DReX: A Declarative Language for Efficiently Evaluating Regular String Transformations

Rajeev Alur  Loris D’Antoni  Mukund Raghothaman

POPL 2015
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},
}

...
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 $Entry_1$.

<table>
<thead>
<tr>
<th>$Entry_1$</th>
<th>$Entry_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>All but title</td>
<td>Title only</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Describing \textit{align-bibtex} Using DReX

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

Function combinators — such as chain — combine smaller functions into bigger ones.
Why DReX?

- DReX is declarative
  
  Languages, $\Sigma^* \rightarrow \text{bool} \equiv$ Regular expressions
  
  Transformations, $\Sigma^* \rightarrow \Gamma^*$ $\equiv$ DReX

- 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 $\sigma$ to $\gamma$, and undefined everywhere else

".c" $\mapsto$ "cpp"

$\sigma \in \Sigma^*$ 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 $g(\sigma)$

```
try [0-9]* → "Number"
else [a-z]* → "Name"
```

Analogue of unambiguous regex union
Split sum: \( \text{split}(f, g) \)

Split \( \sigma \) 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 \( \text{title} \) maps a BibTeX entry to its title, and \( \text{body} \) maps a BibTeX entry to the rest of its body, then \( \text{make-entry} = \text{split}(\text{body}, \text{title}) \)
Iterated sum: \( \text{iterate}(f) \)

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 \( \text{echo} \) echoes a single character, then \( \text{id} = \text{iterate}(\text{echo}) \) 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: \( \text{combine}(f, g) \)

\[
\text{combine}(f, g)(\sigma) = f(\sigma)g(\sigma)
\]

- No regex equivalent
- \( \sigma \mapsto \sigma\sigma\) : combine \((id, id)\)
Chained sum: \( \text{chain}(f, R) \)

\[
\sigma_1 \in L(R) \quad \sigma_2 \in L(R) \quad \sigma_3 \in L(R) \quad \cdots \quad \sigma_k \in L(R)
\]

\[
f(\sigma_1 \sigma_2) \quad f(\sigma_2 \sigma_3) \quad f(\sigma_3 \sigma_4) \quad \cdots \quad f(\sigma_{k-1} \sigma_k)
\]

And similarly for left-chain\( f, R \)
### Summary of Function Combinators

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Regular Transformations</th>
<th>Regular Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>$\bot$, $\sigma \mapsto \gamma$</td>
<td>$\emptyset$, ${\sigma}$</td>
</tr>
<tr>
<td>Concatenation</td>
<td>$\text{split}(f, g)$, $\text{left-split}(f, g)$</td>
<td>$R_1 \cdot R_2$</td>
</tr>
<tr>
<td>Union</td>
<td>$\text{try } f \text{ else } g$</td>
<td>$R_1 \cup R_2$</td>
</tr>
<tr>
<td>Kleene-*</td>
<td>$\text{iterate}(f)$, $\text{left-iterate}(f)$</td>
<td>$R^*$</td>
</tr>
<tr>
<td>Repetition</td>
<td>$\text{combine}(f, g)$</td>
<td></td>
</tr>
<tr>
<td>Chained sum</td>
<td>$\text{chain}(f, R)$, $\text{left-chain}(f, R)$</td>
<td>New!</td>
</tr>
</tbody>
</table>
Regular String Transformations

Or, why our choice of combinators was not arbitrary

\[ \text{Languages, } \Sigma^* \rightarrow \text{bool} \equiv \text{DFA} \]
\[ \text{Transformations, } \Sigma^* \rightarrow \Gamma^* \equiv ? \]
Historical Context

Regular languages

Beautiful theory

Regular expressions $\equiv$ DFA

Analysis questions (mostly) efficiently decidable

Lots of practical implementations
String Transducers

One-way transducers: Mealy machines

\[ a/babc \]

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
Theorem (Completeness, Alur et al 2014)

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

- **Basic functions:** $\bot$, $\sigma \mapsto \gamma$,
- **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

(a, 1)     (b, 2)     (b, 3)     (a, 4)     (b, 5)     (σ_n, n)
|          |          |          |          |          |
(Result, γ)   (Result, γ')

Evaluator for f

(σ_i, i) → (Result, γ)
The Case of split($f, g$)

$$1 \overset{f \text{ defined}}{\to} i \overset{g \text{ defined}}{\to} j \overset{n}{\to}$$

$$(\sigma_i, i) \xrightarrow{T_f} (\text{Result}, \gamma)$$

$$(\sigma_i, i) \xrightarrow{T_g} (\text{Result}, \gamma)$$
The Case of $\text{split}(f, g)$

\[ f \text{ defined} \quad \frac{i}{1} \quad \frac{j}{n} \quad g \text{ defined} \]

\[(\text{Start, } i) \xrightarrow{T_f} (\text{Result, } \gamma) \quad (\text{Start, } i) \xrightarrow{T_g} (\text{Result, } \gamma) \]

\[(\sigma_i, i) \quad (\sigma_i, i) \]
The Case of $\text{split}(f, g)$

$f$ defined
\[
\begin{array}{c}
\text{1} \\
\text{f defined}
\end{array}
\quad
i
\quad
\begin{array}{c}
g \text{ defined}
\end{array}
\quad
\begin{array}{c}
j
\end{array}
\quad
n
\]

(\text{Start, } i) \rightarrow T_f \rightarrow (\text{Result, } \gamma) \rightarrow T_g \rightarrow (\text{Result, } \gamma)

(\sigma_i, i) \rightarrow T_f

\rightarrow (\text{Result, } \gamma)

(\sigma_i, i) \rightarrow T_g

\rightarrow (\text{Result, } \gamma)
The Case of split($f$, $g$)

\[
\begin{array}{c}
\text{f defined} \\
1 \quad f \text{ defined} \quad i \\
\hline
\end{array}
\begin{array}{c}
\text{g defined} \\
\hline
J \quad n
\end{array}
\]

\[(\text{Start, } i) \rightarrow (\text{Result, } j, \gamma) \quad (\text{Start, } i) \rightarrow (\text{Result, } j, \gamma)\]

\[(\text{Start, } i) \rightarrow (\text{Result, } j, \gamma) \quad (\text{Start, } i) \rightarrow (\text{Result, } j, \gamma)\]

\[(\sigma_i, i) \rightarrow T_f \quad (\sigma_i, i) \rightarrow T_g\]
The Case of split\((f, g)\)

\[
\begin{align*}
1 & \quad f \text{ defined} & \quad g \text{ defined} \\
\quad i & \quad j & \quad n
\end{align*}
\]

\[
\begin{align*}
(\text{Start, } i) & \quad \rightarrow \quad (\text{Result, } j, \gamma) \\
(\sigma_i, i) & \quad \rightarrow \quad T_f
\end{align*}
\]

\[
\begin{align*}
(\text{Start, } i) & \quad \rightarrow \quad (\text{Result, } j, \gamma) \\
(\sigma_i, i) & \quad \rightarrow \quad T_g
\end{align*}
\]

<table>
<thead>
<tr>
<th>Thread starting at index</th>
<th>Index at which (T_f) responded</th>
<th>Result reported by (T_f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>9</td>
<td>\textit{aaab}</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>\textit{abbab}</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
<td>\ldots</td>
</tr>
</tbody>
</table>
The Case of $\text{split}(f, g)$

$\begin{align*}
\text{Thread starting at index} & \quad \text{Index at which } T_f \text{ responded} & \quad \text{Result reported by } T_f \\
2 & \quad 9 & \quad \text{aaab} \\
3 & \quad 7 & \quad \text{abbab} \\
\cdots & \quad \cdots & \quad \cdots
\end{align*}$
The Case of $\text{split}(f, g)$

- What if two threads of $T_g$ report results simultaneously?

  - $f$ defined
  - $g$ defined

- Statically disallow!

- $\text{split}(f, g)$ is well-typed iff
  - both $f$ and $g$ are well-typed, and
  - their domains are unambiguously concatenable
Theorem

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

- $\bot, \sigma \mapsto \gamma$ are always well-typed
- $\text{split}(f, g)$ and $\text{left-split}(f, g)$ are well-typed iff
  - $f$ and $g$ are well-typed, and
  - $\text{Dom}(f)$ and $\text{Dom}(g)$ are unambiguously concatenable
- $\text{try } f \text{ else } g$ is well-typed iff
  - $f$ and $g$ are well-typed, and
  - $\text{Dom}(f)$ and $\text{Dom}(g)$ are disjoint
- $\text{iterate}(f)$ and $\text{left-iterate}(f)$ are well-typed iff
  - $f$ is well-typed, and
  - $\text{Dom}(f)$ is unambiguously iterable
- $\text{chain}(f, R)$ and $\text{left-chain}(f, R)$ are well-typed iff
  - $f$ is well-typed, $R$ is an unambiguous regular expression,
  - $\text{Dom}(f)$ is unambiguously iterable, and
  - $\text{Dom}(f) = [R \cdot R]$
Experimental Results
Experimental Results
Streaming evaluation algorithm for well-typed expressions

- `align-bibtex` has 3500 nodes in syntax tree, typechecks in \( \approx \) 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

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Regular Transformations</th>
<th>Regular Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>$\bot, \sigma \mapsto \gamma$</td>
<td>$\emptyset, {\sigma}$</td>
</tr>
<tr>
<td>Concatenation</td>
<td>$\text{split}(f, g), \text{left-split}(f, g)$</td>
<td>$R_1 \cdot R_2$</td>
</tr>
<tr>
<td>Union</td>
<td>$\text{try } f \text{ else } g$</td>
<td>$R_1 \cup R_2$</td>
</tr>
<tr>
<td>Kleene-*</td>
<td>$\text{iterate}(f), \text{left-iterate}(f)$</td>
<td>$R^*$</td>
</tr>
<tr>
<td>Repetition</td>
<td>$\text{combine}(f, g)$</td>
<td></td>
</tr>
<tr>
<td>Chained sum</td>
<td>$\text{chain}(f, R), \text{left-chain}(f, R)$</td>
<td>New!</td>
</tr>
</tbody>
</table>
Future Work

- Implement practical programmer assistance tools
  - Static: Precondition computation, 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.)
  - $\omega$-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
- \( \text{combine}(f, g) \) 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