

DReX: A Declarative Language for Efficiently Evaluating Regular String Transformations^{*}

Rajeev Alur Loris D’Antoni Mukund Raghothaman

University of Pennsylvania
{alur, lorisdan, rmukund}@seas.upenn.edu

Abstract

We present DReX, a declarative language that can express all regular string-to-string transformations, and can still be efficiently evaluated. The class of regular string transformations has a robust theoretical foundation including multiple characterizations, closure properties, and decidable analysis questions, and admits a number of string operations such as insertion, deletion, substring swap, and reversal. Recent research has led to a characterization of regular string transformations using a primitive set of function combinators analogous to the definition of regular languages using regular expressions. While these combinators form the basis for the language DReX proposed in this paper, our main technical focus is on the complexity of evaluating the output of a DReX program on a given input string. It turns out that the natural evaluation algorithm involves dynamic programming, leading to complexity that is cubic in the length of the input string. Our main contribution is identifying a consistency restriction on the use of combinators in DReX programs, and a single-pass evaluation algorithm for consistent programs with time complexity that is linear in the length of the input string and polynomial in the size of the program. We show that the consistency restriction does not limit the expressiveness, and whether a DReX program is consistent can be checked efficiently. We report on a prototype implementation, and evaluate it using a representative set of text processing tasks.

Categories and Subject Descriptors D.3.2 [Language Classifications]: Specialized application languages; F.1.1 [Theory of Computation]: Models of Computation, Automata

Keywords DReX, string transformations, declarative languages

1. Introduction

Programs that transform plain text are ubiquitous and used for many different tasks, from reformatting documents to translating data between different formats. String specific utilities such as sed, AWK, and Perl have been used to query and reformat text files for many years. Since these tools are Turing complete they can express very complex transformations, however this comes at the cost of not being amenable to algorithmic analysis. To address this issue, restricted languages have been proposed in the context of verification of string sanitizers [26] and string coders [13], and for the analysis and optimization of list-manipulating programs [15]. These languages build on variants of finite-state transducers, which are automata-based representations of programs mapping strings to strings, and

each of these languages supports different algorithmic analyses that are enabled by the properties of the underlying transducer model. Due to the focus on analyzability, expressiveness is a limiting factor in all such languages and many programs, in particular those that reorder input chunks, cannot be represented. Moreover, these languages are not declarative and their semantics are tightly coupled to the transducer model, forcing the programmer to reason in terms of finite state machines and process the input left-to-right.

In the theory of string-to-string transformations the class of regular string transformations is a robust class that strikes a balance between decidability and expressiveness. In particular this class captures transformations that involve reordering of input chunks, it is closed under composition [9], it has decidable equivalence [21], and has several equivalent characterizations, such as one-way transducers with a finite set of write-only registers [1], two-way transducers [16] and monadic second-order definable graph transformations [10]. Recently Alur et al [3] proposed a set of combinators that captures the class of regular string transformations. In [3] the focus is on expressiveness and the paper does not provide an efficient procedure to evaluate programs written with these combinators. Efficient evaluation of such programs is the main focus of this paper.

Starting with the combinators presented in [3], we develop DReX, an expressive declarative language to describe string transformations. The base combinator of DReX, $\varphi \mapsto d$, maps any character a that satisfies the predicate φ to the string $d(a)$. This combinator symbolically extends the one proposed in [3] with predicates and can therefore succinctly model strings over large (and potentially infinite) alphabets, such as Unicode. The other combinators supported by DReX are: (a) `split(f, g)` that unambiguously splits the input string into two parts and outputs the concatenation of the results obtained using f on the first part and g on the second part; (b) `iterate(f)` that unambiguously splits the input string into multiple parts and outputs the concatenation of evaluating f on each of such parts; (c) `combine(f, g)` that applies both f and g to the input string and concatenates the obtained results; (d) the conditional `ifelse(f, g)` that first tries to apply f to the input and if f cannot be applied it applies g ; (e) `chain(f, R)` that unambiguously splits the string into multiple parts $\sigma = \sigma_1 \dots \sigma_n$ each belonging to the language described by the regular expression R , applies f to every two pair of adjacent chunks $\sigma_i \sigma_{i+1}$, and finally concatenates these results. In order to model operations such as reversing a string, the operators `split`, `iteration` and `chained sum` also have a *left-additive* version in which the outputs computed on each split of the string are concatenated in reverse order.

A straightforward algorithm to evaluate DReX programs involves “operationalizing” the semantics, i.e. use dynamic programming and evaluate each sub-program on each substring of the input. Unfortunately, this algorithm takes time cubic in the length of the input string, and does not scale to strings longer than approximately a thousand characters. Because of the analogy between DReX oper-

^{*} This is the full version of the paper to be presented at POPL 2015, and includes proofs omitted from the short version of the paper. A prototype implementation of DReX may be downloaded from <http://www.cis.upenn.edu/~rmukund/drex/>. This research was partially supported by NSF Expeditions in Computing grant CCF 1138996.

ators (split sum, conditionals, iteration, etc.) and regular expressions (concatenation, union, Kleene-*, etc.) one approach is to construct an automaton model for evaluating DReX programs similarly to the approach taken to evaluate regular expressions. This is not simple because of various reasons, such as: (a) the conditional operator, $f \text{ else } g$, applies the transformation g to the input only if the input string is not accepted by f . To check whether a string is in the complement of the domain of f one needs to determinize the domain automaton and this is an exponential time operation; and (b) the operator $\text{combine}(f, g)$ is only defined on the intersection of the domains of f and g . Repeating automata intersections multiple times also causes an exponential blow-up.

The main technical challenge is to identify a fragment of DReX which does not sacrifice expressiveness, and still permits “fast” evaluation algorithms. We call this subset of DReX the *consistent* fragment. Intuitively, we require each operator to admit unambiguous parsing, and limit the operators’ ability to express the complement and intersection of languages. For example, $\text{split}(f, g)$ is consistent iff the domains of f and of g are unambiguously concatenable, i.e. there is no string with multiple viable splits. In the case of a conditional $f \text{ else } g$, the domains of f and g are required to be disjoint, making the complementation of the domain of f unnecessary. Similarly, for the operator $\text{combine}(f, g)$, we require the domains of f and g to be identical, so that the domain of the entire program is equal to the domain of its sub-expressions and no language intersection is required. For the $\text{chain}(f, R)$ operator to be consistent, the language R^* is required to be unambiguous, and f is required to be a split operator for which both the left and right hand sides are exactly defined on the language described by R . We show that consistency of a DReX program can be efficiently determined.

We present an algorithm that evaluates a consistent DReX program f on an input string σ , in time polynomial in the size of f and linear in the length of σ . Intuitively, we construct a machine for each sub-program which reads the input in a single left-to-right pass. Each machine keeps track of potential parse trees of σ as multiple *threads*, and updates the threads on reading each input symbol. The goal of the algorithm is to have a number of threads that is linear in the size of the program but does not depend on the length of the input string. This bound is achieved using the consistency requirements to eagerly kill threads whenever they become inactive. For example, the machine for $\text{split}(f, g)$ outputs a result as soon as it discovers a single viable split of the input string σ , since the consistency rules guarantee the absence of any other split. If the program were not consistent the machine would need to delay the output causing the number of alive threads to depend on the length of the string. Similarly, the machine for $f \text{ else } g$ can output the results of g whenever this is defined, because of the requirement that f and g have disjoint domains and cannot simultaneously emit results. We also present a dynamic programming algorithm that can handle all DReX programs, but is limited by its cubic time complexity in the length of the input string.

We implemented our evaluation and consistency-checking algorithms and evaluated them on several text transformations: deletion of comments from a program, insertion of quotes around words, tag extraction from XML documents, reversing dictionaries, and the reordering and aligning of misplaced fields in BibTeX files. The evaluation algorithm for consistent DReX scales to large inputs (less than 8 seconds for 100,000 characters), while the dynamic programming algorithm, due to the cubic complexity in the size of the input, does not scale in practice (more than 60 seconds for 5,000 characters) and therefore has limited applicability. Finally, the consistency-checking algorithm is very fast in practice (less than 0.6 seconds for programs of size $\approx 3,600$ subexpressions), and it is also very helpful in identifying sources of ambiguity in the implemented programs.

In summary we offer the following contributions.

1. DReX, a language for describing string transformations that extends the combinators proposed in [3] to model strings over arbitrary sorts, and consistent DReX, a fragment of DReX that admits efficient evaluation without sacrificing expressiveness (section 2);
2. an algorithm for evaluating consistent DReX programs in a single left-to-right pass that is linear in the size of the input string and polynomial in the size of the program (section 3);
3. a dynamic programming algorithm for evaluating unrestricted DReX programs that has cubic time complexity in the length of the input string (section 4);
4. a proof that adding a composition operator to DReX causes the evaluation problem to become PSPACE-complete, and the dynamic programming algorithm to run in time exponential in the size of the program (section 4); and
5. an implementation of DReX together with an evaluation of our algorithms on practical string transformations (section 5).

2. The Syntax and Semantics of DReX

2.1 Regular combinators for string transformations

Given a character $a \in \Sigma$, and an output string $d \in \Gamma^*$, the function $a \mapsto d : \Sigma^* \rightarrow \Gamma_\perp^*$ maps the single-character input string $\sigma = a$ to the output d , and is undefined for all other inputs:¹

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

Another basic function is $\epsilon \mapsto d$ which maps the empty string ϵ to the output $d \in \Gamma^*$, and is undefined everywhere else. The final basic function bottom is undefined for all input strings.

The split sum operators are the counterparts of concatenation in regular expressions. Given an input string σ , if there exists a unique split $\sigma = \sigma_1\sigma_2$, such that both $\llbracket f_1 \rrbracket(\sigma_1)$ and $\llbracket f_2 \rrbracket(\sigma_2)$ are defined, then

$$\begin{aligned} \llbracket \text{split}(f_1, f_2) \rrbracket(\sigma) &= \llbracket f_1 \rrbracket(\sigma_1)\llbracket f_2 \rrbracket(\sigma_2), \text{ and} \\ \llbracket \text{left-split}(f_1, f_2) \rrbracket(\sigma) &= \llbracket f_2 \rrbracket(\sigma_2)\llbracket f_1 \rrbracket(\sigma_1). \end{aligned}$$

For all other inputs (where there is either no split, or multiple viable splits), both functions are undefined. Note the insistence on a unique parse tree — this is so that programs define functions, rather than relations.

Given two DReX functions f_1 and f_2 , the function $f_1 \text{ else } f_2$ first tries to apply f_1 , and if this fails, applies f_2 :

$$\llbracket f_1 \text{ else } f_2 \rrbracket(\sigma) = \begin{cases} \llbracket f_1 \rrbracket(\sigma) & \text{if } \llbracket f_1 \rrbracket(\sigma) \neq \perp, \text{ and} \\ \llbracket f_2 \rrbracket(\sigma) & \text{otherwise.} \end{cases}$$

This is the unambiguous counterpart of the union operator of traditional regular expressions.

Similarly, if both $\llbracket f_1 \rrbracket(\sigma)$ and $\llbracket f_2 \rrbracket(\sigma)$ are defined, then

$$\llbracket \text{combine}(f_1, f_2) \rrbracket(\sigma) = \llbracket f_1 \rrbracket(\sigma)\llbracket f_2 \rrbracket(\sigma).$$

If either function is undefined for the input σ , $\text{combine}(f_1, f_2)$ is undefined as well. This combinator can be used to make multiple passes over the input string, and a typical example would be the function that copies the input string twice: σ transformed into $\sigma\sigma$. In terms of the input domain, the operator combine is the counterpart of intersection in regular languages, and is necessary to achieve

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

expressive parity with regular string transformations because of the non-commutativity of string concatenation.

If f is a DReX program, and the input string σ can be uniquely split into substrings $\sigma = \sigma_1\sigma_2\dots\sigma_n$, with $n \geq 0$, and such that $\llbracket f \rrbracket(\sigma_i) \neq \perp$, for each i , then

$$\begin{aligned} \llbracket \text{iterate}(f) \rrbracket(\sigma) &= \llbracket f \rrbracket(\sigma_1)\llbracket f \rrbracket(\sigma_2)\dots\llbracket f \rrbracket(\sigma_n), \text{ and} \\ \llbracket \text{left-iterate}(f) \rrbracket(\sigma) &= \llbracket f \rrbracket(\sigma_n)\llbracket f \rrbracket(\sigma_{n-1})\dots\llbracket f \rrbracket(\sigma_1). \end{aligned}$$

Otherwise (if the input σ cannot be split, or if multiple viable splits exist), then both iterated sums are undefined. This is the counterpart of Kleene-* of regular expressions.

The chained sum operator allows us to “mix” outputs produced by different parts of the input string. This is a new operator, without a regular expression counterpart, and is necessary for expressive completeness. Let R be a regular expression that defines the language $\llbracket R \rrbracket = L$, and f be a DReX program. Given an input σ , if there is a unique split $\sigma = \sigma_1\sigma_2\dots\sigma_n$, such that $\sigma_i \in L$ for each i , then, if $n \geq 2$

$$\begin{aligned} \llbracket \text{chain}(f, R) \rrbracket(\sigma) &= \llbracket f \rrbracket(\sigma_1\sigma_2)\llbracket f \rrbracket(\sigma_2\sigma_3)\dots \\ &\quad \llbracket f \rrbracket(\sigma_{n-1}\sigma_n), \text{ and} \\ \llbracket \text{left-chain}(f, R) \rrbracket(\sigma) &= \llbracket f \rrbracket(\sigma_{n-1}\sigma_n)\llbracket f \rrbracket(\sigma_{n-2}\sigma_{n-1})\dots \\ &\quad \llbracket f \rrbracket(\sigma_1\sigma_2). \end{aligned}$$

For notational convenience, we treat $\sigma\perp = \perp\sigma = \perp$, and so if $\llbracket f \rrbracket(\sigma_i\sigma_{i+1})$ is undefined for any i , both functions are undefined. Furthermore, if a unique split of the input string σ does not exist, both the chained and left-chained sums are undefined. Notice that the regular expression R in $\text{chain}(f, R)$ defines the split of the input string, and f is applied to each pair of adjacent splits.

The final operator is function composition. If f_1 and f_2 are DReX programs such that $\llbracket f_1 \rrbracket : \Sigma^* \rightarrow \Gamma_\perp^*$, and $\llbracket f_2 \rrbracket : \Gamma^* \rightarrow \Lambda_\perp^*$, are partial functions, $\text{compose}(f_1, f_2)$ is defined as

$$\llbracket \text{compose}(f_1, f_2) \rrbracket(\sigma) = \llbracket f_2 \rrbracket(\llbracket f_1 \rrbracket(\sigma)),$$

with the notational convention that $\llbracket f_1 \rrbracket(\perp) = \perp$.

Recall that regular string transformations can be defined in multiple equivalent ways: as two-way finite state transducers, as one-way streaming string transducers, and as MSO-definable graph transformations. We summarize the main result of [3]:

Theorem 1 (Expressive completeness). *For every finite input alphabet Σ , and output alphabet Γ , every regular string transformation $f : \Sigma^* \rightarrow \Gamma_\perp^*$ can be expressed by a DReX program.*

More precisely, when we include the chained sum, function composition is unnecessary for expressive completeness, while the chained sum can itself be expressed using function composition, and so if composition is included, the chained sum is unnecessary.

2.2 Character sorts and predicates

Consider the basic combinator $a \mapsto d$ we described in the previous subsection, which maps the input $\sigma = a$ to the output d . For large alphabets, such as the set of all Unicode characters, this approach of explicitly mentioning each character does not scale. Basic transformations in DReX may therefore also reference symbolic predicates and character functions, as we will now describe. This is inspired by the recent development of symbolic transducers [26], which has proved to be useful in several practical applications.

Let Σ, Γ, \dots be a collection of character sorts. For each character sort Σ , we pick a (possibly infinite) collection of predicates P_Σ such that (a) 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 (b) 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 the sort $\Sigma_2 = \{a, b\}$ together with the set of predicates $P_{\Sigma_2} = \{x = a, x = b, \text{true}, \text{false}\}$. Another example is the set of all integers \mathbb{Z} , and with $P_\mathbb{Z} = \{\text{odd}(x), \text{even}(x), \text{true}, \text{false}\}$. We will write \mathbb{U} for the set of all Unicode characters, with the various character properties $P_\mathbb{U} = \{\text{uppercase}(x), \text{digit}(x), \dots\}$.

If $\varphi \in P_\Sigma$ and $d = [d_1, d_2, \dots, d_k]$ is a list of character transformations, i.e. $d_i : \Sigma \rightarrow \Gamma$, then $\varphi \mapsto d$ is a basic transformation which maps every single-character string σ which satisfies $\varphi(\sigma)$ to the output string $d_1(\sigma)d_2(\sigma)\dots d_k(\sigma)$, and is undefined for all other strings.

For example, the function $\text{uppercase}(x) \mapsto \text{tolowercase}(x)$ transforms every upper-case Unicode character to lower-case, while the function $\text{uppercase}(x) \mapsto xx$ 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 $x \in [2-9]$, the function $x \in [2-9] \mapsto x - 2$ 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 outputs 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 [12].

2.3 Consistent DReX programs

We now define consistent DReX, a restricted class which still captures all regular string transformations but for which we can provide an efficient evaluation algorithm (section 3). 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 purpose of the consistency rules is for the correctness of the evaluation algorithm, we defer their motivation to subsection 3.6.

2.3.1 Consistent unambiguous regular expressions

The consistency rules we propose are based on the notion of consistent unambiguous regular expression (CURE). CUREs are similar to conventional regular expressions, but with the additional guarantee that all matched strings have unique parse trees. Unambiguous regular expressions have been studied in the literature [7, 8, 25] — we explicitly qualify them as consistent here to emphasize that there are no strings with multiple parse trees. They are defined inductively as follows:

1. \perp and ϵ are CUREs. The language associated with \perp is the empty set $\llbracket \perp \rrbracket = \emptyset$, and the language associated with ϵ is the singleton $\llbracket \epsilon \rrbracket = \{\epsilon\}$.
2. For each satisfiable predicate $\varphi \in P_\Sigma$, φ is a CURE. The language $\llbracket \varphi \rrbracket$ associated with the CURE φ is the set of all single-character strings $\{x \in \Sigma \mid \varphi(x) \text{ holds}\}$.
3. For each pair of non-empty CUREs R_1 and R_2 , if the associated languages $L_1 = \llbracket R_1 \rrbracket$ and $L_2 = \llbracket R_2 \rrbracket$ are disjoint, then $R_1 \cup R_2$ is also a CURE, and $\llbracket R_1 \cup R_2 \rrbracket = L_1 \cup L_2$.
4. Given a pair of non-empty CUREs 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 $\sigma_1 \in \llbracket R_1 \rrbracket$ and $\sigma_2 \in \llbracket R_2 \rrbracket$. If R_1 and R_2 are unambiguously concatenable, then $R_1 \cdot R_2$ is also a CURE, and $\llbracket R_1 \cdot R_2 \rrbracket = \llbracket R_1 \rrbracket \llbracket R_2 \rrbracket$.
5. A non-empty CURE R is *unambiguously iterable* if for every string σ , there is at most one split $\sigma = \sigma_1\sigma_2\dots\sigma_n$ into substrings such that $\sigma_i \in \llbracket R \rrbracket$ for each i . If R is unambiguously iterable, then R^* is also a CURE, and $\llbracket R^* \rrbracket = \llbracket R \rrbracket^*$.

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 substrings is in Σ — but $(\Sigma^*)^*$ is not unambiguous.²

We call two CUREs R_1 and R_2 *equivalent*, and write $R_1 \equiv R_2$, if $\llbracket R_1 \rrbracket = \llbracket R_2 \rrbracket$.

2.3.2 Consistency rules

A consistent DReX program 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 definitions of subsection 2.1 is unnecessary. The domain of a DReX program f is the set containing every string σ such that $\llbracket f \rrbracket(\sigma)$ is defined. In the following rules, we assign each consistent DReX program a *domain type*, which is a representation of its domain as a CURE.

Most of the consistency rules are straight-forward, except for `chain` and `combine`. Recall from the definition of `chain` that for `chain(f, R)` to be defined on a string σ , we must have a split $\sigma = \sigma_1\sigma_2\dots\sigma_n$ with $n \geq 2$. Therefore the domain type of `chain(f, R)` requires at least two matches of R : $R \cdot R \cdot R^*$. Next, we want expressions of the form `chain(combine(split(f11, f12), ..., split(fk1, fk2)), R · R)`, where R is unambiguously concatenable with itself, and the domain type of each f_{ij} is $R_{ij} \equiv R$, to be consistent. We therefore pay special attention to the rule for `combine`.

1. All basic functions `bottom`, $\epsilon \mapsto d$, and $\varphi \mapsto d$ (where φ is satisfiable) are consistent. Their domain types are \perp , ϵ and φ respectively.
2. If f_1 and f_2 are both consistent and have unambiguously concatenable domain types R_1 and R_2 respectively, then `split(f1, f2)` and `left-split(f1, f2)` are also both consistent and have the domain $R_1 \cdot R_2$.
3. If f is consistent and has domain type R , and R is unambiguously iterable, then `iterate(f)` and `left-iterate(f)` are both consistent, with domain R^* .
4. If f_1 and f_2 are consistent with disjoint domain types R_1 and R_2 respectively, then `f1 else f2` is also consistent with the domain $R_1 \cup R_2$.
5. If f is consistent, and has domain type $R_1 \cdot R_2$, such that $R_1 \equiv R_2 \equiv R$, where R is an unambiguously iterable CURE, then `chain(f, R)` and `left-chain(f, R)` are both consistent, and have the domain $R \cdot R \cdot R^*$.
6. If f_1 and f_2 are consistent with domain types R_1 and R_2 respectively, and $R_1 \equiv R_2$, then `combine(f1, f2)` is also consistent. Depending on the syntactic structure of the CUREs R_1 and R_2 , the domain type of `combine(f1, f2)` is defined as follows:
 - (a) If $R_1 = R_{11} \cdot R_{12}$, and $R_2 = R_{21} \cdot R_{22}$, with $R_{11} \equiv R_{12} \equiv R_{21} \equiv R_{22}$, then the domain type of `combine(f1, f2)` is R_1 .
 - (b) Otherwise, if R_1 is *not* of the form $R_{11} \cdot R_{12}$ with $R_{11} \equiv R_{12}$, then the domain type is R_1 .
 - (c) Otherwise, the domain type is R_2 .

²To be consistent with the traditional notation, we write Σ to denote the CURE that accepts all input characters, but in our setting, it would formally be written as the predicate `true`.

We now strengthen the claim originally made in theorem 1. While consistency was not an explicit goal in the original proof of theorem 1, it is the case that every expression constructed was actually consistent, and we can therefore state:

Theorem 2. *For every finite input alphabet Σ , and output alphabet Γ , every regular function $f : \Sigma^* \rightarrow \Gamma^*$ can be expressed by a consistent DReX program.*

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 CUREs R_1 and R_2 , are R_1 and R_2 unambiguously concatenable?”, “Given a CURE R , is it unambiguously iterable?”, “Given CUREs R_1 and R_2 , are they disjoint, or equivalently, is $R_1 \cup R_2$ also a CURE?” Observe that the traditional algorithm [24] 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 [8]: 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 CUREs R_1 and R_2 , is $R_1 \equiv R_2$?” If Σ is finite, then from [25], we have that this can be checked in time $\mathcal{O}(\text{poly}(|R_1|, |R_2|, |\Sigma|))$. Otherwise, if CUREs 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 exponential³ in the number of predicates appearing in R_1 and R_2 .

Theorem 3. *Given a DReX program 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 $\mathcal{O}(\text{poly}(|f|, |\Sigma|))$.*

Note specifically that programs involving function composition are not consistent. In the rest of this paper, to distinguish the class of consistent DReX programs from the bigger class of *all* DReX programs, we will qualify the latter as the unrestricted class.

2.4 Examples of consistent DReX programs

The simplest non-trivial DReX program is the identity function $id = \text{iterate}(\text{true} \mapsto x)$. Several variations of this program are also useful: `iterate(lowercase(x) \mapsto touppercase(x))` maps strings of lower-case characters to upper-case, and $id_{\text{-space}} = \text{iterate}(\neg \text{space}(x) \mapsto x)$ is the identity function restricted to strings not containing a space.

More interesting functions can be constructed using the conditional operator: the function $sw\text{-case} = \text{uppercase}(x) \mapsto \text{tolowercase}(x) \text{ else } \text{lowercase}(x) \mapsto \text{touppercase}(x)$ flips

³The algorithm proposed in [25] can check in polynomial time whether two unambiguous NFAs are equivalent. The algorithm requires the alphabet to be finite, and using the Minterm generation technique proposed in [14] one can make a symbolic alphabet finite by constructing the Boolean combinations of the predicates in the automaton. This operation however can cause an exponential blow-up.

the case of a single input character, and so `iterate(sw-case)` switches the case of each character in the input string.

Given a string of the form “*First-name Last-name*”, the function `echo-first = split(id-space, space(x) ↦ ε, iterate(true ↦ ε))` outputs “*First-name*”. Similarly, the function `echo-last` which outputs the last name could be written, and the two can be combined into `combine(echo-last, echo-first)`, which outputs “*Last-name First-name*”. Note that the space in between is omitted — the expression `combine(split(echo-last, ε ↦ " "), echo-first)` preserves this space. An example of the use of the left-additive operators is in string reversal: the function `left-iterate(true ↦ x)` reverses the input string.

Finally, to present an example of the chained sum combinator, we consider the situation of misaligned titles in BibTeX files. Assume that, by mistake, the title of the first entry appears in the second entry, the title of the second entry appears in the third entry, and so on. Let R_{bib} be the unambiguous regular expression matching a BibTeX entry. Let f_{bib} be the DReX program which examines pairs of adjacent BibTeX entries and outputs the title of the second entry and all other fields from the first entry. Then `chain(f_{bib} , R_{bib})` corrects the misaligned text in the input file. We now outline the construction of f_{bib} . Let `echo-header` be the function which maps each BibTeX entry to its header (such as “@book{”), and `echo-title` be the function which maps each BibTeX entry to its title. Then `make-title = split(echo-header, echo-title)` copies the header from the first BibTeX entry and the title from the second. Similarly, if `copy-body` echoes all the fields of a BibTeX entry except the title, and `delete-entry` maps an entire BibTeX entry to the empty string ϵ , then `make-body = split(copy-body, delete-entry)` completes the body of the output using the fields of the first BibTeX entry. We can then write $f_{bib} = \text{combine}(\text{make-title}, \text{make-body})$.

The reader is referred to appendix A for a description of the consistent DReX programs used in our evaluation.

3. A Single-Pass Algorithm for Consistent DReX

In this section, we present the main technical contribution of this paper: a single-pass linear time algorithm to evaluate consistent DReX programs. We describe the intuition and present the idea of *function evaluators* in subsection 3.1, and then construct the evaluators for each DReX combinator. We conclude with subsection 3.6, a brief discussion of why this algorithm does not work with unrestricted DReX programs. Full proofs, and the omitted case of the chained sum will be included in the full version of this paper.

3.1 Intuition and the idea of function evaluators

Given a consistent DReX program f , we construct an evaluator T which computes the associated function $\llbracket f \rrbracket$. 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 `split(f , g)`. In this case, T is given the sequence of input signals $(Start, 0), (\sigma_1, 1), (\sigma_2, 2), \dots, (\sigma_n, n)$. The first signal $(Start, 0)$ 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 `split(f , g)` on $\sigma_1\sigma_2 \dots \sigma_i$, 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, where f is defined for the prefixes $\sigma_1\sigma_2 \dots \sigma_i$ and $\sigma_1\sigma_2 \dots \sigma_j$.

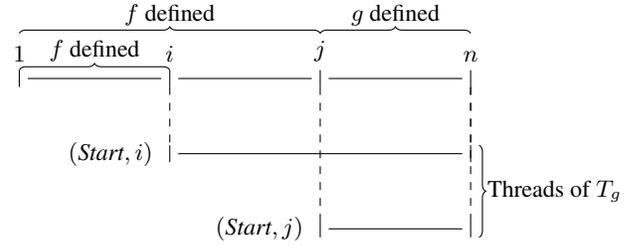


Figure 3.1: Example run of the evaluator T for `split(f , g)` over a string σ . The evaluator T_f emits a result at indices i and j of the input string. The evaluator T_g for g may simultaneously be processing multiple threads, corresponding to different potential parse trees of the input string σ . 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 .

The input to the sub-evaluator T_g is then the sequence $(\sigma_1, 1), (\sigma_2, 2), \dots, (\sigma_i, i), (Start, i), (\sigma_{i+1}, i+1), \dots, (\sigma_j, j), (Start, j), \dots, (\sigma_n, n)$.

For each signal $(Start, i)$ occurring in the input string, we call the subsequent sequence of characters $\sigma_{i+1}\sigma_{i+2} \dots$ 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 $\mathcal{O}(|g|)$, 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 `split(f , g)`. To uniquely identify the thread j reporting the result, the result signal $(Result, j, \gamma_g)$ 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 $(Result, 0, \gamma_f)$ initially obtained from f at position j , and itself emits the result $(Result, 0, \gamma_f\gamma_g)$. 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 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 $\sigma_{i+1}\sigma_{i+2} \dots \sigma_k\tau$, 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.

Formal specification of the evaluator. The input alphabet to each evaluator is therefore $In = (\Sigma \cup \{Start\}) \times \mathbb{N}$ 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 program.

While constructing the evaluator T for a DReX program 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 $\llbracket f \rrbracket(\sigma_{i+1}\sigma_{i+2} \dots \sigma_j) = \gamma$, 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 $\mathcal{O}(|f|)$ active threads, where $|f|$ is the size of f .

If an evaluator T satisfies these requirements for f , then we say that the evaluator *computes* f . On the input $(Start, 0), (\sigma_1, 1), (\sigma_2, 2), \dots, (\sigma_n, n)$, the evaluator outputs a result γ in exactly those cases when f is defined, and in that case, $\llbracket f \rrbracket(\sigma) = \gamma$. We will ensure that the evaluator T processes each input signal in time $\mathcal{O}(\text{poly}(|f|))$.⁴

3.2 Basic evaluators

The simplest case is when $f = \text{bottom}$. 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.

Claim 4. The evaluator T_{\perp} computes bottom .

Proof. First, observe that the basic function bottom is not defined on any string, and the definition of T_{\perp} never returns a result. Thus, the correctness requirement is trivially satisfied. Second, there is never any thread σ and some potential suffix τ such that bottom is defined for the input string $\sigma\tau$. Thus, by killing a thread immediately after receiving the corresponding start signal, $(Start, i)$, T_{\perp} satisfies the requirement for the eagerness of kills. \square

Next, we consider the evaluator $T_{\epsilon \mapsto d}$, for the case when $f = \epsilon \mapsto d$, for some $d \in \Gamma^*$. 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 $(Result, i, d)$. Update $th := th \cup \{i\}$.
2. On input (a, i) , for $a \in \Sigma$, respond with $(Kill, j)$, for each thread start index $j \in th$. Update $th := \emptyset$.

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 $T_{\epsilon \mapsto d}$ to each input signal is bounded by a constant.

Claim 5. The evaluator $T_{\epsilon \mapsto d}$ computes $\epsilon \mapsto d$.

Proof. There are two parts to showing that $T_{\epsilon \mapsto d}$ correctly declares results:

1. First, consider some thread $(Start, i)$ such that at index j along the input string, $\epsilon \mapsto d$ is defined. This can only happen when $j = i$, and by definition, $T_{\epsilon \mapsto d}$ declares a result in this case.
2. Conversely, consider the case when $T_{\epsilon \mapsto d}$ declares a result. By a textual reading of the program, this can only happen in response to a $Start$, and $\epsilon \mapsto d$ is defined in all these cases.

Similarly, it can be shown that $T_{\epsilon \mapsto d}$ kills threads eagerly. Therefore, the evaluator just constructed computes $\epsilon \mapsto d$. \square

The final basic function is $f = \varphi \mapsto d$ for some character predicate φ and $d : (\Sigma \rightarrow \Gamma)^*$. The evaluator $T_{\varphi \mapsto d}$ 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 satisfied the predicate φ .

1. On input $(Start, i)$, update $th := th \cup \{i\}$.

⁴ 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.

2. On input (a, i) , for $a \in \Sigma$:

- (a) Emit $(Kill, j)$, for each thread $j \in th'$.
- (b) If a satisfies the predicate φ , for each thread $j \in th$, emit $(Result, j, d(a))$. Update $th' := th$, and $th := \emptyset$.
- (c) If a does not satisfy the predicate φ , then for each thread $j \in th$, emit $(Kill, j)$. Update $th := \emptyset$, and $th' := \emptyset$.

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

Claim 6. If φ is satisfiable, then the evaluator $T_{\varphi \mapsto d}$ computes $\varphi \mapsto d$.

Proof. Because of the similarity in construction, the proof of this claim is similar to the proof of claim 5. For every thread $(Start, i)$ such that $\varphi \mapsto d$ is defined at input index j , it has to be the case that $j = i + 1$, and σ_j satisfies φ . By construction, $T_{\varphi \mapsto d}$ reports a result on this thread. Conversely, these are the only cases when the evaluator may return a result, and they all correspond to cases where $\varphi \mapsto d$ is actually defined. Therefore, the evaluator correctly reports results, and similarly, it follows that it kills threads eagerly as well. \square

3.3 State-free evaluators: combination and conditionals

The simplest non-trivial evaluator is for $\text{combine}(f, g)$. 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 $T_{\text{combine}(f, g)}$ 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 $(Result, i, \gamma_f)$ from T_f and the result $(Result, i, \gamma_g)$ from T_g (which, according to the consistency requirements for $\text{combine}(f, g)$, have to occur simultaneously), respond with $(Result, i, \gamma_f \gamma_g)$.
4. On receiving the kill signals $(Kill, i)$ from T_f and T_g (by the consistency rules, necessarily simultaneously), emit the kill signal $(Kill, i)$.

Observe that the set of valid input sequences to $T_{\text{combine}(f, g)}$ is the same as the set of valid input sequences to T_f and to T_g . Hence, the set of input signal sequences transmitted to T_f and T_g by $T_{\text{combine}(f, g)}$ are valid. Also, note that $T_{\text{combine}(f, g)}$ responds to each input signal in time $\mathcal{O}(1) + t_f + t_g$, where t_f and t_g are the response times of the sub-evaluators.

Claim 7. The evaluator $T_{\text{combine}(f, g)}$ computes $\text{combine}(f, g)$.

Proof. First, by the consistency requirement, the threads on which $\text{combine}(f, g)$ is defined are the same as the threads on which f is defined (and the threads on which g is defined). Therefore, by the induction hypotheses that T_f and T_g report results correctly, we can establish that $T_{\text{combine}(f, g)}$ correctly reports results as well. Similarly, the set of inactive threads of $\text{combine}(f, g)$ is equal to the set of inactive threads of f and the set of inactive threads of g . Thus, by the induction hypotheses that T_f and T_g eagerly kill threads, it follows that $T_{\text{combine}(f, g)}$ eagerly kills threads as well. \square

The evaluator $T_{f \text{ else } g}$ 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 $T_{f \text{ else } g}$ is defined as follows:

1. On receiving the input $(Start, i)$, update $th_f := th_f \cup \{i\}$, and $th_g := th_g \cup \{i\}$. 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 $(Result, i, \gamma)$ (by the consistency rules, we know that the other sub-evaluator is not responding with a result), emit the result $(Result, i, \gamma)$.
4. If a kill signal $(Kill, i)$ is received from T_f (resp. T_g), update $th_f := th_f \setminus \{i\}$ (resp. $th_g := th_g \setminus \{i\}$). 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 $T_{f \text{ else } g}$ responds to each input signal in time $\mathcal{O}(|f| + |g|) + t_f + t_g$, where t_f and t_g are the response times of T_f and T_g respectively.

Claim 8. $T_{f \text{ else } g}$ computes $f \text{ else } g$ when it is consistent.

Proof. Consider a thread $(Start, i)$ on which $f \text{ else } g$ is defined. Then one of two cases has to hold: first, if f is defined on the thread, by the induction hypothesis that T_f reports results correctly, it follows that $T_{f \text{ else } g}$ responds with a result. Second, in the case when g is defined on the thread, the induction hypothesis for T_g confirms that $T_{f \text{ else } g}$ responds with a result.

Conversely, if $T_{f \text{ else } g}$ reports a result $(Result, i, \gamma)$, this has to always be in response to one of T_f or T_g reporting a result. Therefore, by the induction hypothesis that the sub-evaluators are correct, we establish that $f \text{ else } g$ is defined on the thread beginning at index i .

Finally, consider any inactive thread $(Start, i_0)$ of $T_{f \text{ else } g}$. It has to be the case that $(Start, i_0)$ is an inactive thread of both T_f and T_g . Since th_f and th_g are the sets of active threads in T_f and T_g respectively, and $T_{f \text{ else } g}$ kills a thread as soon as it disappears from both sets, we know that T kills threads eagerly as well. \square

3.4 Stateful evaluators: iteration and split sum

We now construct evaluators for $\text{iterate}(f)$ and $\text{split}(f, g)$. The evaluators for $\text{left-iterate}(f)$ and $\text{left-split}(f, g)$ are symmetric with respect to concatenation and can be constructed similarly.

First, we build the evaluator $T_{\text{iterate}(f)}$, where f is consistent and has the unambiguously iterable domain type R_f . Whenever $T_{\text{iterate}(f)}$ 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 3.2. After reading each input symbol, say (σ_n, n) , T_f may report that f is defined for a suffix of the input stream $(Start, k), (\sigma_{k+1}, k+1), \dots, (\sigma_n, n)$ seen so far. The evaluator $T_{\text{iterate}(f)}$ 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 $T_{\text{iterate}(f)}$, 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 $T_{\text{iterate}(f)}$ does the following:

1. On input $(Start, i)$:
 - (a) Update $th := th \cup \{(i, i, \epsilon)\}$.
 - (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 $(Result, i, \epsilon)$.

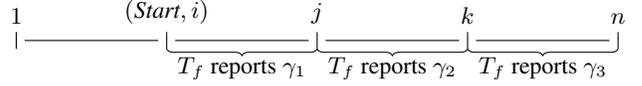


Figure 3.2: For each thread $(Start, i_0)$ of the evaluator $T_{\text{iterate}(f)}$, there may be multiple potential parse trees. The evaluator $T_{\text{iterate}(f)}$ maps individual threads $(Start, i)$ of T_f to the corresponding start signal $(Start, i_0)$ in $T_{\text{iterate}(f)}$ through the entry (i_0, i, γ) in the set th . Thus, after obtaining the response from T_f at index n , $T_{\text{iterate}(f)}$ updates $th := th \cup \{(i, n, \gamma_1 \gamma_2 \gamma_3)\}$.

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 $th := th \cup \{(j_0, i, \gamma \gamma_f)\}$.
 - 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 $th := th \setminus \text{kill-ring}$.
 - iii. For every entry $(j_0, j, \gamma) \in \text{kill-ring}$ if there is no entry of the form (j_0, j', γ') in th , then emit the kill signal $(Kill, j_0)$.

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

Claim 9. For all consistent DReX programs $\text{iterate}(f)$, the evaluator $T_{\text{iterate}(f)}$ computes $\text{iterate}(f)$.

Proof. We first establish that the sequence of input signals sent to T_f is valid. By induction on the number of intermediate start signals sent to T_f in a given thread, we have that whenever T_f responds with a signal, $\text{iterate}(f)$ is defined as well. By the assumption that L_f is unambiguously iterable, and that the input sequence to $T_{\text{iterate}(f)}$ is itself valid, it follows that no more than one active thread of T_f may report a result after each input character, and hence the input sequence is valid. Furthermore, $T_{\text{iterate}(f)}$ only returns results on receiving a start signal, or when it receives a result from T_f . Hence, we have also established that every result from $T_{\text{iterate}(f)}$ corresponds to $\text{iterate}(f)$ actually being defined on the relevant thread.

Conversely, consider the case when $\text{iterate}(f)$ is defined for some thread. By induction on the number of divisions of the input string σ , and the hypothesis that T_f correctly returns results for f , we have that $T_{\text{iterate}(f)}$ returns a result as well.

Finally, each entry (i_0, i, γ) corresponds to an active thread of execution of $T_{\text{iterate}(f)}$. We emit a kill signal as soon as a particular i_0 disappears from th and hence $T_{\text{iterate}(f)}$ eagerly kills threads. Note that the size $|th|$ is bounded by the number of active threads of T_f . \square

The evaluator $T_{\text{split}(f,g)}$ for $\text{split}(f,g)$ 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 $T_{\text{split}(f,g)}$, when T_f was started, and the value γ_f returned by T_f on the prefix. Both sets are initialized to \emptyset , and $T_{\text{split}(f,g)}$ follows the following rules:

1. On input (Start, i) :
 - (a) Update $th_f := th_f \cup \{i\}$.
 - (b) Send (Start, i) to T_f . Let Rsp_f be the responses from T_f . Let $Rsp_g = \emptyset$.
2. On input (a, i) :
 - (a) Send (a, i) to T_f . Let Rsp_f be the set of responses from T_f .
 - (b) Send (a, i) to T_g . Let Rsp_g be the set of responses from T_g .
3. For each response $r \in Rsp_f \cup Rsp_g$, do the following:
 - (a) If r is a result $(\text{Result}, j_0, \gamma_f)$ from T_f ,
 - i. Update $th_g := th_g \cup \{(j_0, i, \gamma_f)\}$.
 - ii. Send the signal (Start, i) to T_g . Update $Rsp_g := Rsp_g \cup Rsp'_g$, where Rsp'_g is the set of responses from T_g .
 - (b) If r is a result $(\text{Result}, j, \gamma_g)$ from T_g , let (j_0, j, γ_f) be the (by the consistency rules, necessarily unique) corresponding record in th_g . Respond with $(\text{Result}, j_0, \gamma_f \gamma_g)$.
 - (c) If r is a kill signal (Kill, j_0) from T_f ,
 - i. Update $th_f := th_f \setminus \{j_0\}$.
 - 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: (Kill, j_0) .
 - (d) Finally, if r is a kill signal (Kill, j) from T_g ,
 - 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 $th_g := th_g \setminus \text{kill-ring}$.
 - iii. For every record $(j_0, j, \gamma_f) \in \text{kill-ring}$, 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 : (Kill, j_0) .

By the consistency restrictions, the set of valid input sequences to $T_{\text{split}(f,g)}$ is a subset of the set of the set of valid input sequences to T_f : hence, by the induction hypothesis, it follows that the input sequences to both T_f and T_g are valid.

It is clear from construction that $T_{\text{split}(f,g)}$ reports results correctly. Consider a thread which cannot return any further results: in particular, T_f cannot return any further results, and for all threads spawned by T_f , T_g cannot return any further results. Thus, $T_{\text{split}(f,g)}$ kills threads eagerly. We therefore have:

Claim 10. Whenever the program $\text{split}(f,g)$ is consistent, the evaluator $T_{\text{split}(f,g)}$ computes $\text{split}(f,g)$.

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, $T_{\text{split}(f,g)}$ responds to each input signal in time $\mathcal{O}(|f| + |g|) + t_f + t_g$, where t_f and t_g are the response times of T_f and T_g respectively.

3.5 Chained sum

By induction on the type-checking rules, it can be shown that whenever $\text{chain}(f, R)$ is consistent, we can write

$$f = \text{combine}(\text{split}(f_{11}, f_{12}), \dots, \text{split}(f_{k1}, f_{k2})).$$

Note that the combine operator is associative, and hence we can freely extend it into a multi-arity function. In this subsection, we will only construct an evaluator for $\text{chain}(\text{split}(f,g), R)$, where L is unambiguously iterable, and the domain types satisfy $R_f \equiv R_g \equiv R$. The full case of the combination of multiple split sums can be done similarly.

Intuitively, the evaluator $T_{\text{chain}(\text{split}(f,g), R)}$ which we will now construct is approximately a combination of the evaluators $T_{\text{split}(f,g)}$ and $T_{\text{iterate}(f)}$ for $\text{split}(f,g)$ and $\text{iterate}(f)$ respectively.

The evaluator maintains three sets (see figure 3.3): (a) $th_f \subseteq \mathbb{N}$ is the set of active threads of T_f which were initiated in response to $T_{\text{chain}(\text{split}(f,g), L)}$ receiving a start signal, (b) $th_g \subseteq \mathbb{N} \times \Gamma^*$ is the set of records (i_g, γ_f) , which relate each active thread (Start, i_g) of T_g to the value γ_f returned by T_f immediately before its invocation, and (c) $th \subseteq \mathbb{N} \times \mathbb{N} \times \Gamma^*$ consists of records (i_0, i_g, γ) , that describe potential parse trees of the input string σ . For each thread i_g of T_g , this notes the index i_0 along the input string at which $T_{\text{chain}(\text{split}(f,g), L)}$ received the start signal, and γ is the value of $\text{chain}(\text{split}(f,g), L)$ at index i_g . All three sets are initialized to \emptyset .

The evaluator $T_{\text{chain}(\text{split}(f,g), R)}$ obeys the following rules:

1. On receiving (Start, i) : (a) update $th_f := th_f \cup \{i\}$, and (b) send (Start, i) to T_f .
2. On receiving (a, i) , send (a, i) to T_f and T_g . Let Rsp_f and rsp_g be the set of their responses. For each response $r \in Rsp_f \cup Rsp_g$,
 - (a) If $r = (\text{Result}, j, \gamma_f)$, and is emitted by T_f , (a) update $th_g := th_g \cup \{(i, \gamma_f)\}$, (b) send the signal (Start, i) to T_g , and (c) if $j \in th_f$, then update $th := th \cup \{(j, j, i, \epsilon)\}$, and send the signal (Start, i) to T_f .
 - (b) If $r = (\text{Result}, j, \gamma_g)$, and is emitted by T_g :
 - i. Let (j, γ_f) be the (by the consistency rules, necessarily unique) record in th_g , for some values of j_0 and γ_f .
 - ii. Let (j_0, j, γ) be the (by the consistency rules, necessarily unique) record in th , for some values of j_0 and γ .
 - iii. Update $th := th \cup \{(j_0, i, \gamma_f \gamma_g)\}$.
 - iv. Send the signal (Start, i) to T_f .
 - v. Respond with $(\text{Result}, j_0, \gamma_f \gamma_g)$.
 - (c) If $r = (\text{Kill}, j)$, and is emitted by T_f :
 - i. Update $th_f := th_f \setminus \{j\}$.
 - ii. If there is no element of the form (j, k, γ) in th , emit (Kill, j) .
 - iii. Delete the record $(j_0, j, \gamma) \in th$. This record is not guaranteed to exist, but if it does, it is unique. If $j_0 \notin th_f$, and there is no other record of the form (j_0, j', γ') in th , then emit (Kill, j_0) .
 - (d) If $r = (\text{Kill}, j)$, and is emitted by T_g , ignore r . Observe that, by the consistency requirements, T_f must have also emitted a kill signal at the same location.

First, observe that the sequence of input signals to both T_f and T_g are valid. This is because $R_f \equiv R_g \equiv R$, which is unambiguously iterable, and a start signal to the sub-evaluators is only sent in response to a result from one of them, or a start signal from the external world.

From the observation of input validity, and the consistency rules, we are guaranteed that both T_f and T_g do not respond to a start signal.

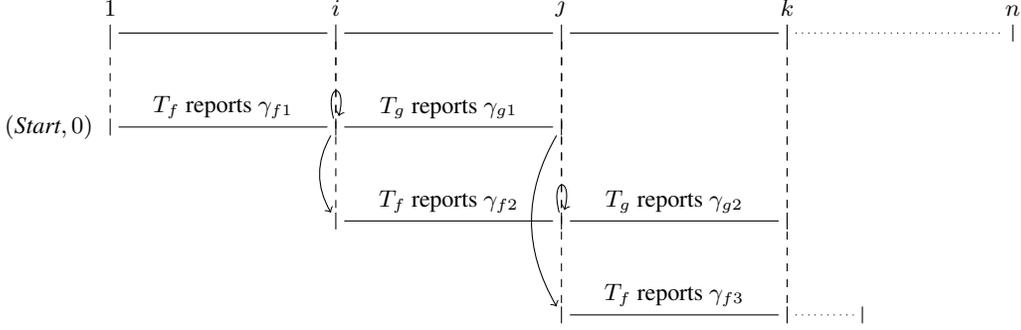


Figure 3.3: The data structures maintained by the evaluator $T_{\text{chain}(\text{split}(f,g),R)}$. The evaluator received $(\text{Start}, 0)$ at the beginning of the string. Until T_f reports $(\text{Kill}, 0)$, $th_f = \{0\}$. When T_f reports γ_{f3} at the bottom of the figure, the entry (k, γ_{f3}) is added to th_g . In response to T_g reporting γ_{g2} , T emits $(\text{Result}, 0, \gamma_{f1}\gamma_{g1}\gamma_{f2}\gamma_{g2})$, and adds $(0, k, \gamma_{f1}\gamma_{g1}\gamma_{f2}\gamma_{g2})$ to th .

By induction on the parse tree of a thread, it can be shown that $T_{\text{chain}(f,R)}$ reports each result. Conversely, every result reported by $T_{\text{chain}(f,R)}$ corresponds to a parse tree of the thread, and therefore results are correctly reported. By similar techniques as for iterated sum, it can be shown that $T_{\text{chain}(f,R)}$ eagerly kills threads. Thus, we have:

Claim 11. Whenever $\text{chain}(f,R)$ is consistent, the evaluator $T_{\text{chain}(f,R)}$ computes $\text{chain}(f,R)$.

Finally, since $|th_f|$ is bounded by the maximum number of active threads of T_f , and $|th_g|$ and $|th|$ are bound by the number of active threads of T_g , it follows that the $T_{\text{chain}(f,R)}$ responds to each input signal in time $\mathcal{O}(|f| + |g|) + t_f + t_g$, where t_f and t_g are the response times of the sub-evaluators.

3.6 What breaks with unrestricted DReX programs?

First, notice that the function composition operator is un-amenable 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 $\text{compose}(f,g)$: when the evaluator T_f returns a result $(\text{Result}, i, \gamma)$, 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 $\text{split}(f,g)$, in the absence of any consistency requirement. Thus, there might exist strings $\sigma = \sigma_1\sigma_2 \dots \sigma_n$ which admit two splits $\sigma = \tau_1\tau_2 = \tau'_1\tau'_2$, such that all of $\llbracket f \rrbracket(\tau_1)$, $\llbracket f \rrbracket(\tau'_1)$, $\llbracket g \rrbracket(\tau_2)$, and $\llbracket g \rrbracket(\tau'_2)$ are defined. In this case, $\text{split}(f,g)$ 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. Consider the case of $f = \text{split}(f_1, \text{split}(f_2, f_3))$, and an input string $\sigma = \sigma_1\sigma_2 \dots \sigma_n$ which can be split in three ways as shown in figure 3.4. In this case, the evaluator T_{23} for $\text{split}(f_2, f_3)$ receives three *Result* signals from T_3 after processing σ_n . However, it must still emit an output signal $(\text{Result}, j, \gamma_2\gamma_3)$, and so the behavior of T_{23} has to be more complicated than “emit a *Result* signal when exactly one thread of T_3 returns”. The consistency rules, which eliminate the case of such strings, provide an easy way to avoid this non-trivial book-keeping.

Similarly, the disjointedness of domains requirement for the conditional combinator ensures that both nested combinators T_f

and T_g of the evaluator T for $f \text{ else } g$ never return simultaneously. This simplifies the construction of T .

By semantics of `combine`, the domain of $\text{combine}(f,g)$ is $\text{Dom}(f) \cap \text{Dom}(g)$, where $\text{Dom}(f)$ and $\text{Dom}(g)$ are the domains over which f and g are respectively defined. By the consistency requirement, we force that $\text{Dom}(f) = \text{Dom}(g) = \text{Dom}(f) \cap \text{Dom}(g)$, thus simplifying the structure of the domain type and hence the type-checking algorithm.

Finally, note that unrestricted use of the chained sum operator does not satisfy the output bound $\llbracket \llbracket f \rrbracket(\sigma) \rrbracket = \mathcal{O}(\text{poly}(|f|)|\sigma|)$, while every consistent DReX program obeys $\llbracket \llbracket f \rrbracket(\sigma) \rrbracket \leq |f||\sigma|$.

4. The Complexity of Unrestricted DReX

In this section we first describe the dynamic programming algorithm to evaluate DReX programs. We show that it has time complexity cubic in the size of the input string, and when function composition is allowed, requires time exponential in the size of the program. We then show that the evaluation problem for DReX programs with composition is PSPACE-complete, and is thus computationally hard. Finally, we argue that for unrestricted DReX programs, even when disallowing composition, there is no evaluation algorithm for which the complexity is linear in the length of the input string and polynomial in the size of the program.

4.1 Evaluation by dynamic programming

We now describe the dynamic programming algorithm to evaluate unrestricted DReX programs. This algorithm also works on programs containing the function composition operator. The algorithm mimics the semantics of DReX by computing the following functions (represented as lookup tables). Given a program f , and a string σ , for any two numbers i and j the algorithm computes the function $\text{OUT}(f, \sigma, i, j)$ representing the output of f on the substring⁵ $\sigma[i, j]$, $\llbracket f \rrbracket(\sigma[i, j])$. To evaluate the operator $\text{iterate}(f)$ we also need to compute the function $\text{COUNT}(\text{iterate}(f), \sigma, i, j)$ that counts the number of possible ways to split $\sigma[i, j]$ so that each split is accepted by f , and to evaluate the operator $\text{chain}(f, R)$ we compute the function $\text{BEL}(R, \sigma, i, j)$ which checks whether a substring $\sigma[i, j]$ belongs to the language $\llbracket R \rrbracket$. Every function OUT , COUNT , and BEL is represented by a table and for each string σ , and each sub-program g of f , each table will have $\mathcal{O}(|\sigma|^2)$ entries, corresponding to the substrings of σ . The final output of the algorithm is $\text{OUT}(f, \sigma, 1, |\sigma| + 1)$.

⁵ We define the substring $\sigma[i, j]$ of σ as $\sigma_i\sigma_{i+1} \dots \sigma_{j-1}$, so that $\sigma[i, i] = \epsilon$, for each i , and $\sigma[1, |\sigma| + 1]$ is the entire string σ .

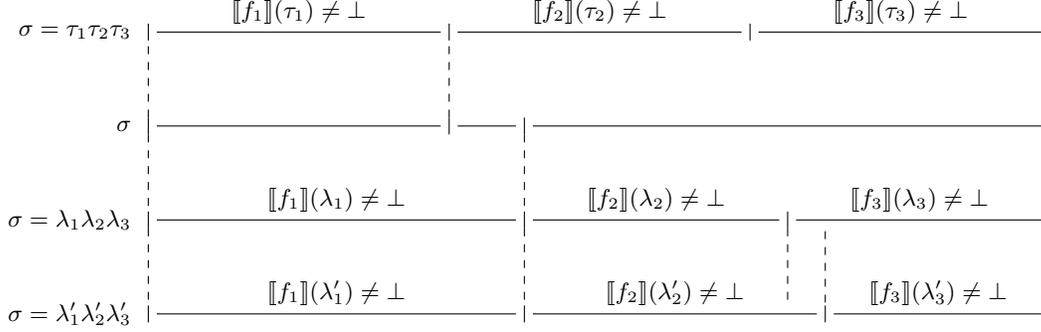


Figure 3.4: The evaluator approach does not work for unrestricted DReX programs. The string $\sigma = \sigma_1\sigma_2 \dots \sigma_n$ is passed as input to the inconsistent program $f = \text{split}(f_1, \text{split}(f_2, f_3))$. The input σ can be split in three ways $\sigma = \tau_1\tau_2\tau_3 = \lambda_1\lambda_2\lambda_3 = \lambda_1\lambda'_2\lambda'_3$. By the semantics of `split`, `split(f2, f3)` is undefined over the input substring $\lambda_2\lambda_3 = \lambda'_2\lambda'_3$, and hence f itself is defined for the entire string σ . After processing the last character σ_n , the evaluator T_{23} for `split(f2, f3)` has to correctly process three *Result* signals from T_3 and this seems tricky to implement correctly.

We explain the intuition of the algorithm by showing how the entries are computed for the iteration and composition operators. The value $\text{OUT}(\text{iterate}(f), \sigma, i, j)$ corresponding to the output of `iterate(f)` on the string $\sigma' = \sigma[i, j]$ is defined iff there is a unique way to split the string σ' into multiple chunks so that f is defined on each chunk, i.e. iff $\text{COUNT}(\text{iterate}(f), \sigma, i, j) = 1$. If this is the case, then we know that there is a unique value k , such that $i \leq k < j$, for which both $\gamma_{pre} = \text{OUT}(\text{iterate}(f), \sigma, i, k)$ and $\gamma_{post} = \text{OUT}(f, \sigma, k, j)$ are defined, and $\gamma_{pre}\gamma_{post}$ is the output of `iterate(f)` on $\sigma[i, j]$. Looking for this witness k takes at most $|\sigma|$ steps if all the required table entries have already been computed. Similarly, the entry $\text{COUNT}(\text{iterate}(f), \sigma, i, j)$ can be computed in at most $|\sigma|$ steps by counting for how many values of l the function $\text{COUNT}(\text{iterate}(f), \sigma, i, l)$ is greater than 0 and the functions $\text{OUT}(f, \sigma, l, j)$ is defined.

The rule for computing $\text{OUT}(\text{compose}(f_1, f_2), \sigma, i, j)$ is what causes an exponential blow-up in the evaluation time. To compute the output of `compose(f1, f2)` on $\sigma[i, j]$, we first need to compute the output $\tau = \text{OUT}(f_1, \sigma, i, j)$ of the program f_1 on the string $\sigma[i, j]$ and then the output $\text{OUT}(f_2, \tau, 1, |\tau| + 1)$ of the program f_2 on the string τ . As we will show, when using the composition operator the size of the output τ may grow exponentially, and since computing each entry of the table $\text{OUT}(f_2, \tau, \cdot, \cdot)$ requires $\mathcal{O}(\tau)$ steps, the resulting complexity is also exponential.

The complete set of rules to construct the lookup table OUT is summarized in figure 4.1. We will now analyze the complexity of this algorithm, parameterized by the number of composition operators appearing in the program. In [3] it is shown that the `chain` operator can be equivalently replaced by the composition operator. Therefore, we define:

Definition 12 (Composition depth). Given a program f in DReX the *composition depth* of f , $C(f)$ is the number of composition or chained-sum operators in f , and it may be computed as follows:

1. $C(\varphi \mapsto d) = 0$, $C(\epsilon \mapsto d) = 0$, and $C(\text{bottom}) = 0$;
2. $C(\text{compose}(f_1, f_2)) = 1 + C(f_1) + C(f_2)$;
3. $C(\text{chain}(f_1, R)) = C(\text{left-chain}(f_1, R)) = 1 + C(f_1)$;
4. $C(\text{combine}(f_1, f_2)) = C(\text{split}(f_1, f_2)) = C(\text{left-split}(f_1, f_2)) = C(f_1 \text{ else } f_2) = C(f_1) + C(f_2)$;
5. $C(\text{iterate}(f_1)) = C(\text{left-iterate}(f_1)) = C(f_1)$;

We can then prove the following bound on the length of the output of a DReX program.

Lemma 13 (Output size). *Given a program f and an input string $\sigma \in \Sigma^*$ such that $\tau = \llbracket f \rrbracket(\sigma)$ is defined, we have $|\tau| \leq |f|^{d+1} |\sigma|$. Here d is the composition depth of f .*

Proof. We prove the statement by structural induction on f . The base case is immediate since the output of $f = \varphi \mapsto d$ has length $d \leq |f|$. We now present the inductive step for the interesting cases:

1. When $f = \text{compose}(f_1, f_2)$. Let $\sigma_1 = \llbracket f_1 \rrbracket(\sigma)$. By the IH, we have $|\sigma'| \leq |f_1|^{C(f_1)+1} |\sigma|$, and $|\tau| \leq |f_2|^{C(f_2)+1} |\sigma'|$. Therefore,

$$\begin{aligned} |\tau| &\leq |f_2|^{C(f_2)+1} |f_1|^{C(f_1)+1} |\sigma| \\ &\leq |f|^{C(f_2)+1} |f|^{C(f_1)+1} |\sigma| \\ &\leq |f|^{C(f_1)+C(f_2)+2} |\sigma| \\ &\leq |f|^{C(f)+1} |\sigma|. \end{aligned}$$

2. When $f = \text{chain}(f_1, R)$. Let $\sigma = \sigma_1\sigma_2 \dots \sigma_n$, with $n \geq 2$ be the split of the input string induced by R . Then by the IH,

$$\begin{aligned} |\tau| &\leq |f_1|^{C(f_1)+1} (|\sigma_1\sigma_2| + |\sigma_2\sigma_3| + \dots + |\sigma_{n-1}\sigma_n|) \\ &\leq 2|f_1|^{C(f_1)+1} |\sigma| \\ &\leq |f|^{C(f_1)+1+1} |\text{strvar}| \\ &\leq |f|^{C(f)+1} |\sigma|. \end{aligned}$$

3. When $f = \text{combine}(f_1, f_2)$. Let $\tau_1 = \llbracket f_1 \rrbracket(\sigma)$ and $\tau_2 = \llbracket f_2 \rrbracket(\sigma)$, so that $\tau = \tau_1\tau_2$. By the IH,

$$\begin{aligned} |\tau| &= |\tau_1| + |\tau_2| \\ &\leq |f_1|^{C(f_1)+1} |\sigma| + |f_2|^{C(f_2)+1} |\sigma| \\ &\leq |f| (|f|^{C(f_1)} + |f|^{C(f_2)}) |\sigma| \\ &\leq |f|^{C(f_1)+C(f_2)+1} |\sigma| \\ &\leq |f|^{C(f)+1} |\sigma|. \end{aligned}$$

The other cases are similar. \square

Theorem 14 (Complexity of dynamic programming). *Given a program f , and an input string $\sigma \in \Sigma^*$, the output $\llbracket f \rrbracket$ can be computed in time $\mathcal{O}(|f|^{2d^2+5d+4} |\sigma|^{2d+3})$ where $d = C(f)$ is the number of composition operators in f . If $d = 0$ we can show that the algorithm has complexity $\mathcal{O}(|f| |\sigma|^3)$.*

$$\begin{aligned}
\text{OUT}(\varphi \mapsto d, \sigma, i, j) &= \begin{cases} d(\sigma_i) & \text{if } i+1 = j, \text{ and } \varphi(\sigma_i) \text{ is true, and} \\ \perp & \text{otherwise.} \end{cases} & \text{OUT}(\epsilon \mapsto d, \sigma, i, j) &= \begin{cases} d & \text{if } \sigma[i, j] = \epsilon, \text{ and} \\ \perp & \text{otherwise.} \end{cases} \\
\text{OUT}(\text{bottom}, \sigma, i, j) &= \perp. \\
\text{OUT}(\text{split}(f_1, f_2), \sigma, i, j) &= \begin{cases} \tau_1 \tau_2 & \text{if } \exists! k \text{ such that } i \leq k \leq j, \text{ and where } \tau_1 = \text{OUT}(f_1, \sigma, i, k) \neq \perp, \text{ and} \\ & \tau_2 = \text{OUT}(f_2, \sigma, k, j) \neq \perp, \text{ and} \\ \perp & \text{otherwise.} \end{cases} \\
\text{OUT}(\text{left-split}(f_1, f_2), \sigma, i, j) &= \begin{cases} \tau_2 \tau_1 & \text{if } \exists! k \text{ such that } i \leq k \leq j, \text{ and where } \tau_1 = \text{OUT}(f_1, \sigma, i, k) \neq \perp, \text{ and} \\ & \tau_2 = \text{OUT}(f_2, \sigma, k, j) \neq \perp, \text{ and} \\ \perp & \text{otherwise.} \end{cases} \\
\text{OUT}(f_1 \text{ else } f_2, \sigma, i, j) &= \begin{cases} \text{OUT}(f_1, \sigma, i, j) & \text{if } \text{OUT}(f_1, \sigma, i, j) \neq \perp, \text{ and} \\ \text{OUT}(f_2, \sigma, i, j) & \text{otherwise.} \end{cases} \\
\text{OUT}(\text{combine}(f_1, f_2), \sigma, i, j) &= \text{OUT}(f_1, \sigma, i, j) \text{OUT}(f_2, \sigma, i, j). \\
\text{OUT}(\text{iterate}(f), \sigma, i, j) &= \begin{cases} \epsilon & \text{if } i = j \text{ and } \text{OUT}(f, \sigma, 0, 0) = \perp, \\ \tau_1 \tau_2 & \text{otherwise if } \exists! k \text{ such that } i \leq k \leq j \wedge \tau_1 = \text{OUT}(\text{iterate}(f), \sigma, i, k) \neq \perp \wedge \tau_2 = \\ & \text{OUT}(f, \sigma, k, j) \neq \perp, \text{ and} \\ \perp & \text{otherwise.} \end{cases} \\
\text{OUT}(\text{left-iterate}(f), \sigma, i, j) &= \begin{cases} \epsilon & \text{if } i = j \text{ and } \text{OUT}(f, \sigma, 0, 0) = \perp, \\ \tau_2 \tau_1 & \text{otherwise if } \exists! k \text{ such that } i \leq k \leq j \wedge \tau_1 = \text{OUT}(\text{iterate}(f), \sigma, i, k) \neq \perp \wedge \tau_2 = \\ & \text{OUT}(f, \sigma, k, j) \neq \perp, \text{ and} \\ \perp & \text{otherwise.} \end{cases} \\
\text{OUT}(\text{chain}(f, R), \sigma, i, j) &= \begin{cases} \perp & \text{if } i = j, \\ \text{OUT}(f, \sigma, i, j) & \text{otherwise if } \text{COUNT}(R^*, \sigma, i, j) = 1, \text{ and } \exists k \text{ such that } i \leq k \leq j, \text{ BEL}(R, \sigma, i, k) \\ & \text{and BEL}(R, \sigma, k, j), \\ \tau_1 \tau_2 & \text{otherwise if } \text{COUNT}(R^*, \sigma, i, j) = 1, \text{ and } \exists k, l \text{ such that } i \leq k \leq l \leq j, \\ & \text{BEL}(R, \sigma, i, k), \text{ BEL}(R, \sigma, k, l), \text{ and where } \tau_1 = \text{OUT}(f, \sigma, i, l) \neq \perp \text{ and } \tau_2 = \\ & \text{OUT}(\text{chain}(f, R), \sigma, k, j) \neq \perp, \text{ and} \\ \perp & \text{otherwise.} \end{cases} \\
\text{OUT}(\text{left-chain}(f, R), \sigma, i, j) &= \begin{cases} \perp & \text{if } i = j, \\ \text{OUT}(f, \sigma, i, j) & \text{otherwise if } \text{COUNT}(R^*, \sigma, i, j) = 1, \text{ and } \exists k \text{ such that } i \leq k \leq j, \text{ BEL}(R, \sigma, i, k) \\ & \text{and BEL}(R, \sigma, k, j), \\ \tau_2 \tau_1 & \text{otherwise if } \text{COUNT}(R^*, \sigma, i, j) = 1, \text{ and } \exists k, l \text{ such that } i \leq k \leq l \leq j, \\ & \text{BEL}(R, \sigma, i, k), \text{ BEL}(R, \sigma, k, l), \text{ and where } \tau_1 = \text{OUT}(f, \sigma, i, l) \neq \perp \text{ and } \tau_2 = \\ & \text{OUT}(\text{chain}(f, R), \sigma, k, j) \neq \perp, \text{ and} \\ \perp & \text{otherwise.} \end{cases} \\
\text{OUT}(\text{compose}(f_1, f_2), \sigma, i, j) &= \text{OUT}(f_2, \tau, 1, |\tau| + 1), \text{ where } \tau = \text{OUT}(f_1, \sigma, i, j) \text{ and } \tau \neq \perp.
\end{aligned}$$

Figure 4.1: Summary of the lookup table construction rules for the dynamic programming algorithm of subsection 4.1. Recall that $\sigma[i, j]$ refers to the substring $\sigma_i \sigma_{i+1} \dots \sigma_{j-1}$, and $\text{OUT}(f, \sigma, i, j)$ is the value of f on the input $\sigma[i, j]$, $\llbracket f \rrbracket(\sigma[i, j])$. For the iteration operation $\text{iterate}(f)$, the lookup table entry $\text{COUNT}(\text{iterate}(f), \sigma, i, j)$ contains the number of splits of the input substring $\sigma[i, j]$ such that each split is accepted by f , and $\text{BEL}(R, \sigma, i, j)$ is true iff $\sigma[i, j] \in \llbracket R \rrbracket$. We use the notation $\exists!$ as shorthand for the phrase “exists a unique”, so that $\exists! x, \varphi(x)$ is true iff there is a unique value x satisfying the predicate φ .

Proof. We first discuss the complexity of the algorithm in the absence of composition operators. For a particular string σ and if the program f that does not contain any composition operators ($d = 0$), computing each entry of the table takes time $\mathcal{O}(|\sigma|)$, and since there are $|f| |\sigma|^2$ entries the algorithm has complexity $\mathcal{O}(|f| |\sigma|^3)$. The linear complexity required for computing each entry can be shown by inspecting each rule in figure 4.1.

In the presence of composition operators which can produce intermediate results, for each intermediate string τ , a new table of size $\mathcal{O}(|\tau|^2)$ must be created. Assume that we are computing the entry $\text{OUT}(f, \sigma, i, j)$ of the table for a string σ and $f = \text{compose}(f_1, f_2)$. At this point we will need the value $\tau = \text{OUT}(f_1, \sigma, i, j)$, whose computation does not require a new table, and the value $\text{OUT}(f_2, \tau, 1, |\tau|)$ which requires building a new table for the string τ . Computing $\text{OUT}(f, \sigma, \cdot, \cdot)$ may therefore require to build $\mathcal{O}(|\sigma|^2)$ new tables that will need to be evaluated on f_2 . This would happen whenever we have a composition: the left sub-program f_1 is evaluated on the same number of tables as f while the right sub-program f_2 will be evaluated on $\mathcal{O}(|\sigma|^2)$ tables, where σ is the size of the biggest string f is evaluated on. The total number of computed tables is therefore $\mathcal{O}(|f| |S|^{2d})$, where S is the size of the biggest intermediate result f can produce and d is the number of composition operators in f .

From lemma 13 we know that the biggest string S that the program can produce has size $\mathcal{O}(|f|^{d+1} |\sigma|)$. Therefore the total number of computed tables is $\mathcal{O}(|f|^{2d^2+1} |\sigma|^{2d})$ and each table has size at most $\mathcal{O}(|f|^{d+1} |\sigma|)$. We know that computing each table takes time cubic in the size of the corresponding string and we can conclude that the algorithm runs in time $\mathcal{O}(|f|^{2d^2+5d+4} |\sigma|^{2d+3})$ where $d = C(f)$. \square

4.2 Evaluating unrestricted DReX programs in PSPACE

While the main appeal of the algorithm in subsection 4.1 is ease of implementation, it can use exponential space. It turns out that, even in the presence of composition, DReX programs can be evaluated in PSPACE. First observe that, since the output computed by a program has at most exponentially many characters (lemma 13), the index of each character in the output is only polynomially many bits long. We therefore adopt an implicit representation of strings with the following operations (in contrast with the traditional explicit list-of-characters representation of strings): (a) check whether σ is defined; (b) compute the length of σ ; and (c) given an index i , compute the i -th character of σ . Finally, by structural induction on the DReX program f , and given an implicit representation of the input string σ , we build an implicit representation of $\llbracket f \rrbracket(\sigma)$ using only polynomial space. For example, the implicit representation of $\llbracket f \text{ else } g \rrbracket(\sigma)$ would function as follows: (a) to check whether the output is defined, simply determine whether either $\llbracket f \rrbracket(\sigma)$ or $\llbracket g \rrbracket(\sigma)$ is defined; (b) to compute the length of the output, if $\llbracket f \rrbracket(\sigma)$ is defined, return the length of $\llbracket f \rrbracket(\sigma)$, and otherwise, return the length of $\llbracket g \rrbracket(\sigma)$; and (c) to compute the i -th character of $\llbracket f \text{ else } g \rrbracket(\sigma)$, if $\tau_1 = \llbracket f \rrbracket(\sigma)$ is defined, then return the i -th character of τ_1 , and otherwise, return the i -th character of $\tau_2 = \llbracket g \rrbracket(\sigma)$. Observe that since both nested implicit representations $\llbracket f \rrbracket(\sigma)$ and $\llbracket g \rrbracket(\sigma)$ consume only polynomial space, $\llbracket f \text{ else } g \rrbracket(\sigma)$ is itself evaluated in polynomial space. The most interesting case is $\text{compose}(f, g)$ where we simply connect the implicit representation of the output of $\llbracket f \rrbracket(\sigma)$ to the input of the function g . The only non-trivial case is when $f = \text{iterate}(g)$. 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 15. *Given a DReX program f , and strings $\sigma \in \Sigma^*$ and $\tau \in \Gamma^*$, the problem of determining whether $\llbracket f \rrbracket(\sigma) = \tau$ is in PSPACE.*

Proof. We will construct, by structural induction on f , three machines isdef_g , len_g , and charat_g for each subexpression g of f . Given an input string σ' represented implicitly, these machines together provide an implicit representation of $\llbracket g \rrbracket(\sigma')$. Therefore:

1. $\text{isdef}_g(\sigma')$ is a boolean value indicating whether g is defined for the input string σ' ,
2. $\text{isdef}_g(\sigma')$ is a non-negative integer indicating the length of $\llbracket g \rrbracket(\sigma')$ (assuming that g is defined on the input σ'), and
3. $\text{charat}_g(\sigma', i)$, assuming that $\llbracket g \rrbracket(\sigma')$ is defined and $1 \leq i \leq |\llbracket g \rrbracket(\sigma')|$, is the i -th character of $\llbracket g \rrbracket(\sigma')$.

We now present the construction for each DReX combinator. We will ensure that each machine maintains only a polynomial amount of book-keeping space, and only accesses its input through the implicit string interface. We make free use of the fact that $\text{PSPACE} = \text{NPSPACE}$.

1. When $f = \varphi \mapsto d$. If σ is undefined, or if $|\sigma| \neq 1$, or otherwise if $\neg\varphi(\sigma_1)$, then $\text{isdef}_f(\sigma) = \text{false}$. In all other cases, $\text{isdef}_f(\sigma) = \text{true}$. $\text{len}_f(\sigma) = |d|$. $\text{charat}_f(\sigma, i) = d_i(\sigma_1)$. The construction is similar for $\epsilon \mapsto d$ and bottom .
2. When $f = \text{split}(f_1, f_2)$ (the construction for the case of $\text{left-split}(f_1, f_2)$ is similar). If σ is undefined, then $\text{isdef}_f(\sigma) = \text{false}$. Otherwise, we non-deterministically guess a viable split index i , $1 \leq i \leq |\sigma| + 1$, such that $\text{isdef}_{f_1}(\sigma[1, i]) \wedge \text{isdef}_{f_2}(\sigma[i, |\sigma| + 1])$.⁶ Because $\text{PSPACE} = \text{NPSPACE}$, isdef_f can guess this split in polynomial space. To confirm that this is the only viable split, we attempt to non-deterministically guess another split index i' , $1 \leq i' \leq |\sigma| + 1$, such that $i' \neq i$ and $\text{isdef}_{f_1}(\sigma[1, i']) \wedge \text{isdef}_{f_2}(\sigma[i', |\sigma| + 1])$. Again, $\text{isdef}_f(\sigma)$ can determine the absence of such a split in polynomial space. In this case, $\text{isdef}_f(\sigma) = \text{true}$. In all other cases, $\text{isdef}_f(\sigma) = \text{false}$.
Next, len_f works as follows. On input σ , it first determines the split index i exactly as isdef_f . It then outputs $\text{len}_{f_1}(\sigma[1, i]) + \text{len}_{f_2}(\sigma[i, |\sigma| + 1])$.
Finally, we describe charat_f . On calling $\text{charat}_f(\sigma, j)$, the machine determines the split index i as before. Next, if $j \leq \text{len}_{f_1}(\sigma[1, i])$, then it outputs $\text{charat}_{f_1}(\sigma[1, i], j)$. Otherwise, it outputs $\text{charat}_{f_2}(\sigma[i, |\sigma| + 1], j')$, where $j' = j - \text{len}_{f_1}(\sigma[1, i])$.
3. The construction for $f = f_1 \text{ else } f_2$ is straightforward. $\text{isdef}_f(\sigma) = \text{isdef}_{f_1}(\sigma) \vee \text{isdef}_{f_2}(\sigma)$. If $\text{isdef}_{f_1}(\sigma)$, then $\text{len}_f(\sigma) = \text{len}_{f_1}(\sigma)$ and $\text{charat}_f(\sigma, i) = \text{charat}_{f_1}(\sigma, i)$, and otherwise $\text{len}_f(\sigma) = \text{len}_{f_2}(\sigma)$ and $\text{charat}_f(\sigma, i) = \text{charat}_{f_2}(\sigma, i)$.
4. The construction for $f = \text{combine}(f_1, f_2)$ is also simple. $\text{isdef}_f(\sigma) = \text{isdef}_{f_1}(\sigma) \wedge \text{isdef}_{f_2}(\sigma)$. $\text{len}_f(\sigma) = \text{len}_{f_1}(\sigma) + \text{len}_{f_2}(\sigma)$. If $i \leq \text{len}_{f_1}(\sigma)$, then $\text{charat}_f(\sigma, i) = \text{charat}_{f_1}(\sigma, i)$, and otherwise, $\text{charat}_f(\sigma, i) = \text{charat}_{f_2}(\sigma, i')$, where $i' = i - \text{len}_{f_1}(\sigma)$.
5. The cases for iterated sum and left-iterated sum are similar, and are technically the most involved in this proof. We will perform the construction for the case of the iterated sum, $f =$

⁶Recall the convention that $\sigma[i, j]$ is the substring $\sigma_i \sigma_{i+1} \dots \sigma_{j-1}$ of σ . Also note that it is straightforward to construct the implicit representation of $\sigma[i, j]$, given an implicit representation of σ , and indices i and j .

`iterate`(f_1). First, if $\text{isdef}_{f_1}(\epsilon)$, then $\text{isdef}_f(\sigma) = \text{false}$, since there are guaranteed to be multiple ways to split σ . Otherwise, given an input string σ , the existence of a viable split $i_1 = 1 \leq i_2 < i_3 < \dots < i_n = |\sigma| + 1$ such that $\text{isdef}_{f_1}(\sigma[i_j, i_{j+1}])$ for each j can be non-deterministically guessed and verified in polynomial space, and hence, by $\text{PSPACE} = \text{NPSpace}$, in deterministic polynomial space. Next, the existence of two distinct splits $i_1 = 1 < i_2 < \dots < i_k, i_k < i_{k+1} < \dots < i_m = |\sigma| + 1$, and $i_k < j_{k+1} < \dots < j_n = |\sigma| + 1$, with $i_{k+1} \neq j_{k+1}$ and such that $\text{isdef}_{f_1}(\sigma[i_k, j_{k+1}])$, $\text{isdef}_{f_1}(\sigma[i_l, i_{l+1}])$, $\text{isdef}_{f_1}(\sigma[j_l, j_{l+1}])$ for each l can also be non-deterministically guessed and verified in polynomial space.

If σ is defined, $\neg \text{isdef}_{f_1}(\epsilon)$, and a unique split exists of the input string σ , then $\text{isdef}_f(\sigma) = \text{true}$. In all other cases, $\text{isdef}_f(\sigma) = \text{false}$.

Let $i_1 = 1 \leq i_2 < i_3 < \dots < i_n = |\sigma| + 1$ be the viable split of σ determined as before. Then $\text{len}_f(\sigma) = \sum_{j=1}^{n-1} \text{len}_{f_1}(\sigma[i_j, i_{j+1}])$.

Finally, $\text{charat}_f(\sigma, k)$ is determined as follows. Initialize $j = 1$, and repeat until $k \leq \text{len}_{f_1}(\sigma[i_j, i_{j+1}])$: update $k := k - \text{len}_{f_1}(\sigma[i_j, i_{j+1}])$, and $j := j + 1$. Return $\text{charat}_{f_1}(\sigma[i_j, i_{j+1}], k)$.

6. We skip the cases of the chained sum operators. From the construction of [3], functions containing the chained sum operator can, in polynomial time, be rewritten to use function composition instead. Note that we take this approach only for brevity: actually constructing the machines for chained sum is actually similar to the other cases presented in this proof.
7. When $f = \text{compose}(f_1, f_2)$. Given an input string σ , let σ' be the implicit representation of $\llbracket f_1 \rrbracket(\sigma)$ produced by isdef_{f_1} , len_{f_1} and charat_{f_1} . Then, $\text{isdef}_f(\sigma) = \text{isdef}_{f_2}(\sigma')$, $\text{len}_f(\sigma) = \text{len}_{f_2}(\sigma')$, and $\text{charat}_f(\sigma, i) = \text{charat}_{f_2}(\sigma', i)$.

This completes the proof. \square

4.3 Evaluating unrestricted DRex programs is PSPACE-hard

First, we show that the following problems polynomially reduce to each other:

Theorem 16 (Equivalent Problems). *There is a polynomial time reduction between all of the following problems:*

1. given a DRex program f , check whether $\llbracket f \rrbracket(\epsilon)$ is defined;
2. given a DRex program f and an input string $\sigma \in \Sigma^*$, check whether $\llbracket f \rrbracket(\sigma)$ is defined; and
3. given a DRex program f , an input string $\sigma \in \Sigma^*$, and a potential output string $\sigma' \in \Gamma^*$, check whether $\llbracket f \rrbracket(\sigma) = \sigma'$.

Proof. We perform the following reductions: $1 \rightarrow 2$, $2 \rightarrow 1$, $2 \rightarrow 3$, and $3 \rightarrow 2$.

1. The reduction $1 \rightarrow 2$ is immediate by setting σ to ϵ .
2. To reduce the problem $2 \rightarrow 1$, consider a DRex program f and an input string $\sigma \in \Sigma^*$. We construct a program f' such that $\llbracket f' \rrbracket(\epsilon)$ is defined iff $\llbracket f \rrbracket(\sigma)$ is defined. Consider the input generation program $f_\sigma = \epsilon \mapsto \sigma$ that given the empty input string ϵ outputs the string σ and define $f' = \text{compose}(f_\sigma, f)$. We then have $\llbracket f' \rrbracket(\epsilon) = \llbracket f \rrbracket(\sigma)$.
3. To reduce the problem $2 \rightarrow 3$, consider a program f and an input string $\sigma \in \Sigma^*$. We construct a program f' such that $\llbracket f' \rrbracket(\sigma) = \epsilon$ iff $\llbracket f \rrbracket(\sigma)$ is defined. We define $f' = \text{compose}(f, f_{\text{def}})$ where $f_{\text{def}} = \text{iterate}(\text{true} \mapsto \epsilon)$ maps every input string to the empty output string ϵ . Notice that f_{def} is undefined exactly when the input itself is undefined.

4. We perform the reduction $3 \rightarrow 2$ as follows. Given a DRex program f , an input string $\sigma \in \Sigma^*$, and a candidate output string $\sigma' \in \Gamma^*$ we construct a DRex program f' such that $\llbracket f' \rrbracket(\sigma)$ is defined iff $\llbracket f \rrbracket(\sigma) = \sigma'$. We define $f' = \text{compose}(f, f_{\sigma'})$ where $f_{\sigma'}$ is defined only on the string σ' . We achieve this by defining $f_{\sigma'} = \text{split}(x = \sigma'_1 \mapsto \epsilon, x = \sigma'_2 \mapsto \epsilon, \dots, x = \sigma'_{|\sigma'|} \mapsto \epsilon)$.

This completes the proof. \square

Next we show that the first problem from theorem 16 is PSPACE-hard.

Theorem 17 (PSPACE-hardness). *Given a DRex program f it is PSPACE-hard to determine whether $\llbracket f \rrbracket(\epsilon)$ is defined.*

Proof. We perform a polynomial time reduction from the problem of determining the validity of a quantified boolean formula (QBF). A QBF is a formula

$$\Phi = \forall x_1 \exists x_2 \dots \exists x_n \varphi(x_1, x_2, \dots, x_n),$$

where $\varphi(x_1, x_2, \dots, x_n)$ is a 3CNF formula over the variables x_1, x_2, \dots, x_n , and the quantifiers in Φ strictly alternate between universal and existential. The problem of determining the validity of a QBF formula is PSPACE-complete [24].

We show how, given a QBF Φ we can construct a DRex program f_Φ such that $\llbracket f_\Phi \rrbracket(\epsilon)$ is defined iff Φ is valid. We first illustrate the general structure of our construction and then explain each gadget in detail. The program f_Φ is the of the form

$$\text{compose}(f_{01}, f_{3\text{CNF}}, f_Q),$$

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.

The program f_{01} is of the form $\text{compose}(f_1, f_2, \dots, f_n)$, and intuitively does the following: (a) f_1 produces the output $0\#1\#$, i.e. all possible assignments of the variable x_1 , separated by $\#$ -s. (b) For each $i > 1$, f_i takes as input a string $in = \sigma_1\#\sigma_2\#\dots\#\sigma_{2^{i-1}}\#$ (all possible assignments of variables x_1, x_2, \dots, x_{i-1} , and produces an output string containing all possible valuations of variables x_1, x_2, \dots, x_i . It does this by copying each assignment in in , and appending a 0 to the end of the first copy, and a 1 to the end of the second copy. The program f_1 is defined as $\epsilon \mapsto 0\#1\#$. All subsequent programs f_i , for $i > 1$, are defined as

$$\begin{aligned} f_i &= \text{let } ID_{\{0,1\}} = \text{iterate}(x \in \{0, 1\} \mapsto x) \text{ in} \\ &\quad \text{let } app_0 = \text{split}(ID_{\{0,1\}}, \# \mapsto 0\#) \text{ in} \\ &\quad \text{let } app_1 = \text{split}(ID_{\{0,1\}}, \# \mapsto 1\#) \text{ in} \\ &\quad \text{iterate}(\text{combine}(app_0, app_1)). \end{aligned}$$

Each f_i replaces each substring $\sigma\# \in \{0, 1\}^* \#$ with the string $\sigma 0\# \sigma 1\#$.

Assume the 3CNF formula φ is of the form $c_1 \wedge c_2 \wedge \dots \wedge c_m$ where each clause c_i is a disjunction of the form $l_{i_1} \vee l_{i_2} \vee l_{i_3}$. The program $f_{3\text{CNF}}$ is of the form

$$\text{compose}(q_1, q_2, \dots, q_m, f_{\text{clean}})$$

such that after executing $\text{compose}(q_1, q_2, \dots, q_i)$ on the sequence of assignments $\sigma_1 \# \sigma_2 \# \dots \# \sigma_{2^n} \#$, the output is of the form $t_1 \# t_2 \# \dots \# t_{2^n} \#$, where $t_j = \sigma_j$ if the assignment σ_j satisfies all the clauses c_1, \dots, c_i , and $t_j = F$ otherwise. After executing $\text{compose}(q_1, q_2, \dots, q_m)$, the program f_{clean} replaces every t_j different from F with a T , since that assignment satisfied all the clauses. We construct

$$q_i = \text{iterate}(\text{split}(k_i \text{ else } ID, \# \mapsto \#)),$$

where k_i maps assignments t which do not satisfy the clause c_i to F , and is undefined otherwise. We construct

$$k_i = \text{split}(r_1, r_2, \dots, r_n, \epsilon \mapsto F),$$

where for each $j \in \{1, 2, \dots, n\}$, the function r_j checks, in case that the QBF variable x_j appears in the clause c_i , whether its assignment causes the clause to be satisfied. We wish k_i to be undefined in this case, and thus we define:

$$r_j = \begin{cases} x \in \{0\} \mapsto \epsilon & \text{if the literal } x_j \text{ appears in } c_i, \\ x \in \{1\} \mapsto \epsilon & \text{if the literal } \neg x_j \text{ appears in } c_i, \text{ and} \\ x \in \{0, 1\} \mapsto \epsilon & \text{otherwise.} \end{cases}$$

Note that it is straightforward to simplify a 3CNF formula so that no clause contains both a variable and its negation.

Last, we describe the program f_Q that checks whether the sequence of assignments makes the overall formula Φ valid. To understand how f_Q works we first illustrate how a classical algorithm would now proceed to evaluate the formula Φ . In the classical algorithm for evaluating a QBF, a binary tree of depth n is built to simulate all the possible assignments of each variable. At the root the left subtree corresponds to setting the variable x_1 to 0 while the right subtree corresponds to setting it to 1. Each leaf then corresponds to a complete assignment to each variable, and the formula φ can be evaluated replacing each leaf with T or F (for true and false). The program $\text{compose}(f_{01}, f_{3CNF})$ performs this part of the algorithm: f_{01} first builds each leaf of the tree and f_{3CNF} replaced each leaf with T or F . The classical algorithm then proceeds as follows. Pick a node $N(l, r)$ such that both children l and r are evaluated to either T or F . The node N corresponds to assigning some variable x_i to either 0 or 1 in some context. If x_i is existentially quantified, assign the node N to the disjunction of the values of l and r , and if x_i is universally quantified, assign the node N to the conjunction of the values of l and r . The whole tree can then be evaluated in the same manner. Finally, if the root has value true the formula is valid. The program f_Q performs the described task:

$$f_Q = \text{compose}(e_n, e_{n-1}, \dots, e_1, f_{\text{fin}})$$

such that after executing $\text{compose}(e_n, e_{n-1}, \dots, e_i)$ on the string produced by f_{3CNF} , the output is the string $t_1 \# t_2 \# \dots \# t_{2^{i-1}} \#$ representing the evaluation of each node at level i in the evaluation tree produced by the classical algorithm we previously described. Therefore, after executing $\text{compose}(e_n, e_{n-1}, \dots, e_1)$ the output will be $T \#$ if the formula is valid or $F \#$ otherwise. The program

$$f_{\text{fin}} = \text{split}(T \mapsto \epsilon, \# \mapsto \epsilon)$$

will be defined only in the first case, that is, if the formula φ is valid. We now describe the construction of e_i . We define the gadgets f_\wedge and f_\vee that given a string over $t_1 \# t_2 \#$ respectively output a string $(t_1 \wedge t_2) \#$ and $(t_1 \vee t_2) \#$. We present only f_\wedge since f_\vee is similar:

$$\begin{aligned} f_{\wedge T} &= \text{split}(T \mapsto T, \# \mapsto \epsilon, T \mapsto \epsilon, \# \mapsto \#) \\ f_{\wedge F} &= \text{split}(\{T, F\} \mapsto F, \# \mapsto \epsilon, \{T, F\} \mapsto \epsilon, \# \mapsto \#) \\ f_\wedge &= f_{\wedge T} \text{ else } f_{\wedge F}. \end{aligned}$$

$f_{\wedge T}$ tries to assign T checks whether both assignments to the current QBF variable evaluate to true, and if so, produces a $T \#$. In all other

cases, $f_{\wedge F}$ produces the output $F \#$. (Note that this program is, by design, ill-typed.) If the variable x_i is universally quantified, then $e_i = \text{iterate}(f_\wedge)$, and otherwise, $e_i = \text{iterate}(f_\vee)$. This last construction completes the reduction of determining the validity of QBF formulas to the problem of evaluating unrestricted DReX programs. \square

Putting together theorems 15, 16 and 17, it follows that:

Theorem 18. *The following problems are PSPACE-complete:*

1. given a program f in DReX check whether $\llbracket f \rrbracket(\epsilon)$ is defined;
2. given a program f in DReX and a string $\sigma \in \Sigma^*$ check whether $\llbracket f \rrbracket(\sigma)$ is defined;
3. given a program f in DReX, a string $\sigma \in \Sigma^*$, and a string $\tau \in \Gamma^*$, check whether $\llbracket f \rrbracket(\sigma) = \tau$;

4.4 Single-pass algorithms for unrestricted DReX

In the proof of theorem 14 we showed that in the absence of the function composition combinator the dynamic programming algorithm has complexity $\mathcal{O}(f|\sigma|^3)$ where σ is the input string and f the program. In this section we argue that, if one wants to obtain an algorithm that is linear in the size of the input, it is necessary to pay at least an exponential complexity in the size of the program.

DReX operators are similar to those offered by regular expressions (iteration, split, etc. are the broad analogues of Kleene-*, concatenation, etc.). Since one can evaluate regular expressions efficiently by transforming them into nondeterministic finite automata one can try to construct an automaton model corresponding to DReX programs. Unfortunately as we discussed in section 2, DReX combinators can also express language intersection and other complex operations. In the presence of such operations, directly constructing an automaton model from the program seems hard (see [23], where the author summarizes the state-of-the-art on matching regular expressions extended with an intersection operator). The following is currently the best claim we can make about evaluating unrestricted DReX programs with a single linear time pass over the input string:

Theorem 19. *Given a DReX program f and an input string $\sigma \in \Sigma^*$, we can compute the output $\llbracket f \rrbracket(\sigma)$ in time linear in the length of the input string σ , and a tower of exponentials with height $\mathcal{O}(|f|)$.*

The algorithm compiles f into an equivalent streaming string transducer (SST) [1] using the procedure described in [3]. An SST is a finite state machine that reads the input in a left-to-right fashion and stores intermediate results inside variables. The final output is a combination of such variables. An SST A can be evaluated on an input string σ in one pass in time $\mathcal{O}(|A| |\sigma|)$. Since SSTs are deterministic, operations such as concatenation and iteration (even in the absence of function composition) cause an exponential blow-up, making the overall complexity non-elementary.

5. Evaluation

We implemented the algorithms described in this paper, and evaluated their performance on a representative set of text and BibTeX file transformations. We show that

1. the evaluation algorithm for consistent DReX scales to inputs with more than 100,000 characters (subsection 5.2.1), and
2. the dynamic programming algorithm presented in section 4.1 does not scale for inputs with more than 3,000 characters (subsection 5.2.2).

Finally, we remark on the subjective experience of expressing string transformations using DReX (subsection 5.3).

Program name	Size	CC (ms)
<i>delete_comm</i>	28	12
<i>insert_quotes</i>	28	6
<i>get_tags</i>	31	6
<i>reverse</i>	5	1
<i>swap_bibtex</i>	1663	262
<i>align_bibtex</i>	3652	537

Table 1: Evaluated programs with sizes and time to check consistency.

5.1 Implementation details

The prototype implementation of DReX was written in Java and uses the recently released Java SE 8. We used the symbolic automata library SVPALib [11] to implement the symbolic operations required by the consistency-checking algorithm (theorem 3). The set of characters were all 16-bit UTF-16 code units, and the predicates were unions of character intervals (such as $[a-z, A-Z, 0-9]$).

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. Each experiment was run 10 times and the results reported are the mean of the obtained running times.

The dynamic programming (DP) algorithm for the extended version of DReX (theorem 14) is implemented lazily. Each entry $OUT(f, \sigma, i, j)$ is only computed and allocated when its value is required by another entry. Without this technique the algorithm runs out of memory for inputs of length smaller than 1,000. We also optimize the DP algorithm to take advantage of the consistency-checking; for consistent programs the algorithm does not need to check whether there is more than one way to match a string.

5.2 Benchmark programs

Table 1 shows the programs we considered in our evaluation together with their sizes and the running time of the consistency-checking algorithm. These programs are described in appendix A. Observe that for every program the consistency-checking algorithm terminates in less than 600 ms.

We evaluated the first four programs on randomly generated text files of size varying between 1 and 100,000 characters, and evaluated the more complex functions *swap_bibtex* and *align_bibtex* on actual BibTeX files of size also varying between 1 and 100,000 characters. We set a timeout of 60 seconds for each operation.

5.2.1 Single-pass algorithm for consistent DReX

Figure 5.1 shows how the running time for the algorithm presented in section 3 depends linearly on the size of the input. For inputs up to 100,000 characters each program takes less than 8 seconds to compute the output. Also observe that the evaluation algorithm can successfully handle reasonably large programs such as *align_bibtex* which has an AST with 3652 nodes.

5.2.2 Dynamic programming for unrestricted DReX

Figure 5.2 shows the running time for the dynamic programming algorithm presented in section 4.1. The x -axis is shown in log scale to better appreciate the difference between the different programs. From the figure we can see that the running time depends polynomially on the length of the input and, as a consequence, all the considered programs time out for inputs with more than 70,000 characters. In some cases large programs such as *swap_bibtex* execute faster than smaller programs such as *get_tags*. This is due to the fact that our implementation uses some optimizations that may depend on the shape of the program: for example in the case of programs that are only defined on strings of a fixed length k , the

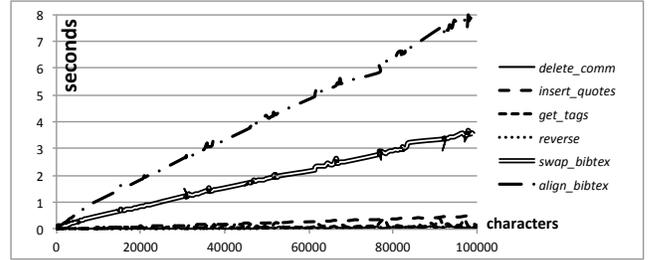


Figure 5.1: Evaluation time for the single-pass algorithm.

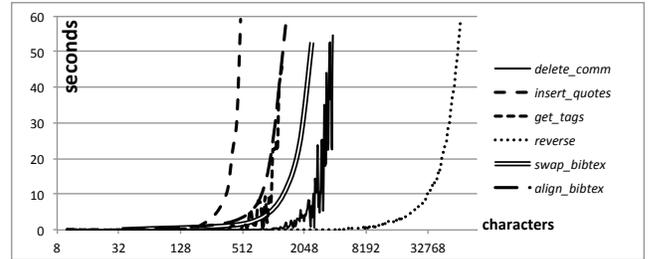


Figure 5.2: Evaluation time for dynamic programming algorithm. Note that the x -axis in log-scale.

algorithm is evaluated only for those i and j such that $j - i = k$. In conclusion, although all DReX programs can be evaluated using the dynamic programming algorithm presented in section 4.1, the procedure does not scale to large inputs.

5.3 User experience and comparison to existing tools

We were able to easily program several non-trivial string-to-string transformations without having to worry about efficiency. The main restrictions of the streaming evaluation algorithm are that programs cannot use compositions and have to be consistent. In our case study we did not find instances where composition was required and all the natural implementations of our programs were consistent. Moreover, in many cases the consistency algorithm helped us in identifying sources of ambiguity that caused our program to be incorrect. As an anecdotal account, we had mistakenly concatenated the sub-program $copy_spaces = iterate(space(x) \mapsto x)$, which copies all whitespace characters (including tabs and newlines), with itself. In this case, the type-checker warned us that this concatenation was ambiguous on the input string “ $\backslash n$ ”. This was clearly a bug in our script, and would have led to unexpected behavior even if we had used the dynamic programming evaluation algorithm.

We also programmed the benchmark transformations in sed, AWK, and Perl. Regular-expression based substitution, as present in all of these tools, is very efficient and usable to substitute or delete substrings based on patterns. This includes the benchmark programs *delete_comm* and *insert_quotes*, for which the sed implementations were ≈ 6 times faster than the DReX ones. On the other hand, *reverse*, *swap_bibtex* and *align_bibtex* were hard to express in line-based tools such as sed and AWK. The Perl implementations of these functions were ≈ 2 times faster than ours.

6. Related Work

Regular string transformations. This class of string-to-string transformations is robust and has many equivalent characterizations, including deterministic two-way string transducers [16, 18], streaming string transducers [1], transducers with origin information [6], and MSO-definable string transformations [10]. Regular

string transformations are also closed under composition [9], and enjoy decidable equivalence [21]. Alur et al proposed a set of combinators that captures the set of regular string-to-string transformations [3] and this paper builds on it. The results in [3] only focus on expressiveness and do not try to answer questions about complexity. In particular the transformation to streaming string transducers proposed in [3] has non-elementary complexity in the size of the program. In this paper, however, we are primarily driven by issues related to the complexity of evaluation.

DSLs for string transformations. DSLs for string transformations mainly fall into two classes: string specific utilities such as sed, AWK, and Perl, and transducer-based languages [13, 15, 26].

Utilities such as sed, AWK, and Perl provide the programmer with powerful programming constructs to manipulate strings. These languages are Turing-complete and in general cannot be efficiently compiled into fast executable code and are not amenable to algorithmic analysis. We also argued in section 5.3 that some of the programs that can be naturally expressed in DReX are actually hard to define using these tools.

All the existing transducer-based languages simply act as frontends to an underlying transducer model that they use to reason about the implemented programs. BEK uses symbolic finite transducers [26] and it has been used to analyze string sanitization functions. BEX is a frontend for extended symbolic finite transducers [12, 13] and it has been used to prove the correctness of string encoders and decoders such as Base64. FAST is based on symbolic tree transducers with regular look-ahead [15] and it is used to reason about programs that manipulate strings and trees over arbitrary domains. While these languages enable powerful analysis and verification techniques, (a) their semantics are tightly coupled to the transducer model, forcing the programmer to think in terms of a finite state machine, and a left-to-right reading of the input string, and (b) they only capture a strict subset of the class of regular string transformations; none of these models can reverse a string.

Another language based on automata is Boomerang, a bidirectional programming language for string editing [5]. Bidirectional programs contain combinators for extracting a view from a concrete input and then reconstructing an updated input from the updated view. Boomerang also supports extractions where each record is associated with a “key”. Although Boomerang has a similar type-system to that of DReX (it forces unambiguous operations), we are not aware of a complexity analysis for the problem of evaluating a Boomerang program. The goals of Boomerang and DReX are orthogonal: the former focuses on bidirectional transformations, while the latter focuses on efficiently evaluating all regular string transformations. Despite this difference we believe that Boomerang could benefit from the evaluation techniques proposed in this paper.

Efficient string manipulation. Little effort has been devoted to design languages and algorithms to efficiently evaluate string transformations in linear time. The general approach has been to identify an automaton model that processes the string in a single left-to-right pass and can express interesting programs. However, all existing tools that use this approach [13, 15, 26] take advantage of composition or combination operators that make the compilation to transducers exponential in the size of the program. Streaming string transducers (SST) [1] capture all programs that can be written in DReX and can be executed in a single left-to-right pass over the input. However, transforming DReX programs into SSTs also causes an super-exponential blow-up in the size of the input program.

In the context of XML processing numerous languages or fragments have been proposed for efficiently querying (XPath, XQuery), stream processing (STX [4]), and manipulating (XSLT) XML trees. Some of these languages particularly focus on efficiently processing the input in a linear time left-to-right pass. Although

in the case of XML documents with bounded depth some XML transformations can be described in DReX, the main goal of DReX remains that of providing a well-defined fragment (regular) of *string* (and not tree) transformations that can be efficiently executed.

Future Directions. A major motivation for choosing the class of regular string transformations was the decidability of analysis questions. In particular, consider regular type-checking: given a program f and two regular languages I and O , is it the case that for every input $\sigma \in I$, $\llbracket f \rrbracket(\sigma) \in O$? Such a tool would be helpful to audit string sanitizers against specific kinds of attacks. Implementing these procedures is an open research direction.

In FlashFill [20] simple string transformations can be synthesized from examples. The expressiveness of the combinators used in FlashFill has not been characterized. Can DReX programs be efficiently learnt or synthesized from input-output examples?

Recently Mytkowicz et al. [22] proposed new techniques for evaluating finite automata in a data-parallel fashion. Can these techniques be used to parallelize the evaluation algorithm proposed in this paper?

Extending our techniques to tree transformations is another open problem. Streaming tree transducers (STTs) [2] are to regular tree transformations (or equivalently macro tree transducers [19], and MSO-definable tree transformations [17]) as SSTs are related to regular string transformations. Can we design a similar declarative language to express regular tree transformations?

Finally, certain operations such as counting the number of substring matches, sorting the elements of a dictionary, or deleting duplicate entries in a list are *not* regular string transformations. Extending DReX with non-regular primitives but which can still be efficiently evaluated is an interesting direction of future work.

7. Conclusion

We presented DReX, a declarative language for describing regular string transformations. The basic transformers are symbolic, so DReX can succinctly express transformations even over large alphabets such as Unicode. We demonstrated that the evaluation problem for unrestricted DReX is PSPACE-complete, and so we considered a restricted fragment, *consistent* DReX which permits a fast single-pass evaluation algorithm, and still retains expressive completeness. In experiments over representative string transformations such as BibTeX file manipulations the evaluation algorithm for consistent DReX scaled to process thousands of characters per second.

References

- [1] R. Alur and P. Černý. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 599–610. ACM, 2011.
- [2] R. Alur and L. D’Antoni. Streaming tree transducers. In A. Czumaj, K. Mehlhorn, A. Pitts, and R. Wattenhofer, editors, *Automata, Languages, and Programming*, volume 7392 of *Lecture Notes in Computer Science*, pages 42–53. Springer, 2012.
- [3] R. Alur, A. Freilich, and M. Raghothaman. Regular combinators for string transformations. In *Proceedings of the 23rd EACSL Annual Conference on Computer Science Logic (CSL) and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS ’14, pages 9:1–9:10. ACM, 2014.
- [4] O. Becker. Streaming transformations for xml-stx. In *XMidX*, volume 24 of *LNI*, pages 83–88. GI, 2003.
- [5] A. Bohannon, N. Foster, B. Pierce, A. Pilikiewicz, and A. 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.

- [6] M. Bojańczyk. Transducers with origin information. In *Automata, Languages, and Programming*, volume 8573 of *Lecture Notes in Computer Science*, pages 26–37. Springer, 2014.
- [7] R. Book, S. Even, S. Greibach, and G. Ott. Ambiguity in graphs and expressions. *IEEE Transactions on Computers*, 20(2):149–153, February 1971.
- [8] A. Brüggemann-Klein. Regular expressions into finite automata. In *LATIN '92*, volume 583 of *Lecture Notes in Computer Science*, pages 87–98. Springer, 1992.
- [9] M. Chytil and V. Jákł. Serial composition of 2-way finite-state transducers and simple programs on strings. In *Automata, Languages, and Programming*, volume 52 of *Lecture Notes in Computer Science*, pages 135–147. Springer, 1977.
- [10] B. Courcelle. Monadic second-order definable graph transductions: a survey. *Theoretical Computer Science*, 126(1):53–75, 1994.
- [11] L. D’Antoni and R. Alur. Symbolic visibly pushdown automata. In *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 209–225. Springer, 2014.
- [12] L. D’Antoni and M. Veanes. Equivalence of extended symbolic finite transducers. In *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 624–639. Springer, 2013.
- [13] L. D’Antoni and M. Veanes. Static analysis of string encoders and decoders. In *Verification, Model Checking, and Abstract Interpretation*, volume 7737 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 2013.
- [14] L. D’Antoni and M. Veanes. Minimization of symbolic automata. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 541–553, New York, NY, USA, 2014. ACM.
- [15] L. D’Antoni, M. Veanes, B. Livshits, and D. Molnar. Fast: A transducer-based language for tree manipulation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 384–394. ACM, 2014.
- [16] J. Engelfriet and H. J. Hoogeboom. MSO definable string transductions and two-way finite-state transducers. *ACM Transactions on Computational Logic*, 2(2):216–254, April 2001.
- [17] J. Engelfriet and S. Maneth. Macro tree transducers, attribute grammars, and MSO definable tree translations. *Information and Computation*, 154(1):34–91, 1999.
- [18] J. Engelfriet, G. Rozenberg, and G. Slutzki. Tree transducers, L systems, and two-way machines. *Journal of Computer and System Sciences*, 20(2):150–202, 1980.
- [19] J. Engelfriet and H. Vogler. Macro tree transducers. *Journal of Computer and System Sciences*, 31(1):71–146, 1985.
- [20] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 317–330. ACM, 2011.
- [21] E. 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.
- [22] T. Mytkowicz, M. Musuvathi, and W. Schulte. Data-parallel finite-state machines. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 529–542. ACM, 2014.
- [23] G. Rosu. An effective algorithm for the membership problem for extended regular expressions. In *Foundations of Software Science and Computational Structures*, volume 4423 of *Lecture Notes in Computer Science*, pages 332–345. Springer, 2007.
- [24] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3rd edition, 2012.
- [25] R. Stearns and H. 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.

- [26] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner. Symbolic finite state transducers: Algorithms and applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 137–150. ACM, 2012.

A. Examples of Consistent DRex Programs

This appendix contains simplified descriptions of the consistent DRex programs that we used in our experiments in section 5. We will use the following macros in our definitions.

$$\begin{aligned} \text{copy}(\varphi) &= \varphi(x) \mapsto [x] \\ \text{del}(\varphi) &= \varphi(x) \mapsto [] \\ \text{iterate-plus}(f) &= \text{split}(\text{iterate}(f), f) \\ \text{split}(f_1, \dots, f_n) &= \text{split}(f_1, \text{split}(\dots, f_n)) \end{aligned}$$

$\text{copy}(\varphi)$ and $\text{del}(\varphi)$ respectively copy or delete a character matching φ . When the predicate is of the form $x = c$ we simply write c . $\text{iterate-plus}(f)$ repeats the function f one or more times. The last macro simply repeats the split sum operator on all the arguments.

A.1 Delete one-line comments from a program

This program deletes all one-line comments from a file (i.e. the lines starting with `//`). We define each component of the program separately. The program `del_comm_line` deletes strings of the form `// σ \n`, where σ does not contain any occurrence of the new line character `\n`.

$$\begin{aligned} \text{del_slashes} &= \text{split}(\text{del}('/'), \text{del}('/')), \\ \text{del_non_nl} &= \text{iterate}(\text{del}(x \neq '\n')), \\ \text{del_comm} &= \text{split}(\text{del_slashes}, \text{del_non_nl}), \\ \text{del_comm_line} &= \text{split}(\text{del_comm}, \text{del}('\n')). \end{aligned}$$

The program `copy_line` copies strings of the form `“ σ \n”`, where the string σ is either the empty string, or it does not start with the character `‘/’`.

$$\begin{aligned} \text{copy_non_nl} &= \text{iterate}(\text{copy}(x \neq '\n')), \\ \text{copy_txt} &= \text{split}(\text{copy}(x \neq '/'), \text{copy_non_nl}), \\ \text{copy_line} &= \text{copy}('\n') \text{ else } \text{split}(\text{copy_txt}, \text{copy}('\n')). \end{aligned}$$

The program `process_line` 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 program `process_last_line` deals with this exception. Finally the program `delete_comm` repeats `process_line` and at the end processes the last line, therefore deleting all the one-line comments in the input.

$$\begin{aligned} \text{process_line} &= \text{del_comm_line} \text{ else } \text{copy_line}, \\ \text{last_line} &= \text{del_comm} \text{ else } \text{copy_txt}, \\ \text{process_last_line} &= \text{process_line} \text{ else } \text{last_line}, \\ \text{process_lines} &= \text{iterate}(\text{process_line}), \\ \text{delete_comm} &= \text{split}(\text{process_lines}, \text{process_last_line}). \end{aligned}$$

A.2 Insert quotes around words

This program inserts quotes (`“”`) around every alphabetic substring appearing in the input. The program `add_qts_skip`, given a string $\sigma_1\sigma_2$ where σ_1 is alphabetic and σ_2 does not contain any letter,

outputs the string “ $\sigma_1\sigma_2$ ”.

```
copy_astr = iterate-plus(copy([a-zA-Z])),
copy_nastr = iterate-plus(copy(x ∉ [a-zA-Z])),
add_qt =  $\epsilon \mapsto [‘”’]$ ,
add_qts = split(add_qt, copy_astr, add_qt),
add_qts_skip = split(add_qts, copy_nastr).
```

The program *insert_quotes* repeats the *add_qts* function. Since the string might start with a symbol that is not alphabetic the program *start* deals with this case. Similarly the program *ending* checks whether the string does not end with an alphabetic sequence. Finally, the program *insert_quotes* inserts quotes around every alphabetic substring in the input.

```
start = iterate(copy(x ∉ [a-zA-Z])),
ending = add_qts else  $\epsilon \mapsto []$ ,
repeat_add = iterate(add_qts_skip),
insert_quotes = split(start, repeat_add, ending).
```

A.3 Extracting tags from a malformed XML file

This program extracts and concatenates all the substrings of the form $\langle\sigma\rangle$ where σ does not contain any character \langle or \rangle (this is a generalization of a program shown in [26]). For simplicity we assume that the string does not contain occurrences of the substring “ $\langle\rangle$ ”. The program *copy_match* copies any string of the form $\langle\sigma\rangle$ where σ does not contain any character \langle or \rangle .

```
copy_ntag = iterate-plus(copy(x ∉ [⟨⟩])),
copy_match = split(copy(‘<’), copy_ntag, copy(‘>’)).
```

The program *del_not_match* deletes any string that does not contain a substring of the form $\langle s\rangle$ where s does not contain any character \langle or \rangle . The program *del_not_match* looks for the following pattern: $s = \sigma_1\rangle\cdots\rangle\sigma_i\langle\cdots\langle\sigma_n$ (with $i \geq 1$). Its sub-program *del_close_op* deletes all the string of the described form containing at least one open character (‘ \langle ’).

```
del_not_opn = iterate(del(x ≠ ‘<’)),
del_not_cls = iterate(del(x ≠ ‘>’)),
del_opn_not_cls = split(del(‘<’), iterate(del(x ≠ ‘>’))),
del_close_op = split(del_not_opn, del_opn_not_cls),
del_not_match = del_not_opn else del_close_op.
```

The program *find_match* keeps looking for tags and removes eventual non-matches at the end of the string. Finally, *get_tags* repeats *find_match*, and therefore outputs all the substrings of the form $\langle\sigma\rangle$.

```
find_match = split(del_not_match, copy_match),
repeat_get_tags = iterate(find_match),
get_tags = split(repeat_get_tags, del_not_match).
```

A.4 Reversing a dictionary

Given a dictionary 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 program *copy_stretch* copies a string of the form σ ; such that σ does not contain a ‘;’. The program *reverse* implements the final transformation by left iterating *copy_stretch*.

```
copy_stretch = split(iterate(copy(x ≠ ‘;’)), copy(‘;’)),
reverse = left-iterate(copy_stretch).
```

A.5 Reformatting BibTeX files

In this section we define two functions that operate over BibTeX files. The first function, *swap_bibtex*, reorders attributes within each

BibTeX entry by moving the title to the top. The second function, *align_bibtex*, rearranges a *misaligned* file by moving the title of each entry inside the previous entry.

To do so we first define a few auxiliary functions that are used for copying and deleting alphabetic strings, spaces, and delimiters.

```
copy_astr = iterate-plus(copy([a-zA-Z])),
copy_anum = iterate-plus(copy([a-zA-Z0-9])),
copy_spaces = iterate(copy([\n \r \b \t])),
del_ astr = iterate-plus(del([a-zA-Z])).
```

It is easy to see how given a DReX program that copies a pattern, one can easily define a DReX program that deletes the same pattern. To simplify the presentation in the following we assume that for every program of the form *copy_something* there is an analogous program *del_something* that is defined on the same input as *copy_something* but always outputs the empty string.

The program *copy_header* copies the header of an entry. In the example of figure A.1, it copies the string “@book{Gal1638” along with the following spaces.

```
copy_header = split(copy(‘@’), copy_ astr, copy(‘{’),
copy_anum, copy(‘, ’), copy_spaces).
```

We now define macros for copying and deleting a particular string s or a set of strings.

```
copy(s) = split(copy(s[1]), ..., copy(s[|s|])),
copy({ $\sigma_1, \dots, \sigma_n$ }) = copy( $\sigma_1$ ) else ... else copy( $\sigma_n$ ).
```

The program *copy_title* copies the string “title”, and the program *copy_non_title* copies every attribute name different from “title”. We omit the full list of attributes for readability.

```
copy_title = copy(‘title’),
copy_non_title = copy({‘author’, ‘year’, ‘place’, ...}).
```

The program *copy_att_value* copies the value of an attribute along with the surrounding parentheses (i.e. “ = { Elzevir},”).

```
copy_non_par = iterate(copy(x ∉ [}{])),
copy_att_value = split(copy_spaces, copy(‘=’),
copy_spaces, copy(‘{’), copy_non_par,
copy(‘}’), copy(‘, ’), copy_spaces).
```

The program *copy_title_att* copies a complete title attribute (i.e. “title = {Two New Sciences},”), while *copy_non_title_att* copies a complete non-title attribute.

```
copy_title_att = split(copy_title, copy_att_value),
copy_non_title_att = split(copy_non_title, copy_att_value).
```

Given a list of attributes, the program *title_only* copies the title and deletes all the non-title attributes, the program *all_but_title* deletes the title and copies all the other attributes, and the program *copy_attrs* copies the entire list.

```
title_only = iterate(copy_title else del_non_title),
all_but_title = iterate(del_title else copy_non_title),
all_but_title = iterate(copy_title else copy_non_title).
```

A.5.1 Polishing a BibTeX file

The following function defines a typical transformation a paper author may perform on BibTeX files. The program *swap_bibtex* moves the title attribute to the top of each entry of an input BibTeX file. Figure A.1 shows the result of applying *swap_bibtex* to a particular BibTeX entry. In this presentation we assume that every

```

@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 A.1: Example application of the transformation *swap_bibtex* from the entry on the left to the entry on the right.

attribute is followed by a comma.⁷ The program *title_first* copies the title first and then all the other attributes, therefore obtaining the desired attribute reordering.

```
title_first = combine(title_only, all_but_title).
```

The program *swap_entry* performs the operation of figure A.1 for a single BibTeX entry.

```
swap_entry = split(copy_header, title_first,
                  copy('}'), copy_spaces).
```

Finally, the program *swap_bibtex* applies the transformation to all the entries in the file.

```
swap_bibtex = iterate(swap_entry).
```

A.5.2 Aligning titles in a misaligned BibTeX file

Assume that by mistake the titles of a BibTeX file have been misaligned: the title of the first entry now appears in the second one, the second title appears in the third entry and so on. The function *align_bibtex*, given such a misaligned file, moves the title of each entry $i + 1$ to the correct entry i , and since the last entry would now not have a title, it gets deleted. This program requires the use of the chained sum operator in order to process two BibTeX entries at a time (first and second, second and third, and so on). The program *header_only_entry*, given a single BibTeX entry, copies only the header and deletes the rest of the entry.

```
header_only_entry = split(copy_header, del_attrs,
                          del('}'), del_spaces).
```

The program *title_only_entry*, given a single BibTeX entry, copies only the title and deletes the rest of the entry, while the program *all_but_title_entry*, deletes the header, the title, and copies the rest of the entry.

```
title_only_entry = split(del_header, title_only,
                          del('}'), del_spaces),
all_but_title_entry = split(del_header, all_but_title,
                             copy('}'), copy_spaces).
```

The program *delete_entry* deletes a single BibTeX entry.

```
delete_entry = split(del_header, del_attrs,
                    del('}'), del_spaces).
```

The program *move_title*, given two BibTeX entries outputs the first entry where the title has been replaced with the title of the second entry: it first copies the header of the first entry, then the title of the second entry, and then the remaining attributes of the first entry.

```
move_title = combine(
  split(header_only_entry, title_only_entry),
  split(all_but_title_entry, delete_entry)).
```

⁷The actual program used for the experiments is able to deal with missing commas.

Finally, the program *align_bibtex* applies the transformation to all the entries in the file. We omit the regular expression describing the format of an entry in the chain operator since DReX can infer it automatically.

```
align_bibtex = chain(move_title).
```