Final Report for Deliverable Nr. 1.1

Finite Automata in Type Theory

Responsible: Thierry Coquand, `coquand@chalmers.se`
Written by: Vincent Siles, `siles@chalmers.se`
Site: University of Gothenburg, Sweden

Deliverable Date: February 2011
Actual Date: May 2011

Introduction

The goal of this project is to write a program for testing the equivalence of regular expressions formally proved correct in type theory. This is a natural attempt, since we are dealing with combinatorial structures that can be described with inductive definitions, and several teams are working on this [3, 2]. There exists furthermore an elegant algorithm, due to Brzozowski [4], and one would like to see how well this algorithm can be represented in type theory. The work [2] explores an interesting variant, due to Mirkin [5].

All these attempts of formalizing algorithms on regular expressions deal with the same fundamental issue related to the problem of representing in type theory general recursive algorithms. For instance, the construction of a finite deterministic automaton from a non deterministic one terminates by an argument involving a finite bound on the number of states. How to represent this naturally in type theory? We wanted to test a new definition of finite sets. This definition is interesting in itself, and is not equivalent constructively to the usual definition, which expresses that one can list all elements in this set. Intuitively this new definition expresses that if we keep removing elements from a finite set, this will stop eventually. We believe that this provides a natural encoding in type theory of algorithms relying on the computation of least fixed point of finite sets, such as the computation of minimal automata or the computation of a deterministic automaton associated to a non deterministic automaton.

1 Finite sets in type theory

In this section, we consider general sets (also called predicates) and binary relations in `Prop`. Given a binary relation R and an element x of type A, $R\ x$ is the set of the elements $\{y : A \mid R\ x\ y \text{ holds }\}$.

Given a type A, a set E and a binary relation R over A, we say that **Bar** $R\ E$[1] holds if

[1] This is a particular case of *Bar* induction, and we kept the name.

and only if for any x in E, **Bar** R $(E \cap (R\ x))$ holds. This is an inductive definition, which expresses intuitively that we can not find an infinite sequence $x_0,\ x_1,\ x_2, \ldots$ of elements satisfying E and such that we have $R\ x_i\ x_j$ if $i < j$.

```
Variable A:Type.
Definition gset := A → Prop.
Definition grel := A → A → Prop.

Variable R: grel A.

Inductive Bar (E: gset A ) : Prop :=
 | cBar : (∀ x:A, E x → Bar (Intersection E (R x))) → Bar E.
```

Figure 1: Definition of **Bar**.

This definition is closely connected to the notion of well-quasi-ordering. Indeed R is a well-quasi-ordering on E if and only if it is transitive and decidable and its complement R' is such that **Bar** R' E holds. If we start from a type A with a decidable equality $=_A$ then we define E to be (inductively) finite if **Bar** $(\lambda\ x.\lambda\ y.\neg\ (x =_A y))$ E. Intuitively, it expresses that for any infinite sequence $x_0,\ x_1,\ x_2, \ldots$ there exists $i < j$ such that $x_i =_A x_j$.

The usual definition of finite (or just "finite" from now) is that we can list all elements in E. It can be shown that E is inductively finite if it is finite, but the converse does not hold. Therefore, we capture more sets with this definition, but in general, it is not possible to describe an inductively finite set as the list of its elements.

Lemma 1.1 Basic properties of inductively finite sets

- *If* **Bar** R F *and* $E \subseteq F$ *then* **Bar** R E.
- *If* **Bar** R E *and* **Bar** R F *then* **Bar** R $(E \cup F)$.
- *If* **Bar** R E *and* **Bar** S E *then* **Bar** $(R \cup S)$ E.
- *If* **Bar** R E *and* **Bar** S F *then* **Bar** $(R \times S)$ $(E \times F)$,
  where $(E \times F)$ (x,y) *means* $E\ x \wedge F\ y$, *and* $(R \times S)$ (x_0,y_0) (x_1,y_1) *means* $R\ x_0\ x_1 \vee F\ y_0\ y_1$.

For any set E over a type A, a list of elements of E can be interpreted as a finite subset of E. Any list defines a subset of E by defining a membership function InA:

- InA $[\,]$ x is always false
- InA $(hd\ ::\ tl)$ x holds if and only if $x =_A hd$ or InA tl x.

```
Lemma Bar_gset_incl : ∀ (F:gset A) (R:grel A), Bar R F →
    ∀ (E:gset A) , Included E F → Bar R E.

Lemma Bar_gset_union: ∀ (E:gset A) (R:grel A), Bar R E →
    ∀ (F:gset A), Bar R F → Bar R (Union E F).

Lemma Bar_grel_union : ∀ (E:gset A) (S T :grel A),
    Bar S E → Bar T E → Bar (r_union S T) E.

Lemma Bar_gset_prod: ∀ (E:gset A) (F:gset B) (S:grel A) (T: grel B), Bar S E →
    Bar T F →    Bar (r_prod S T) (Prod E F).
```

Figure 2: Properties of **Bar**.

```
Definition gpred_list (E: gset A ) : gset (seq A) :=
 fix aux l : Prop :=
  match l with
  | nil ⇒ True
  | x :: xs ⇒ E x ∧ aux xs
end.
```

Figure 3: Definition of gpred_list.

If A is a type with decidable equality $=_A$, InA is a decidable predicate, so we can define a decidable equality $=_{[A]}$ on the type $[A]$ of lists such that $X_0 =_{[A]} X_1$ holds if and only if X_0 and X_1 represent the same subset of E. If X is of type $[A]$, we define gpred_list E X to mean that all elements in X satisfy the predicate E.

Proposition 1.2 *If E is finite on A then gpred_list E is finite on $[A]$.*

This can be expressed as the result that the collection of all subsets (given by lists) of an inductively finite set is inductively finite.

A main lemma for proving this proposition is the fact that if E is finite on A then the relation of strict inclusion (sub) between subsets of E enumerated by a list is well-founded. It follows from this that any monotone operator acting on subsets of E enumerated by a list, has a least fixed-point.

Proposition 1.3 *If f is a function preserving equality ($\forall x\ y, x =_A y \rightarrow fx =_B fy$), and if E is finite, than fE, the image of E by f, is also finite.*

```
Definition sub : grel (seq A) := r_inter nS (r_not nR).
Lemma sub_elim : ∀ (l l': seq A),
  sub l l' ↔ ((∀ (x:A), lnA l x → lnA l' x) ∧ exists2 y, ~~lnA l y & lnA l' y).
Lemma wf_sub : ∀ (E: gset A), E_compat E → Bar neq E → well_founded sub (gpred_list E).
Lemma Bar_gpred_list : ∀ (E: gset A), E_compat E → Bar neq E → Bar neql (gpred_list E).
```

Figure 4: Strict inclusion of lists and properties of gpred_list.

```
Variable f : A → B.
Definition f_set (E: gset A) : gset B := fun (y:B) ⇒ exists2 x, E x & eqB (f x) y.
Variable f_compat : ∀ (a a': A), eqA a a' → eqB (f a) (f a').
Lemma Bar_fun : ∀ (E: gset A) , Bar (neq eqA) E → Bar (neq eqB) (f_set E).
```

Figure 5: Definition of the property Bar_fun.

Both Bar_gpred_list and Bar_fun are the two most important lemmas for the Section 3.

2 Regular expressions

It is direct to represent the type of all regular expressions on a given alphabet Σ as an inductive type. We work with extended regular expressions, having conjunction and negation as constructors, and a "." constructor that matches any letter of the alphabet.

To each regular expression E we associate a boolean predicate (called pred in ssreflect) $L(E)$ on the set of words Σ^* such that a word u satisfies $L(E)$ if and only if u is recognized by E. In ssreflect, it is more convenient to define $L(E)$ in such a way that the proposition $L(E)\ u$ is decidable (see below), which allows us to directly write "u \in E" (which is a notation for mem E u) to state that the word u is recognized by E.

It is direct to define a boolean $\delta(E)$ (or has_eps in our formalization) which tests whether the empty word ϵ is in $L(E)$ or not.

Given a letter a in Σ and a regular expression E, we define E/a (or der a E), the derivative of E by a, by induction on E. A word u is in $L(E/a)$ if and only if the word au is in $L(E)$: $L(E/a)$ is called the *left-residual* of $L(E)$ by a. It is then possible to define E/u (or wder u E) for any word u by recursion on u

$$E/\epsilon = E \qquad E/(au) = (E/a)/u$$

The δ and derivation operators give us a decidable way to check whether a word is recognized by a regular expression. With the previous definitions, we can deduce that a word u

```
Variable symbol : osym_module.

Definition word := seq symbol.
Definition language := pred word.

Inductive regular_expression :=
 | Void
 | Eps
 | Dot
 | Atom of symbol
 | Star of regular_expression
 | Plus of regular_expression & regular_expression
 | And of regular_expression & regular_expression
 | Conc of regular_expression & regular_expression
 | Not of regular_expression.

Fixpoint has_eps (e: regular_expression) :=
  match e with
  | Void ⇒ false
  | Eps ⇒ true
  | Dot ⇒ false
  | Atom x ⇒ false
  | Star e1 ⇒ true
  | Plus e1 e2 ⇒ has_eps e1 || has_eps e2
  | And e1 e2 ⇒ has_eps e1 && has_eps e2
  | Conc e1 e2 ⇒ has_eps e1 && has_eps e2
  | Not e1 ⇒ negb (has_eps e1)
  end.
```

Figure 6: Definition of regular expressions and the δ operator.

is in $L(E)$ if and only if ϵ is in $L(E/u)$, which is equivalent to $\delta(E/u) = true$.

Brzozowski proved in [4] that there is only a finite number of derivatives for a regular expression, up to the following rewriting rules:

$$E + E \sim E \qquad E + F \sim F + E \qquad E + (F + G) \sim (E + F) + G$$

In order to have a more efficient algorithm, we extended theses rules with the following ones:

```
Fixpoint der (x: symbol) (e: regular_expression) :=
  match e with
  | Void ⇒ Void
  | Eps ⇒ Void
  | Dot ⇒ Eps
  | Atom y ⇒ if x == y then Eps else Void
  | Star e1 ⇒ Conc (der x e1) (Star e1)
  | Plus e1 e2 ⇒ Plus (der x e1) (der x e2)
  | And e1 e2 ⇒ And (der x e1) (der x e2)
  | Conc e1 e2 ⇒ Plus (Conc (der x e1) e2)
    (if has_eps e1 then der x e2 else Void )
  | Not e1 ⇒ Not (der x e1)
  end.

Fixpoint wder (u: symbol) (e: regular_expression) :=
  if u is x :: v then wder v (der x e) else e.

Fixpoint mem_der (e: regular_expression) (u: symbol) :=
  if u is x :: v then mem_der (der x e) v else has_eps e.

Lemma mem_derE : ∀ (u: symbol) (e: regular_expression), mem_der E u = (u \in E).

Lemma mem_wder : ∀ (u: symbol) (e: regular_expression),
  mem_der E u = has_eps (wder u E).
```

Figure 7: Definition of the der operator and some of its properties.

$A + \emptyset \sim A$	$A \,\&\, \emptyset \sim \emptyset$	$A \,\&\, \top \sim A$
$A + A \sim A$	$A \,\&\, A \sim A$	$A \,\&\, B \sim B \,\&\, A$
$(A.B).C \sim A.(B.C)$	$A \,\&\, (B \,\&\, C) \sim (A \,\&\, B) \,\&\, C$	$\epsilon.A \sim A.\epsilon \sim A$
$A^{**} \sim A^*$	$\emptyset^* \sim \epsilon^* \sim \epsilon$	$\neg\neg A \sim A$

where $\top$ stands for the regular expression that recognize any word, which we implemented as $\neg\emptyset$. This gives us a decidable equality over regular expressions, called similarity, which also satisfies $L(E) = L(F)$ if $E \sim F$.

For defining $E \sim F$ we introduce a notion of canonical form (with a boolean predicate wf_re for being "a well-formed canonical expression") and a normalization function canonize in such a way that $E \sim F$ is defined as canonize E = canonize F (where = is the structural equality). This function relies on the use of smart-constructors which perform the previous rewriting rules. For example, the rewriting rules of Plus are enforced by keeping a strictly sorted lists of all the regular expressions linked by a "+". We then prove that these smart-

constructors indeed satisfy the similarity requirements (see Figure 8).

```
Fixpoint canonize c : canonical_regexp := match c with
 | Void ⇒ CVoid
 | Eps ⇒ CEps
 | Dot ⇒ CDot
 | Atom n ⇒ CAtom n
 | Star c' ⇒ mkStar (canonize c')
 | Plus c1 c2 ⇒ mkPlus (canonize c1) (canonize c2)
 | And c1 c2 ⇒ mkAnd (canonize c1) (canonize c2)
 | Conc c1 c2 ⇒ mkConc (canonize c1) (canonize c2)
 | Not c1 ⇒ mkNot (canonize c1)
end.

Lemma mkPlusC : ∀ r1 r2, mkPlus r1 r2 = mkPlus r2 r1.

Lemma mkPlus_id : ∀ r, wf_re r → mkPlus r r = r.

Lemma mkPlusA : ∀ r1 r2 r3, wf_re r1 → wf_re r2 → wf_re r3 →
  mkPlus (mkPlus r1 r2) r3 = mkPlus r1 (mkPlus r2 r3).
```

Figure 8: canonize function and properties of the Plus smart constructor.

3 Brzozowski main result

The key property of Brzozowski we need to build the decision procedure is the fact that the set of derivatives of a regular expression is finite (with respect to similarity), if the alphabet Σ is finite. The proof uses the following equalities

$$(E+F)/u = E/u + F/u \qquad (E \,\&\, F)/u = E/u \,\&\, F/u \qquad \neg(E/u) = (\neg E)/u$$

which are clear, and, if $u = a_1 \ldots a_n$

$$(EF)/u = (E/u)F + \delta(E/a_1 \ldots a_{n-1})F/a_n + \delta(E/a_1 \ldots a_{n-2})F/a_{n-1}a_n + \cdots + \delta(E)F/a_1 \ldots a_n$$

and finally

$$E^*/u \sim (E/u)E^* + \Sigma\ \delta(E/u_1) \ldots \delta(E/u_{p-1})(E/u_p)E^*$$

for any decomposition of u in non-empty words $u = u_1 \ldots u_p$.

To simplify we assume that Σ is now the type with two elements $0, 1$, but it would work with any finite set. The particular instance of **regular_expression** over this alphabet is named regexp. We represent the set of all derivatives of a given regular expression E by the predicate

$$\mathcal{D}er\ E = \{F \mid \exists u : \Sigma^*.\ E/u \sim F\}$$

We proved and formalized that this set was also *inductively* finite with respect to similarity. The proof is a combination of the lemmas Bar_gpred_list and Bar_fun described in Section 1.

Lemma 3.1 The set of derivatives is inductively finite

For any regular expression E, the set $\mathcal{D}er\ E$ is inductively finite with respect to the similarity:

$$\forall(E : \mathit{regexp}), \mathbf{Bar}\ (\lambda x \lambda y. \neg(x \sim y))\ (\mathcal{D}er\ E)$$

As we said, it is not possible in the general case to enumerate any inductively finite set with a list, but in this particular case, it was made possible:

Lemma 3.2 Enumeration of the set of derivatives

For any regular expression E, it is possible to enumerate the set $\mathcal{D}er\ E$ with a list with the following properties:

- $\forall(E : \mathit{regexp})\exists(L_E : \mathbf{seq}\ \mathit{regexp}), L_E \subseteq \mathcal{D}er\ E$
- $E\ \in\ L_E$
- $\forall(e : \mathit{regexp}), (e\ \in\ L_E) \rightarrow \exists(e' : \mathit{regexp}), (e'\ \in\ L_E) \wedge (e' \sim e/0)$
- $\forall(e : \mathit{regexp}), (e\ \in\ L_E) \rightarrow \exists(e' : \mathit{regexp}), (e'\ \in\ L_E) \wedge (e' \sim e/1)$

To build such a list, let us consider the following function that maps der on lists of regexp:

$$\mathsf{deriv}\ l = l +\!\!+ \mathsf{map}(\mathsf{der}\ 0)\ l +\!\!+ \mathsf{map}(\mathsf{der}\ 1)\ l$$

We already explained how we can consider lists as finite subsets. Let us consider such subsets of $\mathcal{D}er\ E$. As for the Bar_gpred_list lemma, we use here the fact that the relation of strict inclusion sub on these subsets is well-founded (because $\mathcal{D}er\ E$ is inductively finite) to be able to compute the fixpoint of this deriv function.

As a consequence of these properties, we can show that any derivative of E is represented inside L_E:

Corollary 3.3 the list L_E is complete

For any regular expression E and any word u, there is a regular expression e in L_E such that $L(E/u) = L(e)$.

The proof goes by induction on the length of the word u:

- If the length of u is 0, then $u = \epsilon$, and we have $E/\epsilon = E$. We can close this case since E is in L_E.
- If the length of u is $n+1$, then $u = vi$ where its last letter i is either 0 or 1. By induction, there is e in L_E such that $L(e) = (E/v)$. Using the two last properties of L_E as described in the previous lemma, there is e' in L_E such that $e' \sim e/i$, which implies $L(e') = L(e/i)$. If we combine both conclusions, we get that $L(E/u) = L((E/v)/i) = L(e/i) = L(e')$, which ends this proof.

4 Description of the decision procedure

We consider that two languages are equal if they contains the same words:

$$\forall L_1\ L_2,\ L_1 = L_2 \quad \leftrightarrow \quad \forall u \in \Sigma^*, u \in L_1 = u \in L_2$$

From this definition and the properties of regular expressions, we can assert that:

$$\begin{aligned} \forall E\ F,\ L(E) = L(F) \quad &\leftrightarrow \quad \forall u \in \Sigma^*, u \in L(E) = u \in L(F) \\ &\leftrightarrow \quad \forall u \in \Sigma^*, \delta(E/u) = \delta(F/u) \end{aligned}$$

```
Definition EQUIV (E F:regexp) := ∀ s:bword, (s \in E ) = (s \in F).

Definition all_delta (ef:regexp × regexp) := let (e,f) := ef in has_eps e == has_eps f.

Definition build_list_fun : regexp → regexp → seq (regexp× regexp).

Definition regexp_eq (r1 r2: regexp) : bool := (all all_delta (build_list_fun r1 r2)).

Lemma regexp_eqP : ∀ (r1 r2:regexp), reflect (EQUIV r1 r2) (regexp_eq r1 r2).
```

Figure 9: Decision procedure with its correctness proof

For any regular expressions E and F, we consider the set

$$\mathcal{D}er_2\ E\ F = \{(e, f) \mid \exists (u : word), e \sim E/u \wedge f \sim F/u\}$$

This set is included inside $\mathcal{D}er\ E \times \mathcal{D}er\ F$, so with Lemma 1.1 and Lemma 3.1, we can conclude that $\mathcal{D}er_2\ E\ F$ is inductively finite for any $E\ F$. A similar approach to the proof Lemma 3.2 and Corollary 3.3 allows us to conclude that $\mathcal{D}er_2\ E\ F$ can be enumerated by a list L_{EF} and for all word u in Σ^*, there is (e, f) in L_{EF} such that $L(e) = L(E/u)$ and $L(f) = L(F/u)$.

There is however something subtle about the specification of the algorithm: to deduce that for any word u there exists (e, f) in L_{EF} such that $E/u \sim e$ and $F/u \sim f$, we need a priori a stronger condition on $\sim$, that we have $A/0 \sim B/0$ and $A/1 \sim B/1$ whenever $A \sim B$.

Fortunately, the property of L_{EF} is enough to decide the equivalence: we know that $L(E) = L(F) \quad \leftrightarrow \quad \forall u \in \Sigma^*, \delta(E/u) = \delta(F/u)$ and we proved that, for any u in Σ^*, there is (e, f) in L_{EF} such that $L(e) = L(E/u)$ and $L(f) = L(F/u)$. Since $\delta(e) = \delta(E/u)$ and $\delta(f) = \delta(F/u)$, we can show that

$$L(E) = L(F) \quad \leftrightarrow \quad \forall (e, f) \in L_{EF}, \delta(e) = \delta(f)$$

which is a decidable predicate.

We have successfully formalized every step of this decision procedure along with its correctness proof using Coq and ssreflect [1]. We slightly improved this algorithm when two regular expressions are not equivalent by providing a word which is recognized by only one of the two expressions. As a direct extension of this procedure, we also defined and proved correct a decidable procedure to test the inclusion of languages recognized by regular expressions.

References

[1] T. Coquand, G. Gonthier, V. Siles. Source code of the formalization http://www.cse.chalmers.se/ siles/coq/regexp.tar.bzip2

[2] J.B. Almeida, N. Moreira, D. Pereira, S.M. de Sousa. Partial Derivative Automata Formalized in Coq.

[3] T. Braibant and D. Pous. A tactic for deciding Kleene algebras. First Coq Workshop, August 2009.

[4] J.A. Brzozowski. Derivatives of regular expressions. JACM 11 (4), 1964, 481-494.

[5] B.G. Mirkin. An algorithm for constructing a base in a language of regular expressions. *Engineering Cybernetics*, 5:51-57, 1996.

[6] F. Richman and G. Stolzenberg. Well-Quasi-Ordered sets. *Advances in Mathematics*, 97:145-153, 1993.