Kleene meets Church: Regular expressions as types

Fritz Henglein

Department of Computer Science
University of Copenhagen
Email: henglein@diku.dk

WG 2.8 meeting, Shirahama, 2010-04-11/16

Joint work with Lasse Nielsen, DIKU
TrustCare Project (trustcare.eu)

Previous WG2.8 talks

- **Q:** Can you sort and partition generically in linear time?
- **A:** Yes.

- **Q:** What is a sorting function?
- **A:** Any intrinsically parametric permutation function.

This talk[1]

- **Q:** What is a regular expression?
- **A:** A *simple type* with suitable coercions

[1]None of this is published! Various parts of the applications are under way. But lots of theoretical and practical work remains to be done!

Most used embedded DSLs for programming

- SQL
- *Regular expressions*

Regular language

Definition (Regular language)

A *regular language* is a language (set of strings) over some finite alphabet A that is accepted by some finite automaton.

Regular expression

Definition (Regular expression)

A *regular expression (RE)* over finite alphabet A is an expression of the form

$$E, F ::= 0 \mid 1 \mid a \mid E|F \mid EF \mid E*$$

where $a \in A$ that denotes the language $\mathcal{L}[\![E]\!]$ defined by

$$\begin{array}{rcl@{\qquad}rcl} \mathcal{L}[\![0]\!] & = & \emptyset & \mathcal{L}[\![E|F]\!] & = & \mathcal{L}[\![E]\!] \cup \mathcal{L}[\![F]\!] \\ \mathcal{L}[\![1]\!] & = & \{\epsilon\} & \mathcal{L}[\![EF]\!] & = & \mathcal{L}[\![E]\!] \odot \mathcal{L}[\![F]\!] \\ \mathcal{L}[\![a]\!] & = & \{a\} & \mathcal{L}[\![E*]\!] & = & \bigcup_{i \geq 0} (\mathcal{L}[\![E]\!])^i \end{array}$$

where $S \odot T = \{s\, t \mid s \in S \wedge t \in T\}$, $E^0 = \{\epsilon\}, E^{i+1} = E\, E^i$.

Kleene's Theorem

Theorem (Kleene 1956)

A language is regular if and only it is denoted by a regular expression.

Theory: What we learn about regular expressions

- They're just a way to talk about finite state automata
- All equivalent regular expressions are interchangeable since they accept the same language.
- All equivalent automata are interchangeable since they accept the same language.
  - We might as well choose an efficient one (deterministic, minimal state): it processes its input in linear time and constant space.
- Myhill-Nerode Theorem (for proving a language regular)
- Pumping Lemma (for proving a language nonregular)
- Equivalence is decidable: PSPACE-complete.
- They are closed under complement and intersection.
- Star-height problem
- Good for specifying lexical scanners.

Practice: How regular expressions are used[3]

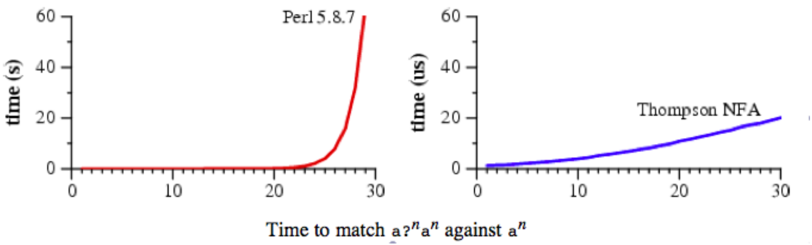
- Full (partial) matching: Does the RE occur (somewhere in) this string?
- Basic grouping: Does the RE match and where in the string?
- Grouping: Does the RE match and where do (some of) its *sub-REs* match in the string?
- Substitution: Replace matched substrings by specified other strings
- Extensions: Backreferences, look-ahead, look-behind,...
- Lazy vs. greedy matching, possessive quantifiers, atomic grouping
- Optimization[2]

---

[2] Friedl, Mastering Regular Expressions, chapter 6: *Crafting an efficient expression*

[3] in Perl and such

Optimization??



Cox (2007)

- Perl-compliant regular expressions (what you get in Perl, Python, Ruby, Java) use *backtracking parsing*.
- Does not handle E^* where E contains ϵ – will typically crash at run-time (stack overflow).

Why discrepancy between theory and practice?

- Theory is *extensional*: About regular *languages*.
  - Does this string match the regular expression? Yes or no?
- Practice is *intensional*: About regular expressions as *grammars*.
  - Does this string match the regular expression and if so *how*—which parts of the string match which parts of the RE?
- Ideally: Regular expression matching = parsing + "catamorphic" processing of syntax tree[4]
- Reality: Regular expression matching = finite automaton + opportunistic instrumentation to get *some* parsing information.

[4] Think about Shenjiang's talk

Example

```
((ab)(c|d)|(abc))*.
```

Match against $\boxed{\texttt{abdabc}}$.
For each parenthesized *group* a substring is returned.[a]

		PCRE	*POSIX*
\$1	=	*abc* or ϵ(!)	*abc* or ϵ(!)
\$2	=	*ab*	ϵ
\$3	=	*c*	ϵ
\$4	=	ϵ	*abc*

[a] Or special *null*-value

Regular expression parsing

Example

Parse `abdabc` according to `((ab)(c|d)|(abc))*`.

- $p_1 = [\mathrm{inl}\,((a, b), \mathrm{inr}\, d), \mathrm{inr}\,(a, (b, c))]$
- $p_2 = [\mathrm{inl}\,((a, b), \mathrm{inr}\, d), \mathrm{inl}\,((a, b), \mathrm{inl}\, c)]$

- p_1, p_2 have *type* $\boxed{((a \times b) \times (c + d) + a \times (b \times c))\,\mathrm{list}}$.
- Compare with *regular expression* $\boxed{\texttt{((ab)(c|d)|(abc))*}}$.
- The *elements* of *type* E correspond to the *syntax trees* for strings parsed according to *regular expression* E!

Type interpretation

Definition (Type interpretation)

The *type interpretation* $\mathcal{T}[\![.]\!]$ compositionally maps a regular expression E to the corresponding simple type:

$$
\begin{array}{rcll}
\mathcal{T}[\![0]\!] & = & \emptyset & \text{empty type} \\
\mathcal{T}[\![1]\!] & = & \{()\} & \text{unit type} \\
\mathcal{T}[\![a]\!] & = & \{a\} & \text{singleton type} \\
\mathcal{T}[\![E + F]\!] & = & \mathcal{T}[\![E]\!] + \mathcal{T}[\![F]\!] & \text{sum type} \\
\mathcal{L}[\![E \times F]\!] & = & \mathcal{T}[\![E]\!] \times \mathcal{T}[\![F]\!] & \text{product type} \\
\mathcal{T}[\![E^*]\!] & = & \{[v_1, \ldots, v_n] \mid v_i \in \mathcal{T}[\![E]\!]\} & \text{list type}
\end{array}
$$

Flattening

Definition

The *flattening* function $\mathrm{flat}(.) : \mathrm{Val}(\mathcal{A}) \to \mathrm{Seq}(\mathcal{A})$ is defined as follows:

$$
\begin{array}{rclrcl}
\mathrm{flat}(()) & = & \epsilon & \mathrm{flat}(a) & = & a \\
\mathrm{flat}(\mathrm{inl}\, v) & = & \mathrm{flat}(v) & \mathrm{flat}(\mathrm{inr}\, w) & = & \mathrm{flat}(w)
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{flat}((v, w)) & = & \mathrm{flat}(v)\, \mathrm{flat}(w) \\
\mathrm{flat}([v_1, \ldots, v_n]) & = & \mathrm{flat}(v_1) \ldots \mathrm{flat}(v_n)
\end{array}
$$

Example

$$
\begin{array}{rcl}
\mathrm{flat}([\mathrm{inl}\, ((a, b), \mathrm{inr}\, d), \mathrm{inr}\, (a, (b, c))]) & = & abdabc \\
\mathrm{flat}([\mathrm{inl}\, ((a, b), \mathrm{inr}\, d), \mathrm{inl}\, ((a, b), \mathrm{inl}\, c)]) & = & abdabc
\end{array}
$$

Regular expressions as types

Informally:

string s with syntax tree p according to *regular expression* E $\cong$ string $\mathrm{flat}(v)$ of value v element of *simple type* E

Theorem

$$\mathcal{L}[\![E]\!] = \{\mathrm{flat}(v) \mid v \in \mathcal{T}[\![E]\!]\}$$

Membership testing versus parsing

Example

E = `((ab)(c|d)|(abc))*` E_d = `(ab(c|d))*`

- E_d is *unambiguous*: If $v, w \in \mathcal{T}[\![E_d]\!]$ and $\mathrm{flat}(v) = \mathrm{flat}(w)$ then $v = w$. (Each string in E_d has exactly one syntax tree.)
- E is *ambiguous*. (Recall p_1 and p_2.)
- E and E_d are *equivalent*: $\mathcal{L}[\![E]\!] = \mathcal{L}[\![E_d]\!]$
- E_d "represents" the minimal deterministic finite automaton for E.
- Matching (membership testing): Easy—use E_d.
- But: How to parse *according to E* using E_d?

Regular expression equivalence and containment

Sometimes we are interested in regular expression containment or equivalence.[5]

Definition

- E is *contained* in F if $\mathcal{L}[\![E]\!] \subseteq \mathcal{L}[\![F]\!]$.
- E is *equivalent* to F if $\mathcal{L}[\![E]\!] = \mathcal{L}[\![F]\!]$.

Regular expression equivalence and containment are easily related: $E \leq F \Leftrightarrow E + F = F$ and $E = F \Leftrightarrow (E \leq F \wedge F \leq E)$.

[5] See e.g. Yasuhiko's talk.

Coercion

Definition (Coercion)

Partial coercion: Function $f : \mathcal{T}[\![E]\!] \to \mathcal{T}[\![F]\!]_\perp$ such that $f(v) = \perp$ or $\mathrm{flat}(v) = \mathrm{flat}(f(v))$.

Coercion: Function $f : \mathcal{T}[\![E]\!] \to \mathcal{T}[\![F]\!]$ such that $\mathrm{flat}(v) = \mathrm{flat}(f(v))$.

Intuition:

- A coercion is a *syntax tree transformer*.
- It maps a *syntax tree* under regular expression E to a syntax tree under regular expression F for *same* string.

Example

$$f : ((a \times b) \times (c + d) + a \times (b \times c))\,\mathrm{list} \to (a \times (b \times (c + d)))\,\mathrm{list}$$

$$\begin{array}{lcl} f([\,]) & = & [\,] \\ f(\mathrm{inl}\,((x, y), z) :: l) & = & (x, (y, z)) :: f(l) \\ f(\mathrm{inr}\,(x, (y, z)) :: l) & = & (x, (y, \mathrm{inl}\,z)) :: f(l) \end{array}$$

- $\mathrm{flat}(f(v)) = \mathrm{flat}(v)$ for all $v : ((a \times b) \times (c + d) + a \times (b \times c))\,\mathrm{list}$.
- So f defines a *coercion* from E = `((ab)(c|d)|(abc))*` to E_d = `(ab(c|d))*`.
- f maps each *proof of membership* ($=$ syntax tree) of a string s in regular language $\mathcal{L}[\![E]\!]$ to a proof of membership of string s in regular language $\mathcal{L}[\![E]\!]$.
- So f is a *constructive proof* that $\mathcal{L}[\![E]\!]$ is *contained* in $\mathcal{L}[\![F]\!]$!

Regular expression containment by coercion

Proposition

$$\mathcal{L}[\![E]\!] \subseteq \mathcal{L}[\![F]\!]$$

if and only if

there exists a coercion from $\mathcal{T}[\![E]\!]$ *to* $\mathcal{T}[\![F]\!]$.

Idea:

- Come up with a sound and complete inference system for proving regular expression containments.
- Interpret it as a language for definining *coercions*:
  - Soundness: Each proof term defines a coercion.
  - Completeness: For each valid regular expression containment there is at least one proof term.

A crash course on regular expression containment

- All classical *sound and complete axiomatizations* basically start with the axioms for *idempotent semirings*.
- Then they add various inference rules to capture the semantics of Kleene star.
- *Algorithms* for *deciding* containment are "coinductive" in nature:
  - transformation to automata or
  - regular expression containment rewriting
- The algorithms have little to do with the axiomatizations!
  - They do not produce a proof (derivation)
  - They cannot be thought of proof search in an axiomatization.

Our approach

Idea:

- Axiomatization =
  Idempotent semiring
  + finitary unrolling for Kleene-star
  + general coinduction rule (for completeness)
  - restriction on coinduction rule (for soundness)
- Each rule can be interpreted as natural *coercion constructor*.
- Algorithms for deciding containment can be thought of as strategies for proof search. They yield coercions, not just decisions (yes/no).

Idempotent semiring axioms

Proviso: $+$ for alternation, $\times$ for concatenation, $*$ for Kleene-star.

$$
\begin{aligned}
E + (F + G) &= (E + F) + G \\
E + F &= F + E \\
E + 0 &= E \\
E + E &= E \\
E \times (F \times G) &= (E \times F) \times G \\
1 \times E &= E \\
E \times 1 &= E \\
E \times (F + G) &= (E \times F) + (E \times G) \\
(E + F) \times G &= (E \times G) + (F \times G) \\
0 \times E &= 0 \\
E \times 0 &= 0
\end{aligned}
$$

Kleene-star

Finitary unrolling:

$$E^* \quad = \quad 1 + E \times E^*$$

General coinduction rule:

$$\frac{\begin{array}{c}[E = F] \\ \cdots \\ E = F\end{array}}{E = F}$$

- Fantastically powerful rule!
- Unfortunately unsound
- But “right idea” – just needs controlling.

Type-theoretic formulation: Idempotent semiring

With explicit proof terms, using judgement form (due to dispatch in coinduction rule) and containment instead of equivalence:

$$\Gamma \vdash \text{shuffle} : E + (F + G) \leq (E + F) + G$$
$$\Gamma \vdash \text{shuffle}^{-1} : E + (F + G) \leq (E + F) + G$$

$$\Gamma \vdash \text{retag} : E + F \leq F + E$$

$$\Gamma \vdash \text{untag} : E + E \leq E$$
$$\Gamma \vdash \text{tagL} : E \leq E + F$$

$$\dots$$
$$\Gamma \vdash \text{proj} : E \times 1 \leq E$$
$$\Gamma \vdash \text{proj}^{-1} : E \leq E \times 1$$

$$\Gamma \vdash \text{distL} : E \times (F + G) \leq (E \times F) + (E \times G)$$
$$\Gamma \vdash \text{distL}^{-1} : (E \times F) + (E \times G) \leq E \times (F + G)$$
$$\dots$$

Primitive coercions

- Each axiom can be interpreted as a *coercion*; e.g.,

$$\begin{aligned}
\mathrm{shuffle}(\mathrm{inl}\, x) &= \mathrm{inl}\,(\mathrm{inl}\, x) \\
\mathrm{shuffle}(\mathrm{inr}\,(\mathrm{inl}\, y)) &= \mathrm{inl}\,(\mathrm{inr}\, y) \\
\mathrm{shuffle}(\mathrm{inr}\,(\mathrm{inr}\, z)) &= \mathrm{inr}\, z
\end{aligned}$$

- The (p, p^{-1}) pairs denote type isomorphisms: $p \circ p^{-1} = \mathrm{id}$ and $p^{-1} \circ p = \mathrm{id}$.
- $(\mathrm{tagL}\,, \mathrm{untag}\,)$ is an *embedding-projection* pair, but *not* an isomorphism even for $E \equiv F$: $\mathrm{untag}\, \circ \mathrm{tagL}\, = \mathrm{id}$, but $\mathrm{tagL}\, \circ \mathrm{untag}\, \neq \mathrm{id}$.

Type-theoretic formulation: Kleene-star, coinduction

$$\Gamma \vdash \mathrm{wrap} : 1 + E \times E^* \leq E^*$$
$$\Gamma \vdash \mathrm{wrap}^{-1} : E^* \leq 1 + E \times E^*$$

$$\frac{\Gamma, f : E \leq F \vdash c : E \leq F}{\Gamma \vdash \mathrm{fix} f.c : E \leq F} \quad (Sx)$$

- Interpret $(\mathrm{wrap}, \mathrm{wrap}^{-1})$ as isomorphism in accordance with isorecursive interpretation of lists.
- Interpret fix as *least fixed point operator*; that is, as *recursively* defined coercion: $\mathrm{fix} = Y(\lambda f.c)$.
- Add side-condition (Sx) that ensures that recursively defined coercions *terminate*.

The mother of all side conditions

Definition

Coercion c in $\Gamma \vdash c : E \leq F$ is *hereditarily total* if whenever its free variables are bound to (total!) coercions then it denotes a (total!) coercion.

Side condition $S1$ (Total): $\boxed{\mathrm{fix} f.c \text{ is hereditarily total}}$

Proposition

It is decidable whether $\Gamma \vdash c : E \leq F$ is hereditarily total.

Other side conditions

Definition

(Informally) Coercion c is *guarded* if all fix-bound variable occurrences are guarded by $\times$ and no proj^{-1} is applied before recursive calls.

Side condition $S2$ (Guarded): $\boxed{\mathrm{fix} f.c \text{ is } \textit{guarded}}$

Side condition $S3$ (constant guarded):

$$\boxed{\mathrm{fix} f.c \text{ has the form } \mathrm{fix} f.a_1 \times c_1 + \ldots + a_n \times c_n}$$

if $A = \{a_1, \ldots, a_n\}$.

Side condition $S4$: ...

Soundness and completeness

Theorem

For any of the side conditions Sx:

$$\mathcal{L}[\![E]\!] \subseteq \mathcal{L}[\![F]\!]$$
if and only if
there exists c *such that* $\vdash c : E \leq F$

So what?

Summary so far:

- A regular expression denotes a *type* ("regular type").
- A proof of regular expression containment denotes a coercion from one regular expression interpreted as a type to the other.

What good is this?

Applications[6]

1. Parametric completeness
2. Coercion synthesis
3. Oracle coding
4. Fast parsing
5. Ambiguity resolution
6. Regular expressions as refinement types for strings

[6] Disclaimer: Some checked work, much belief, everything informal from now on

Parametric completeness

Our side conditions ($S1$ and $S2$) are essentially different from previous axiomatizations:

- No insistence on "no empty word" property.
- Instead control application of proj^{-1}.

Theorem

Assume $\mathcal{L}[\![E[G/X]]\!] \subseteq \mathcal{L}[\![F[G/X]]\!]$ for all RE G where E, F contain a regular expression variable X. Then there exists a parametrically polymorphic coercion c such that $\vdash c : \forall X.E[X] \leq F[X]$.

This does *not* hold of Salomaa (1966) and Grabmeyer (2005). They only work for "closed" regular expressions. (Kozen's axiomatization seems to be parametrically complete in the same sense.)

Parametric completeness

The theorem holds if A is infinite or there exists at least one $a \in A$ that does *not* occur in E or F.

Open problem

Find a parametrically complete axiomatization for finite A and all E, F.

Open problem

Consider functions typed in a substructural version of System F: linear, no commutativity of assumptions; alphabet symbols modeled by quantified type variables; lists Church-coded. Does this yield only coercions? All of them? (And what does "all" mean?)

Coercion synthesis

Our axiomatization under $S1$ (and as far as we have seen practically also for $S2$) admits "many" coercions terms. It appears to contain *practically more efficient* ones than what is derivable in other axiomatizations.

Think of coercion synthesis as a functional programming problem.

Example

Prove that $\models (G+1)^* \leq G^*$ for all G.

Approach: Find list function of type $\forall\alpha.(\alpha+1)\,\mathrm{list} \to \alpha\,\mathrm{list}$. Make sure you haven't permuted, discarded or duplicated input elements.

$$\begin{aligned} f([\,]) &= [\,] \\ f(\mathrm{inl}\, x :: l) &= x :: f(l) \\ f(\mathrm{inr}\, () :: l) &= f(l) \end{aligned}$$

Try to find a proof of $\models (G+1)^* \leq G^*$ in Kozen's axiomatization!

Oracle coding (bit-coding)

Recall syntax trees p_1, p_2 for *abdabc* under $E = ((a \times b) \times (c + d) + a \times (b \times c))^*$.

- $p_1 = [\mathrm{inl}\,((a, b), \mathrm{inr}\, d), \mathrm{inr}\,(a, (b, c))]$
- $p_2 = [\mathrm{inl}\,((a, b), \mathrm{inr}\, d), \mathrm{inl}\,((a, b), \mathrm{inl}\, c)]$

We can *code* them by storing *only* their inl, inr occurrences:

$$\begin{aligned} \mathrm{code}(p_1) &= 011 \\ \mathrm{code}(p_2) &= 0100 \end{aligned}$$

There is a *type-directed* function decode that can reconstitute the syntax trees:

$$\begin{aligned} \mathrm{decode}_E(011) &= [\mathrm{inl}\,((a, b), \mathrm{inr}\, d), \mathrm{inr}\,(a, (b, c))] \\ \mathrm{decode}_E(0100) &= [\mathrm{inl}\,((a, b), \mathrm{inr}\, d), \mathrm{inl}\,((a, b), \mathrm{inl}\, c)] \end{aligned}$$

Oracle coding (bit-coding)

- Oracle coding combines *orthogonally* with ordinary string compression: Compression of bitcoded syntax trees can be substantially better than compression of the string.
- Coercion judgements can be interpreted directly into bit string transformations without explicit application of $\mathrm{code}, \mathrm{decode}$; e.g.

$$\begin{aligned} \mathrm{retag}(0d) &= 1d \\ \mathrm{retag}(1d) &= 0d \\ \mathrm{assoc}(d) &= d \end{aligned}$$

- For coding purposes it is better to use *right-regular grammars* as a formalism for regular expressions.

Ambiguity resolution

- All regular expression equivalences yield coercion isomorphisms, except for one: $(\mathrm{tagL}, \mathrm{untag}) : E = E + E$.
- This is where ambiguity is introduced/eliminated! Always choosing tagL (from left to right) favors the *left* alternative, as in Perl.
- Eager matching seems to correspond to choosing the *right* alternative in $E^* = 1 + E \times E*$; lazy matching to choosing the *left* alternative.

Open problem

Design an expressive annotation for regular expressions that specifies a choice function for deterministically choosing one of potentially multiple syntax trees for a string and that can (at a minimum) express POSIX and PCRE rules.

Fast parsing

Recall E = `((ab)(c|d)|(abc))*` E_d = `(ab(c|d))*`.
Perform fast parsing as follows:

1. Construct $c : E_d \leq E$ (with suitable ambiguity resolution principle applied in c)
2. Use deterministic automaton for E_d to build a syntax tree for input string in linear time.
3. Apply c to the syntax tree.
4. Generate and operate on *bit-coded* representation of syntax trees.

Implemented by Brabrand/Thomsen (2010, unpublished). Dube et al. (2000-) and Frisch/Cardelli (2004) seem to be doing something that can be understood as the above. (They do not operate on bit codes, however.)

Regular expressions as refinement types for strings

- Add regular expressions as refinement types
- They're already there: Regular types! What needs to be added is coercion synthesis ($\sim$ deciding regular expression containment).
- Use bit coding for run-time representations and bit-coded coercions for bit transformations.

Open problem

Polymorphic regular type and coercion inference.

Related to Hosoya/Frisch/Castagna (2005), which is for regular *expression* types, however.

Related work

- Frisch, Cardelli (2004): Regular *types* corresponding to regular expressions, linear-time parsing for REs;
- Hosoya et al. (2000-): Regular expression types, *proper* extension of regular types (!), axiomatization of tree containment
- Aanderaa (1965), Salomaa (1966), Krob (1990), Pratt (1990), Kozen (1994, 2008), Grabmeyer (2005), Rutten et al. (2008): RE axiomatizations (extensional)
- Rutten et al. (1998-): Coalgebraic approach to systems, including finite automata, *extensional*—does not distinguish between equivalent REs (important for parsing)
- Brandt/Henglein (1998): Coinduction rule and computational interpretation for recursive types
- Necula/Rahul (2001): Oracle coding in PCC
- Cox (2010): RE2 regular expression library

Future work

- Projection/substitution: efficient composition of parsing, containment (coercions) and catamorphic postprocessing.
- Build a PCRE- and RE2-killer library.

Summary

- Regular expressions denote *types*, not languages, when used grammatically. Apart from singletons no special type constructions are needed – they're already present in a typed programming language.
- Regular expression containment proofs denote *coercions*, not just yes/no answers (with or without logical certificate).
- Sound and complete axiomatization with computational interpretation of proofs as coercions.
- Applications for *regular expressions as types*: Parsing (not just membership testing), bit coding, fast parsing, parametricity, ambiguity resolution, refinement type system for strings.