Regular Combinators for String Transformations

Rajeev Alur   Adam Freilich   Mukund Raghothaman

CSL-LICS, 2014

ExCAPE
Expeditions in Computer Augmented Program Engineering

Penn Engineering

Our Goal

$$
\begin{aligned}
\text{Languages, } \Sigma^* \to \texttt{bool} \quad &\equiv \quad \text{Regular expressions} \\
\text{Tranformations, } \Sigma^* \to \Gamma^* \quad &\equiv \quad ?
\end{aligned}
$$

String Transformations

… are all over the place

- Find and replace
  Rename variable `foo` to `bar`
- Spreadsheet macros
  Convert phone numbers like "(123) 456-7890" to "123-456-7890"
- String sanitization
- …

String Transformations

Tool and theory support

- Good tool support: sed, AWK, Perl, domain-specific tools, ...
- Renewed interest: Recent transducer-based tools such as Bek, Flash-Fill, ...
- But unsatisfactory theory ...
- Expressibility: Can I express ⟨*favorite transformation*⟩ using ⟨*favorite tool*⟩?
- Analysis questions:
  - Is the transformation well-defined for all inputs?
  - Does the output always have some "nice" property? $\forall \sigma$, is it the case that $f(\sigma) \in L$?
  - Are two transformations equivalent?

Historical Context

Regular languages

Beautiful theory

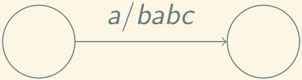
$$\text{Regular expressions} \quad \equiv \quad \text{DFA}$$

Analysis questions (mostly) efficiently decidable

Lots of practical implementations

String Transducers

One-way transducers: Mealy machines



Folk knowledge [Aho et al 1969]

Two-way transducers strictly more powerful than one-way transducers

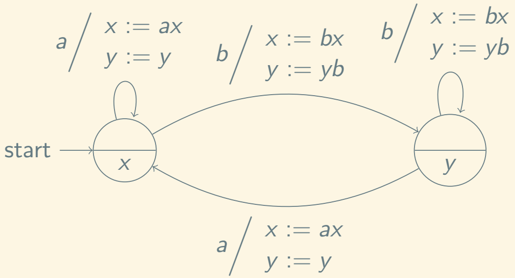
Gap includes many transformations of interest

Examples: string reversal, copy, substring swap, etc.

Regular String Transformations

- Two-way finite state transducers are our notion of regularity
- Known results
  - Closed under composition [Chytil, Jákl 1977]
  - Decidable equivalence checking [Gurari 1980]
  - Equivalent to MSO-definable string transformations [Engelfriet, Hoogeboom 2001]
- Recent result: Equivalent one-way deterministic model with applications to the analysis of list-processing programs [Alur, Černý 2011]

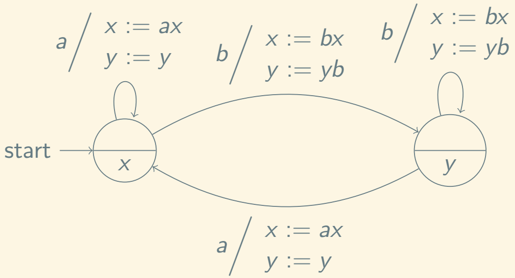
Streaming String Transducers (SST)



If input ends with a *b*, then delete all *a*-s, else reverse

- *x* contains the reverse of the input string seen so far
- *y* contains the list of *b*-s read so far

Streaming String Transducers (SST)



- Finitely many locations
- Finite set of registers
- Transitions test-free
- Registers concatenated (copyless updates only)
- Final states associated with registers (output functions)

Regular String Transformations

Rephrasing our goal

$$\begin{aligned} \text{Languages, DFA} &\equiv \text{Regular expressions} \\ \text{Tranformations, SST} &\equiv \text{?} \end{aligned}$$

Can we Find an Equivalent Regex-like Characterization?

Motivation

- Theoretical: To understand regular functions
- Practical: As the basis for a domain-specific language for string transformations

Base functions: $R \mapsto \gamma$

If $\sigma \in L(R)$, then γ, and otherwise undefined

$$(\{\text{``.c''}\} \cup \{\text{``.cpp''}\}) \mapsto \text{``.cpp''}$$

Analogue of basic regular expressions: $\{a\}$, for $a \in \Sigma$
R is a regular expression and γ is a constant

If-then-else: ite $R\ f\ g$

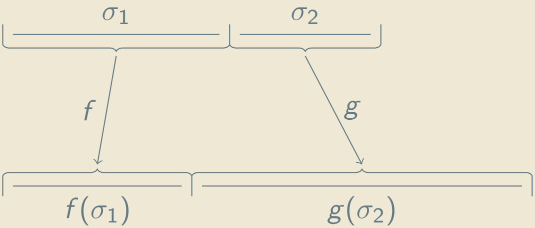
If $\sigma \in L(R)$, then $f(\sigma)$, and otherwise $g(\sigma)$

$$\text{ite}\ [0-9]^*\ (\Sigma^* \mapsto \text{“}\texttt{Number}\text{”})\ (\Sigma^* \mapsto \text{“}\texttt{Non-number}\text{”})$$

Analogue of unambiguous regex union

Split sum: split(f, g)

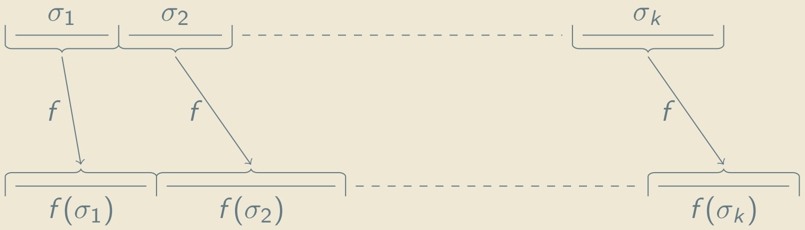
Split σ into $\sigma = \sigma_1\sigma_2$ with both $f(\sigma_1)$ and $g(\sigma_2)$ defined. If the split is unambiguous then split$(f, g)(\sigma) = f(\sigma_1)g(\sigma_2)$



Analogue of regex concatenation

Iterated sum: iterate(f)

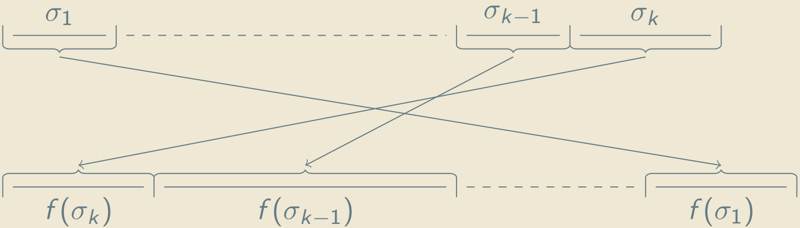
Split $\sigma = \sigma_1\sigma_2 \ldots \sigma_k$, with all $f(\sigma_i)$ defined. If the split is unambiguous, then output $f(\sigma_1)f(\sigma_2)\ldots f(\sigma_k)$



- Kleene-*
- If *echo* echoes a single character, then iterate(*echo*) is the identity function

Left-iterated sum: left-iterate(f)

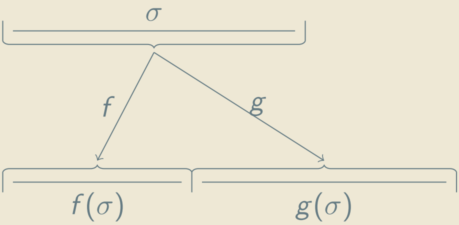
Split $\sigma = \sigma_1 \sigma_2 \ldots \sigma_k$, with all $f(\sigma_i)$ defined. If the split is unambiguous, then output $f(\sigma_k) f(\sigma_{k-1}) \ldots f(\sigma_1)$



Think of $\sigma \mapsto \sigma^{rev}$: left-iterate(*echo*)

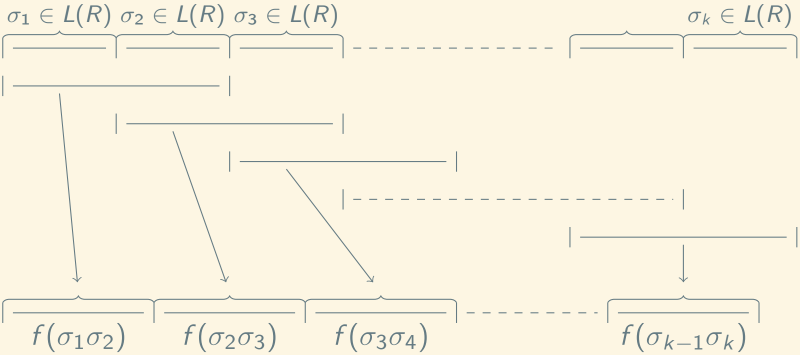
“Repeated” sum: combine(f, g)

combine$(f, g)(\sigma) = f(\sigma)g(\sigma)$



- No regex equivalent
- $\sigma \mapsto \sigma\sigma$: combine$(id, id)$

Chained sum: chain(f, R)



And similarly for left-chain(f, R)

Function composition: $f \circ g$

$f \circ g(\sigma) = f(g(\sigma))$

$$\sigma \longrightarrow \boxed{g} \longrightarrow \boxed{f} \longrightarrow f(g(\sigma))$$

Regular string transformations are closed under composition

Function Combinators are Expressively Complete

Theorem (Completeness)

All regular string transformations can be expressed using the following combinators:

- *Basic functions:* $a \mapsto \gamma$, $\epsilon \mapsto \gamma$, $\bot$,
- $ite\ R\ f\ g$, $split(f, g)$, $combine(f, g)$, *and*
- *chained sums:* $chain(f, R)$, *and* $left\text{-}chain(f, R)$.

Function Combinators are Expressively Complete

Arbitrary monoids $(\mathbb{D}, \otimes, 0)$

- Functions $\Sigma^* \to \mathbb{D}$ for an arbitrary monoid $(\mathbb{D}, \otimes, 0)$
- All machinery still works: Function combinators remain expressively complete
  Base functions: $a \mapsto \gamma$, $\epsilon \mapsto \gamma$, for $\gamma \in \mathbb{D}$
- Strings $(\Gamma^*, \cdot, \epsilon)$ just a special case
- Monoid of discounted costs $(\mathit{cost}, \mathit{discount}) \in \mathbb{R} \times [0, 1]$
  $(c, d) \otimes (c', d') = (c + dc', dd')$
  Identity element: $(0, 1)$
  Potentially useful for quantitative analysis

The Special Case of Commutative Monoids

Expressive completeness of function combinators

- Integers under addition $(\mathbb{Z}, +, 0)$, and integer-valued cost functions $\Sigma^* \to \mathbb{Z}$
- Example: Count number of a-s followed by b

$$\text{split}(b^* \mapsto 0, \text{iterate}(a^+ \cdot b^+ \mapsto 1), a^* \mapsto 0)$$

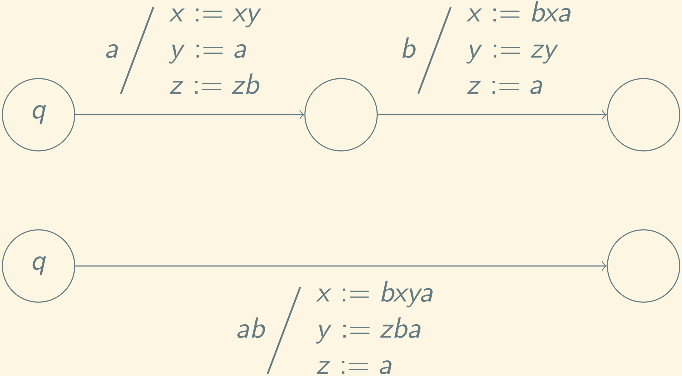
- Smaller set of combinators needed for expressive completeness
  - Basic functions: $a \mapsto \gamma$, $\epsilon \mapsto \gamma$, $\perp$
  - ite $R\ f\ g$, split(f, g), and
  - iterate(f)
- Unnecessary combinators: combine(f, g), chain(f, R), left-chain(f, R)

A Taste of the Proof

Broadly similar to DFA-to-Regex translation

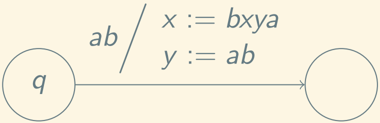
A Taste of the Proof

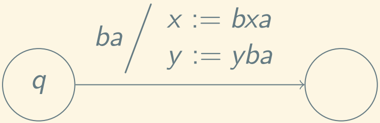
Summmarize effect of (individual) strings



A Taste of the Proof

Shapes





$$x := \overset{\gamma_{x1}}{\longleftrightarrow} x \overset{\gamma_{x2}}{\longleftrightarrow} y \overset{\gamma_{x3}}{\longleftrightarrow}$$

$$y := \overset{\gamma_{y1}}{\longleftrightarrow}$$

$$x := \overset{\gamma_{x1}}{\longleftrightarrow} x \overset{\gamma_{x2}}{\longleftrightarrow}$$

$$y := \overset{\gamma_{y1}}{\longleftrightarrow} y \overset{\gamma_{y2}}{\longleftrightarrow}$$

A Taste of the Proof

Summarizing effect of (a set of) strings

"Summarize" = "Give expression for each patch"

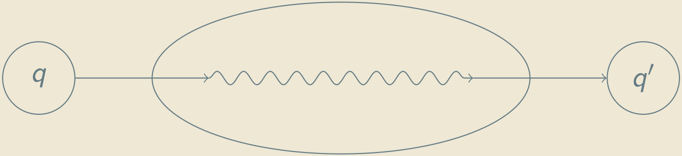
$$x := \xleftrightarrow{\gamma_{x1}} x \xleftrightarrow{\gamma_{x2}} y \xleftrightarrow{\gamma_{x3}}$$

$$y := \xleftrightarrow{\gamma_{y1}}$$

A Taste of the Proof

Piggyback on the Regex-to-DFA Translation Algorithm

Summarize all paths $q \to q'$ with shape S

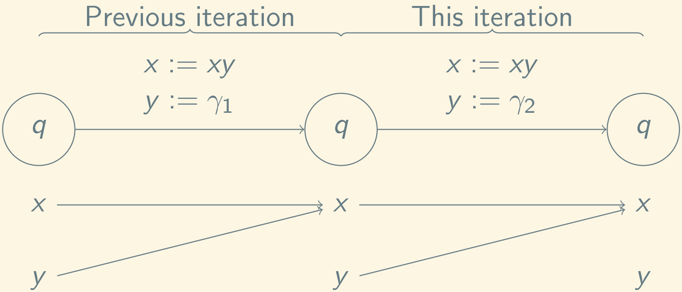


$Q_r \subseteq Q$

Start with $Q_r = \emptyset$ and iteratively add states until $Q_r = Q$

A Taste of the Proof

Summarizing loops: Or why the chained sum is needed

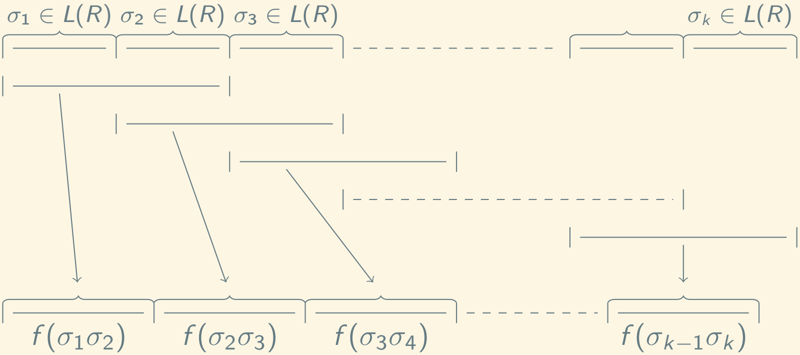


Value appended to x at the end of *this* loop iteration (γ_1) depends on value computed in y during the *previous* iteration

Chained sum

A Taste of the Proof

Recall the chained sum: chain(f, R)



Conclusion

Introduced a declarative notation for regular string transformations

Conclusion

Summary of operators

Purpose	Regular Transformations	Regular Expressions
Base	$R \mapsto \gamma$	$\{a\}$, for $a \in \Sigma$
Union	ite R f g	$R_1 \cup R_2$
Concatenation	split(f, g)	$R_1 \cdot R_2$
Kleene-*	iterate(f) (also left-iterate(f))	R^*
Repetition	combine(f, g)	New!
Chained sum	chain(f, R) (and left-chain(f, R))	
Composition	$f \circ g$	

Future Work

- Design and implement a DSL for string transformations based on these foundations
- Lower bounds on expressibility of certain functions
- Theory of regular functions
  - Strings to numerical domains
  - Strings to semirings
  - Trees to trees / strings (Processing hierarchical data, XML documents, etc.)
  - ω-strings to strings
- Automatically learn transformations
  - from input/output examples
  - from teachers (L*)

Thank you! Questions?
Suggestions? Brickbats?