

  exception $Variable\_rhs\_not\_included\_in\_lhs$ of $string$
  exception $Does\_not\_rewrite\_on\_top$
  exception $Badly\_formed\_rule$ of $string$

95. the empty trs and other constructors

  val $empty \; : \; t$
  val $new\_rule \; : \; term \; \rightarrow \; term \; \rightarrow \; rule$
  val $is\_empty \; : \; t \; \rightarrow \; bool$
  val $mem \; : \; rule \; \rightarrow \; t \; \rightarrow \; bool$

96. adding a rule in a trs, and union of two trs

  val $add \; : \; rule \; \rightarrow \; t \; \rightarrow \; t$
  val $union \; : \; t \; \rightarrow \; t \; \rightarrow \; t$

97. if the rule is not in the trs we can catenate without testing membership

  val $add\_fast \; : \; rule \; \rightarrow \; t \; \rightarrow \; t$

98. if trs are known to be disjoint we can catenate without testing membership for union

  val $union\_fast \; : \; t \; \rightarrow \; t \; \rightarrow \; t$

99. first rule of a ruleSystem and remainder of the system

  val $first \; : \; t \; \rightarrow \; rule$
  val $remainder \; : \; t \; \rightarrow \; t$

100. right-hand side and left-hand side of a rule

val rhs : $rule$ $\rightarrow$ $term$
val lhs : $rule$ $\rightarrow$ $term$

101. equality on rules

val $rule\_equal$: $rule$ $\rightarrow$ $rule$ $\rightarrow$ $bool$

102. is a rule left or right or left and right linear ?

val is_left_linear : $rule$ $\rightarrow$ $bool$
val is_right_linear : $rule$ $\rightarrow$ $bool$
val is_linear : $rule$ $\rightarrow$ $bool$

103. list of non linear lhs of a ruleSystem

val non_linear_lhs : t $\rightarrow$ $term\ list$

104. intersection of two trs

val $inter$: t $\rightarrow$ t $\rightarrow$ t

105. moving from list to ruleSystem and conversely

val to_list : t $\rightarrow$ $rule\ list$
val $list\_to\_trs$: $rule\ list \rightarrow$ t

106. prettyprint

val $rule\_to\_string$: $rule$ $\rightarrow$ $string$
val to_string : t $\rightarrow$ $string$

107. Checking one rule with regards to an alphabet: checks construction of lhs and rhs as well as inclusion of var(rhs) in var(lhs)

val $check\_rule$: $rule$ $\rightarrow$ $alphabet$ $\rightarrow$ $rule$

108. Checking a trs with regards to an alphabet: checks construction of lhs and rhs as well as inclusion of var(rhs) in var(lhs)

val $check$: t $\rightarrow$ $alphabet$ $\rightarrow$ t

109. parsing of a rule, given an alphabet a variable set $varset$

val $parse\_rule$:
  $a : alphabet$ $\rightarrow$
    $varset : variable\_set$ $\rightarrow$ $Genlex.token\ Stream.t$ $\rightarrow$ $rule$

110. parsing of a trs given an alphabet a variable set $varset$

val $parse$:
    $a : alphabet$ $\rightarrow$

$varset : variable\_set \ \rightarrow \ Genlex.token \ Stream.t \ \rightarrow \ t$

111. rewrite once on top position of term *t1* with any rule of trs r

val $rewrite\_top\_once \ : \ r : t \ \rightarrow \ t1 : Term\_type.t \ \rightarrow \ Term\_type.t$

112. leftmost innermost normalisation of the term *t1* thanks to a trs r

val $left\_inner\_norm \ : \ r : t \ \rightarrow \ t1 : Term\_type.t \ \rightarrow \ Term\_type.t$

113. bottom up normalisation of term *t1* thanks to trs r. Useful when the trs is a transition table of an automaton

val $bottom\_up\_norm \ : \ r : t \ \rightarrow \ t1 : Term\_type.term \ \rightarrow \ Term\_type.t$

114. similar functions but for rules and trs built on special terms ...

val $check\_special\_rule$:
$rule \ \rightarrow \ alphabet \ \rightarrow \ alphabet \ \rightarrow \ rule$
val $check\_special$:
$t \ \rightarrow \ alphabet \ \rightarrow \ alphabet \ \rightarrow \ t$
val $parse\_special\_rule$:
$alphabet \ \rightarrow$
$alphabet \ \rightarrow$
$variable\_set \ \rightarrow \ Genlex.token \ Stream.t \ \rightarrow \ rule$
val $parse\_special$:
$alphabet \ \rightarrow$
$alphabet \ \rightarrow$
$variable\_set \ \rightarrow \ Genlex.token \ Stream.t \ \rightarrow \ t$

val $parse\_ground\_special$:
$alphabet \ \rightarrow$
$alphabet \ \rightarrow \ Genlex.token \ Stream.t \ \rightarrow \ t$
val $parse\_ground\_special\_rule$:
$alphabet \ \rightarrow \ alphabet \ \rightarrow \ Genlex.token \ Stream.t \ \rightarrow \ rule$

val $left\_inner\_norm\_special$:
$t \ \rightarrow \ Term\_type.t \ \rightarrow \ Term\_type.t$

val $left\_inner\_norm\_special\_system$:
$t \ \rightarrow \ t \ \rightarrow \ t$

end

Interface for module Alphabet

115. This is the interface for alphabets which are sets of symbols associated with their arity i.e. their number of arguments

module $Alphabet$:
  functor ($Symbol\_type$: $PRINTABLE\_TYPE$) $\rightarrow$
sig
  type $symbol$ $=$ $Symbol\_type.t$
  type t

  exception $Symbol\_not\_in\_alphabet$ of $string$
  exception $Multiply\_defined\_symbol$ of $string$

116. One alphabet constructor

  val $new\_alphabet$: t

117. Parsing of alphabets (another constructor)

  val $parse$: $Genlex.token\ Stream.t \rightarrow t$

118. Testing the occurrence of a symbol in an alphabet

  val $occur$: $symbol \rightarrow t \rightarrow bool$

119. Adding a symbol with its arity in an alphabet

  val add_symbol : $symbol \rightarrow int \rightarrow t \rightarrow t$

120. Getting the arity of a symbol in an alphabet. This function raises the exception $Symbol\_not\_in\_alphabet(s)$ where s is the string associated with the symbol if it is not in the alphabet

  val get_arity : $symbol \rightarrow t \rightarrow int$

  val to_list : $t \rightarrow (symbol \times int)\ list$

121. Testing disjointness of two alphabets

  val $disjoint$: $t \rightarrow t \rightarrow bool$

122. Construct the union of two disjoint alphabets

  val $union\_fast$: $t \rightarrow t \rightarrow t$

123. Construct the union of two alphabets, possibly non-disjoint

  val $union$: $t \rightarrow t \rightarrow t$

124. Pretty print

val to_string : $t \rightarrow string$

end

Interface for module State_set

125. This is the interface for state sets. See the automaton module for a detailed description of the representation of states. State sets are sets of states associated with a state content which can be of various form: formulas, text, automaton (why not?)

module $State\_set$
  $(Symbol\_type :\ PRINTABLE\_TYPE)$
  $(Alphabet\_type :\ ALPHABET\_TYPE$ with type $symbol = Symbol\_type.t)$
  $(State\_type :\ TERM\_TYPE$ with type $symbol =\ Symbol\_type.t$
    and type $alphabet =\ Alphabet\_type.t)$
  $(State\_content :\ STATE\_CONTENT\_TYPE) :$
sig
  type $alphabet =\ Alphabet\_type.t$
  type $symbol =\ Symbol\_type.t$
  type $state\_content =\ State\_content.t$
  type $state =\ State\_type.t$
  type t

  exception $State\_not\_in\_state\_set$ of $string$
  exception Not_a_state of $string$
  exception $Structured\_state\_sets$ of $string$

126. The empty state set

  val $empty\ :\ t$

127. Add a state with no state content to a state set

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

128. Add a state with its content

  val $add\_verb\ :\ state\ \rightarrow\ state\_content\ \rightarrow\ t\ \rightarrow\ t$

129. Transform a list of state into a state set

  val $list\_to\_set\ :\ state\ list \rightarrow\ t$

130. Add all states of a list to a state set

  val $add\_all\ :\ state\ list \rightarrow\ t\ \rightarrow\ t$

131. Adds a list of states to a set *s1*, using their description coming from another set *s2*

  val $add\_all\_verb\ :\ state\ list \rightarrow\ s1 : t\ \rightarrow\ s2 : t\ \rightarrow\ t$

132. Is a state set empty? and is a state member of a state set?

val $is\_empty : t \rightarrow bool$
val $mem : state \rightarrow t \rightarrow bool$

133. The first element of a state set and the remainder

val $first : t \rightarrow state$
val $remainder : t \rightarrow t$

134. pretty print

val $to\_string : t \rightarrow string$

135. pretty print in verbose mode, where the content is also printed in front of its corresponding state

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

136. get the state content associated to a state in a state set

val $state\_description : state \rightarrow t \rightarrow state\_content$

137. The default binary symbol used for representing product of states

val $default\_prod\_symbol : symbol$

138. construction of a product state from two states

val $state\_product : state \rightarrow state \rightarrow state$

139. construction of the cartesian product of two state sets (in a symbolic way i.e. the cartesian product is not computed

val $symbolic\_product : t \rightarrow t \rightarrow t$

140. boolean operations on state sets

val $inter : t \rightarrow t \rightarrow t$
val $union : t \rightarrow t \rightarrow t$
val $minus : t \rightarrow t \rightarrow t$
val $union\_disjoint : t \rightarrow t \rightarrow t$

141. are all states from the list member of the state set

val $all\_mem : state\ list \rightarrow t \rightarrow bool$

142. produce and add to a state operator alphabet *s1* all symbols labeled by *str*^`"k"` where k takes the values from i to j

val $produce$:
$i : int \rightarrow j : int \rightarrow str : string \rightarrow s1 : Alphabet\_type.t \rightarrow Alphabet\_type.t$

143. Parsing of symbols of state set

val $parse\_ops$: $Genlex.token\ Stream.t \rightarrow Alphabet\_type.t$

144. Parsing of a state set

val $parse$:
  $alphabet \rightarrow$
    $alphabet \rightarrow Genlex.token\ Stream.t \rightarrow t$

145. Parsing of a state set with associated state contents

val $parse\_verb$:
  $alphabet \rightarrow$
    $alphabet \rightarrow Genlex.token\ Stream.t \rightarrow t$
end

Interface for module Variable_set

146. This is the interface for variable sets

```
module Variable_set :
  functor (Variable_type : PRINTABLE_TYPE) →
sig
  type variable = Variable_type.t
  type t
  val empty : t
  val is_empty : t → bool
  val mem : variable → t → bool
  val to_string : t → string
  val parse : Genlex.token Stream.t → t
end
```

Interface for module Gamma

147. This is the interface for the gamma approximation module. The role of the gamma function is to realize an approximate normalisation of new transition in order to make completion terminate as often as possible. This function is defined thanks to a database called *gamma_content* which can be enriched during completion. The role of this database is to store rules of normalisation, events encountered during the completion, ... or anything necessary for making good heuristical choices for normalisation of new critical pairs. The gamma content as well as the gamma function itself can be defined by the user. However we propose one simple gamma function here which is very simple and only take into account a set of prior transitions (see *tutorial_compl.mlml* for more details on prior transitions) in the *gamma_content* database. This set of prior transitions is being enriched by the user during completion and used for normalising transitions before giving them to an interactive normalisation process.

```
module Gamma
  (Symbol_type : PRINTABLE_TYPE)
  (Alphabet_type : ALPHABET_TYPE with type symbol = Symbol_type.t)
  (Variable_set_type : VARIABLE_SET_TYPE)
  (Term : TERM_TYPE with type symbol = Symbol_type.t
                    and type alphabet = Alphabet_type.t
                    and type variable_set = Variable_set_type.t)
  (TRS_type : TRS_TYPE with type alphabet = Alphabet_type.t
                       and type term = Term.t
                       and type variable_set = Variable_set_type.t)
  (State_set_type : STATE_SET_TYPE with type state = Term.t
                                   and type alphabet = Alphabet_type.t
                                   and type symbol = Symbol_type.t)
  (Automata_type : AUTOMATA_TYPE with type symbol = Symbol_type.t
                                 and type alphabet = Alphabet_type.t
                                 and type term = Term.term
                                 and type transition_table = TRS_type.t
                                 and type rule = TRS_type.rule) :
sig
  type alphabet = Alphabet_type.t
  type automaton = Automata_type.t
  type state = Automata_type.state
  type transition_table = TRS_type.t
  type transition = TRS_type.rule
  type gamma_content
  type variable_set = Variable_set_type.t
  type ('a, 'b) folder = ('a, 'b) Automata_type.folder
```

type $symbol = Symbol\_type.t$
type $rule = TRS\_type.rule$

148. the empty gamma content of gamma approximation

val $empty$: $gamma\_content$

149. adding a prior transition table to the gamma content

val $add\_prior\_to\_gamma$:
$transition\_table \rightarrow gamma\_content \rightarrow gamma\_content$

150. setting the gamma content to a new prior transition table

val $set\_prior\_in\_gamma$:
$transition\_table \rightarrow gamma\_content \rightarrow gamma\_content$

151. getting back the transition table from the gamma content

val get_prior : $gamma\_content \rightarrow transition\_table$

152. getting the current strategy of gamma

val $get\_strategy$: $gamma\_content \rightarrow string\ list$

153. setting the strategy in the gamma content

val $set\_strategy$: $string\ list \rightarrow gamma\_content \rightarrow gamma\_content$

154. the parsing function for gamma content

val $parse$:
$alphabet \rightarrow$
$variable\_set \rightarrow Genlex.token\ Stream.t \rightarrow gamma\_content$

155. the pretty printing function for gamma content

val to_string :
$gamma\_content \rightarrow string$

156. The gamma function used for approximation of new transitions obtained during completion. The *new_trans* are the new transitions to be normalised, *delta* is the transition table of the current completion step (stored into a folder for a better efficiency), a is the alphabet and *stops* are the state operators set on which transitions are defined, and finally i is the index of the lastly created new state. After an application of one step of gamma, the gamma content is updated, the new state operators contains all initial state operators as well as new state operators given by the user or constructed automatically as new states. The index has been incremented if automatic new states have been constructed

val $gamma$:
$new\_trans : transition\_table \rightarrow$

$$
\begin{array}{l}
delta : (symbol,\ (state,\ rule\ list)\ folder)\ folder\ \rightarrow \\
\quad gc : gamma\_content\ \rightarrow \\
\quad\quad a : alphabet\ \rightarrow \\
\quad\quad\quad stop : alphabet\ \rightarrow \\
\quad\quad\quad\quad ind : int \rightarrow\ transition\_table\ \times\ gamma\_content\ \times\ alphabet\ \times\ int
\end{array}
$$

157. the in and out channel used for user interaction

val $out\_chan\ :\ out\_channel\ ref$
val $in\_chan\ :\ in\_channel\ ref$

end

Index