DReX: A Declarative Language for Efficiently Evaluating Regular String Transformations

Rajeev Alur Loris D'Antoni Mukund Raghothaman

POPL 2015

ExCAPE
Expeditions in Computer Augmented Program Engineering

Penn Engineering

DReX is a DSL for String Transformations

align-bibtex

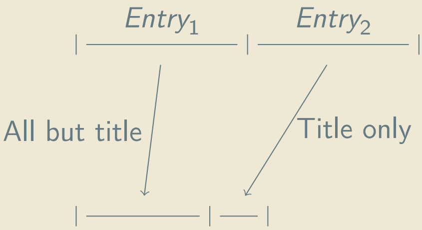
```
...

@book{Book1,
  title = {Title0},
  author = {Author1},
  year = {Year1},
}

@book{Book2,
  title = {Title1},
  author = {Author2},
  year = {Year2},
}

...
```

```
...

@book{Book1,
  title = {Title1},
  author = {Author1},
  year = {Year1},
}

...
```

Describing *align-bibtex* Using DReX

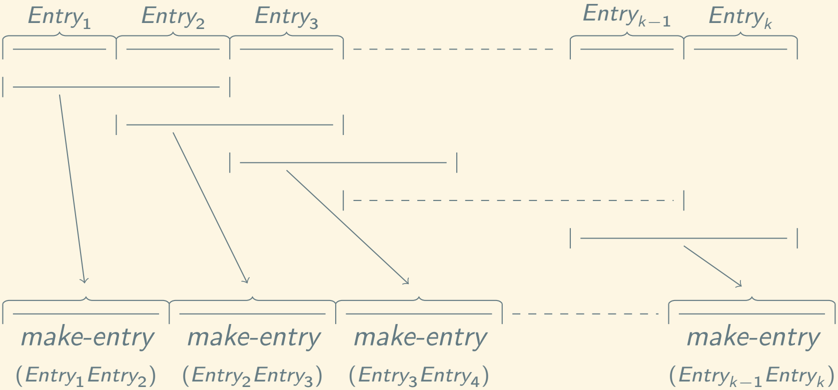
The simpler issue of *make-entry*

Given two entries, $Entry_1$ and $Entry_2$, *make-entry* outputs the title of $Entry_2$ and the remaining body of $Entry_1$



Describing *align-bibtex* Using DReX

$\textit{align-bibtex} = \text{chain}(\textit{make-entry}, R_{\textit{Entry}})$



Function combinators — such as chain — combine smaller functions into bigger ones

Why DReX?

- DReX is declarative

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

- DReX is fast: Streaming evaluation algorithm for well-typed expressions
- Based on robust theoretical foundations
  - Expressively equivalent to regular string transformations
  - Multiple characterizations: two-way finite state transducers, MSO-definable graph transformations, streaming string transducers
  - Closed under various operations: function composition, regular look-ahead etc.
- DReX supports algorithmic analysis
  - 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?

DReX is publicly available! Go to drexonline.com

Function Combinators

Base functions: $\sigma \mapsto \gamma$

Map input string σ to γ, and undefined everywhere else

$$\text{``}.\mathrm{c}\text{''} \mapsto \text{``}.\mathrm{cpp}\text{''}$$

$\sigma \in \Sigma^*$ and $\gamma \in \Gamma^*$ are constant strings
Analogue of basic regular expressions: $\{\sigma\}$, for $\sigma \in \Sigma^*$

Conditionals: try f else g

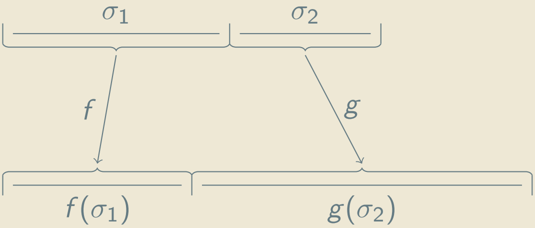
If $f(\sigma)$ is defined, then output $f(\sigma)$, and otherwise output $g(\sigma)$

$$\text{try } [0\text{-}9]^* \mapsto \text{“}\texttt{Number}\text{”}$$
$$\text{else } [a\text{-}z]^* \mapsto \text{“}\texttt{Name}\text{”}$$

Analogue of unambiguous regex union

Split sum: split(f, g)

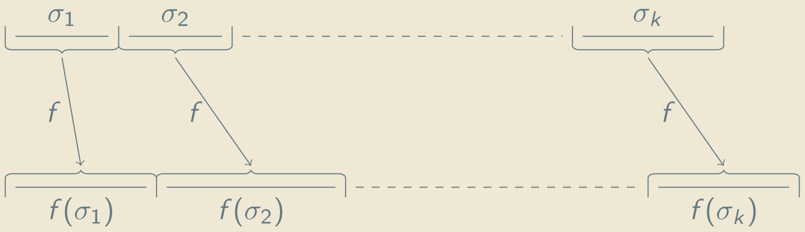
Split σ into $\sigma = \sigma_1\sigma_2$ with both $f(\sigma_1)$ and $g(\sigma_2)$ defined. If the split is unambiguous then $\text{split}(f,g)(\sigma) = f(\sigma_1)g(\sigma_2)$



- Analogue of regex concatenation
- If *title* maps a BibTeX entry to its title, and *body* maps a BibTeX entry to the rest of its body, then *make-entry* = split(*body*, *title*)

Iterated sum: iterate(f)

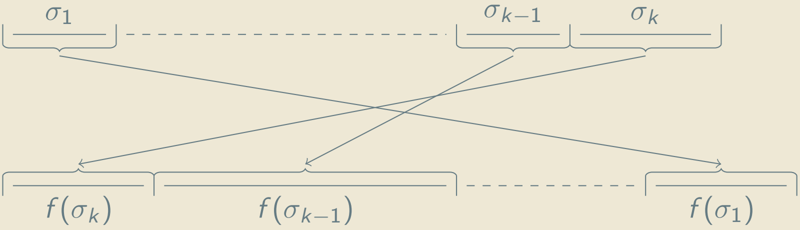
Split $\sigma = \sigma_1\sigma_2\ldots\sigma_k$, with all $f(\sigma_i)$ defined. If the split is unambiguous, then output $f(\sigma_1)f(\sigma_2)\ldots f(\sigma_k)$



- Kleene-*
- If *echo* echoes a single character, then $id = \text{iterate}(echo)$ is the identity function

Left-iterated sum: left-iterate(f)

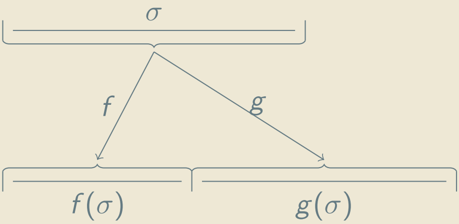
Split $\sigma = \sigma_1\sigma_2\ldots\sigma_k$, with all $f(\sigma_i)$ defined. If the split is unambiguous, then output $f(\sigma_k)f(\sigma_{k-1})\ldots f(\sigma_1)$



Think of string reversal: left-iterate(*echo*)

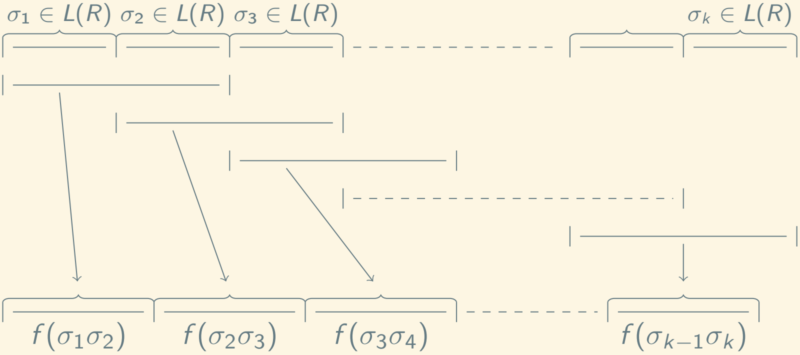
"Repeated" sum: combine(f, g)

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)

Summary of Function Combinators

Purpose	Regular Transformations	Regular Expressions
Base	$\bot$, $\sigma \mapsto \gamma$	$\emptyset$, $\{\sigma\}$
Concatenation	split(f, g), left-split(f, g)	$R_1 \cdot R_2$
Union	try f else g	$R_1 \cup R_2$
Kleene-*	iterate(f), left-iterate(f)	R^*
Repetition	combine(f, g)	
Chained sum	chain(f, R), left-chain(f, R)	New!

Regular String Transformations

Or, why our choice of combinators was not arbitrary

$$\begin{array}{rcl} \text{Languages, } \Sigma^* \to \texttt{bool} & \equiv & \text{DFA} \\ \text{Tranformations, } \Sigma^* \to \Gamma^* & \equiv & ? \end{array}$$

Historical Context

Regular languages

Beautiful theory

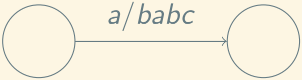
$$\text{Regular expressions} \quad \equiv \quad \text{DFA}$$

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 interesting transformations

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

String Transducers

Two-way finite state transducers

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

Function Combinators are Expressively Complete

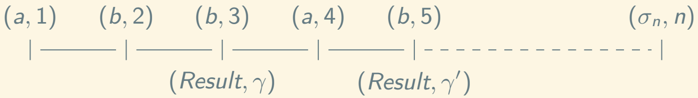
Theorem (Completeness, Alur et al 2014)

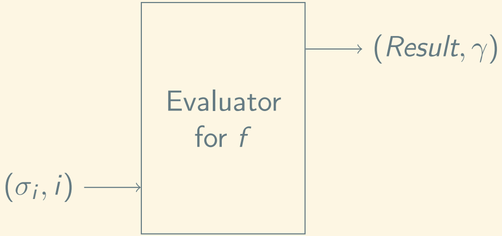
All regular string transformations can be expressed using the following combinators:

- *Basic functions:* $\perp$, $\sigma \mapsto \gamma$,
- *split*(f, g), *left-split*(f, g),
- *try* f *else* g,
- *iterate*(f), *left-iterate*(f),
- *combine*(f, g),
- *chained sums: chain*(f, R), *and left-chain*(f, R).

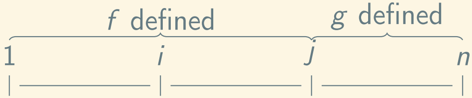
Evaluating DReX Expressions

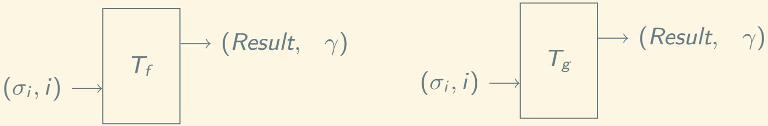
The Anatomy of a Streaming Evaluator



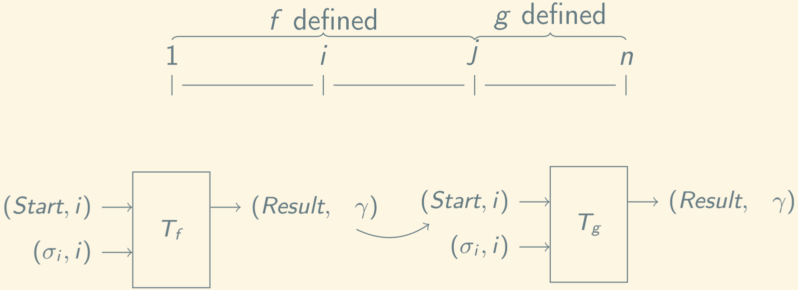


The Case of split(f, g)

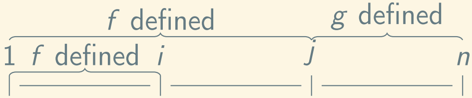


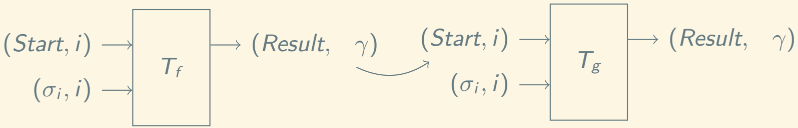


The Case of split(f, g)

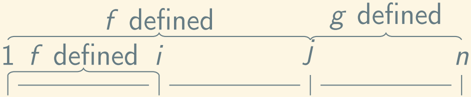


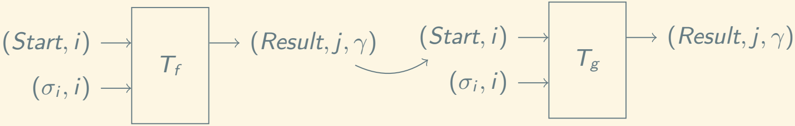
The Case of split(f, g)



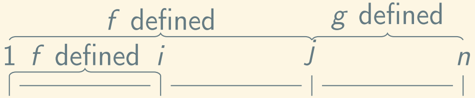


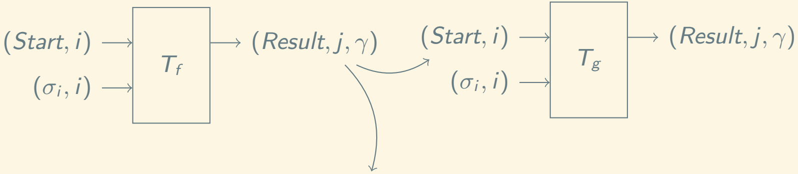
The Case of split(f, g)





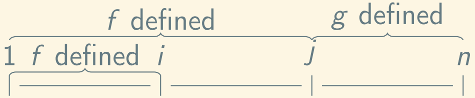
The Case of split(f, g)

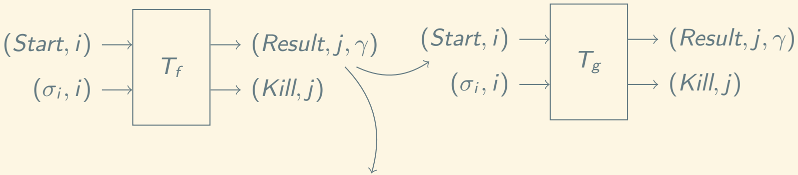




Thread starting at index	Index at which T_f responded	Result reported by T_f
2	9	*aaab*
3	7	*abbab*
...	...	...

The Case of split(f, g)

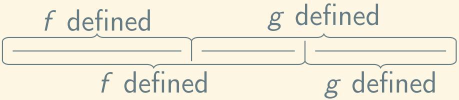




Thread starting at index	Index at which T_f responded	Result reported by T_f
2	9	*aaab*
3	7	*abbab*
...	...	...

The Case of split(f, g)

- What if two threads of T_g report results simultaneously?



- Statically disallow!
- split(f, g) is well-typed iff
  - both f and g are well-typed, and
  - their domains are unambiguously concatenable

Main Result

Theorem

1. *All regular string transformations can be expressed as well-typed DReX expressions.*
2. *DReX expressions can be type-checked in* $O(\mathit{poly}(|f|, |\Sigma|))$.
3. *Given a well-typed DReX expression* f, *and an input string* σ, $f(\sigma)$ *can be computed in time* $O(|\sigma|, \mathit{poly}(|f|))$.

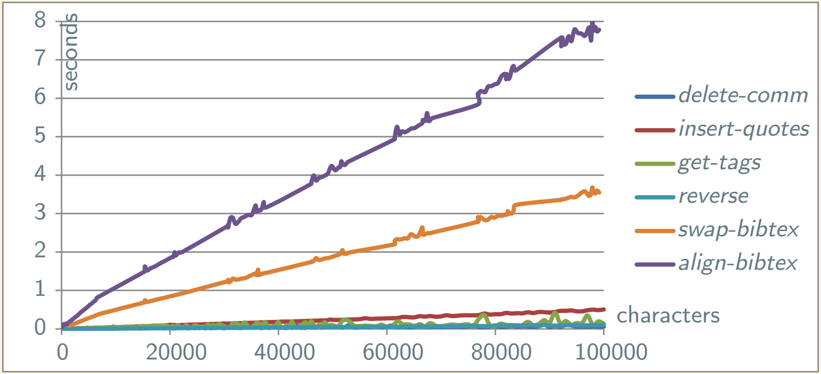
Summary of Typing Rules

- $\bot$, $\sigma \mapsto \gamma$ are always well-typed
- split(f, g) and left-split(f, g) are well-typed iff
  - f and g are well-typed, and
  - Dom(f) and Dom(g) are unambiguously concatenable
- try f else g is well-typed iff
  - f and g are well-typed, and
  - Dom(f) and Dom(g) are disjoint
- iterate(f) and left-iterate(f) are well-typed iff
  - f is well-typed, and
  - Dom(f) is unambiguously iterable
- chain(f, R) and left-chain(f, R) are well-typed iff
  - f is well-typed, R is an unambiguous regular expression,
  - Dom(f) is unambiguously iterable, and
  - Dom(f) $= [\![R \cdot R]\!]$

Experimental Results

Experimental Results

Streaming evaluation algorithm for well-typed expressions



- *align-bibtex* has 3500 nodes in syntax tree, typechecks in ≈half a second
- Type system did not get in the way

Conclusion

- Introduced a DSL for regular string transformations
- Described a fast streaming algorithm to evaluate well-typed expressions

Conclusion

Summary of operators

Purpose	Regular Transformations	Regular Expressions
Base	$\bot$, $\sigma \mapsto \gamma$	$\emptyset$, $\{\sigma\}$
Concatenation	split(f, g), left-split(f, g)	$R_1 \cdot R_2$
Union	try f else g	$R_1 \cup R_2$
Kleene-*	iterate(f), left-iterate(f)	R^*
Repetition	combine(f, g)	
Chained sum	chain(f, R), left-chain(f, R)	New!

Future Work

- Implement practical programmer assistance tools
  - Static: Precondition computatation, equivalence checking
  - Runtime: Debugging aids
- Theory of regular functions
  - Automatically learn transformations from teachers (L*), from input / output examples, etc.
  - Trees to trees / strings (Processing hierarchical data, XML documents, etc.)
  - ω-strings to strings
- Non-regular extensions
  - "Count number of a-s in a string"

Thank you! Questions?

drexonline.com

What About Unrestricted DReX Expressions?

Evaluating Unrestricted DReX Expressions is Hard

Or, why the typing rules are essential

- With function composition, it is PSPACE-complete
- combine(f, g) is defined iff both f and g are defined
  Flavour of regular expression intersection
  The best algorithms for this are either
  - Non-elementary in regex size, or
  - Cubic in length of input string