DReX: A Declarative Language for Efficiently Computable Regular String Transformations

Anonymized for Submission

Abstract
We present DReX, a declarative language that can express all regular string-to-string transformations, and yet can be evaluated efficiently. 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.

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 libraries such as sed, AWK, and Perl have been used to query and reformat text files for many years. Since these libraries are Turing complete they can express very complex transformations, however this comes at the cost of not supporting any algorithmic analysis. To address this issue, restricted languages have been proposed in the context of verification of string sanitizers [25] and string coders [12], and for analysis and optimization of list-manipulating programs [14]. 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 analysis 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 or delete 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 [8], it has decidable equivalence [20], and has several equivalent characterizations, such as one-way transducers with a finite set of write-only registers [1], two-way transducers [15] and monadic second-order definable graph transformations [9]. Recently Alur et al proposed a set of combinators that captures the class of regular string transformations [3]. 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 that satisfies the predicate \(\varphi\) to a string that is a function of such a character. 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 \(f\ll g\rr\) that first tries to apply \(f\) to the input and if it fails it applies \(g\); (e) chain\((f,R)\) that unambiguously splits the string into multiple parts \(\sigma_1 \ldots \sigma_n\) each belonging to the language described by the regular expression \(R\), applies \(f\) to every two splits \(\sigma_i \sigma_{i+1}\), 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 operators (split sum, conditionals, iteration, etc.) and regular expressions (concatenation, union, Kleene-h, etc.) one approach is to construct a similar automaton model for evaluating DReX programs. This is not simple because of various reasons, such as: (a) the conditional operator, \(f\ll g\rr\), 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 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 \( g \) 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. 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 \( \text{split} \) 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 \( \sigma \), in time polynomial in the size of \( f \) and linear in the length of \( \sigma \). Intuitively, we construct a machine for each sub-program which reads the input in a single left-to-right pass. The machines keep track of potential parse trees of \( \sigma \) 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 \( \sigma \). 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, while processing \( f \text{ else } g \), the machine can simply echo the results of \( g \), because by the disjointedness of domains requirement, \( f \) and \( g \) cannot simultaneously emit results.

On the other hand, the dynamic programming algorithm can handle all DReX programs, but is limited by its cubic complexity in the size 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 an XML document, shuffle of chunks of a string, reverse of a dictionary, and reorder of tags in BHT files. The evaluation algorithm for consistent DReX scales to very large inputs (less than for 5 seconds for 100000 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 5000 characters) and therefore has limited applicability. Finally, the consistency-checking algorithm is very fast in practice (less 0.4 sec for programs of size >1500), 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, an equi-expressive fragment of DReX that admits efficient evaluation (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 2);
3. a dynamic programming algorithm for evaluating DReX programs that are not consistent that has cubic complexity in the size of the input (section 4);
4. a proof that adding a composition operator to DReX causes the evaluation problem to become \text{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 an input character \( a \in \Sigma \), and an output string \( d \in \Gamma^* \), the function \( a \mapsto d : \Sigma^* \rightarrow \Gamma^* \) maps the single-character input string \( \sigma \mapsto a \) to the output \( d \), and is undefined for all other inputs:

\[
[a \mapsto d](\sigma) = \begin{cases} d & \text{if } \sigma = a, \\ \bot & \text{otherwise}. \end{cases}
\]

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

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

\[
\text{split}(f_1, f_2)(\sigma) = [f_1](\sigma_1) [f_2](\sigma_2),
\]

\[
\text{left-split}(f_1, f_2)(\sigma) = [f_2](\sigma_2) [f_1](\sigma_1).
\]

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 DReX programs \( f_1 \) and \( f_2 \), \( f_1 \text{ else } f_2 \) first tries to apply \( f_1 \), and if this fails, applies \( f_2 \):

\[
[f_1 \text{ else } f_2](\sigma) = \begin{cases} [f_1](\sigma) & \text{if } [f_1](\sigma) \neq \bot, \\ [f_2](\sigma) & \text{otherwise}. \end{cases}
\]

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

Similarly, if both \( [f_1](\sigma) \) and \( [f_2](\sigma) \) are defined, then

\[
\text{combine}(f_1, f_2)(\sigma) = [f_1](\sigma) [f_2](\sigma).
\]

If either function is undefined for the input \( \sigma \), \( \text{combine}(f_1, f_2) \) is undefined as well. This combinator can be used to make multiple passes over the input string, and a typical example would be to copy a string: \( \sigma \) transformed into \( \sigma \). In terms of the input domain, the operator \( \text{combine} \) is the counterpart of intersection in regular languages, and is necessary for expressive completeness in our case because of the non-commutativity of string concatenation.

If \( f \) is a DReX program, and the input string \( \sigma \) can be uniquely split into substrings \( \sigma = \sigma_1 \sigma_2 \ldots \sigma_n \), with \( n \geq 0 \), and such that \( \text{split}(\sigma_i) \neq \bot \), for each \( i \), then

\[
\text{iterate}(f)(\sigma) = [f](\sigma_1) [f](\sigma_2) \ldots [f](\sigma_n),
\]

\[
\text{left-iterate}(f)(\sigma) = [f](\sigma_n) [f](\sigma_{n-1}) \ldots [f](\sigma_1).
\]

Otherwise (if the input \( \sigma \) cannot be split, or if multiple viable splits exist), then both iterated sums are undefined. This is the counterpart of Kleene*-\( \star \) 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 \( [R] = L \), and \( f \) be a DReX program. Given an input \( \sigma \), if there is a unique split \( \sigma = \sigma_1 \sigma_2 \ldots \sigma_n \), such that \( \sigma_i \in L \) for
Another example is the set of all integers with the notational convention that
\( J \) operate on individual characters in isolation, and cannot relate functions, i.e. \( \sigma \) transformations, i.e. \( x \to \phi \) each that (\( \Sigma \) applications.

2.2 Character sorts and predicates

For notational convenience, we treat \( \sigma \downarrow = \downarrow \sigma = \downarrow \), and so if \( \{ f \}(\sigma, \sigma_{i+1}) \) is undefined for any \( i \), both functions are undefined. Furthermore, if a unique split of the input string \( \sigma \) does not exist, both the chained and left-chained sums are undefined.

The final operator is function composition. If \( f_1 \) and \( f_2 \) are DReX programs such that \( [f_1] : \Sigma^* \to \Gamma^*_1 \), and \( [f_2] : \Gamma^* \to \Lambda^*_2 \), are partial functions, \( \text{compose}(f_1, f_2) \) is defined as

\[
\text{compose}(f_1, f_2)(\sigma) = [f_2](\{ f_1 \}(\sigma)),
\]

with the notational convention that \( \{ f_1 \}(\bot) = \bot \).

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 \( \Sigma \) and output alphabet \( \Gamma \), every regular string transformation \( f : \Sigma^* \to \Gamma^*_1 \) 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 \( a \to d \) we described in the previous subsection, which maps the input \( \sigma = a \) to the output \( d \). For large, and potentially infinite 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 [25], which has proved to be useful in several practical applications.

Let \( \Sigma, \Gamma, \ldots \) be a collection of character sorts. For each character sort \( \Sigma \), we pick (a possibly infinite) collection of predicates \( P_\Sigma \) such that (a) \( P_\Sigma \) is closed under the standard boolean operations: for each \( \varphi, \psi \in P_\Sigma \), \( \land \varphi \land \psi \in P_\Sigma \), 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 \( \mathbb{Z} \), with \( P_\mathbb{Z} = \{ \text{odd}(x), \text{even}(x), \text{true}, \text{false} \} \). We will write \( \cup \) for the set of all Unicode characters, with the various character properties \( P_\cup = \{ \text{uppercase}(x), \text{digit}(x), \ldots \} \).

If \( \varphi \in P_\Sigma \) and \( d = \{ d_1, d_2, \ldots, d_k \} \) is a list of character transformations, i.e. \( d_i : \Sigma \to \Gamma \), then \( \varphi \mapsto d \) is a basic transformation which maps every single-character string \( \sigma \) which satisfies \( \varphi(x) \) to the output string \( d_1(\sigma)d_2(\sigma) \ldots d_k(\sigma) \), 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. The function \( x \geq 0 \mapsto x - 1 \) is defined for all lists containing a single non-negative integer, and subtracts one from them. Given an input digit \( x \in [0-9] \), the function \( x \in [1-9] \mapsto x - 2 \) 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 a transformation 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 [11].

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 their ability to express expensive automata operations such as intersection and complement.

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, 24] — 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. \( \bot \) and \( \epsilon \) are CUREs. The language associated with \( \bot \) is the empty set \( [\bot] = \emptyset \), and the language associated with \( \epsilon \) is the singleton \( [\epsilon] = \{ \epsilon \} \).
2. For each satisfiable predicate \( \varphi \in P_\Sigma \), \( \varphi \) is a CURE. The language \( [\varphi] \) associated with the CURE \( \varphi \) 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 \( L_1 = [R_1] \) and \( L_2 = [R_2] \) are disjoint, then \( R_1 \cup R_2 \) is also a CURE, and \( [R_1 \cup R_2] = [R_1] \cup [R_2] \).
4. Given a pair of non-empty CUREs \( R_1 \) and \( R_2 \), we say that they are unambiguously concatenatable, if for each string \( \sigma \in \Sigma^* \), there is at most one split \( \sigma = \sigma_1\sigma_2 \) such that \( \sigma_1 \in [R_1] \) and \( \sigma_2 \in [R_2] \). If \( R_1 \) and \( R_2 \) are unambiguously concatenatable, then \( R_1 \cdot R_2 \) is also a CURE, and \([R_1 \cdot R_2] = [R_1]\cdot[R_2]\).
5. A non-empty CURE \( R \) is unambiguously iterable if for every string \( \sigma \), there is at most one split \( \sigma = \sigma_1\sigma_2\ldots\sigma_n \) into substrings such that \( \sigma_i \in [R] \) for each \( i \). If \( R \) is unambiguously iterable, then \( R^* \) also is a CURE, and \([R^*] = [R] \cdot [R]^* \).

For example, the regular expressions \( p \) and \( (\neg p)^* \) are unambiguously concatenatable: any string \( \sigma \) matching \( p \cdot (\neg p)^* \) has to be split after the first character. On the other hand, \( \Sigma^* \) is not unambiguously concatenatable 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 \( a \), \( a \), or \( aa \). The regular expression \( \Sigma^* \) is unambiguous — there is only one way to split each string \( \sigma \) such that each substring is in \( \Sigma \) — but \( (\Sigma^*)^* \) is not unambiguous.

We call two CUREs \( R_1 \) and \( R_2 \) equivalent, and write \( R_1 \equiv R_2 \), if \([R_1] = [R_2] \).

2.3.2 Consistency rules

A consistent DReX program is one that satisfies the following rules. 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 of strings \( \sigma \) such that \([f](\sigma)\) is defined. In the following rules, we assign each DReX program a domain type, which is a representation of its domain as a CURE.

1. All basic functions \( \text{bottom}, \epsilon \mapsto d, \) and \( \varphi \mapsto d \) (where \( \varphi \) is satisfiable) are consistent. Their domain types are \( \bot, \epsilon \) and \( \varphi \) respectively.
2. If \( f_1 \) and \( f_2 \) are both consistent and have unambiguously concatenate domain types \( R_1 \) and \( R_2 \) respectively, then \( \text{split}(f_1, f_2) \) and \( \text{left-split}(f_1, f_2) \) are also both consistent and have the domain \( R_1 \cup R_2 \).

3. If \( f \) is consistent has domain type \( R \) and \( R \) is unambiguously iterable, then \( \text{iterate}(f) \) and \( \text{left-iterate}(f) \) are both consistent, with domain \( R^\ast \).

4. If \( f_1 \) and \( f_2 \) are consistent with disjoint domain types \( R_1 \) and \( R_2 \) respectively, then \( f_1 \text{ else } f_2 \) is also consistent with the domain \( R_1 \cap R_2 \).

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 \( \text{chain}(f, R) \) and \( \text{left-chain}(f, R) \) are both consistent, and have the domain \( R \cdot R \cdot R^\ast \).

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 \( \text{combine}(f_1, f_2) \) is also consistent. Depending on the syntactic structure of the CUREs \( R_1 \) and \( R_2 \), the domain type of \( \text{combine}(f_1, f_2) \) is defined as follows:

\( \text{combine}(f_1, f_2) \) is consistent, with domain type \( R \cdot R \cdot R^\ast \).

(c) Otherwise, the domain type is \( \{ \emptyset \} \).

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 \( \Sigma \), and output alphabet \( \Gamma \), every regular function \( f : \Sigma^* \to \Gamma^* \) can be expressed by a consistent DReX program.

The consistency and domain computation rules are syntax-directed, and straightforward to implement directly. We need the following basic operations over 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 [23] 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: take the product of the corresponding (\( \epsilon \)-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 \( A_R \). Thus, if the input alphabet \( \Sigma \) 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 \( \Sigma \) is finite, then from [24], we have that this can be checked in time \( \mathcal{O}(\text{poly}(|R_1|, |R_2|, |\Sigma|)) \). Otherwise, if automata are expressed using the symbolic notation of section 2.2, they can be translated into symbolic automata, and the equivalence of symbolic automata is decidable in polynomial time in the size of \( R_1 \) and \( R_2 \) and exponential\(^2\) in the number of predicates appearing in \( R_1 \) and \( R_2 \).

We therefore have:

**Theorem 3.** Given a DReX program \( f \) over an input alphabet \( \Sigma \), checking whether \( f \) is consistent is decidable. Furthermore, if the input alphabet \( \Sigma \) 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 or the untyped class.

### 2.4 Examples of consistent DReX programs

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

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

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

Consider the function \( \text{shuffle} \), which maps an input string \( \sigma \in \{a, b\}^\ast \) of the form

\[ \sigma = a^{n_1}ba^{n_2}b \ldots a^{n_k}b, \]

with \( k \geq 2 \) to the output

\[ \gamma = a^{n_2}ba^{n_1}b a^{n_3}ba^{n_2}b \ldots a^{n_k}ba^{n_{k-1}}b. \]

The function can be expressed using the chained sum operator as

\[ \text{shuffle} = \text{let } id_{a\cdot b} = \text{iterate}(x = a \mapsto x) \text{ in } \text{let } id_{a\cdot b} = \text{split}(id_{a\cdot b}, x = b \mapsto b) \text{ in } \text{let } f = \text{left-split}(id_{a\cdot b}, id_{a\cdot b}) \text{ in } \text{chain}(f, a^\ast \cdot b). \]

The sub-expression \( id_{a\cdot b} \) is the identity function restricted to strings of the form \( a^{n} \cdot b \), and so \( f \) looks at pairs of such patches and places the second patch before the first. Note also the use of let-expressions: our implementation supports them, but we do not include them in the core syntax of DReX since the evaluation tool desugars them before processing.

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

---

\(^2\) The algorithm proposed in [24] 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 [13] 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.
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.

3.1 Function evaluators

Given a consistent DReX program \( f \), the idea is to construct an evaluator \( T \) which computes the associated function \( f \). 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 \( \text{split}(f, g) \).

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

Assume that \( f \) and \( g \) are consistent, and have unambiguously concatenatable 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 \( \text{Start} \) to \( T_g \) to start processing the suffix. Consider the situation in figure 3.1, where \( f \) is defined for the prefixes \( \sigma_1 \sigma_2 \ldots \sigma_i \) and \( \sigma_i \sigma_2 \ldots \sigma_j \). The input to the sub-evaluator \( T_g \) is then the sequence \( (\sigma_1, 1), (\sigma_2, 2), \ldots, (\sigma_i, i), (\text{Start}, i), (\sigma_{i+1}, i+1), \ldots, (\sigma_j, j), (\text{Start}, j), \ldots, (\sigma_n, n) \).

For each signal \((\text{Start}, i)\) occurring in the input string, we call the subsequent sequence of characters \( \sigma_{i+1} \sigma_{i+2} \ldots \text{the thread} \) beginning at index \( i \). Note that each thread corresponds to a potential parse tree of \( \sigma \), and that \( T_g \) may be processing multiple such threads simultaneously. The main challenge is to ensure that the number of active threads in \( T_g \) is bound by \( O(|g|) \), and is independent of the length of the input string. After reading \( \sigma_n \), \( T_g \) reports a result to \( T \). To uniquely identify the thread \( j \) returning the result, the result signal \((\text{Result}, j, \gamma_j)\) is annotated with the index \( j \) at which the corresponding \( \text{Start} \) 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 \((\text{Result}, 0, \gamma)\) initially obtained from \( f \) at position \( j \), and itself emits the result \((\text{Result}, 0, \gamma | \gamma_j)\). To do this, it maintains a set \( \theta \) (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 \( T_g \) reported a result and \( T_g \) was started, and \( \gamma f \) was the result reported by \( T_f \). In order to prevent this set \( \theta \) from becoming too large, \( T_f \) emits kill signals. Say that, at index \( k \), \( T_f \) discovers that for every possible suffix \( \tau \in \Sigma^* \), \( g \) will be undefined for the string \( \sigma_{i+1} \sigma_{i+2} \ldots \sigma_{i+k} \), and so the thread \((\text{Start}, i)\) of \( T_f \) initiated at the input index \( i \) can never return a result. It then emits \((\text{Kill}, i)\) to signal to \( T \) that the relevant entries in the set \( \theta \) can be deleted.

The input alphabet to each evaluator is therefore \( \Sigma = (\Sigma \cup \{\text{Start}\}) \times \mathbb{N} \) and the output alphabet is \( \text{Out} = (\{\text{Result}\} \times \mathbb{N} \times \Gamma^*) \cup (\{\text{Kill}\} \times \mathbb{N}) \), where \( \Sigma \) is the input and \( \Gamma \) 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 \( m \), there is at most one thread for which \( f \) is defined. Thus, for example, \( T \) never can see two consecutive \text{Start} signals. In return, we make the following guarantees:

Correctness of results. After reading each input signal \((\sigma_j, i)\) in \( in \), we report the result \((\text{Result}, i, \gamma)\) exactly for that thread \((\text{Start}, i)\) such that \( [f](\sigma_1 \sigma_2 \ldots \sigma_i) = \gamma \), if it exists.

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

If an evaluator \( T \) satisfies these requirements for \( f \), then we say that the evaluator computes \( f \). On the input \((\text{Start}, 0), (\sigma_1, 1), (\sigma_2, 2), \ldots, (\sigma_n, n)\), the evaluator outputs a result \( \gamma \) in exactly those cases when \( f \) is defined, and in that case, \( f(\sigma) = \gamma \).

In the rest of this section, we construct \( T \) by structural induction on the consistent DReX program \( f \). We also prove that the evaluator \( T \) processes each input signal in time \( O(|\{f\}|) \). Omitted proofs and the construction for the case of the chained sum can be found in the full version of this paper, which has been supplied as additional material with this paper submission.

3.2 Basic evaluators

The simplest case is when \( f = \bot \). The evaluator \( T_{\bot} \) is defined by the following rules:

1. On input \((\text{Start}, i)\), respond with \((\text{Kill}, i)\).

2. On input \((a, i)\), for \( a \in \Sigma \), do nothing.

Next, we consider the evaluator \( T_{e \rightarrow d} \), for the case when \( f = e \rightarrow d \), 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 \( \theta \subseteq \mathbb{N} \) of currently active threads, which are to be killed on reading the next input symbol. The set \( \theta \) is initialized to \( \emptyset \).

1. On input \((\text{Start}, i)\), respond with \((\text{Result}, i, d)\). Update \( \theta := \theta \cup \{i\} \).

2. On input \((a, i)\), for \( a \in \Sigma \), respond with \((\text{Kill}, j)\), for each thread start index \( j \) in \( \theta \). Update \( \theta := \emptyset \).

Observe that by the condition of input validity, we can never observe two consecutive start signals in the input stream. Therefore, \( |\theta| \leq 1 \), and the response time of \( T_{e \rightarrow d} \) to each input signal is bounded by a constant.

2 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 length of the string.
The simplest non-trivial evaluator is for \( \varphi \mapsto d \) for some character predicate \( \varphi \) and \( d : (\Sigma \rightarrow \Gamma)^* \). The evaluator \( T_{\varphi \rightarrow d} \) 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 \( \varphi \).

1. On input \((Start, i)\), update \( th := th \cup \{i\} \).
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 \( \varphi \), 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 \( \varphi \), then for each thread \( j \in th \), emit \((Kill, j)\). Update \( th := \emptyset \) and \( th' := \emptyset \).

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

### 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 behaviour:

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 \((Result, i, \gamma_f \land \gamma_g)\).
4. On receiving the kill signals \((Kill, i)\) from \( T_f \) and \( T_g \) (by the consistency rules, necessarily simultaneously), emit the kill signal \((Kill, i)\).

The evaluator \( T_{\text{combine}(f,g)} \) is similar, except

### 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.

![Figure 3.2](image-url)

**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 \((i_0, i, \gamma)\) in the set \( th \). Thus, after obtaining the response from \( T_g \) at index \( n \), \( T_{\text{iterate}(f)} \) updates \( th := th \cup \{i, n, \gamma \gamma_2 \gamma_3 \} \).

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 \((Start, i_0)\), or an input signal \((a, i)\), this is passed to \( T_f \). Consider a sequence of input signals \( \sigma \), as shown in figure 3.2. After reading each input symbol, say \((\sigma_n, n)\), \( T_f \) may report that \( f \) is defined for a suffix of the input stream. 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 \((i_0, j, \gamma)\) 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 \( \gamma \) 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 \((Result, i, \epsilon)\).
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 \((Result, j, \gamma_f)\), then, find the corresponding entry \((j_0, j, \gamma) \in th\), for some values of \( j_0 \) and \( \gamma \). Assert (by the invariant that \( th \) records the active threads of \( T_f \)) that this entry exists, and is unique.
      i. Update \( th := th \cup \{j_0, i, \gamma_f\} \).
      ii. Send the signal \((Start, i)\) to \( T_f \). Confirm that \( T_f \) does not respond with a result \((Result, i, 0, \gamma_f)\), for that would violate the consistency requirements.
      iii. Respond with the result \((Result, j_0, \gamma_f)\).
   (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 \( \gamma \). By the consistency requirements, \( kill-ring \) is asserted to be a singleton set.
      ii. Update \( th := th \ \backslash \ \text{kill-ring} \).
      iii. For every entry \((j_0, j, \gamma) \in kill-ring\) if there is no entry of the form \((j_0, j', \gamma') \in th\), 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 \( O(|f|) \). The response time of \( T_{\text{iterate}(f)} \) to each input signal is therefore \( O(|f|) + t_f \), where \( t_f \) is the response time of \( T_f \).

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
We now briefly present the dynamic programming algorithm to evaluate unrestricted DReX programs that can also contain the composition operators. As we will show in subsection 4.2, the problem of evaluating DReX is allowed, requires time exponential in the program size. In fact, the rule for computing \( \text{iterate}(f) \) on \( \sigma = \sigma[i,j] \) is defined if there is a unique way to split the string \( \sigma' \) into multiple chunks so that \( f \) is defined on each chunk, i.e., if \( \text{count}(\text{iterate}(f), \sigma, i, j) = 1 \). If this is the case, then we know that there is a unique value \( k \), such that \( i < k < j \), such that \( \gamma_{\text{pre}} = \text{OUT}(\text{iterate}(f), \sigma, i, k) \) and \( \gamma_{\text{post}} = \text{OUT}(f, \sigma, k, j) \) are defined, and \( \gamma_{\text{pre}} \gamma_{\text{post}} \) is the output of \( \text{iterate}(f) \) 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 \( \text{count}(\text{iterate}(f), \sigma, i, k) \) can be computed in at most \( |\sigma| \) steps by counting the values of \( k \) for which \( \text{count}(\text{iterate}(f), \sigma, i, k) > 0 \) and \( \text{out}(f, \sigma, k, j) \) is defined. The rule for computing \( \text{out}(f, \sigma, k, j) \) is what causes an exponential blow-up in evaluation time. To compute the output of \( \text{compose}(f_1, f_2, \sigma, i, j) \), we first need to compute the output \( \tau = \text{OUT}(f_1, \sigma, i, j) \) and then the output \( \text{OUT}(f_2, \tau, k, j) \) of \( f_1 \) on \( \sigma[i,j] \). As we will show, when using the composition operator, the size of the output may grow exponentially. Since computing each entry of the table \( \text{OUT}(f_2, \tau) \) will require at most \( r \) steps, and \( r \) has exponential size, the resulting complexity will also be exponential.

To analyze the complexity of this algorithm we need the following bound on the size of the computed output.

**Lemma 4 (Output size).** Given a string \( \sigma \in \Sigma^* \), and a DReX program \( f \), the output \( [f](\sigma) \) when defined has size \( O(|f|^{d+1}|\sigma|) \), where \( d \) is the number of composition operators in \( f \).

We now state the complexity of the dynamic programming routine to evaluate DReX programs:

**Theorem 5 (DReX evaluation by DP).** Given a string \( \sigma \in \Sigma^* \), and a DReX program \( f \), the dynamic programming algorithm computes the output \( [f](\sigma) \) in time \( O(|f|^{2d+4}+|\sigma|^{2d+3}) \) where \( d \) is the number of composition operators in \( f \). If \( d = 0 \), the algorithm has complexity \( O(|f|^{3}) \).

**Proof Sketch.** For a particular string \( \sigma \) and if the program \( f \) that does not contain any composition operators \( (d = 0) \), computing each entry of the tables \( \text{OUT}, \text{COUNT}, \) and \( \text{BEL} \) takes time \( O(|\sigma|) \) and since there are \( |f||\sigma|^2 \) entries the algorithm has complexity \( O(|f|^{3}) \). In the presence of composition operators which can produce intermediate results, for each intermediate string \( \tau \), a new table of size \( \tau^2 \) must be created. By Lemma 4 we know that an intermediate result has size at most \( O(f^{d+1} |\sigma|) \), hence the complexity is exponential in \( d \).

## 4.2 DReX evaluation with composition is PSPACE-complete

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. We observe that, since the output computed by a program

\[ \sigma[i,j] \]

representing the output of \( f \) on the substring \( \sigma[i,j] \) of \( \sigma \), 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 \( \text{chain}(f, P) \) we compute the function \( \text{BEL}(R, \sigma, i, j) \) which checks whether a substring \( \sigma[i,j] \) belongs to the language \( [R] \). Every function \( \text{OUT}, \text{COUNT}, \) and \( \text{BEL} \) is represented by a table and for each string \( \sigma \) and each sub-program \( q \) of \( f \) each table will have \( O(|\sigma|^2) \) entries, corresponding to the substrings of \( \sigma \). The final output of the algorithm is \( \text{OUT}(f, \sigma, 1, |\sigma| + 1) \).

We briefly illustrate the intuition of the algorithm by showing how the entries are computed for the iteration and composition. The value \( \text{OUT}(\text{iterate}(f), \sigma, i, j) \) corresponding to the output of \( \text{iterate}(f) \) on \( \sigma' = \sigma[i,j] \) is defined if there is a unique way to split the string \( \sigma' \) into multiple chunks so that \( f \) is defined on each chunk, i.e., if \( \text{count}(\text{iterate}(f), \sigma, i, j) = 1 \). If this is the case, then we know that there is a unique value \( k \), such that \( i < k < j \), such that \( \gamma_{\text{pre}} = \text{OUT}(\text{iterate}(f), \sigma, i, k) \) and \( \gamma_{\text{post}} = \text{OUT}(f, \sigma, k, j) \) are defined, and \( \gamma_{\text{pre}} \gamma_{\text{post}} \) is the output of \( \text{iterate}(f) \) on \( \sigma[i,j] \).

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 program size. In fact, as we will show in subsection 4.2, the problem of evaluating DReX programs with composition is PSPACE-complete. Finally, we argue that even for composition-free untyped DReX programs, there is no evaluation algorithm whose 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 briefly present the dynamic programming algorithm to evaluate unrestricted DReX programs that can also contain the composition operator. The algorithm mimics the semantics of DReX by computing the following functions (represented as dynamic programming tables). Given a program \( f \), a string \( \sigma \), and for any two numbers \( i, j \) the algorithm computes the function \( \text{OUT}(f, \sigma, i, j) \)
has at most exponentially many characters (lemma 4), the index of each character is only polynomially many bits long. Concretely, the following procedures can be implemented in \textsc{pspace}: for every combinator \( f \), given the implicit representation of a string \( \sigma \): (a) check whether \( f \) is defined on \( \sigma \); (b) compute the length of the output of \( f \) on \( \sigma \); and (c) given an index \( i \) compute the \( i \)-th character in the output of \( f \) on \( \sigma \). The most interesting case of our construction is when \( f = \text{iterate}(g) \). To check whether \( f \) is defined on a string \( \sigma \) we need to make sure that there is exactly one way to split \( \sigma \) so that \( g \) is defined on each split. Consider each position in the string as a vertex in a graph, with an edge between two vertices iff \( g \) is defined on the substring between them. The program \( f \) is defined on \( \sigma \) iff there is a unique path from the initial node to the final node (i.e. a unique split of the input string) in this implicitly represented graph of exponential size, and this problem can be solved can be solved in \textsc{pspace}.

Finally, we show that when we allow the use of composition operators polynomial space is required, and the problem of evaluating a \textsc{drex} program is \textsc{pspace}-complete.

\textbf{Theorem 6 (\textsc{pspace}-complete).} The following problems are \textsc{pspace}-complete:

1. given a program \( f \) in \textsc{drex} check whether \( [f](\epsilon) \) is defined;
2. given a program \( f \) in \textsc{drex} and a string \( \sigma \in \Sigma^* \) check whether \( [f](\sigma) \) is defined;
3. given a program \( f \) in \textsc{drex}, a string \( \sigma \in \Sigma^* \), and a string \( \tau \in \Gamma^* \), check whether \( [f](\sigma) = \tau \);

\textbf{Proof Sketch.} The problems above can be reduced to each other using the composition operator. To show that the first problem is \textsc{pspace}-hard we can reduce from the validity problem for quantified Boolean formulas (QBF). Intuitively given a QBF \( \Phi = \forall x_1 \exists x_2 \ldots \exists x_n \varphi(x_1, \ldots, x_n) \) we can construct a \textsc{drex} program \( f_\Phi \) such that \([f](\epsilon)\) is defined iff \( \Phi \) is valid. The program \( f_\Phi \) is the composition of three programs \( f_{01}, f_{3SAT}, \) and \( f_Q \) where:

- \( f_{01} \) takes as input \( \epsilon \) 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.
- \( f_{3SAT} \) 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 3SAT formula \( \varphi \) and \( F \) otherwise.
- \( f_Q \) takes as input the string over \( \{\{T, F\} \#\}^n \) and checks whether such a sequence of satisfying assignments is valid for the quantified formula \( \Phi \). If it is valid it outputs \( \epsilon \) and otherwise it is undefined.

\section{Single-pass algorithms for unrestricted \textsc{drex}}

In the proof of theorem 5 we showed that in the absence of compositions the dynamic programming algorithm has complexity \( \Theta(f(|\sigma|)) \) where \( \sigma \) 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.

\textsc{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 that can model \textsc{drex} programs. Unfortunately as we discussed in section 2, \textsc{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 [22], 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 \textsc{drex} programs with only a single pass over the input string:

\textbf{Theorem 7 (A left-to-right algorithm).} Given a string \( \sigma \in \Sigma^* \), and a \textsc{drex} program \( f \), there exists an algorithm to compute the output \( [f](\sigma) \) in time linear in the length of the input string \( \sigma \), and a tower of exponentials with height \( O(|f|) \).

The idea is to compile the \textsc{drex} program into an equivalent streaming string transducer (SST) [1, 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 can be evaluated on a string in linear time. 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.

\section{Evaluation}

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

- the evaluation algorithm for consistent \textsc{drex} scales to inputs with more than 100000 characters (subsection 5.2.1), and
- the dynamic programming algorithm for general \textsc{drex} programs does not scale for input with more than 3000 characters (subsection 5.2.2).

Finally, we remark on the subjective experience of expressing string transformations using \textsc{drex} (subsection 5.3).

\subsection{Implementation details}

Our prototype implementation of \textsc{drex} has been supplied with this paper submission. The implementation is written in Java, and uses the recently released Java SE 8. We used the symbolic automata library SVPAlib from [10] 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 \([a-z, k-Z, 0-9]\)).

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.

The dynamic programming (DP) algorithm for the extended version of \textsc{drex} (theorem 5) is implemented lazily. Each entry \( \text{out}(f, \sigma, i, j) \) 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 1000. We also optimize the DP algorithm to take advantage of the consistency-checking routine. In particular, for consistent programs the algorithm does not need to check whether there is more than one way to match a particular string.

\begin{table}
\centering
\begin{tabular}{|l|c|c|}
\hline
Program Name & Size & CC (ms) \\
\hline
\texttt{delete}\_\texttt{comm} & 28 & 12 \\
\texttt{insert}\_\texttt{quotes} & 28 & 6 \\
\texttt{get}\_\texttt{tags} & 31 & 6 \\
\texttt{shuffle}\_\texttt{dic} & 20 & 6 \\
\texttt{reverse} & 5 & 1 \\
\texttt{bibtex}\_\texttt{swap} & 1455 & 366 \\
\hline
\end{tabular}
\caption{Evaluated programs together with sizes and time taken to check consistency.}
\end{table}
Figure 5.2: Evaluation time for dynamic programming algorithm.

Figure 5.3: Evaluation time for dynamic programming algorithm. Note that the x-axis is in log-scale.

5.2 Benchmark programs

Table 5.1 shows the programs we considered in our evaluation together with their sizes and the running time of the consistency-checking algorithm. We can observe how for every program the consistency-checking algorithm terminates in less than 400 ms. The full description of these programs can be found in appendix A.

We evaluated the first five programs on randomly generated strings of size varying between 1 and 100000, and evaluated the more complex function bibTeX_swap on actual BibTeX files of size varying between 1 and 100000. We set a timeout of 60 seconds for each operation.

5.2.1 Single-pass algorithm for consistent DReX

Figure 5.2 shows the running time for the algorithm presented in section 3. The figure illustrates how the running time of the algorithm depends linearly in the size of the input. For every input up to 100000 characters every program takes less than 5 seconds to terminate. Also observe that the evaluation algorithm can successfully handle reasonably large programs (bibTeX_swap, with an AST of size 1455 nodes).

5.2.2 Dynamic programming for unrestricted DReX

Figure 5.3 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. For each program we see how the running time polynomially depends on the length of the input. Every program timed out for inputs of length greater than 70000. Given the complexity of the algorithm this is not particularly surprising. Interestingly the size of the program is not necessarily an indicator of the running time of this algorithm. This is due to the fact that the implementation uses some optimizations that may depend on the shape of the program: for example in the case of programs that only defined on strings of a fixed length k, the 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

We report our experience on programming in DReX. The implementation of DReX is only at an early stage however we were able to easily program many interesting and non-trivial string-to-string transformations without having to worry about efficiency. The main restrictions imposed in this paper 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 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) → x), which copies all spaces, with itself. In this case, the type-checker warned us that this concatenation was ambiguous on the input string “\n”. This was clearly a bug in our program.

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 [15, 17], streaming string transducers [1], transducers with origin information [6], and MSO definable string transformations [9]. Regular string transformations are also closed under composition [8], and enjoy decidable equivalence [20]. 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 mostly fall into two classes: string specific utilities such as sed, AWK, and Perl, and transducer-based languages [12, 14, 25].

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 or enjoy any algorithmic analysis. All transducer-based languages simply act as frontend to an underlying transducer model and their goal is typically that of reasoning about the implemented programs. BEK uses symbolic finite transducers [25] and it has been used to analyze string sanitization functions. BEX is a frontend for extended symbolic finite transducers [11, 12] and it has been used to prove the correctness string encoders and decoders such as Base64. Fast is based on symbolic tree transducers [14] 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. In particular 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 of 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. However, 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 adopted for this problem is that of identify an automaton model for which programs can be compiled into and that can process the string in a single left-to-right pass. However, all the existing tools that use this approach [12, 14, 25] 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 the programs that can be written in DReX and can be executed in a single left-to-right linear pass over the input. However, transforming DReX programs into an SST also causes a 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 [26]), 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. When XML document has bounded depth some of the transformations captured by such languages can be described in DReX. However, the goals of these languages are orthogonal to those of DReX. In particular DReX focuses on strings and on providing a well-defined fragment (regular) of 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 the problem of regular type-checking: given a program $f$ and two regular languages $I$ and $O$, is it the case that for every input $\sigma \in I$, $[p](\sigma) \in O$? 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 [19] 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?

Recently Mytkowicz et al. [21] 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 [18], and MSO-definable tree transformations [16]) as SSTs are to regular string transformations. Can we design a similar declarative language to express regular tree transformations?

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 and potentially infinite alphabets such as Unicode. We demonstrated that the evaluation problem for unrestricted DReX is PSPACE-complete, and so we considered a restrictive fragment, consistent DReX which permits a fast single-pass evaluation algorithm, and still retains expressive completeness. Experiments over representative string transformations such as BitToX file manipulations confirm that the evaluation algorithm for consistent DReX gracefully scales to process thousands of characters per second.

**References**


We also write

\begin{verbatim}
process_line ← del_comm_line else copy_line
last_line ← del_comm else copy_txt
process_last_line ← process_line else last_line
process_lines ← iterate(process_line)
del_comm ← split(process_lines, process_last_line)
\end{verbatim}

### 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 \( \sigma_1 \) is alphabetic and \( \sigma_2 \) does not contain any letter, outputs the string "\( \sigma_1 \)\( \sigma_2 \)"

\begin{verbatim}
copy_astr ← iterate-plus(copy([a-zA-Z]))
copy_nastr ← iterate-plus(copy(x \notin [a-zA-Z]))
add_qt ← ε \mapsto ["]
add_qts ← split(add_qt, copy_astr, add_qt)
add_qts_skip ← split(add_qts, copy_nastr)
\end{verbatim}

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_qts` inserts quotes around every alphabetic substring in the input.

\begin{verbatim}
start ← iterate(copy(x \notin [a-zA-Z]))
ending ← add_qts else ε \mapsto []
repeat_add ← iterate(add_qts_skip)
insert_quotes ← split(start, repeat_add, ending)
\end{verbatim}

### A.3 Extracting tags from a malformed XML file

This program extracts and concatenates all the substrings of the form \( < \sigma > \) where \( \sigma \) does not contain any character \( < \) or \( > \) (this is a generalization of a program shown in [25]). For simplicity we assume that the string does not contain occurrences of the substring \( <> \). The program `copy_match` copies any string of the form \( <> \) where \( \sigma \) does not contain any character \( < \) or \( > \).

\begin{verbatim}
copy_ntag ← iterate-plus(copy([<][])\])
copy_match ← split(copy(<), copy_ntag, copy(>')]
\end{verbatim}

The program `del_not_match` deletes any string that does not contain a substring of the form \( <> \) where \( s \) does not contain any character \( < \) or \( > \). The program `del_not_match` looks for the following pattern: \( s = \sigma_1 \sigma_2 \ldots <\sigma_i <\sigma_{i+1} \ldots \sigma_s \) (with \( i \geq 1 \)). Its sub-program `del_close_op` deletes all the string of the described form containing at least one open character (\'\( < \)\)'.

\begin{verbatim}
del_not_opn ← iterate(del(x \notin '<'))
del_not_ccls ← iterate(del(x \notin '>'))
del_opn_not_ccls ← split(del('<'), iterate(del(x \notin '>')))
del_close_op ← split(del_not_opn, del_opn_not_ccls)
del_not_match ← del_not_opn else del_close_op
\end{verbatim}

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 \( <> \).

\begin{verbatim}
find_match ← split(del_not_match, copy_match)
repeat_get_tags ← iterate(find_match)
get_tags ← split(repeat_get_tags, del_not_match)
\end{verbatim}
Figure A.1: Example application of the transformation bibtext\_swap from the entry on the left to the entry on the right.

### A.4 String shuffle

Given a string of the form $\sigma_1; \sigma_2; \ldots; \sigma_n$; the program \texttt{shuffle\_dic} outputs the string $\sigma_1; \sigma_3; \sigma_2; \ldots; \sigma_n; \sigma_{n-1}$ using the chained sum operator. The program \texttt{copy\_stretch} copies a stretch $\sigma_i$. The program \texttt{flip\_stretches} given a string $\sigma_i; \sigma_{i+1}; \ldots; \sigma_n$ outputs the swap $\sigma_{i+1}; \sigma_i$. Finally, \texttt{shuffle\_dic} implements such a transformation by left iterating the program \texttt{flip\_stretches} and implements the desired transformation.\footnote{Our implementation does not actually require to annotate the \texttt{chain}(f, R) operator with the unambiguous regular expression $R$ since it can be inferred using the consistency rules.}

\[
\begin{align*}
\text{copy\_not\_delim} & \leftarrow \text{iterate}(\text{copy}(x \neq ';')) \\
\text{copy\_stretch} & \leftarrow \text{split}(\text{copy\_not\_delim}, \text{copy}('')) \\
\text{flip\_stretches} & \leftarrow \text{left-split}(\text{copy\_stretch}, \text{copy\_stretch}) \\
\text{shuffle\_dic} & \leftarrow \text{chain}(\text{flip\_stretches}, (x \neq ';')^* (x = ';'))
\end{align*}
\]

### A.5 Reversing a dictionary

Given a dictionary of the form $\sigma_1; \sigma_2; \ldots; \sigma_n$; we want to output the reversed version $\sigma_n; \sigma_{n-1}; \ldots; \sigma_1$. The program \texttt{reverse} implements such a transformation by left iterating the program \texttt{copy\_stretch} from the shuffle example.

\[
\text{reverse} \leftarrow \text{left-iterate}(\text{copy\_stretch})
\]

### A.6 Reformattin Bib\TeX files

The following function defines a typical transformation a paper author may perform on Bib\TeX files. The program \texttt{bibtext\_swap} moves the title attribute to the top of each entry of an input Bib\TeX file. Figure A.1 shows the result of applying \texttt{bibtext\_swap} to a particular Bib\TeX entry. In this presentation we assume that every attribute is followed by a comma.\footnote{The actual program used for the experiments is able to deal with missing commas.}

We start by defining the program that processes each Bib\TeX entry. To do so we first need a few auxiliary functions, for copying and deleting alphabetic strings, spaces, and delimiters.

\[
\begin{align*}
\text{copy\_astr} & \leftarrow \text{iterate Plus}(\text{copy}[a-zA-Z]) \\
\text{copy\_num} & \leftarrow \text{iterate Plus}(\text{copy}[a-zA-Z0-9]) \\
\text{copy\_spaces} & \leftarrow \text{iterate}(\text{copy}([n,r,b,d])) \\
\text{del\_spaces} & \leftarrow \text{iterate}(\text{del}([n,r,b,d]))
\end{align*}
\]

The program \texttt{copy\_header} copies the header of an entry. In the example of figure A.1, it copies the string \texttt{\#book\{Galilei\}}, along with the following spaces.

\[
\begin{align*}
\text{copy\_header} & \leftarrow \text{split}(\text{copy}('@'), \text{copy\_astr}, \text{copy}(''), \\
& \quad \text{copy\_astr}, \text{copy}(''), \text{copy\_spaces})
\end{align*}
\]

The program \texttt{copy\_title} then copies the string title, while \texttt{copy\_non\_title} copies every attribute name different from title. We omit the full list of attributes for readability. Similarly we can define the corresponding delete operators.

\[
\begin{align*}
\text{copy\_title} & \leftarrow \text{copy}('title') \\
\text{copy\_non\_title} & \leftarrow \text{copy}(['author', 'year', 'place', '…']) \\
\text{del\_title} & \leftarrow \text{del}('title') \\
\text{del\_non\_title} & \leftarrow \text{del}(['author', 'year', 'place', '…'])
\end{align*}
\]

The program \texttt{copy\_att\_value} (\texttt{del\_att\_value}) copies (deletes) the value of an attribute along with the surrounding parentheses (i.e. \texttt{\{Elzevir\}}).

\[
\begin{align*}
\text{copy\_att\_value} & \leftarrow \text{split}(\text{copy\_spaces}, \text{copy}(''), \\
& \quad \text{copy\_num}, \text{copy\_par}, \text{copy\_astr}, \text{copy\_spaces}) \\
\text{del\_att\_value} & \leftarrow \text{split}(\text{del\_spaces}, \text{del}(''), \\
& \quad \text{del\_num}, \text{del\_par}, \text{del\_astr}, \text{del\_spaces})
\end{align*}
\]

The program \texttt{copy\_title\_att} (\texttt{del\_title\_att}) copies (deletes) a complete title attribute (i.e. \texttt{\{Two New Sciences\}}).

\[
\begin{align*}
\text{copy\_title\_att} & \leftarrow \text{split}(\text{copy\_title}, \text{copy\_att\_value}) \\
\text{del\_title\_att} & \leftarrow \text{split}(\text{del\_title}, \text{del\_att\_value})
\end{align*}
\]

Similarly, the program \texttt{copy\_non\_title\_att} (\texttt{del\_non\_title\_att}) copies (deletes) a complete non-title attribute.

\[
\begin{align*}
\text{copy\_non\_title\_att} & \leftarrow \text{split}(\text{copy\_num\_title}, \text{copy\_att\_value}) \\
\text{del\_non\_title\_att} & \leftarrow \text{split}(\text{del\_num\_title}, \text{del\_att\_value})
\end{align*}
\]

The program \texttt{title\_only}, given a list of attributes, copies the title and deletes all the non-title attributes.

\[
\text{title\_only} \leftarrow \text{iterate}(\text{copy\_title\_att} \text{del\_non\_title\_att})
\]

Similarly the program \texttt{all\_but\_title} reads all the attributes, deletes the title, and copies all the non-title attributes.

\[
\text{all\_but\_title} \leftarrow \text{iterate}(\text{del\_title\_att} \text{copy\_non\_title\_att})
\]

The program \texttt{title\_first} copies the title first and then all the other attributes, therefore obtaining the desired reordering.

\[
\text{title\_first} \leftarrow \text{combine}(\text{title\_only}, \text{all\_but\_title})
\]

The program \texttt{swap\_entry} performs the operation of figure A.1 for a single entry.

\[
\text{swap\_entry} \leftarrow \text{split}(\text{copy\_header}, \text{title\_first}, \text{copy}(''), \text{copy\_spaces})
\]

Finally, the program \texttt{bibtext\_swap} applies the transformation to all the entries in the file.

\[
\text{bibtext\_swap} \leftarrow \text{iterate}(\text{swap\_entry})
\]