Proactive Synthesis of Recursive Tree-to-String Functions from Examples*†

Mikaël Mayer[1], Jad Hamza[1], and Viktor Kunčak[1]

1 **EPFL IC IINFCOM LARA, INR 318, Station 14, CH-1015 Lausanne**
`firsname.lastname@epfl.ch`

Abstract

Synthesis from examples enables non-expert users to generate programs by specifying examples of their behavior. A domain-specific form of such synthesis has been recently deployed in a widely used spreadsheet software product. In this paper we contribute to foundations of such techniques and present a complete algorithm for synthesis of a class of recursive functions defined by structural recursion over a given algebraic data type definition. The functions we consider map an algebraic data type to a string; they are useful for, e.g., pretty printing and serialization of programs and data. We formalize our problem as learning deterministic sequential top-down tree-to-string transducers with a single state (1STS).

The first problem we consider is learning a tree-to-string transducer from any set of input/output examples provided by the user. We show that, given a set of input/output examples, checking whether there exists a 1STS consistent with these examples is NP-complete in general. In contrast, the problem can be solved in polynomial time under a (practically useful) closure condition that each subtree of a tree in the input/output example set is also part of the input/output examples.

Because coming up with relevant input/output examples may be difficult for the user while creating hard constraint problems for the synthesizer, we also study a more automated active learning scenario in which the algorithm chooses the inputs for which the user provides the outputs. Our algorithm asks a worst-case linear number of queries as a function of the size of the algebraic data type definition to determine a unique transducer.

To construct our algorithms we present two new results on formal languages.

First, we define a class of word equations, called sequential word equations, for which we prove that satisfiability can be solved in deterministic polynomial time. This is in contrast to the general word equations for which the best known complexity upper bound is in linear space.

Second, we close a long-standing open problem about the asymptotic size of test sets for context-free languages. A test set of a language of words L is a subset T of L such that any two word homomorphisms equivalent on T are also equivalent on L. We prove that it is possible to build test sets of cubic size for context-free languages, matching for the first time the lower bound found 20 years ago.

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.13

1 Introduction

Synthesis by example has been very successful to help users deal with the tedious task of writing a program. This technique allows the user to specify input/output examples to describe the intended behavior of a desired program. Synthesis will then inspect the examples

* This work was partially supported by European Research Council (ERC) Project Implicit Programming and an EPFL-Inria Post-Doctoral grant.
† The full version of this paper including detailed proofs is available at [33]

© Mikaël Mayer, Jad Hamza and Viktor Kunčak;
licensed under Creative Commons License CC-BY
31st European Conference on Object-Oriented Programming (ECOOP 2017).
Editor: Peter Müller; Article No. 13; pp. 13:1–13:30

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

given by the user, and generalize them into a program that respects these examples, and that is also able to handle other inputs.

Therefore, synthesis by example allows non-programmers to write programs without programming experience, and gives experienced users one more way of programming that could fit their needs. Current synthesis techniques usually rely on domain-specific heuristics to try and infer the desired program from the user. When there are multiple (non-equivalent) programs which are compatible with input/output examples provided by the user, these heuristics may fail to choose the program that the user had in mind when writing the examples.

We believe it is important to have algorithms that provide formal guarantees based on strong theoretical foundations. Algorithms we aim for ensure that the solution is found whenever it exists in a class of functions of interest. Furthermore, the algorithms ensure that the generated program is indeed the program the user wants by detecting once the solution is unique and otherwise identifying a differentiating example whose output reduces the space of possible solutions.

In this paper, we focus on synthesizing printing functions for objects or algebraic data types (ADT), which are at the core of many programming languages. Converting such structured values to strings is very common, including uses such as pretty printing, debugging, and serialization. Writing methods to convert objects to strings is repetitive and usually requires the user to code himself mutually recursive `toString` functions. Although some languages have default printing functions, these functions are often not adequate. For example, the object Person("Joe", 31) might have to be printed "Joe is 31 years old" for better readability, or "<td>Joe</td><td>31</td>" if printed as part of an HTML table. How *feasible* is it for the computer to learn these "printing" functions from examples?

The state of the art in this context [27, 26] requires the user to provide *enough* examples. If the user gives *too few* examples, the synthesis algorithm is not guaranteed to return a valid printing function, and there is no simple way for the user to know which examples should be added so that the synthesis algorithm finishes properly.

Our contribution is to provide an algorithm that is able to determine exactly which questions to ask the user so that the desired function can be derived. Moreover, in order to learn a function, our algorithm (Algorithm 3) only needs to ask a linear number of questions (as a function of the size of the ADT declaration).

Our results hold for recursive functions that take ADT as input, and output strings. We model these functions by tree-to-string transducers, called *single-state sequential top-down tree-to-string transducers* [9, 14, 19, 27, 43], or 1STS for short. In this formalism, objects are represented as labelled trees, and a transducer goes through the tree top down in order to display it as a string. *Single-state* means the transducer keeps no memory as it traverses the tree. *Sequential* is a shorthand for *linear* and *order-preserving*, meaning that each subtree is printed only once (linear), and the subtrees of a node are displayed in order (order-preserving). In particular, such transducers cannot directly represent recursive functions that have extra parameters alongside the tree to print. Our work on 1STSs establishes a foundation that may be used for larger classes of transducers.

Our goal is to learn a 1STS from a set of positive input/output examples, called a *sample*. We prove the problem of checking whether there exists a 1STS consistent with a given sample is NP-complete in general. Yet, we prove that when the given sample is closed under subtree, i.e., every tree in the sample has all of its subtrees in the sample, the problem of finding a compatible 1STS can be solved in polynomial time. For this, we reduce the problem of checking whether there exists an 1STS consistent with a sample to the problem of solving

word equations. The best known algorithm to solve word equations takes linear space, and exponential time [40, 22]. However, we prove that the word equations we build are of a particular form, which we call sequential, and our first algorithm learns 1STSs by solving sequential equations in polynomial time.

We then tackle the problem of ambiguities that come from underspecified samples. More precisely, it is possible that, given a sample, there exist two 1STSs that are consistent with the sample, but that are not equivalent on a domain D of trees. We thus define the notion of *tree test set* of a domain D, which guarantees that, any two 1STSs which are equivalent on the tree test set are also equivalent on the whole domain D. We give a method to build tree test sets of size $O(|D|^3)$ from a domain of trees given as a non-deterministic top-down automaton. Our second learning algorithm takes as input a domain D, builds the tree test set of D, and asks for the user the output to all trees in the tree test set. Our second algorithm then invokes our first algorithm on the given sample.

This construction relies on fundamental results on a known relation between sequential top-down tree-to-string transducers and morphisms (a morphism is a function that maps the concatenation of two words to the concatenation of their images), and on the notion of *test set* [43]. Informally, a test set of a language of words L is a subset $T \subseteq L$ such that any two morphisms which are equivalent on T are also equivalent on L. In the context of 1STSs, the language L is a context-free language, intuitively representing the yield of the domain D mentioned above. Prior to our work announced in [32], the best known construction for a test set of a context-free grammar G produced test sets of size $O(|G|^6)$, while the best known lower bound was $O(|G|^3)$ [38, 39]. We show the $O(|G|^3)$ is in fact tight, and give a construction that, given any grammar G, produces a test set for G of size $O(|G|^3)$.

Finally, our third and, from a practical point of view, the main algorithm, improves the second one by analyzing the previous outputs entered by the user, in order to infer the next output. More specifically, the outputs previously entered by the user give constraints on the transducer being learned, and therefore restrict the possible outputs for the next questions. Our algorithm computes these possible outputs and, when there is only one, skips the question. Our algorithm only asks the user a question when there are at least two possible outputs for a particular input. The crucial part of this algorithm is to prove that such ambiguities happen at most $O(|D|)$ times. Therefore, our third algorithm asks the user only $O(|D|)$ questions, greatly improving our second one that asks $O(|D|^3)$ questions. Our result relies on carefully inspecting the word equations produced by the input/output examples.

We implemented our algorithms in an open-source tool available at `https://github.com/epfl-lara/prosy`. In sections 9 and 10, we describe how to extend our algorithms and tool to ADTs which contain String (or Int) as a primitive type. We call the implementation of our algorithms *proactive* synthesis, because it produces a *complete* set of questions ahead-of-time whose answers will help to synthesize a unique tree-to-string function, *filters out* future questions whose answer could be actively inferred after each user's answer, and produces *suggestions* as multiple choice or pre-filled answers to minimize the answering effort.

Contributions

Our paper makes the following contributions:

1. A new efficient algorithm to synthesize recursive functions from examples. We give a polynomial-time algorithm to obtain a 1STS from a sample *closed under subtree*. When the sample is not necessarily closed under subtree, we prove that the problem of checking whether there exists a 1STS consistent with the sample is NP-complete (Section 6). This

result is based on a fundamental contribution:
   - A polynomial-time algorithm for solving a class of word equations that come from a synthesis problem (*sequential* word equations, Section 6).
2. An algorithm that synthesize recursive functions without ambiguity by generating an exhaustive set of questions to ask to the user, in the sense that any two recursive functions that agree on these inputs, are equal on their entire domain (Section 7). This is based on the following fundamental contribution:
   - A constructive upper bound of $O(|G|^3)$ on the size of a test set for a context-free grammar G, improving on the previous known bound of $O(|G|^6)$ [38, 39] (Section 7).
3. A proactive and efficient algorithm that synthesizes recursive functions, which only requires the user to enter outputs for the inputs determined by the algorithm. Formally, we present an interactive algorithm to learn a 1STS for a domain of trees, with the guarantee that the obtained 1STS is functionally unique. Our algorithm asks the user only a *linear* number of questions (Section 8).
4. A construction of a linear tree test set for data types with Strings, which enables constructing a small set of inputs that distinguish between two recursive functions (Section 9).
5. An implementation of our algorithms as an interactive command-line tool (Section 10)

We note that the fundamental contributions of (1) and (2) are new general results about formal languages and may be of interest on their own.

For space purposes, we only show proof sketches and intuition; detailed proofs can be found in the extended version of this paper [33].

2 Example Run of Our Synthesis Algorithm

To motivate our problem domain, we present a run of our algorithm on an example. The example is an ADT representing a context-free grammar. It defines its custom alphabet (Char), words (CharList), and non-terminals indexed by words (NonTerminal). A rule (Rule) is a pair made of a non-terminal and a sequence of symbols (ListSymbol), which can be non-terminals or terminals (Terminal). Finally, a grammar is a pair made of a (starting) non-terminal and a sequence of rules.

The input of our algorithm is the following file (written in Scala syntax):

```
abstract class Char
case class a() extends Char
case class b() extends Char

abstract class CharList
case class NilChar() extends CharList
case class ConsChar(c: Char, l: CharList) extends CharList

abstract class Symbol
case class Terminal(t: Char) extends Symbol
case class NonTerminal(s: CharList) extends Symbol

case class Rule(lhs: NonTerminal, rhs: ListSymbol)

abstract class ListRule
case class ConsRule(r: Rule, tail: ListRule) extends ListRule
case class NilRule() extends ListRule
```

```
abstract class ListSymbol
case class ConsSymbol(s: Symbol, tail: ListSymbol) extends ListSymbol
case class NilSymbol() extends ListSymbol

case class Grammar(s: NonTerminal, r: ListRule)
```

We would like to synthesize a recursive tree-to-string function `print`, such that if we compute, for example:

```
print(Grammar(NonTerminal(NilChar()),
    ConsRule(Rule(NonTerminal(NilChar()),
      ConsSymbol(Terminal(a()),
        ConsSymbol(NonTerminal(NilChar()),
          ConsSymbol(Terminal(b()), NilSymbol())))),
      ConsRule(Rule(NonTerminal(NilChar()),
          NilSymbol())), NilRule())))
```

the result should be:

```
Start: N
N -> a N b
N ->
```

We would like the `print` function to handle any valid `Grammar` tree.

When given these class definitions above, our algorithm precomputes a set of terms from the ADT, so that any two single-state recursive functions which output the same Strings for these terms also output the same Strings for any term from this ADT. (This is related to the notion of *tree test set* defined in Section 7.2.) Our algorithm will determine the outputs for these terms by interacting with the user and asking questions. Overall, for this example, our algorithm asks the output for 14 terms.

For readability, question lines provided by the synthesizer are indented. Lines entered by the user finish by the symbol ↩ , meaning that she pressed the ENTER key. Everything after ↩ on the same line is our comment on the interaction. "It" usually refers to the synthesizer. After few interactions, the questions themselves are shortened for conciseness. The interaction is the following:

```
  Proactive Synthesis.
  If you ever want to enter a new line, terminate your line by \ and press Enter.
  What should be the function output for the following input tree?
  a
a↩
  What should be the function output for the following input tree?
  b
b↩
  NilChar ?
↩              indeed, NilChar is an empty string.
  NilSymbol ?
↩              No symbol at the right-hand-side of a rule
  NilRule ?
↩              No rule left describing the grammar
  What should be the function output for the following input tree?
  Terminal(a)
  Something of the form: [...]a[...]
a↩             Terminals contain only one char. Note the hint provided by the synthesizer.
  NonTerminal(NilChar) ?
```

```
N↩
   ConsChar(b,NilChar) ? Something of the form: [...]b[...]
b↩                    A ConsChar is a concatenation of a char and a string
   What should be the function output for the following input tree?
   NonTerminal(ConsChar(b,NilChar))
   1) Nb
   2) bN
   Please enter a number between 1 and 2, or 0 if you really want to enter your answer manually
1↩                    Note that it was able to infer only two possibilities, thus the closed question.
   Grammar(NonTerminal(NilChar),NilRule) ? Something of the form: [...]N[...]
Start: N↩
   ConsSymbol(Terminal(a),NilSymbol) ? Something of the form: [...]‘a‘[...]
 a↩                   Symbols on the right-hand-side of a Rule are prefixed with a space
Rule(NonTerminal(NilChar),NilSymbol) ? Something of the form: [...]N[...]
N −>↩                 A rule with no symbols on the right-hand-side
   ConsRule(Rule(NonTerminal(NilChar),NilSymbol),NilRule) ?
   Something of the form: [...]N −>[...]
\↩                    A newline
N −>↩
   What should be the function output for the following input tree?
   Rule(NonTerminal(NilChar),ConsSymbol(Terminal(‘a‘),NilSymbol))
   1) N ‘a‘−>
   2) N − ‘a‘>
   3) N −> ‘a‘
   4) N ‘a‘ −>
   Please enter a number between 1 and 4, or 0 if you really want to enter your answer manually
3↩
```

The synthesizer then emits the desired recursive tree-to-string function, along with a complete set of the tests that determine it:

```
def print(t: Any): String = t match {
   case a() ⇒ "a"
   case b() ⇒ "b"
   case NilChar() ⇒ ""
   case ConsChar(t1,t2) ⇒ print(t1) + print(t2)
   case Terminal(t1) ⇒ "‘" + print(t1) + "‘"
   case NonTerminal(t1) ⇒ "N" + print(t1)
   case Rule(t1,t2) ⇒ print(t1) + " −>" + print(t2)
   case ConsRule(t1,t2) ⇒ "\n" + print(t1) + print(t2)
   case NilRule() ⇒ ""
   case ConsSymbol(t1,t2) ⇒ " " + print(t1) + print(t2)
   case NilSymbol() ⇒ ""
   case Grammar(t1,t2) ⇒ "Start: " + print(t1) + print(t2)
} // the part below is a contract, not needed to execute the recursive function
ensuring { (res: string) => res == (t match {
   case a() => "a"
   case b() => "b"
   case NilChar() => ""
   case NilSymbol() => ""
   case NilRule() => ""
   case Terminal(a()) => "a"
   case NonTerminal(NilChar()) => "N"
   case ConsChar(b(),NilChar()) => "b"
   case NonTerminal(ConsChar(b(),NilChar())) => "Nb"
```

```
  case Grammar(NonTerminal(NilChar()),NilRule()) => "Start: N"
  case ConsSymbol(Terminal(a()),NilSymbol()) => " a"
  case Rule(NonTerminal(NilChar()),NilSymbol()) => "N ->"
  case ConsRule(Rule(NonTerminal(NilChar()),NilSymbol()),NilRule()) => "\nN ->"
  case Rule(NonTerminal(NilChar()),ConsSymbol(Terminal(a()),NilSymbol())) => "N -> a"
  case _ => res})
}
```

Observe that, in addition to the program, the synthesis system emits as a postcondition (after the `ensuring` construct) the recorded input/output examples (tests). Our work enables the construction of an IDE that would automatically maintain the bidirectional correspondence between the body of the recursive function and the postcondition that specifies its input/output tests. If the user modifies an example in the postcondition, the system could re-synthesize the function, asking for clarification in cases where the tests become ambiguous. If the user modifies the program, such system can regenerate the tests.

Depending on user's answers, the total number of questions that the synthesizers asks varies (see section 11). Nonetheless, the properties that we proved for our algorithm guarantee that the number of questions remains at most *linear* as a function of the size of the algebraic data type declaration.

When the user enters outputs which are not consistent, i.e., for which there exists no printing function in the class of functions that we consider, our tool directly detects it and warns the user. For instance, for the tree ConsRule(Rule(NonTerminal(NilChar),NilSymbol),NilRule), if the user enters N- > with the space and the dash inverted, the system detects that this output is not consistent with the output provided for tree Rule(NonTerminal(NilChar),NilSymbol), and asks the question again.

```
We cannot have the transducer convert ConsRule(Rule(NonTerminal(NilChar),NilSymbol),NilRule)
to N- >.
Please enter something consistent with what you previously entered (e.g. 'N ->','N ->bar',...)?
```

3 Discussion

3.1 Advantages of Synthesis Approach

It is important to emphasize that in the approach we outline, the developer not only enters less text in terms of the number of characters than in the above source code, but that the input from the user is entirely in terms of concrete input-output *values*, which can be easier to reason about for non-expert users than recursive programs with variable names and control-flow.

It is notable that the synthesizer in many cases offered suggestions, which means that the user often simply needed to check whether one of the candidate outputs is acceptable. Even in cases where the user needed to provide new parts of the string, the synthesizer in many cases guided the user towards a form of the output consistent with the outputs provided so far. Because of this knowledge, the synthesizer could also be stopped early by, for example, guessing the unknown information according to some preference (e.g. replacing all unknown string constants by empty strings), so the user can in many cases obtain a program by providing a very small amount of information.

Such easy-to-use interactions could be implemented as a pretty printing wizard in an IDE, for example triggered when the user starts to write a function to convert an ADT to a String.

Our experience in writing pretty printers manually suggests that they often require testing to ensure that the generated output corresponds to the desired intuition of the developer, suggesting that input-output tests may be a better form of specification even if in cases where they are more verbose. We therefore believe that it is valuable to make available to users and developers sucn an alternative method of specifying recursive functions, a method that can co-exist with the conventional explicitly written recursive functions and the functions derived automatically (but generically) by the compiler (such as default printing of algebraic data type values in Scala), or using polytypic programming approaches [21] and serialization libraries [35]. (Note that the generic approaches can reduce the boilerplate, but do not address the problem of unambiguously generalizing *examples* to recursive functions.)

3.2 Challenges in Obtaining Efficient Algorithms

The problem of inferring a program from examples requires recovering the constants embedded in the program from the results of concatenating these constants according to the structure of the given input tree examples. This presents two main challenges. The first one is that the algorithm needs to split the output string and identify which parts correspond to constants and which to recursive calls. This process becomes particularly ambiguous if the alphabet used is small or if some constants are empty strings. A natural way to solve such problems is to formulate them as a conjunction of word equations. Unfortunately, the best known deterministic algorithms for solving word equations run in exponential time (the best complexity upper bound for the problem takes linear space [40, 22]). Our paper shows that, under an assumption that, when specifying printing of a tree, we also specify printing of its subtrees, we obtain word equations solvable in *polynomial time*.

The next challenge is the number of examples that need to be solved. Here, a previous upper bound derived from the theory of test sets of context-free languages was $\Omega(n^6)$, which, even if polynomial, results in impractical number of user interactions. In this paper we improve this theoretical result and show that tests sets are in fact in $O(n^3)$, asymptotically matching the known lower bound.

Furthermore, if we allow the learning algorithm to choose the inputs one by one after obtaining outputs, the overall learning algorithm has a *linear* number of queries to user and to equation solving subroutine, as a function of the size of tree data type definition. Our contributions therefore lead to tools that have completeness guarantees with much less user input and a shorter running time than the algorithms based on prior techniques.

We next present our algorithms as well as the results that justify their correctness and completeness.

4 Notation

We start by introducing our notation and terminology for some standard concepts. Given a (partial) function from $f : A \to B$, and a set C, $f_{|C}$ denotes the (partial) function $g : A \cap C \to B$ such that $g(a) = f(a)$ for all $a \in A \cap C$.

A word (string) is a finite sequence of elements of a finite set Σ, which we call an *alphabet*.

A *morphism* $f : \Sigma^* \to \Gamma^*$ is a function such that $f(\varepsilon) = \varepsilon$ and for every $u, v \in \Sigma^*$, $f(u \cdot v) = f(u) \cdot f(v)$, where the symbol '$\cdot$' denotes the concatenation of words (strings).

A *non-deterministic finite automaton (NFA)* is a tuple $(\Gamma, Q, q_i, F, \delta)$ where Γ is the alphabet, Q is the set of states, $q_i \in Q$ is the initial state, F is the set of final states, $\delta \subseteq Q \times \Gamma \times Q$ is the transition relation. When the transition relation is deterministic, that

is for all $q, p_1, p_2 \in Q, a \in \Gamma$, if $(q, a, p_1) \in \delta$ and $(q, a, p_2) \in \delta$, then $p_1 = p_2$, we say that A is a *deterministic finite automaton (DFA)*.

A *context-free grammar* G is a tuple (N, Σ, R, S) where:

- N is a set of *non-terminals*,
- Σ is a set of *terminals*, disjoint from N,
- $R \subseteq N \times (N \cup \Sigma)^*$ is a set of *production rules*,
- $S \in N$ is the starting non-terminal symbol.

A production $(A, rhs) \in R$ is denoted $A \to rhs$. The *size* of G, denoted $|G|$, is the sum of sizes of each production in R: $\sum_{A \to rhs \in R} 1 + |rhs|$. A grammar is *linear* if for every production $A \to rhs \in R$, the *rhs* string contains at most one occurrence of N. By an abuse of notation, we denote by G the set of words produced by G.

4.1 Trees and Domains

A *ranked alphabet* Σ is a set of pairs (f, k) where f is a symbol from a finite alphabet, and $k \in \mathbb{N}$. A pair (f, k) of a ranked alphabet is also denoted $f^{(k)}$. We say that symbol f has a *rank* (or *arity*) equal to k. We define by $\mathcal{T}_\Sigma$ the set of trees defined over alphabet Σ. Formally, $\mathcal{T}_\Sigma$ is the smallest set such that, if $t_1, \dots, t_k \in \mathcal{T}_\Sigma$, and $f^{(k)} \in \Sigma$ for some $k \in \mathbb{N}$, then $f(t_1, \dots, t_k) \in \mathcal{T}_\Sigma$. A set of trees T is *closed under subtree* if for all $f(t_1, \dots, t_k) \in T$, for all $i \in \{1, \dots, k\}$, $t_i \in T$.

A top-down tree automaton T is a tuple (Σ, Q, I, δ) where Σ is a ranked alphabet, $I \subseteq Q$ is the set of initial states, and $\delta \subseteq \Sigma \times Q \times Q^*$. The set of trees $\mathcal{L}(T)$ recognized by T is defined recursively as follows. For $f^{(k)} \in \Sigma$, $q \in Q$, and $t = f(t_1, \dots, t_k) \in \mathcal{T}_\Sigma$, we have $t \in \mathcal{L}(T)_q$ iff there exists $(f, q, q_1 \cdots q_k) \in \delta$ such that for $1 \le i \le k$, $t_i \in \mathcal{L}(T)_{q_i}$. The set $\mathcal{L}(T)$ is then defined as $\bigcup_{q \in I} \mathcal{L}(T)_q$.

Algebraic data types are described by the notion of *domain*, which is a set of trees recognized by a top-down tree automaton $T = (\Sigma, Q, I, \delta)$. The *size* of the domain is the sum of sizes of each transition in δ, that is $\sum_{(f^{(k)}, q, q_1 \cdots q_k) \in \delta} 1 + k$.

▶ **Example 1.** In this example and the following ones, we illustrate our notions using an encoding of HTML-like data structures. Consider the following algebraic data type definitions in Scala:

```
abstract class Node
case class node(t: Tag, l: List) extends Node

abstract class Tag
case class div() extends Tag
case class pre() extends Tag
case class span() extends Tag

abstract class List
case class cons(n: Node, l: List) extends List
case class nil() extends List
```

The corresponding domain D_{html} is described by the following:

$$\begin{aligned}
\Sigma &= \{\text{nil}^{(0)}, \text{cons}^{(2)}, \text{node}^{(2)}, \text{div}^{(0)}, \text{pre}^{(0)}, \text{span}^{(0)}\} \\
Q &= \{\text{Node}, \text{Tag}, \text{List}\} \\
I &= \{\text{Node}, \text{Tag}, \text{List}\} \\
\delta &= \{(\text{node}, \text{Node}, (\text{Tag}, \text{List})), \\
&\quad (\text{div}, \text{Tag}, ()), (\text{pre}, \text{Tag}, ()), (\text{span}, \text{Tag}, ()), \\
&\quad (\text{cons}, \text{List}, (\text{Node}, \text{List})), \\
&\quad (\text{nil}, \text{List}, ())\}
\end{aligned}$$

4.2 Transducers

A *deterministic, sequential, single-state, top-down tree-to-string transducer* τ (1STS for short) is a tuple (Σ, Γ, δ) where:

- Σ is a ranked alphabet (of trees),
- Γ is an alphabet (of words),
- δ is a function over Σ such that $\forall f^{(k)} \in \Sigma.\ \delta(f) \in (\Gamma^*)^{k+1}$.

Note that the transducer does not depend on a particular domain for Σ, but instead can map any tree from $\mathcal{T}_\Sigma$ to a word. Later, when we present our learning algorithms for 1STSs, we restrict ourselves to particular domains provided by the user of the algorithm.

We denote by $[\![\tau]\!]$ the function from trees to words associated with the 1STS τ. Formally, for every $f^{(k)} \in \Sigma$, we have $[\![\tau]\!](f(t_1, \dots, t_k)) = u_0 \cdot [\![\tau]\!](t_1) \cdot u_1 \cdots [\![\tau]\!](t_k) \cdot u_k$ if $\delta(f) = (u_0, u_1, \dots, u_k)$. When clear from context, we abuse notation and use τ as a shorthand for the function $[\![\tau]\!]$.

▶ **Example 2.** A transducer $\tau = (\Sigma, \Gamma, \delta)$ converting HTML trees into a convenient syntax for some programmatic templating engines[1] may be described by:

$$\begin{aligned}
\Sigma &= \{\text{nil}^{(0)}, \text{cons}^{(2)}, \text{node}^{(2)}, \text{div}^{(0)}, \text{pre}^{(0)}, \text{span}^{(0)}\}\} \\
\Gamma &= [\textit{All symbols}] \\
\delta(\text{node}) &= (\text{“<.”}, \varepsilon, \varepsilon)
\end{aligned}$$

$$\delta(\text{div}) = (\text{“div”}) \qquad \delta(\text{pre}) = (\text{“pre”}) \qquad \delta(\text{span}) = (\text{“span”}),$$

$$\delta(\text{cons}) = (\text{“(”}, \text{“)”}, \varepsilon) \qquad \delta(\text{nil}) = (\varepsilon)$$

In Scala, this is written as follows:

```
def tau(input: Tree) = input match {
  case node(t, l) ⇒ "<." + tau(t) + "" + tau(l) + ""
  case div() ⇒ "div"
  case pre() ⇒ "pre"
  case span() ⇒ "span"
  case cons(n, l) ⇒ "(" + tau(n) + ")" + tau(l) + ""
  case nil() ⇒ ""
}
```

For example, `tau(node(div,cons(node(span,nil,cons(node(pre,nil)))))) = "<.div(<.span())(<.pre())"`

[1] https://github.com/lihaoyi/scalatags

```
def tree(w: List[Σ̄]): Tree =
  if w is empty or does not start with some (f, 0):
    throw error
  let (f, 0) = w.head
  w ← w.tail
  for i from 1 to arity(f)
    t_i = tree(w)
    assert(w starts with (f, i))
    w ← w.tail
  return f(t_1, ..., t_k)
```

Figure 1 Parsing algorithm to obtain $\mathsf{tree}(w)$ from a word $w \in \overline{\Sigma}^*$. When the algorithm fails, because of a pattern matching error or because of the thrown exception, it means there exists no t such that $\tau_\Sigma(t) = w$.

5 Transducers as Morphisms

For a given alphabet Σ, a 1STS (Σ, Γ, δ) is completely determined by the constants that appear in δ. This allows us to define a one-to-one correspondence between transducers and morphisms. This correspondence is made through what we call the *default transducer*. More specifically, Γ is the set $\overline{\Sigma} = \{(f, i) \mid f^{(k)} \in \Sigma \wedge 0 \leq i \leq k\}$ and for all $f^{(k)} \in \Sigma$, we have $\delta(f) = ((f, 0), (f, 1), \dots, (f, k))$. The default transducer produces sequences of pairs from $\overline{\Sigma}$.

▶ **Example 3.** For $\Sigma = \{\text{nil}^{(0)}, \text{cons}^{(2)}, \text{node}^{(2)}, \text{div}^{(0)}, \text{pre}^{(0)}, \text{span}^{(0)}\}$, τ_Σ is:

$$\Gamma = \{ \text{(node, 0), (node, 1), (node, 2), (div, 0), (pre, 0), (span, 0)} \\ \text{(cons, 0), (cons, 1), (cons, 2), (nil, 0)}\}$$

$$\delta(\text{node}) = ((\text{node}, 0), (\text{node}, 1), (\text{node}, 2))$$

$$\delta(\text{div}) = (\text{div}, 0) \qquad \delta(\text{pre}) = (\text{pre}, 0) \qquad \delta(\text{span}) = (\text{span}, 0)$$

$$\delta(\text{cons}) = ((\text{cons}, 0), (\text{cons}, 1), (\text{cons}, 2)) \qquad \delta(\text{nil}) = (\text{nil}, 0))$$

In Scala, τ_Σ can be written as follows (+ is used to concatenate elements and lists):

```
def tauSigma(input: Tree): List[Σ̄] = input match {
  case node(t, l) ⇒ (node,0) + tauSigma(t) + (node,1) + tauSigma(l) + (node,2)
  case div() ⇒ (div,0)
  case pre() ⇒ (pre,0)
  case span() ⇒ (span,0)
  case cons(n, l) ⇒ (cons,0) + tauSigma(n) + (cons,1) + tauSigma(l) + (cons,2)
  case nil(n, l) ⇒ (nil,0)
}
```

▶ **Lemma 5.1.1.** *For any ranked alphabet Σ, the function $[\![\tau_\Sigma]\!]$ is injective.*

Following Lemma 5.1.1, for a word $w \in \overline{\Sigma}^*$, we define $\mathsf{tree}(w)$ to be the unique tree (when it exists) such that $\tau_\Sigma(\mathsf{tree}(w)) = w$. We show in Figure 1 how to obtain $\mathsf{tree}(w)$ in linear time from w.

For a 1STS $\tau = (\Sigma, \Gamma, \delta)$, we define the morphism $\mathtt{morph}[\tau]$ from $\overline{\Sigma}$ to Γ^*, and such that, for all $f^{(k)} \in \Sigma$, $i \in \{0, \dots, k\}$, $\mathtt{morph}[\tau](f, i) = u_i$ where $\delta(f) = (u_0, u_1, \dots, u_k)$. Conversely,

given a morphism $\mu : \overline{\Sigma} \to \Gamma^*$, we define $\texttt{sts}(\mu)$ as τ_Σ where each output $l \in \overline{\Sigma}$ is replaced by $\mu(l)$.

▶ **Example 4.** For Example 2, $\texttt{morph}[\tau]$ is defined by:

$$\begin{array}{ll} \texttt{morph}[\tau](\text{node}, 0) = \text{“<.”} & \texttt{morph}[\tau](\text{cons}, 0) = \text{“(”} \\ \texttt{morph}[\tau](\text{node}, 1) = \varepsilon & \texttt{morph}[\tau](\text{cons}, 1) = \text{“)”} \\ \texttt{morph}[\tau](\text{node}, 2) = \varepsilon & \texttt{morph}[\tau](\text{cons}, 2) = \varepsilon \\ \texttt{morph}[\tau](\text{div}, 0) = \text{“div”} & \texttt{morph}[\tau](\text{nil}, 0) = \varepsilon \\ \texttt{morph}[\tau](\text{pre}, 0) = \text{“pre”} & \texttt{morph}[\tau](\text{span}, 0) = \text{“span”} \end{array}$$

Note that for any morphism: $\mu : \overline{\Sigma} \to \Gamma^*$, $\texttt{morph}[\texttt{sts}(\mu)] = \mu$ and for any 1STS τ, $\texttt{sts}(\texttt{morph}[\tau]) = \tau$. Moreover, we have the following result, which expresses the output of a 1STS τ using the morphism $\texttt{morph}[\tau]$.

▶ **Lemma 5.1.2.** *For a* 1STS τ*, and for all* $t \in \mathcal{T}_\Sigma$*,* $\texttt{morph}[\tau](\tau_\Sigma(t)) = \tau(t)$*.*

Proof. Follows directly from the definitions of $\texttt{morph}[\tau]$ and τ_Σ. ◀

▶ **Example 5.** Let $t = \text{cons}(\text{node}(\text{div}, \text{nil}), \text{nil})$. For $\texttt{morph}[]$ defined as in Example 4 and the transducer τ as in Example 2, the left-hand-side of the equation of Lemma 5.1.2 translates to:

$$\begin{aligned} & \texttt{morph}[\tau](\tau_\Sigma(t)) \\ = {} & \texttt{morph}[\tau](\tau_\Sigma(\text{cons}(\text{node}(\text{div}, \text{nil}), \text{nil}))) \\ = {} & \texttt{morph}[\tau]((\text{cons}, 0)(\text{node}, 0)(\text{div}, 0)(\text{node}, 1)(\text{nil}, 0)(\text{node}, 2)(\text{cons}, 1)(\text{nil}, 0)(\text{cons}, 2)) \\ = {} & \text{“(”} \cdot \text{“<.”} \cdot \text{“div”} \cdot \varepsilon \cdot \varepsilon \cdot \varepsilon \cdot \text{“)”} \cdot \varepsilon \cdot \varepsilon \\ = {} & \text{“(<.div)”} \end{aligned}$$

Similarly, the right-hand-side of the equation can be computed as follows:

$$\begin{aligned} & \tau(t) \\ = {} & \tau(\text{cons}(\text{node}(\text{div}, \text{nil}), \text{nil})) \\ = {} & \text{“(”} \cdot \tau(\text{node}(\text{div}, \text{nil})) \cdot \text{“)”} \cdot \tau(\text{nil}) \cdot \varepsilon \\ = {} & \text{“(”} \cdot \text{“<.”} \cdot \tau(\text{div}) \cdot \varepsilon \cdot \tau(\text{nil}) \cdot \varepsilon \cdot \text{“)”} \cdot \varepsilon \cdot \varepsilon \\ = {} & \text{“(<.div)”} \end{aligned}$$

We thus obtain that checking equivalence of 1STSs can be reduced to checking equivalence of morphisms on a context-free language.

▶ **Lemma 5.1.3** (See [43]). *Let* τ_1 *and* τ_2 *be two* 1STS*s, and* $D = (\Sigma, Q, I, \delta)$ *a domain. Then* $[\![\tau_1]\!]_{|D} = [\![\tau_2]\!]_{|D}$ *if and only if* $\texttt{morph}[\tau_1]_{|G} = \texttt{morph}[\tau_2]_{|G}$ *where* G *is the context-free language* $\{\tau_\Sigma(t) \mid t \in D\}$*.*

Proof. Follows from Lemma 5.1.2. G is context-free, as it can be recognized by the grammar $(N_G, \overline{\Sigma}, R_G, S_G)$ where:

- $N_G = \{S_G\} \cup \{A_q \mid q \in Q\}$, where S_G is a fresh symbol used as the starting non-terminal,
- The productions are:

$$\begin{aligned} R_G = \; & \{A_q \to (f, 0) \cdot A_{q_1} \cdot (f, 1) \cdots A_{q_k} \cdot (f, k) \mid f^{(k)} \in \Sigma \wedge (q, f, (q_1, \ldots, q_k)) \in \delta\} \\ & \cup \{S_G \to A_q \mid q \in I\} \end{aligned}$$

Note that the size of G is linear in the size of $|D|$ (as long as there are no unused states in D). ◀

6 Learning 1STS from a Sample

We now present a learning algorithm for learning 1STSs from sets of input/output examples, or a *sample*. Formally, a sample $\mathcal{S} : \mathcal{T}_\Sigma \nrightarrow \Gamma^*$ is a partial function from trees to words, or alternatively, a set of pairs (t, w) with $t \in \mathcal{T}_\Sigma$ and $w \in \Gamma^*$ such that each t is paired with at most one w.

6.1 NP-completeness of the general case

In general, we prove that finding whether there exists a 1STS consistent with a given a sample is an NP-complete problem. To prove NP-hardness, we reduce the one-in-three positive SAT problem. This problem asks, given a formula φ with no negated variables, whether there exists an assignment such that for each clause of φ, exactly one variable (out of three) evaluates to true.

▶ **Theorem 6.1.1.** *Given a sample $\mathcal{S}$, checking whether there exists a* 1STS *τ such that for all $(t, w) \in \mathcal{S}$, $\tau(t) = w$ is an* NP*-complete problem.*

Proof. (Sketch) We can check for the existence of τ in NP using the following idea. Every input/output example from the sample gives constraints on the constants of τ. Therefore, to check for the existence of τ, it is sufficient to non-deterministically guess constants which are subwords of the given output examples. We can then verify in polynomial-time whether the guessed constants form a 1STS τ which is consistent with the sample $\mathcal{S}$.

To prove NP-hardness, we consider a formula φ, instance of the one-in-three positive SAT. The formula φ has no negated variables, and is satisfiable if there exists an assignment to the boolean variables such that for each clause of φ, exactly one variable (out of three) evaluates to true.

We construct a sample $\mathcal{S}$ such that there exists a 1STS τ such that for all $(t, w) \in \mathcal{S}$, $\tau(t) = w$ if and only if φ is satisfiable. For each clause $(x, y, z) \in \varphi$, we construct an input/output example of the form $\mathcal{S}(x(y(z(\mathsf{nil})))) = a\#$ where x, y and z are symbols of arity 1 corresponding to the variables of the same name in φ, nil is a symbol of arity 0, and a and $\#$ are two special characters. Moreover, we add an input/output example stating that $\mathcal{S}(\mathsf{nil}) = \#$.

This construction forces the fact that a 1STS τ consistent with $\mathcal{S}$ will have a non-empty output (a) for exactly one symbol out of x, y, and z (therefore matching the requirements of one-in-three positive SAT formulas). ◀

In the sequel, we prove that if the domain of the given sample is closed under subtree, this problem can be solved in polynomial time.

6.2 Word Equations

Our learning algorithm relies on reducing the problem of learning a 1STS from a sample to the problem of solving word equations. In general, the best known algorithm for solving word equations is in linear space [40, 22], and takes exponential time to run. When the domain of the sample $\mathcal{S}$ is closed under subtree, the equations we construct have a particular form, and we call them *sequential formulas*. We show there is a polynomial-time algorithm for checking whether a sequential word formula is satisfiable.

▶ **Definition 6.** Let $\mathbb{X}$ be a finite set of *variables*, and Γ a finite alphabet. A *word equation* e is a pair $y_1 = y_2$ where $y_1, y_2 \in (\mathbb{X} \cup \Gamma)^*$. A *word formula* φ is a conjunction of word

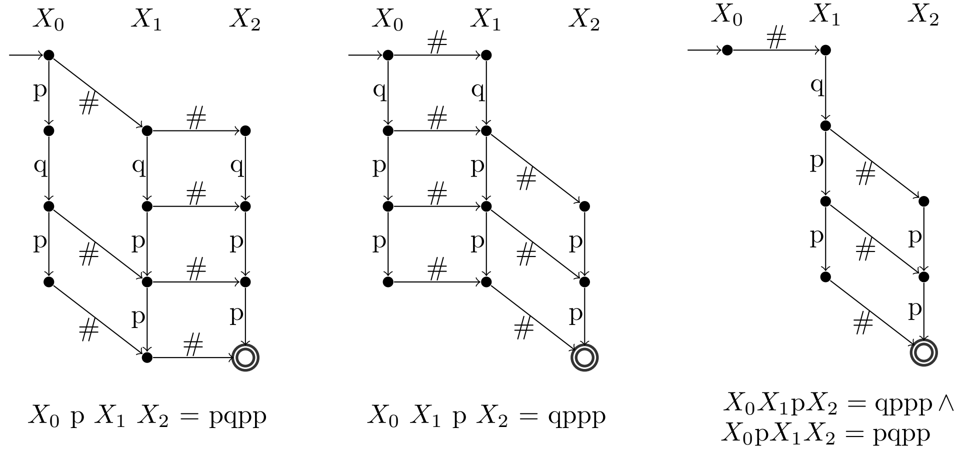

Figure 2 On the left, two automata representing the solutions of equations X_0 p X_1 X_2 = pqpp and X_0 X_1 p X_2 = qppp respectively. On the right, their intersection represents the solutions of the conjunction of equations. Note that the third automaton can be obtained from the first (and the second) by removing states and transitions.

equations. An *assignment* is a function from $\mathbb{X}$ to Γ^*, and can be seen as a morphism $\mu : (\mathbb{X} \cup \Gamma) \to \Gamma^*$ such that $\mu(a) = a$ for all $a \in \Gamma$.

A word formula is satisfiable if there exists an assignment $\mu : (\mathbb{X} \cup \Gamma) \to \Gamma^*$ such that for all equations $y_1 = y_2$ in φ, $\mu(y_1) = \mu(y_2)$.

A word formula φ is called *sequential* if: 1) for each equation $y_1 = y_2 \in \varphi$, $y_2 \in \Gamma^*$ contains no variable, and $y_1 \in (\Gamma \cup \mathbb{X})^*$ contains at most one occurrence of each variable, 2) for all equations $y = _$ and $y' = _$ in φ, either y and y' do not have variables in common, or $y_{|\mathbb{X}} = y'_{|\mathbb{X}}$, that is y and y' have the same sequence of variables. We used the name *sequential* due to this last fact.

▶ **Example 7.** For $X_1, X_2, X_3, X_4, X_5 \in \mathbb{X}$ and $p, q \in \Gamma^*$, each of the four formulas below is sequential:

$$X_1 = pq \qquad X_1X_3 = qpqpqqpqpq \wedge X_1qX_3 = qpqpqqqpqpq$$
$$X_1pX_2qX_3 = qppq \qquad X_1pqX_2X_3 = pqpqpp \wedge X_1X_2qpX_3 = pqppqp \wedge X_5pX_4 = qpq$$

The following formulas (and any formula containing them) are not sequential:

$$X_1pqX_2X_3 = pX_3pq \quad \text{(rhs is not in } \Gamma^*)$$
$$X_1pqX_2pX_3X_2 = ppqqpp \quad (X_2 \text{ appears twice in lhs})$$
$$X_1pqX_2X_3 = pqpqpp \wedge X_2pX_5 = qpq \quad (X_2 \text{ is shared})$$
$$X_1pqX_2X_3 = pqpqpp \wedge X_1pX_3X_2 = pqppp \quad \text{(different orderings of } X_1\ X_2\ X_3)$$

We prove that any sequential word formula φ can be solved in polynomial time.

▶ **Lemma 6.2.1.** *Let φ be a sequential word formula. Let n be the number of equations in φ, V the number of variables, and C be the size of the largest constant appearing in φ. We can determine in polynomial time $O(nVC)$ whether φ is satisfiable. When it is, we can also produce a satisfying assignment for φ.*

Proof. (Sketch) We construct for each equation in φ a DFA which represents succinctly all the possible assignments for this equation. Then, we take the intersection of all these DFAs, and obtain the possible assignments that satisfy all equations (i.e. the assignments that satisfy formula φ). The crucial part of the proof is to prove that this intersection can be computed in polynomial time, and does not produce an exponential blow-up as can be the case with arbitrary DFAs. We prove this by carefully inspecting the DFAs representing the assignments, and using the special form they have. We show the intersection of two such DFAs A and B is a DFA whose size is smaller than both the sizes of A and B (instead of being the product of the sizes of A and B, as can be the case for arbitrary DFAs). See Figure 2 for an illustration of this intersection. ◀

6.3 Algorithm for Learning from a Sample

Algorithm 1 Learning 1STSs from a sample.

Input: A sample $\mathcal{S}$ whose domain is closed under subtree.
Output: If there exists a 1STS τ such that $\tau(t) = w$ for all $(t, w) \in \mathcal{S}$, output Yes and τ, otherwise, output No.

1. Build the sequential formula $\varphi \equiv \bigwedge_{(t,w)\in\mathcal{S}} \mathsf{regEquation}(t, w, \mathcal{S})$
2. Check whether φ has a satisfying assignment μ as follows: (see Lemma 6.2.1):
   - For every word equation $\mathsf{regEquation}(t, w, \mathcal{S})$ where t has root f, build a DFA that represents all possible solutions for the words $\mu(f, 0),\dots,\mu(f, k)$.
   - Check whether the intersection of all DFAs contains some word w.
     - If no, exit the algorithm and return No.
     - If yes, define the words $\mu(f, 0),\dots,\mu(f, k)$ following w.
3. Return (Yes and) $\mathtt{sts}(\mu)$.

Consider a sample $\mathcal{S}$ such that $dom(\mathcal{S})$ is closed under subtree. Given $(t, w) \in \mathcal{S}$, we define the word equation $\mathsf{equation}(t, w)$ as:

$$\tau_\Sigma(t) = w$$

where the left hand side $\tau_\Sigma(t)$ is a concatenation of elements from $\overline{\Sigma}$, considered as word variables, and the right hand side $w \in \Gamma^*$ is considered to be a word constant.

Assume all equations corresponding to a set of input/output examples are simultaneously satisfiable, with an assignment $\mu : \overline{\Sigma} \to \Gamma^*$. Our algorithm then returns the 1STS $\tau = \mathtt{sts}(\mu)$, thus guarantying that $\tau(t) = w$ for all $(t, w) \in \Sigma$.

If the equations are not simultaneously satisfiable, our algorithm returns No.

▶ **Example 8.** For $\Sigma = \{\text{nil}^{(0)}, \text{cons}^{(2)}, \text{node}^{(2)}, \text{div}^{(0)}, \text{pre}^{(0)}, \text{span}^{(0)}\}$, given the examples:

$$\tau_\Sigma(\text{node}(\text{div}, \text{nil})) = \text{“<.div”}$$

$$\tau_\Sigma(\text{div}) = \text{“div”} \qquad \tau_\Sigma(\text{span}) = \text{“span”} \qquad \tau_\Sigma(\text{pre}) = \text{“pre”}$$

$$\tau_\Sigma(\text{cons}(\text{node}(\text{div}, \text{nil}), \text{nil}) = \text{“(<.div)”} \qquad \tau_\Sigma(\text{nil}) = \text{“”}$$

we obtain the following equations:

$$\begin{aligned}
(\text{node},0)\cdot(\text{div},0)\cdot(\text{node},1)\cdot(\text{nil},0)\cdot(\text{node},2) &= \text{``<.div''}\\
(\text{div},0) &= \text{``div''}\\
(\text{span},0) &= \text{``span''}\\
(\text{pre},0) &= \text{``pre''}\\
(\text{cons},0)\cdot(\text{node},0)\cdot(\text{div},0)\cdot(\text{node},1)\cdot(\text{nil},0)\cdot&\\
(\text{node},2)\cdot(\text{cons},1)\cdot(\text{nil},0)\cdot(\text{cons},2) &= \text{``(<.div)''}\\
(\text{nil},0) &= \text{``''}
\end{aligned}$$

A satisfying assignment for these equations is the morphism $\texttt{morph}[\tau]$ given in Example 4. Note that this assignment is not unique (see Example 9). We resolve ambiguities in Section 7.

To check for satisfiability of $\bigwedge_{(t,w)\in\mathcal{S}} \mathsf{equation}(t,w)$, we slightly transform the equations in order to obtain a sequential formula. For $(t,w)\in\mathcal{S}$, with $t = f(t_1,\ldots,t_k)$, we define the word equation $\mathsf{regEquation}(t,w,\mathcal{S})$ as:

$$(f,0)\, w_1\, (f,1)\cdots w_k\, (f,k) = w$$

where for all $i\in\{1,\ldots,k\}$, $w_i = \mathcal{S}(t_i)$. Note that $\mathcal{S}(t_i)$ must be defined, since t is in the domain of $\mathcal{S}$, which is closed under subtree. Moreover, the formula

$$\varphi \equiv \bigwedge_{(t,w)\in\mathcal{S}} \mathsf{regEquation}(t,w,\mathcal{S})$$

is satisfiable iff $\bigwedge_{(t,w)\in\mathcal{S}} \mathsf{equation}(t,w)$ is satisfiable.

Finally, φ is a sequential formula. Indeed, two equations corresponding to trees having the same root $f^{(k)}\in\Sigma$ have the same sequence of variables $(f,0)\ldots(f,k)$ in their left hand sides. And two equations corresponding to trees not having the same root have disjoint variables. Thus, using Lemma 6.2.1, we can check satisfiability of φ in polynomial time (and obtain a satisfying assignment for φ if there exists one).

▶ **Theorem 6.3.1** (Correctness and running time of Algorithm 1). *Let $\mathcal{S}$ be a sample whose domain is closed under subtree. If there exists a* 1STS *τ such that $\tau(t) = w$ for all $(t,w)\in\mathcal{S}$, Algorithm 1 returns one such* 1STS. *Otherwise, Algorithm 1 returns* No. *Algorithm 1 terminates in time polynomial in the size of $\mathcal{S}$.*

Proof. Assume φ has a satisfying assignment $\mu : \overline{\Sigma}\to\Gamma^*$, in step (2) of Algorithm 1. In that case, Algorithm 1 returns $\tau = \texttt{sts}(\mu)$. By definition of φ, we know, for all $(t,w)\in\mathcal{S}$, $\mu(\tau_\Sigma(t)) = w$. Moreover, since $\texttt{morph}[\tau] = \mu$, we have by Lemma 5.1.2 that $\tau(t) = \mu(\tau_\Sigma(t))$, so $\tau(t) = w$.

Conversely, if there exists τ such that $\tau(t) = w$ for all $(t,w)\in\mathcal{S}$. Then, again by Lemma 5.1.2, $\texttt{morph}[\tau]$ is a satisfying assignment for φ, and Algorithm 1 must return Yes.

The polynomial running time follows from Lemma 6.2.1.

▶ Remark. For samples whose domains are not closed under subtree, we may modify Algorithm 1 to check for satisfiability of word equations which are not necessarily sequential. In that case, we are not guaranteed that the running time is polynomial.

◀

7 Learning 1STSs Without Ambiguity

The issue with Algorithm 1 is that the 1STS expected by the user may be different than the one returned by the algorithm (see Example 9 below). To circumvent this issue, we use the notion of *tree test set*. Formally, a set of trees $T \subseteq D$ is a *tree test set for the domain D* if for all 1STSs τ_1 and τ_2, $[\![\tau_1]\!]_{|T} = [\![\tau_2]\!]_{|T}$ implies $[\![\tau_1]\!]_{|D} = [\![\tau_2]\!]_{|D}$.

▶ **Example 9.** The transducer τ_2 defined below satisfies the requirements of Example 8 but is different than the transducer in Example 2. Namely, the values in the box have been switched.

$$\delta_2(\text{node}) = (\text{“<.”}, \varepsilon, \varepsilon)$$

$$\delta_2(\text{div}) = (\text{“div”}) \qquad \delta_2(\text{pre}) = (\text{“pre”}) \qquad \delta_2(\text{span}) = (\text{“span”})$$

$$\delta_2(\text{cons}) = (\text{“(”}, \boxed{\varepsilon, \text{“)”}}) \qquad \delta_2(\text{nil}) = (\varepsilon)$$

We can verify that the two transducers are not equal on the domain D_{html}:

$$\tau(\text{cons}(\text{node}(\text{div}, \text{nil}), \text{cons}(\text{node}(\text{div}, \text{nil}), \text{nil}))) = \text{“(<.div)(<.div)”}$$
$$\tau_2(\text{cons}(\text{node}(\text{div}, \text{nil}), \text{cons}(\text{node}(\text{div}, \text{nil}), \text{nil}))) = \text{“(<.div(<.div))”}$$

Therefore, if a user had the 1STS τ in mind when giving the sample of Example 8, it is still possible that Algorithm 1 returns τ_2. However, by definition of *tree test set*, if the sample given to Algorithm 1 contains a tree test set for D_{html}, we are guaranteed that the resulting transducer is equivalent to the transducer that the user has in mind, for all trees on D_{html}.

Our goal in this section is to compute from a given domain D a tree test set for D. The notion of tree test set is derived from the well-known notion of *test set* in formal languages. The *test set* of a language L (a set of words) is a subset $T \subseteq L$ such that for any two morphisms $f, g : \Sigma^* \to \Gamma^*$, $f_{|T} = g_{|T}$ implies $f_{|L} = g_{|L}$.

To compute a tree test set T for D, we first compute a test set T_G for the context-free language $G = \{\tau_\Sigma(t) \mid t \in D\}$ (built in Lemma 5.1.3), and then define $T = \{\mathsf{tree}(w) \mid w \in T_G\}$. We prove in Lemma 7.2.1 that T is indeed a tree test set for D.

We introduce in Section 7.1 a new construction, asymptotically optimal, for building test sets of context-free languages. We show in Section 7.2 how this translates to a construction of a tree test set for a domain D. We also give a sufficient condition of D so that the obtained tree test set is closed under subtree. This allows us to present, in Section 7.3, an algorithm that learns 1STSs from a domain D in polynomial-time (by building the tree test set T of D, and asking to the user the outputs corresponding to the trees of T).

7.1 Test Sets for Context-Free Languages

We show in this section how to build, from a context-free grammar G, a test set of size of $O(|G|^3)$. Our construction is asymptotically optimal. We reuse lemmas from [38, 39], which were originally used to give a $O(|G|^6)$ construction.

7.1.1 Plandowski's Test Set

The following lemma was originally used in [38, 39] to show that any linear context-free grammar has a test set containing at most $O(|R|^6)$ elements. We show in Section 7.1.2 how this lemma can be used to show a $2|R|^3$ bound.

Let $\Sigma_4 = \{a_i, \overline{a_i}, b_i, \overline{b_i} \mid i \in \{1, 2, 3, 4\}\}$ be an alphabet. We define:

$L_4 = \{x_4\, x_3\, x_2\, x_1\, \overline{x_1}\, \overline{x_2}\, \overline{x_3}\, \overline{x_4} \mid \forall i \in \{1, 2, 3, 4\}.\ (x_i, \overline{x_i}) = (a_i, \overline{a_i}) \vee (x_i, \overline{x_i}) = (b_i, \overline{b_i})\}$

and $T_4 = L_4 \setminus \{b_4\, b_3\, b_2\, b_1\, \overline{b_1}\, \overline{b_2}\, \overline{b_3}\, \overline{b_4}\}$.

The sets $L_4, T_4 \subseteq \Sigma_4$ have 16 and 15 elements respectively.

▶ **Lemma 7.1.1** ([38, 39]). *T_4 is a test set for L_4.*

7.1.2 Linear Context-Free Grammars

We now prove that for any linear context-free grammar G, there exists a test set whose size is $2|R|^3$. Like the original proof of [38, 39] that gave a $O(|R|^6)$ upper bound, our proof relies on Lemma 7.1.1. However, our proof uses a different construction to obtain the new, tight, bound.

▶ **Theorem 7.1.1.** *Let $G = (N, \Sigma, R, S)$ be a linear context-free grammar. There exists a test set $T \subseteq G$ for G containing at most $2|R|^3$ elements.*

Proof. (Sketch) Our proof relies on the fact that a linear grammar G can be seen as a labelled graph whose nodes are non-terminals and whose transitions are rules of the grammar. A special node labelled $\perp$ is used for rules whose right-hand-sides are constant. We define the notion of *optimal path* in this graph. We use optimal paths to define paths which are piecewise optimal. More precisely, for $k \in \mathbb{N}$, a word belongs to the set $\Phi_k(G)$ if it can be derived in G by a path that can be split into $k+1$ optimal paths. We then prove that $\Phi_3(G)$ forms a test set for G (by using Lemma 7.1.1), which ends our proof as $\Phi_3(G)$ contains $O(|R|^3)$ elements. ◀

We make use of this theorem in the next section to obtain test sets for context-free grammars which are not necessarily linear.

7.1.3 Context-Free Grammars

To obtain a test set for a context-free grammar G which is not necessarily linear, [38] constructs from G a linear context-free grammar, $\mathrm{Lin}(G)$, which produces a subset of G, and which is a test set for G.

Formally, $\mathrm{Lin}(G)$ is derived from G as follows:

- For every productive non-terminal symbol A in G, choose a word x_A produced by A.
- Every rule $r : A \to x_0 A_1 x_1 \ldots A_n x_n$ in G, where for every i, $x_i \in \Sigma^*$ and $A_i \in N$ is productive, is replaced by n different rules, each one obtained from r by replacing all A_i with x_{A_i}, except one.

Note that the definition of $\mathrm{Lin}(G)$ is not unique, and depends on the choice of the words x_A. The following result holds for any choice of the words x_A.

▶ **Lemma 7.1.2** ([38, 39]). *$Lin(G)$ is a test set for G.*

Using Theorem 7.1.1, we improve the $O(|G|^6)$ bound of [38, 39] for the test set of G to $2|G|^3$.

▶ **Theorem 7.1.2.** *Let $G = (N, \Sigma, R, S)$ be a context-free grammar. There exists a test set $T \subseteq G$ for G containing at most $2|G|^3$ elements.*

Proof. Follows from Theorem 7.1.1, Lemma 7.1.2, and from the fact that $\mathrm{Lin}(G)$ has at most $|G| = \sum_{A \to rhs \in R}(|rhs| + 1)$ rules. (When constructing $\mathrm{Lin}(G)$, each rule $A \to rhs$ of G is duplicated at most $|rhs|$ times.) ◀

7.2 Tree Test Sets for Transducers

We use the results of the previous section to construct a tree test set for a domain D.

▶ **Lemma 7.2.1.** *Any domain $D = (\Sigma, Q, I, \delta)$ has a tree test set T of size at most $O(|D|)^3$. Moreover, if $I = Q$, then we can build T such that T is closed under subtree.*

Proof. Intuitively, we build the tree test set for D by taking the set of trees corresponding to the test set of G, where G is the grammar built in Lemma 5.1.3.

Let τ_1 and τ_2 be two 1STSs. Let T_G be a test set for G. Define $T = \{\mathsf{tree}(w) \mid w \in T_G\}$. By Theorem 7.1.2, we can assume T_G has size at most $|G|^3$, and hence, T has size at most $|D|^3$. Let μ_1 and μ_2 be $\mathtt{morph}[\tau_1]$ and $\mathtt{morph}[\tau_2]$, respectively. We have:

$$
\begin{aligned}
&[\![\tau_1]\!]_{|T} = [\![\tau_2]\!]_{|T} \iff \\
&\forall t \in T.\ \tau_1(t) = \tau_2(t) \iff \\
&\forall w \in T_G.\ \tau_1(\mathsf{tree}(w)) = \tau_2(\mathsf{tree}(w)) \iff \text{(by Lemma 5.1.2)} \\
&\forall w \in T_G.\ \mu_1(\tau_\Sigma(\mathsf{tree}(w))) = \mu_2(\tau_\Sigma(\mathsf{tree}(w))) \iff \text{(by definition of } \mathsf{tree}) \\
&\forall w \in T_G.\ \mu_1(w) = \mu_2(w) \iff \text{(since } T_G \text{ is a test set for } G) \\
&\forall w \in G.\ \mu_1(w) = \mu_2(w) \iff \text{(see Lemma 5.1.3)} \\
&[\![\tau_1]\!]_{|D} = [\![\tau_2]\!]_{|D}
\end{aligned}
$$

This ends the proof that T is a tree test set for D.

We now show how to construct T such that it is closed under subtree. For every non-terminal A of G, we define the minimal word w_A. These words are built inductively, starting from the non-terminals which have a rule whose right-hand-side is only made of terminals. In the definition of $\mathrm{Lin}(G)$, we use these words when modifying the rules of G into linear rules.

When then define T_G as the test set of $\mathrm{Lin}(G)$ (which is also a test set of G), and $T = \{\mathsf{tree}(w) \mid w \in T_G\} \cup \{\mathsf{tree}(w_A) \mid A \in G\}$. As shown previously, T is a tree test set for D. We can now prove that T is closed under subtree. Let $t = f(t_1, \ldots, t_k) \in T$. Let $i \in \{1, \ldots, k\}$. We want to prove that $t_i \in T$.

We consider two cases. Either there exists $w \in T_G$ such that $t = f(t_1, \ldots, t_k) = \mathsf{tree}(w)$, or there exists $A \in G$, $t = f(t_1, \ldots, t_k) = \mathsf{tree}(w_A)$.

- First, if there exists $w \in T_G$ such that $t = f(t_1, \ldots, t_k) = \mathsf{tree}(w)$. Consider a derivation p for w in the $\mathrm{Lin}(G)$. By construction of $\mathrm{Lin}(G)$, the first rule is an ε-transition of the form $S \to N$ while the second rule is of the form:

  $$N \to (f, 0) \cdot w_1 \cdot (f, 1) \cdots w_{j-1} \cdot (f, j-1) \cdot N_j \cdot (f, j) \cdot w_{j+1} \cdots w_k \cdot (f, k).$$

  This second rule corresponds to a rule in G, of the form:

  $$N \to (f, 0) \cdot N_1 \cdot (f, 1) \cdots N_{j-1} \cdot (f, j-1) \cdot N_j \cdot (f, j) \cdot N_{j+1} \cdots N_k \cdot (f, k).$$

  We then have two subcases to consider. Either $i \neq j$, and in that case $t_i = \mathsf{tree}(w_i)$. By construction of $\mathrm{Lin}(G)$, w_i must be equal to w_A for some $A \in G$. Thus, we have $t_i \in T$ by definition of T.
  Or $i = j$, in that case $t_i = \mathsf{tree}(w')$, where w' is derived by the derivation p where the first two derivation rules, outlined above, are replaced with the ε-rule $S \to N_i$. This production rule is ensured to exist in $\mathrm{Lin}(G)$, as all states of D are initial, so there exists a rule $S \to N_q$ for all $q \in Q$. (see definition of G in Lemma 5.1.3). Then, since $w \in \Phi_3(\mathrm{Lin}(G))$, and by construction of $\Phi_3(\mathrm{Lin}(G))$, we conclude that $w' \in \Phi_3(\mathrm{Lin}(G))$. This ensures that $w' \in T_G$, and $t_i \in T$.

- Otherwise, there exists $A \in G$ such that $t = f(t_1, \dots, t_k) = \mathsf{tree}(w_A)$. Using the fact that w_A was build inductively in the grammar G, using other minimal words $w_{A'}$ for $A' \in G$, we deduce there exists $A' \in G$ such that $t_i = \mathsf{tree}(w_{A'})$, and $t_i \in T$.

◀

Lemma 7.2.2 shows the bound given in Lemma 7.2.1 is tight, in the sense that there exists an infinite class of growing domains D for which the smallest tree test set has size $|D|^3$.

▶ **Lemma 7.2.2.** *There exists a sequence of domains $D_1, D_2, \dots$ such that for every $n \geq 1$, the smallest tree test set of D_n has at least n^3 elements, and the size of D_n is linear in n. Furthermore, this lower bound holds even with the extra assumption that all states of the domain are initial.*

Proof. (Sketch) Our proof is inspired by the lower bound proof for test sets of context-free languages [38, 39]. For $n \geq 1$, we build a particular domain D_n (whose states are all initial), and we assume by contradiction that it has a test set T of size less than n^3. From this assumption, we expose a tree $t \in D_n$, as well as two 1STSs τ_1 and τ_2 such that $\tau_{1|T} = \tau_{2|T}$ but $\tau_1(t) \neq \tau_2(t)$. ◀

7.3 Learning 1STSs Without Ambiguity

Algorithm 2 Learning 1STSs from a domain.

Input: A domain D, and an oracle 1STS τ_u.

Output: A 1STS τ functionally equivalent to τ_u.

1. Build a tree test set $\{t_1 \dots t_n\}$ of D, following Lemma 7.2.1.
2. For every $t_i \in \{t_1 \dots t_n\}$, ask the oracle for $w_i = \tau_u(t_i)$.
3. Run Algorithm 1 on the sample $\{(t_i, w_i) \mid 1 \leq i \leq n\}$.

Our second algorithm (see Algorithm 2) takes as input a domain D, and computes a tree test set $T \subseteq D$. It then asks the user the expected output for each tree $t \in T$. The user is modelled by a 1STS τ_u that can be used as an oracle in the algorithm. Algorithm 2 then runs Algorithm 1 on the obtained sample. The 1STS τ_u expected by the user may still be syntactically different the 1STS τ returned by our algorithm, but we are guaranteed that $[\![\tau]\!]_{|D} = [\![\tau_u]\!]_{|D}$ (by definition of tree test set).

▶ **Theorem 7.3.1** (Correctness and running time of Algorithm 2)**.** *Let τ_u be a 1STS (used as an oracle), and $D = (\Sigma, Q, I, \delta)$ a domain such that $I = Q$. The output τ of Algorithm 2 is a 1STS τ such that $[\![\tau]\!]_{|D} = [\![\tau_u]\!]_{|D}$.*

Furthermore, Algorithm 2 invokes the oracle $O(|D|^3)$ times, and terminates in time polynomial in $|D|$.

Proof. The correctness of Algorithm 2 follows from the correctness of Algorithm 1 and from the fact that T is a tree test set for D. The fact that Algorithm 2 invokes the algorithm $O(|D|^3)$ times follows from the size of the tree test set (see Lemma 7.2.1).

Moreover, since all states of D are initial, the tree test set of D that we build is closed under subtree. The polynomial running time then follows from the fact that Algorithm 1 ends in polynomial time for samples whose domains are closed under subtree.

▶ Remark. Similarly to Algorithm 1, Algorithm 2 also applies for domains such that $I \neq Q$, but the running time is not guaranteed to be polynomial.

◀

8 Learning 1STS Interactively

Algorithm 3 Interactive learning of 1STSs.

Input: A domain D, and an oracle 1STS τ_u whose output alphabet is Γ.

Output: A 1STS τ functionally equivalent to τ_u.

1. Initialize a map `sol` from Σ to Automata, such that for $f^{(k)} \in \Sigma$, `sol`(f) recognizes $\{x_0\# \cdots \#x_k \mid x_i \in \Gamma^*\}$,
2. Build a tree test set T of D, following Lemma 7.2.1.
3. Initialize a partial function $\mathcal{S} : \mathcal{T}_\Sigma \nrightarrow \Gamma^*$, initially undefined everywhere.
4. While $dom(\mathcal{S}) \neq T$:
   - Choose a tree $f(t_1, \dots, t_k) \notin dom(\mathcal{S})$ such that all subtrees of t belong to $dom(\mathcal{S})$ (possible since T is closed under subtree).
   - Build the automaton A recognizing $\{x_0\, \mathcal{S}(t_1)\, x_1 \cdots \mathcal{S}(t_k)\, x_k \mid x_0\#x_1 \cdots \#x_k \in \texttt{sol}(f)\}$, representing all possibles values of $\tau_u(t)$ that do not contradict previous outputs.
     - If A recognizes only 1 word w, define $\mathcal{S}(t) = w$.
     - Otherwise (A recognizes at least 2 words), define $\mathcal{S}(t) = \tau_u(t)$ using the oracle.
   - Update `sol`$(f) =$ `sol`$(f) \cap \mathsf{automaton}(t, \mathcal{S}(t))$.
5. Run Algorithm 1 on $\mathcal{S}$.

Our third algorithm (see Algorithm 3) takes as input a domain D, and computes a tree test set $T \subseteq D$. For this algorithm, we require from the beginning that all states of D are initial, so that T is closed under subtree. For a sample $\mathcal{S}$ such that $dom(\mathcal{S})$ is closed under subtree, and for $(t, w) \in \mathcal{S}$, we denote by $\mathsf{automaton}(t, w)$ the automaton $\mathsf{automaton}(y, w)$ where $y = w$ is the equation $\mathsf{regEquation}(t, w, \mathcal{S})$.

Instead of building the sample $\mathcal{S}$ and the intersection $\bigcap_{(t,w)\in\mathcal{S}} \mathsf{automaton}(t, w)$ all at once, like algorithms 1 and 2 do, Algorithm 3 builds $\mathcal{S}$ and the intersection incrementally. It then uses the intermediary results to infer outputs, in order to avoid calling the oracle τ_u too many times. Overall, we prove that Algorithm 3 invokes the oracle τ_u at most $O(|D|)$ times, while Algorithm 2 invokes it $O(|D|^3)$ times.

To infer outputs, Algorithm 3 maintains the following invariant for the while loop. First $\mathcal{S}$ is such that $dom(\mathcal{S}) \subseteq T$, and its domain increases at each iteration. Then, for any $f^{(k)} \in \Sigma$, `sol`(f) is equal to $\bigcap_{(t,w)\in\mathcal{S}} \mathsf{automaton}(t, w)$, and thus recognizes the set

$$\{\mu(f, 0)\#\mu(f, 1)\# \dots \#\mu(f, k) \mid \mu : \overline{\Sigma} \to \Gamma \text{ satisfies } \bigwedge_{(t,w)\in\mathcal{S}} \mathsf{regEquation}(t, w, \mathcal{S})\}.$$

Intuitively, `sol`(f) represents the possible values for the output of f in the transducer τ_u, based on the constraints given so far.

To infer the output of a tree $t = f(t_1, \dots, t_k)$, for some $f^{(k)} \in \Sigma$, Algorithm 3 uses the fact that $\tau_u(f(t_1, \dots, t_k))$ must be of the form $\mu(f, 0)\mathcal{S}(t_1)\mu(f, 1) \cdots \mathcal{S}(t_k)\mu(f, k)$ for some morphism $\mu : \overline{\Sigma} \to \Gamma$ satisfying $\bigwedge_{(t,w)\in\mathcal{S}} \mathsf{equation}(t, w)$. By construction, the NFA A, that recognizes the set $\{x_0\, \mathcal{S}(t_1)\, x_1 \cdots \mathcal{S}(t_k)\, x_k \mid x_0\#x_1 \cdots \#x_k \in \texttt{sol}(f)\}$, recognizes exactly these words of the form $\mu(f, 0)\mathcal{S}(t_1)\mu(f, 1) \cdots \mathcal{S}(t_k)\mu(f, k)$.

We then check whether A recognizes exactly one word w, in which case, we know $\tau_u(t) = w$, and we do not need to invoke the oracle. Otherwise, there are several alternatives which are consistent with the previous outputs provided by the user, and we cannot infer $\tau_u(t)$. We thus invoke the oracle (the user) to obtain $\tau_u(t)$.

Before proving the theorem corresponding to Algorithm 3, we give a lemma on words which we use extensively in the theorem.

▶ **Lemma 8.1.1.** *Let $u, v, w \in \Gamma^*$. If $uv = vu$ and $uw = wu$ and $u \neq \varepsilon$, then $vw = wv$.*

Proof. A word $p \in \Gamma^*$ is *primitive* if there does not exist $r \in \Gamma^*$, $i > 1$ such that $p = r^i$. Proposition 1.3.2 of [30] states that the set of words commuting with a non-empty word u is a monoid generated by a single primitive word p. Since v and w both commute with u, there exist i and j such that $v = p^i$ and $w = p^j$, thus $vw = wv = p^{i+j}$. ◀

The difficult part of Theorem 8.1.2 is to show the number of times the oracle τ_u is invoked is $O(|D|)$. We prove this by assuming by contradiction that the number of times τ_u is invoked is strictly greater than $3|D| + |Q|$ times. We prove this entails there are four trees which are nearly identical and for which our algorithm invokes the oracle (the four trees have the same root, and differ only for one child). Then, by a close analysis of the word equations corresponding to these four terms, we obtain a contradiction by proving our algorithm must have been able to infer the output for at least one of those terms.

▶ **Theorem 8.1.2** (Correctness and running time of Algorithm 3)**.** *Let τ_u be a* 1STS *(used as an oracle), and $D = (\Sigma, Q, I, \delta)$ a domain such that $I = Q$. The output τ of Algorithm 3 is a* 1STS *τ such that $[\![\tau]\!]_{|D} = [\![\tau_u]\!]_{|D}$.*

Algorithm 3 ends in time polynomial in $|D|$ and the number of times it invokes the oracle τ_u is in $O(|D|)$.

Proof. (Sketch) The correctness and the polynomial running time of Algorithm 3 can be proved similarly to Algorithm 2. Note that we can check whether the NFA A recognizes exactly one word. For that, we obtain a word w that A recognizes, and we intersect A with the complement of an automaton recognizing w.

The crucial part of Algorithm 3 is that it invokes the oracle τ_u at most $O(|D|)$ times. More precisely, we show that Algorithm 3 invokes τ_u at most $|Q| + 3\sum_{(q,f^{(k)},(q_1,\ldots,q_k))\in\delta} 1 + k$ times, which is $|Q| + 3|D|$, and in $O(|D|)$.

The main goal is to prove that for any trees four trees of the same root (t_a, t_b, t_c, t_d) differing from only one their ith subtree (respectively t_i^a, t_i^b, t_i^c, t_i^d), if we know the output of τ_u on all subtrees of t_a, t_b, t_c, t_d, then we can infer the output for at least one of t_a, t_b, t_c, t_d based on the previous outputs. Let $x_i^l = \tau_u(t_i^l)$ be the already known outputs of the sub-trees and $w_l = \tau_u(t_l)$ the outputs to ask to the user, for $l \in \{a, b, c, d\}$. We obtain the following equations where u, v represent the parts which do not change:

$$w_a = ux_i^a v \quad w_b = ux_i^b v \quad w_c = ux_i^c v \quad w_d = ux_i^d v$$

We prove by contradiction that we could not have asked the user for all w_l for $l \in \{a, b, c, d\}$, because at least one of the answer can be inferred from the previous ones. Here we illustrate two representative cases of the proof.

(1) One case is when x_i^a and x_i^b are neither prefix nor suffix of each other. By observing where w_a and w_b differ, we can recover u and v, and the algorithm could have inferred w_c and w_d.

(2) Another case is when x_i^a, x_i^b, and x_i^c are respectively of the form x_1, x_1x_2 and $x_1x_2x_3$ for some $x_1, x_2, x_3 \in \Gamma^*$ with $x_2x_3 = x_3x_2$, and $x_2 \neq \varepsilon$, $x_3 \neq \varepsilon$. Since we asked the output w_a, w_b and w_c, then after the first two questions, the values of u and v could not be determined. In particular, this means that there are some u, v and u', v' such that: $ux_1v = u'x_1v'$ and $ux_1x_2v = u'x_1x_2v'$ but $ux_1x_2x_3v \neq u'x_1x_2x_3v'$.

By assuming without loss of generality that $u = u'u''$ and $v' = v''v$, we obtain that $u''x_1 = x_1v''$ and $u''x_1x_2 = x_1x_2v''$, thus $v''x_2 = x_2v''$, and then x_2 commutes with v''. Since x_2 also commutes with x_3, we deduce v'' commutes with x_3, and then $u''x_1x_2x_3 = x_1x_2x_3v''$, which is a contradiction. ◀

9 Tree with Values

Until now, we have considered a set of trees $\mathcal{T}_\Sigma$ which contained only other trees as subtrees, and with a test set of size $O(n^3)$, although we have a linear learning time if we have interactivity. However, in practice, data structures such as XML are usually trees containing *values*. Values are typically of type stringor int, and may be used instead of subtrees. For convenience, we will suppose that we only have string elements, and that string elements are rendered *raw*. We will demonstrate how we can directly obtain a test set of size $O(n)$.

Formally, let us add a special symbol $v \in \Sigma$, of arity 0, which has another version which can have a parameter. For each string $s \in \Gamma^*$ we can thus define the symbol v_s and extend the notion of trees and domains as follows.

For a set of trees $\mathcal{T}$, we define the extended set $\mathcal{T}'$ by:

$$\mathcal{T}' = \{t' \mid \exists t \in \mathcal{T}, t' \text{ is obtained from } t \text{ by replacing each } v \text{ by a } v_s \text{ for some } s \in \Gamma^*\}$$

Note that given a domain D and a height h, there is an infinite number of trees of height h in D', while only a finite number in D. Fortunately, thanks to the semantics of the transducers on v_s we define below, finding the tree test sets is easier in this setting.

For any transducer τ we extend the definition of $[\![\tau]\!]$ to $\mathcal{T}'_\Sigma$ by defining $[\![\tau]\!](v_s) = s$. We naturally extend the definition of tree test set of an extended domain D' to be a set $T' \subset D'$ such that for all 1STSs τ_1 and τ_2, $[\![\tau_1]\!]_{|T'} = [\![\tau_2]\!]_{|T'}$ implies $[\![\tau_1]\!]_{|D'} = [\![\tau_2]\!]_{|D'}$. After proving the following lemma, we will state and prove the theorem on linear test sets.

▶ **Lemma 9.1.2.** *For $a, b, x, y \in \Gamma^*$, $c \neq d$ in Γ, if $acx = bcy$ and $adx = bdy$, then $a = b$.*

Proof. Either a or b is a prefix of the other. Let us suppose that $a = bk$ for some suffix $k \in \Gamma^*$. It follows that $kcx = cy$ and $kdx = dy$. If k is not empty, then k starts with c and with d, which is not possible. Hence k is empty and $a = b$. ◀

▶ **Theorem 9.1.3.** *If the domain $D = (\Sigma, Q, I, \delta)$ is such that for every $f \in \Sigma$ of arity $k > 0$, there exist trees in $t_1, \ldots, t_k \in D$ such that $f(t_1, \ldots, t_k) \in D$ and each t_i contains at least one v, then there exists a tree test set of D' of linear size $O(|\Sigma| \cdot A)$ where A is the maximal arity of a symbol of Σ.*

Proof. (Intuition) Using the trees provided in the theorem's hypothesis, we build a linear set of trees of D' where the v nodes are replaced successively by two different symbols $v_{\text{“}\#\text{”}}$ and $v_{\text{“}?\text{”}}$. Then, we prove that any two 1STSs which are equal on this set of trees, are syntactically equal. ◀

10 Implementation

Our tool (walkthrough in Section 2) is open-source and available at `https://github.com/epfl-lara/prosy`. It takes as input an ADT represented by case class definitions written

in a Scala-like syntax, and outputs a recursive printer for this ADT. For the automata constructions of Algorithm 3, we used the `brics` Java library[2].

In the walkthrough, notice that our tool gives propositions to the user so that the user does not have to enter the answers manually. The user may choose how many propositions are to be displayed (default is 9). To obtain these propositions, we use the following procedure. Remember that for each tree t for which we need to obtain the output, Algorithm 3 builds an NFA A that recognizes the set of all possible outputs for t (see Section 8). We check for the existence of an accepted word w_0 in A, and compute the intersection A_1 between A and an automaton recognizing all words except w_0. We then have two cases. Either A_1 is empty, and therefore we know the output for tree t is w_0. In that case, we do not need to interact with the user, and can continue on to the next tree. Otherwise, A_1 recognizes some word $w_1 \neq w_0$, which we display as a proposition to the user (alongside w_0). We then obtain A_2 as the intersection between A and an automaton recognizing all words except w_0 and w_1. We continue this procedure until we have 9 propositions (or whichever number the user entered), or when the intersected automaton becomes empty.

Concerning support for the String data type, we use ideas from Section 9 and reused our code from Algorithm 3 to infer outputs. Technically, we replace the String data type with an abstract class with two case classes, `foo`, and `bar`, that must be printed as "foo" and "bar" respectively. We then obtain an ADT without Strings, on which we apply the implementation of Algorithm 3 described above. We handle the Int and Boolean data types similarly, each with two different values *which are not prefix of each other* (we refer to the proof of theorem 8.1.2).

11 Evaluation

Although this work is mostly theoretical, we now depict through some benchmarks how many and which kind of questions our system is able to ask (Figure 3).

The first column is the name of the benchmark. The first two appear in Section 2 and in the examples. The third is a variation of the second where we add attributes as well, rendered "^.foo := "bar"". The fourth is the same but rendered in XML instead of tags. Note that because we do not support duplication, we need to have a finite number of tags for XML.

The four rows "binary" illustrate how the number and type of questions may vary only depending on the user's answers. We represent binary numbers as either `Empty` or `Zero(x)` or `One(x)` where `x` is a binary number. We put in parenthesis what a user willing to print `Zero(One(Zero(Zero(One(Empty)))))` would have in mind. The second and the third "discard" `Zero` when printing. The fourth one prints `Empty` as empty, `Zero(x)` as `{x}ab` and `One(x)` as `a{x}b`, which result in an ambiguity not resolved until asking a 3-digit number.

The last five rows of Figure 3 also illustrate how the number of asked questions grows linearly, whereas the number of elements in the test set grows cubically. These five rows represent a set of classes of type A taking as argument a class of type B, which themselves take as argument a class of type F. We report on the statistics by varying the number of concrete classes between 1, 2, 4, 8 and 16 (see proof of Lemma 7.2.2)

The second column is the size of the test set. For the last five rows, the test set contains a cubic number of elements. The third column is the number of answers our tool was able to

[2] http://www.brics.dk/automaton/

Name	Test set size *Total*	The output was inferred *total*	asked *total*	asked with... nothing	a hint	suggestions
Grammar (Sec. 2)	116	102	14	6	6	2
Html tags (Ex. 2, 8, 9)	35	28	7	4	2	1
Html tags+attributes	60	52	8	2	4	2
Html xml+attributes	193	179	14	5	3	6
Binary (01001x)	15	12	3	1	2	0
Binary (11x)	15	12	3	3	0	0
Binary (ababx)	15	11	4	3	0	1
Binary (01001)	15	10	5	3	0	2
Binary (aababbab)	15	9	6	3	0	3
$A_x(B_y(F_z))$ 1	3	0	3	1	2	0
$A_x(B_y(F_z))$ 2	14	8	6	3	3	0
$A_x(B_y(F_z))$ 4	84	67	17	8	4	5
$A_x(B_y(F_z))$ 8	584	552	32	19	5	8
$A_x(B_y(F_z))$ 16	4368	4305	63	32	16	15

Figure 3 Comparison of the number of questions asked for different benchmarks.

"infer" based on previously "asked" questions, whose total number is in the fourth column. The fourth column plus the third one thus equal the second one.

Columns five, six and seven decompose the fourth column into the questions which were either asked without any indication, or with a hint of type "[...]foo[...]" (because the arguments were known), or with explicit suggestions where the user just had to enter a number for the choice (see Section 10).

12 Related Work

Our approach of proactively learning transducers by example, or tree-to-string programs, can be viewed as a particular case of Programming-by-Example. Programming-by-example, also named inductive programming [?] or test-driven synthesis [37], is gaining more and more attention, notably thanks to Flash Fill in Excel 2013 [16]. Subsequent work demonstrated that these techniques could widely be applicable not only to strings, but when extracting documents [28], normalizing text [24] and number transformations [42]. However, most state-of-the-art programming-by-example techniques rely on the fact that examples are unambiguous and/or that the example provider can check the validity of the final program [6] [44] [12]. The scope of their algorithms may be larger but they do not guarantee formal result such as polynomial time or non-ambiguity, and often require the user to come up with the examples by himself. More generally, synthesizing recursive functions has recently gained an interest among computer scientists from repairing fragments [25] to very precise types [41], even by formalizing programming-by-example [13].

Recently, research has pointed out that solving ambiguities is a key to make programming by example accessible, trustful and reduce the number of errors [34][20]. The power of interaction is already well known in more statistical approaches, e.g. machine learning [45], although recent machine-learning based formatting techniques could benefit from more interaction, because they acknowledge some anomalies [36]. In [18] and even [17], the authors solve ambiguities by presenting different code snippets, obtained from synthesizing expressions

of an expected type and from other sources of information. Nonetheless, the user has to choose between hard-to-read *code snippets*. Instead of asking which transducer is correct, we ask for what is the right output. Asking sub-examples at run-time proved to be a successful strategy when synthesizing recursive functions [1]. To deal with ambiguous samples, they developed a SATURATE rule to ask for inputs covering the inferred program. In our case, however, such coverage rule still yield the ambiguity raised in example 9, leaving the chance of finding the right program to heuristics.

Researchers have investigated fundamental properties of tree-to-string or tree-to-word transducers [5], including expressiveness of even more complex classes than we consider [4], but none of them proposed a practical learning algorithm for such transducers. The situation is analogous for Macro Tree Transducers [7] [11]. Lemay [29] explores the synthesis of top-down tree-to-tree transducers using an algorithm similar to L^* for automata [6] and tree automata [8]. These learning algorithms require the user to be in possession of a set of examples that uniquely defines the top-down tree transducer. We instead are able to incrementally ask for examples which resolve ambiguities, although our transducers are single-state. There are also probabilistic tree-to-string transducers [14], but they require the use of a corpus and are not adapted to synthesizing small-size code portions with a few examples.

A Gold-style learning algorithm [27, 26, 29] was created for sequential tree-to-string transducers. It runs in polynomial-time, but has a drawback: it requires the input/output examples to form a *characteristic sample* for the transducer which is being learned. The transducer which is being learned is however not known in advance. As such, it is not clear in practice how to construct such a characteristic sample. When the input/output examples do not form a characteristic sample, the algorithm might fail, and the user of the algorithm has no indication on which input/output examples should be added to obtain a characteristic sample.

In the case when trees to be printed are programming abstract syntax trees, our work is the dual of the mixfix parsing problem [23]. Mixfix parsing takes strings to parse and the wrapping constants to print the trees, and produces the shape of the tree for each string. Our approach requires the shape of the trees and strings of some trees, and produces the wrapping constants to print the trees.

12.1 Equivalence of top-down tree-to-string transducers

Since tree test sets uniquely define the behavior of tree-to-string transducers, they can be used for checking tree-to-word transducers equivalence. Checking equivalence of sequential (order-preserving, non-duplicating) tree-to-string transducers can already be solved in polynomial time [43], even when they are duplicating, and not necessarily order-preserving [31].

It was also shown [19] that checking equivalence of deterministic top-down macro tree-to-string transducers (duplication is allowed, storing strings in registers to output them later is allowed) is decidable. Complexity-wise, this result gives a co-randomized polynomial time algorithm for linear (non-duplicating) tree-to-string transducers. This complexity result was recently improved in [10], where it was proved that checking equivalence of linear tree-to-string transducers can be done in polynomial time.

12.2 Test sets

The polynomial time algorithms of [43, 10] exploit a connection between the problem of checking equivalence of sequential top-down tree-to-string transducers and the problem of

checking equivalence of morphisms over context-free languages [43].

This latter problem was shown to be solvable in polynomial time [38, 39] using test sets. More specifically, this work shows that each context-free language L has a (finite) test set whose size is $O(n^6)$ (originally "finite" in [3, 15] and then "exponential" in [2]), where n is the size of the grammar. They also provide a lower bound on the sizes of the test sets of context-free languages, by exposing a family of grammars for which the size of the smallest test is $O(n^3)$.

As a result, when checking the equivalence of two morphisms f and g over a context-free language L, it is enough to check the equivalence on the test set of L whose size is polynomial. This result translates (as described in [43]) to checking equivalence between sequential top-down tree-to-string transducers in the following sense. When checking the equivalence of two such transducers P_1 and P_2, it is enough to do so for a finite number of trees, which correspond to the test set of a particular context-free language. This language can be constructed from P_1 and P_2 in time $|P_1||P_2|$.

▶ Remark. Theorem 7.1.1 also helps improve the bound for checking equivalence of 1STS with states, using the known reduction from equivalence of 1STS with states to morphisms equivalence over a context-free language (reduction similar to Lemma 5.1.3, see [43, 26]).

13 Conclusion

We have presented a synthesis algorithms that can learn from examples tree-to-string functions with the input tree as the only argument. This includes functions such as pretty printers. Crucially, our algorithm can automatically construct a sufficient finite set of input trees, resulting in an interactive synthesis approach that in which the user needs to answer only a linear number of questions in the grammar size. Furthermore, the interaction process driven by our algorithm guarantees that there is no ambiguity: the recursive function of the expected form is unique for a given set of input-output examples. Moreover, we have analyzed the structure of word equations that the algorithm needs to solve and shown that they have a special structure allowing them to be solved in deterministic polynomial time, which results in overall polynomial running time of our synthesizer. Our results make a case that providing tests for tree-to-string functions is a viable alternative to writing the recursive programs directly, an alternative that is particularly appealing for non-expert users.

References

1. Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *International Conference on Computer Aided Verification*, 2013.
2. Jürgen Albert, Karel Culik, and Juhani Karhumäki. Test sets for context free languages and algebraic systems of equations over a free monoid. *Information and Control*, 52(2):172–186, 1982.
3. Michael H Albert and J Lawrence. A proof of Ehrenfeucht's conjecture. *Theoretical Computer Science*, 41:121–123, 1985.
4. Rajeev Alur and Loris D'Antoni. Streaming tree transducers. In *Automata, Languages, and Programming*, pages 42–53. Springer, 2012.
5. Rajeev Alur and Pavol Černý. Expressiveness of streaming string transducers. In Kamal Lodaya and Meena Mahajan, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2010, December 15-18, 2010, Chennai, India*, volume 8 of *LIPIcs*, pages 1–12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

6 Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, pages 87–106, 1987.

7 Patrick Bahr and Laurence E. Day. Programming macro tree transducers. In *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming*, pages 61–72. ACM, 2013.

8 Jérôme Besombes and Jean-Yves Marion. Learning tree languages from positive examples and membership queries. 2004.

9 Adrien Boiret. Normal Form on Linear Tree-to-word Transducers. In *10th International Conference on Language and Automata Theory and Applications*, 2016.

10 Adrien Boiret and Raphaela Palenta. Deciding equivalence of linear tree-to-word transducers in polynomial time. *CoRR*, abs/1606.03758, 2016.

11 Joost Engelfriet and Sebastian Maneth. Output string languages of compositions of deterministic macro tree transducers. *Journal of Computer and System Sciences*, 64(2):350–395, 2002.

12 John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing Data Structure Transformations from Input-output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 229–239, New York, NY, USA, 2015. ACM.

13 Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, 2016.

14 Jonathan Graehl and Kevin Knight. Training tree transducers. Technical report, DTIC Document, 2004.

15 Victor Sergeevich Guba. Equivalence of infinite systems of equations in free groups and semigroups to finite subsystems. *Mathematical Notes*, 40(3):688–690, 1986.

16 Sumit Gulwani. Synthesis from Examples. In *WAMBSE Special Issue, Infosys Labs Briefings*, volume 10(2), 2012.

17 Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. 2013.

18 Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. Interactive Synthesis of Code Snippets. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 418–423, Berlin, Heidelberg, 2011. Springer-Verlag.

19 Helmut Seidl, Sebastian Maneth, and Gregor Kemper. Equivalence of deterministic top-down tree-to-string transducers is decidable. In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 943–962. IEEE, 2015.

20 Thibaud Hottelier, Ras Bodik, and Kimiko Ryokai. Programming by manipulation for layout. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*, 2014.

21 Patrik Jansson. *Functional Polytypic Programming.* PhD thesis, Institutionen för datavetenska, Göteborg : Chalmers University of Technology, 2000.

22 Artur Jeż. Word equations in linear space. *arXiv preprint arXiv:1702.00736*, 2017.

23 Jean-Pierre Jouannaud, Claude Kirchner, Hélène Kirchner, and Aristide Megrelis. Programming with equalities, subsorts, overloading, and parametrization in OBJ. *The Journal of Logic Programming*, 12(3):257–279, 1992.

24 Dileep Kini and Sumit Gulwani. FlashNormalize: Programming by Examples for Text Normalization.

25 Manos Koukoutos, Etienne Kneuss, and Viktor Kuncak. An update on deductive synthesis and repair in the leon tool. 2016.

26 Grégoire Laurence. *Normalisation et Apprentissage de Transductions d'Arbres en Mots.* PhD thesis, Université des Sciences et Technologie de Lille-Lille I, 2014.

27 Grégoire Laurence, Aurélien Lemay, Joachim Niehren, Sławek Staworko, and Marc Tommasi. Learning sequential tree-to-word transducers. In *International Conference on Language and Automata Theory and Applications*, pages 490–502. Springer, 2014.

28 Vu Le and Sumit Gulwani. FlashExtract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 55. ACM, 2014.

29 Aurelien Lemay, Sebastian Maneth, and Joachim Niehren. A learning algorithm for top-down XML transformations. PODS '10, pages 285–296, New York, NY, USA, 2010. ACM.

30 M Lothaire. *Combinatorics on words*, volume 17. Cambridge University Press, 1997.

31 Sebastian Maneth and Helmut Seidl. Deciding equivalence of top-down XML transformations in polynomial time. In *PLAN-X*, pages 73–79, 2007.

32 Mikaël Mayer and Jad Hamza. Optimal test sets for context-free languages. *CoRR*, abs/1611.06703, 2016. URL: `http://arxiv.org/abs/1611.06703`.

33 Mikaël Mayer, Jad Hamza, and Viktor Kuncak. Polynomial-time proactive synthesis of tree-to-string functions from examples. *CoRR*, abs/1701.04288, 2017. URL: `http://arxiv.org/abs/1701.04288`.

34 Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Alex Polozov, Rishabh Singh, Ben Zorn, and Sumit Gulwani. User interaction models for disambiguation in programming by example. In *28th ACM User Interface Software and Technology Symposium*, 2015.

35 Heather Miller, Philipp Haller, Eugene Burmako, and Martin Odersky. Instant pickles: generating object-oriented pickler combinators for fast and extensible serialization. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 183–202, 2013.

36 Terence Parr and Jurgen Vinju. Towards a universal code formatter through machine learning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 137–151. ACM, 2016.

37 Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 43. ACM, 2014.

38 Wojciech Plandowski. Testing equivalence of morphisms on context-free languages. In *European Symposium on Algorithms*, pages 460–470. Springer, 1994.

39 Wojciech Plandowski. *The complexity of the morphism equivalence problem for context-free languages*. PhD thesis, Department of Mathematics, Informatics, and Mechanics, Warsaw University, 1995.

40 Wojciech Plandowski. Satisfiability of word equations with constants is in PSPACE. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 495–500. IEEE, 1999.

41 Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, 2016.

42 Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *Proc. of the 24th CAV conference*, pages 634–651, Berlin, Heidelberg, 2012. Springer-Verlag.

43 Sławomir Staworko, Grégoire Laurence, Aurélien Lemay, and Joachim Niehren. Equivalence of deterministic nested word to word transducers. In *International Symposium on Fundamentals of Computation Theory*, pages 310–322. Springer, 2009.

44 Kuat Yessenov, Shubham Tulsiani, Aditya Menon, Robert C. Miller, Sumit Gulwani, Butler Lampson, and Adam Kalai. A colorful approach to text processing by example. pages 495–504. ACM, 2013.

45 Chicheng Zhang and Kamalika Chaudhuri. Active learning from weak and strong labelers. In *Advances in Neural Information Processing Systems*, 2015.