

Case analysis on a dependently typed fixed point for regular languages

Vladimir Komendantsky
YesLogic Pty Ltd

6th July 2009

Abstract

We present a dependently typed fixed-point function for the proof of equivalence of NFAs and regular languages. The fixed-point function has a dependent sum return type and is used for simple case analysis on words, i.e., lists of alphabetic symbols. It provides an efficient and relatively easy method for reasoning by cases about decidability of regular languages in Coq. In particular, use of dependent types allows to reason directly by structural induction without employing arithmetical methods such as wellfounded induction.

As an illustration of method application, we sketch a proof in Coq of the statement that if a language l is recognised by an NFA M then its Kleene closure l^* is recognised by the Kleene closure M^* of M .

1 Introduction

In this extended abstract, it is described a practical proof method that can be used as alternative to wellfounded induction in certain cases where a wellfounded proof would be cumbersome. The method is employed on a particular step of a proof of equivalence of regular languages and languages recognised by non-deterministic finite automata [5]. On this step one is required to prove that if the accepted languages of an automaton M and the *Kleene closure of the automaton M* accepts w , then w is contained in the *Kleene closure of the language l* . The Kleene closure of a language is defined in §2 in a standard way as the reflexive transitive closure using a power language fixed point function. There is a choice of definitions for the Kleene closure of an automaton, which could be a Thompson automaton or a Glushkov automaton [9]. It is not essential which of those is taken because the proof has the same shape in either case. The actual Coq proof is going to be maintained at [4].

Although the subject of formal languages is well exploited, the requirements towards a machine-checkable proof, such as the one presented here, often include constructions not found anywhere on paper, which eventually needs to be dealt with in a proof assistant. Popular paper proofs may omit the transition from closure automata to closure languages [6]. Some proofs outline a wellfounded construction by induction on the length of the given word [7]. The wellfounded construction is visually appealing and arises naturally as an algorithmic procedure, as we illustrate below in Algorithm 1. However, attempts to code this procedure in a proof assistant face technical difficulties related to tail call handling.

To get around technicalities of the wellfounded procedure, it can be reconstructed as a structural fixed point construction, as in Algorithm 2. The natural proof method for dealing with this construction is Coq's native structural induction [8, 2].

Input:

- an automaton M
- a word w
- a proof of type $M^* w$ that w is accepted by the closure automaton M^* of M

Output: a non-empty word w_0 accepted by M and a word w_1 accepted by M^*

```

1  $w_0 \leftarrow nil$ 
2  $w_1 \leftarrow w$ 
3 repeat
4   if  $w_1 = a :: w'_1$  then
5      $w_0 \leftarrow w_0 ++ a :: nil$ 
6      $w_1 \leftarrow w'_1$ 
7   fi
8 until  $M$  accepts  $w_0$  and  $M^*$  accepts  $w_1$ 

```

Algorithm 1: Computational behaviour of a wellfounded proof of $M^* w \rightarrow l^* w$

The termination condition in Algorithm 1 requires to run it again with $w_0 \leftarrow nil$ and $w_1 \leftarrow w_1$. In fact, this is not a proper termination condition for structural recursion.

```

1 compute  $F nil w$ 
2 function  $F (w_0 : word)(w_1 : word) : (word, word, nat) \{$ 
3   if  $w_1 = a :: w'_1$  then
4     if  $M w_0$  and  $M^* w_1$  then
5        $n \leftarrow (3rd \text{ projection of } F (a :: nil) w'_1) + 1$ 
6     else
7        $(w_0, w_1, n) \leftarrow F (w_0 ++ a :: nil) w'_1$ 
8     fi
9   else
10    if  $M w_0$  then
11       $n \leftarrow 1$ 
12    else
13      anomaly
14    fi
15  fi
16  return  $(w_0, w_1, n)$ 
17  $\}$ 

```

Algorithm 2: Computational behaviour of a structural proof of $M^* w \rightarrow l^* w$

The logical information has been deliberately left out to highlight the computationally relevant steps. The logical information can be merged in as follows:

1. Instead of terminating the computation with an anomaly, one can organise a specialised branch of the computation for the case when $w_1 = nil$ and not $M w_0$.
2. One should guarantee the equation $w = w_0 ++ w_1$ in order to establish language containment $l^{n+1} w$ given $l w_0$ and $l^n w_1$.
3. Notably, the language l is not used in the algorithm. This is due to the fact that the computation proceeds by case analysis on decidability of the acceptability predicates on M and M^* , for which l is not required. In fact, languages are only logically relevant. Since one has the inductive hypothesis about the equivalence between l and the language accepted by M , one can

logically derive $I^{(n+1)}(w_0::w_1)$ from the proof that w_0 is accepted by M and w_1 is accepted by n iterations of M .

2 Language definitions

Some basic language definitions are given below. Note that the type of languages is computationally relevant while being a Prop-valued predicate. This allows to get the best of both computational and logical worlds.

Definition word (A:Type) := list A.

Definition language (A:Type) : Type := (word A) -> Prop.

Definition language_eq (A:Type) (l1 l2:language A) : Prop :=
forall (w:word A), l1 w <-> l2 w.

Next, we define the Kleene closure of a language (with implicit arguments omitted):

Variable (Symbol:Type).

Definition unit_language : language Symbol := fun (w0:word Symbol) => w0 = nil.

Inductive language_cat : (language Symbol) -> (language Symbol) -> (language Symbol) :=
| Language_cat : forall (l1 l2:language Symbol) (w1 w2:word Symbol),
l1 w1 -> l2 w2 -> language_cat l1 l2 (w1++w2).

Fixpoint language_power (l:language Symbol) (n:nat) : (language Symbol) :=
match n with
| 0 => unit_language
| S p => language_cat l (language_power l p)
end.

Inductive language_closure : (language Symbol) -> (language Symbol) :=
| Language_closure : forall (l:language Symbol) (n:nat) (w:word Symbol),
language_power l n w -> language_closure l w.

3 Automaton definitions

Inductive nf_automaton : Prop :=

Nf_automaton :
In init_state states ->
subset acc_states states ->
subset transition (list_prod (list_prod states symbols) states) ->
nf_automaton.

Inductive nf_path : State -> State -> language Symbol :=
| Nf_path_refl : forall s, In s states -> nf_path s s nil
| Nf_path_trans : forall s1 s2 w, nf_path s1 s2 w ->
forall a, In a symbols ->
forall s, In s states ->
In (s, a, s1) transition ->
nf_path s s2 (cons a w).

Inductive nf_accepts : language Symbol :=

Nf_accepts : forall w acc, In acc acc_states -> nf_path init_state acc w ->
nf_accepts w.

Inductive nf_regular_language : language Symbol -> Type :=

```
Nf_regular_language : forall l, nf_automaton ->
  language_eq l nf_accepts -> nf_regular_language l.
```

4 The fixed point function

We accept Thompson automata with empty transitions (labelled with `empty_symbol`) as implementations of closure automata. This means that we use an artificial initial state `init_closure` connected to the given automaton by means of a transition (`init_closure`, `empty_symbol`, `init`) and a set of transitions from the set of accepting states of the given automaton to `init_closure` labelled with `empty_symbol`.

Now we can specify a fixed-point function that implements Algorithm 2 while merging in logically relevant information, as in points 1–3 in §1. See [4] for the full implementation including the function body which does not fit in here.

```
Fixpoint nf_accepts_rec
  w0 w1 symbols states (init init_closure:State)
  (acc:list State) (tr tr_loop:list (State*Symbol*State)) (l:language Symbol)
  (Hinit:In init states) (Hacc:subset acc states)
  (Hinit_closure: ~In init_closure states)
  (Htr:subset tr (list_prod (list_prod states symbols) states))
  (Hloop:set_eq tr_loop
    (list_prod (list_prod acc (empty_symbol :: nil)) (init_closure :: nil)))
  (Hl:language_eq (A:=Symbol) l (nf_accepts symbols states init acc tr)) :
  {whd : word Symbol &
   {wtl : word Symbol &
    {n : nat &
     l whd /\
     language_power l n wtl /\
     whd++wtl=w0++w1}}}} +
  {w : word Symbol &
   ~nf_accepts symbols (init_closure :: states) init_closure
     (init_closure :: nil)
     ((init_closure, empty_symbol, init) :: tr ++ tr_loop) w /\
   w=w0++w1} := ...
```

To implement the case analysis in this function one needs a decidability principle for automata, that is a constructive disjunction $\forall w, (Mw) + (\neg Mw)$. It is defined in Coq as follows:

```
Lemma nf_accepts_dec : forall w symbols states
  (init:State) (acc:list State) (tr:list (State*Symbol*State)),
  In init states ->
  subset acc states ->
  subset tr (list_prod (list_prod states symbols) states) ->
  {nf_accepts symbols states init acc tr w} +
  {~nf_accepts symbols states init acc tr w}.
```

and the corresponding principle for closure automata, with a modified context, is defined as follows:

```

Lemma nf_accepts_closure_dec : forall w symbols states
  (init init_closure:State) (acc:list State) (tr tr_loop:list (State*Symbol*State)),
  In init states ->
  subset acc states ->
  ~In init_closure states ->
  subset tr (list_prod (list_prod states symbols) states) ->
  set_eq tr_loop
    (list_prod (list_prod acc (empty_symbol :: nil)) (init_closure :: nil)) ->
  {nf_accepts symbols (init_closure :: states) init_closure (init_closure::nil)
    ((init_closure, empty_symbol, init) :: tr ++ tr_loop)
  w} +
  {~nf_accepts symbols (init_closure :: states) init_closure (init_closure::nil)
    ((init_closure, empty_symbol, init) :: tr ++ tr_loop)
  w}.

```

5 Proof by case analysis

The goal has the following premises (leaving out implicit ones):

```

Hinit_closure : ~ In init_closure states
Hloop : set_eq tr_loop
  (list_prod (list_prod acc (empty_symbol :: nil))
  (init_closure :: nil))
Hrlang : language_eq (A:=Symbol) 1 (nf_accepts symbols states init acc tr)
Hinit : In init states
Hacc : subset acc states
Htr : subset tr (list_prod (list_prod states symbols) states)
w : word Symbol
Hacc_w : nf_accepts symbols (init_closure :: states) init_closure
  (init_closure :: nil)
  ((init_closure, empty_symbol, init) :: tr ++ tr_loop) w

```

The **goal formula** that we are going to prove is `language_closure (A:=Symbol) 1 w`.
 For the proof, analyse cases of the fixed point computation of `nf_accepts_rec`:

```

case (nf_accepts_rec nil w Hinit Hacc Hinit_closure Htr Hloop Hrlang).

```

The first case is given by the constructive existence premise

```

{whd : word Symbol &
 {wt1 : word Symbol &
 {n : nat &
 1 whd /\ language_power (A:=Symbol) 1 n wt1 /\ whd ++ wt1 = nil ++ w}}}

```

which is deconstructed as follows:

```

intros [w0 [w1 [n [Hw0 [Hw1 Hw_eq]]]]].

```

and respective constructors and assumptions are applied:

```

apply Language_closure with (n:=S n).
simpl in Hw_eq; rewrite <- Hw_eq.
apply Language_cat; assumption.

```

which completes the first case. The second case leads to contradiction with the hypothesis `Hacc_w`, which can be viewed as the logical analogue of the computation anomaly in Algorithm 2:

```
intros [w0 [Hnotacc Hw_eq]].
simpl in Hw_eq; rewrite Hw_eq in Hnotacc; contradiction.
```

which completes the second case. Since we deconstructed a result of a fixed point computation, in fact, no inductive hypothesis is needed.

6 Conclusions and further directions

In §4 we specified a fixed point function with a dependent sum type that we used to prove a technical step in §5 required for the proof that Kleene closure languages are regular, for which the complete proof is maintained at [4]. The use of the dependent return type guarantees transparency of the proof. The fact that the result is computed in a fixed point makes a simple case analysis sufficient for the proof, which contrasts with wellfounded proofs where a hypothesis with a premise of the kind “for all natural numbers m such that $m < n$ it holds that Pm ” is required to prove Pn (along with the whole library of arithmetic).

The presented work has some applications in program specification and program extraction [5]. We are especially interested in constructing partial derivative NFAs [1] and extending this work from regular languages to regular languages with additional features such as backreferences [3].

References

- [1] V. M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.
- [2] B. Barras and B. Werner. Coq in Coq. <http://pauillac.inria.fr/~barras/coqincoq.ps.gz>.
- [3] C. Campeanu, K. Salomaa, and S. Yu. A formal study of practical regular expressions. *International Journal of Foundations of Computer Science*, 14:1007 – 1018, 2003.
- [4] M. Day and V. Komendantsky. Coq proof developments of regular expressions, their partial derivatives, and NFAs. <http://similare.org/regexp-nfa.tgz>.
- [5] M. Day and V. Komendantsky. A compiler of regular expressions to NFAs. <http://similare.org/regexp-nfa.pdf>, May 2009. Submitted.
- [6] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2nd edition, 2000.
- [7] D. Kozen. *Automata and Computability*. Undergraduate Texts in Computer Science. Springer, 1997.
- [8] B. Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris 7, 1994.
- [9] S. Yu. Regular languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of formal languages*, volume 1: Word, language, grammar, pages 41–110. Springer-Verlag, 1997.