Verified Lexical Analysis

Tobias Nipkow

Technische Universität München
Institut für Informatik, 80290 München, Germany
`http://www.in.tum.de/~nipkow/`

Abstract. This paper presents the development and verification of a (very simple) lexical analyzer generator that takes a regular expression and yields a functional lexical analyzer. The emphasis is on simplicity and executability. The work was carried out with the help of the theorem prover Isabelle/HOL.

1 Introduction

Admittedly, lexical analysis is not exactly safety critical. But if the dream of a verified compiler is to be taken seriously, it must include the front end as well. Practical applications aside, lexical analysis is an excellent example of computational discrete mathematics, and as such an ideal test case for any aspiring theorem prover.

We formalize and verify the process of taking a regular expression and turning it into a lexical analyzer (also called *scanner*). The design goals are simplicity and executability. The result is an almost executable functional program, except for one place, where simplicity has prevailed over executability. The overall structure of both the verified theories and the main sections of the paper is shown in Fig. 1.

The vertical arrows describe the well-known translation of a regular expression into a deterministic automaton. This is the subject of §3–4. We follow the standard textbook treatment but rely on functions to represent automata.

The horizontal arrows describe the actual scanner. Roughly speaking, a scanner converts a string into a list of 'tokens'. We have simplified the model by replacing the tokens by the substrings themselves. In addition, the scanner returns the unrecognized suffix of the input. Thus function `scan` takes a string w and returns a pair of

- a list $[u_1, \ldots, u_n]$ that is obtained by repeatedly chopping off the remaining input the maximal nonempty prefix u_i that is recognized by A,
- and the remaining unrecognized suffix v.

In particular this means the concatenation $u_1 \ldots u_n v$ yields the input w. Although this scanning process is not given much attention in the literature, a precise specification and a verified implementation of `scan` turns out to be very interesting and is the subject of §5. All theories are available online at `http://www.in.tum.de/~isabelle/library/HOL/Lex/`.

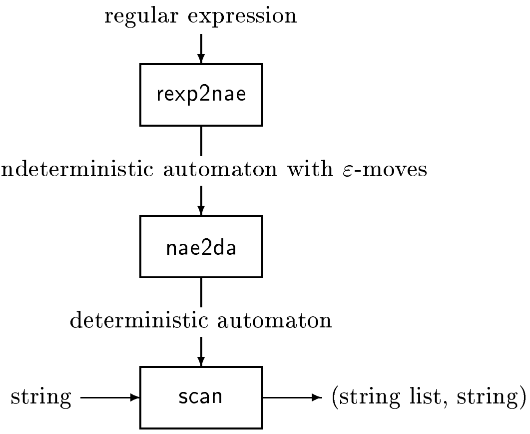

Fig. 1. The structure

We assume that the reader is familiar with the standard theory of finite automata and regular expressions as described, for example, in the textbook by Hopcroft and Ullman [6].

1.1 Notation

Although we have talked about 'strings' above, there is no need for a new datatype: strings are simply lists, and we don't even need to fix the alphabet. In the sequel, type variable α always represents this alphabet and we use 'string' as a synonym for 'list over type α'.

A few words about notation in Isabelle/HOL (abbreviated to HOL below). List notation is similar to ML (e.g. `@` is 'append' and concat distributes `@` over a list of lists) except that the 'cons' operation is denoted by `#` instead of `::`. There is also the function set that returns the set of elements of a list. Set comprehension syntax is $\{e \ . \ P\}$. Function types are denoted by $\Rightarrow$.

Thanks to Markus Wenzel, Isabelle has recently acquired *long identifiers* of the form $T.n$ where T is the name of a theory and n a name defined in T.

To distinguish variables from constants, the latter are shown in sans-serif.

2 Automata

All our automata will be triples of a start state, a next state function and a test for final states. We define three corresponding projections start, next and fin:

```
start(q,d,f) ≡ q        next(q,d,f) ≡ d        fin(q,d,f) ≡ f
```

Our formalization differs from standard automata theory in the following aspects:

- there are no finiteness assumptions;
- neither the alphabet nor the set of states is a component of the automaton; both are implicit in the *type* of the components.

2.1 Deterministic automata

Theory `DA` defines the parameterized type

```
(α,σ)da = σ × (α ⇒ σ ⇒ σ) × (σ ⇒ bool)
```

of deterministic automata, where σ is the type of states. The only painful choice is the order of arguments of the transition function: $\sigma \Rightarrow \alpha \Rightarrow \sigma$ or $\alpha \Rightarrow \sigma \Rightarrow \sigma$? Both appear in the literature and have their minor advantages and disadvantages. I prefer the state transformer view. Final states are encoded via a test function rather than a set of states to allow direct execution.

The extension of `next` to strings is called `delta`:[1]

```
delta :: (α,σ)da ⇒ α list ⇒ σ ⇒ σ
delta A []     s = s
delta A (a#w) s = delta A w (next A a s)
```

A word is accepted by a `da` if `delta` maps the start state to a final state:

```
accepts ::  (α,σ)da ⇒ α list ⇒ bool
accepts A w ≡ fin A (delta A w (start A))
```

2.2 Nondeterministic automata

Nondeterministic automata come in two flavours, with and without ε-moves. The latter are defined by the type

```
(α,σ)na = σ × (α ⇒ σ ⇒ σ set) × (σ ⇒ bool)
```

and merely serve as the stepping stone towards the former. Adjoining a new element ε to the alphabet is naturally modeled by the standard datatype

```
(α)option = None | Some α
```

where `None` represents ε. By this device a nondeterministic automaton with ε-moves over alphabet α is simply a nondeterministic automaton without ε-moves over alphabet (α)`option`:

```
(α,σ)nae = (α option,σ)na
```

That was easy. The only choice we had was whether to model the transition function as a set-valued function (as we did) or as a relation. The argument in favour of a set-valued function is purely computational: provided the set of next states of every state is finite, it can be represented by a list, and hence the transition function is computable. Using a relation, it is unclear in what sense the set of next states is computable.

---

[1] With a different order of arguments we could have defined `delta A ≡ foldl (next A)`.

Although relations are not so nice for computing, they are handy for reasoning. Hence we define `step`, the relational version of `next`:

```
step :: (α,σ)nae ⇒ α option ⇒ (σ × σ)set
step A a ≡ {(p,q) . q ∈ next A a p}
```

The term `eps A` is short for `step A None` and denotes all ε-moves.

Before we can continue, we need two operations from the standard theory of relations: `r^*` is the reflexive transitive closure of `r` and `s ⊙ r` is the composition of `r` and `s` (mind the order!):

```
s ⊙ r ≡ {(x,z). ∃ y. (x,y) ∈ r ∧ (y,z) ∈ s}
```

The extension of `step` to lists is straightforward:[2]

```
steps :: (α,σ)nae ⇒ α list ⇒ (σ × σ)set
steps A []    = (eps A)^*
steps A (a#w) = steps A w  ⊙  step A (Some a)  ⊙  (eps A)^*
```

The term `(eps A)^*` is the so-called *epsilon closure* of an `nae A` that relates state `s` to state `t` iff `t` is reachable from `s` by a finite sequence of ε-moves.

The words accepted by an `nae` are defined as usual:

```
accepts :: (α,σ)nae ⇒ α list ⇒ bool
accepts A w ≡ ∃ q. (start A,q) ∈ steps A w ∧ fin A q
```

Note that `step`, `steps` and `accepts` are used only in proofs. Hence their non-executability is of no concern.

All the definitions in this subsection reside in theory `NAe`. Thus we can distinguish, for example, `DA.accepts` and `NAe.accepts`.

2.3 Discussion of nondeterministic automata

Apart from the fact that transition functions are arbitrary functions and hence automata need not be finite, the above treatment of nondeterministic automata is standard. However, it was not until after a number of painful iterations that I arrived at this formulation. There are three different options when dealing with the extension of the next state function to words, which behave quite differently in proofs:

1. The standard one is of type `(α,σ)nae ⇒ (α)list ⇒ σ ⇒ (σ)set`. This is how we started, but it leads to proofs with a lot of duplication because of the asymmetry between input (single states) and output (sets of states).
2. A much slicker version is defined directly on sets of states, i.e. it is of type `(α,σ)nae ⇒ (α)list ⇒ (σ)set ⇒ (σ)set`. This eliminates the asymmetry of the first version and results in some compact algebraic laws like

   ```
   delta A (u@v) = delta A v o delta A u
   ```

   Unfortunately it also leads to very complicated arguments in those cases where only single states are involved, e.g. the start state.

---

[2] I have used `delta` for functions and `steps` for relations.

3. steps is an excellent compromise because it it only talks about individual states in the input and output, and it is close to our intuition. On the other hand, there are also some drawbacks that we discuss in §4.

The touchstone for these different formulations was the correctness proof of the translation of a regular expression into an `nae` (see §4). Our conclusion is corroborated by the corresponding textbook proof [6]: the latter does not use a set-valued transition function at all (although it has been defined) but argues informally in terms of 'paths', which corresponds to the relation steps.

2.4 Equivalences

Every `nae` can be translated into an equivalent `na` which can then be translated into an equivalent `da`. Since we are not interested in `na`s, we have defined a direct translation from `nae`s into `da`s which combines the powerset and ε-closure construction:

```
nae2da :: (α,σ)nae ⇒ (α,σ set)da
nae2da A ≡ ({start A},
            λ a Q. ⋃(next A (Some a) ‘‘ ((eps A)^* ^^ Q)),
            λ Q. ∃ p ∈ (eps A)^* ^^ Q. fin A p)
```

We use two further standard constructs, the image of a set under a function and a relation:

```
f ‘‘ S ≡ {f x . x ∈ S}
r ^^ S ≡ {y. ∃ x ∈ S. (x,y) ∈ r}
```

The actual equivalence proof, i.e. the proof of

```
DA.accepts (nae2da A) w = NAe.accepts A w
```
(1)

is by rewriting with the lemma

```
(eps A)^* ^^ (DA.delta (nae2da A) w S) = steps A w ^^ S
```

which is proved by induction on `w`.

3 Regular expressions

Regular expressions represent *regular sets*. The latter are sets of strings finitely generated from finite sets by union, concatenation and iteration (the star operation). Concatenation is defined explicitly

```
conc :: α list set ⇒ α list set ⇒ α list set
conc A B ≡ {xs@ys . xs ∈ A ∧ ys ∈ B}
```

whereas the star operation is defined inductively:

```
star :: α list set ⇒ α list set
[] ∈ star A
a ∈ A ∧ as ∈ star A ⟹ a@as ∈ star A
```

Two easy inductions yield an alternative characterization of `star`:

```
w ∈ star A = (∃ as. (∀ a ∈ set as. a ∈ A) ∧ (w = concat as))
```

Regular expressions are defined as usual

```
datatype α rexp = Empty
                | Atom α
                | Union (α rexp) (α rexp)
                | Conc (α rexp) (α rexp)
                | Star (α rexp)
```

as is the `language` denoted by a regular expression:

```
lang :: α rexp ⇒ α list set
lang Empty       = {}
lang (Atom a)    = {[a]}
lang (Union r s) = (lang r) ∪ (lang s)
lang (Conc r s)  = conc (lang r) (lang s)
lang (Star r)    = star (lang r)
```

Note that there is no separate constructor for a regular expression denoting the set { `[]` } because `Star Empty` does just that.

4 Regular expressions into nondeterministic automata

This section is the core of the paper. It discusses the transformation of regular expressions into nondeterministic automata with ε-transitions. We follow the spirit of the standard inductive construction [6], but simplify things a little: we do not insist that each automaton has only one final state and no transitions out of this state. The simplified construction of the union and iteration of automata is shown in Fig. 2. The capital F represents a set of final states.

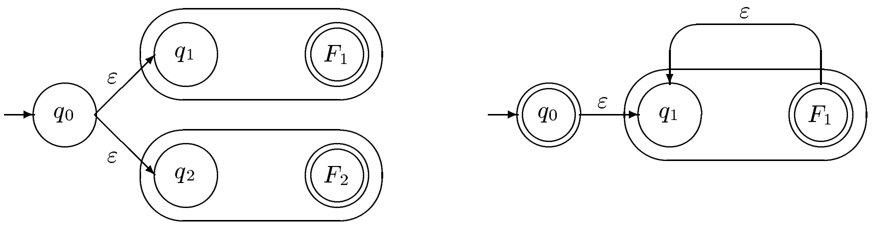

Fig. 2. Union and iteration of automata

The function we want to define has to be of type `(α)rexp ⇒ (α,σ)nae`. It remains to be determined what σ should be. The main criterion is the ease of renaming the states of an automaton to ensure they are disjoint from some other automaton. Graphically, this is easy: simply draw the two automata in nonoverlapping areas (e.g. as in Fig. 2 on the left). Adding offsets to natural numbers comes to mind, but this can be messy in proofs. Instead we use lists of Booleans and stick True or False in front to guarantee distinctness. Thus the above σ is simply `(bool)list` and we define the type

```
(α)bitsNAe = (α,(bool)list)nae
```

and the function

```
rexp2nae :: α rexp ⇒ α bitsNAe
rexp2nae Empty        = ([], λ a s. {}, λ s. False)
rexp2nae(Atom a)      = atom a
rexp2nae(Union r s)   = union (rexp2nae r) (rexp2nae s)
rexp2nae(Conc r s)    = conc (rexp2nae r) (rexp2nae s)
rexp2nae(Star r)      = star (rexp2nae r)
```

Let us first examine the translation of Empty. The initial state is the empty list. The transition function always returns the empty set. Hence there is no transition out of any state, in particular not out of `[]`. Thus the only reachable state is `[]`. There is no final state because the last component (fin) always returns False. Hence the automaton accepts no word, as required by Empty.

The definition of atom is analogous.

We could now go through the remaining constructions one by one, but it will suffice to examine conc in detail:

```
conc :: α bitsNAe ⇒ α bitsNAe ⇒ α bitsNAe
conc ≡ λ(ql,dl,fl)(qr,dr,fr).
   (True#ql,
    λa s. case s of
            [] ⇒ {}
          | left#s ⇒ if left then (True ## dl a s) ∪
                                  (if fl s ∧ a=None then {False#qr}
                                                    else {})
                             else False ## dr a s,
    λs. case s of [] ⇒ False | left#s ⇒ ¬left ∧ fr s)
```

The idea is to prefix states of the left automaton (let us call it L) with True and states of the right automaton (let us call it R) with False. Hence the start state of the concatenation is `True#ql`, where `ql` is the start state of L. There are no transitions out of the (unreachable) state `[]`. To describe the remaining transitions we have introduced an abbreviation: `##` is `#` lifted to sets of lists, i.e. `x##XS` stands for `(λxs. x#xs) ‘‘ XS`.

Transitions out of a state `left#s` depend on `left`. If `left` is True, i.e. we are in L, we take the transitions of L out of `s` together with an ε-transition to `False#qr`, where `qr` is the start state of R, in case `s` is a final state of R. If `left` is False, we simply take the transitions of R. The operation `##` lifts states of L and R to states of their concatenation. The final states are those of R.

The definitions of union and star are analogous.

If the reader finds the above treatment in terms of bit lists revoltingly concrete, I cannot disagree. A more abstract approach is clearly desirable.

4.1 The proof

The proof plan in the large is easy: show

```
accepts (rexp2nae r) w = (w ∈ lang r)                    (2)
```

by induction on `r` using the obvious lemmas

```
accepts (atom a)    w = (w = [a])
accepts (union L R) w = (accepts L w ∨ accepts R w)
accepts (conc L R)  w = (∃ u v. w = u@v ∧ accepts L u ∧ accepts R v)
accepts (star A)    w =
  (∃ us. (∀ u ∈ set us. accepts A u) ∧ (w = concat us))
```

The realization of this plan is, unfortunately, a textbook example of the gap between graphical intuition and formal proof. All of the lemmas appear obvious given a picture of the composition of automata such as Fig. 2. Yet their proofs require a painful amount of detail. For your amusement, the lemmas for the conc-case are shown in Fig. 3.

```
fin (conc L R) (True#p) = False
fin (conc L R) (False#p) = fin R p
(True#p,q) ∈ step (conc L R) a =
  ((∃ r. q=True#r ∧ (p,r) ∈ step L a) ∨ (fin L p ∧ a=None ∧ q=False#start R))
(False#p,q) ∈ step (conc L R) a = (∃ r. q = False#r ∧ (p,r) ∈ step R a)
(False#p,fq) ∈ (eps(conc L R))^* ⟹ ∃ q. (p,q) ∈ (eps R)^* ∧ fq = False#q
(p,q) ∈ (eps R)^* ⟹ (False#p, False#q) ∈ (eps(conc L R))^*
(False#p,fq) ∈ (eps(conc L R))^* = (∃ q. fq = False#q ∧ (p,q) ∈ (eps R)^*)
(False#p,fq) ∈ steps (conc L R) w = (∃ q. fq = False#q ∧ (p,q)∈ steps R w)
(p,q) ∈ (eps L)^* ⟹ (True#p,True#q) ∈ (eps(conc L R))^*
(p,q) ∈ steps L w ⟹ (True#p,True#q) ∈ steps (conc L R) w
(True#p,tq) ∈ (eps(conc L R))^* ⟹
  (∃ q. tq = True#q ∧ (p,q) ∈ (eps L)^*) ∨
  (∃ q r. tq = False#q ∧ (p,r)∈(eps L)^* ∧ fin L r ∧ (start R,q) ∈ (eps R)^*)
(p,q) ∈ (eps L)^* ⟹ (True#p,True#q) ∈ (eps(conc L R))^*
(p,q) ∈ step R None ⟹ (False#p,False#q) ∈ step (conc L R) None
(p,q) ∈ (eps R)^* ⟹ (False#p,False#q) ∈ (eps(conc L R))^*
fin L p ⟹ (True#p,False#start R) ∈ eps(conc L R)
((True#p,q) ∈ (eps(conc L R))^*) =
  ((∃ r. (p,r) ∈ (eps L)^* ∧ q = True#r) ∨
   (∃ r. (p,r) ∈ (eps L)^* ∧ fin L r ∧
         (∃ s. (start R, s) ∈ (eps R)^* ∧ q = False#s)))
(True#p,q) ∈ steps (conc L R) w ⟹
  ((∃ r. (p,r) ∈ steps L w ∧ q = True#r)  ∨
   (∃ u v. w = u@v ∧ (∃ r. (p,r) ∈ steps L u ∧ fin L r ∧
           (∃ s. (start R,s) ∈ steps R v ∧ q = False#s))))
```

Fig. 3. Lemmas for correctness of conc

If you examine the lemmas in Fig. 3 carefully, you will find that each one is very reasonable, i.e. none of them is contrived to fit the needs of the theorem prover. Apart from the last two, which require 10 and 24 steps respectively, all of them are proved in 3 or 4 steps: induction plus (automatic) predicate calculus reasoning and a bit of simplification. However, because of the form of the lemmas, predicate calculus reasoning dominates. Fortunately, Isabelle now provides the right kind of automation [10], whereas the previous generation of Isabelle's predicate calculus reasoning tools [9] floundered on some of the lemmas. Unfortunately, predicate calculus reasoning is inherently less pleasant than simplification because a failed attempt of an automatic procedure yields no information on what is missing. Hence you have to start your own manual single step proof to discover where things go wrong, which is how we found the lemmas in Fig. 3. This is in contrast to simplification, where a failed proof attempt results in a new goal that, in many cases, is a strong clue as to what the missing lemma is. Hence the design decision for a relational treatment as discussed in §2.3 also has its drawbacks. For example, in a functional style, the lemma

```
(False#p,fq) ∈ steps (conc L R) w =
  (∃ q. fq = False#q ∧ (p,q) ∈ steps R w)
```

becomes

```
delta (conc L R) a (False ## Q) = False ## delta R a Q
```

Despite these difficulties, the relational approach appears simpler. Proponents of relation algebra might point out that the reduction to predicate calculus is responsible for all complications, and a purely relation-algebraic treatment would have been much slicker. They may well be right.

The derivation of the lemmas for `union` and `star` is entirely similar. If we now put (1) and (2) together, we obtain the main correctness theorem:

```
DA.accepts (nae2da(rexp2nae r)) w = (w ∈ lang r)
```

5 The scanner

We will now turn deterministic automata into scanners as described in the introduction. It is easy to see that the concept of repeatedly chopping a maximal prefix off a string is independent of automata theory and can be parameterized by an arbitrary predicate on strings. Thus we will first do a bit of list processing, followed by two applications: the scanner, and, as an afterthought, paragraph filling. The nice thing is that the hard part of the development, including most of the proofs, is confined to the generic list processing functions.

5.1 Chopping up lists

We start by specifying the requirements. What does it mean to chop a list up into maximal prefixes? The prefix ordering on lists is defined as usual:

```
xs ≤ zs  ≡  ∃ ys. zs = xs@ys
```

Note the overloading of $\leq$. Then what is a maximal prefix of a list w.r.t. a predicate? The answer is almost obvious

```
is_maxpref P xs ys ≡
xs ≤ ys ∧ (xs=[] ∨ P xs) ∧ (∀ zs. zs ≤ ys ∧ P zs ⟶ zs ≤ xs)
```

except that we also allow `[]` to be a maximal prefix in case `ys` has no prefix that satisfies `P`. This definition makes sense in our context where the maximal prefix should never be empty (because chopping it off should reduce the list). Thus we can use `[]` as an indication that there is no nonempty prefix that satisfies `P`.

We now come to the main specification. The class of functions we want to specify are of type

```
α chopper  =  α list ⇒ α list list × α list
```

The specification is a predicate

```
is_maxchopper :: (α list ⇒ bool) ⇒ α chopper ⇒ bool
```

that expresses when its second argument correctly chops up lists according to its first argument:

```
is_maxchopper P chopper ≡
∀ xs zs yss.
    (chopper(xs) = (yss,zs)) =
    (xs = concat yss @ zs ∧ (∀ ys ∈ set yss. ys ≠ []) ∧
    (case yss of
       [] ⇒ is_maxpref P [] xs
     | us#uss ⇒ is_maxpref P us xs ∧
                 chopper(concat(uss)@zs) = (uss,zs)))
```

Let's recast this into words: `chopper(xs)` returns `(yss,zs)` iff

1. the concatenation of the outputs yields the input,
2. all elements of `yss` are nonempty, and
3. if `yss` is empty, then there is no nonempty prefix of `xs` that satisfies `P`, and if `yss = us#uss`, then `us` is the maximal prefix of `xs` w.r.t. `P` and chopping up the remaining list yields `(uss,zs)`.

Note that instead of an unjustified axiom specifying a constant chopper, the predicate is_maxchopper is merely an abbreviation.

Note also that although the specification only says that the first element `us` of `yss` must be a maximal prefix, the remaining elements are covered by the "recursive" call of `chopper` in the final line. A direct specification of the maximal prefix property for all elements of `yss` is more involved because the list of which they are a prefix is not directly at hand.

Now that we have the main specification, let us look at an implementation:

1. function maxsplit splits a list into a maximal prefix and the remaining list;
2. function chop iterates the process of splitting off a prefix.

To make things more modular, we introduce the type

```
α splitter  =  α list ⇒ α list × α list
```

and a separate specification

```
is_maxsplitter :: (α list ⇒ bool) ⇒ α splitter ⇒ bool
is_maxsplitter P splitf ≡
(∀ xs ps qs. (splitf xs = (ps,qs)) = (xs=ps@qs ∧ is_maxpref P ps xs))
```

that `maxsplit` should satisfy. The definition of

```
maxsplit :: (α list ⇒ bool) ⇒ α list × α list ⇒ α list
            ⇒ α splitter
maxsplit P r ps []     = (if P ps then (ps,[]) else r)
maxsplit P r ps (q#qs) = maxsplit P (if P ps then (ps,q#qs) else r)
                                    (ps@[q]) qs
```

is fairly easy: `r` is the maximal result found so far, `ps` the prefix accumulated since the initial call, and `qs` is the suffix that remains to be examined; `r` is updated every time a longer prefix that satisfies `P` is found.

Once you come up with and prove (by induction on `qs`) the lemma

```
(maxsplit P r ps qs = (xs,ys)) =
(if ∃ us. us ≤ qs ∧ P(ps@us)
 then xs@ys=ps@qs ∧ is_maxpref P xs (ps@qs) else (xs,ys)=r)
```

it follows easily that `maxsplit`, suitably initialized, meets its specification:

```
is_maxsplitter P (λ xs. maxsplit P ([],xs) [] xs)
```

Note that `maxsplit` traverses the whole list. Iterating `maxsplit` may therefore lead to quadratic run times. This problem could be overcome if there were an additional test whether `ps` can at all be extended to a list satisfying `P`. In terms of automata theory, this corresponds to a test whether the current state is an 'error' state which does not lead to any final state.

We now come to our main function `chop` that turns splitters into choppers by iterating them:

```
chop :: α splitter ⇒ α chopper
reducing splitf ⟹
  chop splitf xs = (let (pre,post) = splitf xs
                    in if pre=[] then ([],xs)
                       else let (xss,zs) = chop splitf post
                            in (pre#xss,zs))
```

Note that this a direct consequence of the actual definition by wellfounded recursion, which is not shown. The precondition involving

```
reducing :: α splitter ⇒ bool
reducing splitf ≡
∀ xs ys zs. splitf xs = (ys,zs) ∧ ys ≠ [] ⟶ length zs < length xs
```

is necessary to guarantee termination of `chop`. With the help of a few lemmas (proved by induction on the length of a list) one can establish

```
is_maxsplitter P splitf ⟹ is_maxchopper P (chop splitf)
```

5.2 Scanning

Now we specialize the above generic functions to perform scanning, i.e. chopping up strings based on the acceptance by a deterministic automaton. A naïve solution is to call `maxsplit` with the predicate `(accepts A)`. But since `accepts` is applied to longer and longer prefixes, this leads to quadratic run times.

Thus we need to re-implement `maxsplit`, replacing the predicate by an accepting `da` together with its current state:

```
auto_split :: (α,σ)da ⇒ σ  ⇒ α list × α list ⇒ α list
              ⇒ α splitter
auto_split A q r ps []     = (if fin A q then (ps,[]) else r)
auto_split A q r ps (x#xs) =
 auto_split A (next A x q) (if fin A q then (ps,x#xs) else r) (ps@[x]) xs
```

Although it may seem that `maxsplit` is completely superseded by `auto_split` and need never have been defined, the opposite is true: it is now trivial (an induction on `xs`) to show

```
auto_split A (delta A ps q) r ps xs =
maxsplit (λ ys. fin A (delta A ys q)) r ps xs
```

which, putting the results of §5.1 together, yields the corollary

```
is_maxchopper (accepts A) (scan A)
```

where

```
scan :: (α,σ)da ⇒ α chopper
scan A ≡ chop (λ xs. auto_split A (start A) ([],xs) [] xs)
```

is our main function. As predicted above, specializing the generic development to automata is easy.

If the whole development appears overly modular, I recommend the following more direct definition of the scanner

```
acc A s r ps []     ys = (if ys=[] then r else (ys#fst(r),snd(r)))
acc A s r ps (x#xs) ys = (let t = next A x s in
    if fin A t
    then acc A t (acc A (start A) ([],xs) [] xs [])
             (ps@[x]) xs (ps@[x])
    else acc A t r (ps@[x]) xs ys)
```

due to Roland Handl [5]. It mixes the generic and the specific and, on top of that, is primitive recursive.[3] Although `acc` confuses me to this day, Richard

---

[3] Nested primitive recursion can be reduced to ordinary primitive recursion [3]. Handl was forced to use primitive recursion because at the time HOL did not provide easy access to wellfounded recursion.

Mayr managed to verify it. However, the proof is sufficiently unpleasant that there had to be a better way to do it. What I presented above is the result of my quest for a more appealing solution.

5.3 Filling paragraphs

After the completion of the above development I suddenly remembered that Bird and Wadler [1] also define a function `scan`. On looking it up, I found that it is used in a similar application, namely filling paragraphs (pages 91–92). This made me realize that one can define their function

```
fill :: nat ⇒ (α list list) ⇒ (α list list list)
```

that takes a list of words and returns a list of lines that are no longer than the given line width (the first parameter), as an instance of our `scan`:

```
fill n ≡ fst(scan (0, λ xs i. i+length(xs)+1, λ i. i ≤ n+1))
```

The second component of the result of `scan` is dropped (`fst` selects the first component) to make the function conform to the type in [1]. If none of the input words is longer than the line width, the second component is always `[]`.

6 Does it run?

To be more precise: can the definitions of the main functions `rexp2nae`, `nae2da` and `scan` (and of their supporting functions) be interpreted directly in a functional programming language? For `scan`, the answer is yes: only primitive or wellfounded recursion on lists is used. Deterministic automata are also easy, but nondeterministic automata cause a little problem: we need to implement sets. In full generality, this is impossible, but finite sets can be represented as lists, which is one of the standard examples of implementation concepts for abstract data types. Fortunately, the sets arising in `rexp2nae` are all finite, thanks to the finite nature of regular expressions. Hence `rexp2nae` is also executable (although a replacement of finite sets by lists would be tricky to perform automatically in HOL).

The real problem arises with the definition of `nae2da`, which contains the inductively defined transitive closure operator `^*` and is therefore definitely not directly executable. Even if all sets in sight are finite, we would still need a recursive function for computing the transitive closure. Hence the answer to the section title is 'almost'.

There are a number of solutions to this problem:

- Show that `rexp2nae` only produces finite automata and define a recursive version of `^*` that operates on finite relations. This is possible but most likely messier than the next alternative.
- Generate a nondeterministic automaton *without* ε-steps directly from a regular expression. Although this complicates the construction a little, I expect the proofs actually become simpler because ε-steps are eliminated.

- Give a concrete finite representation of the transition function of automata in terms of, for example, association lists. This does entail rephrasing `rexp2nae` in terms of this representation, but I believe that one can reuse most of the proofs by showing that the concrete representation is a correct implementation of the abstract automaton model of this paper.

We intend to investigate the last two options in the near future. Note that there is a fundamental difference between them. Performing the conversion of nondeterministic automata on our functional representation postpones most of the work until run time, where states of the `da` are represented as sets of states of the `na`, all of which have to be processed. Given a concrete data structure for the transition function of the `na`, it is possible to eliminate this overhead by representing each of the (finitely many!) sets of states by a single new state. Nevertheless, the speedup is only a constant factor that depends on the size of the state space. Both representations allow `DA.accepts` to operate in time linear in the size of the input string. Scanning, however, is quadratic, because the recognition of each maximal prefix requires traversing the whole (remaining) string.

7 Related work

I am aware of three other papers on formalized automata theory [7, 4, 2], all of which use constructive type theory (i.e. they extract their algorithms from the proofs rather than providing them as part of the definitions) and follow [6] closely. The main result of Kreitz [7] is the pumping lemma and the main result of Constable *et al.* [2] the Myhill/Nerode theorem. Both of them use the Nuprl system.

Closest to our work is that by Filliâtre [4] who gives a constructive proof for the translation of regular expressions into nondeterministic finite automata with ε-moves in the Coq system.[4] Although the transition relation of the resulting automaton has a nice concrete representation as a finite set of triples, he does not consider the further conversion into a deterministic automaton (nor the scanning aspect). It is the latter conversion where executability breaks down for us because we use the transitive closure operator `^*`.

Thompson [11] presents an implementation (no proofs) of regular expressions and finite automata in Miranda.

8 Conclusion

We have seen a formalization of a (very simple) lexical analyzer generator taking us from a regular expression right to the actual scanner. Almost all of the

---

[4] Contrary to the title of that paper, the opposite direction is not mentioned. I have formalized and verified the translation of automata into regular sets as a recursive algorithm similar to Warshall's. The details are beyond the current paper.

functions involved are directly executable. Ignoring the small executability gap in our development (see §6), this work shows that HOL is eminently suitable to verify (total) functional programs, although HOL is neither constructive (where you often worry if the extracted program will be what you think it should be) nor a quantifier-free logic of recursively defined functions.

The size of the combined theories and proofs is quite acceptable: roughly 1000 lines dedicated to automata and regular expressions, and fewer than 400 lines involving the scanner. My first attempts in this direction go back a number of years and include dead alleys explored by students. The bulk of the development presented in this paper took me about 3 intensive weeks.

Although the work was not intended as a formalization of a specific textbook (in contrast to [2] or [8]), I feel that Hopcroft and Ullman's treatment has influenced me more than necessary, and that a development bypassing ε-moves might have been better. This will be explored in the future.

Acknowledgment Stefan Weber helped with an initial version of the proofs in §4. David Basin, David von Oheimb, Larry Paulson and Markus Wenzel read a draft and commented on it at short notice, for which I am very grateful.

References

1. R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.
2. R. Constable, P. Jackson, P. Naumov, and J. Uribe. Constructively formalizing automata. In G. Plotkin and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner.* MIT Press, 1998. To appear.
3. W. Felscher. *Berechenbarkeit.* Springer-Verlag, 1993.
4. J.-C. Filliâtre. Finite automata theory in Coq. A constructive proof of Kleene's theorem. Technical Report 97-04, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, 1997.
5. R. Handl. Verifikation eines Scanners (mit Isabelle). Master's thesis, Institut für Informatik, TU München, 1993.
6. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.
7. C. Kreitz. Constructive automata theory implemented with the Nuprl proof development system. Technical Report TR 86-779, Dept. of Computer Science, Cornell University, 1986.
8. T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lect. Notes in Comp. Sci.*, pages 180–192. Springer-Verlag, 1996.
9. L. C. Paulson. Generic automatic proof tools. In R. Veroff, editor, *Automated Reasoning and its Applications.* MIT Press, 1997. Also Report 396, Computer Laboratory, University of Cambridge.
10. L. C. Paulson. A generic tableau prover and its integration with Isabelle. Technical Report 441, University of Cambridge, Computer Laboratory, 1998.
11. S. Thompson. Regular expressions and automata using Miranda. Available at `http://www.cs.ukc.ac.uk/pubs/1995/212`, 1995.

This article was typeset using the LaTeX macro package with the LLNCS2E class.