Handout 4 (Sulzmann & Lu Algorithm)

So far our algorithm based on derivatives was only able to say yes or no depending on whether a string was matched by regular expression or not. Often a more interesting question is to find out *how* a regular expression matched a string? Answering this question will help us with the problem we are after, namely tokenising an input string. The algorithm we will be looking at for this was designed by Sulzmann & Lu in a rather recent paper. A link to it is provided on KEATS, in case you are interested.[1]

In order to give an answer for how a regular expression matched a string, Sulzmann and Lu introduce *values*. A value will be the output of the algorithm whenever the regular expression matches the string. If not, an error will be raised. Since the first phase of the algorithm by Sulzmann & Lu is identical to the derivative based matcher from the first coursework, the function $nullable$ will be used to decide whether as string is matched by a regular expression. If $nullable$ says yes, then values are constructed that reflect how the regular expression matched the string. The definitions for regular expressions r and values v is shown next to each other below:

$$
\begin{array}{ccc@{\qquad}ccc}
\multicolumn{3}{c}{\text{regular expressions}} & \multicolumn{3}{c}{\text{values}} \\
r & ::= & \varnothing & v & ::= & \\
 & \mid & \epsilon & & & Empty \\
 & \mid & c & & \mid & Char(c) \\
 & \mid & r_1 \cdot r_2 & & \mid & Seq(v_1, v_2) \\
 & \mid & r_1 + r_2 & & \mid & Left(v) \\
 & & & & \mid & Right(v) \\
 & \mid & r^* & & \mid & [v_1, \ldots v_n]
\end{array}
$$

The point is that there is a very strong correspondence between them. There is no value for the $\varnothing$ regular expression, since it does not match any string. Otherwise there is exactly one value corresponding to each regular expression with the exception of $r_1 + r_2$ where there are two values, namely $Left(v)$ and $Right(v)$ corresponding to the two alternatives. Note that r^* is associated with a list of values, one for each copy of r that was needed to match the string. This means we might also return the empty list $[]$, if no copy was needed.

Graphically the algorithm by Sulzmann & Lu can be represneted by the picture in Figure 1 where the path from the left to the right involving $der/nullable$ is the first phase of the algorithm and $mkeps/inj$, the path from right to left, the second phase. This picture shows the steps required when a regular expression, say r_1, matches the string abc. We first build the three derivatives (according to a, b and c). We then use $nullable$ to find out whether the resulting regular expression can match the empty string. If yes we call the function $mkeps$.

The $mkeps$ function calculates a value for how a regular expression has matched the empty string. Its definition is as follows:

[1] In my humble opinion this is an interesting instance of the research literature: it contains a very neat idea, but its presentation is rather sloppy. In earlier versions of their paper, students and I found several rather annoying typos in their examples and definitions.

$$\begin{tikzcd}
r_1 \arrow[r, "der\,a"] & r_2 \arrow[r, "der\,b"] & r_3 \arrow[r, "der\,c"] & r_4\ nullable \arrow[d, "mkeps"] \\
v_1 & v_2 \arrow[l, "inj\,a"] & v_3 \arrow[l, "inj\,b"] & v_4 \arrow[l, "inj\,c"]
\end{tikzcd}$$

Figure 1: The two phases of the algorithm by Sulzmann & Lu.

$$\begin{array}{lcl}
mkeps(\epsilon) & \stackrel{\text{def}}{=} & Empty \\
mkeps(r_1 + r_2) & \stackrel{\text{def}}{=} & \text{if } nullable(r_1) \\
 & & \text{then } Left(mkeps(r_1)) \\
 & & \text{else } Right(mkeps(r_2)) \\
mkeps(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & Seq(mkeps(r_1), mkeps(r_2)) \\
mkeps(r^*) & \stackrel{\text{def}}{=} & []
\end{array}$$

There are no cases for ϵ and c, since these regular expression cannot match the empty string. Note also that in case of alternatives we give preference to the regular expression on the left-hand side. This will become important later on.

The second phase of the algorithm is organised recursively such that it will calculate a value for how the derivative regular expression has matched a string whose first character has been chopped off. Now we need a function that reverses this "chopping off" for values. The corresponding function is called *inj* for injection. This function takes three arguments: the first one is a regular expression for which we want to calculate the value, the second is the character we want to inject and the third argument is the value where we will inject the character. The result of this function is a new value. The definition of *inj* is as follows:

$$\begin{array}{ll}
inj\,(c)\,c\,Empty & \stackrel{\text{def}}{=} Char\,c \\
inj\,(r_1 + r_2)\,c\,Left(v) & \stackrel{\text{def}}{=} Left(inj\,r_1\,c\,v) \\
inj\,(r_1 + r_2)\,c\,Right(v) & \stackrel{\text{def}}{=} Right(inj\,r_2\,c\,v) \\
inj\,(r_1 \cdot r_2)\,c\,Seq(v_1, v_2) & \stackrel{\text{def}}{=} Seq(inj\,r_1\,c\,v_1, v_2) \\
inj\,(r_1 \cdot r_2)\,c\,Left(Seq(v_1, v_2)) & \stackrel{\text{def}}{=} Seq(inj\,r_1\,c\,v_1, v_2) \\
inj\,(r_1 \cdot r_2)\,c\,Right(v) & \stackrel{\text{def}}{=} Seq(mkeps(r_1), inj\,r_2\,c\,v) \\
inj\,(r^*)\,c\,Seq(v, vs) & \stackrel{\text{def}}{=} inj\,r\,c\,v\ ::\ vs
\end{array}$$

This definition is by recursion on the regular expression and by analysing the shape of the values. Therefore there are, for example, three cases for sequnece regular expressions. The last clause for the star regular expression returns a list where the first element is $inj\, r\, c\, v$ and the other elements are vs. That mean $\_ :: \_$ should be read as list cons.

To understand what is going on, it might be best to do some example calculations and compare with Figure 1. For this note that we have not yet dealt with the need of simplifying regular expressions (this will be a topic on its own later). Suppose the regular expression is $a \cdot (b \cdot c)$ and the input string is abc. The derivatives from the first phase are as follows:

$$\begin{array}{ll}
r_1: & a \cdot (b \cdot c) \\
r_2: & \epsilon \cdot (b \cdot c) \\
r_3: & (\varnothing \cdot (b \cdot c)) + (\epsilon \cdot c) \\
r_4: & (\varnothing \cdot (b \cdot c)) + ((\varnothing \cdot c) + \epsilon)
\end{array}$$

According to the simple algorithm, we would test whether r_4 is nullable, which in this case it is. This means we can use the function *mkeps* to calculate a value for how r_4 was able to match the empty string. Remember that this function gives preference for alternatives on the left-hand side. However there is only ϵ on the very right-hand side of r_4 that matches the empty string. Therefore *mkeps* returns the value

$$v_4 : \; Right(Right(Empty))$$

The point is that from this value we can directly read off which part of r_4 matched the empty string. Next we have to "inject" the last character, that is c in the running example, into this value v_4 in order to calculate how r_3 could have matched the string c. According to the definition of *inj* we obtain

$$v_3 : \; Right(Seq(Empty, Char(c)))$$

This is the correct result, because r_3 needs to use the right-hand alternative, and then ϵ needs to match the empty string and c needs to match c. Next we need to inject back the letter b into v_3. This gives

$$v_2 : \; Seq(Empty, Seq(Char(b), Char(c)))$$

Finally we need to inject back the letter a into v_2 giving the final result

$$v_1 : \; Seq(Char(a), Seq(Char(b), Char(c)))$$

This now corresponds to how the regular expression $a \cdot (b \cdot c)$ matched the string abc.

There are a few auxiliary functions that are of interest in analysing this algorithm. One is called *flatten*, written $|\_|$, which extracts the string "underlying" a value. It is defined recursively as

$$\begin{array}{lcl}
|Empty| & \stackrel{\text{def}}{=} & [] \\
|Char(c)| & \stackrel{\text{def}}{=} & [c] \\
|Left(v)| & \stackrel{\text{def}}{=} & |v| \\
|Right(v)| & \stackrel{\text{def}}{=} & |v| \\
|Seq(v_1, v_2)| & \stackrel{\text{def}}{=} & |v_1| @ |v_2| \\
|[v_1, \ldots, v_n]| & \stackrel{\text{def}}{=} & |v_1| @ \ldots @ |v_n|
\end{array}$$

Using flatten we can see what is the string behind the values calculated by *mkeps* and *inj* in our running example are:

$$\begin{array}{ll}
|v_4|: & [] \\
|v_3|: & c \\
|v_2|: & bc \\
|v_1|: & abc
\end{array}$$

This indicates that *inj* indeed is injecting, or adding, back a character into the value.

Simplification

Generally the matching algorithms based on derivatives do poorly unless the regular expressions are simplified after each derivatives step. But this is a bit more involved in algorithm of Sulzmann & Lu. Consider the last derivation step in our running example

$$r_4 = der\, c\, r_3 = (\varnothing \cdot (b \cdot c)) + ((\varnothing \cdot c) + \epsilon)$$

Simplifying the result would just give us ϵ. Running *mkeps* on this regular expression would then provide us with *Empty* instead of $Right(Right(Empty))$ that was obtained without the simplification. The problem is we need to recreate this more complicated value, rather than just *Empty*.

This requires what I call *rectification functions*. They need to be calculated whenever a regular expression gets simplified. Rectification functions take a value as argument and return a (rectified) value. Our simplification rules so far are

$$\begin{array}{lcl}
r \cdot \varnothing & \mapsto & \varnothing \\
\varnothing \cdot r & \mapsto & \varnothing \\
r \cdot \epsilon & \mapsto & r \\
\epsilon \cdot r & \mapsto & r \\
r + \varnothing & \mapsto & r \\
\varnothing + r & \mapsto & r \\
r + r & \mapsto & r
\end{array}$$

Applying them to r_4 will require several nested simplifications in order end up with just ϵ.

We can implement this by letting simp return not just a (simplified) regular expression, but also a rectification function. Let us consider the alternative case, say $r_1 + r_2$, first. We would first simplify the component regular expressions r_1 and r_2. This will return simplified versions (if they can be simplified), say r_{1s} and r_{2s}, but also two rectification functions f_{1s} and f_{2s}. We need to assemble them in order to obtain a rectified value for $r_1 + r_2$. In case r_{1s} simplified to $\varnothing$, we would continue the derivative calculation with r_{2s}. The Sulzmann & Lu algorithm would return a corresponding value, say v_{2s}. But now this needs to be "rectified" to the value

$$Right(v_{2s})$$

Unfortunately, this is not enough because there might be some simplifications that happened inside r_2 and for which the simplification function retuned also a rectification function f_{2s}. So in fact we need to apply this one too which gives

$$Right(f_{2s}(v_{2s}))$$

So if we want to return this as function, we would need to return

$$\lambda v.\, Right(f_{2s}(v))$$

which is the lambda-calculus notation for a function that expects a value v and returns everything after the dot where v is replaced by whatever value is given.

Let us package these ideas into a single function (still only considering the alternative case):

```
$simp(r)$:
  case $r = r_1 + r_2$
    let $(r_{1s}, f_{1s}) = simp(r_1)$
        $(r_{2s}, f_{2s}) = simp(r_2)$
    case $r_{1s} = \varnothing$: return $(r_{2s}, \lambda v.\, Right(f_{2s}(v)))$
    case $r_{2s} = \varnothing$: return $(r_{1s}, \lambda v.\, Left(f_{1s}(v)))$
    otherwise: return $(r_{1s} + r_{2s}, f_{alt}(f_{1s}, f_{2s}))$
```

We first recursively call the simlification with r_1 and r_2. This gives simplified regular expressions, r_{1s} and r_{2s}, as well as two rectification functions f_{1s} and f_{2s}. We next need to test whether the simplified regular expressions are $\varnothing$ so as to make further simplifications. In case r_{1s} is $\varnothing$ then we can return r_{2s} (the other alternative). However we need to now build a rectification function, which as said above is

$$\lambda v.\, Right(f_{2s}(v))$$

The case where $r_{2s} = \varnothing$ is similar. We return r_{1s} but now have to rectify such that we return

$$\lambda v.\, Left(f_{1s}(v))$$

Note that in this case we have to apply f_{1s}, not f_{2s}, which is responsible to rectify the inner parts of v. The otherwise-case is slightly interesting. In this case neither r_{1s} nor r_{2s} are $\varnothing$ and no further simplification can be applied. Accordingly, we return $r_{1s} + r_{2s}$ as the simplified regular expression. In principle we also do not have to do any rectification, because no simplification was done in this case. But this is actually not true: There might have been simplifications inside r_{1s} and r_2s. We therefore need to take into account the calculated rectification functions f_{1s} and f_{2s}. We can do this by defining a rectification function f_{alt} which takes two rectification functions as arguments

```
$f_{alt}(f_1, f_2) \stackrel{\text{def}}{=}$
    $\lambda v.$ case $v = Left(v')$:  return $Left(f_1(v'))$
         case $v = Right(v')$: return $Right(f_2(v'))$
```

In essence we need to apply in this case the appropriate rectification function to the inner part of the value v, whereby v can only be of the form $Right(\_)$ or $Left(\_)$.

The other interesting case with simplification is the sequence case. Here the main simplification function is as follows

```
$simp(r)$:          (continued)
  case $r = r_1 \cdot r_2$
    let $(r_{1s}, f_{1s}) = simp(r_1)$
        $(r_{2s}, f_{2s}) = simp(r_2)$
    case $r_{1s} = \varnothing$: return $(\varnothing, f_{error})$
    case $r_{2s} = \varnothing$: return $(\varnothing, f_{error})$
    case $r_{1s} = \epsilon$: return $(r_{2s}, \lambda v. Seq(f_{1s}(Empty), f_{2s}(v)))$
    case $r_{2s} = \epsilon$: return $(r_{1s}, \lambda v. Seq(f_{1s}(v), f_{2s}(Empty)))$
    otherwise: return $(r_{1s} \cdot r_{2s}, f_{seq}(f_{1s}, f_{2s}))$
```

whereby f_{seq} is again pushing the two rectification functions into the two components of the Seq-value:

```
$f_{seq}(f_1, f_2) \stackrel{\text{def}}{=}$
    $\lambda v.$ case $v = Seq(v_1, v_2)$: return $Seq(f_1(v_1), f_2(v_2))$
```

Note that in the case of r_{1s} (similarly r_{2s}) we use the function f_{error} for rectification. If you think carefully, then you will see that this function will actually never been called. Because a sequence with $\varnothing$ will never recognise any string and therefore the second phase of the algorithm would never been called. The simplification function still expects us to give a function. So in my own implementation I just returned a function which raises an error.

Records and Tokenisation

Algorithm by Sulzmann, Lexing