Matching regular expressions, revisited, (revisited)*

James McKinna, University of St.Andrews

james.mckinna@st-andrews.ac.uk

FUN in Nottingham, February 23, 2007

Last minute GADT hacking: Edwin Brady

Thanks to: Conor, Graham, and Jeremy

The *ur*-problem

Prove that the language recognition problem for regular grammars is

decidable

This should work. . .

You want decidable? Write a boolean valued function!

```
/* accept : RegExp -> String -> bool */

fun accept r s = acccps r (String.explode s) (fn Nil => true | _ => false)

/* acccps : RegExp -> char list -> (char list -> bool) -> bool */

fun acccps (RE0) cs k = false
| acccps (RE1) cs k = k cs
| acccps (REchar a) Nil k = false
| acccps (REchar a) (c :: cs) k = if (a = c) then k cs else false
| acccps (REplus r1 r2) cs k = acccps r1 cs k orelse (acccps r2 k cs)
| acccps (REtimes r1 r2) cs k = acccps r1 cs (fn cs' => acccps r2 cs' k)
| acccps (r as (REstar r1)) cs k = k cs orelse acccps r1 cs (fn cs' => acccps r cs' k)
```

Problem This function is not *total*. Why?

The *ur*-specification

Imagine a programming language whose type system is expressive enough to allow one to write down, as a *datatype*, the type of:

> "successful partial matches of a string str against a regexp r, exhibited as a parse tree p : L(r), plus a suffix string s, such that $str =$ render p s"

where render p s is the obvious flattening observation on the ADT of parse trees p, displaying them as strings.

String recognition with regexps (sketch): typing the recogniser

Consider the inductive family $\mathsf{Recog}\ r\ s$ of *partially successful matches* with constructor

$$\frac{p\ :\ \mathsf{L}(r) \quad s\ :\ \mathsf{String}A}{\mathsf{consume}\,p\,s\ :\ \mathsf{Recog}\ r\ (\mathsf{render}\ p\ s)}$$

which captures successful parsing of *an initial segment* of the input string.

Then a plausible typing for a recogniser is

$$\frac{p\ :\ \mathsf{L}(r) \quad s\ :\ \mathsf{String}A}{\mathsf{recog}\,p\,s\ :\ \mathsf{Recog}\ r\ s}$$

I: Introduction

Harper (JFP, 1999): “proof-directed debugging”

A Lakatos-style “failed proof” analysis, with repair, of:

- a continuation-passing style regexp matcher in SML
- partial correctness: tricky informal inductive argument, *assuming* inductively that continuations involved terminate: “*validity*”
- termination: *fails* for matching against 0^*
- repair:
  - eliminate null transitions: compute $\delta(r) = 0, 1$
  - put r in *standard form*: $r = \delta(r) + r^-$
- termination secured for $0, 1$ and regexps of the form r^-

Harper's broken matcher

```
/* accept : RegExp -> String -> bool */

fun accept r s = acccps r (String.explode s) (fn Nil => true | _ => false)

/* acccps : RegExp -> char list -> (char list -> bool) -> bool */

fun acccps (RE0) cs k = false
| acccps (RE1) cs k = k cs
| acccps (REchar a) Nil k = false
| acccps (REchar a) (c :: cs) k = if (a = c) then k cs else false
| acccps (REplus r1 r2) cs k = acccps r1 cs k orelse (acccps r2 k cs)
| acccps (REtimes r1 r2) cs k = acccps r1 cs (fn cs' => acccps r2 cs' k)
| acccps (r as (REstar r1)) cs k = k cs orelse acccps r1 cs (fn cs' => acccps r cs' k)
```

This function is not structural recursive because of the highlighted recursive call on `acccps`, and there is no guarantee that any of the string will have been consumed at that point.

A natural candidate for formalisation?

- Tricky inductive correctness argument
- Need to analyse possible non-termination of SML programs
- Re-frame the problem!
  - Work in a theory of *terminating* functions (EPIGRAM)
  - Proof by induction, definition by structural recursion
  - Distinguish *types* of regexps, resp. those in standard form; a *view* (Wadler (1987); McBride/McKinna (2004))
  - Matching can itself also be expressed as a *view* (of Strings)
  - Correctness evident by *type* (Curry-Howard as usual)
  - Lose CPS; recover direct-style matcher (continuations in the tail)

end of part I

II: Regular expressions and regular languages

Why dependent types matter

Our treatment makes use of dependent types to represent, in a uniform framework,

- the language sets $\mathsf{L}(r)$ for a given regexp r;
- the function rendering each $p \ : \ \mathsf{L}(r)$ as a string;
- derivation trees $i \ : \ r \leq s$ axiomatising the inclusion $\mathsf{L}(r) \subseteq \mathsf{L}(s)$;
- the mapping $[\![i]\!] \ : \ \mathsf{L}(r) \to \mathsf{L}(s)$, together with $\langle\langle i \rangle\rangle$ (which witnesses this inclusion), the proof that $[\![i]\!]$ preserves renderings;

in other words, the syntax, semantics *and* proof theory of regular languages.

The type of regular expressions

RE A,

- defined over an alphabet A,
- is definable as usual, with
  - 0,
  - 1,
  - characters a,
  - alternation $r + s$,
  - composition $r \,.\, s$,
  - and repetition (Kleene star) r^*; iteration, $r^+ = r \,.\, r^*$

The inductive family of language sets

We do *not* define $\mathsf{L}(r)$ directly as sets of strings, but as abstract syntax trees for successful parses of such strings:

data $\quad \dfrac{r \ : \ \mathsf{RE}\ A}{\mathsf{L}(r) \ : \ \star}$ $\quad$ where $\quad \cdots$

where $\quad \dfrac{}{\epsilon \ : \ \mathsf{L}(1)} \qquad \dfrac{a \ : \ A}{[a] \ : \ \mathsf{L}(a)} \qquad \dfrac{p \ : \ \mathsf{L}(r) \ ; \ q \ : \ \mathsf{L}(s)}{p \bullet q \ : \ \mathsf{L}(r \,.\, s)}$

$$\frac{p \ : \ \mathsf{L}(r)}{\mathsf{l}(p) \ : \ \mathsf{L}(r + s)} \qquad \frac{q \ : \ \mathsf{L}(s)}{\mathsf{r}(q) \ : \ \mathsf{L}(r + s)}$$

$$\frac{}{\varepsilon \ : \ \mathsf{L}(r^*)} \qquad \frac{p \ : \ \mathsf{L}(r) \ ; \ q \ : \ \mathsf{L}(r^*)}{p \star q \ : \ \mathsf{L}(r^*)}$$

NB. $\mathsf{L}(0)$ is an *empty* type: no constructor is declared for it.

Alternatively: a recursive definition

let $\dfrac{r \ : \ \mathsf{RE}\ A}{\mathsf{L}(r) \ : \ \star}$

$$
\begin{array}{lcl}
\mathsf{L}(0) & \Rightarrow & \mathbf{0} \\
\mathsf{L}(1) & \Rightarrow & \mathbf{1} \\
\mathsf{L}(a) & \Rightarrow & \{a\} \\
\mathsf{L}(r \,.\, s) & \Rightarrow & \mathsf{L}(r) \times \mathsf{L}(s) \\
\mathsf{L}(r + s) & \Rightarrow & \mathsf{L}(r) \oplus \mathsf{L}(s) \\
\mathsf{L}(r^*) & \Rightarrow & \mathsf{List}\ (\mathsf{L}(r))
\end{array}
$$

Rendering parse trees as strings: obvious fusion

$$\underline{\text{let}} \quad \frac{p \ : \ \mathsf{L}(r) \ ; \ str \ : \ \mathsf{String}\ A}{\mathsf{render}\ p\ str \ : \ \mathsf{String}\ A}$$

$$\begin{array}{lcl}
\mathsf{render}\ \epsilon\ str & \Rightarrow & str \\
\mathsf{render}\ [a]\ str & \Rightarrow & \mathsf{prefix}\ a\ str \\
\mathsf{render}\ (p \bullet q)\ str & \Rightarrow & \mathsf{render}\ p\ (\mathsf{render}\ q\ str) \\
\mathsf{render}\ (\mathsf{l}(p))\ str & \Rightarrow & \mathsf{render}\ p\ str \\
\mathsf{render}\ (\mathsf{r}(q))\ str & \Rightarrow & \mathsf{render}\ q\ str \\
\mathsf{render}\ \varepsilon\ str & \Rightarrow & str \\
\mathsf{render}\ (p \star q)\ str & \Rightarrow & \mathsf{render}\ p\ (\mathsf{render}\ q\ str)
\end{array}$$

An obvious 'fusion' lemma

exercise: prove the following

$$\mathsf{render}\ p\ str = \mathsf{append}\ (\mathsf{render}\ p\ [])\ str$$

with $[]$ the null string, and string concatenation append as usual.

Standard form regular expressions

The crux of the termination argument:

- identify the sublanguage of expressions s : SRE A
- such that their parse trees p : $\mathsf{L}(s)$ render as *non-empty* strings.
- (matching against such an s is guaranteed to terminate, by well-founded induction on string suffixes, using the fusion lemma)

$$\underline{\text{data}} \quad \frac{A \ : \ \star}{\text{SRE } A \ : \ \star} \qquad \underline{\text{where}} \quad \frac{}{0 \ : \ \text{SRE } A} \qquad \frac{a \ : \ A}{a \ : \ \text{SRE } A}$$

$$\ldots \quad \frac{s, s' \ : \ \text{SRE } A}{s + s' \ : \ \text{SRE } A} \qquad \frac{s, s' \ : \ \text{SRE } A}{s \,.\, s' \ : \ \text{SRE } A} \qquad \frac{s \ : \ \text{SRE } A}{s^{+} \ : \ \text{SRE } A}$$

Standardising regular expressions: the theorem

- An obvious erasure from SRE A to RE A, $|s|$
- re-presentation of Harper's analysis by proving the following normal form theorem for regexps: every r : RE A is equivalent to either
  - $|s|$ for s : SRE A, or
  - $1 + |s|$ for s : SRE A
- 'equivalent' here means: generates the same strings via render

Standardising regular expressions: the function

- the two cases of the theorem correspond to two constructors of a type family, $\mathsf{Std}\ r$, (the cases where $\delta(r) = 0$, $\delta(r) = 1$ in Harper)
- data $\quad \dfrac{r\ :\ \mathsf{RE}\ A}{\mathsf{Std}\ r\ :\ \star}$ $\quad$ where

$$\frac{s\ :\ \mathsf{SRE}\ A \quad i\ :\ |s| \equiv r}{\mathsf{Std}_0\ s\ :\ \mathsf{Std}\ |s|} \qquad \frac{s\ :\ \mathsf{SRE}\ A \quad i\ :\ 1 + |s| \equiv r}{\mathsf{Std}_1\ s\ i\ :\ \mathsf{Std}\ r}$$

- the theorem is witnessed by a function of type

$$\frac{r\ :\ \mathsf{RE}\ A}{\mathsf{std}\ r\ :\ \mathsf{Std}\ r}$$

- incl. all the machinery behind $r \equiv \delta(r) + r^-$

Axiomatising the equational theory of regular expressions

- $r \equiv s$ is (a fragment of) the familar equational theory of Kleene algebra (semantically, "$\mathsf{L}(r) \equiv \mathsf{L}(s)$")
- technically easier to mix the equational and inequational theory, capturing "$r \leq r'$" as well: just another inductive family...
- crucial soundness lemma: if $r \leq r'$, then $\mathsf{L}(r) \subseteq \mathsf{L}(r')$
- this lemma becomes a *function*, factorised as
  - a mapping $[\![i]\!] : \mathsf{L}(r) \to \mathsf{L}(r')$; defined by structural induction over $i : r \leq r'$, respectively $i : r \equiv r'$;
  - a proof $\langle\langle i \rangle\rangle$ that render $([\![i]\!]\, p)\ str =$ render $p\ str$;

end of part II

III: Writing the matcher

Specifying the matcher

- The problem of (partial) matching a given string str against a regexp r is then to construct
  - a parse tree p : $\mathsf{L}(r)$ and
  - a suffix sfx
  - such that $str = \mathsf{render}\ p\ sfx$,

  that is, to invert the rendering function.
- The classical recognition problem $str ? \in \mathsf{L}(r)$ then reduces to the problem of testing whether the suffix $sfx = []$

Specifying the family

$$\underline{\text{data}} \quad \frac{r \ : \ \mathsf{RE}\ A \ ; \ str \ : \ \mathsf{String}\ A}{\mathsf{Recog}\ r\ str \ : \ \star}$$

$$\underline{\text{where}} \quad \frac{p \ : \ \mathsf{L}(r) \ ; \ sfx \ : \ \mathsf{String}\ A}{\mathsf{consume}\ p\ sfx \ : \ \mathsf{Recog}\ r\ (\mathsf{render}\ p\ sfx)}$$

$$\frac{e \ : \ \mathsf{Err}\ r}{\mathsf{err}\ e \ : \ \mathsf{Recog}\ r\ \overline{e}}$$

- Writing a recogniser then amounts to writing a function recog of type

$$\forall r : \mathsf{RE}\ A.\ \forall s : \mathsf{String}\ A.\ \mathsf{Recog}\ r\ s$$

- Non-dependent elimination (the technique of *views*) over the family

  Recog r s exposes s so as to invert render

CPS revisited: why partial matching matters

- when attempting to match str against a composition $r \,.\, s$, we can recursively match the suffix sfx against s
- what holds this together is that the corresponding result types match up, because for $p : \mathsf{L}(r), q : \mathsf{L}(s)$ we have as a *definitional* equality

$$\mathsf{render}\ (p \bullet q)\ sfx = \mathsf{render}\ p\ (\mathsf{render}\ q\ sfx).$$

- similar considerations apply when matching against r^+ *etc.*
- as in Wand's influential analysis, the suffix strings encode, as usual, the continuation of the computation

The recogniser

- We define the recogniser as a function recog, declared with signature

$$\underline{\text{let}} \quad \frac{r \ : \ \mathsf{RE}\ A \ ; \ str \ : \ \mathsf{String}\ A}{\mathsf{recog}\ r\ str \ : \ \mathsf{Recog}\ r\ str}$$

- recog r str is computed as follows:
  - standardise r, yielding s : SRE A, a constructor tag indicating whether r recognises ϵ, and a proof i : $s \leq r$;
  - match using a specialised recogniser srecog, declared with signature $\underline{\text{let}} \quad \dfrac{s \ : \ \mathsf{SRE}\ A \quad str \ : \ \mathsf{String}\ A}{\mathsf{srecog}\ s\ str \ : \ \mathsf{Recog}\ |s|\ str}$
  - use the proof $\langle\langle i \rangle\rangle$ to fix up the types!

The specialised recogniser

Matching a specialised recogniser srecog, declared with signature

let $\dfrac{s \ : \ \mathsf{SRE}\ A \quad str \ : \ \mathsf{String}\ A}{\mathsf{srecog}\ s\ str \ : \ \mathsf{Recog}\ |s|\ str}$ does the obvious thing:

- fail on 0
- on a: succeed if the head character is matched; fail otherwise
- on $+$: try to match the left, try the right if you fail
- on $.$: try to match an initial segment, continue with the tail
- on s^+: try to match one copy of s. . . once you fail, return the suffix and figure out if you have succeeded or failed!

Termination relies on the fact that you must consume tokens at each success step which gives rise to a recursive call

end of part III

Conclusions

- relativisation to a given r makes $\mathsf{L}(r)$, and render , evidently "correct"
- (category-theoretic) treatment of Kleene algebra: $\mathsf{L}()$ is a *functor*
- Harper-Sethi/Berry-McNaughton/Yamada normalisation to SRE A
- well-founded recursion on suffices secures termination for regexps in standard form
- the consume constructor encodes, direct-style, (the continuation on) the suffix of the string, having successfully parsed a prefix of the string
- the matcher is itself, evidently "correct" by virtue of its type

Anti-Conclusions

- the eventual program
  - has a very high ‘deBruijn ratio’ compared to Harper’s original
  - still isn’t entirely finished. . . oops!
  - is harder to understand, . . . or is it?
- OTT might help hide all the $\langle\langle i \rangle\rangle$ equational reasoning in types

Questions?

Dependent families of types [Martin-Löf 1971]

- The key device we exploit to achieve this is the idea of a

  dependent family of types $F : T \to \star$

  a function on type $T : \star$ which returns *types* $F\,t : \star$ given $t : T$.

- allow *arbitrary* T as the domain of variation (not just $\star$ itself)

- then F behaves like a *predicate* on T

- quantification $\forall$, $\exists$ given by type constructors Π, Σ . . . so we have typed programs and logic with explicit proofs

- an important class of datatypes arise by considering *inductively-defined* F [Dybjer 1991].

A uniform generalisation of GADTs and related notions

the spectrum of possible instances $T\,\vec{a}$ occurring in source and target types of term constructors and functions:

Hindley-Milner $\vec{a}$ can be type variables only; *uniform* choice over all constructors of a datatype; function instances $T\vec{\tau}$ similarly uniform

polymorphic recursion non-uniform instances in *source* types for constructors, on a per-constructor basis

GADTs non-uniform instances in source and *target* types: $\vec{a}$ may be *arbitrary* type expressions (necessarily in type-constructor form; no type-level functions)

Ωmega $\vec{a}$ may be *arbitrary* type expressions (*not* necessarily in type-constructor form)

Inductive Families

- Dybjer's families: $\vec{a}$ may be *term* expressions (*not* necessarily types!)
- can consider further stratifications of this idea
  - only consider $\vec{a}$ to be *variables* (Cayenne)
  - . . . to be *constructor form patterns*
  - *arbitrary* expressions
- EPIGRAM makes the last, most permissive choice
- NB. these types are **not** ascribed to untyped terms
- they *prescribe*, and/or *describe* very rich properties
- don't understand data or computations *independently* of their types
- give up all partial recursive functions. . .

Phase distinction: the event horizon for type systems

There seems to be a fundamental problem with keeping static and dynamic layers apart:

> you cannot say that the thing you construct is *related* to the input you started with

For others, there seems to be a fundamental problem with mixing static and dynamic layers:

> types might (have to) get passed at run-time