Automates modulaires

Benoit Razet

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

automates modulaires

Benoît Razet

N° 5788

Décembre 2005

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--5788--FR

INRIA
ROCQUENCOURT

automates modulaires

Benoît Razet

Thème SYM — Systèmes symboliques
Projet Cristal

Rapport de recherche n° 5788 — Décembre 2005 — 43 pages

Résumé : L'algorithme de Berry-Sethi pour la compilation des expressions régulières en automates est efficace et présenté de manière élégante. Ce rapport fournit une implémentation de cette technique de compilation dans le langage de programmation **PidginML** avec l'innovation de compiler des expressions régulières que nous avons étendues. Son élégance tient dans son style de programmation purement fonctionnel et son respect de la complexité théorique de l'algorithme. La propriété de localité des automates obtenus est à la base de la modularité d'automates dans la boîte à outils de linguistique computationnelle *Zen2*. Ce rapport comporte également une justification formelle de la bonne formation des automates compilés, par un développement dans l'assistant de preuves **Coq**. Il discute également de l'adaptation de la boîte à outils *Zen2* au problème classique de recherche de chaînes de caractères dans un texte (commande *grep* Unix).

Ce travail a été effectué dans le cadre d'un stage de deuxième année du master parisien de recherche en informatique (MPRI), dont ce rapport constitue la version finale du mémoire.

Mots-clés : automate local, expression régulière, algorithme de Berry-Sethi, grep, automate modulaire

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Modular Automata

Abstract: The Berry-Sethi algorithm is efficient for the regular expression compilation into automata and presented with elegance. In this report we present an implementation of this compilation technique in the **PidginML** language with an inprovement consisting in the treatment of some extended regular expressions. The elegance comes from the purely fonctional style of the inplementation which respects the theoretical complexity of the algorithm. The locality property of the produced automata is the key of the notion of modular automata in the toolkit *Zen2*. This report presents also a formal justification of the well-formedness of the produced automata, through a session with the **Coq** proof assistant. It also discusses the adaptation of the *Zen2* toolkit to the classical problem of searching a string pattern in a text (the *grep* Unix command).

This work was effected as the final research internship of the MPRI master. This report is the final version of the master's thesis.

Key-words: local automaton, regular expression, Berry-Sethi algorithm, grep, modular automaton

Introduction

Gérard Huet a introduit la boîte à outils *Zen* [6] de traitement morphologique et phonétique des langues naturelles. Elle inclue une structure de données d'automates, appelée **aum** [5], pour automates mixtes, amalgamant une structure de décision déterministe et une structure de suites de points de choix non déterministes accommodant éventuellement des sorties (transduction). Cette structure a été notamment utilisée pour la représentation des lexiques, les opérations phonétiques et morphologiques et les algorithmes d'analyse syntaxique superficielle (segmentation, étiquetage). L'étape suivante serait de vouloir représenter un deuxième niveau de morphologie. Cette morphologie peut être décrite par une expression régulière, dont les symboles de base sont des **aums**, et pour qui la compilation en automate n'est pas possible du fait de l'explosion combinatoire. Il faut donc garder cette structure à deux étages : un ensemble d'aums pour différents lexiques et un automate pour la morphologie contrôlant l'exploration dans ces lexiques. L'intégration de cette considération dans *Zen* conduira à sa nouvelle version *Zen2*.

Nous présentons l'implémentation de la compilation en automate local de Berry-Sethi [1] pour une expression régulière décrite dans un langage que nous introduirons. Ce langage est défini pour spécifier la morphologie de deuxième niveau de manière concise par un système d'expressions régulières permettant un certain partage. Le code est donné en **PidginML**, qui est un sous-ensemble de **OCaml** [2]. Avec le souci de garantir la complexité minimale de l'algorithme de Berry-Sethi nous fournissons une preuve formelle **Coq** [3] d'un invariant sur la structure de données compilée traduisant l'automate local.

Pour illustrer le bon fonctionnement de *Zen2* nous montrons aussi comment l'adapter pour implanter la recherche de motif dans un texte (avec l'algorithme de Knuth-Morris-Pratt [7]), comme la commande *grep* de **UNIX**.

1 Rappels : langages, automates et expressions régulières

Cette partie n'a pas pour but d'être exhaustive sur les notions qu'elle présente, seulement d'introduire pour le lecteur des notions de base sur les langages rationnels, automates et expresions régulières que nous utiliserons dans la suite.

1.1 Langages rationnels

Nous rappelons qu'un mot sur un alphabet fini Σ est la juxtaposition finie de lettres de Σ. On considérera aussi le mot vide ϵ qui est l'élément neutre de cette opération de juxtaposition. Par exemple $abbbaa$ est un mot sur $\{a, b\}$. On appelle langage sur Σ un ensemble de mots sur Σ.

Le monoïde libre engendré par Σ est noté Σ^* et il correspond à l'ensemble des mots définis sur Σ.

On distingue le sous-ensemble de $\mathcal{P}(\Sigma^*)$ des langages rationnels défini inductivement de la manière suivante :

- Tout singleton de $\Sigma \cup \{\epsilon\}$ est rationnel.
- Si L_1 et L_2 sont des langages rationnels alors il en est de même de
  $L_1 \cdot L_2 = \{xy \mid x \in L_1, y \in L_2\}$ (concaténation)
  $L_1 \cup L_2$ (union)
  $L^* = \cup_{i=0}^{\infty} L^i$ (Cloture par opérateur de Kleene)
- Il n'y a pas d'autre langage rationnel.

Les langages rationnels sur Σ^* sont notés $Rat(\Sigma^*)$ et sont clos par les opérateurs définis ci-dessus.

1.2 Automates finis déterministes

Un automate sur l'alphabet Σ est défini par un ensemble fini Q d'*états*, un ensemble $I \subset Q$ d'*états initaux*, un ensemble $T \subset Q$ d'*états terminaux* et un ensemble $E \subset (Q \times \Sigma \times Q)$ de transitions. Il est noté :

$$(Q, I, T, E)$$

Un mot est *accepté* par l'automate si en partant d'un état initial, en passant d'une transition à une autre en stockant la lettre de la transition un nombre indéterminé de fois et si on arrive à un état terminal le mot formé par les transitions est bien le mot qu'on cherche à reconnaître. Les ensembles de mots reconnaissables par un automate fini sur Σ sont appelés *langages reconnaissables* et notés $Rec(\Sigma^*)$.

1.3 Expressions régulières

Les expressions régulières sur l'alphabet Σ sont les éléments de l'algèbre sur $\Sigma \cup \{0,1\}$ avec les symboles de fonctions $+$, $.$, $*$.
Il y a une correspondance L entre les termes de cette algèbre et les langages rationnels, elle est définie inductivement comme suit :

$$L(0) = \emptyset,\ L(1) = \epsilon,\ L(a) = \{a\},$$

$$L(E + F) = L(E) \cup L(F),$$
$$L(E.F) = L(E)L(F),$$
$$L(E^*) = L(E)^*.$$

1.4 Correspondances

Les automates et les expressions régulières sont deux manières de décrire des langages rationnels. Nous avons déjà vu la correspondance entre les expressions régulières et les langages rationnels. Le théorème de Kleene, énoncé ci-dessous, établit le lien entre les langages reconnaissables et les langages rationnels.

Théorème 1.1 (Kleene) *Les langages rationnels et reconnaissables sur un alphabet fini* Σ *sont équivalents :*

$$Rat(\Sigma^*) = Rec(\Sigma^*).$$

Une expression régulière est ce qui permet de représenter le plus intuitivement un langage rationnel. Elle est facilement représentable sous forme de structure récursive dans un langage de programmation informatique. Les automates sont facilement adaptables en un programme de reconnaissance de mots. On peut donc passer d'une représentation assez intuitive (expression régulière) des langages rationnels à un programme pour les reconnaître (automate). A chaque opérateur d'expression régulière correspond une opération de construction d'automates. Ces opérations font appel à diverses généralisations ou restrictions de la notion d'automate déterministe (automates non déterministes, automates déterministes minimaux, etc). L'assemblage de ces constructions nécessite donc a priori des intertraductions très coûteuses entre ces diverses caractérisations théoriquement équivalentes, par exemple la déterminisation et la minimisation.

Pour cette raison on souhaiterait introduire une notion de modularité pour les automates. Il est naturel que cette notion de modularité corresponde à la notion de clôture par substitution dans les expressions régulières.

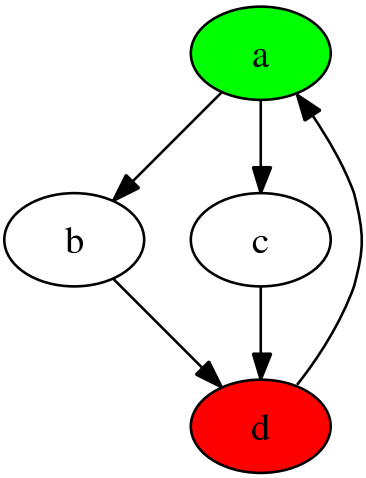

FIG. 1 – L'automate local de $(a \cdot (b + c) \cdot d)^*$

1.5 Automates locaux

Dans la grande famille des langages rationnels on distingue les langages locaux [4] pour lesquels à chaque lettre on associe un ensemble fini de lettres pouvant lui succéder. L'automate associé à un tel langage confond donc son ensemble d'états et les étiquettes des transitions : si on lit un caractère a on va vers l'unique état associé au caractère a. Un exemple d'automate local pour le langage rationnel $(a \cdot (b + c) \cdot d)^*$ est donné par la figure 1.

L'état initial est a (atteint si la première lettre lue est a) et l'état final est d. Les lettres sur les transitions sont omises parce qu'elles sont identiques à l'état d'arrivée de la transition. Il ne faut pas les confondre avec des automates à ϵ-transitions.

Dans l'article de Berry et Setthi [1] on trouve un algorithme de compilation des expressions régulières linéaires, c'est-à-dire des expressions régulières dans lesquels une lettre de l'alphabet n'apparaît qu'une fois, vers les automates locaux. Dans la suite nous parlerons de l'algorithme de Berry-Sethi pour désigner cet algorithme de construction.

2 Automates modulaires

Gérard Huet a introduit [6] une boîte à outils pour effectuer des traitements morphologiques et phonologiques. Dans cette boîte à outils est décrit une structure de donnée **aum** pour représenter des automates sous forme d'arbres déterministes décorés par des transitions non-déterministes pouvant représenter des règles d'euphonies. Il fournit aussi une machine réactive fonction d'une entrée, représentant une phrase, et l'analyse pour dire si celle-ci est une suite de mots appartenant au lexique respectant les règles d'euphonies.

L'ensemble des solutions fournies par la machine réactive est complète, toutes les segmentations possibles de la phrase sont explorées. Ce nombre de solutions peut être trop grand pour une étude complète par l'homme. Le problème que nous cherchons à résoudre est celui de passer à un niveau de description supérieur qui est de permettre la spécification de la morphologie d'une phrase pour éliminer des solutions ne respectant pas cette morphologie.

Pour ce faire il faut considérer que le lexique est maintenant subdivisé en différents lexiques représentant des variétés lexicales différentes. En français cela correspondrait à classer le lexique dans différentes catégories tels que les noms, adjectifs, verbes...

Ensuite il faut définir la morphologie. Nous avons choisi une représentation à base d'expression régulière. L'alphabet de ces expressions est l'ensemble des lexiques du langage. Utilisons la propriété de clôture par substitution des expressions régulières : En considérant les lexiques comme des expressions régulières nommées et la morphologie comme une expression régulière sur ces noms alors par substitution des noms de lexique dans la morphologie on obtient une grande expression régulière définissant le langage avec sa morphologie. Cependant dans notre approche nous garderons la morphologie dissociée du lexique. Pour ce faire nous utiliserons une structure à deux niveaux. La morphologie sera compilée en son automate local associé (algorithme de Berry-Sethi [1]) , lui-même macro-généré par **CamlP4** en un module **PidginML**. Le module introduira un type *phase* qui correspond aux états de l'automate. La correspondance entre les élements du type *phase* et les **aums** sera assuré par la fonction *transducer*. La reconnaissance des mots sera controlée par l'ensemble des phases accessibles depuis la phase en cours auquel on fait correspondre un ensemble d'**aums**.

Par exemple, si on définit une morphologie nécessitant un nom suivi d'un verbe suivi d'un nom, alors on commence la reconnaissance de la phrase *"Charles aime Odette"* en cherchant *"Charles"* dans le lexique des noms, *"aime"* dans le lexique des verbes et *"Odette"* dans le lexique des noms.

Nous rendons compte de ce travail par un texte en anglais rédigé dans le style *Literate programming*.

2.1 Introduction

We present a language for specifying rational languages in a modular style describing the alphabet (seen as aums) and using regular expressions on this alphabet. We may recognize strings over the language using the reactive engine linked with a switching module Dispatch. We provide a library to macro-generate this Dispatch module from the regular expression specification.

In this module, we describe the compilation of regular expression to local automata inspired by Berry-Sethi, then we give an algorithm to reduce a regular expression system into a simple regular expression. After giving the interface for generating code we show the Camlp4 parser that defines our regular expressions description language.

2.2 Module Compile_BS

We present an implementation of the Berry-Sethi algorithm for compiling regular expressions. We use Pidgin-ML, a functional programming language of the ML family. We define a module *Compile_BS*, which provides a data type for representing regular expressions, a type for standard local automata and a function *compile* that takes an initial state, a regular expression, and returns a local automaton representation.

```
module Compile_BS : sig
  type regexp α =
    [ One
    | Symb of α (* α is the type parameter of symbols *)
    | Union of regexp α and regexp α
    | Conc of regexp α and regexp α
    | Star of regexp α
    | Epsilon of regexp α
    | Plus of regexp α
    ]
  ;
  type marked α = (α × int)
  and local_auto α =
      ( marked α
      × list (marked α)
      × list (marked α × list (marked α))
      × list (marked α)
      )
  ;
  value compile : marked α → regexp α → local_auto α
;
end = struct
```

We describe regular expressions using a type abstracted from a basis alphabet α. This type provides extra constructors such as *Plus*, useful for practical applications

```
type regexp α =
  [ One
  | Symb of α
  | Union of regexp α and regexp α
  | Conc of regexp α and regexp α
  | Star of regexp α
  | Epsilon of regexp α
  | Plus of regexp α
  ]
;
```

One, *Symb*, *Union*, *Conc*, and *Star* are the classical operators. *Epsilon e* has the meaning of *Union One e* and *Plus e* the meaning of *Union e* (*Star e*), but treated as an atomic expression. In the following we call a "regular expression" a "regexp".
Berry-Sethi compilation applies to linear regexps to produce a local automaton. Let us recall that a linear regexp has the property that any symbol appears only once in it. This can be made by adding an integer to the symbol making the whole unique in the regexp, thus it changes a bit the structure of symbols for linear regexps. A local automaton is an automaton in which the guarded symbol of the transition corresponds with the arrival state of the transition. The types for marked symbols and local automata are :

```
type marked α = (α × int)
and local_auto α =
      ( marked α (* initial state *)
      × list (marked α) (* other states *)
      × list (marked α × list (marked α)) (* transitions *)
      × list (marked α) (* terminal states *)
      )
;
```

A local automata is structurally represented with four components. The first is the initial state, this shows that we have decided to produce standard local automata. The second is for the set of states of the automaton, excluding the initial one. The third is the set of transitions : a transition is a state and the list of states associated with the transition. The forth component is for the set of terminal states.
Now that we have given the data-structures for representing regexps let us give our algorithm to mark a regexp linear. To get the list of symbols from any regexp, reading left to right, we use the function
symb_lr : *regexp* α $\rightarrow$ *list* α

```
value symb_lr e = symb [] e
  where rec symb accu = fun
```

```
  [ One → accu
  | Symb s → [s :: accu]
  | Union e1 e2 | Conc e1 e2 →
    symb (symb accu e2) e1
  | Star e | Epsilon e | Plus e → symb accu e
  ]
;
```

Having computed the resulting list from *symb_lr*, we want to glue a unique number to each symbol, meaning the number of times we have encountered a symbol reading left to right the list, beginning to count from 1 except when a symbol is present only once in the list, then we count 0. Thus symbols that appear only once have the special mark 0.
mark_list : *list* α → *list* (*marked* α)

```
value mark_list l = markr [] l
  where rec markr laccu = fun
  [ [] → []
  | [x :: l] → let count_x = count 1
        where rec count sum = fun
        [ [] → sum
        | [y :: l] → count (if y = x then sum + 1 else sum) l
        ] in
    if List.mem x laccu ∨ List.mem x l (* multiples *)
    then [ (x, count_x laccu) :: markr [x :: laccu] l ]
    else [ (x,0) :: markr laccu l ]
  ]
;
```

mark_list has a *laccu* list that records symbols present at least twice.
Now we can define the *mark* function using a function *map_lr* that replace in a regexp, from left to right, the symbols of a regexp with a list of marked symbols (resulting from *mark_list*).
mark : *regexp* α → (*regexp* α × *marked* α × *list* (*marked* α))

```
value mark e =
  let rec map_lr li = fun
    [ One → (One, li)
    | Symb s →
        (Symb (List.hd li), List.tl li)
    | Union e1 e2 →
        let (e1_m, lj) = map_lr li e1 in
        let (e2_m, lk) = map_lr lj e2 in
        (Union e1_m e2_m, lk)
    | Conc e1 e2 →
        let (e1_m, lj) = map_lr li e1 in
```

```
        let (e2_m, lk) = map_lr lj e2 in
        (Conc e1_m e2_m, lk)
    | Star e1 →
        let (e1_m, lj) = map_lr li e1 in
        (Star e1_m, lj)
    | Epsilon e1 →
        let (e1_m, lj) = map_lr li e1 in
        (Epsilon e1_m, lj)
    | Plus e1 →
        let (e1_m, lj) = map_lr li e1 in
        (Plus e1_m, lj)
    ] in
  let symbols = symb_lr e in
  let symbols_m = mark_list symbols in
  let (e_m, li) = map_lr symbols_m e in
    (* note li must be [] *)
  (e_m, symbols_m)
;
```

The list of symbols mapped into the expression is *symbols_m* . Note that the first component of a marked symbol is the original one, so one can easily recover the original symbol from a marked one.

By now, we assume our regexps linear and we define the functions of Berry-Sethi compiling. It is very useful to get the information whether the empty string belongs to the regexp to compute the automaton efficiently. We then present a new type *d_regexp* for discriminating regexp that generates or not the empty string. It is almost the same as type *regexp* but with an information coding the discrimnination, and this for all sub-construction of the expression. It is represented with a boolean : *True* if the empty string is generated by the regexp, *False* otherwise.

```
type d_regexp α =
  [ DOne
  | DSymb of α
  | DUnion of bool and d_regexp α and d_regexp α
  | DConc of bool and d_regexp α and d_regexp α
  | DStar of d_regexp α
  | DEpsilon of d_regexp α
  | DPlus of bool and d_regexp α
  ]
;
```

DOne, *DSymb*, *DStar* and *DEpsilon* don't need this boolean because they have the information implicitly.

We simply extract this information from a regexp in a constant time analyzing the top node of a regexp :
delta : *d_regexp* α $\rightarrow$ *bool*

```
value delta = fun
  [ DOne → True
  | DSymb _ → False
  | DUnion b _ _ | DConc b _ _ → b
  | DStar _ | DEpsilon _ → True
  | DPlus b _ → b
  ]
;
```

The following algorithm transforms a regexp of type *regexp* into the discriminating one of type *d_regexp*.
discr : *regexp* α $\rightarrow$ *d_regexp* α

```
value rec discr = fun
  [ One → DOne
  | Symb s → DSymb s
  | Union e1 e2 →
      let de1 = discr e1
      and de2 = discr e2 in
      DUnion (delta de1 ∨ delta de2) de1 de2
  | Conc e1 e2 →
      let de1 = discr e1
      and de2 = discr e2 in
      DConc (delta de1 ∧ delta de2) de1 de2
  | Star e → DStar (discr e)
  | Epsilon e → DEpsilon (discr e)
  | Plus e →
      let de = discr e in
      DPlus (delta de) de
  ]
;
```

The cost of this computation is linear with the size of the regexp. Then we give an implementation of the "first" function, that gives the first symbols from a regexp, in linear time :
first : *list* α $\rightarrow$ *d_regexp* α $\rightarrow$ *list* α

```
value rec first l = fun
  [ DOne → l
  | DSymb d → [d :: l]
  | DUnion _ e1 e2 → first (first l e2) e1
```

```
  | DConc _ e1 e2 →
      let b1 = delta e1 in
      if b1 then first (first l e2) e1
      else first l e1
  | DStar e | DEpsilon e | DPlus _ e → first l e
  ]
;
```

The parameter l is for already computed first elements, a partial result.
A follow set is the list of directly accessible symbols from one in a regexp, it corresponds to the notion of "continuation" in the Berry-Sethi article. Now we have all the routines to present an implementation of the "F" function presented in the Berry-Sethi that computes the set of all follow sets.
follow : $\alpha \rightarrow$ *regexp* $\alpha \rightarrow$ *list* $(\alpha \times$ *list* $\alpha)$

```
value follow initial_state exp =
  let rec f1 exp l fol =
    match exp with
    [ DOne → fol
    | DSymb d → [(d, l) :: fol]
    | DUnion _ e1 e2 →
       let fol2 = f1 e2 l fol in
       f1 e1 l fol2
    | DConc _ e1 e2 →
       let fol2 = f1 e2 l fol in
       let l1 = if delta e2 then first l e2 else first [] e2 in
       f1 e1 l1 fol2
    | DStar e | DPlus _ e →
       let l_res = first l e in
       f2 e l_res fol
    | DEpsilon e → f1 e l fol ]
  and f2 exp l fol = (* (first [] exp) already in l *)
    match exp with
    [ DOne → fol
    | DSymb d → [(d, l) :: fol]
    | DUnion _ e1 e2 →
       let fol2 = f2 e2 l fol in
       f2 e1 l fol2
    | DConc _ e1 e2 →
       let b1 = delta e1
       and b2 = delta e2 in
       if b1 (* l1 and l2 in l *)
       then if b2
            then f2 e1 l (f2 e2 l fol)
```

```
            else f1 e1 (first [] e2) (f2 e2 l fol)
        else if b2
            then f2 e1 (first l e2) (f1 e2 l fol)
            else f1 e1 (first [] e2) (f1 e2 l fol)
    | DStar e | DEpsilon e | DPlus _ e → f2 e l fol
    ] in
  let fl_sets = f1 exp [] []
  and l = first [] exp in
  [(initial_state, l) :: fl_sets]
;
```

The initial state is a parameter of the function because it is not a state derived from symbols of the regexp. An outside hand have to choose this value taking care not already being in the regexp. Because of our implementation of sets with list we must guarantee not returning a list with duplicated elements. This can be done having the property that (*first exp*) is already in l or not. The computation is different in both case this is why we have *f1* and *f2*. The initial state of the resulting automaton is a parameter of the function and we add a link between this initial state and the first states of the regexp.
To decide the ending symbols we introduce the function *last* that gives the last symbols from a regexp
last : $\alpha \rightarrow$ *d_regexp* $\alpha \rightarrow$ *list* α

```
value last initial_state e =
  let rec last_rec exp l = match exp with
    [ DOne → l
    | DSymb d → [d :: l]
    | DUnion _ e1 e2 →
        let l2 = last_rec e2 l in
        last_rec e1 l2
    | DConc _ e1 e2 →
        let b2 = delta e2 in
        if b2 then last_rec e1 (last_rec e2 l)
        else last_rec e2 l
    | DStar e | DEpsilon e | DPlus _ e → last_rec e l
    ] in
  let l = last_rec e [] in
  if delta e then [initial_state :: l] else l
;
```

initial_state is the initial state of the automaton. We add it to the set of last states if the empty word belongs to the language.
In terms of the note on "Local languages and the Berry-Sethi algorithm" by Jean Berstel and Jean-Eric Pin, we have presented algorithms to compute a linear regular expression into a local standard automaton. Adding the *Plus e* directly as an operator in the abstract syntax

is an optimisation because the equivalent *Conc e* (*Star e*) duplicates marked symbols and thus the number of states of the automaton.
We now present the *compile* function that computes the automaton from a regexp.
compile : *marked* α $\rightarrow$ *regexp* α $\rightarrow$ *local_auto* α

```
value compile initial_state exp =
  let (exp_m, states) = mark exp in
  let d_exp = discr exp_m in
  let fol = follow initial_state d_exp
  and lasts = last initial_state d_exp in
  (initial_state, states, fol, lasts)
;
end ;
```

This is the end of our Berry-Sethi compilation algorithm. We ensure that we have presented an implementation that takes care of respecting the theoretical complexity, that is quadratic in the number of symbols in the regexp. The proof of the complexity is by induction on the regexp structure.

2.3 Module Regexp_system

We extend the way to define regexps with a regexp system that allows sharing.

```
module Regexp_system = struct
```

Let use the structure of regular expressions defined in module *Compile_BS*

```
open Compile_BS;
```

We mean by modularity the possibility of naming a regexp which may be used in another regexp as if it was a basic symbol. Now a *Symb* in a regexp can be a name or a symbol from an alphabet and we define a type *mix_symb* for describing this mix.

```
type name = string
and mix_symb α =
    [ Name of name
    | Alph of α
    ]
;
```

Then we define a *system* as a list of names and associated regexp :

```
type system α = list (name × regexp (mix_symb α))
;
```

We give an algorithm to transform a system into a simple regular expression.
flatten : *system* α $\rightarrow$ *regexp* α

```
value flatten sys =
  let rec flatten_regexp system l = fun
    [ One → (One, l)
    | Symb (Name s) →
      try (* we try to find s in already flattened regexp *)
        let e_flattened = List.assoc s l in
        (e_flattened, l)
      with [ Not_found →
             let rec extract_s = fun
                 [ [] → failwith "no extraction"
                 | [(s2, e) :: sys] → if s = s2 then (e, sys)
                                      else extract_s sys
                 ] in
             (* knowing that dependancies are always in the rest of system *)
             let (e, new_sys) = extract_s system in
             let (e_flattened, new_l) = flatten_regexp new_sys l e in
             (e_flattened, [(s, e_flattened) :: new_l])]
    | Symb (Alph s) → (Symb s, l)
    | Union e1 e2 →
        let (e1_f, l_left) = flatten_regexp system l e1 in
        let (e2_f, l_right) = flatten_regexp system l_left e2 in
        (Union e1_f e2_f, l_right)
    | Conc e1 e2 →
        let (e1_f, l_left) = flatten_regexp system l e1 in
        let (e2_f, l_right) = flatten_regexp system l_left e2 in
        (Conc e1_f e2_f, l_right)
    | Star e →
        let (e_f, new_l) = flatten_regexp system l e in
        (Star e_f, new_l)
    | Epsilon e →
        let (e_f, new_l) = flatten_regexp system l e in
        (Epsilon e_f, new_l)
    | Plus e →
        let (e_f, new_l) = flatten_regexp system l e in
        (Plus e_f, new_l)
    ] in
  let (e, system) = match sys with
    [ [] → failwith "empty system!!"
    | [(_, e) :: system] → (e, system)
    ] in
  let (e_f, _) = flatten_regexp system [] e in
  e_f
```

```
;
end;
```

Parameter *l* of *flatten_regexp* defines the list of regular expressions already flattened for a kind of lazy evaluation, it is initialized to the empty list. Parameter *system* represents the list of couple - name and regexp - not treated. The function *flatten_regexp*, providing a system and a list of expressions already flattened, replaces each symbol that is a name of regular expression by the associated one.

2.4 The concrete syntax for modular aums

A modular aum is a two-level structure : a regexp defined over an aum alphabet. And now we precise the concrete syntax for defining modular aums. It includes a name for the initial state, together with the name of the aum which recognizes the empty word, a basic alphabet of symbols represented as lower case strings between delimiters **alphabet** and **end** (corresponding to names of aums), then follows the definition of the regexp as a regexps system, between delimiters **automaton** and **end** together with a parameter for the name of the module implementing the state transitions.

```
module Parser = struct

open Compile_BS;
open Regexp_system;

Using a standard lexer

value gram =
  let lexer = Plexer.gmake () in
  Grammar.gcreate lexer;

We define the entry point of grammar def_auto

value def_auto = Grammar.Entry.create gram "def_auto";

Here is the definition of the grammatical construction for a concrete system of regular
expressions.

EXTEND
  GLOBAL : def_auto;
  def_auto :
    [ [ "initial"; init = LIDENT; empty_aum = LIDENT;
        "alphabet"; aums = aum_names ; "end";
        "automaton"; module_name = UIDENT;
        system = rule_list; "end"; EOI →
           (empty_aum, init, aums, module_name, system)
      ]
```

```
    ];
  aum_names :
    [ [ l = LIST1 LIDENT SEP ";" → l ]
    ];
  rule_list :
    [ [ r = LIST1 rule SEP "in" → r ]
    ];
  rule :
    [ [ "node"; name = UIDENT; "="; e = expreg → (name, e) ]
    ];
  expreg :
    [ [ e1 = expreg; "|"; e2 = expreg → Union e1 e2 ]
    | [ e1 = expreg; "."; e2 = expreg → Conc e1 e2 ]
    | [ e = expreg; "*" → Star e
      | e = expreg; "?" → Epsilon e
      | e = expreg; "+" → Plus e ]
    | [ n = INT →
          match int_of_string n with
            [ 1 → One
            | _ → failwith "integer not autorized"
            ]
      | name = LIDENT → Symb (Alph name)
      | name = UIDENT → Symb (Name name)
      | "("; e = expreg ; ")" → e ]
    ];
END;
```

Let us give an example of modular aum defined in such a description :

```
initial init epsilon_aum

alphabet
  noun ; root; unde; abso; iic; iiv; auxi; ifc; prev
end

automaton Disp
  node INVAR = prev.abso | unde in
  node CONJUG = prev? . root in
  node SUBST = (iic* ).noun | (iic +).ifc in
  node VERB = CONJUG | iiv.auxi in
  node PHRASE = (SUBST | VERB | INVAR)+
end
```

It denotes a not so easy to write regexp after being flattened.
the construction `automaton ... end` is parameterized by a string which will be the name of the module for the resulting automaton. We assume that this functor will be used with a parameter module *Auto* defining a type *auto* of aums, and a vector *auto_vect* indexed by the alphabet, plus *epsilon_aum*. Note that uppercase strings stand for regexp metavariables, whereas lowercase strings stand for aum names.

```
end ;
```

2.5 Module Generate_ast

Let us define a module for generating the abstract syntax tree of a program implementing a phase automaton, assuming we provided a name for the aum of empty word, a list of aums used in the definition of the automaton, a name for the module associated to the automaton, the phase representing the first state of the automaton, the list of phases, the follow sets and the list of terminal states.
Dummy location needed for the quotation mechanism - must be called loc

```
value loc = Token.make_loc (0, 0)
;
module Generate_ast : sig

  type aum_name = string
  and module_name = string
  and phase = (string × int)
  and follows = list (phase × list phase)
  and program = MLast.str_item;
  value gen_ast : aum_name → list aum_name → module_name →
        phase → list phase → follows →
        list phase → program;

  end = struct
```

The implementation consists in encapsulating the structures in the given datatypes and functions, with help of the macro-generating facilities of Camlp4. The reader is advised to skip this section at first reading.

```
  type aum_name = string
  and module_name = string
  and phase = (string × int)
  and follows = list (phase × list phase)
  and program = MLast.str_item
;
```

To glue a number i to *name* capitalized, except for the number 0 :

```
value convert_uid_int (name, i) =
  let cap = String.capitalize name in
  if i = 0 (* the name is used once in the regular expression *)
  then cap else cap ^ string_of_int i
;
```

generates the type record *auto_vect*

```
value gen_type_vect phases =
  let f = fun n → (loc, n, False, <: ctyp < Auto.auto >>) in
  let type_record = List.map f phases in
    < :str_item < type auto_vect = { $list :type_record$ } >>
;
```

generates the type *phase*

```
value gen_type_phase phases =
  (* first compute the names of all phases *)
  let list_type = List.map convert_uid_int phases in
  let sslt = List.map (fun x → (loc , x, [])) list_type
  in < :str_item < type phase = [ $list :sslt$ ] >>
;
```

generates the *transducer* function

```
value gen_fun_morphism phases =
  let pwel =
      let process (x, y) =
          let auto = <: expr < $lid : y$ >> in
          (<: patt < $uid : x$ >>, None,
           < :expr < Fsm.autos.$auto$ >>) in
      List.map process phases in
  < :str_item < value transducer = fun [ $list :pwel$ ] >>
;
```

generates the *dispatch* function

```
value gen_fun_dispatch follows =
  (* translates a follow *)
  let trad_a_follow (n, ln) =
    let tln = List.fold_right tr ln <: expr < [] >>
        where tr x l =
          let x' = convert_uid_int x in
          < :expr < [$uid : x'$ :: $l$] >>
    in (<: patt < $uid : convert_uid_int n$ >> , None, tln)
  in (* translates follow sets *)
  let match_list = List.map trad_a_follow follows
```

```
  in < :str_item < value dispatch  = fun [ $list :match_list$ ] >>
;
value gen_initial_state initial_phase  =
  < :str_item < value initial  =  $uid : convert_uid_int initial_phase$  >>
;
```

Generation of the list of terminal states

```
value gen_fun_terminal l  =
  let the_list  =
    List.fold_right tr l  <: expr <  [ ]  >>
    where tr e l  =
          let e'  = <: expr < $uid :  convert_uid_int e$ >> in
          < :expr < [ $ e'$  ::  $l$] >>
  in < :str_item <value terminal  =
       fun phase  →  List.mem phase [$list :the_list$]  >>
;
```

generates the module with name *module_name*

```
value gen_module empty_aum module_name initial_phase phases
                  follows terminal  =
  (* declaration and definition of types and functions *)
  let type_phase  =  gen_type_phase [initial_phase  ::  phases]
  and fun_morphism  =  gen_fun_morphism
    [ (convert_uid_int initial_phase,  empty_aum) ::
        (List.map (fun (x, y)  →  (convert_uid_int (x, y), x)) phases)]
  and fun_dispatch  =  gen_fun_dispatch follows
  and value_initial  =  gen_initial_state initial_phase
  and fun_terminal  =  gen_fun_terminal terminal in
  let type_and_def  =
    [ type_phase
    ; fun_morphism
    ; fun_dispatch
    ; value_initial
    ; fun_terminal
    ] in
  let me  = <: module_expr < struct $list :type_and_def$ end >> in
  (* end of decl and def *)
  < :str_item < module $module_name$  =
       functor ( Fsm  : sig value autos  :  auto_vect; end )  →  $me$ >>
;
```

generates all declarations of the file we want to generate

```
value gen_ast empty_aum aums module_name initial_phase phases
                follows terminal =
  let type_vect = gen_type_vect [empty_aum :: aums] in
  let module_body =
    gen_module empty_aum module_name initial_phase phases
                follows terminal in
  let structure = [type_vect; module_body] in
  let module_content = <: module_expr < struct $list :structure$ end >> in
  let automata_functor =
      < :str_item < module Automata =
        (* Automata is a functor with parameter module Auto *)
        functor (Auto : sig type auto = α; end) → $module_content$ >>
  in automata_functor
;
end;
```

2.6 Printing from an input channel

```
open Compile_BS;
open Parser;
```

Given an input channel for a phase automaton, we define a function *print_automaton* that parses using the entry point *def_auto* in a tuple that includes the system of regular expressions and pretty-print the code generated after compiling by Berry-Sethi.

```
value print_automaton ch =
  let (empty_aum, init, aums, module_name, system) =
      Grammar.Entry.parse def_auto (Stream.of_channel ch) in
  let exp = Regexp_system.flatten (List.rev system)
  and initial_phase = (init,0) in
  let (initial_phase, phases, follows, terminal) =
    compile initial_phase exp in
  let ml_sentence = Generate_ast.gen_ast
                        empty_aum aums module_name
                        initial_phase phases
                        follows terminal in
  Pcaml.print_implem.val [(ml_sentence, loc)]
;
```

For generating the code from an concrete automaton in file "dispatch.aut" one may call *print_automaton* (*open_in* `"dispatch.aut"`); which will pretty-print the result on standard output, where it may be redirected to a file. For instance, the modular aum described above generates the following code :

```
module Automata (Auto : sig type auto = 'a; end) =
   struct
    type auto_vect =
      { epsilon_aum : Auto.auto;
        noun : Auto.auto;
        root : Auto.auto;
        unde : Auto.auto;
        abso : Auto.auto;
        iic : Auto.auto;
        iiv : Auto.auto;
        auxi : Auto.auto;
        ifc : Auto.auto;
        prev : Auto.auto }
    ;
    module Disp (Fsm : sig value autos : auto_vect; end) =
      struct
        type phase =
          [ Init
          | Iic1
          | Noun
          | Iic2
          | Ifc
          | Prev1
          | Root
          | Iiv
          | Auxi
          | Prev2
          | Abso
          | Unde ]
        ;
        value transducer =
          fun
          [ Init -> Fsm.autos.empty_aum
          | Iic1 -> Fsm.autos.iic
          | Noun -> Fsm.autos.noun
          | Iic2 -> Fsm.autos.iic
          | Ifc -> Fsm.autos.ifc
          | Prev1 -> Fsm.autos.prev
          | Root -> Fsm.autos.root
          | Iiv -> Fsm.autos.iiv
          | Auxi -> Fsm.autos.auxi
          | Prev2 -> Fsm.autos.prev
```

```
          | Abso -> Fsm.autos.abso
          | Unde -> Fsm.autos.unde ]
        ;
        value dispatch =
          fun
          [ Init -> [Iic1; Noun; Iic2; Prev1; Root; Iiv; Prev2; Unde]
          | Iic1 -> [Iic1; Noun]
          | Noun -> [Iic1; Noun; Iic2; Prev1; Root; Iiv; Prev2; Unde]
          | Iic2 -> [Iic2; Ifc]
          | Ifc -> [Iic1; Noun; Iic2; Prev1; Root; Iiv; Prev2; Unde]
          | Prev1 -> [Root]
          | Root -> [Iic1; Noun; Iic2; Prev1; Root; Iiv; Prev2; Unde]
          | Iiv -> [Auxi]
          | Auxi -> [Iic1; Noun; Iic2; Prev1; Root; Iiv; Prev2; Unde]
          | Prev2 -> [Abso]
          | Abso -> [Iic1; Noun; Iic2; Prev1; Root; Iiv; Prev2; Unde]
          | Unde -> [Iic1; Noun; Iic2; Prev1; Root; Iiv; Prev2; Unde] ]
        ;
        value initial = Init;
        value terminal phase =
          List.mem phase [Noun; Ifc; Root; Auxi; Abso; Unde]
        ;
      end
    ;
  end
;
```

3 Preuve de programme

Nous nous sommes intéressés à une preuve concernant la partie de compilation de l'automate local à partir d'une expression régulière. Pour effectuer cette preuve nous avons utilisé le système de preuve formelle **Coq** [3].

3.1 Motivations

L'algorithme de compilation de Berry-Sethi a une complexité quadratique en fonction de la taille de l'expression régulière. Lors de l'implémentation nous avons pris soin de conserver cette complexité tout en fournissant un code purement fonctionnel. Le plus souvent, l'implémentation d'un algorithme dans un langage purement fonctionnel diminue ses performances théoriques. Pour notre problème nous avons réussi à utiliser seulement les opérations primitives sur les listes tout en garantissant des constructions de listes sans doublons. Cela a permis de conserver la complexité souhaitée. En revanche la preuve de ce résultat a nécessité de comprendre dans quels cas l'algorithme pouvait construire des listes avec doublons si on implémentait de manière trop naïve la fonction **F** de Berry-Sethi [1]. Cette compréhension se retrouve dans le codage des fonctions mutuellement récursives *f1* et *f2* qui effectuent le travail de la fonction **F**. Pour retrouver l'algorithme produisant des doublons il suffit de supprimer *f2* et remplacer ses occurrences par *f1*. Le meilleur moyen pour vérifier que notre construction ne produisait pas de liste avec doublons a été d'en réaliser une preuve formelle assistée par le système **Coq**.

Nous rappelons que l'ensemble des transitions de l'automate est codé par le type :

$$list\ (\alpha\ \times\ (list\ \alpha))$$

La non présence de doublon concerne sa partie $(list\ \alpha)$.

La construction *list* n'assure pas l'unicité des éléments. Pour coder la fonction *f1*, le choix d'une structure plus haut-niveau, interdisant les doublons aurait affecté la complexité. Pour cette raison la fonction *f2* est appelée à la place de *f1* si les premiers symboles accessibles dans l'expression sont déjà dans la liste des successeurs. Nous voulons prouver, par ce mécanisme, que nous ne construisons que des listes de transitions sans doublons.

3.2 Adaptation des algorithmes à l'assistant de preuves Coq

Le système **Coq** permet de décrire des programmes fonctionnnels qui terminent et d'effectuer des preuves sur ces programmes. Aussi tous les algorithmes du module *Compile_BS* terminent. Pour chaque fonction traduite vers **Coq**, la preuve de terminaison se fait par récursion structurelle sur l'expression régulière. Le fait que la traduction réussisse le prouve. Voici la liste des fonctions nécessaires pour expliciter le théorème que nous souhaitons prouver :

- *mark*
- *delta*
- *discr* renommée en *delta_init* en **Coq**
- *first*
- *f1* renommée en *cont* en **Coq**
- *f2* renommée en *cont2* en **Coq**
- *follow* renommée en *continuations* en **Coq**

Le langage **PidginML** étant lui-même un langage fonctionnel il a été facile de traduire le code **PidginML** vers du code **Coq**. Cette opération a consisté à modifier la syntaxe et indiquer les types des valeurs et arguments de terminaison.

La formalisation **Coq** de ces fonctions est donnée en annexe. Quelques changements par rapport à la version **PidginML** des fonctions et structures des expressions régulières sont à noter :

- Nous considérons les expressions régulières sur l'alphabet des entiers de type *nat*.
- La fonction de marquage d'une expression linéarise l'expression régulière sans conserver l'ancienne valeur du symbole. Typiquement *mark (Or (Symb 43) (Symb 57))* retournera l'expression régulière linéarisée *(Or (Symb 0) (Symb 1))*.
- La fonction *continuations* regroupe plusieurs étapes : la linéarisation, le changement de structure d'expression régulière vers l'expression régulière discriminante, l'appel à *cont* et l'ajout des premiers éléments à l'état initial.
- Le constructeur *Plus* n'est pas traité.

3.3 La propriété à prouver

Nous cherchons à démontrer la propriété qu'il n'y a pas de redondance dans la liste des transitions possibles de l'automate local.

Spécifions les notions absence et unicité d'élément dans une liste :

```
Inductive Not_in_list : nat -> list nat -> Prop :=
| Not_in_list_nil : forall n, Not_in_list n nil
| Not_in_list_cons :  forall n n0 l,
  Not_in_list n l -> n <> n0 ->
  Not_in_list n (n0::l)
.

Inductive Unique_list : nat -> list nat -> Prop :=
| Unique_list_eq : forall n l, Not_in_list n l -> Unique_list n (n::l)
| Unique_list_neq : forall n1 n2 l,
  n1 <> n2 -> Unique_list n1 l -> Unique_list n1 (n2::l)
.
```

Maintenant définissons un type inductif pour spécifier des listes sans doublons :

```
Definition Wf_list (l:list nat) :=
(forall n, Not_in_list n l \/ Unique_list n l).
```

Et étendons cette propriété au type de retour de *follow* :

```
Inductive Wf_follow : list (nat * (list nat)) -> Prop :=
| Wf_fol_nil : Wf_follow (nil)
| Wf_fol_cons : forall n l fl,
  Wf_list l ->  Wf_follow fl -> Wf_follow ((n,l)::fl)
.
```

Nous pouvons maintenant énoncer la propriété *wf_continuations* à prouver :

```
Theorem wf_continuations : forall exp n,
  Wf_follow (continuations n exp).
```

Un certain nombre de lemmes intermédiaires ont été nécessaires pour cette preuve de théorème, ils sont aussi fournis en annexe. Les preuves ne présentent pas de difficulté notable et nous ne les fournissons pas (elles se font principalement par récurrence sur la structure des expressions régulières).

4 Grep

Nous avons choisi d'implémenter la commande **UNIX** *grep* dans *Zen2* pour montrer la flexibilité de cette boîte à outils.

On rappelle que la commande *"grep pat file"* affiche toutes les lignes qui incluent le motif *pat* dans le fichier *file*.

Zen2 permet de réaliser des automates modulaires qui réalisent des sorties fonctions du parcours dans le transducteur, ce sont donc des transducteurs modulaires. Cependant nous n'utilisons pas cette fonction de tansduction pour l'adaptation de l'algorithme de Knuth-Morris-Pratt [7] à *Zen2* que nous présentons. Ce qui justifie le fait que nous n'utilisons pas toutes les possibilités des structures de données présentées.

4.1 Alphabet et mots dans *Zen2*

Le module suivant décrit les structures de données pour l'alphabet et mots sur cet alphabet.

```
module Word : sig

type letter = int
and word = list letter;
type delta = (int × word);

end ;
```

Les lettres sont représentées par des entiers machine et les mots sont des listes de lettres. Le type *delta* ne sera pas utilisé dans la suite.

4.2 Aums dans *Zen2*

La structure d'automates fournies dans *Zen2* est une structure décrite par le module suivant :

```
module Auto : sig

type continuation = (Word.word × Word.word)
and transition =
  [ External of (Word.word × continuation)
  | Internal of (Word.word × Word.delta)
  ]
;

type auto = [ State of (deter × choices) ]
and deter = list (Word.letter × auto)
and choices = list transition;
```

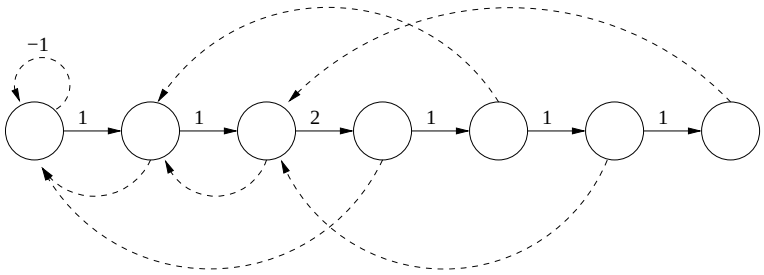

FIG. 2 – **aum**grep du motif **112111**

end ;

Nous n'utiliserons pas les transitions de type *Internal* dans la suite. Une transition *External(w,c)* reconnaît w sur l'entrée et effectue la continuation $c = (u,v)$ vers une des phases suivantes. Cette continuation vers une phase *phi* consiste à aller à l'état accessible depuis l'**aum** associé à *phi* de chemin d'accès déterministe v.

4.3 Construction de la table en aum

L'idée simple de l'algorithme de Knuth, Morris et Pratt est la suivante : Lors de l'exploration de la correspondance du motif avec la chaîne d'entrée de départ, lorsqu'un caractère du motif ne correspond plus, on n'a pas besoin de recommencer l'exploration depuis le début du motif en décalant la lecture de la chaîne d'entrée d'une lettre, on peut exploiter l'information de coincidence du segment de motif déjà validé avec un segment de motif plus petit qu'on explorera à l'avancée de la chaîne.

Un exemple d'**aum** qui implémente la relation la recherche de motif est donné à la figure 2 et représente l'**aum** du motif **112111**.

Les transitions déterministes sont représentées en trait plein et les non-déterministes en pointillés. Aucune des transitions non-déterministes ne possède de mot pour condition de saut excepté l'état initial qui a une transition étiquetée par le code -1 qui sera interprété par n'importe quelle lettre.

Voici la description de l'algorithme de construction de la table :

open *Auto*;

Nous devons définir un certain nombre de fonctions auxiliaires triviales pour faciliter la construction de la table associée au motif
value *get_continuation* : *nat* → *auto* → *word* ;

```
value access : word → word → word ;
value add : letter → choices → auto ;

value rec get_continuation n state =
  if n = 0
  then match state with
    [ State (_, [External(_, (_, v))]) → v
    | _ → failwith "bad argument for next with n = 0"
    ]
  else match state with
    [ State ([(_, next_state)], _) →
        get_continuation (n − 1) next_state
    | _ → failwith "bad argument for next with n != 0"
    ]
and access l = fun
  [ [] → l
  | [_ :: rest] → access (List.tl l) rest
  ]
and add letter loop = fun
  [ State ([], cont) → State ([(letter, State([], loop))], cont )
  | State ([(b, next_state)], cont) →
      State ([(b, add letter loop next_state)], cont)
  | _ → failwith "error adding state"
  ]
;
```

La fonction de construction de la table associée à un motif est *table_building* avec les paramètres suivant :
patt pour le motif restant à explorer,
word pour le motif complet,
partial pour le motif restant qui correspond avec le motif à explorer,
read pour l'inverse du motif qui correspond et
state pour l'automate resultant de ce qui a déjà été construit.

```
value table_building : word → word → word → word → auto → auto ;

value rec table_building patt word partial read state =
  match patt with
    [ [] → state
    | [ letter :: rest ] → match partial with
        [ [] → failwith "error 1"
        | [ l :: r] →
            if (letter = l)
            then
              let new_read = [l :: read] in
```

```
                let new_state =
                  add letter [External([], ([], List.rev new_read))] state in
                  table_building rest word r new_read new_state
              else
                if (read = [])
                then
                  let next_state = add letter [External([], ([], []))] state in
                    table_building rest word word [] next_state
                else
                  let v = get_continuation (List.length read) state in
                  let new_partial = access word v in
                    table_building patt word new_partial (List.rev v) state
        ]
    ]
;
```

On appelle *table_building* avec l'état initial *initial_state* défini comme suit :

```
value create_aum : word → auto ;

value create_aum pattern =
  let initial_state = (State([], [External([−1], ([], []))]) )
  in table_building pattern pattern [−1 :: pattern] [] initial_state;
```

La définition de l'aum représentant le mot vide est le suivant :

```
value empty_aum = State ([], [External([], ([], []))]);
```

Certaine parties de la machine réactive de *Zen2* ont du être modifiées. En particulier la condition d'acceptance de la chaîne est maintenant de se trouver dans un état de l'**aum** qui n'a plus de transition déterministe. Aussi il faut avoir la propriété qui assure un parcours prioritaire des branches déterministes de l'**aum**, ce qui était déjà le cas dans la machine réactive dans *Zen* (la fonction *react*). Dans ce cas, si le motif est dans l'entrée alors le résultat de l'analyse est obtenu en première passe de la machine réactive de *Zen2* (sans avoir besoin de revenir en arrière).

4.4 L'automate de phase de *grep*

L'automate de phase est trivial pour *grep*, il consiste à reconnaître au moins une fois l'**aum** construit à partir du motif :

```
initial init empty_aum

alphabet
  pattern
end
```

```
automaton Disp
  node GREP = pattern +
end
```

4.5 Résultats

Cette partie a montré que la boîte à outils *Zen* est assez générale et facilement adaptable à la résolution d'un problème classique. Il faut admettre que les performances ne sont pas celles obtenues par la commande *grep*. En revanche l'expérimentation a montré la complexité attendue : linéaire en la taille du texte à explorer.

Conclusion

Les parties programmation et preuve dans ce travail sont basés sur une méthodologie de génie logiciel bien pensée et caractérisée par les aspects suivants :

- L'utilisation de la programmation applicative conduit à des programmes bien structurés, dont la sûreté est accrue et sur lesquels des preuves formelles peuvent être réalisées.
- La programmation fonctionnelle, grâce au langage **OCaml**, a été essentielle pour écrire des algorithmes concis.
- La modularité a permis une structuration du programme traduisant les interactions entre les différentes parties du programme avec des interfaces claires et bien documentées, et un développement rapide d'unités de compilation indépendantes.
- La meta-programmation avec **CamlP4** [2] permet d'adapter des algorithmes généraux pour des besoins plus spécifiques et d'intégrer au langage de programmation les notations adéquates aux structures pariculières des applications.
- Malgré tous ces points qui constituent une vision très *haut-niveau* de la réalisation d'un programme, le travail produit résout effectivement le problème initial dans un environnement d'exécution réaliste (traitement du sanscrit : `http ://sanskrit.inria.fr`).

Références

[1] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48(1) :117–126, 1986.

[2] Objective Caml. http ://caml.inria.fr.

[3] Coq. http ://coq.inria.fr.

[4] Jean Berstel et Jean-Eric Pin. Local languages and the Berry-Sethi algorithm. *Theoretical Computer Science*, 155 :439–446, 1996.

[5] G. Huet. Automata mista. *verification : theory and practice : essays dedicated to zohar manna on the occasion of his 64th birthday. Springer-Verlag LNCS*, 2772 :359–372. `http ://pauillac.inria.fr/~huet/PUBLIC/zohar.pdf`, 2003.

[6] G. Huet. A functional toolkit for morphological and functional procession, application to a sanskrit tagger. *J. Functional programming*, 15, 2005, `http ://pauillac.inria.fr/~huet/PUBLIC/tagger.pdf`.

[7] Robert Sedgewick. Algorithms. *Addisson-Wesley*, 1988.

A Définitions de fonctions et structures de données Coq pour l'algorithme de Berry-Sethi

```
Require Import List.
Require Import Bool.

Inductive regexp : Set :=
  | One : regexp
  | Symb : nat -> regexp
  | Or :  regexp -> regexp -> regexp
  | And : regexp  -> regexp  -> regexp
  | Star : regexp  -> regexp
.

Fixpoint mark_rec (exp : regexp) (n: nat) {struct exp}:
  (regexp * nat) :=
  match exp with
  | One => (One, n)
  | Symb s => (Symb n, S n)
  | Or e1 e2 =>
      let c1 := mark_rec e1 n in
      let (e1_m,n1) := (fst c1, snd c1) in
      let c2 := mark_rec e2 n1 in
      let (e2_m,n2) := (fst c2, snd c2) in
```

```
      (Or e1_m e2_m, n2)
  | And e1 e2 =>
      let c1 := mark_rec e1 n in
      let (e1_m,n1) := (fst c1, snd c1) in
      let c2 := mark_rec e2 n1 in
      let (e2_m,n2) := (fst c2, snd c2) in
      (And e1_m e2_m, n2)
  | Star e1 =>
      let c1 := mark_rec e1 n in
      let (e1_m, n1) := (fst c1, snd c1) in (Star e1_m, n1)
  end.

Definition mark (exp : regexp) : regexp :=
  fst (mark_rec exp 0)
.


Inductive regexp_d : Set :=
  | DOne : regexp_d
  | DSymb : nat -> regexp_d
  | DOr : bool -> regexp_d -> regexp_d -> regexp_d
  | DAnd : bool -> regexp_d -> regexp_d -> regexp_d
  | DStar : regexp_d -> regexp_d
.

Definition delta (exp:regexp_d) : bool :=
  match exp with
  | DOne => true
  | DSymb _ => false
  | DOr b _ _ => b
  | DAnd b _ _ => b
  | DStar _ => true
  end
.

Fixpoint delta_init (exp:regexp) : regexp_d :=
  match exp with
  | One => DOne
  | Symb s => DSymb s
  | Or e1 e2 =>
      let de1 := delta_init e1 in
      let de2 := delta_init e2 in
```

```
      (DOr ((delta de1)||(delta de2)) de1 de2)
  | And e1 e2 =>
      let de1 := delta_init e1 in
      let de2 := delta_init e2 in
      (DAnd ((delta de1)&&(delta de2)) de1 de2)
  | Star e => DStar (delta_init e)
  end
.

Fixpoint first (exp:regexp_d) (l :list nat) {struct exp} : list nat :=
  match exp with
  | DOne => l
  | DSymb d => d::l
  | DOr _ e1 e2 =>
      let l2 := first e2 l in first e1 l2
  | DAnd _ e1 e2 =>
      let b1 := delta e1 in
      if b1
      then let l2 := first e2 l in first e1 l2
      else first e1 l
  | DStar e => first e l
  end
.

Fixpoint cont (exp : regexp_d) (l:list nat) (fl : list (nat * list nat) )
  {struct exp} : list (nat * list nat) :=
    match exp with
    | DOne => fl
    | DSymb d => (d,l)::fl
    | DOr _ e1 e2 => let fl2 := cont e2 l fl in cont e1 l fl2
    | DAnd _ e1 e2 =>
        let fl2 := cont e2 l fl in
        let l1 := if delta e2 then first e2 l else first e2 nil in
        cont e1 l1 fl2
    | DStar e => let l_res := first e l in cont2 e l_res fl
    end
  with cont2 (exp : regexp_d) (l:list nat) (fl : list (nat * list nat) )
  {struct exp} : list (nat * list nat) :=
    match exp with
    | DOne => fl
    | DSymb d => (d,l) :: fl
    | DOr _ e1 e2 => let fl2 := cont2 e2 l fl in cont2 e1 l fl2
```

```
    | DAnd _ e1 e2 =>
        let b1 := delta e1 in
        let b2 := delta e2 in
          if b1 (* l1 and l2 in l *)
          then if b2
            then (cont2 e1 l (cont2 e2 l fl) )
            else (cont e1 (first e2 nil) (cont2 e2 l fl))
          else if b2
            then (cont2 e1 (first e2 l) (cont e2 l fl))
            else (cont e1 (first e2 nil) (cont e2 l fl))
    | DStar e => cont2 e l fl
    end
.

Definition continuations (initial_state : nat)  (exp : regexp) :
 list (nat * (list nat)) :=
 let mexp := mark exp in
 let dexp := delta_init mexp in
 let fl_sets := cont dexp nil nil in
 let l := first dexp nil in
   cons (initial_state,l) fl_sets
.
```

B Lemmes nécessaires à la preuve du théorème `wf_continuations`

```
Require Import Le.
Require Import Omega.
Load "berry_sethi".

Inductive Not_in_list : nat -> list nat -> Prop :=
| Not_in_list_nil : forall n, Not_in_list n nil
| Not_in_list_cons :  forall n n0 l,
  Not_in_list n l -> n <> n0 ->
  Not_in_list n (n0::l).

Inductive Unique_list : nat -> list nat -> Prop :=
| Unique_list_eq : forall n l, Not_in_list n l -> Unique_list n (n::l)
| Unique_list_neq : forall n1 n2 l,
  n1 <> n2 -> Unique_list n1 l -> Unique_list n1 (n2::l)
.
```

```
Definition Wf_list (l:list nat) :=
(forall n, Not_in_list n l \/ Unique_list n l).

Inductive Wf_follow : list (nat * (list nat)) ->  Prop:=
| Wf_fol_nil : Wf_follow (nil)
| Wf_fol_cons : forall n l fl,
  Wf_list l ->  Wf_follow fl -> Wf_follow ((n,l)::fl).


Inductive Not_symb : nat -> regexp -> Prop :=
  | Not_symb_One : forall n, Not_symb n One
  | Not_symb_Symb : forall n m, n<>m -> Not_symb n (Symb m)
  | Not_symb_Or : forall n e1 e2,
    Not_symb n e1 -> Not_symb n e2 -> Not_symb n (Or e1 e2)
  | Not_symb_And : forall n e1 e2,
    Not_symb n e1 -> Not_symb n e2 -> Not_symb n (And e1 e2)
  | Not_symb_Star : forall n e, Not_symb n e -> Not_symb n (Star e)
.

Inductive Unique_symb : nat -> regexp -> Prop :=
| Unique_symb_Symb : forall n, Unique_symb n (Symb n)
| Unique_symb_Or_l : forall n e1 e2, Unique_symb n e1 -> Not_symb n e2 ->
                     Unique_symb n (Or e1 e2)
| Unique_symb_Or_r : forall n e1 e2, Not_symb n e1 -> Unique_symb n e2 ->
                      Unique_symb n (Or e1 e2)
| Unique_symb_And_l : forall n e1 e2, Unique_symb n e1 -> Not_symb n e2 ->
                      Unique_symb n (And e1 e2)
| Unique_symb_And_r : forall n e1 e2, Not_symb n e1 -> Unique_symb n e2 ->
                      Unique_symb n (And e1 e2)
| Unique_symb_Star : forall n e, Unique_symb n e -> Unique_symb n (Star e)
.

Definition lin_regexp (exp:regexp) : Prop :=
  forall n, Not_symb n exp \/ Unique_symb n exp.


Inductive Not_symb_d : nat -> regexp_d -> Prop :=
  | Not_symb_DOne : forall n, Not_symb_d n DOne
  | Not_symb_DSymb : forall n m, n<>m -> Not_symb_d n (DSymb m)
  | Not_symb_DOr : forall n b e1 e2,
    Not_symb_d n e1 -> Not_symb_d n e2 -> Not_symb_d n (DOr b e1 e2)
  | Not_symb_DAnd : forall n b e1 e2,
```

```
    Not_symb_d n e1 -> Not_symb_d n e2 -> Not_symb_d n (DAnd b e1 e2)
  | Not_symb_DStar : forall n e, Not_symb_d n e -> Not_symb_d n (DStar e)
.

Inductive Unique_symb_d : nat -> regexp_d -> Prop :=
| Unique_symb_DSymb : forall n, Unique_symb_d n (DSymb n)
| Unique_symb_DOr_l : forall n b e1 e2,
    Unique_symb_d n e1 -> Not_symb_d n e2 -> Unique_symb_d n (DOr b e1 e2)
| Unique_symb_DOr_r : forall n b e1 e2,
    Not_symb_d n e1 -> Unique_symb_d n e2 -> Unique_symb_d n (DOr b e1 e2)
| Unique_symb_DAnd_l : forall n b e1 e2,
    Unique_symb_d n e1 -> Not_symb_d n e2 -> Unique_symb_d n (DAnd b e1 e2)
| Unique_symb_DAnd_r : forall n b e1 e2,
    Not_symb_d n e1 -> Unique_symb_d n e2 -> Unique_symb_d n (DAnd b e1 e2)
| Unique_symb_DStar : forall n e,
    Unique_symb_d n e -> Unique_symb_d n (DStar e)
.

Definition lin_regexp_d (dexp:regexp_d) : Prop := forall n,
   Not_symb_d n dexp \/ Unique_symb_d n dexp.

Lemma le_mark_rec : forall exp m,
  le m (snd(mark_rec exp m)).

Lemma le_Unique_fst : forall n exp m,
   Unique_symb n (fst (mark_rec exp m)) ->
   le m n.

Lemma le_Unique_snd : forall n exp m,
   Unique_symb n (fst (mark_rec exp m)) ->
   le (S n) (snd (mark_rec exp m)).


Lemma correction_mark_rec : forall n exp m,
  Not_symb n (fst (mark_rec exp m)) \/ Unique_symb n (fst (mark_rec exp m)).


Lemma correction_mark : forall exp,
  lin_regexp (mark exp).

Lemma Not_conservative : forall n exp,
  Not_symb n exp -> Not_symb_d n (delta_init exp).
```

```
Lemma Unique_conservative : forall n exp,
  Unique_symb n exp -> Unique_symb_d n (delta_init exp).

Lemma dinit_conservation : forall exp,
  lin_regexp exp -> lin_regexp_d (delta_init exp).

Lemma correction_delta_mark : forall exp,
  lin_regexp_d (delta_init (mark exp)).

Lemma destr_lin_DOr : forall b e1 e2,
  lin_regexp_d (DOr b e1 e2) -> lin_regexp_d e1 /\ lin_regexp_d e2.

Lemma destr_lin_DAnd : forall b e1 e2,
  lin_regexp_d (DAnd b e1 e2) -> lin_regexp_d e1 /\ lin_regexp_d e2.

Lemma destr_lin_DStar : forall e,
  lin_regexp_d (DStar e) -> lin_regexp_d e.

Lemma Wf_list_nil: Wf_list nil.

Lemma exclusion_Symb : forall n dexp,
  Not_symb_d n dexp /\ Unique_symb_d n dexp -> False.

Lemma first_dec : forall dexp l,
  first dexp l = first dexp nil ++ l.

Lemma not_dec : forall n l1 l2,
  Not_in_list n (l1++l2) -> Not_in_list n l1 /\ Not_in_list n l2.

Lemma unique_dec : forall n l1 l2,
  Unique_list n (l1++l2) ->
  (Unique_list n l1 /\ Not_in_list n l2) \/
  (Unique_list n l2 /\ Not_in_list n l1).

Lemma not_first_dec : forall n dexp l,
  Not_in_list n (first dexp l) ->
  Not_in_list n (first dexp nil) /\ Not_in_list n l.

Lemma dec_Unique_first_aux : forall dexp l n,
  lin_regexp_d dexp ->
  Unique_list n (first dexp l) ->
```

```
  (Unique_list n (first dexp nil) /\ Not_in_list n l) \/
  (Unique_list n l /\ Not_in_list n (first dexp nil) ).


Lemma dec_Unique_first : forall dexp l n,
  lin_regexp_d dexp ->
  Unique_list n (first dexp l) ->
  Unique_list n (first dexp nil) \/ Unique_list n l.

Lemma incl_first : forall dexp n,
  lin_regexp_d dexp ->
  Unique_list n (first dexp nil) -> Unique_symb_d n dexp.

Lemma aux_first_DOr : forall b dexp1 dexp2 l,
  lin_regexp_d (DOr b dexp1 dexp2) ->
  Wf_list l ->
  (forall n, Unique_list n l -> Not_symb_d n (DOr b dexp1 dexp2)) ->
  (forall n , Unique_list n (first dexp2 l) -> Not_symb_d n dexp1).

Lemma aux_first_DAnd : forall b dexp1 dexp2 l,
  lin_regexp_d (DAnd b dexp1 dexp2) ->
  Wf_list l ->
  (forall n, Unique_list n l -> Not_symb_d n (DAnd b dexp1 dexp2)) ->
  (forall n , Unique_list n (first dexp2 l) -> Not_symb_d n dexp1).

Lemma wf_first : forall dexp l,
  lin_regexp_d dexp -> Wf_list l ->
  (forall n, Unique_list n l -> Not_symb_d n dexp)  ->
  Wf_list (first dexp l) .

Lemma hyp_DOr : forall b dexp1 dexp2 l,
  (forall n, Unique_list n l -> Not_symb_d n (DOr b dexp1 dexp2)) ->
  (forall n, Unique_list n l -> Not_symb_d n dexp1) /\
  (forall n, Unique_list n l -> Not_symb_d n dexp2).

Lemma hyp_DOr_2 : forall b dexp1 dexp2 l,
  lin_regexp_d (DOr b dexp1 dexp2) ->
  (forall n,
  Unique_list n l ->
  Unique_list n (first dexp1 (first dexp2 nil)) \/
  Not_symb_d n (DOr b dexp1 dexp2)) -> (
  (forall n, Unique_list n l -> Unique_list n (first dexp1 nil)  \/
```

```
  Not_symb_d n dexp1 ) /\
  (forall n, Unique_list n l -> Unique_list n (first dexp2 nil)  \/
  Not_symb_d n dexp2 ) ).

Lemma separation_lin :forall b dexp1 dexp2,
  (lin_regexp_d (DOr b dexp1 dexp2) \/ lin_regexp_d (DAnd b dexp1 dexp2)) ->
  (forall n : nat, Unique_list n (first dexp2 nil) -> Not_symb_d n dexp1).

Lemma separation_lin_And2 : forall b exp1 exp2 l,
  lin_regexp_d (DAnd b exp1 exp2) ->
  (forall n : nat, Unique_list n l -> Not_symb_d n (DAnd b exp1 exp2)) ->
  (forall n : nat, Unique_list n (first exp2 l) -> Not_symb_d n exp1).

Lemma destr_cont2_And : forall b dexp1 dexp2 l,
  lin_regexp_d (DAnd b dexp1 dexp2) ->
  (forall n, Unique_list n l ->
  (Unique_list n (first dexp1 (first dexp2 nil)) \/
   Not_symb_d n (DAnd b dexp1 dexp2))) -> (
  (forall n, Unique_list n l ->
    (Unique_list n (first dexp1 nil) \/ Not_symb_d n dexp1 )) /\
  (forall n, Unique_list n l ->
    (Unique_list n (first dexp2 nil) \/ Not_symb_d n dexp2 ))).

Lemma hyp1 : forall b dexp1 dexp2 l,
  lin_regexp_d (DAnd b dexp1 dexp2) ->
  (forall n , Unique_list n l ->
  Unique_list n (first dexp1 nil) \/ Not_symb_d n (DAnd b dexp1 dexp2)) ->
  (forall n : nat, Unique_list n l -> Not_symb_d n dexp2).

Lemma hyp2 : forall b dexp1 dexp2 l,
  lin_regexp_d (DAnd b dexp1 dexp2) ->
  (forall n , Unique_list n l ->
  Unique_list n (first dexp1 nil) \/ Not_symb_d n (DAnd b dexp1 dexp2))
  -> (forall n : nat, Unique_list n (first dexp2 l) ->
  Unique_list n (first dexp1 nil) \/ Not_symb_d n dexp1 ).

Lemma hyp3 : forall b dexp1 dexp2,
  lin_regexp_d (DAnd b dexp1 dexp2) ->
  (forall n, Unique_list n (first dexp2 nil) -> Not_symb_d n dexp1).

Lemma wf_cont : forall dexp l fl,
  lin_regexp_d dexp -> Wf_list l -> Wf_follow fl ->
```

```
  ((forall n , Unique_list n l -> Not_symb_d n dexp) ->
  Wf_follow (cont dexp l fl)) /\
  ((forall n, Unique_list n l ->
  (Unique_list n (first dexp nil)) \/ (Not_symb_d n dexp))->
   Wf_follow (cont2 dexp l fl)).

Theorem wf_continuations : forall exp n,
  Wf_follow (continuations n exp).
```

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399