Inference of Tree Data Structure Invariant based on Language Identification from Samples

Naoshi Tabuchi, Naoki Kobayashi, and Hiroshi Unno

Tohoku University

Abstract. We propose a method to infer invariants of tree data structures for higher-order functional programs. The method uses a machine learning technique for identifying a regular tree language from samples. The proposed method is combined with a recent technique for program verification based on higher-order model checking, yielding a fully-automated verification method for higher-order, tree-processing functional programs. We have implemented the prototype verifier and conducted preliminary experiments.

1 Introduction

Finding good data invariants is often a key to the success of program verification. For example, in Hoare logic, loop invariants allow us to reduce the problem of proving a Hoare triple to that of proving the verification conditions. In type systems, types express certain data invariants, and type checking, (i.e., checking whether a program annotated with types is well-typed) is often much easier than type inference.

In this paper, we are interested in the problem of finding invariants of tree data structures, for a higher-order, tree-processing functional program. By the invariant, we mean a regular tree language that overapproximates the set of tree data that an expression may evaluate to. An immediate application of the tree invariant inference is Unno et al.'s recent work on verification of higher-order tree processing programs [21]. They developed a semi-automated method for proving that a given higher-order functional program satisfies given input and output specifications, *provided that the program is annotated with certain invariants on tree data.* For example, consider the following program, where ℓ_1 and ℓ_2 are labels to identify subterms:

$$\begin{array}{rl}
\mathit{Reverse}\ x = & \textbf{case}\ x\ \textbf{of}\ \mathtt{e} \Rightarrow \mathtt{e} \\
& \quad \mid \mathtt{a}\ x \Rightarrow \mathit{Append}\ ((\mathit{Reverse}\ x)^{\ell_1})\ (\mathtt{a}\ \mathtt{e}) \\
& \quad \mid \mathtt{b}\ x \Rightarrow \mathit{Append}\ ((\mathit{Reverse}\ x)^{\ell_2})\ (\mathtt{b}\ \mathtt{e}) \\
\mathit{Append}\ x\ y = & \textbf{case}\ x\ \textbf{of}\ \mathtt{e} \Rightarrow y \\
& \quad \mid \mathtt{a}\ x \Rightarrow \mathtt{a}\ (\mathit{Append}\ x\ y) \\
& \quad \mid \mathtt{b}\ x \Rightarrow \mathtt{b}\ (\mathit{Append}\ x\ y)
\end{array}$$

The program manipulates tree data constructed from a tree constant `e` and unary constructors `a` and `b`. The function *Reverse* takes a tree and returns the tree obtained by reversing the sequence of `a` and `b`. For example, *Reverse*(`a`(`a`(`b`(`e`)))) returns $\mathtt{b(a(a(e)))}$. Suppose that we wish to verify that, given an element of $\mathtt{a}^*\mathtt{b}^*\mathtt{e}$[1], *Reverse* returns an element of $\mathtt{b}^*\mathtt{a}^*\mathtt{e}$. By using Unno et al.'s method, it suffices to declare invariants that the expression labeled by ℓ_1 evaluates to an element of $\mathtt{b}^*\mathtt{a}^*\mathtt{e}$, and the expression labeled by ℓ_2 evaluates to an element of $\mathtt{b}^*\mathtt{e}$. Thus, the combination of an algorithm to find such invariants with Unno et al.'s method provides a fully-automated verification method for higher-order tree-processing programs. Similar applications would be possible also to other verification methods for functional programs that require certain invariant annotations, such as refinement type checking [4] and a combination of verification condition generation and decision procedures for data structures [19].

The standard approach to finding invariants is to use static analyses. For tree data structure invariants, Jones and Andersen [8] developed a kind of flow analysis to approximate the value of each variable by a regular tree grammar. Kochems and Ong [14] extended it to obtain more precise approximations by using indexed grammars. Minamide [16] developed a method to infer the output of a string-processing program as a context-free grammar. A limitation of the static analysis approaches is that they are often not precise enough for the verification methods mentioned above. For example, even for the simple program above, we need context-sensitivity, to infer that *Reverse* x labeled with ℓ_1 returns an element of $\mathtt{b}^*\mathtt{a}^*\mathtt{e}$ while the same expression labeled with ℓ_2 returns an element of $\mathtt{b}^*\mathtt{e}$. If a context-insensitive analysis is used for invariant inference and is combined with Unno et al.'s method, which is based on a kind of higher-order model checking [13], then the advantage of the preciseness of higher-order model checking is lost. Another limitation, related to the above point, is that the static analysis approaches mentioned above are a kind of *forward* inference, which propagates the input specification but does not take into account the output specification. Thus, it is difficult to adjust the precision and efficiency of the analysis based on the final verification goal. For example, for the program above, if the output specification is replaced with $(\mathtt{a}|\mathtt{b})^*\mathtt{e}$, then the simple invariant $(\mathtt{a}|\mathtt{b})^*\mathtt{e}$ would suffice for ℓ_1 and ℓ_2, but the static analyses may spend much time in vain for finding more precise invariants.

In the present paper, we propose an alternative approach to invariant inference, based on a technique for learning regular tree languages from samples. We use Besombes and Marion's algorithm [2] that, given a finite set of (positive) samples and an oracle to answer a membership query, infers a regular tree language. We assume that the overall verification goal is, given a program e and regular tree languages $\mathcal{L}_I$ and $\mathcal{L}_O$ as the input and output specifications, to prove that e conforms to the input and output specifications, i.e., for any tree t that belongs to $\mathcal{L}_I$, the value of $e\,t$ belongs to $\mathcal{L}_O$. The goal of invariant inference is then to find an invariant $\mathcal{L}$ for the subterm of M labeled by ℓ that is *sound* in the sense that the subterm can only evaluate to an element of $\mathcal{L}$, and *strong* enough to prove that the value of $e\,t$ belongs to $\mathcal{L}_O$. (This is analogous to the problem of, given a Hoare triple, finding loop invariants so that the

[1] We often identify linear trees with sequences, and use a regular expression for a set of linear trees.

Hoare triple can be reduced to verification conditions.) Our method proceeds as follows. We first pick a finite set of elements of $\mathcal{L}_I$ randomly, and run the program, and check whether the outputs belong to $\mathcal{L}_O$. If so, we collect a set of pairs $(t_i, E_i)(i \in \{1, \ldots, n\})$ such that the program evaluates to $E_i[t_i^\ell]$. We then use $t_1, \ldots, t_n$ as positive samples, and, as a membership oracle, the function that takes a tree t' and returns whether the value of $E_i[t']$ belongs to $\mathcal{L}_O$ for every $i \in \{1, \ldots, n\}$. If the inferred language is not a valid invariant, we expand the set of inputs and repeat the procedure, until either a counterexample against the input/output specifications is found, or a valid invariant is found. Our procedure does not terminate in general, but with certain assumptions, the procedure can eventually find a valid invariant.

Advantages of our invariant inference method include: (i) it is a combination of forward and backward analyses, taking both the input and output specifications into account, and (ii) it is largely independent of the target language; it does not matter whether the target is higher-order/first-order, or pure/impure, as long as we have a sound procedure to check the correctness of inferred invariants.

We have implemented the invariant inference algorithm and combined it with Unno et al.'s verification method. According to preliminary experiments, the algorithm works reasonably well, especially for linear trees (i.e., list-like data structures).

The rest of the paper is organized as follows: Section 2 presents preliminary definitions and notational conventions. Section 3 introduces a minimal functional calculus as a target language. Section 4 reviews language identification techniques, the back-end of our method. Section 5 formalizes the procedure of invariant inference. Section 6 reports implementation and experiments. Section 7 compares our method with related work and Section 8 concludes.

2 Preliminaries

This section introduces (bottom-up) tree automata, which are used for expressing the specifications and invariants of programs.

We write $\mathrm{dom}(f)$ for the domain of a mapping f. If X is a set, X^* denotes the set of finite sequences of elements of X. We sometimes use the standard regular expressions to denote a set of sequences, e.g., $(\mathtt{a} \mid \mathtt{b})$ for $\{\mathtt{a}, \mathtt{b}\}$, and $(\mathtt{a} \mid \mathtt{b})^*$ for $\{\mathtt{a}, \mathtt{b}\}^*$. We write ϵ for the empty sequence. We write $s_1 \cdot s_2$ for the concatenation of sequences s_1 and s_2. A (possibly empty) sequence $v_1 \ldots v_n$ is often abbreviated to $\widetilde{v}$ and $|\widetilde{v}|$ denotes the length of $\widetilde{v}$.

A *ranked alphabet* Σ is a mapping from a finite set of symbols, called *terminal symbols*, to non-negative integers. For each symbol $a \in \mathrm{dom}(\Sigma)$, $\Sigma(a)$ denotes the *arity* of a. The set $\mathcal{T}_\Sigma$ of *(finite) Σ-labeled ranked trees* is inductively defined by: $a\ t_1\ \cdots\ t_n \in \mathcal{T}_\Sigma$ if $t_1, \ldots, t_n \in \mathcal{T}_\Sigma \wedge \Sigma(a) = n$. In particular, $a \in \mathcal{T}_\Sigma$ if $\Sigma(a) = 0$. We let metavariable t range over $\mathcal{T}_\Sigma$. A *tree context* C is a term obtained by replacing a subterm of a tree with a hole $[\,]$. When C is a tree context and t is a tree, we write $C[t]$ for the tree obtained by replacing $[\,]$ with t. We write $\mathcal{C}_\Sigma$ for the set of tree contexts.

A (bottom-up) *finite tree automaton* over Σ is a quadruple $\mathcal{M} = (Q, \Sigma, Q_f, \Delta)$ where Q is a finite set of *states*, $Q_f \subseteq Q$ is a set of *final states* and Δ is a set of *transition rules*. A transition rule is a rewrite rule of the form $a\ q_1\ \cdots\ q_n \to q$ where $q, q_1, \ldots, q_n \in Q$ and $\Sigma(a) = n$. We define the binary relation $\longrightarrow_\mathcal{M}$ on $\mathcal{T}_{\Sigma \cup \{q \mapsto 0 | q \in Q\}}$ by: $C[a\ q_1\ \cdots\ q_n] \longrightarrow_\mathcal{M} C[q] \iff a\ q_1\ \cdots\ q_n \to q \in \Delta$ and C is a tree context. A tree t is *accepted by* $\mathcal{M}$ iff $t \longrightarrow^*_\mathcal{M} q_f$ for some $q_f \in Q_f$. We write $\mathcal{L}(\mathcal{M})$ for the set of Σ-labeled ranked trees accepted by $\mathcal{M}$ and call it the *language* of $\mathcal{M}$. An automaton is deterministic if for every $a, q_1, \ldots, q_n$, there is at most one transition rule of the form $a\,q_1\ \cdots\ q_n \to q$. This paper only concerns deterministic bottom-up tree automata.

3 Target Language and Invariant Inference Problem

Our target language is an untyped, call-by-value λ-calculus with recursion and tree constructors/destructors; as mentioned in Section 1, our invariant inference technique is largely independent of the choice of the language. We fix a ranked alphabet Σ, and use its elements as tree constructors.

Definition 1 (Expressions). *The set of expressions, ranged over by e, is given by:*

$$\begin{aligned} e\ (\textit{expressions}) ::= {} & x \mid (\mathbf{fix}\ f, x, e) \mid e_1\ e_2 \\ & \mid a \mid \mathbf{case}\ e\ \mathbf{of}\ \{a_i\ \widetilde{y_i} \Rightarrow e_i\}_{i=1}^k \mid e^\ell \end{aligned}$$

Here, e^ℓ denotes a labeled expression, for which we wish to infer invariants. We assume that the whole expression (called a program) contains only one label ℓ, just for the sake of simplicity; the actual implementation discussed in Section 6 handles a program with multiple labels.

The first line of the definition shows the standard constructs of λ-calculus: $(\mathbf{fix}\ f, x, e)$ denotes a (possibly) recursive function. We write $\lambda x.e$ when f is not free in e. We sometimes use the **letrec**-binding of the form **letrec** $f\ x\,\widetilde{y} = e_1$ **in** e_2 as a syntax sugar for $(\lambda f.e_2)(\mathbf{fix} f, x, \lambda\widetilde{y}.e_1)$. The expression $a (\in \mathit{dom}(\Sigma))$ denotes a tree constructor. The expression **case** e **of** $\{a_i\ \widetilde{y_i} \Rightarrow e_i\}$ evaluates to $[\widetilde{t}/\widetilde{y_i}]e_i$ if the value of e is of the form $a_i\ \widetilde{t}$, and gets stuck otherwise. The operational semantics is shown in Figure 1. The evaluation gets stuck when there is no applicable rule, e.g. because of type errors.

$$\begin{array}{lll} v & \text{(values)} & ::= a\, v_1 \cdots v_k \mid (\mathbf{fix}\ f, x, e) \\ E & \text{(evaluation contexts)} & ::= [\,] \mid E[[\,]^\ell] \mid E\ e \mid v\ E \mid \mathbf{case}\ E\ \mathbf{of}\ \{a_i\ \widetilde{y_i} \Rightarrow e_i\}_{i=1}^{k} \end{array}$$

$$\frac{}{E[(\mathbf{fix}\ f, x, e)\ v] \longrightarrow E[[(\mathbf{fix}\ f, x, e)/f, v/x]e]} \qquad \frac{}{E[t^\ell] \longrightarrow E[t]}$$

$$\frac{|\widetilde{t}| = |\widetilde{y_i}|}{E[\mathbf{case}\ a_i\ \widetilde{t}\ \mathbf{of}\ \{a_i\ \widetilde{y_i} \Rightarrow e_i\}_{i=1}^{n}] \longrightarrow E[[\widetilde{t}/\widetilde{y_i}]e_i]}$$

Fig. 1. Operational Semantics

Example 1. We use the following program e_{rev}, discussed in Section 1, as a running example:

$$\begin{array}{lll} \mathbf{letrec}\ \mathit{append}\ x\ y = & \mathbf{case}\ x\ \mathbf{of}\ \mathtt{e} \Rightarrow y \\ & \mid \mathtt{a}\ x \Rightarrow \mathtt{a}\ (\mathit{append}\ x\ y) \\ & \mid \mathtt{b}\ x \Rightarrow \mathtt{b}\ (\mathit{append}\ x\ y) \\ \mathbf{in} & \\ \mathbf{letrec}\ \mathit{reverse}\ x = & \mathbf{case}\ x\ \mathbf{of}\ \mathtt{e} \Rightarrow \mathtt{e} \\ & \mid \mathtt{a}\ x \Rightarrow \mathit{append}\ (\mathit{reverse}\ x)^\ell\ (\mathtt{a}\ \mathtt{e}) \\ & \mid \mathtt{b}\ x \Rightarrow \mathit{append}\ (\mathit{reverse}\ x)\ (\mathtt{b}\ \mathtt{e}) \\ \mathbf{in}\ \mathit{reverse} & \end{array}$$

As mentioned in Section 1, given (i) a program e that takes a tree as an input and returns a tree, and that has a labeled subexpression e_0, and (ii) regular tree languages $\mathcal{L}_I$ and $\mathcal{L}_O$ as the input and output specifications, the goal of our invariant inference is to infer an invariant $\mathcal{L}$ such that (1) $\mathcal{L}$ is a sound approximation of the value of e_0 and (2) $\mathcal{L}$ is small enough to prove that the value of $e\,t$ belongs to $\mathcal{L}_O$ whenever t belongs to $\mathcal{L}_I$.

To formalize the two conditions above, we introduce the extended reduction relation $\longrightarrow_{\mathcal{L}}$ shown in Figure 2. Rule SWAP allows the value of a labeled expression to be replaced by an arbitrary element of the invariant $\mathcal{L}$. Given a program e and input/output specifications $\mathcal{L}_I$ and $\mathcal{L}_O$, we say that a tree language $\mathcal{L}$ is a *forward invariant* if $\mathcal{L} \supseteq \{t \mid \exists t_I \in \mathcal{L}_I.(e\,t_I \longrightarrow_{\mathcal{L}}^{*} E[t^\ell])\}$ holds. Here, we use $\longrightarrow_{\mathcal{L}}^{*}$ instead of $\longrightarrow^{*}$ to ensure that, when the label ℓ occurs inside a recursion (or a loop), the forward invariant is general enough for inductively showing that it is indeed an invariant. For example, consider the following function:[2]

$$\mathbf{letrec}\ f\ x\ y = (\mathbf{if}\ y = 0\ \mathbf{then}\ x\ \mathbf{else}\ (f\ x\ (y-1)) + (f\ x\ (y-1)))^\ell$$

If x ranges over the set $\{0, 1\}$, then the set of values of the body of f is $\{0, 1, 2, 4, 8, \ldots\}$, but it is not large enough to prove inductively that the return value of f belongs to $\{0, 1, 2, 4, 8, \ldots\}$. In our definition, the set of all non-negative integers is a forward invariant but $\{0, 1, 2, 4, 8, \ldots\}$ is not.

[2] We use integers for the sake of simplicity.

$$\frac{e \longrightarrow e'}{e \longrightarrow_{\mathcal{L}} e'}\ \textsc{Base} \qquad \frac{t' \in \mathcal{L}}{E[t^{\ell}] \longrightarrow_{\mathcal{L}} E[t']}\ \textsc{Swap}$$

Fig. 2. Extended Operational Semantics

We write $\mathcal{L}^f_{e,\mathcal{L}_I,\mathcal{L}_O}$ for the least foward invariant, i.e., $\bigcap\{\mathcal{L} \mid \mathcal{L} \supseteq \{t \mid \exists t_I \in \mathcal{L}_I.(e\, t_I \longrightarrow^* E[t^{\ell}])\}\}$. We just write $\mathcal{L}^f$ for the least foward invariant if $e, \mathcal{L}_I, \mathcal{L}_O$ are clear from the context. We say that a tree language $\mathcal{L}$ is a *backward invariant* if

$$\forall t \in \mathcal{L}.\forall t_I \in \mathcal{L}_I.\forall E, t', t''.(e\, t_I \longrightarrow^*_{\mathcal{L}^f} E[t'] \wedge E[t] \longrightarrow^* t'' \implies t'' \in \mathcal{L}_O)$$

holds, and write $\mathcal{L}^b_{e,\mathcal{L}_I,\mathcal{L}_O}$ for the largest backward invariant.

Definition 2 (Invariant Inference Problem). Let e be a program and $\mathcal{L}_I, \mathcal{L}_O$ regular tree languages. We say that $\mathcal{L}$ is a *valid invariant* and write $\models (M, \mathcal{L}_I, \mathcal{L}_O, \mathcal{L})$ if $\mathcal{L}^f_{e,\mathcal{L}_I,\mathcal{L}_O} \subseteq \mathcal{L} \subseteq \mathcal{L}^b_{e,\mathcal{L}_I,\mathcal{L}_O}$. An *invariant inference problem* $\mathit{Inf}(e, \mathcal{L}_I, \mathcal{L}_O)$ is the problem of checking whether there exists a regular tree language $\mathcal{L}$ such that $\models (e, \mathcal{L}_I, \mathcal{L}_O, \mathcal{L})$ holds, and if so, returns $\mathcal{L}$.

Note that the problem above is undecidable in general. In Section 5, we assume the existence of a sound (but not necessarily complete) procedure to *check* the condition $\models (e, \mathcal{L}_I, \mathcal{L}_O, \mathcal{L})$, and give an (incomplete) procedure for the above problem.

Remark 1. In the above formalization, a valid invariant $\mathcal{L}$ may not be a forward invariant. Depending on the verification method, we may need to require that $\mathcal{L}$ is also a forward invariant.

Example 2. Recall the program e_{rev} in Example 1. The invariant inference problem discussed in Section 1 is formalized as $\mathit{Inf}(e_{rev}, \mathtt{a}^*\mathtt{b}^*\mathtt{e}, \mathtt{b}^*\mathtt{a}^*\mathtt{e})$, and a solution is $\mathtt{b}^*\mathtt{a}^*\mathtt{e}$. Unno et al.'s method [21] serves as an oracle to check the condition $\models (e_{rev}, \mathtt{a}^*\mathtt{b}^*\mathtt{e}, \mathtt{b}^*\mathtt{a}^*\mathtt{e}, \mathcal{L})$.[3]

4 Algorithm for Identifying Regular Tree Languages from Samples

This section reviews Besombes and Marion's algorithm [2], called Altex, for learning regular tree languages from positive samples and membership queries. We adopt this algorithm as a core machinery of our invariant inference procedure. The algorithm Altex ($\in 2^{\mathcal{T}_\Sigma} \rightarrow (\mathcal{T}_\Sigma \rightarrow \{0,1\}) \rightarrow$ TreeAutomata) takes a finite set $\mathcal{E}$ of positive sample trees and an oracle o to answer membership queries,

[3] Actually, to use Unno et al.'s method, we also need an invariant for the other occurences of *reverse x* in the body of *reverse*. It is easy to extend our invariant inference algorithm to infer invariants for multiple labels.

and returns a (bottom-up) tree automaton whose language includes the given samples. The oracle o should output 1 if the given tree is a member of the language to be learned, and 0 otherwise. In our settings, (continuations of) the program itself acts as the oracle, as we shall see later. In the sequel, we write $\mathcal{S}(\mathcal{E})$ for the set of subtrees of trees in $\mathcal{E}$.

We briefly sketch how ALTEX works. The algorithm is mostly treated as a blackbox in this paper, so the reader may safely skip this paragraph for the first reading. Given a set $\mathcal{E}$ of samples and a membership oracle o, ALTEX constructs an *observation table*, where each row is indexed by a subtree of an element of $\mathcal{E}$ and each column by a tree context. A cell $\mathbb{T}(t, C)$ of the table $\mathbb{T}$ is filled by $o(C[t])$. Let $\mathbf{row}(t)$ be the sequence of $0, 1$ in the row t. Then, ALTEX outputs the following automaton $\mathcal{M}_{\mathbb{T}} = (Q, \Sigma, Q_f, \Delta)$ induced by $\mathbb{T}$ as the result of learning: $Q = \{q_{\mathbf{row}(t)} \mid t \in \mathcal{S}(\mathcal{E})\}$, $Q_f = \{q_{\mathbf{row}(t)} \mid o(t) = 1\}$, $\Delta = \{a\ q_{\mathbf{row}(t_1)}\ \cdots\ q_{\mathbf{row}(t_n)} \rightarrow q_{\mathbf{row}(a\ t_1\ \cdots\ t_n)} \mid a\ t_1\ \cdots\ t_n \in \mathcal{S}(\mathcal{E})\}$. At the initial phase of the learning, the set of tree contexts $\{C \mid \exists t \in \mathcal{E}.t = C[t'] \text{ for some } t'\}$ generated from the sample set is used as the column set of the observation table. During the learning, new tree contexts are added to the table until $\mathcal{M}_{\mathbb{T}}$ becomes a deterministic automaton.

We now review the property of ALTEX. The following is a sufficient condition for the sample set.

Definition 3 (Representative Sample). *Let $\mathcal{L}$ be a regular tree language, $\mathcal{E} \subseteq \mathcal{L}$ and $\mathcal{M}$ the minimal automaton for $\mathcal{L}$. A finite set $\mathcal{E}\,(\subseteq \mathcal{L})$ is called a* representative sample *of $\mathcal{L}$ if for each transition $a\ q_1\ \cdots\ q_n \rightarrow q$ of $\mathcal{M}$, there exists a tree $a\ t_1\ \cdots\ t_n \in \mathcal{S}(\mathcal{E})$ such that $t_i \longrightarrow^*_{\mathcal{M}} q_i$ $(\forall 1 \leq i \leq n)$.*

Intuitively, $\mathcal{E}$ is a representative sample if all the transition rules of $\mathcal{M}$ are used for accepting trees in $\mathcal{E}$.

The following is the key property of ALTEX, which states that learning of a regular language always succeeds if the sample set $\mathcal{E}$ is large enough, i.e., if it is a representative sample.

Theorem 1 (Besombes and Marion [2]). *Let $\mathcal{E}$ be a finite subset of a regular tree language $\mathcal{L}$ and o the oracle for $\mathcal{L}$. Then ALTEX$(\mathcal{E}, o)$ terminates and returns a deterministic (bottom-up) tree automaton $\mathcal{M}$. Further, if $\mathcal{E}$ is a representative sample of $\mathcal{L}$, $\mathcal{M}$ is the minimal (deterministic) automaton for $\mathcal{L}$.*

Example 3. Let $\mathcal{L} = \mathtt{b}^*\mathtt{a}^*\mathtt{e}$, and $\mathcal{E} = \{\mathtt{b(b(a(e)))}\}$. ALTEX constructs the following observation table.

	[ ]	b([ ])	b(b([ ]))	b(b(a([ ])))
e	1	1	1	1
a(e)	1	1	1	1
b(a(e))	1	1	1	0
b(b(a(e)))	1	1	1	0

The automaton constructed from the table is $\mathcal{M} = (Q, \{\mathtt{a}, \mathtt{b}, \mathtt{e}\}, Q, \Delta)$ where $Q = \{q_{1110}, q_{1111}\}$ and $\Delta = \{\mathtt{e} \rightarrow q_{1111}, \mathtt{a}\, q_{1111} \rightarrow q_{1111}, \mathtt{b}\, q_{1111} \rightarrow q_{1110}, \mathtt{b}\, q_{1110} \rightarrow q_{1110}\}$. Note that $\mathcal{L}(\mathcal{M}) = \mathtt{b}^*\mathtt{a}^*\mathtt{e}$.

5 Invariant Inference

We now present our procedure for invariant inference, using ALTEX as a backend. The procedure roughly proceeds as follows. We first run a given program for some inputs (chosen randomly but in a fair manner) and collect the sets $\mathcal{U}$ of values and the set $\mathcal{K}$ of *continuations* of the labeled expression. We then use $\mathcal{U}$ as a sample set for the invariant, and the function $\lambda x.$"for every $k \in \mathcal{K}$, $k\,x$ returns a valid output" as (an approximation of) the membership oracle for the invariant, and invoke ALTEX. We then check whether the language output by ALTEX is a valid invariant by using an outside procedure. If the output is not a valid invariant, then we run the program for more inputs, to collect more samples and refine the approximation of the membership oracle.

Below we first extend the operational semantics to collect values and continuations of the labeled expression in Section 5.1. We then describe the procedure and discuss its properties in Sections 5.2 and 5.3.

5.1 Extended Operational Semantics

In order to collect values and continuations, we extend the reduction relation $e \longrightarrow e'$ to $(\mathcal{U}, \mathcal{K}, e) \longrightarrow (\mathcal{U}', \mathcal{K}', e')$. It is defined by the following rules.

$$\frac{e \longrightarrow e'}{(\mathcal{U}, \mathcal{K}, e) \longrightarrow (\mathcal{U}, \mathcal{K}, e')}\ \textsc{Red} \qquad \frac{E \in \mathcal{K}, t \in \mathcal{U}}{(\mathcal{U}, \mathcal{K}, e) \longrightarrow (\mathcal{U}, \mathcal{K}, E[t])}\ \textsc{Switch}$$

$$\frac{}{(\mathcal{U}, \mathcal{K}, E[t^\ell]) \longrightarrow (\mathcal{U} \cup \{t\}, \mathcal{K} \cup \{E\}, E[t])}\ \textsc{Memorize}$$

Note that the reduction relation $(\mathcal{U}, \mathcal{K}, e) \longrightarrow (\mathcal{U}', \mathcal{K}', e')$ is non-deterministic. In a state $(\mathcal{U}, \mathcal{K}, e)$, e can be reduced normally by using RED, or a pair of an evaluation context E and a tree t is picked up from $\mathcal{K}$ and $\mathcal{U}$ respectively, and $E[t]$ is evaluated. When a labeled expression is evaluated to a tree value t, the tree t and its context E are memorized by rule MEMORIZE.

5.2 Invariant Inference Procedure

Figure 3 shows our procedure for invariant inference. The procedure takes a program e, an input specification $\mathcal{L}_I$ and an output specification $\mathcal{L}_O$, and returns an invariant $\mathcal{L}$ satisfying $\mathit{Verify}(e, \mathcal{L}_I, \mathcal{L}_O, \mathcal{L}) = \mathbf{true}$ if such an $\mathcal{L}$ is found. On line 4, a sample input is chosen from $\mathcal{L}_I$. Here, we assume that $\mathit{Enum}_{\mathcal{L}_I}(i)$ is a surjective mapping from the set of positive integers to $\mathcal{L}_I$, so that every tree in $\mathcal{L}_I$ is eventually chosen. On line 5, *Eval* is a (non-deterministic) procedure that returns a triple $(\mathcal{U}_i, \mathcal{K}_i, t')$ such that $(\mathcal{U}_{i-1}, \mathcal{K}_{i-1}, e\ t_i) \longrightarrow^* (\mathcal{U}_i, \mathcal{K}_i, t')$. If the tree t' does not conform to the output specification, the procedure is aborted on line 6. Otherwise, an (approximation of) membership oracle is constructed on line 7. It returns 1 (i.e., true) if it is provable (in some decidable logic) that $E[t]$ evaluates to a valid tree for every $E \in \mathcal{K}_i$. On line 8, ALTEX is used to infer an invariant candidate. On line 9, we assume that *Verify* is a sound procedure to check whether $\models (e, \mathcal{L}_I, \mathcal{L}_O, \mathcal{L})$ holds. If *Verify* returns $\mathcal{L}$, then the inference succeeds and $\mathcal{L}$ is returned as a witness. Otherwise, the procedure goes back to line 3 and iterates for next i.

```
1 : $\mathit{InferInvariants}(e, \mathcal{L}_I, \mathcal{L}_O) =$
2 :   let $(\mathcal{U}_0, \mathcal{K}_0) = (\emptyset, \emptyset)$ in
3 :   for $i \in \{1, 2, \ldots\}$ do begin
4 :     $t_i := \mathit{Enum}_{\mathcal{L}_I}(i)$;
5 :     let $(\mathcal{U}_i, \mathcal{K}_i, t') = \mathit{Eval}(\mathcal{U}_{i-1}, \mathcal{K}_{i-1}, e\ t_i)$ in // Evaluate the program with $t_i$
6 :     if $t' \notin \mathcal{L}_O$ then (output "There is no valid invariant"; abort);
7 :     let $o_{\mathcal{K}_i} = \lambda t.$if $\vdash \forall E \in \mathcal{K}_i.\forall t'.(E[t] \longrightarrow^* t' \implies t' \in \mathcal{L}_O)$ then 1 else 0 in
8 :     let $\mathcal{L} = \mathcal{L}(\mathrm{ALTEX}(\mathcal{U}_i, o_{\mathcal{K}_i}))$ in
9 :     if $\mathit{Verify}(e, \mathcal{L}_I, \mathcal{L}_O, \mathcal{L}) =$ true then return $\mathcal{L}$
10 :  end
```

Fig. 3. The Procedure of Invariant Inference

Example 4. Recall the program in Example 1. Let $\mathcal{L}_I = \mathtt{a}^*\mathtt{b}^*\mathtt{e}$ and $\mathcal{L}_O = \mathtt{b}^*\mathtt{a}^*\mathtt{e}$. By choosing an input $t_1 = \mathtt{a(a(b(b(e))))}$, we get:

$$\begin{aligned}
&(\emptyset, \emptyset, e_{rev}\, t_1) \longrightarrow^* (\emptyset, \emptyset, E_1[(\mathit{reverse}\,(\mathtt{a(b(b(e)))}))^\ell])\\
&\longrightarrow^* (\emptyset, \emptyset, E_2[(\mathit{reverse}\,(\mathtt{b(b(e))}))^\ell])\\
&\longrightarrow^* (\emptyset, \emptyset, E_2[(\mathtt{b(b(e))})^\ell])\\
&\longrightarrow^* (\{\mathtt{b(b(e))}\}, \{E_2\}, E_2[\mathtt{b(b(e))}])\\
&\longrightarrow^* (\{\mathtt{b(b(e))}\}, \{E_2\}, E_1[(\mathtt{b(b(a(e)))})^\ell])\\
&\longrightarrow^* (\{\mathtt{b(b(a(e)))}, \mathtt{b(b(e))}\}, \{E_1, E_2\}, \mathtt{b(b(a(a(e))))})
\end{aligned}$$

where $E_1 = \mathit{append}\,[\,]\,(\mathtt{a}\,\mathtt{e})$ and $E_2 = E_1[E_1]$. Thus, we get $\mathcal{U}_1 = \{\mathtt{b(b(a(e)))}, \mathtt{b(b(e))}\}$ and $\mathcal{K}_1 = \{E_1, E_2\}$. The membership oracle $o_{K_1}(x)$ returns 1 if and only if x is an element of $\mathtt{b}^*\mathtt{a}^*\mathtt{e}$. ALTEX constructs a table similar to the one in Example 3, and returns the minimal automaton accepting $\mathtt{b}^*\mathtt{a}^*\mathtt{e}$. This is a valid invariant, so that the procedure terminates and returns $\mathtt{b}^*\mathtt{a}^*\mathtt{e}$.

If the output specification $\mathcal{L}_O$ is replaced with $(\mathtt{a} \mid \mathtt{b})^*\mathtt{e}$, then o_{K_1} returns 1 for every element of $\mathcal{T}_{\mathtt{a,b,e}}$. Thus, ALTEX returns an automaton accepting $(\mathtt{a} \mid \mathtt{b})^*\mathtt{e}$, which is again a valid invariant. □

5.3 Properties of the Procedure

This section discusses properties of our invariant inference procedure.

We make several assumptions. Let $(\mathcal{U}_\infty, \mathcal{K}_\infty)$ be the least fixedoint of the function $\lambda(\mathcal{U}, \mathcal{K}).\bigcup\{(\mathcal{U}', \mathcal{K}') \mid (\mathcal{U}, \mathcal{K}, e\, t_I) \longrightarrow^* (\mathcal{U}', \mathcal{K}', e'), t_I \in \mathcal{L}_I\}$. First, we assume that *Eval* is fair in the sense that (i) every pair (E, t) in $(\mathcal{U}_\infty, \mathcal{K}_\infty)$ is eventually put into $\mathcal{U}_i, \mathcal{K}_i$ for some i, and (ii) for every t such that $(\mathcal{U}_\infty, \mathcal{K}_\infty, e\, t_I) \longrightarrow^*$

$(\mathcal{U}', \mathcal{K}', t)$ for some $t_I \in \mathcal{L}_I$, *Eval* eventually returns t. Note that $\mathcal{U}_\infty$ is the least forward invariant (recall Section 3) and the set $\mathcal{L}_B$ defined by:

$$\mathcal{L}_B := \{t \mid \forall E \in \mathcal{K}_\infty, t'.E[t] \longrightarrow^* t' \implies t' \in \mathcal{L}_O\}$$

is a backward invariant under this assumption. Secondly, we assume that *Verify* is a complete procedure. Thirdly, $\vdash \forall E \in \mathcal{K}_i.\forall t'.(E[t] \longrightarrow^* t' \implies t' \in \mathcal{L}_O)$ is decidable, so that the oracle $o_{\mathcal{K}_i}$ is complete for the language defined by $\mathcal{K}_i$.

The following properties follow immediately from the definition of the procedure and the above assumptions.

Fact 2 *Let e be a program, and $\mathcal{L}_I$ and $\mathcal{L}_O$ are regular tree languages.*

1. *If there exist trees t_I, t_O such that $e\, t_I \longrightarrow^* t_O$ with $t_I \in \mathcal{L}_I$ and $t_O \notin \mathcal{L}_O$, then the procedure eventually reports that there is no valid invariant.*
2. *If the procedure returns $\mathcal{L}$, then $\models (e, \mathcal{L}_I, \mathcal{L}_O, \mathcal{L})$ holds.*

The algorithm does not terminate in general, however, even under the assumptions above. The main problems are:

1. The function $o_{\mathcal{K}_i}$ is only an *approximation* of the membership oracle for the backward invariant $\mathcal{L}_B$. Although $o_{\mathcal{K}_i}$ approaches the ideal oracle $\lambda x.x \overset{?}{\in} \mathcal{L}_B$ in the limit, $o_{\mathcal{K}_i}$ may return a wrong answer for some input, for any finite i.
2. The (least) forward invariant $\mathcal{U}_\infty$ may not contain a representative sample for $\mathcal{L}_B$. Thus, the assumption for the correctness of Altex (recall Theorem 1) is not met.

The first point can be remedied by slightly changing the algorithm: On line 8, instead of calling Altex for the pair $(\mathcal{U}_i, o_{\mathcal{K}_i})$, call it for every pair $(\mathcal{U}_i, o_{\mathcal{K}_j})$ such that $i \leq j$. Then, we have:

Theorem 3. *Suppose that $\mathcal{L}_B$ is a regular tree language and $\mathcal{U}_\infty$ contains a representative sample of $\mathcal{L}_B$, and that $\mathcal{U}_\infty \subseteq \mathcal{L}_B$. Then, the refined algorithm eventually terminates and returns $\mathcal{L}_B$.*

Proof. Let $\mathcal{E}$ be the representative sample contained in $\mathcal{U}_\infty$. Then, by the assumption on the fairness of *Eval*, $\mathcal{E} \subseteq \mathcal{U}_i$ holds for some i. (Here, note that $\mathcal{E}$ is a finite set by the definition of representative samples.) For $\mathcal{U}_i$, Altex issues only a finite set S of membership queries. Therefore, there exists j such that $o_{\mathcal{K}_j}$ outputs the same answers for all the membership queries in S. Thus, by Theorem 1, Altex$(\mathcal{U}_i, o_{\mathcal{K}_j})$ outputs $\mathcal{L}_B$. By the assumption on the completeness of *Verify*, the procedure returns $\mathcal{L}_B$. □

The above assumptions that $\mathcal{L}_B$ is regular and that $\mathcal{U}_\infty$ contains a representative sample of $\mathcal{L}_B$ may be too strong in general. See Appendix for the discussion on how to relax it.

6 Experimental Results

We have implemented the algorithm presented in Section 5, and combined it with Unno et al.'s EHMTT verifier, which is used as *Verify* in Figure 3. As a result, we have obtained a fully-automated verifier for higher-order, tree-processing functional programs. In the prototype, the reduction rule Switch is suppressed, so that the evaluation for sample inputs is deterministic. A Web interface to test the prototype and bechmark programs are available at `http://www.kb.ecei.tohoku.ac.jp/~tabee/atva/exp/`.

Table 1 shows the experimental results. The columns "P" shows the number of labeled expressions for which invariants are inferred. "R" and "S" show the number of functions and the size of the program respectively. The size of a program is measured by the number of symbols occurring in the program. The column "I" shows how sample inputs are supplied: an integer k indicates that trees of height up to k are automatically generated, while "m(n,k)" means that n trees whose heights are up to k are manually given: for example, "m(2,12)" indicates that two trees whose heights are ≤ 12 are manually given as test cases. Q_I and Q_O respectively show the number of the states of tree automata for input and output specifications, while Σ_{Q_A} shows the sum of the number of the states of tree automata for inferred invariants. Finally, "Y/N" indicates whether the inference was successful ("Y") or not ("N") and "T" shows the time taken for the inference measured in seconds. "time-out" indicates that the inference did not terminate. "fail" indicates that the inference did not succeed, but no more sample inputs were available (on line 4 of the algorithm).

Program	P	R	S	I	Q_I	Q_O	Σ_{Q_A}	Y/N	T
`reverse`	2	2	24	4	2	2	3	Y	0.058
`rev-rmsp`	1	4	27	3	2	2	2	Y	0.139
`inssort`	1	3	21	4	1	2	2	Y	0.053
`mergesort`	4	7	123	5	1	2	6	Y	11.669
`lc-ucfirst`	1	3	32	2	1	2	1	Y	0.019
`rtrim`	2	5	46	3	2	1	3	Y	0.145
`ltrim-rtrim`	3	6	39	4	3	1	7	Y	5.501
`homrep-rev`	1	11	78	3	4	2	2	Y	0.067
`flatten`	1	2	16	5	3	2	2	Y	0.250
`foldr`	1	4	27	3	4	2	2	Y	0.014
`map-foldr`	2	6	46	4	4	2	5	Y	22.860
`addexp`	1	2	17	4	3	2	2	Y	0.013
`addexp-add1`	2	4	33	4	3	2	5	Y	0.088
`split`	4	5	86	12	7	9	56	N	time-out
`split`	4	5	86	m(2,12)	7	9	56	Y	362.929
`split'`	1	5	75	m(2,12)	7	9	17	Y	51.387
`id-fail`	1	2	6	2	2	3	1	N	fail

Table 1. Experimental Results

The programs from `reverse` to `homrep-rev` manipulate strings (represented in the form of linear trees). `reverse` is the same as the program shown in Example 1: it takes an element of $\mathtt{a}^*\mathtt{b}^*\mathtt{e}$ and reverses it. `rev-rmsp` takes an element of $(\mathtt{a} \mid \mathtt{s})^*(\mathtt{b} \mid \mathtt{s})^*\mathtt{e}$, where `s` stands for the whitespace character, reverses the input and removes all whitespace. `inssort` performs insertion sort on strings over $(\mathtt{a} \mid \mathtt{b})^*\mathtt{e}$, while `mergesort` performs the merge sort on strings of the same form. `lc-ucfirst` takes a string over $(\mathtt{a} \mid \mathtt{b} \mid \mathtt{A} \mid \mathtt{B})^*\mathtt{e}$, first lowers all `A`s and `B`s and finally capitalizes the initial character (if any). `rtrim` takes a string over $(\mathtt{a} \mid \mathtt{b})^*\mathtt{s}^*\mathtt{e}$ and removes all the trailing whitespace. Similarly, `ltrim-rtrim` takes a string over $\mathtt{s}^*(\mathtt{a} \mid \mathtt{b})^*\mathtt{s}^*\mathtt{e}$ and removes all the heading and trailing whitespace. Finally, `homrep-rev` takes a word homomorphism h, a natural number n and a string $w \in (\mathtt{a} \mid \mathtt{b})^*\mathtt{e}$ and produces the reverse of the image $h^n(w)$. We let $h = \{\mathtt{a} \mapsto \mathtt{bb}, \mathtt{b} \mapsto \mathtt{a}\}$ and verified that if n is an even number and $w \in \mathtt{a}^*\mathtt{b}^*\mathtt{e}$, then the reversed image is in $\mathtt{b}^*\mathtt{a}^*\mathtt{e}$.

The programs from `flatten` to `map-foldr` manipulate lists. `flatten` takes a list of lists and returns the flattened list. `foldr` takes a list l of natural numbers and a natural number x and performs foldr $*$ l x where $*$ is the multiplicative operator. It is verified that the program returns an even number if x is even. `map-foldr` also takes a natural-number list l and a natural x and performs foldr $*$ (map $(\lambda x.2 * x + 1)$ l) x. It is verified that it returns an odd number if x is odd.

The program `addexp` is an evaluator of arithmetic expressions over natural numbers where the additive is the only operator. The verified property is that an expression over even numbers evaluates to an even number. Similarly, `addexp-add1` takes an additive expression. It first adds one to each leaf of the expression and evaluates it. It is verified that an expression over odd numbers evaluates to an even number, respectively.

The program `split` is taken from a sample program of CDuce [1], a higher-order XML-centric functional language. It takes an arbitrarily nested lists of persons representing parent-child relationship and splits the children of each person into a list of men (sons) and of women (daughters). The program `split'` is semantically equivalent to `split`, while the number of labeled expressions is reduced to 1 by fusioning some functions and redundant expressions.

Finally, `id-fail` is the following identity function: `let id x = x in id (id x)`$^\ell$, where the input and output specifications are respectively given by $\mathcal{L}_I = \{\mathtt{e}, \mathtt{a}(\mathtt{e})\}$ and $\mathcal{L}_O = \{\mathtt{e}, \mathtt{a}(\mathtt{e}), \mathtt{a}(\mathtt{a}(\mathtt{e}))\}$. It is an artificial example constructed to show a limitation of our invariant inference.

As shown in the table, our algorithm could successfully infer appropriate invariants for all programs, except `split` and `id-fail`. As for programs from `reverse` to `addexp-add1`, the total numbers of the states of the inferred regular tree languages range from 1 to 7 and the program size varies from 16 to 123. Compared with them, the time taken for inference to terminate ranges widely from 14 milliseconds to more than 22 seconds. This suggests that the effectiveness of our algorithm heavily depends on the choice of the sample input series, not only on the complexity of the program and invariants.

For the program `split`, the naïve enumeration of input samples resulted in combinatorial explosion and did not terminate. Thus, sample input trees had to be supplied manually, A smarter strategy of choosing samples would be required for fully automated inference. The manual enumeration of inputs was also necessary for `split'`, but the inference was much faster. This indicates that the programming style is also relevant to the efficiency of the inference.

For `id-fail`, the inference fails because of a limitation of ALTEX. The sample set $\mathcal{U}_i$ and the oracle $o_{\mathcal{K}_i}$ passed to ALTEX are $\{\mathtt{e}, \mathtt{a}(\mathtt{e})\}$ and $\lambda t.(t \overset{?}{\in} \{\mathtt{e}, \mathtt{a}(\mathtt{e}), \mathtt{a}(\mathtt{a}(\mathtt{e}))\}$ respectively, so that the assumption of Theorem 1 fails, i.e., $\mathcal{U}_i$ is not a representative sample of the language $\{\mathtt{e}, \mathtt{a}(\mathtt{e}), \mathtt{a}(\mathtt{a}(\mathtt{e}))\}$. ALTEX$(\mathcal{U}_i, o_{\mathcal{K}_i})$ returns $\mathtt{a}^*\mathtt{e}$, and there is no way to refine it as there are no further inputs.

7 Related Work

As mentioned in Section 1, several static analysis methods have been proposed for inferring invariants of tree- or string-processing programs [8, 14, 16]. A main limitation of these approaches is context-insensitiveness. For example, consider the following program `let id x = x in let f y z = ... in f (id a) (id b)`. Jones and Andereson's analysis [8] would infer $\{a, b\}$ as the values of y and z, rather than $\{a\}$ and $\{b\}$ for y and z. Kochems and Ong [14] refined their analysis by introducing a restricted form of context-sensitivity by using index grammars, but it still suffers from similar problems. In contrast, our learning-based approach is context-sensitive, as we actually evaluate the program (for sample inputs) and collect the values of each expression. There are also limitations of our approach, that the procedure may not terminate in general, and (even it terminates) that there is no theoretical bound for the complexity of invariant inference. Thus, we think the standard static analysis approach and our approach should be used as complementary methods.

In Section 6, we have combined our invariant inference technique with Unno et al.'s method for verification of tree-processing programs [21]. Their method is based on higher-order model checking [11, 13, 17], but requires invariant annotations for certain program points. By the combination with the invariant inference technique, we have enabled fully automated verification of higher-order tree-processing programs. We expect that a similar combination with other verification methods is possible; for example, our invariant inference can be used for replacing the binding analysis part of Ong and Ramsay's verification method [18]. As their method is also based on higher-order model checking, the context-sensitivity of the binding analysis is important for fully exploiting the precision of higher-order model checking.

There are other pieces of work to infer invariants based on test execution or machine learning techniques [3, 5, 6, 10]. For example, Jung et al. [9, 10] presented a method for inferring loop invariants for a simple **while**-language. Our combination of forward and backward information for invariant inference has also some similarity to interpolant-based techniques for invariant or predicate discovery [7, 15, 20]. To our knowledge, however these techniques are mainly for the inference of numeric or non-recursive symbolic constraints, and it is not clear how to lift them to infer invariants in the form of regular tree languages.

8 Conclusion and Future Work

We have proposed a method for inferring regular tree language invariants of tree processing programs. The proposed approach is based on test execution and a technique of language identification from positive samples and membership queries. We have combined it with a previous verification method requiring invariant annotations, and implemented a fully automated verifier for higher-order tree processing programs. The preliminary experiments indicate that our invariant inference works reasonably well for small programs. Future work includes optimizations of the invariant inference procedure to achieve more scalability. For example, a more elaborated strategy for test-case generation would enable faster convergence of invariant inference.

References

1. V. Benzaken, G. Castagna, and A. Frisch. Cduce: an xml-centric general-purpose language. *SIGPLAN Not.*, 38:51–63, August 2003.
2. J. Besombes and J.-Y. Marion. Learning tree languages from positive examples and membership queries. *Theor. Comput. Sci.*, 382:183–197, September 2007.
3. C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: dynamic symbolic execution for invariant inference. In *International Conference on Software Engineering 2008*, volume 2008, pages 0–18. Association for Computing Machinery, One Astor Plaza 1515 Broadway, 17 th Floor New York NY 10036-5701 USA, 2008.
4. R. Davies. *Practical refinement-type checking*. PhD thesis, Pittsburgh, PA, USA, 2005.
5. M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *Software Engineering, IEEE Transactions on*, 27(2):99–123, 2002.
6. S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 291–301, New York, NY, USA, 2002. ACM.
7. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Proceedings of POPL 2004*, pages 232–244. ACM Press, 2004.
8. N. D. Jones and N. Andersen. Flow analysis of lazy higher-order functional programs. *Theor. Comput. Sci.*, 375(1-3):120–136, 2007.
9. Y. Jung, S. Kong, B. Wang, and K. Yi. Deriving invariants by algorithmic learning, decision procedures, and predicate abstraction. In *Verification, Model Checking, and Abstract Interpretation*, pages 180–196. Springer, 2010.
10. Y. Jung, W. Lee, B. Wang, and K. Yi. Predicate Generation for Learning-Based Quantifier-Free Loop Invariant Inference. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 205–219, 2011.
11. N. Kobayashi. Model-checking higher-order functions. In *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*, PPDP '09, pages 25–36, New York, NY, USA, 2009. ACM Press.

12. N. Kobayashi. TRecS. `http://www.kb.ecei.tohoku.ac.jp/~koba/trecs/`, 2009.
13. N. Kobayashi, N. Tabuchi, and H. Unno. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In *Proc. of POPL*, pages 495–507. ACM Press, 2010.
14. J. Kochems and C.-H. L. Ong. Improved functional flow and reachability analyses using indexed linear tree grammars. In *Proceedings of RTA 2011*, 2011.
15. K. McMillan. Quantified invariant generation using an interpolating saturation prover. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 413–427, 2008.
16. Y. Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th international conference on World Wide Web (WWW 2005)*, pages 432–441. acm, 2005.
17. C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS 2006*, pages 81–90. IEEE Computer Society Press, 2006.
18. C.-H. L. Ong and S. J. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 587–598, New York, NY, USA, 2011. ACM.
19. P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *Proc. of POPL*, pages 199–210, 2010.
20. H. Unno and N. Kobayashi. Dependent type inference with interpolants. In *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*, PPDP '09, pages 277–288, New York, NY, USA, 2009. ACM.
21. H. Unno, N. Tabuchi, and N. Kobayashi. Verification of tree-processing programs via higher-order model checking. In *Proceedings of APLAS 2010*, volume 6461 of *LNCS*, pages 312–327. Springer-Verlag, 2010.

A Further Discussion on Termination of Invariant Inference

Theorem 3 makes a rather strong assumption that $\mathcal{U}_\infty$ is a representative sample of $\mathcal{L}_B$. Without this assumption, the algorithm may not terminate. In fact, as discussed in Section 6, for the following input:

$$\begin{aligned} &e \equiv \texttt{let id x = x in id (id x)}^\ell \\ &\mathcal{L}_I = \{\mathtt{e}, \mathtt{a}(\mathtt{e})\} \\ &\mathcal{L}_O = \{\mathtt{e}, \mathtt{a}(\mathtt{e}), \mathtt{a}(\mathtt{a}(\mathtt{e}))\} \end{aligned}$$

Altex is called with the sample set $\{\mathtt{e}, \mathtt{a}(\mathtt{e})\}$ and the oracle $\lambda x.(x \overset{?}{\in} \mathcal{L}_O)$. With these inputs, Altex constructs the following observation table:

	$[\,]$	$\mathtt{a}([\,])$
$\mathtt{e}$	1	1
$\mathtt{a}(\mathtt{e})$	1	1

The language of the automaton $\mathcal{M} = (\{q_{11}\}, \{\mathtt{a}, \mathtt{e}\}, \{q_{11}\}, \{\mathtt{e} \to q_{11}, \mathtt{a}\ q_{11} \to q_{11}\})$ induced by the table is thus $\mathtt{a}^*(\mathtt{e})$, which is larger than $\mathcal{L}_\mathcal{B} = \mathcal{L}_O$, so that the verification of $\models (e, \mathcal{L}_I, \mathcal{L}_O, \mathtt{a}^*(\mathtt{e}))$ fails.

A possible way to overcome the above limitation is to extend Altex. In addition to positive samples $\mathcal{E}$ and the membership oracle o, let us assume that another oracle f to test the inclusion relation is available: The new oracle f should take a regular tree language X as input, check whether $X \subseteq \mathcal{L}$ (where $\mathcal{L}$ is the language to be learned), and if $X \not\subseteq \mathcal{L}$, return an element of $X \setminus \mathcal{L}$ as a counterexample. We can then modify Altex so that, given $\mathcal{E}$, o and f, it returns a subautomaton of the minimal automaton for accepting $\mathcal{L}$, no matter whether $\mathcal{E}$ is a representative sample.

In the example above, if we get $\mathtt{a}(\mathtt{a}(\mathtt{a}(\mathtt{e})))$ as a counterexample of $\mathtt{a}^*\mathtt{e} \subseteq \{\mathtt{e}, \mathtt{a}(\mathtt{e}), \mathtt{a}(\mathtt{a}(\mathtt{e}))\}$, then we can add the context $\mathtt{a}(\mathtt{a}([\,]))$ to the table, and obtain the new observation table:

	$[\,]$	$\mathtt{a}([\,])$	$\mathtt{a}(\mathtt{a}([\,]))$
$\mathtt{e}$	1	1	1
$\mathtt{a}(\mathtt{e})$	1	1	0

The inferred automaton is then $\mathcal{M}_1 = (\{q_{110}, q_{111}\}, \{\mathtt{a}, \mathtt{e}\}, \{q_{111}, q_{110}\}, \{\mathtt{e} \to q_{111}, \mathtt{a}\ q_{111} \to q_{110}\})$ and $\mathcal{L}(\mathcal{M}_1) = \{\mathtt{e}, \mathtt{a}(\mathtt{e})\}$.

By using the extended version of Altex, we expect that the termination of the invariant inference is guaranteed without the assumption that $\mathcal{U}_\infty$ is a representative sample of $\mathcal{L}_B$. The technical details are rather complex, and left for future work.