Regular Expressions II

`match` code, struct induct on `regexp`, and other fun stuff

15-150 M21

Lecture 0709
09 July 2021

Recap of last time

- `''S` ranges over *equality types*
- Total functions `D : t -> bool` decide/compute sets of values:

aux-library/Language.sml

```
  type 'S language = 'S list -> bool
```

- `string`s and `char list`s are effectively the same:

```
String.explode : string -> char list
String.implode : char list -> string
```

aux-library/Language.sml

```
  val str : char language -> string -> bool
```

Regexp

```
datatype ’’S regexp =
    Zero
  | One
  | Const of ’’S
  | Plus of ’’S regexp * ’’S regexp
  | Times of ’’S regexp * ’’S regexp
  | Star of ’’S regexp
```

$$
\begin{aligned}
\mathcal{L}(\texttt{Zero}) &= \emptyset \\
\mathcal{L}(\texttt{One}) &= \{\texttt{[]}\} \\
\mathcal{L}(\texttt{Const(c)}) &= \{\texttt{[c]}\} \\
\mathcal{L}(\texttt{Plus(r1,r2)}) &= \mathcal{L}(\texttt{r1}) \cup \mathcal{L}(\texttt{r2}) \\
\mathcal{L}(\texttt{Times(r1,r2)}) &= \{\texttt{v}_1\texttt{@}\texttt{v}_2 \mid \texttt{v}_1 \in \mathcal{L}(\texttt{r1}) \text{ and } \texttt{v}_2 \in \mathcal{L}(\texttt{r2})\} \\
\mathcal{L}(\texttt{Star(r)}) &= \{\texttt{v}_1\texttt{@}\texttt{v}_2\texttt{@} \ \ldots \ \texttt{@}\texttt{v}_n \mid n \in \mathbb{N},\ \texttt{v}_1, \texttt{v}_2, \ldots, \texttt{v}_n \in \mathcal{L}(\texttt{r})\}
\end{aligned}
$$

`LL : ''S regexp -> ''S language`

ENSURES: `(LL R) : Sigma list -> bool` is a total function such that

$$\texttt{LL R cs} \Longrightarrow \texttt{true} \qquad \text{iff} \qquad \texttt{cs} \in \mathcal{L}(\texttt{R})$$

Demonstration: *A*B**

0 Implementing the matcher

```
''S regexp -> ''S list -> bool
```

Different ways of doing control flow

- `t -> bool`
- `t -> (bool -> 'a) -> 'a`
- `t -> (unit -> 'a) -> (unit -> 'a) -> 'a`
- `t -> (t' -> 'a) -> (unit -> 'a) -> 'a`
- `t -> (t' -> 'a) -> 'a` with an exception to indicate failure
- `t -> 'a` with exceptions to indicate success and failure

```
’’S regexp -> ’’S list -> (’’S list -> ’b)
-> ’b

exception NoMatch
```

```
''S regexp -> ''S list ->
(''S list * ''S list -> 'b)
-> 'b
```

Predicate Continuation with NoMatch

aux-library/Regexp.sml

```
exception NoMatch
```

We'll be working with predicate functions `k : Sigma list * Sigma list -> t` that are “**almost total**”: for all `(p,s)`, either `k(p,s)` evaluates to a value or it raises `NoMatch`

- `k(p,s)` $\hookrightarrow$ `v` to **accept** `(p,s)` **with value** `v`
- `k(p,s)` raises `NoMatch` to **reject** `(p,s)`

Defn. Given `cs : Sigma list`, a **splitting** of `cs` is a pair `(p,s) : Sigma list * Sigma list` such that `cs` $\cong$ `p@s`.

match Spec

```
match : ''S regex -> ''S list
     -> (''S list * ''S list -> 'b)
     -> 'b
```

REQUIRES: `k` is almost total

ENSURES:

$$\texttt{match R cs k} \cong \begin{cases} \texttt{v} & \text{where } \texttt{(p,s)} \text{ is a splitting of } \texttt{cs} \text{ such that } \texttt{p} \in \mathcal{L}(\texttt{R}) \text{ and } \texttt{k} \text{ accepts } \texttt{(p,s)} \text{ with result } \texttt{v}. \\ \texttt{raise NoMatch} & \text{if there is no such } \texttt{(p,s)} \end{cases}$$

LL

aux-library/Regexp.sml

```
 val LL = fn r => fn s =>
   match r s (fn (_,[]) => true | _ => raise NoMatch)
   handle NoMatch => false
```

$$\texttt{match Zero cs k} \cong \texttt{raise NoMatch}$$

$$\mathcal{L}(\texttt{Zero}) = \emptyset$$

aux-library/Regexp.sml

```
    fun match Zero _ _ = raise NoMatch
```

$$\texttt{match One cs k} \cong \begin{cases} \texttt{v} & \text{if } \texttt{k} \text{ accepts } \texttt{([],}\\ & \texttt{cs)} \text{ with result } \texttt{v} \\ \texttt{raise NoMatch} & \text{if } \texttt{k([],cs)} \text{ raises } \texttt{NoM} \end{cases}$$

$$\mathcal{L}(\texttt{One}) = \{\texttt{[]}\}$$

aux-library/Regexp.sml

```
      | match One cs k = k([],cs)
```

`match (Const c) cs k` ≅

- `v`
  if `cs=c'::cs'` such that `k` accepts `([c],cs')` with result `v`
- `raise NoMatch`
  if `cs=[]` or `cs=c'::cs'` such that either `c<>c'` or `k([c'],cs')` raises `NoMatch`

$$\mathcal{L}(\texttt{Const c}) = \{\texttt{[c]}\}$$

aux-library/Regexp.sml

```
     | match (Const(c)) [] k = raise NoMatch
     | match (Const(c)) (c'::cs') k =
         if c=c'
         then k([c'], cs')
         else raise NoMatch
```

Plus

aux-library/Regexp.sml

```
    | match (Plus(R1,R2)) cs k =
          (match R1 cs k
            handle NoMatch => match R2 cs k)
```

Times

aux-library/Regexp.sml

```
    | match (Times(R1,R2)) cs k =
        match R1 cs (fn (res',cs') =>
          match R2 cs' (fn (res'',cs'') =>
            k (res'@res'',cs'')))
```

aux-library/Regexp.sml

```
    | match (Star(r)) cs k =
      k([],cs)
        handle NoMatch =>
          match r cs (fn (res',cs') =>
            if (cs = cs')
            then raise NoMatch
            else
              match (Star(r)) cs' (fn (res'',
cs'') =>
                k(res'@res'',cs'')))
```

5-minute break

Documentation: Regexp Correctness Proof

match Spec

```
match : ''S regex -> ''S list
      -> (''S list * ''S list -> 'b)
      -> 'b
```

REQUIRES: `k` is almost total

ENSURES:

$$\texttt{match R cs k} \cong \begin{cases} \texttt{v} & \text{where } \texttt{(p,s)} \text{ is a splitting of } \texttt{cs} \text{ such that } \texttt{p} \in \mathcal{L}(\texttt{R}) \text{ and } \texttt{k} \text{ accepts } \texttt{(p,s)} \text{ with result } \texttt{v}. \\ \texttt{raise NoMatch} & \text{if there is no such } \texttt{(p,s)} \end{cases}$$

Demonstration: Converting Regex into the POSIX syntax

Thank you!