Regular Expression Containment: Coinductive Axiomatization and Computational Interpretation
At POPL 2011

Lasse Nielsen
Joint work with Fritz Henglein

University of Copenhagen (DIKU)

January 27, 2011
What is Regular Expression Matching?

What is the result of \texttt{match } \left( (ab)(c + d) + (abc) \right)^* \texttt{"abdabc"}?

- Faster regular expression matching [Bille, Thorup 2009]
  
  Model: \( \mathcal{L}[a^*] = \{ "", "a", "aa", \ldots \} \).
  
  Answer: Yes / No

- PERL/POSIX
  
  Answer: One substring for each set of parentheses

- XDuce [2000] and CDuce [2003]
  
  Answer: List of bindings

- Greedy Regular Expression Matching [Frisch, Cardelli 2004]
  
  Answer: Parse tree
What is Regular Expression Matching?

What is the result of \texttt{match \((ab)(c + d) + (abc))^* "abdabc"}\texttt{?}

- Faster regular expression matching [Bille, Thorup 2009]
  Model: \( \mathcal{L}[a^*] = \{ "", "a", "aa", \ldots \} \).
  \textbf{Yes}

- PERL/POSIX
  Answer: One substring for each set of parentheses

- XDuce [2000] and CDuce [2003]
  Answer: List of bindings

- Greedy Regular Expression Matching [Frisch, Cardelli 2004]
  Answer: Parse tree
What is Regular Expression Matching?

What is the result of match \(((ab)(c + d) + (abc))\ast \text{"abdabc"}\)?

- Faster regular expression matching [Bille, Thorup 2009]
  
  Model: \(\mathcal{L}[a^\ast] = \{\text{""}, \text{"a"}, \text{"aa"}, \ldots\}\).

  **Yes**

- PERL/POSIX
  
  \[\text{"abc"}, \text{"ab"}, \text{"c"}, \text{""}] / [\text{""}, \text{""}, \text{""}, \text{"abc"}]\] (Ambiguous)

- XDuce [2000] and CDuce [2003]
  
  Answer: List of bindings

- Greedy Regular Expression Matching [Frisch, Cardelli 2004]
  
  Answer: Parse tree
What is Regular Expression Matching?

What is the result of \texttt{match (}(ab)(c + d) + (abc))\texttt{*} "abdabc"?

- Faster regular expression matching [Bille, Thorup 2009]
  Model: \( \mathcal{L}[a^*] = \{ "", "a", "aa", \ldots \} \).
  \textbf{Yes}

- \texttt{PERL/POSIX}
  \[ ["abc", "ab", "c", "]/ ["", "]/ ["", "]/ ["", "abc"] \]
  (Ambiguous)

- \texttt{XDuce [2000] and CDuce [2003]}
  \[ [(1, "abd"), (2, "ab", (3, "d")), (1, "abc"), (2, "ab", (3, "c"))] \]
  (Ambiguous)

- \texttt{Greedy Regular Expression Matching [Frisch, Cardelli 2004]}
  Answer: Parse tree
Regular Expressions - Parsing

- Regular expression as grammar: Matching returns parse tree (or error)

\[ E_1 = ((ab)(c + d) + (abc)) \]

parse \( E_1^* \) "abdabc" →

```
cons → cons → nil
  ↓    ↓
  inl  inl
  ↓    ↓
  pair  pair
  ↓    ↓
  pair  pair
  ↓    ↓
  inr  inl
  ↓    ↓
  a    b    d    a    b    c
```
Regular Expressions - Parsing

- Regular expression as grammar: Matching returns parse tree (or error)
  \[ E_1 = ((ab)(c + d) + (abc)) \]
  parse \( E_1^* \) "abdabc" →

- Diagram:
  \[
  \begin{array}{c}
  \text{cons} \rightarrow \text{cons} \rightarrow \text{nil} \\
  \downarrow \quad \downarrow \quad \downarrow \\
  \text{inl} \quad \text{inl} \\
  \downarrow \quad \downarrow \\
  \text{pair} \quad \text{pair} \\
  \downarrow \quad \downarrow \\
  \text{inr} \quad \text{inl} \\
  \downarrow \quad \downarrow \\
  a \quad b \\
  \downarrow \quad \downarrow \\
  d \\
  \end{array}
  \]
  or
  \[
  \begin{array}{c}
  \text{cons} \rightarrow \text{cons} \rightarrow \text{nil} \\
  \downarrow \quad \downarrow \quad \downarrow \\
  \text{inl} \quad \text{inr} \\
  \downarrow \quad \downarrow \\
  \text{pair} \quad \text{pair} \\
  \downarrow \quad \downarrow \\
  \text{inr} \quad \text{inl} \\
  \downarrow \quad \downarrow \\
  a \quad b \\
  \downarrow \quad \downarrow \\
  d \quad c \\
  \end{array}
  \]
Regular Expressions - Parsing

- Regular expression as grammar: Matching returns parse tree (or error)
  \[ E_1 = ((ab)(c + d) + (abc)) \]
  parse \( E_1^* "abdabc" \) →

```
cons → cons → nil
  ↓   ↓
inl   inl
  ↓   ↓
pair  pair
  ↓   ↓
pair  inr  pair  inl
  ↓   ↓   ↓   ↓
a   b   d   a   b   c
```
Regular Expressions - Parsing

- Regular expression as grammar: Matching returns parse tree (or error)
  \[ E_1 = ((ab)(c + d) + (abc)) \]
  parse \( E_1^* "abdabc" \) →

```
cons: E_1^*  →  cons: E_1^*  →  nil: E_1^*
   ↓         ↓         ↓
  inl: E_1    inl: E_1    
         ↓         ↓
  pair: (ab)(c + d) pair: (ab)(c + d)
               ↓               ↓
  pair: ab inr: c + d pair: ab inl: c + d
                     ↓                     ↓
                   a : a                   a : a
                   b : b                   b : b
                   d : d                   c : c
```
Regular Expressions - Parsing

- Regular expression as grammar: Matching returns parse tree (or error)
  \[ E_1 = ((ab)(c + d) + (abc)) \]
  `parse E_1^* "abdabc"` →

```
fold (inr pair): E_1^*  →  fold (inr pair): E_1^*  →  fold (inl ()): E_1^*
  ↓                        ↓                        ↓
  inl: E_1                  inl: E_1
  ↓                        ↓                        ↓
  pair: (ab)(c + d)        pair: (ab)(c + d)
  ↓                        ↓                        ↓
  pair: ab                 pair: ab
  ↓                        ↓                        ↓
  a : a                    a : a
  b : b                    b : b
  d : d                    d : d
  inr: c + d               inr: c + d
  ↓                        ↓                        ↓
  a : a                    a : a
  b : b                    b : b
  c : c                    c : c
```
Regular Expressions - Parsing

- Regular expression as grammar: Matching returns parse tree (or error)

\[ E_1 = ((ab)(c + d) + (abc)) \]

\[
\text{parse } E_1^* \text{ "abdabc" } \rightarrow \\
\text{fold (inr (inl ((a, b), inr d), fold (inr (inl ((a, b), inl c), fold (inl ()))))))} \\
\text{fold (inr pair): } E_1^* \rightarrow \text{fold (inr pair): } E_1^* \rightarrow \text{fold (inl ()): } E_1^*
\]

```
gof (inr pair): E_1^* \\
  inl: E_1 \\
  inl: E_1 \\
  pair: (ab)(c + d) \\
  pair: (ab)(c + d) \\
  pair: ab \\
  pair: ab \\
  inr: c + d \\
  inr: c + d \\
  a: a \\
  a: a \\
  b: b \\
  b: b \\
  d: d \\
  d: d \\
  c: c \\
```
Parse trees = Regular expression as type

- Language of expressions $\text{Reg}_A$:
  \[
  E, F, G, H ::= 0 \mid 1 \mid a \mid E + F \mid E \times F \mid E^* 
  \]

- Type interpretation $\mathcal{T}[E]$:
  \[
  \mathcal{T}[0]=\emptyset \\
  \mathcal{T}[1]=\{()\} \\
  \mathcal{T}[a]=\{a\} \\
  \mathcal{T}[E + F]=\mathcal{T}[E] + \mathcal{T}[F] \\
  \mathcal{T}[E \times F]=\mathcal{T}[E] \times \mathcal{T}[F] \\
  \mathcal{T}[E^*]=\{[v_1, \ldots , v_n] \mid n \geq 0 \land v_i \in \mathcal{T}[E]\} = \mathcal{T}[E] \text{ list}
  \]

where $S + T = \{\text{inl } v \mid v \in S\} \cup \{\text{inr } w \mid w \in T\}$,

$[v_1, \ldots , v_n] = v_1 :: \ldots :: v_n :: []$,

$[] = \text{fold (inl ())}$ and $v_1 :: v = \text{fold (inr (v_1, v))}$. 
Value Flattening

- Value = element of type = parse tree
- Flattening (unparsing): Mapping value to string

\[
\begin{align*}
\text{flat}(()) & = \varepsilon \\
\text{flat}(\text{inl } v) & = \text{flat}(v) \\
\text{flat}(\text{inr } w) & = \text{flat}(w) \\
\text{flat}((v, w)) & = \text{flat}(v) \text{flat}(w) \\
\text{flat}((\text{fold } v)) & = \text{flat}(v)
\end{align*}
\]

Theorem: \( L[|E|] = \{\text{flat}(v) \mid v \in T[|E|]\} \)
Regular Expression Containment

- Containment: $\models E \leq F$ iff $\mathcal{L}[E] \subseteq \mathcal{L}[F]$
- Equivalence: $\models E = F$ iff $\mathcal{L}[E] = \mathcal{L}[F]$
Regular Expression Containment

- Containment: \( \models E \leq F \) iff \( \mathcal{L}[E] \subseteq \mathcal{L}[F] \)
- Equivalence: \( \models E = F \) iff \( \mathcal{L}[E] = \mathcal{L}[F] \)
- Containment and Equivalence are interdefinable
  \( \models E \leq F \) iff \( \models E + F = F \)
  \( \models E = F \) iff \( \models E \leq F \) and \( \models F \leq E \).
Regular Expression Containment

- Containment: $\models E \leq F$ iff $\mathcal{L}[E] \subseteq \mathcal{L}[F]
- Equivalence: $\models E = F$ iff $\mathcal{L}[E] = \mathcal{L}[F]
- Containment and Equivalence are interdefinable:
  $\models E \leq F$ iff $\models E + F = F$
  $\models E = F$ iff $\models E \leq F$ and $\models F \leq E$.
- Existing Axiomatizations
  [Salomaa 66, Kozen 94, Grabmayer 2005, and others]
Regular Expression Containment

- Containment: \( \models E \leq F \) iff \( \mathcal{L}[E] \subseteq \mathcal{L}[F] \)
- Equivalence: \( \models E = F \) iff \( \mathcal{L}[E] = \mathcal{L}[F] \)
- Containment and Equivalence are interdefinable
  \( \models E \leq F \) iff \( \models E + F = F \)
  \( \models E = F \) iff \( \models E \leq F \) and \( \models F \leq E \).

- Existing Axiomatizations
  [Salomaa 66, Kozen 94, Grabmayer 2005, and others]
  \( \models E \leq F \) iff \( \mathcal{L}[E] \subseteq \mathcal{L}[F] \)
  iff \( \{ \text{flat}(v) \mid v \in T[E] \} \subseteq \{ \text{flat}(w) \mid w \in T[F] \} \)
  iff \( \forall v \in T[E]. \exists w \in T[F]. \text{flat}(v) = \text{flat}(w) \)
Regular Expression Containment

- Containment: \( \models E \leq F \) iff \( \mathcal{L}[E] \subseteq \mathcal{L}[F] \)
- Equivalence: \( \models E = F \) iff \( \mathcal{L}[E] = \mathcal{L}[F] \)
- Containment and Equivalence are interdefinable
  \( \models E \leq F \) iff \( \models E + F = F \)
  \( \models E = F \) iff \( \models E \leq F \) and \( \models F \leq E \).
- Existing Axiomatizations
  [Salomaa 66, Kozen 94, Grabmayer 2005, and others]
  \( \models E \leq F \) iff \( \mathcal{L}[E] \subseteq \mathcal{L}[F] \)
  iff \( \{ \text{flat}(v) \mid v \in T[E] \} \subseteq \{ \text{flat}(w) \mid w \in T[F] \} \)
  iff \( \forall v \in T[E]. \exists w \in T[F]. \text{flat}(v) = \text{flat}(w) \)
- We call a string-preserving function from \( T[E] \) to \( T[F] \) a coercion from \( E \) to \( F \)
Regular Expression Containment

- Containment: $\models E \leq F$ if $\mathcal{L}[E] \subseteq \mathcal{L}[F]$
- Equivalence: $\models E = F$ if $\mathcal{L}[E] = \mathcal{L}[F]$
- Containment and Equivalence are interdefinable
  $\models E \leq F$ if $\models E + F = F$
  $\models E = F$ if $\models E \leq F$ and $\models F \leq E$.
- Existing Axiomatizations
  [Salomaa 66, Kozen 94, Grabmayer 2005, and others]
  $\models E \leq F$ if $\mathcal{L}[E] \subseteq \mathcal{L}[F]$
  iff $\{\text{flat}(v) \mid v \in \mathcal{T}[E]\} \subseteq \{\text{flat}(w) \mid w \in \mathcal{T}[F]\}$
  iff $\forall v \in \mathcal{T}[E]. \exists w \in \mathcal{T}[F]. \text{flat}(v) = \text{flat}(w)$
- We call a string-preserving function from $\mathcal{T}[E]$ to $\mathcal{T}[F]$ a coercion from $E$ to $F$
- **Theorem:** $\models E \leq F$ if there exists a coercion from $E$ to $F$
Regular Expression Containment

- **Containment:** \( \models E \leq F \) iff \( \mathcal{L}[E] \subseteq \mathcal{L}[F] \)
- **Equivalence:** \( \models E = F \) iff \( \mathcal{L}[E] = \mathcal{L}[F] \)
- **Containment and Equivalence are interdefinable**
  \( \models E \leq F \) iff \( \models E + F = F \)
  \( \models E = F \) iff \( \models E \leq F \) and \( \models F \leq E \).
- **Existing Axiomatizations**
  [Salomaa 66, Kozen 94, Grabmayer 2005, and others]
- **Containment:** \( \models E \leq F \) iff \( \mathcal{L}[E] \subseteq \mathcal{L}[F] \)
  iff \( \{ \text{flat}(v) \mid v \in T[E] \} \subseteq \{ \text{flat}(w) \mid w \in T[F] \} \)
  iff \( \forall v \in T[E]. \exists w \in T[F]. \text{flat}(v) = \text{flat}(w) \)
- **We call a string-preserving function from** \( T[E] \) **to** \( T[F] \)
a coercion from \( E \) **to** \( F \)
- **Theorem:** \( \models E \leq F \) iff there exists a coercion from \( E \) **to** \( F \)
- **Suggests "proof of containment by functional programming"**
Example: Proof by functional programming

- Problem: Kozen’s denesting rule: \((a + b)^* = a^* \times (ba^*)^*\)
- Proof by functional programming:
  
  Find \(f: (\text{'a }+ \text{'b}) \text{ list } \rightarrow \text{'a list } \times (\text{'b }\times \text{'a list}) \text{ list}\)
  
  such that \(f\) does not discard, duplicate or reorder its input

\[
\begin{align*}
  f([ ]) & = ([ ], [ ]) \\
  f(\text{inl } u::zs) & = \text{let } (xs, ys) = f(zs) \text{ in } (u::xs, ys) \\
  f(\text{inr } v::zs) & = \text{let } (xs, ys) = f(zs) \text{ in } ([ ], (v, xs)::ys)
\end{align*}
\]

- \(f\) terminates since it is called recursively with smaller sized arguments
- \(f\) is string-preserving
- Therefore \(f\) proves \(\models (a + b)^* \leq a^* \times (b \times a^*)^*\)
- Find a complete DSL where soundness is built in
Axiomatization: Weak Equivalence

Idempotent Semiring:

Equality:

\[ E + (F + G) = (E + F) + G \]
\[ E + F = F + E \]
\[ E + 0 = E \]
\[ E + E = E \]
\[ E \times (F \times G) = (E \times F) \times G \]
\[ 1 \times E = E \]
\[ E \times 1 = E \]
\[ E \times (F + G) = (E \times F) + (E \times G) \]
\[ (E + F) \times G = (E \times G) + (F \times G) \]
\[ 0 \times E = 0 \]
\[ E \times 0 = 0 \]

Kleene-star Fold/Unfold: \[ E^* = 1 + E \times E^* \]
Axiomatization: Equivalence

Salomaa’s rules for axiomatization $F_1$

\[
\begin{align*}
E &= F \\
E^* &= F^* \\
E &= F \times E + G \\
E &= F^* \times G \\
E^* &= (1 + E)^*
\end{align*}
\]

(Kozen’s rules for axiomatization of Kleene Algebras)

\[
\begin{align*}
E \times F &\leq F \\
E^* \times F &\leq F \\
E \times F &\leq F \\
E \times F^* &\leq E
\end{align*}
\]

Grabmayer’s coinduction rule COMP/FIX

\[
\begin{align*}
[E = F] & \\
\vdots & \\
E_{a_1} = F_{a_1} & \quad E_{a_n} = F_{a_n} \\
E &= F \\
( o(E) = o(F) )
\end{align*}
\]
Our axiomatization: Weak containment

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>shuffle</td>
<td>( E + (F + G) = (E + F) + G )</td>
</tr>
<tr>
<td>retag</td>
<td>( E + F = F + E )</td>
</tr>
<tr>
<td>untagL</td>
<td>( 0 + F = F )</td>
</tr>
<tr>
<td>untag</td>
<td>( E + E \leq E )</td>
</tr>
<tr>
<td>tagL</td>
<td>( E \leq E + F )</td>
</tr>
<tr>
<td>assoc</td>
<td>( E \times (F \times G) = (E \times F) \times G )</td>
</tr>
<tr>
<td>swap</td>
<td>( E \times 1 = 1 \times E )</td>
</tr>
<tr>
<td>proj</td>
<td>( 1 \times E = E )</td>
</tr>
<tr>
<td>abortR</td>
<td>( E \times 0 = 0 )</td>
</tr>
<tr>
<td>abortL</td>
<td>( 0 \times E = 0 )</td>
</tr>
<tr>
<td>distL</td>
<td>( E \times (F + G) = (E \times F) + (E \times G) )</td>
</tr>
<tr>
<td>distR</td>
<td>( (E + F) \times G = (E \times G) + (F \times G) )</td>
</tr>
<tr>
<td>wrap</td>
<td>( 1 + E \times E^* = E^* )</td>
</tr>
<tr>
<td>id</td>
<td>( E = E )</td>
</tr>
</tbody>
</table>

\( c : E \leq E' \quad d : E' \leq E'' \)

\( c + d : E + F \leq E' + F' \)

\( c : E \leq E' \quad d : F \leq F' \)

\( c \times d : E \times F \leq E' \times F' \)
Finitary coinduction principle

- Adding a general coinduction rule

\[
\begin{align*}
\vdash & \quad E \leq F \\
\vdash & \quad E \leq F \\
\vdash & \quad E \leq F \\
\vdash & \quad E \leq F \\
\vdash & \quad E \leq F \\
\vdash & \quad E \leq F
\end{align*}
\]

(coinduction rule)
Finitary coinduction principle

- Adding a general coinduction rule

\[
\begin{align*}
  [f : E \leq F] \\
  \vdots \\
  c : E \leq F \\
  \text{fix}_f c : E \leq F
\end{align*}
\]  
(coinduction rule)
Finitary coinduction principle

- Adding a general coinduction rule
  \[ [f : E \leq F] \]
  \[ \vdots \]
  \[ c : E \leq F \]
  \[ \text{fix} f . c : E \leq F \]  (coinduction rule)

- Unsound without constraint: \( \text{fix} f . f : E \leq F \)

- But right idea
Computational interpretation of proof terms

- Each axiom denotes a coercion:

\[
\text{shuffle} : E + (F + G) \leq (E + F) + G
\]

\[
\text{shuffle}(\text{inl } \nu) = \text{inl}(\text{inl } \nu)
\]

- Inference rules denote combinators:

\[
(c \times d)(v, w) = (c(v), d(w))
\]

- Coinduction principle denotes recursion:

\[
\text{fix} f. c = \text{recursively defined function } f = c.
\]

\[
(\text{fix} f. c)(v) = c[\text{fix} f. c/f](v)
\]

- Idea: Problem with \text{fix} f. f is that it does not terminate.
Semantic Side Condition

- **Totality:**
  \( c \) is total from \( E \) to \( F \) iff \( \forall v \in \mathcal{T}[E] . c(v) = v' \) for some \( v' \in \mathcal{T}[F] \).

- **Hereditary Totality:**
  If \( f_1 : E_1 \leq F_1, \ldots, f_n : E_n \leq F_n \vdash c : E \leq F \)
  we say that \( c \) is hereditarily total iff
  \( c[c_1/f_1, \ldots, c_n/f_n] \) is total from \( E \) to \( F \)
  whenever \( f_i \) is total from \( E_i \) to \( F_i \) for \( i = 1 \ldots n \).

---

1 Proved by Eijiro Sumii, Yasuhiko Minamide, Naoki Kobayashi, Atsushi Igarashi and Fritz Henglein
Semantic Side Condition

- **Totality:**
  
  \[ c \text{ is total from } E \text{ to } F \iff \forall v \in \mathcal{T}[E].c(v) = v' \text{ for some } v' \in \mathcal{T}[F]. \]

- **Hereditary Totality:**
  
  If \( f_1 : E_1 \leq F_1, \ldots, f_n : E_n \leq F_n \vdash c : E \leq F \)
  
  we say that \( c \) is hereditarily total iff

  \[ c[c_1/f_1, \ldots, c_n/f_n] \text{ is total from } E \text{ to } F \]

  whenever \( f_i \) is total from \( E_i \) to \( F_i \) for \( i = 1 \ldots n \).

- **Theorem:** If a predicate \( P \) implies hereditary totality, then using \( P \) as a side condition for the coinduction rule results in a sound axiomatization.

- **Proposition:** Hereditary totality is undecidable

  \((2\text{-register machines as } c : 1^* \times 1^* \leq 1^* \times 1^*)\)

---

1 Proved by Eijiro Sumii, Yasuhiko Minamide, Naoki Kobayashi, Atsushi Igarashi and Fritz Henglein
Syntactic Side Condition \((S_2)\)

\[ \text{fix} f . c_1 ; (c_2 \times c_3) ; c_4 \]

- \(f\) may only occur under \(\times\) and if \(\text{proj}^{-1}\) does not occur "before"
- Remember: \(\text{proj}^{-1} : E \leq 1 \times E\).

**Lemma:** \(S_2\) implies hereditary totality

Proof: Induction on \(|v|_1\), where

\[
\begin{align*}
|()|_1 &= 1 \\
|\text{inl} \, v|_1 &= |v|_1 \\
|\text{inr} \, v|_1 &= |v|_1 \\
|\text{fold} \, v|_1 &= |v|_1 \\
|(v, w)|_1 &= |v|_1 + |w|_1
\end{align*}
\]
Syntactic Side Condition ($S_4$)

\[ \text{fix} f \ldots c_1 \times c_2 \ldots \]

- if $c_1 : E_1 \leq F_1$ and $o(E_1) = 0$
  then $f$ may occur in $c_2$.
- We allow folds: $\text{fix} f.\text{wrap}^{-1}; \text{id} + \text{id} \times f; c$, where $c$ is closed.

**Lemma:** $S_4$ implies hereditary totality

**Proof:** Induction on $|v|_0$ (string size), where

\[
\begin{align*}
|()|_0 &= 0 & |a|_0 &= 1 \\
|\text{inl} \ v|_0 &= |v|_0 & |\text{inr} \ v|_0 &= |v|_0 \\
|\text{fold} \ v|_0 &= |v|_0 & |(v, w)|_0 &= |v|_0 + |w|_0
\end{align*}
\]
Completeness

Completeness is proved by encoding other complete axiomatizations Salomaa($|v|_0$), Kozen($|v|_1$) and Grabmayer($|v|_0$)

**Theorem:** Let $P$ be hereditary totality, $S_2$ or $S_4$ then $|E \leq F|$ implies $\vdash_P E \leq F$
Completeness

Completeness is proved by encoding other complete axiomatizations Salomaa($|\nu|_0$), Kozen($|\nu|_1$) and Grabmayer($|\nu|_0$)

**Theorem:** Let $P$ be hereditary totality, $S_2$ or $S_4$ then $\models E \leq F$ implies $\vdash_P E \leq F$

**Theorem:** Let $P$ be either hereditary totality or $S_2$ then $\models \forall X_1, \ldots X_n. E \leq F$ iff $\vdash_P E \leq F$ (Parametrically Complete)
Bit Coded Strings

- If the regular expression is known, parts of a value can be inferred.
- Most Regular Expressions only allows one value-constructor:
  - 0 allows none.
  - 1 only allows ()
  - a only allows a
  - $E_1 \times E_2$ only allows $(v_1, v_2)$
  - $E^*$ only allows fold
- Only $E_1 + E_2$ allows two value-constructors $\text{inl } v$ and $\text{inr } v$.
- Translates $\text{inl } v$ to 0, $\text{inr } v$ to 1 and $(v, w)$ to $\text{code}(v)\text{code}(w)$. 
Example Bit Codings

- Bit codings for the string "abcbcba"

<table>
<thead>
<tr>
<th>Regular expression</th>
<th>Representation</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latin1</td>
<td>abcbcba00000000</td>
<td>64</td>
</tr>
<tr>
<td>Σ*</td>
<td>1a1b1c1b1c1b1a0</td>
<td>64</td>
</tr>
<tr>
<td>((a + b) + (c + d))*</td>
<td>1001011101011101011000</td>
<td>22</td>
</tr>
<tr>
<td>((a + b) + c)*</td>
<td>1001011110111101111110111000</td>
<td>20</td>
</tr>
<tr>
<td>a × (b + c)* × a</td>
<td>10111011100</td>
<td>11</td>
</tr>
<tr>
<td>a × b × c × b × c × b × a</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

- Optimization by generalising *.
- Coercions can work directly on the bit codings.
Summary

- Correspondence between parse trees and regular expression as type
- Containment proof by functional programming
- Axiomatization using general coinduction rule
- Observes key role of coercion totality
- Formulates sound and complete syntactic side conditions
- Encoding of existing axiomatizations, giving them a computational interpretation
- Bit coded strings
Future Work

- Disambiguation strategies
- Parsing using coercions $c : \Sigma^* \leq E + \Sigma^*$. (Left-disambiguation)
- Coercion synthesis
- Catamorphic post processing
- Application as type system

\[
f :: (a+b)^* \rightarrow \text{int}
f(v) = \text{case } v \text{ of}
\]
\[
\begin{align*}
    b^* & \Rightarrow 0 \\
    | b^*a(a+b)^* \text{ as } (_,(_,v2)) & \Rightarrow 1+f(v2)
\end{align*}
\]
Future Work

- Disambiguation strategies
- Parsing using coercions $c : \Sigma^* \leq E + \Sigma^*$. (Left-disambiguation)
- Coercion synthesis
- Catamorphic post processing
- Application as type system
  
  $\vdash c : (a + b)^* \leq b^* + b^*a(a + b)^*$

\[
f :: (a+b)^* \rightarrow \text{int}
f(v) = \text{case } v \text{ of}
\]
\[
\begin{align*}
b^* & \Rightarrow 0 \\
\mid b^*a(a+b)^* \text{ as } (_,(_,v2)) & \Rightarrow 1+f(v2)
\end{align*}
\]
Future Work

- Disambiguation strategies
- Parsing using coercions $c : \Sigma^* \leq E + \Sigma^*$. (Left-disambiguation)
- Coercion synthesis
- Catamorphic post processing
- Application as type system

$$\vdash c : (a + b)^* \leq b^* + b^* a(a + b)^*$$

\[
f : : (a+b)^* \to \text{int} \\
f(v) = \text{case } v \text{ of} \\
\quad b^* \Rightarrow 0 \\
\quad b^* a(a+b)^* \text{ as } (\_,(\_,v2)) \Rightarrow 1+f(v2)
\]

\[
f : : (a+b)^* \to \text{int} \\
f(v) = \text{case } c(s) \text{ of} \\
\quad \text{inl } v \Rightarrow 0 \\
\quad \text{inr } (\_,(\_,v2)) \Rightarrow 1+f(v2)
\]
Future Work

- Disambiguation strategies
- Parsing using coercions $c : \Sigma^* \leq E + \Sigma^*$. (Left-disambiguation)
- Coercion synthesis
- Catamorphic post processing
- Application as type system

$$\vdash c : (a + b)^* \leq b^* + b^*a(a + b)^*$$

```
f :: (a+b)^* -> int
f(v) = case v of
  b* => 0
| b*a(a+b)* as (_,(_,v2)) => 1+f(v2)
```

Questions?
Appendix: Syntactic Side Conditions

- $S_1(\Gamma \vdash \text{fix} f . c : E \leq F)$
  if and only if each occurrence of $f$ in $c$ is left-guarded by a $d$ where $\Gamma, \ldots \vdash d : E' \leq F'$ is the coercion judgement for $d$ occurring in the derivation of $\Gamma \vdash \text{fix} f . c : E \leq F$ and $o(E') = 0$.

- $S_3(\Gamma \vdash \text{fix} f . c : E \leq F)$
  if $c$ is of the form $\text{wrap}^{-1} ; (\text{id} + \text{id} \times f); d$ where $d$ is closed.

- $S_4 = S_1 \lor S_3$. 
Appendix: Syntactic Side Conditions

- \( S_1(\Gamma \vdash \text{fix} f . c : E \leq F) \) if and only if each occurrence of \( f \) in \( c \) is left-guarded by a \( d \) where \( \Gamma, \ldots \vdash d : E' \leq F' \) is the coercion judgement for \( d \) occurring in the derivation of \( \Gamma \vdash \text{fix} f . c : E \leq F \) and \( o(E') = 0 \).

- \( S_2(\Gamma \vdash \text{fix} f . c : E \leq F) \) if and only if each occurrence of \( f \) in \( c \) is left-guarded and for each subterm of the form \( c_1; c_2 \) in \( c \) at least one of the following conditions is satisfied:
  - \( c_1 \) is closed and \( \text{proj}^{-1} \)-free; \( \text{(proj}^{-1} \text{ says } E \leq 1 \times E) \)
  - \( c_2 \) is closed.

- \( S_3(\Gamma \vdash \text{fix} f . c : E \leq F) \) if \( c \) is of the form \( \text{wrap}^{-1}; (\text{id} + \text{id} \times f); d \) where \( d \) is closed.

- \( S_4 = S_1 \lor S_3. \)
Appendix: code

code(() : 1) = \epsilon

code(a : a) = \epsilon

code(inl v : E + F) = 0 \cdot \text{code}(v : E)

code(inr w : E + F) = 1 \cdot \text{code}(w : F)

code((v, w) : E \times F) = \text{code}(v : E) \cdot \text{code}(w : F)

code(fold v : E^*) = \text{code}(v : 1 + E \times E^*)
Appendix: decode

\[
\begin{align*}
\text{decode}'(d : 1) & = (((), d) \\
\text{decode}'(d : a) & = (a, d) \\
\text{decode}'(0d : E + E') & = \text{let } (v, d') = \text{decode}'(d : E) \\
& \quad \text{in } (\text{inl } v, d') \\
\text{decode}'(1d : E + E') & = \text{let } (w, d') = \text{decode}'(d : E) \\
& \quad \text{in } (\text{inr } w, d') \\
\text{decode}'(d : E \times E') & = \text{let } (v, d') = \text{decode}'(d : E) \\
& \quad \quad \text{let } (w, d''') = \text{decode}'(d' : E') \\
& \quad \quad \text{in } ((v, w), d''') \\
\text{decode}'(d : E^*) & = \text{let } (v, d') = \text{decode}'(d : 1 + E \times E^*) \\
& \quad \text{in } (\text{fold } v, d') \\
\text{decode}(d : E) & = \text{let } (w, d') = \text{decode}'(d : E) \\
& \quad \text{in if } d' = \epsilon \text{ then } w \text{ else error}
\end{align*}
\]
Appendix: Encoding Salomaa

\[
E = F \\
E^* = F^*
\]

By induction hypothesis there exist \( \vdash_{S_4} c : E \leq F \) and \( \vdash_{S_4} d : F \leq E \).

Assume \( f : E^* \leq F^* \).

\[
\begin{align*}
E^* &\leq (1 + E \times E^*) & \text{by wrap}^{-1} \\
&\leq (1 + F \times F^*) & \text{by id + c \times f} \\
&\leq F^* & \text{by wrap}
\end{align*}
\]

Assume \( g : F^* \leq E^* \).

\[
\begin{align*}
F^* &\leq (1 + F \times F^*) & \text{by wrap}^{-1} \\
&\leq (1 + E \times E^*) & \text{by id + d \times g} \\
&\leq E^* & \text{by wrap}
\end{align*}
\]
Appendix: Encoding Salomaa

\[ E^* = (1 + E)^* \]

\[ \vdash S_4 \text{tagL}_1; \text{retag} : E \leq 1 + E. \]

Apply rule: \[ \frac{E = F}{E^* = F^*} \]

Assume \( f : (1 + E)^* \leq E^*. \)

\[
\begin{align*}
(1 + E)^* & \leq 1 + (1 + E) \times (1 + E)^* \quad \text{by wrap}^{-1} \\
& \leq 1 + (1 + E) \times E^* \quad \text{by } f \\
& \leq 1 + 1 \times E^* + E \times E^* \quad \text{by distR} \\
& \leq 1 + E^* + E \times E^* \quad \text{by proj} \\
& \leq 1 + E \times E^* + E^* \quad \text{by retag} \\
& \leq E^* + E^* \quad \text{by wrap} \\
& \leq E^* \quad \text{by untag}
\end{align*}
\]
Appendix: Encoding Salomaa

\[
E = F \times E + G \\
E = F^* \times G
\]

(if \( o(F) = 0 \)).

IH: \( \vdash_{S_4} c_1 : E \leq F \times E + G \) and \( \vdash_{S_4} d_1 : F \times E + G \leq E \).

Assume \( f : F^* \times G \leq E \)

- \[ F^* \times G \leq (1 + F \times F^*) \times G \] by \( \text{wrap}^{-1} \)
- \[ \leq 1 \times G + F \times F^* \times G \] by \( \text{distR} \)
- \[ G + F \times F^* \times G \] by \( \text{proj} \)
- \[ \leq G + F \times E \] by \( f \)
- \[ \leq F \times E + G \] by \( \text{retag} \)
- \[ \leq E \] by \( d_1 \)

Assume \( \vdash_{S_4} g : E \leq F^* \times G \)

- \[ E \leq F \times E + G \] by \( c_1 \)
- \[ \leq G + F \times E \] by \( \text{retag} \)
- \[ \leq G + F \times F^* \times G \] by \( g \)
- \[ \leq 1 \times G + F \times F^* \times G \] by \( \text{proj}^{-1} \)
- \[ \leq (1 + F \times F^*) \times G \] by \( \text{distR}^{-1} \)
- \[ \leq F^* \times G \] by \( \text{wrap} \)
Appendix: Semantics as Regular Sets

\[ \mathcal{L}[0] = \emptyset \]
\[ \mathcal{L}[1] = \{\epsilon\} \]
\[ \mathcal{L}[a] = \{a\} \]
\[ \mathcal{L}[E + F] = \mathcal{L}[E] \cup \mathcal{L}[F] \]
\[ \mathcal{L}[E \times F] = \mathcal{L}[E] \cdot \mathcal{L}[F] \]
\[ \mathcal{L}[E^*] = \bigcup_{i \geq 0} (\mathcal{L}[E])^i \]

where \( S \cdot T = \{s t \mid s \in S \land t \in T\}, \ S^0 = \{\epsilon\} \) and \( S^{i+1} = S \cdot S^i \).