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, that is splitting it up into its "word" components. The algorithm we will be looking at 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 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 below:

$$
\begin{array}{rcll}
r & ::= & \varnothing & \\
 & | & \epsilon & \\
 & | & c & \\
 & | & r_1 \cdot r_2 & \\
 & | & r_1 + r_2 & \\
 & | & r^* &
\end{array}
\qquad
\begin{array}{rcl}
v & ::= & \\
 & & Empty \\
 & | & Char(c) \\
 & | & Seq(v_1, v_2) \\
 & | & Left(v) \\
 & | & Right(v) \\
 & | & [v_1, \ldots v_n]
\end{array}
$$

As you can see there is a very strong correspondence between regular expressions and values. There is no value for the $\varnothing$ regular expression (since it does not match any string). Then there is exactly one value corresponding to each regular expression. with the exception of $r_1 + r_2$ where there are two values $Left(v)$ 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 mean we might also return the empty list [], if no copy was needed.

Graphically the algorithm can be represneted by the picture in Figure 1 where the path involving *der*/*nullable* is the first phase of the algorithm and *mkeps*/*inj* 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 could have 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.

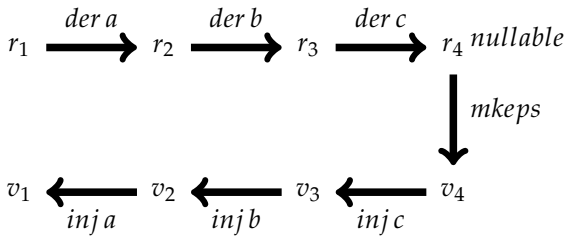

Figure 1: The two phases of the algorithm by Sulzmann & Lu.

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

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

The algorithm is organised recursively such that it will calculate a value for how the derivative regular expression has matched a string where the 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 returns a list where the first element is $inj\,r\,c\,v$ and the other elements are vs.

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 expreesions (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 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 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 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. This is the expected result. So at least in this case the algorithm seems to calculate what it is supposed to.

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

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

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.

Algorithm by Sulzmann, Lexing