t-regex

Matching using tree regular expressions

Alejandro Serrano Mena
Universiteit Utrecht
A.SerranoMena@uu.nl

NL-FP Day 2015
U. Twente, 9 January 2015
Main Goal

In short: more powerful pattern matching
Main Goal

In short: more powerful pattern matching

1. Disjunctive patterns
   - Pattern match on 0 : xs or 1 : xs
Main Goal

In short: more powerful pattern matching

1. Disjunctive patterns
   ▶ Pattern match on \(0:xs\) or \(1:xs\)

2. Regular sequences
   ▶ Match lists of 1s, 2s or 3s on it: \((1 \mid 2 \mid 3)^*\)
   ▶ Match lists with a 123 sequence on it: \((123)^*\)
Main Goal

In short: more powerful pattern matching

1. Disjunctive patterns
   ▶ Pattern match on $0 : xs$ or $1 : xs$

2. Regular sequences
   ▶ Match lists of 1s, 2s or 3s on it: $(1 \mid 2 \mid 3)^*$
   ▶ Match lists with a 123 sequence on it: $(123)^*$

3. More exotic shapes
   ▶ Match binary trees with 2s everywhere
   ▶ Match an AST of the form $\text{Lit} 0 \text{`Plus` } x$ or $\text{Lit} 1 \text{`Times` } x$
     to perform simplification
Tree Regular Expressions

This reminds me a lot to regular expressions...
Tree Regular Expressions

This reminds me a lot to regular expressions...

Meet tree regular expressions!

1. Powerful way to describe tree-shaped structures
2. Well studied and related to tree automata

Tree Automata Techniques and Applications, Chapter 2
This reminds me a lot to regular expressions...

Meet tree regular expressions!

1. Powerful way to describe tree-shaped structures
2. Well studied and related to tree automata

*Tree Automata Techniques and Applications*, Chapter 2

- Lists of 1s or 2s: \((\text{Cons } 1 \ □ \ | \ \text{Cons } 2 \ □ \ | \ \text{Nil})^*,□\)
- Lists of the form \((12)^*\): \((\text{Cons } 1 \ (\text{Cons } 2 \ □) \ | \ \text{Nil})^*,□\)
- All-2s binary trees: \((\text{Node } □ 2 \ □ \ | \ \text{Leaf } 2)^*,□\)
t-regex is a Haskell implementation of tree regular expressions

- Building on pattern functors and generics
- Keeping well-formedness of expressions via PHOAS
- With custom patterns via quasi-quotation
- <shameOnMe> Includes one call to unsafePerformIO </shameOnMe>

Lists of 1s or 2s: $\text{(Cons 1 } \square | \text{ Cons 2 } \square | \text{ Nil})^*$

\[
\text{listOfOnesOrTwos} = \text{Regex }\$
\text{iter }\lambda k \rightarrow \text{ inj (Cons 1 (var k))}
\langle|\rangle \text{ inj (Cons 2 (var k))}
\langle|\rangle \text{ inj Nil}
\]
Pattern Functors

Usually we define things in a recursive way

```haskell
data List a = Cons a (List a) | Nil
```
Pattern Functors

Usually we define things in a recursive way

```haskell
data List a = Cons a (List a) | Nil
```

But we can use two steps instead

- Separate description of constructors from fix-point

```haskell
data List_ a l = Cons a l | Nil
type List a = Fix (List_ a)
newtype Fix f = In { out :: f (Fix f) }
```
Pattern Functors

Usually we define things in a recursive way

```haskell
data List a = Cons a (List a) | Nil
```

But we can use two steps instead

- Separate description of constructors from fix-point

```haskell
data List' a l = Cons a l | Nil
type List a = Fix (List' a)
newtype Fix f = In { out :: f (Fix f) }
```

\(\text{List'} a\) is a functor which we can reuse

```haskell
listOfOnesOrTwos :: Regex c (List' Int)
```
Sidewalk: Pattern Synonyms

But now we have to write a lot more!

- One \( \text{In} \) per constructor we want to inject

\[
\text{oneAndTwo} = \text{In} \, \text{Cons} \, 1 \, (\text{In} \, \text{Cons} \, 2 \, (\text{In} \, \text{Nil}))
\]

Since GHC 7.8, pattern synonyms come to our rescue

\[
\begin{align*}
\text{pattern Cons'} \, x \, xs &= \text{In} \, (\text{Cons} \, x \, xs) \\
\text{pattern Nil'} &= \text{In} \, \text{Nil} \\
\text{oneAndTwo} &= \text{Cons'} \, 1 \, (\text{Cons'} \, 2 \, \text{Nil}) \\
\text{map} \, f \, \text{Nil'} &= \text{Nil'} \\
\text{map} \, f \, (\text{Cons'} \, x \, xs) &= \text{Cons'} \, (f \, x) \, (\text{map} \, f \, xs)
\end{align*}
\]
The Five Main Operators
Whole Language:  \textit{any\_}

\textit{any\_} matches any term of the corresponding type

\begin{itemize}
\item Not that useful by itself\ldots
\item But as part of a larger regular expression
\end{itemize}
Injection: $inj$

To match a constructor $C$, you should use $inj (C...)$

$$inj \ Node \ any\_2 \ any\_$$

Each recursive element is again a tree regular expression

$$inj \ Node \ (inj \ Node \ any\_3 \ any\_)$$

$$\begin{array}{c}
\text{2} \\
\text{\textbackslash \textbackslash} \\
\text{?} \quad ?
\end{array}$$

$$inj \ Node \ (inj \ Node \ any\_4 \ any\_)$$

$$\begin{array}{c}
\text{2} \\
\text{\textbackslash \textbackslash \textbackslash} \\
\text{3} \quad \text{4} \\
\text{\textbackslash \textbackslash \textbackslash} \\
\text{?} \quad ? \quad ? \quad ?
\end{array}$$
Injection: __

Non-recursive positions are checked for equality
  ▶ As 2, 3 or 4 in the previous examples

You can use __ to match anything in those positions

\[
inj \ Node \ (inj \ Node \ 3 \ \text{any\_} \ \text{any\_})
\]
\[
\_\_
\]
\[
(inj \ Node \ 4 \ \text{any\_} \ \text{any\_})
\]

Note the difference between both:
  ▶ \text{any\_} is part of the tree regular expression language
  ▶ __ bypasses equality checking on non-recursive positions
Alternation: \( \langle|\rangle \)

To provide a choice between expressions, use \( \langle|\rangle \)

\[
inj \; \$ \; Cons \; 1 \; (\; inj \; (Cons \; 2 \; (inj \; Nil)))
\]
\[
\langle|\rangle \; inj \; (Cons \; 3 \; (inj \; Nil)))
\]

matches both

- \( Cons' \; 1 \; (Cons' \; 2 \; Nil') \)
- \( Cons' \; 1 \; (Cons' \; 3 \; Nil') \)
Iteration: *iter and var*

- Kleene star * is used in regular expressions for iteration
- In trees, more than one iteration position are available
  - In strings, we only iter at the end
- We mark the position with a hole
  - And include its name next to the star

\[(\text{Node} \ 2 \ \text{□} \ \text{Leaf} \ 2)^*;\ □\]
Iteration: \textit{iter} and \textit{var}

Iteration is defined using \textit{iter}

\begin{itemize}
    \item Hole reference is represented as a binder
    \item We ensure well-formedness of the expression
\end{itemize}

Referring to a hole is done via \textit{var}

\[
\text{iter} \; \lambda k \rightarrow \; \text{inj} \; (\text{Node} \; (\text{var} \; k) \; 2 \; (\text{var} \; k)) \\
\langle|\rangle \; \text{inj} \; (\text{Leaf} \; 2)
\]

Or using the nice \texttt{PostfixOperators} GHC syntax extension

\[
(\lambda k \rightarrow \; \text{inj} \; (\text{Node} \; (k\#) \; 2 \; (k\#))) \langle|\rangle \; \text{inj} \; (\text{Leaf} \; 2))^*\]
Capture: *capture*

You would not call it pattern matching without *capturing*

\[
\text{iter } \$ \lambda k \rightarrow \text{inj} \ (\text{Node} (k\#) \_ \_ (k\#)) \\
\langle | \rangle \text{ capture } \text{"leaves"} \ (\text{inj} \ (\text{Leaf} \ _ \_))
\]

- Each capture group returns a list of terms
  - Actually, of your choice of *Alternative*
  - A more powerful analysis could detect unicity

An alternative name for *capture* is <<-
1. Declare your pattern functor and derive `Generic1`

   ```haskell
   {-# LANGUAGE DeriveGeneric #-}
   import GHC.Generics
   data Tree t = Node t Int t | Leaf Int
   deriving (Show, Generic1)
   ```
How do I use it?

1. Declare your pattern functor and derive *Generic1*

   ```haskell
   {-# LANGUAGE DeriveGeneric #-}
   import GHC.Generics
   data Tree t = Node t Int t | Leaf Int
   deriving (Show, Generic1)
   ```

2. If desired, take its fix-point and add pattern synonyms

   ```haskell
   {-# LANGUAGE PatternSynonyms #-}
   type Tree = Fix Tree
   pattern Node’ l x r = In (Node l x r)
   pattern Leaf’ x = In (Leaf x)
   ```
How do I use it?

1. Declare your pattern functor and derive \textit{Generic1}
   \begin{verbatim}
   {-# LANGUAGE DeriveGeneric #-}
   import GHC.Generics
   data Tree_ t = Node t Int t | Leaf Int deriving (Show, Generic1)
   \end{verbatim}

2. If desired, take its fix-point and add pattern synonyms
   \begin{verbatim}
   {-# LANGUAGE PatternSynonyms #-}
   type Tree = Fix Tree_
   pattern Node' l x r = In (Node l x r)
   pattern Leaf' x = In (Leaf x)
   \end{verbatim}

3. Import the needed modules
   \begin{verbatim}
   import Data.Regex.Generics
   import Data.Regex.TH
   \end{verbatim}
How do I use it?

4. Use the \texttt{rx} quasi-quoter with a tree regular expression

\begin{verbatim}
{-# LANGUAGE QuasiQuotation #-}
allLeaves [rx| iter $ \k -> inj (Node (k#) __ (k#)) <||> leaves <- inj (Leaf __) |]
= map (\(Leaf i) -> i) leaves
\end{verbatim}

Capture groups become \texttt{bindings} in the body
How do I use it?

4. Use the \texttt{rx} quasi-quoter with a tree regular expression

\begin{verbatim}
{-# LANGUAGE QuasiQuotation #-}
allLeaves [rx| iter $ \k -> inj (Node (k#) __ (k#)) ] leaves <- inj (Leaf __) ]
= map (\(Leaf i) -> i) leaves
\end{verbatim}

Capture groups become \textit{bindings} in the body

5. Or just call \texttt{match} or \texttt{matches} to check compliance

\begin{verbatim}
isTreeOfTwos = matches (Regex $ iter $ \lambda k ->
inj (Node (k#) 2 (k#)) \langle|\rangle inj (Leaf 2))
\end{verbatim}
Side-effect: HaRP Regular Patterns

HaRP is a preprocessor to match patterns on Haskell lists

▶ t-regex allows this via regular expressions on lists
▶ Quite useful to work on XML-like data

*Regular Expression Patterns* by Broberg, Farre and Svenningsson (U. Chalmers)
**Side-effect: (Restricted) OCaml Or-Patterns**

If we use \( \langle | \rangle \) in the top of a regular expression, we effectively have a **disjunctive pattern**

- OCaml has **or-patterns built-in**
- But are more powerful and more performant

\[
\textbf{data} \quad \text{Expr} \ e = \text{Plus} \ e \ e \mid \text{Times} \ e \ e \mid \text{Var} \ \text{Int}
\]

\[
\textbf{deriving} \quad (\text{Show}, \text{Generic1})
\]

\[
\begin{align*}
\text{simplify} & \quad : \quad \text{Fix} \ \text{Expr} \rightarrow \ \text{Fix} \ \text{Expr} \\
\text{simplify} \ [\langle \mid \rangle \ \text{inj} \ \text{Plus} \ (\text{inj} \ \text{Var} \ 0) \ (x \ <-\ any\_)] \\
& \qquad \langle \mid \rangle \ \text{inj} \ \text{Plus} \ (x \ <-\ any\_) \ (\text{inj} \ \text{Var} \ 0) \\& \qquad \langle \mid \rangle \ \text{inj} \ \text{Times} \ (\text{inj} \ \text{Var} \ 1) \ (x \ <-\ any\_) \\& \qquad \langle \mid \rangle \ \text{inj} \ \text{Times} \ (x \ <-\ any\_) \ (\text{inj} \ \text{Var} \ 1) \\& \quad = \quad \text{simplify} \ \text{head} \ x \\
\text{simplify} \ x & \quad = \quad x
\end{align*}
\]
Attribute Grammar

Define **attributes** on a tree-like structure, which are computed at each node by an evaluator

- **Inherited** flow top down, and **synthesized** bottom up

**t-regex** supports grammars with **regular lookahead**

- A regular shape needs to be matched for a rule to apply

\[
\text{evalExpr} :: \text{Grammar } \epsilon \text{ Expr } () \text{ Integer } \\
\text{evalExpr} = [ \\
\text{rule } \lambda x \ y \rightarrow \text{inj} (\text{Plus} (x \ \ll \ \text{any}_-) (y \ \ll \ \text{any}_-)) \Rightarrow \textbf{do} \\
\quad \text{valueX} \leftarrow \text{use} (\text{at } x \circ \text{syn}) \\
\quad \text{valueY} \leftarrow \text{use} (\text{at } y \circ \text{syn}) \\
\quad \text{this} \circ \text{syn} . = \text{valueX} + \text{valueY} \\
\text{, rule } \lambda x \ y \rightarrow \text{inj} (\text{Times} (x \ \ll \ \text{any}_-) (y \ \ll \ \text{any}_-)) \Rightarrow \textbf{do} \\
\quad -- \text{Similar to Plus} \\
\text{, rule } \text{inj} (\text{Var} \ \_-) \Rightarrow \lambda (\text{Var}^\prime \ v) \rightarrow \textbf{do} \\
\quad \text{this} \circ \text{syn} . = v \\
]\]
Random Generation

If we can express the shape of data with regular expressions...
Random Generation

If we can express the shape of data with regular expressions... why not use it to **generate** arbitrary values?

```haskell
*> import Data.Regex.Generics
    sample $ arbitraryFromRegex oneTwoOrOneThree
In (Cons 1 (In (Cons 3 (In Nil))))
In (Cons 1 (In (Cons 2 (In Nil))))
In (Cons 1 (In (Cons 2 (In Nil))))
...
```
Summary

Tree regular expressions are great!

- Pattern matching
- Attribute grammars with lookahead
- QuickCheck value generation

`t-regex` brings its power to Haskell

- Works with built-in `GHC.Generics`
  - Nice showcase for generic programming
- Nice syntax thanks to quasi-quotiation
- Also supports mutually-recursive data types