Regular Programming Over Data Streams

Mukund Raghothaman
Advisor: Rajeev Alur

April 14, 2015

Abstract

With large data streams being increasingly common, such as event streams from sensors, packets at internet routers, and gene sequences, there is an emerging need to program stream transformations. A large class of transformations are instances of simple patterns, such as inserting, deleting and replacing substrings, reordering substrings, or applying a function to each element in the stream. Traditional implementation techniques for these transformations sacrifice either ease of expressivity or amenability to static analysis. We propose a simple, expressive programming model for stream transformations, with strong theoretical foundations, fast evaluation algorithms, and with tools for automated analysis.

We provide a set of basic functions, and a small collection of *combinators*, analogous to the operations of regular expressions. The operators are very natural, and can be used to combine small, easy-to-understand functions into more complicated functions. The functions thus expressible are exactly the class of *regular string transformations*, which is a robust class with multiple characterizations and appealing closure properties (under composition, input reversal, and regular look-ahead). We then present a single-pass linear-time evaluation algorithm for function expressions, and implement a domain specific language, DReX, around the idea of function combinators and express several string transformations in the formalism. We show that the prototype implementation is fast enough for practical use.

We are working on a similar characterization of quantitative functions and evaluation algorithms for the resulting calculus. An example data stream would be a sequence of temperature measurements and electricity meter readings, and a potential query would be to find the temperature on the day when the least amount of electricity was consumed. The aim is to achieve expressive completeness with the more general class of regular functions, i.e. those which are expressible by a regular string-to-term transducer over a parametrized grammar. Other directions of current work include developing practical static analysis and programmer assistance tools, and parallel evaluation algorithms which scale to input streams spread over many servers in a data-center.

Chapter 1

Introduction

Over the last fifty years, strings have emerged as one of the most important data structures in computer science. We now know that genetic information is encoded sequentially, and many cellular processes can be understood as string transformations. With the emergence of the internet, internet service providers have massive packet streams available at their routers. Sensors are now everywhere, and many analysts wish to make sense of event streams to which they have access. A building manager who sees the event stream "`...; Id #437 swiped at card reader; Door opens; Person enters building; Door closes; ...`", may wish to instead see "`...; Person #437 enters building; ...`". String transformations are also important everyday activities for computer users, such as the case of programmers who wish to consistently rename variables in their code.

String transformations are traditionally implemented either using general-purpose programming languages, or domain-specific languages such as sed, AWK, and Perl. All these tools are Turing-complete, and functions expressed in them are difficult to mechanically reason about. On the other hand, attempts to formally verify string transformations [32, 43, 7] build on finite state transducers, and usually provide a programming model which is tightly coupled to the transducer semantics, and limited in expressivity or difficult to use.

We propose a simple programming model to describe functions over strings: many interesting functions can be easily expressed using our abstraction, function expressions can be evaluated quickly, they have a well-understood theory and expressiveness results, and are amenable to static analysis. Our existing work [11, 9] has been devoted to the study of string-to-string transformations; the most important direction of ongoing work is on obtaining similar abstractions for quantitative properties.

We provide a set of basic functions, which are combined into more complicated functions using a small collection of function combinators. Function expressions are therefore modular and easier to write: to appreciate the benefits, consider the function *swapBibtex*, shown in figure 1.0.1. This function takes a BibTEX file as input, and moves the title of each entry to the top. Say that we have already written the function *swapEntry*, which takes a single entry as input and moves its title to the top. Because *swapEntry* deals with exactly one entry as input, it is a simpler function to express and can be tested separately. We then use the iterated sum combinator: $iter(f)$ splits the input string σ into substrings $\sigma = \sigma_1\sigma_2\cdots\sigma_k$, and applies f to each substring σ_i and concatenates the results, and is the analogue of Kleene-* of regular expressions. We can then write:

$$swapBibtex = iter(swapEntry).$$

We may be accused of being arbitrary in our choice of combinators. To answer this charge, we turn to finite state transducers, just as finite state automata inform the study of regular languages. The simplest class of finite state transducers is one-way: on reading each input symbol, they produce a sequence of output symbols, and update the state. Unlike the case of acceptors, two-way finite state transducers are more expressive than their one-way counterparts [4]. One-way transducers lack the ability to do "regular look-ahead": consider the transformation which echoes the daily security-related sensor events at a bank only if a guard does not show up for duty some time during the day. Furthermore, the gap between one-way and two-way finite state

```
@book{Gal1638,
    publisher = {Elzevir},
    place = {Leiden},
    year = {1638},
    title = {Two New Sciences},
    author = {Galileo},
}
```

```
@book{Gal1638,
    title = {Two New Sciences},
    publisher = {Elzevir},
    place = {Leiden},
    year = {1638},
    author = {Galileo},
}
```

Figure 1.0.1: Example application of the transformation *swapBibtex*. The input entry on the left is transformed into the entry on the right. Note that the title attribute is moved to the top.

transducers contains many interesting transformations, such as the function $\sigma \mapsto \sigma^{rev}$ which reverses the input string, and the function $\sigma \mapsto \sigma\sigma$, which repeats the input string twice.

Two-way finite state transducers retain many appealing properties: they are closed under composition [19], can be mechanically checked for equivalence [30], and are expressively equivalent [25] to MSO-definable string-to-string transformations described by [20]. The term "regular string transformations" was first used to describe this class by Engelfriet and Hoogeboom [25]. Alur and Černý then developed the model of streaming string transducers [6], which process the input string in a single left-to-right pass, but maintain multiple write-only registers to store partially computed chunks, and showed that they are expressively equivalent to regular string transformations.

String transducers have recently become an interesting target for formal verification, such as while analyzing string sanitizers [32, 43], list processing programs [7], and in automatically learning string transformations from input-output examples [29]. The decidability of equivalence checking and preimage computation also encourages the choice of regular string transformations as the yardstick for expressive completeness.

It turns out that we need a new combinator, without a regular expression analogue, the *chained sum*, to be able to express all regular string transformations. We motivate this combinator with the example function *alignBibtex*. Imagine a paper author who made a mistake while aligning the entries of a BibTEX file: the title of the first entry now appears in the second entry, the title of the second entry now appears in the third, and so on. See figure 1.0.2. The function *alignBibtex*, given this file as input, moves the title of entry $i+1$ to its correct place in entry i, for each i. We conjecture that *alignBibtex* cannot be expressed using just the iterated sum combinator. The chained sum combinator takes a function f and regular expression R, and produces the new function $chain(f, R)$ which works as follows. Given an input string σ, $chain(f, R)$ first divides it into substrings $\sigma = \sigma_1\sigma_2 \cdots \sigma_k$, where each substring σ_i matches R. It then applies f to each pair of adjacent substrings, $\sigma_i\sigma_{i+1}$, and concatenates the outputs, thus producing $f(\sigma_1\sigma_2)f(\sigma_2\sigma_3) \cdots f(\sigma_{k-1}\sigma_k)$. Let *makeEntry* be the function which takes two entries as input, and outputs the title from the second entry, and the body from the first. The function *alignBibtex* can then be expressed as

$$alignBibtex = chain(makeEntry, R_{Entry}),$$

where R_{Entry} is the regular expression matching a single BibTEX entry.

Finally, an important component of a useful programming framework is fast evaluation algorithms. The naive translation to streaming string transducers is non-elementary in the size of f, and complicated to implement. On the other hand, the natural algorithm to compute $f(\sigma)$, given a function expression f and input string σ, has complexity cubic in the length of the input string. In fact, with combinators such as function composition, just determining whether $f(\epsilon)$ is defined is PSPACE-complete.

The key idea in building a fast evaluation algorithm is to impose *consistency requirements* on function expressions. The consistency requirements limit the ability of function combinators to easily express certain difficult-to-evaluate idioms such as language intersection, and also allow us to declare that $|f(\sigma)| \leq poly(f)|\sigma|$. The consistency requirements *do not limit expressivity*, are simple to describe—they enforce unambiguous parsing of all strings, and are similar to unambiguous regular expressions [16, 41, 17]—and fast to check (the consistency of *alignBibtex* can be verified in ≈ 0.5 seconds). We then present a single-pass evaluation

```
...

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

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

...
```

```
...

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

...
```

Figure 1.0.2: Example application of the transformation *alignBibtex*. In the input on the left, the title of `Book1` has been mistakenly placed in the description of `Book2`, the title of `Book2` in the body of `Book3`, and so on. The output on the right corrects this error.

algorithm, which takes time polynomial in the size of f, and linear in the length of the input string σ. Intuitively, we construct a machine for each sub-expression which reads the input in a single left-to-right pass. The machine tracks potential parse trees of σ as multiple *threads*, and updates the threads on reading each input symbol. The key property we maintain is that the number of threads is linear in the size of f, and independent of the number of input characters read. Achieving this bound involves exploiting the consistency rules and eagerly killing threads which are guaranteed to no longer produce an output. This algorithm is easy to implement, and fast in practice: while the natural cubic-time algorithm for *alignBibtex* becomes impractical for input files roughly $5,000$ characters long, the streaming evaluator gracefully handles input files hundreds of kilobytes long, and processes them at roughly $10,000$ characters per second on commodity desktops.

1.1 Contributions

We make the following specific contributions:

Function combinators We present the first algebraic characterization of regular string transformations, which could previously only be represented operationally (as two-way finite state transducers or streaming string transducers) or logically (as graph transformations defined in monadic second order logic).

Design of DReX and evaluation algorithms We develop DReX, a domain specific language, around the idea of function combinators, and present a single-pass linear-time evaluation algorithm for function expressions.

Implementation and case studies We have implemented DReX and expressed several string transformations in the formalism (see appendix A). We argue that the evaluation algorithm is fast enough for practical use.

Combinators for quantitative cost functions (ongoing) We are currently working on a similar characterization of cost functions. The resulting calculus would be able to express all regular cost functions, and would also come with similar streaming evaluation algorithms.

Portions of the work presented in this proposal have been adapted from:

[1] Rajeev Alur, Loris D'Antoni, Jyotirmoy Deshmukh, Mukund Raghothaman, and Yifei Yuan. Regular functions and cost register automata. In *28th Annual Symposium on Logic in Computer Science*, pages 13–22, 2013.

[2] Rajeev Alur, Loris D'Antoni, and Mukund Raghothaman. DReX: A declarative language for efficiently evaluating regular string transformations. In *Proceedings of the 42nd Annual Symposium on Principles of Programming Languages*, POPL '15, pages 125–137. ACM, 2015.

[3] Rajeev Alur, Adam Freilich, and Mukund Raghothaman. Regular combinators for string transformations. In *Proceedings of the Joint Meeting of the 23rd Annual Conference on Computer Science Logic and the 29th Annual Symposium on Logic in Computer Science*, CSL-LICS '14, pages 9:1–9:10. ACM, 2014.

1.2 Proposal Outline

We define string-to-string function combinators in chapter 2. We will consider the evaluation problem for function expressions in chapter 3. We will describe the streaming evaluation algorithm for consistent DReX expressions, and outline a proof that relaxing the consistency requirements leads to a PSPACE-complete evaluation problem. In chapter 4, we will describe the theory of regular string transformations, and outline the proof that functions expressible using DReX are exactly the class of regular functions. Finally, in chapter 5, we will present ongoing work which we hope to include in the final thesis. We conclude with a rough timeline for meeting these goals in section 5.5.

Chapter 2

Function Combinators

2.1 Definitions

Basic functions. Let Σ and Γ be the input and output alphabets respectively. Then, for each symbol $a \in \Sigma$, and string $\gamma \in \Gamma^*$, we define the basic function $a \mapsto \gamma : \Sigma^* \to \Gamma^*_\perp$ as[1]

$$(a \mapsto \gamma)(\sigma) = \begin{cases} \gamma & \text{if } \sigma = a, \text{ and} \\ \perp & \text{otherwise.} \end{cases}$$

The second class of basic functions is $\epsilon \mapsto \gamma : \Sigma^* \to \Gamma^*_\perp$, for each $\gamma \in \Gamma^*$, which maps the empty input string ϵ to the output γ, and is undefined everywhere else. Finally, the *everywhere-undefined function* $bottom : \Sigma^* \to \Gamma^*_\perp$ is defined as $bottom(\sigma) = \perp$.

Conditional choice. If f and g are two string transformations, then we define the *conditional choice* f *else* g as

$$f \textit{ else } g(\sigma) = \begin{cases} f(\sigma) & \text{if } f(\sigma) \neq \perp, \text{ and} \\ g(\sigma) & \text{otherwise.} \end{cases}$$

Combine and split sums. The function $combine(f, g)$ of two functions $f, g : \Sigma^* \to \Gamma^*_\perp$ is defined as $combine(f, g)(\sigma) = f(\sigma)g(\sigma)$. If there exist unique strings σ_1 and σ_2 such that $\sigma = \sigma_1\sigma_2$, and $f(\sigma_1)$ and $g(\sigma_2)$ are both defined, then the *split sum* $split(f, g)(\sigma) = f(\sigma_1)g(\sigma_2)$. Otherwise, $split(f, g)(\sigma) = \perp$. Because string concatenation is non-commutative, in certain cases, this may be different from the *left-split sum* *left-split*(f, g): if there exist unique strings σ_1 and σ_2, such that $\sigma = \sigma_1\sigma_2$, and $f(\sigma_1)$ and $g(\sigma_2)$ are both defined, then *left-split*$(f, g)(\sigma) = g(\sigma_2)f(\sigma_1)$. Otherwise, *left-split*$(f, g)(\sigma) = \perp$.

Iteration. The iterated sum $iter(f)$ of a string transformation f is defined as follows. If there exist unique strings σ_1, σ_2, ..., σ_k, such that $\sigma = \sigma_1\sigma_2 \cdots \sigma_k$, and $f(\sigma_i)$ is defined for each i, then $iter(f)(\sigma) = f(\sigma_1)f(\sigma_2) \cdots f(\sigma_k)$. Otherwise, $iter(f)(\sigma)$ is undefined. Similarly, if there exist unique strings σ_1, σ_2, ..., σ_k, such that $\sigma = \sigma_1\sigma_2 \cdots \sigma_k$, and $f(\sigma_i)$ is defined for each i, then *left-iter*$(f)(\sigma) = f(\sigma_k)f(\sigma_{k-1}) \cdots f(\sigma_1)$. Otherwise *left-iter*$(f)(\sigma) = \perp$.

Chained sums. Let R is a regular expression over Σ, and $f : \Sigma^* \to \Gamma^*_\perp$ be a string transformation. If there exists a unique decomposition $\sigma = \sigma_1\sigma_2 \ldots \sigma_k$ such that $k \geq 2$ and for each i, $\sigma_i \in [\![R]\!]$, then the *chained sum*

$$chain(f, R)(\sigma) = f(\sigma_1\sigma_2)f(\sigma_2\sigma_3) \cdots f(\sigma_{k-1}\sigma_k).$$

[1] We adopt the convention of writing $f(x) = \perp$ when f is undefined for the input x, and write $A_\perp$ for $A \cup \{\perp\}$, when $\perp \notin A$.

Otherwise, $\mathit{chain}(f, R)(\sigma) = \bot$. Similarly, if there exist unique strings σ_1, σ_2, ..., σ_k such that $k \geq 2$ and for all i, $\sigma_i \in [\![R]\!]$, then the *left-chained sum*

$$\mathit{left\text{-}chain}(f, R)(\sigma) = f(\sigma_{k-1}\sigma_k)f(\sigma_{k-2}\sigma_{k-1}) \cdots f(\sigma_1\sigma_2).$$

Otherwise, $\mathit{left\text{-}chain}(f, R)(\sigma) = \bot$.

Domain restriction. If $f : \Sigma^* \to \Gamma^*_\bot$ is a string transformation, and R is a regular expression over Σ, then the function $\mathit{restrict}(f, R)$ is defined as follows:

$$\mathit{restrict}(f, R)(\sigma) = \begin{cases} f(\sigma) & \text{if } \sigma \in [\![R]\!], \text{ and} \\ \bot & \text{otherwise.} \end{cases}$$

Function composition. If $f : \Sigma^* \to \Gamma^*_\bot$ and $g : \Gamma^* \to \Lambda^*_\bot$, then the function $\mathit{compose}(f, g)$ is defined as follows: if $f(\sigma)$ is defined, then $\mathit{compose}(f, g)$ maps the input string σ to the output $g(f(\sigma))$, and is otherwise undefined.

Remarks. Note the emphasis we placed on ensuring that the definitions were unambiguous. This is the principal difference between the definitions here, and those of traditional regular expressions.

For example, consider the traditional concatenation operator $R_1 \cdot R_2$ in regular expressions. If there were a string σ which can be split in two different ways, $\sigma = \sigma_1\sigma_2 = \sigma_3\sigma_4$, such that R_1 accepts both σ_1 and σ_3, and R_2 accepts both σ_2 and σ_4, then the entire string σ is still accepted by $R_1 \cdot R_2$. On the other hand, if all of $f(\sigma_1)$, $f(\sigma_3)$, $g(\sigma_2)$, and $g(\sigma_4)$ are defined, then it may not necessarily be the case that $f(\sigma_1)g(\sigma_2) = f(\sigma_3)g(\sigma_4)$, and this would cause a problem in assigning an unambiguous value to $\mathit{split}(f, g)(\sigma)$. The easiest way to avoid this difficulty is to define $\mathit{split}(f, g)(\sigma)$ only when the split $\sigma = \sigma_1\sigma_2$ into inputs for f and g is unique.

2.2 Large Alphabets and Character Predicates

Consider the basic combinator $a \mapsto d$, which maps the input $\sigma = a$ to the output d. For large alphabets, such as the set of all Unicode characters ($\approx 110,000$ characters and growing), this approach of explicitly mentioning each character does not scale. Basic transformations may therefore also reference symbolic predicates and character functions, as we will now describe. This is inspired by the recent development of symbolic transducers [43], which has proved to be useful in several practical applications.

Let Σ, Γ, ... be a collection of alphabets. For each alphabet Σ, we pick a (possibly infinite) collection of predicates P_Σ such that:

1. P_Σ is closed under the standard boolean operations: for each $\varphi, \psi \in P_\Sigma$, $\neg\varphi, \varphi \wedge \psi, \varphi \vee \psi \in P_\Sigma$, and
2. the satisfiability of predicates is decidable: given $\varphi \in P_\Sigma$, whether there exists an $x \in \Sigma$ such that $\varphi(x)$ holds is decidable.

A simple example is $\Sigma_2 = \{a, b\}$ together with the set of predicates $P_{\Sigma_2} = \{x = a, x = b, \mathit{true}, \mathit{false}\}$. Another example is the set of all integers $\mathbb{Z}$, and with $P_\mathbb{Z} = \{\mathit{isOdd}(x), \mathit{isEven}(x), \mathit{true}, \mathit{false}\}$. We will write $\mathbb{U}$ for the set of all Unicode characters, with the various character properties $P_\mathbb{U} = \{\mathit{isUpperCase}(x), \mathit{isDigit}(x), \ldots\}$. We also need to express character transformations, such as $\mathit{toLowerCase} : \mathbb{U} \to \mathbb{U}$. For mechanical static analysis, we require the following: For each pair of alphabets Σ and Γ, for each pair of character transformations $d_1, d_2 : \Sigma \to \Gamma$, and for each predicate $p_\Gamma \in P_\Gamma$, the predicates $d_1(x) = d_2(x)$ and $p_\Gamma(d_1(x))$ occur in P_Σ. If $\varphi \in P_\Sigma$ and $\gamma = [d_1, d_2, \ldots, d_k]$ is a list of character transformations, i.e. $d_i : \Sigma \to \Gamma$, then $\varphi \mapsto \gamma$ is a basic transformation which maps every single-character string σ which satisfies $\varphi(\sigma)$ to the output string $d_1(\sigma)d_2(\sigma) \cdots d_k(\sigma)$, and is undefined for all other strings.

Example 1. The function $isUpperCase(x) \mapsto toLowerCase(x)$ transforms every upper-case Unicode character to lower-case, while the function $isUpperCase(x) \mapsto xx$ outputs two copies of an upper-case character. The function $x \geq 0 \mapsto x - 1$ transforms a non-negative integer by subtracting one from it. Given an input digit $x \in [2, 3, \ldots, 9]$, the function $x \in [2, 3, \ldots, 9] \mapsto x - 2$ subtracts 2 from it.

Note that the basic symbolic transformations can still only operate on individual characters in isolation, and cannot relate properties of adjacent characters. For example, we do not allow transformations such as $[x > 0, y > x] \mapsto x, y$, which would conceivably output two consecutive symbols x and y, if $x > 0$ and $y > x$. It is known that allowing such "multi-character predicates" makes several analysis questions undecidable [22].

2.3 Consistency Requirements

We now define consistent function expressions. This is a restricted class of function expressions which is still expressively complete, but for which we can provide an efficient evaluation algorithm. Intuitively, we restrict each operator to only allow unambiguous parsing, and limit the operators' ability to express expensive automata operations such as intersection and complement. Since the main purpose of the consistency rules is for the correctness of the evaluation algorithm, we defer a convincing motivation to subsection 3.1.6.

2.3.1 Unambiguous regular expressions

The consistency rules we propose are based on the notion of unambiguous regular expression (UREs) [16, 41, 17]. UREs are similar to conventional regular expressions, but with the additional guarantee that all matched strings have unique parse trees. They are defined inductively as follows:

1. $\emptyset$ and ϵ are UREs.
2. For each satisfiable predicate $\varphi \in P_\Sigma$, φ is a URE.
3. For each pair of non-empty UREs R_1 and R_2, if the associated languages $L_1 = [\![R_1]\!]$ and $L_2 = [\![R_2]\!]$ are disjoint, then $R_1 \cup R_2$ is also a URE.
4. Given a pair of non-empty UREs R_1 and R_2, we say that they are *unambiguously concatenable*, if for each string $\sigma \in \Sigma^*$, there is at most one split $\sigma = \sigma_1\sigma_2$ such that $\sigma \in [\![R_1]\!]$ and $\sigma_2 \in [\![R_2]\!]$. If R_1 and R_2 are unambiguously concatenable, then $R_1 \cdot R_2$ is also a URE.
5. A non-empty URE R is *unambiguously iterable* if for every string σ, there is at most one split $\sigma = \sigma_1\sigma_2 \cdots \sigma_k$ into substrings such that $\sigma_i \in [\![R]\!]$ for each i. If R is unambiguously iterable, then R^* is also a URE.

For example, the regular expressions φ and $(\neg\varphi)^*$ are unambiguously concatenable for every character predicate φ: every string σ matching $\varphi \cdot (\neg\varphi)^*$ has to be split after the first character. On the other hand, Σ^* is not unambiguously concatenable with itself: there are three ways to parse the string aa in $\Sigma^* \cdot \Sigma^*$, because the left part of the concatenation can either match ϵ, a, or aa. The regular expression Σ^* is unambiguous—there is only one way to split each string σ such that each substring is in Σ—but $(\Sigma^*)^*$ is not unambiguous.

We call two UREs R_1 and R_2 *equivalent*, and write $R_1 \equiv R_2$, if $[\![R_1]\!] = [\![R_2]\!]$.

2.3.2 Consistency rules for function expressions

A consistent function expression is one that satisfies the rules defined in this section. One major effect of these rules is to guarantee that no string has multiple parse trees, so the word "unique" in the combinator definitions is unnecessary. The domain of a transformation $f : \Sigma^* \to \Gamma^*_\perp$ is the set containing every string σ such that $f(\sigma)$ is defined.

1. All basic functions *bottom*, $\epsilon \mapsto d$, and $\varphi \mapsto d$ (where φ is satisfiable) are consistent. Their domains are $\emptyset$, ϵ and φ respectively.

2. If f and g are consistent with disjoint domain types R_f and R_g respectively, then f *else* g is also consistent with the domain $R_f \cup R_g$.

3. If f and g are both consistent with domain types R_f and R_g respectively, and $R_f \equiv R_g$, then $combine(f, g)$ is also consistent and has domain R_f.

4. If f and g are both consistent and have unambiguously concatenable domain types R_f and R_g respectively, then $split(f, g)$ and $left\text{-}split(f, g)$ are also both consistent and have the domain $R_f \cdot R_g$.

5. If f is consistent and has domain R, and R is unambiguously iterable, then $iter(f)$ and $left\text{-}iter(f)$ are both consistent, and both have domain R^*.

6. If f is consistent, and has domain R_f, R_f is unambiguously iterable, and R is a URE with $R \equiv R_f \cdot R_f$, then $chain(f, R)$ and $left\text{-}chain(f, R)$ are both consistent and have domain $R_f \cdot R_f \cdot R_f^*$.

Note specifically that expressions involving function composition are not consistent. Programs containing the *restrict* combinator are also not consistent. This is just a historical accident, and will be corrected in the final thesis (section 3.3). In the rest of this proposal, to distinguish the class of consistent expressions from the bigger class of *all* function expressions, we will explicitly qualify the latter as the unrestricted class. The consistency and domain computation rules are syntax-directed, and straightforward to implement directly. We need to be able to answer the following basic questions about unambiguous regular expressions:

1. "Given UREs R_1 and R_2, are R_1 and R_2 unambiguously concatenable?", "Given a URE R, is it unambiguously iterable?", "Given UREs R_1 and R_2, are they disjoint, or equivalently, is $R_1 \cup R_2$ also a URE?" Observe that the traditional algorithm [40] to convert regular expressions to NFAs converts unambiguous regular expressions to unambiguous NFAs, where each accepted string has exactly one accepting path. Whether a regular expression R is unambiguous can therefore be checked in polynomial time [17]: take the product of the corresponding (ϵ-transition free) NFA A_R with itself, and check for the presence of a reachable state (q, q'), with $q \neq q'$, which can itself reach a pair of accepting states $(q_f, q'_f) \in F \times F$, where F is the set of accepting states of A_L. Thus, if the input alphabet Σ is finite, these questions can be answered in polynomial time. Otherwise, the same problems for symbolic automata (representing R_1, R_2, etc.) are also decidable in polynomial time assuming that we can check in polynomial time whether a predicate is satisfiable.

2. "Given UREs R_1 and R_2, is $R_1 \equiv R_2$?" If Σ is finite, then from [41], we have that this can be checked in time $O(poly(|R_1|, |R_2|, |\Sigma|))$. Otherwise, if UREs are expressed using the symbolic notation of section 2.2, they can be translated into symbolic automata, and the equivalence of symbolic automata is decidable in polynomial time in the size of R_1 and R_2 and exponential (in the worst case) in the number of predicates appearing in R_1 and R_2.

Theorem 2. *Given a function expression f over an input alphabet Σ, checking whether f is consistent is decidable. Furthermore, if the input alphabet Σ is finite, then the consistency of f can be determined in time $O(poly(|f|, |\Sigma|))$.*

2.4 Examples of Function Expressions

The simplest non-trivial function expression is the identity function $id = iter(true \mapsto x)$. Several variations of this function are also useful: $iter(isLowerCase(x) \mapsto toUpperCase(x))$ maps strings of lower-case characters to upper-case, and $id_{\neg space} = iter(\neg isSpace(x) \mapsto x)$ is the identity function restricted to strings not containing a space.

Table 2.1: Benchmark expressions with sizes (number of AST nodes) and time to check consistency.

Function name	Function size	Time
deleteComments	28	12 ms
insertQuotes	28	6 ms
getTags	31	6 ms
reverse	5	1 ms
swapBibtex	1663	262 ms
alignBibtex	3652	537 ms

More interesting functions can be constructed using the conditional operator: the function $swCase = isUpperCase(x) \mapsto toLowerCase(x)$ *else* $isLowerCase(x) \mapsto toUpperCase(x)$ flips the case of a single input character, and so $iter(swCase)$ switches the case of each character in the input string.

Given a string of the form "`FirstName LastName`", the function $echoFirst = split(id_{\neg \texttt{space}}, isSpace(x) \mapsto \epsilon, iter(true \mapsto \epsilon))$ outputs "`FirstName`". Similarly, the function *echoLast* which outputs the last name could be written, and the two can be combined into $combine(echoLost, echoFirst)$, which outputs "`LastNameFirstName`". Note that the space in between is omitted—the expression $combine(split(echoLast, \epsilon \mapsto \texttt{" "}), echoFirst)$ preserves this space. An example of the use of the left-additive operators is in string reversal: the function $left\text{-}iter(true \mapsto x)$ reverses the input string.

We have already seen an application of the chained sum combinator while describing *alignBibtex* in the introduction. Recall that we wrote

$$alignBibtex = chain(makeEntry, R_{Entry}),$$

where R_{Entry} is the regular expression matching a single BibTEX entry, and *makeEntry* mapped a pair of entries to a single new entry containing the title of the second entry and the body of the first. In appendix A.5.3, we define the following functions: *headerOnlyFromEntry*, which maps a single entry to just its header (for example, `@book{Book1`), *titleOnlyFromEntry*, which maps an entry to its title (for example, `title = {Title0},`) and *allButTitleFromEntry* which maps a single entry to just its body. Similarly, the expression *deleteEntry* maps the entire BibTEX entry to the empty string ϵ, and R_{Entry} is the regular expression describing a single entry. Then,

$$\begin{aligned} makeEntry = combine(&split(headerOnlyFromEntry, titleOnlyFromEntry),\\ &split(allButTitleFromEntry, deleteEntry)). \end{aligned}$$

2.5 Experimental Evaluation of Consistency Checking

The prototype implementation of DReX is written in Java, and uses the symbolic automata library SVPAlib [21] for the consistency checking subroutines. In table 2.1, we present the time by our prototype implementation to type-check some benchmark function expressions.[2] These function expressions are described in appendix A. Note that consistency checking is not prohibitive in practice.

In our experience, consistency checking is not a burden on the programmer and often discovers subtle problems in the functions expressed. For example, consider a decoder for a prefix code

$$\begin{aligned} decodePrefixCodeChar = \texttt{"0"} \mapsto \texttt{"a"}\ &else\ \texttt{"10"} \mapsto \texttt{"b"}\\ &else\ \texttt{"110"} \mapsto \texttt{"c"}\\ &else\ \texttt{"111"} \mapsto \texttt{"d"}.\\ decodePrefixCode = iter(&decodePrefixCodeChar). \end{aligned}$$

[2]The experiments were run on regular contemporary hardware: Windows 7 running on a 64-bit quad-core Intel Core i7-2600 CPU, at 3.40 GHz with 8 GB of RAM. The results reported are the mean of 10 runs.

Note that *decodePrefixCode* is a consistent expression. Now, instead of a prefix code, let us target (a subset of) Morse code:

$$\begin{aligned} \mathit{decodeMorseChar} &= \texttt{".-"} \mapsto \texttt{"a"}\ \mathit{else}\ \texttt{"-.."} \mapsto \texttt{"d"}\ \mathit{else}\ \texttt{"."} \mapsto \texttt{"e"},\ \text{in} \\ \mathit{decodeMorse} &= \mathit{iter}(\mathit{decodeMorseChar}). \end{aligned}$$

The consistency checker promptly complains about the string "`.-..`", which can be decoded either as "`aee`", or as "`ed`".[3] This is very similar to the experience reported by the developers of Boomerang [14], whose type system discovered ambiguities in interchange formats such as SwissProt.

2.6 Ongoing Work

The current DReX front-end [10] is an early prototype. With the hindsight of having written and debugged actual DReX expressions, we now believe that defining custom combinators is an important part of developing large expressions. It is therefore more useful to have DReX embedded in a full-fledged scripting language. The static analysis tools currently in development (section 5.2) would also be deeply integrated with this front-end. We expect to have a more stable and useful DReX front-end by the conclusion of this thesis.

[3] Telegraph operators resolve such ambiguities by using inter-character and inter-word gaps, effectively adding a third "comma" character into the code.

Chapter 3

Evaluation Algorithms

In this chapter, we consider the evaluation problem for DReX expressions: Given a function expression f, and an input string σ, how do we determine $f(\sigma)$? We start by describing the promised single-pass linear-time evaluation algorithm in section 3.1. This algorithm crucially requires that the expressions under consideration be consistent. In section 3.2, we consider the problem of evaluating unrestricted DReX expressions: we outline a proof that this problem is PSPACE-complete. Finally, in section 3.3 we describe ongoing work on the DReX implementation.

3.1 A Single-Pass Algorithm for Consistent DReX

Given a consistent DReX expression f, the idea is to construct an evaluator T which computes the associated function. The evaluator T processes the input string from left-to-right, one character at a time. After reading each character, it outputs the value of f on the string read so far, if it is defined.

To understand the input / output specifications of T, we consider the example program $\mathit{split}(f, g)$. In this case, T is given the sequence of input signals $(\mathit{Start}, 0), (\sigma_1, 1), (\sigma_2, 2), \ldots, (\sigma_n, n)$. The first signal $(\mathit{Start}, 0)$ indicates the beginning of the string, and each character σ_i is annotated with its index i in the input string. After reading (σ_i, i), T responds with the value of $\mathit{split}(f, g)$ on $\sigma_1\sigma_2 \ldots \sigma_i$, if it is defined.

Assume that f and g are consistent, and have unambiguously concatenable domain types R_f and R_g respectively. The evaluator T maintains two sub-evaluators T_f and T_g for the functions f and g respectively. Each time T receives the input (a, i), it forwards this signal to both T_f and T_g. Whenever T_f reports a result, i.e. that f is defined on the input string read so far, T sends the signal *Start* to T_g to start processing the suffix. Consider the situation in figure 3.1.1, where f is defined for the prefixes $\sigma_1\sigma_2 \ldots \sigma_i$ and $\sigma_1\sigma_2 \ldots \sigma_j$. The input to the sub-evaluator T_g is then the sequence $(\sigma_1, 1), (\sigma_2, 2), \ldots, (\sigma_i, i), (\mathit{Start}, i), (\sigma_{i+1}, i+1), \ldots, (\sigma_j, j), (\mathit{Start}, j), \ldots, (\sigma_n, n)$.

For each signal (Start, i) occurring in the input string, we call the subsequent sequence of characters $\sigma_{i+1}\sigma_{i+2} \ldots$ the *thread* beginning at index i. Note that each thread corresponds to a potential parse tree of σ, and that T_g may be processing multiple such threads simultaneously. The main challenge is to ensure that the number of active threads in T_g is bound by $O(|g|)$, and is independent of the length of the input string. After reading σ_n, T_g reports a result to T, the evaluator for the $\mathit{split}(f, g)$. To uniquely identify the thread j reporting the result, the result signal $(\mathit{Result}, j, \gamma_g)$ is annotated with the index j at which the corresponding *Start* was received.

Note that the consistency rules guarantee that, after reading each input symbol, T_g emits at most one result, for otherwise the prefix of the input string read so far would have multiple parse trees.

When T receives this result signal from T_g, it combines it with the response $(\mathit{Result}, 0, \gamma_f)$ initially obtained from T_f at position j, and itself emits the result $(\mathit{Result}, 0, \gamma_f\gamma_g)$. To do this, it maintains a set th_g (for threads) of triples $(i_{0f}, i_{0g}, \gamma_f)$, where i_{0f} is the index along the input string at which T_f was started, i_{0g} was the index at which T_f reported a result and T_g was started, and γ_f was the result reported by T_f. In

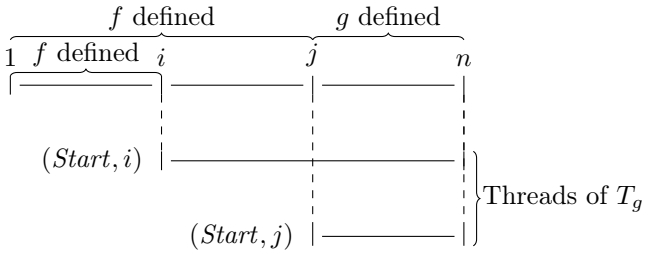


Figure 3.1.1: Example run of the evaluator T for $split(f, g)$ over the string σ. The evaluator T_f emits a result at indices i and j of the input string. The evaluator T_g for g may simultaneously be processing multiple threads, corresponding to different potential parse trees of the input string σ. From the consistency rules, we know that at most one thread may return a result at each index, and so T can safely emit a result in response to getting a result from T_g.

order to prevent this set th_g from becoming too large, T_g emits *kill signals*. Say that, at index k, T_g discovers that for every possible suffix $\tau \in \Sigma^*$, g will be undefined for the string $\sigma_{i+1}\sigma_{i+2}\ldots\sigma_k\tau$, and so the thread $(Start, i)$ of T_g initiated at the input index i can never return a result. It then emits $(Kill, i)$ to signal to T that the relevant entries in the set th_g can be deleted.

3.1.1 Formal specification of the evaluator

The input alphabet to each function evaluator is $In = (\Sigma \cup \{Start\}) \times \mathbb{N}$ and the output alphabet is $Out = (\{Result\} \times \mathbb{N} \times \Gamma^*) \cup (\{Kill\} \times \mathbb{N})$, where Σ is the input and Γ is the output alphabet of the DReX expression.

While constructing the evaluator T for a DReX expression f, we assume the following condition of *input validity*: for each prefix of the input stream in, there is at most one thread for which f is defined. Thus, for example, T can never see two consecutive $(Start, i)$ signals for the same i. In return, we make the following guarantees:

Correctness of results. After reading each input signal (σ_j, j) in in, we report the result $(Result, i, \gamma)$ exactly for that thread $(Start, i)$ such that $f(\sigma_{i+1}\sigma_{i+2}\ldots\sigma_j) = \gamma$, if it exists.

Eagerness of kills. Every thread σ beginning at $(Start, i)$ of in, such that there is no suffix τ for which $f(\sigma\tau)$ is defined, is killed exactly once while reading in. Furthermore, there are always at most $O(|f|)$ active threads, where $|f|$ is the size (in number of subexpressions) of f.

If an evaluator T satisfies these requirements for f, then we say that the evaluator *computes* f. On the input $(Start, 0), (\sigma_1, 1), (\sigma_2, 2)\ldots, (\sigma_n, n)$, the evaluator outputs a result γ in exactly those cases when f is defined, and in that case, $f(\sigma) = \gamma$. We will ensure that the evaluator T processes each input signal in time $O(poly(|f|))$.[1]

3.1.2 Basic evaluators

The simplest case is when $f = bottom$. The evaluator $T_\perp$ is defined by the following rules:

1. On input $(Start, i)$, respond with $(Kill, i)$.
2. On input (a, i), for $a \in \Sigma$, do nothing.

[1]We assume a representation for strings with concatenation requiring only constant time. Specifically, strings are only concatenated symbolically using a pointer representation. Such "lazily" represented strings can be converted into the traditional sequence-of-characters representation in time linear in the string length.

Next, we consider the evaluator $T_{\epsilon \mapsto \gamma}$, for the case when $f = \epsilon \mapsto \gamma$, for some $\gamma \in \Gamma^*$. Intuitively, this evaluator returns a result immediately on receiving a start signal, but can only kill the thread after reading the next symbol. It therefore maintains a set $th \subseteq \mathbb{N}$ of currently active threads, which are to be killed on reading the next input symbol. The set th is initialized to $\emptyset$.

1. On input (Start, i), respond with $(\mathit{Result}, i, \gamma)$. Update $th \coloneqq th \cup \{i\}$.
2. On input (a, i), for $a \in \Sigma$, respond with (Kill, j), for each thread start index $j \in th$. Update $th \coloneqq \emptyset$.

Observe that by the condition of input validity, we can never observe two consecutive start signals in the input stream. Therefore, $|th| \leq 1$, and the response time of $T_{\epsilon \mapsto \gamma}$ to each input signal is bounded by a constant.

The final basic function is $f = a \mapsto \gamma$ for some $a \in \Sigma$ and $\gamma \in \Gamma^*$. The evaluator $T_{a \mapsto \gamma}$ maintains two sets $th, th' \subseteq \mathbb{N}$ of thread start indices, initialized to $th = th' = \emptyset$. th is the set of threads for which no symbol has yet been seen, while th' is the set of threads for which one input symbol has been seen, and that input symbol was equal to a.

1. On input (Start, i), update $th \coloneqq th \cup \{i\}$.
2. On input (c, i), for $c \in \Sigma$:
   (a) Emit (Kill, j), for each thread $j \in th'$.
   (b) If $c = a$, for each thread $j \in th$, emit $(\mathit{Result}, j, \gamma)$. Update $th' \coloneqq th$, and $th \coloneqq \emptyset$.
   (c) If $c \neq a$, then for each thread $j \in th$, emit (Kill, j). Update $th \coloneqq \emptyset$, and $th' \coloneqq \emptyset$.

Just as in the case of $\epsilon \mapsto \gamma$, we have $|th|, \left|th'\right| \leq 1$, and so $T_{a \mapsto \gamma}$ responds to each input signal in time bounded by some constant.

3.1.3 State-free evaluators: Combine and conditional choice

The simplest non-trivial evaluator is for $\mathit{combine}(f, g)$. Recall that, by the consistency requirements, we have $R_f \equiv R_g$ for the domain types R_f and R_g of the sub-expressions. Thus, all state can be maintained by the sub-evaluators T_f and T_g and $T_{\mathit{combine}(f,g)}$ can be entirely state-free. It has the following behavior:

1. On input (Start, i), send the signal (Start, i) to both sub-evaluators T_f and T_g.
2. On input (a, i), send the signal (a, i) to both T_f and T_g.
3. On receiving the result $(\mathit{Result}, i, \gamma_f)$ from T_f and the result $(\mathit{Result}, i, \gamma_g)$ from T_g (which, according to the consistency requirements for $\mathit{combine}(f, g)$, have to occur simultaneously), respond with $(\mathit{Result}, i, \gamma_f \gamma_g)$.
4. On receiving the kill signals (Kill, i) from T_f and T_g (by the consistency rules, necessarily simultaneously), emit the kill signal (Kill, i).

The evaluator $T_{f\mathit{elseg}}$ maintains two sub-evaluators T_f and T_g. In addition, it maintains two sets $th_f, th_g \subseteq \mathbb{N}$ of threads currently active in T_f and T_g respectively. Both sets are initialized to $\emptyset$. The behavior of $T_{f\mathit{elseg}}$ is defined as follows:

1. On receiving the input (Start, i), update $th_f \coloneqq th_f \cup \{i\}$, and $th_g \coloneqq th_g \cup \{i\}$. Send the start signal (Start, i) to both T_f and T_g.
2. On receiving the input (a, i) for some $a \in \Sigma$, send the input (a, i) to both T_f and T_g.
3. When either T_f or T_g respond with the result $(\mathit{Result}, i, \gamma)$ (by the consistency rules, we know that the other sub-evaluator is not responding with a result), emit the result $(\mathit{Result}, i, \gamma)$.

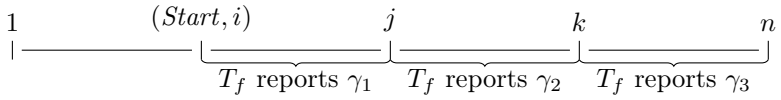


Figure 3.1.2: For each thread $(Start, i_0)$ of the evaluator $T_{iter(f)}$, there may be multiple potential parse trees. The evaluator $T_{iter(f)}$ maps individual threads $(Start, i)$ of T_f to the corresponding start signal $(Start, i_0)$ in $T_{iter(f)}$ through the entry (i_0, i, γ) in the set *th*. Thus, for example, after obtaining the response from T_f at index n, $T_{iter(f)}$ updates $th \coloneqq th \cup \{(i, n, \gamma_1\gamma_2\gamma_3)\}$.

4. If a kill signal $(Kill, i)$ is received from T_f (resp. T_g), update $th_f \coloneqq th_f \setminus \{i\}$ (resp. $th_g \coloneqq th_g \setminus \{i\}$). If $i \notin th_f$ and $i \notin th_g$, then kill the thread by emitting $(Kill, i)$.

The sizes of th_f and th_g are bounded by the number of active threads of T_f and T_g respectively, and hence it follows that T_{felseg} responds to each input signal in time $O(|f| + |g|) + t_f + t_g$, where t_f and t_g are the response times of T_f and T_g respectively.

3.1.4 Stateful evaluators: Iteration and split sum

We now construct evaluators for *iter*(f) and *split*(f, g). The evaluators for *left-iter*(f) and *left-split*(f, g) are symmetric with respect to concatenation and can be constructed similarly.

First, we build the evaluator $T_{iter(f)}$, where f is consistent and has the unambiguously iterable domain type R_f. Whenever $T_{iter(f)}$ receives a start signal $(Start, i)$, or an input signal (a, i), this is passed to T_f. Consider a sequence of input signals σ, as shown in figure 3.1.2. After reading each input symbol, say (σ_n, n), T_f may report that f is defined for a suffix of the input stream $(Start, k), (\sigma_{k+1}, k+1), \ldots, (\sigma_n, n)$ seen so far. The evaluator $T_{iter(f)}$ responds by initiating a new thread of T_f by sending it the start signal $(Start, n)$. Furthermore, it has to record the result $(Result, k, \gamma_3)$ just reported by T_f. It does this by adding the entry $(i, n, \gamma_1\gamma_2\gamma_3)$ to the set *th*. Each entry $(i_0, j_0, \gamma) \in th$ refers to an active thread j_0 of T_f, the index of the signal $(Start, i_0)$ received by $T_{iter(f)}$, and the cumulative result γ obtained so far.

Formally, the set $th \subseteq \mathbb{N} \times \mathbb{N} \times \Gamma^*$ is initialized to $\emptyset$. The evaluator $T_{iter(f)}$ does the following:

1. On input $(Start, i)$:
   (a) Update $th \coloneqq th \cup \{(i, i, \epsilon)\}$.
   (b) Send $(Start, i)$ to T_f. Assert that T_f does not respond with a result $(Result, i, \gamma)$, because by the consistency rules, $f(\epsilon)$ is undefined for R_f to be unambiguously iterable.
   (c) Respond with the result $(Result, i, \epsilon)$.
2. On input (a, i), send the signal (a, i) to T_f. For each response of T_f, do the following:
   (a) If the response is a result, $(Result, j, \gamma_f)$, then, find the corresponding entry $(j_0, j, \gamma) \in th$, for some values of j_0 and γ. Assert (by the invariant that *th* records the active threads of T_f) that this entry exists, and is unique.
      i. Update $th \coloneqq th \cup \{(j_0, i, \gamma\gamma_f)\}$.
      ii. Send the signal $(Start, i)$ to T_f. Confirm that T_f does not respond with a result $(Result, i, \gamma'_f)$, for that would violate the consistency requirements.
      iii. Respond with the result $(Result, j_0, \gamma\gamma_f)$.
   (b) If the response is a kill signal, $(Kill, j)$:
      i. Let *kill-ring* be the set of all tuples $(j_0, j, \gamma) \in th$, for some values of j_0 and γ. By the consistency requirements, *kill-ring* is asserted to be a singleton set.
      ii. Update $th \coloneqq th \setminus$ *kill-ring*.

iii. For every entry $(j_0, j, \gamma) \in \textit{kill-ring}$ if there is no entry of the form $(j_0, j', \gamma') \in th$, then emit the kill signal $(\textit{Kill}, j_0)$.

Observe that an element is added to th exactly when it is sent a start signal, and an entry is deleted exactly when $T_{iter(f)}$ receives a kill signal. Thus, the entries of th correspond to the active threads of T_f, and its size is bounded by $O(|f|)$. The response time of $T_{iter(f)}$ to each input signal is therefore $O(|f|) + t_f$, where t_f is the response time of T_f.

The evaluator $T_{split(f,g)}$ for $\textit{split}(f, g)$ is similar, except that it maintains two sets: the first set $th_f \subseteq \mathbb{N}$ is the set of thread start indices which are still active in T_f, and the second set $th_g \subseteq \mathbb{N} \times \mathbb{N} \times \Gamma^*$ is the set of triples (i_0, i, γ_f) which indicates, for each active thread in T_g, the index i at which T_g was signaled to start, the index i_0 of the original start received by $T_{split(f,g)}$, when T_f was started, and the value γ_f returned by T_f on the prefix. Both sets are initialized to $\emptyset$, and $T_{split(f,g)}$ follows the following rules:

1. On input $(\textit{Start}, i)$:
   (a) Update $th_f \coloneqq th_f \cup \{i\}$.
   (b) Send $(\textit{Start}, i)$ to T_f. Let $\textit{Rsp}_f$ be the responses from T_f. Let $\textit{Rsp}_g = \emptyset$.
2. On input (a, i):
   (a) Send (a, i) to T_f. Let $\textit{Rsp}_f$ be the set of responses from T_f.
   (b) Send (a, i) to T_g. Let $\textit{Rsp}_g$ be the set of responses from T_g.
3. For each response $r \in \textit{Rsp}_f$, do the following:
   (a) If r is a result $(\textit{Result}, j_0, \gamma_f)$ from T_f,
      i. Update $th_g \coloneqq th_g \cup \{(j_0, i, \gamma_f)\}$.
      ii. Send the signal $(\textit{Start}, i)$ to T_g, and let $\textit{Rsp}'_g$ be the set of its responses. Update $\textit{Rsp}_g \coloneqq \textit{Rsp}_g \cup \textit{Rsp}'_g$.
   (b) If r is a kill signal $(\textit{Kill}, j_0)$ from T_f,
      i. Update $th_f \coloneqq th_f \setminus \{j_0\}$.
      ii. If there is no element of the form $(j_0, j, \gamma_f) \in th_g$, for some values of j, γ_f, kill the thread: emit $(\textit{Kill}, j_0)$.
4. For each response $r \in \textit{Rsp}_g$, do the following:
   (a) If r is a result $(\textit{Result}, j, \gamma_g)$, let (j_0, j, γ_f) be the (by the consistency rules, necessarily unique) corresponding record in th_g. Respond with $(\textit{Result}, j_0, \gamma_f\gamma_g)$.
   (b) If r is a kill signal $(\textit{Kill}, j)$,
      i. Let *kill-ring* be the set of tuples $(j_0, j, \gamma_f) \in th_g$ for some values of j_0, γ_f.
      ii. Update $th_g \coloneqq th_g \setminus \textit{kill-ring}$.
      iii. For every record $(j_0, j, \gamma_f) \in \textit{kill-ring}$, if there is no longer a record of the form $(j_0, j', \gamma'_f) \in th_g$, and $j_0 \notin th_f$, kill the thread beginning at j_0: emit $(\textit{Kill}, j_0)$.

Observe that $|th_f|$ is bound by the number of active threads of T_f and $|th_g|$ is bound by the number of active threads of T_g. Thus, $T_{split(f,g)}$ responds to each input signal in time $O(|f| + |g|) + t_f + t_g$, where t_f and t_g are the response times of T_f and T_g respectively.

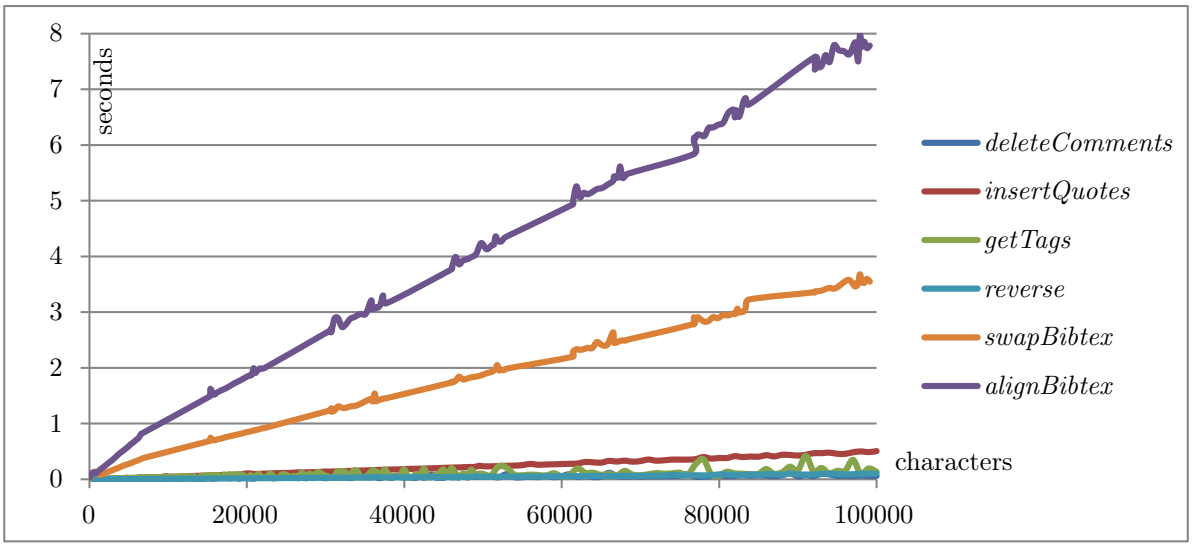


Figure 3.1.3: Experimental runtimes of the streaming evaluation algorithm.

3.1.5 Experimental evaluation

For the benchmarks *swapBibtex* and *alignBibtex*, our input files were constructed randomly out of a soup of available entries. For all other benchmarks, we considered the printable ASCII characters—`space` (decimal code 32) to ~ (decimal code 126)—and picked strings uniformly at random. Figure 3.1.3 shows the running time of the streaming evaluation algorithm (mean over 10 runs), and confirms that it depends linearly on the length of the input string. Note that the evaluation algorithm scales gracefully—$\approx 10,000$ characters per second—to reasonably large function expressions such as *alignBibtex*, with 3652 subexpressions in the AST.

We also implemented the functions *deleteComm*, *insertQuotes*, *getTags*, and *reverse* in sed and Perl.[2] The first three functions depend primarily on regular expression-based substitution and deletion mechanisms, and both tools were ≈ 6 times faster than DReX. The function *reverse* is difficult to implement in sed, and all implementations we have seen need quadratic time. On this benchmark, Perl is ≈ 2 times faster than DReX.

3.1.6 What breaks with unrestricted DReX expressions?

First, notice that the function composition operator is unamenable to the evaluator model we just described. We wish to process each character in bounded time, regardless of the length of the input string. Consider the case of $f \circ g$: when the evaluator T_f returns a result $(\mathit{Result}, i, \gamma)$, we have to pass the *entire* intermediate result string γ to T_g, and this is possibly as long as the input seen so far. Notice that this limitation should be unsurprising, because nested function compositions—such as the transformation $\sigma \mapsto \sigma\sigma$ composed with itself k times—can cause an exponential blowup in the length of the output string.

Next, consider a potential evaluator T for $\mathit{split}(f, g)$, in the absence of any consistency requirement. Thus, there might exist strings $\sigma = \sigma_1\sigma_2 \ldots \sigma_n$ which admit two splits $\sigma = \tau_1\tau_2 = \tau_1'\tau_2'$, such that all of $f(\tau_1)$, $f(\tau_1')$, $g(\tau_2)$, and $g(\tau_2')$ are defined. In this case, $\mathit{split}(f, g)$ is undefined for the entire string σ. We have to drop the requirement of input validity, because the nested evaluator T_g emits *two Result* signals after reading σ_n. We could conceivably modify T to emit an output signal when exactly one thread of T_g returns a result. Unfortunately, it turns out that this modification is insufficient, as the induction hypothesis now breaks—the evaluator T has to perform additional book-keeping to report results correctly. The consistency rules provide an easy way to avoid this non-trivial book-keeping.

[2]Cross-language benchmarks are extremely difficult to execute correctly, and often biased. The only claim being made in this paragraph is that DReX is not too much slower than existing tools.

3.2 The Complexity of Unrestricted DReX

In this section, we justify the imposition of the consistency restriction by showing that evaluating unrestricted DReX expressions is PSPACE-complete. The function composition combinator features prominently in the hardness proof; recall that the most restrictive of the consistency rules was the prohibition on its usage. We first describe the polynomial space evaluation algorithm for unrestricted DReX expressions, which depends on the following bound on the output length:

Proposition 3. *For each function expression f, which is possibly not even consistent, and input string $\sigma \in \Sigma^*$ such that $f(\sigma)$ is defined, $|f(\sigma)| \leq |f|^{d+1} |\sigma|$, where d is the number of composition operators appearing in f.*

The index of each character in the output is therefore only polynomially many bits long. We adopt an implicit representation of strings with the following operations (in contrast with the traditional explicit list-of-characters representation of strings):

1. check whether the string σ is defined,
2. compute the length of σ, and
3. given an index i, compute the i-th character of σ.

Then, by structural induction on the expression f, and given an implicit representation of the input string σ, we build an implicit representation of $f(\sigma)$ using only polynomial space. For example, the implicit representation of $(f \mathit{\ else\ } g)(\sigma)$ would function as follows:

1. to check whether the output is defined, simply determine whether either $f(\sigma)$ or $g(\sigma)$ is defined;
2. to compute the length of the output, if $f(\sigma)$ is defined, return the length of $f(\sigma)$, and otherwise, return the length of $g(\sigma)$; and
3. to compute the i-th character of $(f \mathit{\ else\ } g)(\sigma)$, if $\tau_f = f(\sigma)$ is defined, then return the i-th character of τ_f, and otherwise, return the i-th character of $\tau_g = g(\sigma)$.

Observe that since both nested implicit representations $f(\sigma)$ and $g(\sigma)$ consume only polynomial space, $(f \mathit{\ else\ } g)(\sigma)$ is itself evaluated in polynomial space. The most interesting case is $f \circ g$ where we simply connect the implicit representation of the output of $g(\sigma)$ to the input of the function f. The only non-trivial case is when $f = \mathit{iter}(g)$. To check whether f is defined on the input σ, we need to determine whether there is exactly one way to split σ such that g is defined on each split. Consider each position in the string σ as a vertex in a graph, with an edge between two vertices iff g is defined on the substring between them. Then each path from the initial to the final node of this graph corresponds to a viable split of σ, and thus f is defined on σ iff there is a unique path from the initial node to the final node in this implicitly represented graph of potentially exponential size. This problem can be solved in PSPACE.

Theorem 4. *The following problems are* PSPACE*-complete:*

1. *given an unrestricted DReX expression f, determining whether $f(\epsilon)$ is defined,*
2. *given an unrestricted DReX expression f and an input string $\sigma \in \Sigma^*$, determining whether $f(\sigma)$ is defined, and*
3. *given an unrestricted DReX expression f, an input string $\sigma \in \Sigma^*$, and a candidate output string $\tau \in \Gamma^*$, determining whether $f(\sigma) = \tau$.*

Proof sketch of PSPACE*-hardness.* All problems can be reduced in polynomial time to the first problem. We then reduce the problem of determining the validity of quantified boolean formulas (QBF) to the first problem. Given a QBF $\Phi = \forall x_1, \exists x_2, \ldots, \exists x_n, \varphi(x_1, \ldots, x_n)$ we construct an expression f_Φ such that $f_\Phi(\epsilon)$ is defined iff Φ is valid. The expression f_Φ is the composition of three programs f_{01}, f_{3CNF}, and f_Q where:

1. f_{01} takes as input ϵ and outputs all the strings in $\{0,1\}^n$ in lexicographic order and separated by a #; this program generates all the possible assignments of the boolean variables.

2. $f_{3\text{CNF}}$ takes as input the string of all the assignments produced by f_{01} and replaces each assignment in $a \in \{0,1\}^n$ with T if the assignment a satisfies the 3CNF formula φ and F otherwise.

3. f_Q takes as input the string over $\{T,F\}^*$ and checks whether such a sequence of satisfying assignments is valid for the quantified formula Φ. If it is valid it outputs ϵ and otherwise it is undefined.

□

3.3 Ongoing Work

Expressions containing the *restrict* combinator are currently, by definition, ill-typed. This is because the combinator was not considered in the original description of the streaming algorithm. We expect to to deem $\mathit{restrict}(f, R)$ as well-typed iff f is well-typed, and R is an unambiguous regular expression with $[\![R]\!] \subseteq \mathit{Dom}(f)$. Note that the natural evaluator for the *restrict* combinator breaks the induction hypothesis in subtle ways. While we believe that the algorithm is still correct, the proof remains incomplete.

Chapter 4

Theory of Regular Functions

As mentioned in the introduction, there are multiple equivalent definitions of regular functions. In this proposal, we will define regularity using the operational model of streaming string transducers [6, 8]. An SST is a finite state machine which makes a single left-to-right pass over the input string. It maintains a set of registers which are updated on each transition. Examples of register updates include $v := uv\gamma$ and $v := \gamma v$, where $\gamma \in \Gamma^*$ is a constant string. The important restrictions are that transitions and updates are test-free—we do not permit conditions such as "q goes to q' on input a, provided $|v| \geq 5$"—and that the update expressions satisfy the copyless (or single-use) requirement. In this chapter, we outline the result that the functions expressible using function combinators are exactly the class of regular functions. The full proof of expressive completeness can be found in [11].

4.1 Regular String Transformations

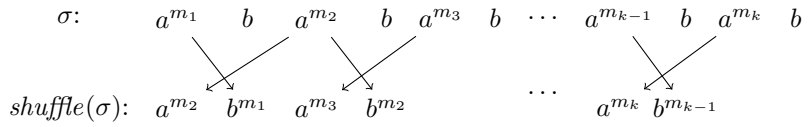
Definition 5. Let V be a finite set of registers. We call a function $f : V \to (V \cup \Gamma^*)^*$ *copyless* if the following two conditions hold:

1. For all registers $u, v \in V$, v occurs at most once in $f(u)$, and
2. for all registers $u, v, w \in V$, if $u \neq w$ and v occurs in $f(u)$, then v does not occur in $f(w)$.

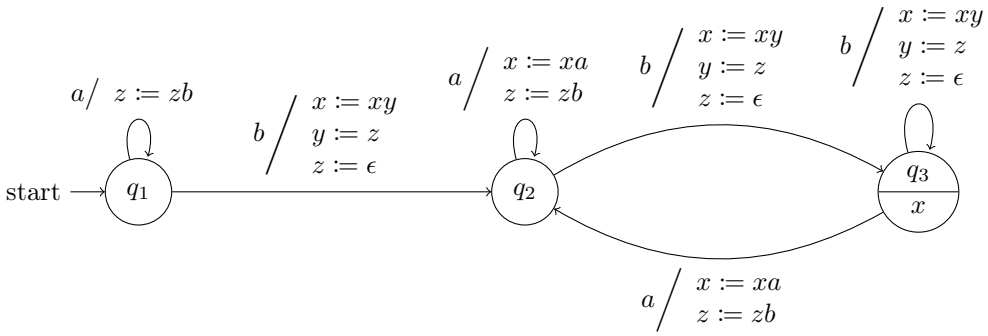
A string $e \in (V \cup \Gamma^*)^*$ is copyless if each register v occurs at most once in e.

Definition 6 (Streaming string transducers)**.** An SST is a tuple $M = (Q, \Sigma, \Gamma, V, \delta, \mu, q_{Start}, F, \nu)$, where

1. Q is a finite set of states,
2. Σ and Γ are finite input and output alphabets,
3. V is a finite set of registers,
4. $\delta : Q \times \Sigma \to Q$ is the state transition function,
5. $\mu : Q \times \Sigma \times V \to (V \cup \Gamma^*)^*$ is the register update function such that for all q and a, the partially applied function $\mu(q, a) : V \to V^*$ is a copyless function over V,
6. $q_{Start} \in Q$ is the initial state,
7. $F \subseteq Q$ is the set of final states, and
8. $\nu : F \to (V \cup \Gamma^*)^*$ is the output function, such that for all q, the output expression $\nu(q)$ is copyless.



(a) Definition of *shuffle*.



(b) SST $M_{shuffle}$ to compute *shuffle*. q_3 is the only accepting state. The annotation "x" in state q_3 specifies the output function. On each transition, registers whose updates are not specified are left unchanged.

Figure 4.1.1: Defining and implementing *shuffle* using a streaming string transducer.

The semantics of an SST M is specified using configurations. A *configuration* is a tuple $\gamma = (q, val)$ where $q \in Q$ is the current state and $val : V \to \Gamma^*$ is the register valuation. The initial configuration is $\gamma_{Start} = (q_{Start}, val_{Start})$, where $val_{Start}(v) = \epsilon$, for all v. For simplicity of notation, we first extend val to $V \cup \Gamma^* \to \Gamma^*$ by defining $val(\gamma) = \gamma$, for all $\gamma \in \Gamma^*$, and then further extend it to strings $val : (V \cup \Gamma^*)^* \to \Gamma^*$, by defining $val(v_1 v_2 \dots v_k) = val(v_1)val(v_2) \cdots val(v_k)$. If the machine is in the configuration $\gamma = (q, val)$, then on reading the symbol a, it transitions to the configuration $\gamma' = (q', val')$, and we write $\gamma \to^a \gamma'$, where $q' = \delta(q, a)$, and for all v, $val'(v) = val(\mu(q, a, v))$.

We now define the function $[\![M]\!] : \Sigma^* \to \Gamma_\perp^*$ *computed by* M. On input $\sigma \in \Sigma^*$, say $\gamma_{Start} \to^\sigma (q_f, val_f)$. If $q_f \in F$, then $[\![M]\!](\sigma) = val_f(\nu(q_f))$. Otherwise, $[\![M]\!](\sigma) = \perp$. A cost function is *regular* if it can be computed by an SST.

Example 7. Consider the function *shuffle*, which maps input strings of the form $a^{m_1}ba^{m_2}b \dots a^{m_k}b$, with $k \geq 2$, to the output $a^{m_2}b^{m_1}a^{m_3}b^{m_2} \dots a^{m_k}b^{m_{k-1}}$. See figure 4.1.1. This function is computed by the machine $M_{shuffle}$. It maintains three registers x, y and z, all initially holding the value ϵ. The register x holds the current output. On viewing each a in the input string, the machine commits to appending the symbol to its output. Depending on the suffix, this a may also be used to eventually produce a b in the output. This provisional value is stored in the register z. The register y holds the b-s produced by the previous run of a-s while the machine is reading the next patch of a-s. This function will be of interest to us later, in conjecture 23.

4.1.1 Closure under composition and regular look-ahead

Regular string transformations obey several appealing closure properties:

Proposition 8 (Closure under composition [19])**.** *If $f : \Sigma^* \to \Gamma_\perp^*$, and $g : \Gamma^* \to \Lambda_\perp^*$ are two regular string transformations, then so is compose(f, g).*

Another important property of regular functions is that they are closed under *regular look-ahead*: an SST can make transitions based not simply on the next symbol of the input, but on regular properties of the as-yet-unseen suffix. To formalize this, we introduce the notion of a *look-ahead labelling*. Let $\sigma = \sigma_1\sigma_2 \dots \sigma_n \in \Sigma^*$

be a string, and $A = (Q, \Sigma, \delta, q_{Start})$ be a DFA over Σ. Starting in state q_{Start}, and reading σ in reverse, say A visits the sequence of states $q_{Start} = q_1 \to^{\sigma_n} q_2 \to^{\sigma_{n-1}} q_3 \to^{\sigma_{n-2}} \cdots \to^{\sigma_1} q_{n+1}$. Then, the state of A at position i, q_i determines a regular property of the suffix $\sigma_{n-i+2}\sigma_{n-i+3}\ldots\sigma_n$. We term the string of states $q_{n+1}q_{n-1}\ldots q_1$ the labelling of σ by the *look-ahead automaton* A.

Proposition 9. *Let A be a look-ahead automaton over Σ, and let M be an SST with input alphabet Q. Then, there is an SST machine M' over Σ, such that for every $\sigma \in \Sigma^*$, $[\![M']\!](\sigma) = [\![M]\!](lab(\sigma))$ where $lab(\sigma)$ is the labelling of σ by A.*

4.2 Regular Functions are Closed Under Combinator Application

Theorem 10. *Every cost function expressible using the basic functions combined using the split, left-split, conditional choice, iterated sum, left-iterated sum, combine, chained sum, left-chained sum, restrict, and function composition operators is regular.*

Proof sketch. This can be proved by structural induction on the function expression:

1. To convert function expressions involving the split or iterated sum operators, we use regular look-ahead. For example, while evaluating $split(f, g)$, whenever M_f reaches an accepting state, we use regular look-ahead to check whether the suffix $\tau \in L_g$, where L_g is the domain of g. If so, the SST for $split(f, g)$ starts executing M_g.
2. For f *else* g and $combine(f, g)$, we simultaneously evaluate the machines M_f and M_g for f and g using the product construction.
3. For $restrict(f, R)$, we take the product of the SST M_f for f with the DFA A_R accepting R.
4. Closure under function composition of regular functions was stated in proposition 8.

□

4.3 Expressive Completeness of Function Combinators

In this section, we outline a proof of the fact that all regular functions $f : \Sigma^* \to \Gamma^*$ can be written as function expressions:

Theorem 11 ([11])**.** *For each finite input alphabet Σ and output alphabet Γ, every regular function $f : \Sigma^* \to \Gamma^*_\perp$ can be expressed by a consistent expression using the basic functions combined with the split sum (split, left-split), conditional choice, combine, and the chained sum (chain, left-chain) combinators.*

4.3.1 From DFAs to regular expressions: A review

In this subsection, we describe the classical algorithm [40] that transforms a DFA $A = (Q, \Sigma, \delta, q_{Start}, F)$ into an equivalent unambiguous regular expression. The algorithm to translate SSTs into function expressions relies on this. Hence this review.

Let $Q = \{q_1, q_2, \ldots, q_n\}$. For each pair of states $q, q' \in Q$, and for $i \in \mathbb{N}$, $0 \leq i \leq n$, $r^{(i)}(q, q')$ is the set of non-empty strings σ such that $\delta(q, \sigma) = q'$, and which only pass through the intermediate states $\{q_1, q_2, \ldots, q_i\}$. This can be inductively constructed as follows:

1. $r^{(0)}(q, q') = \{a \in \Sigma \mid q \to^a q'\}$.
2. $r^{(i+1)}(q, q') = r^{(i)}(q, q_{i+1})r^{(i)}(q_{i+1}, q_{i+1})^* r^{(i)}(q_{i+1}, q') \cup r^{(i)}(q, q')$.

Let $R_A = \bigcup_{q_f \in F} r^{(n)}(q_{Start}, q_f)$. Then, if $q_{Start} \notin F$, the language L accepted by A is given by the regular expression R_A , and otherwise L is given by the regular expression $\{\epsilon\} \cup R_A$. Note that all regular expressions obtained during the construction are unambiguous.

4.3.2 Proof outline

The SST-to-function expression translation algorithm proceeds in lockstep with the DFA-to-regular expression translator from section 4.3.1. The idea is to compute, in step i, for each pair of states q and q', a collection of function expressions which together summarize all strings $\sigma \in r^{(i)}(q, q')$.

In the SST $M_{shuffle}$ from figure 4.1.1b, consider the string aab processed from the state q_2. After processing aab, $M_{shuffle}$ is left in q_3. The final values of the registers x, y, and z, in terms of their initial values, are $xaay$, zbb, and ϵ respectively. Thus, the string aab processed from q_2 may be summarized by the pair $(q_3 = \delta(q_2, aab), \{x \mapsto xaay, y \mapsto zbb, z \mapsto \epsilon\})$, indicating the state of the machine after processing the string, and the final values of the registers. Similarly, the summaries for the strings $baab$ and $baabb$ processed from the initial state q_2 are $(q_3, \{x \mapsto xyaaz, y \mapsto bb, z \mapsto \epsilon\})$ and $(q_3, \{x \mapsto xyaazbb, y \mapsto \epsilon, z \mapsto \epsilon\})$ respectively.

Let us concentrate on the patterns in which register values are updated during computation. For the strings aab, $baab$, and $baabb$, these are, respectively, $\{x \mapsto \gamma_1 x \gamma_2 y \gamma_3, y \mapsto \gamma_4 z \gamma_5, z \mapsto \gamma_6\}$, $\{x \mapsto \gamma_1 x \gamma_2 y \gamma_3 z \gamma_4, y \mapsto \gamma_5, z \mapsto \gamma_6\}$, and $\{x \mapsto \gamma_1 x \gamma_2 y \gamma_3 z \gamma_4, y \mapsto \gamma_5, z \mapsto \gamma_6\}$, for some input-dependent string constants $\gamma_1, \ldots, \gamma_6 \in \Gamma^*$. We call these patterns the "shapes" of the input strings. Note that $baab$ and $baabb$ have identical shapes.

At each step i of the DFA-to-regular expression translation algorithm, for each pair of states q, q', and for each shape S, we compute an "expression vector" $\mathbf{R}_S^{(i)}(q, q')$ which summarizes all paths $\sigma \in r^{(i)}(q, q')$ with shape S: for each patch j in S, there is a function expression $\mathbf{R}_{S,j}^{(i)}(q, q') : \Sigma^* \to \Gamma^*$ such that the string constant γ_j appearing in the update summary of σ satisfies $\gamma_j = \mathbf{R}_{S,j}^{(i)}(q, q')(\sigma)$. The expression vector $\mathbf{R}_S^{(n)}(q_{Start}, q_f)$ then gives us the output of the SST on all strings σ that reach the final state q_f with shape S, and this can be used to construct a function expression f which is equivalent to the given SST.

4.4 Ongoing Work

Dana Fisman pointed out a bug in the original completeness proof in [11]. This bug can be fixed, and we are in the process of writing the complete corrected proof.

Chapter 5

Ongoing Work

In this chapter, we outline our current directions of work. We begin with a discussion of regular cost functions, and our attempts to describe a calculus of function combinators in section 5.1. We then turn back to the topic of string transformations, and describe our work on static analysis tools and on parallel evaluation algorithms in sections 5.2 and 5.3 respectively. We then present some conjectures about lower bounds on expressiveness in section 5.4, and conclude with an approximate timeline for meeting these goals in section 5.5.

5.1 Quantitative Extensions

The work we have done so far has focused on string-to-string transformations. The most important direction of current work, in collaboration with Dana Fisman, is in obtaining a similar representation for quantitative properties. There are three important questions we want to answer:

1. *What are regular cost functions?* In subsection 5.1.1, we describe a general notion of regularity, applicable to cost domains other than strings, as developed in [8].
2. Then, for each cost domain of interest:
   (a) *Is there an expressively complete calculus of function expressions for this class?* We describe our conjectures in subsection 5.1.2.
   (b) *Are there fast evaluation algorithms for the calculi thus identified?*

5.1.1 A theory of regular cost functions

We first generalize the theory of regular functions to target an arbitrary *cost domain*. The idea is to map input strings to expression terms over the cost domain and define regular functions as those which are realized by a *streaming string-to-term transducer*. Pick a cost domain $\mathbb{D}$ and an associated set G of operations (with arbitrary arity). Typical examples include:

1. the set $\mathbb{D} = \mathbb{Z}$ of integers under addition, i.e. $G = \{+\}$,
2. the set $\mathbb{D} = \mathbb{R} \cup \{\infty\}$ of extended real numbers with $G = \{+, \min\}$, i.e. the tropical semiring, and
3. the set $\mathbb{D} = \mathbb{R} \cup \{\infty\}$ with $G = \{+, \cdot \leq \cdot \,?\, \cdot : \cdot\}$, where $a \leq b \,?\, c : d$ is the 4-ary operator corresponding to "if $a \leq b$ then c, else d".

Note that string-to-string transformations are simply an instantiation with the monoid $(\mathbb{D} = \Gamma^*, \cdot, \epsilon)$ of strings under concatenation.

A streaming string-to-term transducer is similar to an SST, except that the registers now hold *expression terms*, such as "$\min(3, 4 + ?)$", instead of concrete values. The ? is a parameter hole into which other terms can

later be substituted. Given a set of registers V, let $E_u(V)$ and $E_g(V)$ be the set of ungrounded terms e_u (with exactly one occurrence of ?) and grounded terms e_g (with no unsubstituted occurrences of ?) respectively:

$$
\begin{aligned}
e_u ::= {} & ? \mid v \text{ (for } v \in V\text{)} \\
& \mid f(e_1, e_2, \ldots, e_k) \text{ (for each } k\text{-arity operator } f \in G, \text{ and exactly one ungrounded child } e_i\text{)} \\
& \mid e_u[e'_u] \\
e_g ::= {} & c \text{ (for } c \in \mathbb{D}\text{)} \\
& \mid f(e_1, e_2, \ldots, e_k) \text{ (for each } k\text{-arity operator } f \in G, \text{ and all children } e_i \text{ grounded)} \\
& \mid e_u[e_g]
\end{aligned}
$$

Note that the notation $e_1[e_2]$ corresponds to the operation of substituting the term e_2 into the parameter hole of e_1. The notion of a copyless update is identical to that in SSTs:

Definition 12. A function $f : V \to E(V)$ (grounded or ungrounded) over a set of registers V is *copyless* if

1. for each pair of registers $u, v \in V$, v occurs at most once in $f(u)$, and
2. for all registers $u, v, w \in V$, if $u \neq w$, and v occurs in $f(u)$, then v does not occur in $f(w)$.

Definition 13. A *streaming string-to-term transducer (SSTT)* over the input alphabet Σ is a tuple $M = (Q, \Sigma, V, \delta, \mu, q_{Start}, F, \nu)$, where

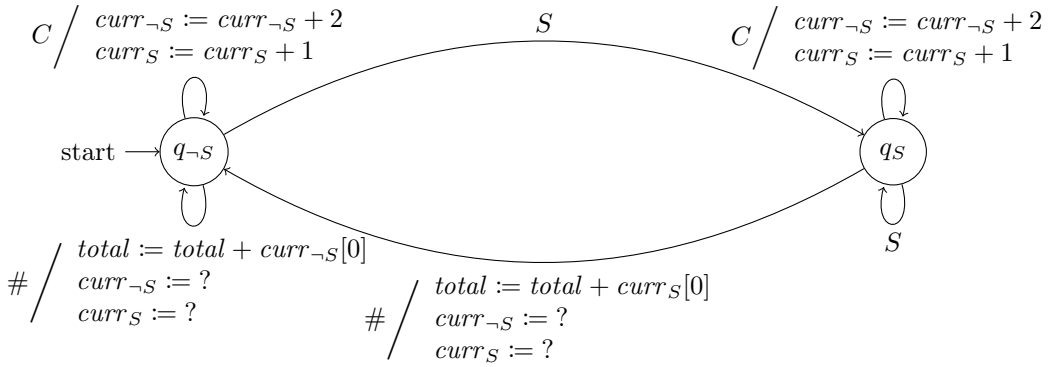
1. Q is a finite set of states,
2. V is a finite set of registers,
3. $\delta : Q \times \Sigma \to Q$ is the state transition function,
4. $\mu : Q \times \Sigma \times V \to E_u(V)$ is the register update function,
5. $q_{Start} \in Q$ is the initial state,
6. $F \subseteq Q$ is the set of accepting states, and
7. $\nu : F \to E_g(V)$ is the output function.

The semantics of SSTTs can be defined using combinators in the natural way, and we will not do it formally in this proposal.

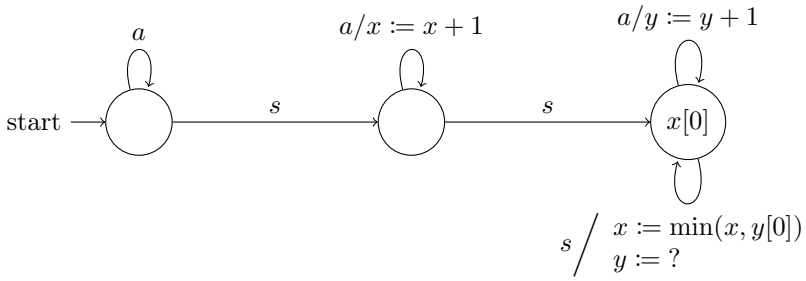
Example 14. We present several examples of SSTTs in figure 5.1.1. Consider the situation of a customer who frequents a coffee shop. Every cup of coffee he purchases costs \$2, but if he fills out a survey, then all cups of coffee purchased that month cost only \$1 (including cups already purchased). Here $\Sigma = \{C, S, \#\}$ denoting respectively the purchase of a cup of coffee, completion of the survey, and the passage of a calendar month. The function *coffee* maps the purchase history of the customer to the amount he owes the store, and is implemented by the machine M_{coffee}. The states q_S and $q_{\neg S}$ indicate the completion and non-completion of a survey respectively. The register *total* indicates the total amount owed to the store at the end of the previous month, and the registers $curr_S$ and $curr_{\neg S}$ indicate the tally for the current month.

The function *minWordLength* maps the input string to the distance between the closest pair of b-s. The machine $M_{minWordLength}$ implements this function.

Regular cost functions, i.e. those realized by streaming string-to-term transducers, obey many of the same closure properties as regular string transformations. They coincide with the class of MSO-definable string-to-term transformations, and are closed under operations such as input reversal and regular look-ahead.



(a) The machine M_{coffee} computes the amount owed to the store by a customer regularly purchasing coffee. Both states are accepting, and the output function in $q_{\neg S}$ is $(total + curr_{\neg S}[0])[0]$, and the output function in q_S is $(total + curr_S[0])[0]$.



(b) Machine $M_{minWordLength}$ maps the input string to the length of the shortest word. Say $\Sigma = \{a, s\}$, where the symbol s denotes a space, and a is a non-space symbol.

Figure 5.1.1: Examples of streaming string-to-term transducers.

5.1.2 Quantitative function expressions

Given numerical partial functions f and g, the following function combinators are straightforward extensions of the corresponding string-to-string function combinators.

1. The minimum, $\mathit{min}(f, g)$, is defined as $\mathit{min}(f, g)(\sigma) = \min(f(\sigma), g(\sigma))$, whenever both are defined, and is undefined otherwise.
2. The sum, $\mathit{sum}(f, g)$, is defined as $\mathit{sum}(f, g)(\sigma) = f(\sigma) + g(\sigma)$, if both are defined, and $\perp$ otherwise.
3. If there exists a unique split $\sigma = \sigma_1\sigma_2$, such that both $f(\sigma_1)$ and $g(\sigma_2)$ are defined, then we define $\mathit{split\text{-}min}(f, g)(\sigma) = \min(f(\sigma_1), g(\sigma_2))$ and $\mathit{split\text{-}plus}(f, g)(\sigma) = f(\sigma_1) + g(\sigma_2)$. Otherwise, both functions are undefined.
4. If there exists a unique split $\sigma = \sigma_1\sigma_2 \cdots \sigma_k$, such that $f(\sigma_i)$ is defined for each i, then we define $\mathit{iter\text{-}min}(f)(\sigma) = \min_i(f(\sigma_i))$, and $\mathit{iter\text{-}plus}(f)(\sigma) = \sum_i f(\sigma_i)$.

Motivated by idioms encountered during early attempts to recreate the proof approach of theorem 11, we define two new combinators $\mathit{min\text{-}suffix}(f)$ and $\mathit{min\text{-}prefix}(f)$, as follows:

$$\mathit{min\text{-}suffix}(f)(\sigma) = \min\{f(\tau) \mid \tau \in \Sigma^* \text{ is a suffix of } \sigma \text{ and } f(\tau) \text{ is defined}\}, \text{ and}$$
$$\mathit{min\text{-}prefix}(f)(\sigma) = \min\{f(\tau) \mid \tau \in \Sigma^* \text{ is a prefix of } \sigma \text{ and } f(\tau) \text{ is defined}\}.$$

Example 15. Consider again the case of the customer in a coffee shop, from example 14. Recall that each cup of coffee ordinarily costs \$2, unless he fills out a survey some time during that month, in which case, it costs only \$1. First, the function $\mathit{noSurvey} = \mathit{iter\text{-}plus}(C \mapsto 2)$ is the monthly cost to the customer if he does not fill out the survey. Similarly,

$$\mathit{survey} = \mathit{split\text{-}plus}(\mathit{iter\text{-}plus}(C \mapsto 1), S \mapsto 0, \mathit{iter\text{-}plus}(C \mapsto 1 \mathit{\ else\ } S \mapsto 0))$$

is the cost incurred if the survey is filled out. Thus, $\mathit{monthlyCost} = \mathit{noSurvey\ else\ survey}$ computes the monthly cost incurred by the customer. We can finally write $\mathit{coffee} = \mathit{iter\text{-}plus}(\mathit{split\text{-}plus}(\mathit{monthlyCost}, \# \mapsto 0))$.

Example 16. Recall the function *minWordLength*, from example 14. First, the function aCount_{a^*s}, which maps strings σ of the form a^*s to the number of a-s present, can be expressed as $\mathit{aCount}_{a^*s} = \mathit{split\text{-}plus}(\mathit{iter\text{-}plus}(a \mapsto 1), s \mapsto 0)$. This expression can be used in the expression for *minWordLength* as follows:

$$\mathit{minWordLength} = \mathit{split\text{-}min}(\mathit{iter\text{-}min}(\mathit{aCount}_{a^*s}), \mathit{iter\text{-}plus}(a \mapsto 1)).$$

Conjecture 17. *Regular cost functions $\Sigma^* \to \mathbb{R} \cup \{\infty\}_\perp$ are expressively equivalent to functions expressible using the basic functions (bottom, $\epsilon \mapsto d$, and $\varphi \mapsto d$, for $d \in \mathbb{R} \cup \{\infty\}$) combined with the min, sum, split-min, split-plus, iter-min, iter-plus, min-suffix, and min-prefix combinators.*

5.2 Static Analysis and Programmer Assistance Tools

We are currently working on developing practical programmer assistance tools for string-to-string DReX. Programmer assistance can range from run-time debugging aids and auto-completion routines, to compile-time formal verification tools. In this thesis, we plan to focus on the narrow task of statically analyzing function expressions.

5.2.1 Computing preimages

Formally, the preimage computation problem for DReX expressions is the following: Given a function expression $f : \Sigma^* \to \Gamma^*$ and a regular language $L_{post} \subseteq \Gamma^*$, determine the largest set $L_{pre} \subseteq \Sigma^*$ such that for each $\sigma \in L_{pre}$, $f(\sigma) \in L_{post}$. It is relatively straightforward to prove the following result:

Proposition 18. *Given a function expression $f : \Sigma^* \to \Gamma^*$ and a regular language $L_{post} \in \Sigma^*$ (represented by a DFA A_{post}), determining whether there exists a string $\sigma \in \Sigma^*$ such that $f(\sigma) \in L_{post}$ is* PSPACE*-complete.*

Proof sketch. Hardness follows from the fact that checking the emptiness of the intersection of a set $\{A_1, A_2, \ldots, A_k\}$ of DFAs is PSPACE-complete: each DFA A_i can be translated into an expression f_i, such that $f_i(\sigma) = \epsilon$ if σ is accepted by A_i, and $f_i(\sigma)$ is undefined otherwise. Now consider $f = \mathit{combine}(f_1, f_2, \ldots, f_k)$. For each string σ, $f(\sigma)$ is defined (and equal to ϵ) iff σ is simultaneously accepted by all the DFAs.

A PSPACE algorithm can be constructed by modifying the streaming evaluation algorithm. The evaluator is modified to maintain string summaries instead of concrete output strings. In this context, the "string summary" of each string γ is a function $\mathit{summ}_\gamma : Q_{post} \to Q_{post}$, which indicates, for each state $q \in Q_{post}$ of A_{post}, where the automaton would have terminated after reading γ. Such summaries can be represented with only polynomial space, and concatenated in polynomial time. Then, by non-deterministically guessing each subsequent symbol of the witness string, and running this modified evaluator, we can determine the existence of a witness string in non-deterministic polynomial space. □

Note that the precondition algorithm sketched above is only for the sake of obtaining a complexity bound, and is unlikely to be a very good algorithm in practice. We are also aware of another algorithm based on structural recursion on the function expression. For example, observe that

$$\mathit{preimage}(\mathit{restrict}(f, R), L_{post}) = \mathit{preimage}(f, L_{post}) \cap [\![R]\!] .$$

However, not only do the resulting regular expressions make free use of intersection, as above, they are also of exponential size. We are investigating whether this can still be turned into a practical algorithm by various heuristics, for example, by eagerly minimizing the intermediate DFAs.

5.2.2 Equivalence checking algorithms

Given function expressions $f, g : \Sigma^* \to \Gamma^*$, is it the case that for all $\sigma \in \Sigma^*$, $f(\sigma) = g(\sigma)$? It is known that this problem is decidable in polynomial space for SSTs [7]. Unfortunately, not much more is known. It seems extremely difficult to obtain any sort of hardness results, and we believe that this problem may even be solvable in polynomial time. For now, the strongest claim we can conclusively make is the following:

Proposition 19. *Given two SSTs M_1 and M_2 over a singleton input alphabet, we can determine in polynomial time whether they are equivalent.*

5.2.3 Potential applications

Over the last few years, strings have become an increasingly exploited attack vector, for example, in SQL injection and cross-site scripting. The solution is to use sanitizers, such as Google Caja [26], which check the "safety" of untrusted inputs. One application of formal verification tools is therefore in string sanitizers and other security-critical string manipulating programs.

Arjun Radhakrishna recently proposed a novel application of static analysis tools in a less security-critical setting. Consider a hypothetical paper author who has just obtained experimental results from a long running program. She uses Bash and other command line tools to manipulate the log files generated, and runs the command "`sed s/\_[0-9]*//`" (which converts filenames such as "`tmp_123`" to "`tmp`") to initialize a variable `BASE`. Then, in an attempt to delete all temporary files, she runs the command "`rm BASE*`". Unfortunately, the input to sed was the filename "`_123`", so that `BASE` was initialized to the empty string, and she ended up catastrophically deleting the entire contents of the directory. By modeling these Bash commands as DReX expressions, or by using DReX in place of traditional tools such as sed, can we improve the programmer

experience, for example by showing the helpful pop-up, "Are you sure there are no files with the name `\_[0-9]*`, such as `_0`?"

5.3 Parallel Evaluation Algorithms

We saw in figure 3.1.3 that the streaming evaluation algorithm gracefully scales to input strings several hundred kilobytes long. The procedure however, is still sequential, and unable to exploit the parallelism present in modern computers. Furthermore, several data streams, such as event logs from long-running processes, are several terabytes long, and too large to even fit on a single server. It is therefore essential to develop parallel evaluation algorithms, both at the level of a single server—multi-core systems and SIMD capabilities—and at the level of multiple servers each processing different portions of the input stream and "combining" their results.

5.3.1 NFA membership testing

We motivate parallel evaluation by describing a simple membership testing algorithm for NFAs. Consider an NFA $A = (Q, \Sigma, \delta, q_{Start}, F)$, where the state space $Q = \{q_1, q_2, \ldots, q_n\}$, and $q_{Start} = q_1$. Each string $\sigma \in \Sigma^*$ may then be summarized by an $n \times n$ integer matrix M_σ, such that for each pair of states q_i, q_j, if there is some execution of A with $q_i \to^\sigma q_j$, then $e_{\sigma ij} \geq 1$, and otherwise $e_{\sigma ij} = 0$. Note that σ is accepted by A iff for some i where $q_i \in F$, $e_{\sigma 1i} \geq 1$. Observe that the empty string is summarized by the identity matrix, and for each symbol $a \in \Sigma$, M_a can be constructed directly from the transition relation δ. The following result allows us to effectively summarize longer strings:

Proposition 20. *For each pair of strings $\sigma_1, \sigma_2 \in \Sigma^*$ summarized by the $n \times n$ matrices M_{σ_1} and M_{σ_2} respectively, the concatenation $\sigma_1\sigma_2$ is summarized by the matrix product $M_{\sigma_1}M_{\sigma_2}$.*

We can thus separately compute the summaries of disjoint substrings of the original input, and use proposition 20 to combine them and summarize the entire input string.

5.3.2 Conjectured algorithm

Given a DReX expression f, we first compute the "subexpression closure" E_f of f. For example, the subexpression closure of $split(a \mapsto 0, split(b \mapsto 1, c \mapsto 2))$ contains the expressions:

1. $a \mapsto 0, split(a \mapsto 0, b \mapsto 1), split(a \mapsto 0, split(b \mapsto 1, c \mapsto 2))$,
2. $b \mapsto 1, split(b \mapsto 1, c \mapsto 2)$, and
3. $c \mapsto 2$.

Informally speaking, E_f is the smallest set containing f such that for each pair of strings $\sigma_1, \sigma_2 \in \Sigma^*$, the value of every expression $g \in E_f$ on the input $\sigma_1\sigma_2$ is completely determined by the values of every expression $g' \in E_f$ on σ_1 and σ_2. Note that, as is evident in the example, E_f is possibly different from the set of all subexpressions of f. Thus, the key challenge is to prevent the number of expressions in E_f from becoming too large, and ideally, at most a small polynomial function of the size of f.

Several details in this proposed algorithm still need to be worked out. For example, in the case of left-additive operators, it is likely that the values of expressions are actually "strings with holes", where the results from the neighboring substrings eventually get substituted into these holes.

5.3.3 Potential applications

This project began out of a discussion at POPL 2015 with Madan Musuvathi, and is directly inspired by similar work on finite state state machines [39] and sequential transducers [44]. We intend to apply these evaluation routines to parallelize compression algorithms. LZ77 [47], which is based on a sliding window,

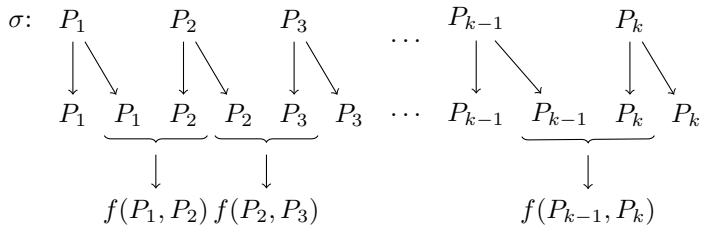


Figure 5.4.1: Expressing *shuffle*(σ) using function combinators. Each patch P_i is a string of the form a^*b.

seems particularly amenable to the technique we described in subsection 5.3.2. We also intend to study potential "table-maintaining" combinators, so that more complicated algorithms such as LZW [45] can also be easily expressed.

5.4 Expressiveness Lower Bounds

A final direction of future work is in tightening the completeness result of chapter 4. We have shown that all regular string transformations are expressible using the basic functions combined with the split sum, conditional choice, sum, chained sum, and the left-chained sum combinators. We believe that all these combinators are necessary for expressive completeness. In rough order of increasing complexity, we wish to establish the following results:

Conjecture 21 (Necessity of left-additive operators)**.** *The string reversal function $\sigma \mapsto \sigma^{rev}$ cannot be expressed using the basic functions combined using the split sum, conditional choice, sum, and the iterated sum combinators.*

Conjecture 22 (Necessity of sum)**.** *The regular string transformation copy(σ) = $\sigma\sigma$ is inexpressible using the basic functions combined using the split sum, conditional choice, iterated sum, and the left-iterated sum combinators.*

Conjecture 23 (Necessity of chained sum)**.** *The function shuffle (from example 7), cannot be expressed using the basic functions combined using the split sum, conditional choice, sum, iterated sum, and the left-iterated sum combinators.*

The function *shuffle* can be defined using the function combinators as follows. We first divide σ into patches P_i, each of the form a^*b. Similarly the output may also be divided into patches, P'_i. Each input patch P_i should be scanned twice, first to produce the a-s to produce P'_{i-1}, and then again to produce the b-s in P'_i. Let $R = a^*b$ be the language of these patches. See figure 5.4.1. It follows that *shuffle* = *chain*(f, R), where

$$f = \textit{left-split}(\textit{split}(\textit{iter}(a \mapsto b), b \mapsto \epsilon), \textit{split}(\textit{iter}(a \mapsto a), b \mapsto \epsilon)).$$

We hope to employ a similar strategy to prove all three conjectures. First, we annotate the output with "origin information", indicating which symbol of the input string resulted in each symbol of the output. We then establish invariants on the patterns that such origin information may possess. For example, in the conjecture 21, we would show that for each function expression, subsequent portions of the input can only influence subsequent portions of the output string, and that the string reversal function violates this property. Similarly, in the case of conjecture 22, the invariant would be that each symbol can only influence a single contiguous sequence of output symbols. Conjecture 23 would probably have the most complicated invariant: that it is always possible to split the input and output strings, each into two parts, such that each substring of the output is only influenced by one substring of the input. The formalization of input-output information flows as origin information is due to Bojanczyk [15], who showed that SSTs with origin information admit a normal form, in the context of developing an Angluin-style learning algorithm for regular string transformations.

5.5 Timeline

Note that our story is relatively complete for string-to-string transformations. The largest unfinished part of the thesis is developing combinators for quantitative properties. Over the next year, I plan to do the following, in rough order of importance:

1. Expressing quantitative properties: ≈ 3 months. We have concrete conjectures, and the project seems similar in depth and scope to a typical summer internship.
2. Developing static analysis tools: ≈ 3 months. Building a robust and practical preimage computation tool for function expressions should be straightforward, despite the PSPACE-completeness of the problem. Given that equivalence checking is a difficult problem even for lower bounds, the key would be to choose a useful set of applications, and tailor heuristics which work well for transformations arising in these applications.
3. Parallel evaluation algorithms: 2–3 months. Developing and proving the correctness of a parallel evaluation algorithm should take no more than a few weeks. Actually implementing this algorithm will take longer, and expressing compression / decompression algorithms as function expressions is probably a stretch goal.
4. Lower bounds regarding expressiveness: 1 month. Conjectures 21 and 22 seem straightforward to prove. Conjecture 23 would be harder to establish.
5. Thesis writing: 2–3 months.

I plan to work on the side on smaller bits of unfinished work and have therefore not included them in this timeline. This includes tasks such such as front-end improvements, fixes to the completeness proof, and building evaluators for the *restrict* combinator. Thus, I anticipate having a complete dissertation over the next 12–14 months.

5.5.1 Outline of the proposed thesis

The thesis itself will be similar in structure to this proposal, except that there will be two parallel settings of interest—string-to-string and string-to-cost transformations—and we will jump back and forth between them. Chapters 1 and 2 would be an introduction and function combinator tutorial respectively. Chapters 3 and 4 would describe the theory of regular functions and evaluation algorithms, in some order. The description of evaluation algorithms would be of interest to a broader community, and is technically less challenging than the proof of expressive completeness. I am therefore inclined to describe the evaluation algorithms in chapter 3 and regular function theory in chapter 4. Chapter 5 would describe the implementation, and our work on static analysis algorithms. While the benchmark expressions would of interest to a typical reader of the thesis, the actual implementation itself would be less interesting. Therefore, I plan to have an appendix listing the benchmark expressions, but only make the implementation available as an open-source project, as I have done in this proposal.

Chapter 6

Related Work

Finite state transducers Finite state transducers have been studied since the 1950s. Typical application areas include linguistics and natural language processing [34, 33].

It is well-known that two-way finite state transducers are strictly more expressive than one-way transducers, and that the post-image of a regular language with respect to a two-way transducer need not be regular [4]. While the equivalence problem for non-deterministic two-way finite state transducers is undecidable [27], the case when the machines are deterministic is decidable [30]. Closure under composition of deterministic two-way finite state transducers is established in [19].

Courcelle [20] introduced the idea of graph transformations defined in monadic second-order logic. Engelfriet and Hoogeboom [25] then showed that MSO-definable string transformations are the same as those which can be implemented by two-way finite state transducers. This mirrors the seminal results of Büchi [18], Elgot [24], and Trakhtenbrot [42], who showed that languages definable in the MSO-logic of one successor, S1S, are exactly those which are accepted by finite automata. This correspondence prompted [25] to label the class of functions definable by two-way finite state transducers as *regular* string transformations.

We are not aware of a prior characterization of regular string transformations by a set of combinators analogous to the characterization of regular languages by regular expressions.

Weighted automata String-to-number transformations have traditionally been modelled using weighted automata [23]. Given a semiring $(\mathbb{D}, +, \cdot, 0, 1)$, a weighted automaton is an NFA A over the input alphabet Σ, where each transition is labelled with a weight $d \in \mathbb{D}$. Given an input string $\sigma \in \Sigma^*$, and an accepting path π through A corresponding to σ, the weight of π is obtained by multiplying the weights of all transitions along the path. The weight of σ is the sum of the weights of all accepting paths corresponding to σ. Equivalence for weighted automata over the tropical semiring, $(\mathbb{Z} \cup \{\infty\}, \min, +, \infty, 0)$ is known to be undecidable, by a reduction from Hilbert's tenth problem [36], and by a reduction from the halting problem for two-counter machines [5]. Weighted automata over $(\mathbb{Z} \cup \{\infty\}, \min, +, \infty, 0)$ are strictly more expressive than regular cost functions, while *single-valued* weighted automata are equi-expressive as regular cost functions with $\mathbb{D} = \mathbb{N} \cup \{\infty\}$ and $G = \{+\}$.

DSLs for string transformations String transformations are traditionally specified using tools such as sed, AWK, Perl, and Perl Compatible Regular Expressions [31]. All these tools are Turing complete, making mechanical verification impossible. Furthermore, various popular extensions to the regular expression notation, such as "backreferences", typically have NP-complete matching problems, and this makes predicting performance hard. Finally, we do not know of a theoretical study of the class of string transformations which are expressible using these tools.

There has been a renewed interest in the formal verification of security-critical string transformations, resulting in tools such as Bek [32]. These languages are usually tightly coupled to the underlying transducer model, forcing the programmer to think in terms of low-level primitives such as a machine making a single

left-to-right pass over the input. Furthermore, these languages capture a strict subset of the class of regular string transformations, disallowing functions such as string reversal.

Boomerang [14] is a language to implement bidirectional string transformations, where updates to the output can be mapped back to appropriate updates to the input. There are two parts to the Boomerang system: basic string lenses, and dictionary lenses, which allow substrings to be associated with keys, and updates to be propogated meaningfully even when they involve changes in the output order. The basic string lenses of Boomerang are largely similar to the function combinators of string-to-string DReX, with the absence of *combine*, *chain*, and left-additive operators. Furthermore, their type system is identical to our consistency requirements. Note that our objectives are orthogonal to theirs: we focussed on achieving expressive completeness with respect to regular string transformations and on developing fast evaluation algorithms.

Constraint solving over strings A common theme in the formal verification of string manipulating programs is to assert state invariants, and use constraint solvers to find strings which violate these invariants. Starting with the seminal work of Makanin on solving string equations [38], this approach is exemplified by tools such as HAMPI [35], Norn [3], Z3-Str [46], and the string solver in CVC4 [37]. In contrast, our proposed approach to the static analysis of DReX—also adopted by tools such as Bek [32]—is to model sanitizers as transducers and prove properties using properties of the transducer model.

Data stream management systems There is a large body of work on data stream management systems, such as Aurora [2], Borealis [1], and STREAM [12, 13]. We are particularly interested in the query languages supported by these systems. A common feature is the use of a sliding window to maintain the recent history of each stream, and this disallows global transformations such as the reversal of records in a stream. On the other hand, most of these tools support elaborate techniques for composing queries, in contrast to DReX, where we explicitly forbid function composition from consistent function expressions. The STREAM project has compiled a repository of data stream queries [28]. We plan to use example queries from this repository as benchmarks.

Bibliography

[1] Daniel Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley Zdonik. The design of the Borealis stream processing engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 277–289, 2005.

[2] Daniel Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, August 2003.

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Phillip Rümmer, and Jari Stenman. String constraints for verification. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 150–166. Springer, 2014.

[4] Alfred Aho, John Hopcroft, and Jeffrey Ullman. A general theory of translation. *Mathematical Systems Theory*, 3(3):193–221, 1969.

[5] Shaull Almagor, Udi Boker, and Orna Kupferman. What's decidable about weighted automata? In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis*, volume 6996 of *Lecture Notes in Computer Science*, pages 482–491. Springer, 2011.

[6] Rajeev Alur and Pavol Černý. Expressiveness of streaming string transducers. In Kamal Lodaya and Meena Mahajan, editors, *Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 8 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1–12. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

[7] Rajeev Alur and Pavol Černý. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *Proceedings of the 38th Annual Symposium on Principles of Programming Languages*, POPL '11, pages 599–610. ACM, 2011.

[8] Rajeev Alur, Loris D'Antoni, Jyotirmoy Deshmukh, Mukund Raghothaman, and Yifei Yuan. Regular functions and cost register automata. In *28th Annual Symposium on Logic in Computer Science*, pages 13–22, 2013.

[9] Rajeev Alur, Loris D'Antoni, and Mukund Raghothaman. DReX: A declarative language for efficiently evaluating regular string transformations. In *Proceedings of the 42nd Annual Symposium on Principles of Programming Languages*, POPL '15, pages 125–137. ACM, 2015.

[10] Rajeev Alur, Loris D'Antoni, and Mukund Raghothaman. DReX demonstration website, 2015. `http://drexonline.com`.

[11] Rajeev Alur, Adam Freilich, and Mukund Raghothaman. Regular combinators for string transformations. In *Proceedings of the Joint Meeting of the 23rd Annual Conference on Computer Science Logic and the 29th Annual Symposium on Logic in Computer Science*, CSL-LICS '14, pages 9:1–9:10. ACM, 2014.

[12] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. STREAM: The Stanford data stream management system. Technical Report 2004-20, Stanford InfoLab, 2004.

[13] Arvind Arasu, Shivnath Babu, and Jennifer Widom. CQL: A language for continuous queries over streams and relations. In Georg Lausen and Dan Suciu, editors, *Database Programming Languages*, volume 2921 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2004.

[14] Aaron Bohannon, Nate Foster, Benjamin Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 407–419. ACM, 2008.

[15] Mikołaj Bojańczyk. Transducers with origin information. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming*, volume 8573 of *Lecture Notes in Computer Science*, pages 26–37. Springer, 2014.

[16] Ronald Book, Shimon Even, Sheila Greibach, and Gene Ott. Ambiguity in graphs and expressions. *IEEE Transactions on Computers*, 20(2):149–153, February 1971.

[17] Anne Brüggemann-Klein. Regular expressions into finite automata. In *LATIN '92*, volume 583 of *Lecture Notes in Computer Science*, pages 87–98. Springer, 1992.

[18] Richard Büchi. Weak second-order arithmetic and finite automata. *Mathematical Logic Quarterly*, 6(1-6):66–92, 1960.

[19] Michal Chytil and Vojtěch Jákl. Serial composition of 2-way finite-state transducers and simple programs on strings. In Arto Salomaa and Magnus Steinby, editors, *Automata, Languages, and Programming*, volume 52 of *Lecture Notes in Computer Science*, pages 135–147. Springer, 1977.

[20] Bruno Courcelle. Monadic second-order definable graph transductions. In Jean-Claude Raoult, editor, *Proceedings of the 17th Colloquium on Trees in Algebra and Programming*, volume 581 of *Lecture Notes in Computer Science*, pages 124–144. Springer, 1992.

[21] Loris D'Antoni and Rajeev Alur. Symbolic visibly pushdown automata. In *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 209–225. Springer, 2014.

[22] Loris D'Antoni and Margus Veanes. Equivalence of extended symbolic finite transducers. In *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 624–639. Springer, 2013.

[23] Manfred Droste, Werner Kuich, and Heiko Vogler. *Handbook of Weighted Automata*. Springer, 1st edition, 2009.

[24] Calvin Elgot. Decision problems of finite automata design and related arithmetics. *Transactions of the American Mathematical Society*, 98(1):21–51, 1961.

[25] Joost Engelfriet and Hendrik Jan Hoogeboom. MSO definable string transductions and two-way finite-state transducers. *ACM Transactions on Computational Logic*, 2(2):216–254, April 2001.

[26] Google. The Caja compiler. `https://developers.google.com/caja/` (accessed on: April 1, 2015).

[27] Thomas Griffiths. The unsolvability of the equivalence problem for λ-free nondeterministic generalized machines. *Journal of the ACM*, 15(3):409–413, July 1968.

[28] The Stanford STREAM Group. Stream query repository, 2002. `http://infolab.stanford.edu/stream/sqr/`.

[29] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual Symposium on Principles of Programming Languages*, POPL '11, pages 317–330. ACM, 2011.

[30] Eitan Gurari. The equivalence problem for deterministic two-way sequential transducers is decidable. In *21st Annual Symposium on Foundations of Computer Science*, pages 83–85, 1980.

[31] Philip Hazel. PCRE: Perl compatible regular expressions, 2015. `http://www.pcre.org/`.

[32] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and precise sanitizer analysis with BEK. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11. USENIX Association, 2011.

[33] Mans Hulden. Foma: A finite-state compiler and library. In *Proceedings of the Demonstrations Session at EACL 2009*, pages 29–32. Association for Computational Linguistics, 2009.

[34] Ronald Kaplan and Martin Kay. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378, September 1994.

[35] Adam Kiezun, Vijay Ganesh, Philip Guo, Pieter Hooimeijer, and Michael Ernst. HAMPI: A solver for string constraints. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, ISSTA '09, pages 105–116. ACM, 2009.

[36] Daniel Krob. The equality problem for rational series with multiplicities in the tropical semiring is undecidable. In Werner Kuich, editor, *Automata, Languages and Programming*, volume 623 of *Lecture Notes in Computer Science*, pages 101–112. Springer, 1992.

[37] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 646–662. Springer, 2014.

[38] Gennady Makanin. The problem of solvability of equations in a free semigroup. *Mathematics of the USSR-Sbornik*, 32(2):129–198, 1977.

[39] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. Data-parallel finite-state machines. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 529–542. ACM, 2014.

[40] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3rd edition, 2012.

[41] Richard Stearns and Harry Hunt. On the equivalence and containment problems for unambiguous regular expressions, grammars, and automata. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, pages 74–81. IEEE Computer Society, 1981.

[42] Boris Trakhtenbrot. Finite automata and the logic of one-place predicates. *Siberian Mathematical Journal*, 3:101–131, 1962.

[43] Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj Bjorner. Symbolic finite state transducers: Algorithms and applications. In *Proceedings of the 39th Annual Symposium on Principles of Programming Languages*, pages 137–150. ACM, 2012.

[44] Margus Veanes, Todd Mytkowicz, David Molnar, and Benjamin Livshits. Data-parallel string-manipulating programs. In *Proceedings of the 42nd Annual Symposium on Principles of Programming Languages*, POPL '15, pages 139–152. ACM, 2015.

[45] Terry Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, June 1984.

[46] Yunhi Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A Z3-based string solver for web application analysis. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 114–124. ACM, 2013.

[47] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

Appendix A

Benchmark Expressions

This appendix describes the function expressions that we used in our experiments in figure 3.1.3. We define the following analogues of the Kleene-plus operator:

$$\begin{aligned} \textit{iter-plus}(f) &= \textit{split}(f, \textit{iter}(f)), \text{ and} \\ \textit{left-iter-plus}(f) &= \textit{left-split}(f, \textit{left-iter}(f)). \end{aligned}$$

A.1 Delete Single-Line Comments

The function *deleteComments* deletes all one-line comments from a file (i.e. the lines starting with `//`). First, the expression *delCommLine* deletes strings of the form `//`σ`\n`, where σ does not contain any occurrence of the newline character `\n`:

$$\begin{aligned} \textit{delComm} &= \textit{split}(\texttt{`/'} \mapsto \epsilon, \texttt{`/'} \mapsto \epsilon, \textit{iter}(x \neq \texttt{`\textbackslash n'} \mapsto \epsilon)), \\ \textit{delCommLine} &= \textit{split}(\textit{delComm}, \texttt{`\textbackslash n'} \mapsto \epsilon). \end{aligned}$$

Next, the expression *copyLine* echoes strings of the form σ`\n`, where the string σ is either empty or does not start with the character `/`:

$$\begin{aligned} \textit{copyNonEndingLine} &= \textit{split}(x \neq \texttt{`/'} \mapsto x, \textit{iter}(x \neq \texttt{`\textbackslash n'} \mapsto x)), \\ \textit{copyLine} &= \texttt{`\textbackslash n'} \mapsto \texttt{`\textbackslash n'} \textit{ else } \textit{split}(\textit{copyNonEndingLine}, \texttt{`\textbackslash n'} \mapsto \texttt{`\textbackslash n'}). \end{aligned}$$

The function *processLine* reads a line and deletes it if it is a comment line and copies it otherwise. The last line might not end with a `\n` and the expression *processLastLine* deals with this exception. Finally the function *deleteComments* repeats *processLine* and at the end processes the last line, therefore deleting all the one-line comments in the input.

$$\begin{aligned} \textit{processLine} &= \textit{delCommLine else copyLine}, \\ \textit{processLastLine} &= \textit{processLine else delComm else copyNonEndingLine}, \\ \textit{deleteComments} &= \textit{split}(\textit{iter}(\textit{processLine}), \textit{processLastLine}). \end{aligned} \tag{A.1.1}$$

A.2 Insert Quotes Around Words

The expression *insertQuotes* inserts quotes (`"`) around every alphabetic substring appearing in the input. First, the expression *addQtsSkip*, given a string $\sigma_1\sigma_2$ where σ_1 is alphabetic and σ_2 does not contain any

letter, outputs the string "σ_1"σ_2.

$$\begin{aligned} \mathit{copyAlphaStr} &= \mathit{iter\text{-}plus}(x \in \texttt{[a-zA-Z]} \mapsto x), \\ \mathit{copyNonAlphaStr} &= \mathit{iter\text{-}plus}(x \notin \texttt{[a-zA-Z]} \mapsto x), \\ \mathit{addQts} &= \mathit{split}(\epsilon \mapsto \texttt{`"'}, \mathit{copyAlphaStr}, \epsilon \mapsto \texttt{`"'}), \\ \mathit{addQtsSkip} &= \mathit{split}(\mathit{addQts}, \mathit{copyNonAlphaStr}). \end{aligned}$$

Finally, the expression *insertQuotesStart* handles strings which start with a non-alphabetic character sequence, and the expression *insertQuotesEnd* handles strings ending with an alphabetic character sequence.

$$\begin{aligned} \mathit{insertQuotesStart} &= \mathit{iter}(x \notin \texttt{[a-zA-Z]} \mapsto x), \\ \mathit{insertQuotesEnd} &= \mathit{addQts}\ \mathit{else}\ \epsilon \mapsto \epsilon, \\ \mathit{insertQuotes} &= \mathit{split}(\mathit{insertQuotesStart}, \mathit{iter}(\mathit{addQtsSkip}), \mathit{insertQuotesEnd}). \end{aligned} \tag{A.2.1}$$

A.3 Extracting Tags From an XML File

The expression *getTags* extracts and concatenates all the substrings of the form <σ> where σ does not contain any occurrences of < or > (this is a generalization of a transducer shown in [43]). For simplicity we assume that the string does not contain occurrences of the substring <>. The expression *copyMatch* echoes strings of the form <σ> where σ does not contain the characters < or >.

$$\mathit{copyMatch} = \mathit{split}(\texttt{`<'} \mapsto \texttt{`<'}, \mathit{iter\text{-}plus}(x \notin \texttt{[<>]} \mapsto x), \texttt{`>'} \mapsto \texttt{`>'}).$$

The expression *delNotMatch* deletes any string which does not contain a substring of the form <σ> where σ does not contain the characters < or >.

$$\begin{aligned} \mathit{delNotOpn} &= \mathit{iter}(x \neq \texttt{`<'} \mapsto \epsilon), \\ \mathit{delNotMatch} &= \mathit{delNotOpn}\ \mathit{else}\ \mathit{split}(\mathit{delNotOpn}, \texttt{`<'} \mapsto \epsilon, \mathit{iter}(x \neq \texttt{`>'} \mapsto \epsilon)). \end{aligned}$$

Finally, we can write:

$$\mathit{getTags} = \mathit{split}(\mathit{iter}(\mathit{split}(\mathit{delNotMatch}, \mathit{copyMatch})), \mathit{delNotMatch}). \tag{A.3.1}$$

A.4 Reversing a List

Given a list of the form "$\sigma_1;\sigma_2;\cdots;\sigma_n$", we want to output the reverse, "$\sigma_n;\sigma_{n-1};\cdots;\sigma_1$". The expression *copyStretch* copies a string of the form "$\sigma;$" where σ does not contain a ';'. The program *reverse* implements the final transformation by left iterating *copyStretch*.

$$\begin{aligned} \mathit{copyStretch} &= \mathit{split}(\mathit{iter}(x \neq \texttt{`;'} \mapsto x), \texttt{`;'} \mapsto \texttt{`;'}), \\ \mathit{reverse} &= \mathit{left\text{-}iter}(\mathit{copyStretch}). \end{aligned} \tag{A.4.1}$$

A.5 Reformatting BibTEX files

In this section, we define two functions which operate over BibTEX files:

1. the first function, *swapBibtex*, moves the title attribute of each entry to the top of the list of attributes (figure 1.0.1), and
2. the second function, *alignBibtex*, rearranges a *misaligned* file by moving the title attribute of each entry to the previous entry (figure 1.0.2).

The idea is to encode typical transformations paper authors may perform on their BibTEX libraries, and thus demonstrate that DReX is a practical tool for many document transformation tasks.

A.5.1 Basic definitions

When we refer to the "header" of a BibTEX entry, we are referring to the entry type and entry name: for example, `@book{Gal1638` is the header of the entries in figure 1.0.1. In this description we assume that every attribute is followed by a comma.[1] The function *copyHeader* is defined over entry headers, and simply echoes them:

$$\begin{aligned} \mathit{copyHeader} &= \mathit{split}(\texttt{`@'} \mapsto \texttt{`@'}, \mathit{copyAlphaStr}, \texttt{`\{'} \mapsto \texttt{`\{'}, \mathit{copyAlphaNum}, \mathit{copySpaces}), \text{ where} \\ \mathit{copyAlphaStr} &= \mathit{iter\text{-}plus}(x \in \texttt{[a-zA-Z]} \mapsto x), \\ \mathit{copyAlphaNum} &= \mathit{iter\text{-}plus}(x \in \texttt{[a-zA-Z0-9]} \mapsto x), \text{ and} \\ \mathit{copySpaces} &= \mathit{iter}(x \in \texttt{[\textbackslash n\textbackslash r\textbackslash b\textbackslash t]} \mapsto x). \end{aligned}$$

Given a string σ, the expression

$$\mathit{copy}(\sigma) = \mathit{split}(\sigma_1 \mapsto \sigma_1, \sigma_2 \mapsto \sigma_2, \ldots, \sigma_{|\sigma|} \mapsto \sigma_{|\sigma|})$$

echoes the input string if it is equal to σ and is undefined otherwise. Furthermore, given a set of strings $\{\sigma^{(1)}, \sigma^{(2)}, \ldots, \sigma^{(n)}\}$, the expression

$$\mathit{copy}(\{\sigma^{(1)}, \sigma^{(2)}, \ldots, \sigma^{(n)}\}) = \mathit{copy}(\sigma^{(1)}) \textit{ else } \mathit{copy}(\sigma^{(2)}) \textit{ else } \cdots \textit{ else } \mathit{copy}(\sigma^{(n)})$$

echoes the input string if it is equal to any of $\{\sigma^{(1)}, \sigma^{(2)}, \ldots, \sigma^{(n)}\}$, and is undefined otherwise. For example, the functions

$$\begin{aligned} \mathit{copyTitle} &= \mathit{copy}(\texttt{"title"}), \text{ and} \\ \mathit{copyNonTitle} &= \mathit{copy}(\{\texttt{"author"}, \texttt{"year"}, \texttt{"place"}, \ldots\}) \end{aligned}$$

echo the attribute names `title`, and any of `author`, `year`, `place`, etc. respectively. The expression *copyAttrValue* echoes the value of an attribute (for example, `= {Elzevir}`):

$$\begin{aligned} \mathit{copyAttrValue} = \mathit{split}(&\mathit{copySpaces}, \texttt{`='} \mapsto \texttt{`='}, \mathit{copySpaces}, \\ &\texttt{`\{'} \mapsto \texttt{`\{'}, \mathit{copyNonBr}, \texttt{`\}'} \mapsto \texttt{`\}'}, \texttt{`,'} \mapsto \texttt{`,'}, \mathit{copySpaces}), \text{ where} \\ \mathit{copyNonBr} = \mathit{iter}(&x \notin \texttt{[}\{\texttt{]}\texttt{[}\} \mapsto x). \end{aligned}$$

Put together, we can write

$$\begin{aligned} \mathit{copyTitleAttr} &= \mathit{split}(\mathit{copyTitle}, \mathit{copyAttrValue}), \text{ and} \\ \mathit{copyNonTitleAttr} &= \mathit{split}(\mathit{copyNonTitle}, \mathit{copyAttrValue}) \end{aligned}$$

for the expressions which copy all of the attribute, both name and value.

Similarly, we can write the expressions *delTitleAttr* and *delNonTitleAttr* which delete the title and non-title attributes respectively. The functions *delAttributes* and *delHeader* delete an attribute-value pair, and the header of an entry respectively.

Given a list of attributes, the function *titleOnly* copies the title and deletes all non-title attributes, and the function *allButTitle* deletes the title and echoes all other attributes:

$$\begin{aligned} \mathit{titleOnly} &= \mathit{iter}(\mathit{copyTitleAttr} \textit{ else } \mathit{delNonTitleAttr}), \\ \mathit{allButTitle} &= \mathit{iter}(\mathit{delTitleAttr} \textit{ else } \mathit{copyNonTitleAttr}). \end{aligned}$$

[1] The actual functions used for the experiments are able to deal with missing commas.

A.5.2 Reordering attributes

The expression *titleFirst* copies the title first and then all the other attributes, therefore obtaining the desired attribute reordering.

$$titleFirst = combine(titleOnly, allButTitle).$$

The expression *swapEntry* performs the operation of figure 1.0.1 for a single BibTEX entry:

$$swapEntry = split(copyHeader, titleFirst, \texttt{`}\texttt{}\texttt{'} \mapsto \texttt{`}\texttt{}\texttt{'}, copySpaces),$$

so that we can write:

$$swapBibtex = iter(swapEntry). \tag{A.5.1}$$

A.5.3 Correcting misaligned files

We introduced the function *alignBibtex* in chapter 1, and described some parts of it in section 2.4. We now complete its description. The functions *headerOnlyFromEntry*, *titleOnlyFromEntry* and *allButTitleFromEntry* map a single entry to just its header, title, and the body respectively:

$$\begin{aligned} headerOnlyFromEntry &= split(copyHeader, delAttributes, \texttt{`}\texttt{}\texttt{'} \mapsto \epsilon, delSpaces), \\ titleOnlyFromEntry &= split(delHeader, titleOnly, \texttt{`}\texttt{}\texttt{'} \mapsto \epsilon, delSpaces), \\ allButTitleFromEntry &= split(delHeader, allButTitle, \texttt{`}\texttt{}\texttt{'} \mapsto \texttt{`}\texttt{}\texttt{'}, copySpaces). \end{aligned}$$

On the other had, the expression *deleteEntry* deletes the entire BibTEX entry:

$$deleteEntry = split(delHeader, delAttributes, \texttt{`}\texttt{}\texttt{'} \mapsto \epsilon, delSpaces).$$

Finally, we can write:

$$\begin{aligned} alignBibtex &= chain(makeEntry, R_{Entry}), \text{ where} \\ makeEntry &= combine(split(headerOnlyFromEntry, titleOnlyFromEntry), \\ &\qquad split(allButTitleFromEntry, deleteEntry)), \end{aligned} \tag{A.5.2}$$

and R_{Entry} is the regular expression matching a single entry. R_{Entry} can be automatically inferred from the domain of any of the functions *headerOnlyFromEntry*, *titleOnlyFromEntry*, *allButTitleFromEntry*, or *deleteEntry*.