*Math. Struct. in Comp. Science* (2015), *vol.* 25, *pp.* 841–866. 
doi:10.1017/S0960129513000054 First published online 10 November 2014

# Verification of tree-processing programs via higher-order mode checking†

HIROSHI UNNO‡, NAOSHI TABUCHI§ and NAOKI KOBAYASHI¶

‡*Department of Computer Science, Graduate School of Systems and Information Engineering, University of Tsukuba, 1-1-1 Tennodai, Tsukuba-shi, Ibaraki 305-8573, Japan*
*Email:* uhiro@cs.tsukuba.ac.jp
§*Trek Inc., 14-15 Futsuka-machi, Aoba-ku, Sendai-shi, Miyagi 980-0802, Japan*
*Email:* tabuchi_naoshi@trek.co.jp
¶*Department of Computer Science, Graduate School of Information Science and Technology, The University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-0033, Japan*
*Email:* koba@is.s.u-tokyo.ac.jp

*Received 28 May 2011, Revised 31 January 2012*

We propose a new method to verify that a higher-order, tree-processing functional program conforms to an input/output specification. Our method reduces the verification problem to multiple verification problems for higher-order multi-tree transducers, which are then transformed into higher-order recursion schemes and model-checked. Unlike previous methods, our new method can deal with arbitrary higher-order functional programs manipulating algebraic data structures, as long as certain invariants on intermediate data structures are provided by a programmer. We have proved the soundness of the method and implemented a prototype verifier.

## 1. Introduction

The model checking of higher-order recursion schemes (Ong 2006), or higher-order model checking for short, has been extensively studied recently. Ong (2006) has shown the decidability of higher-order model checking. Kobayashi (2009a,b) then developed a practical model checking algorithm and applied it to program verification. The present work is an extension of that line of work, trying to apply higher-order model checking to verification of a wider range of higher-order programs.

From a programming language point of view, recursion schemes are terms of the simply-typed $\lambda$-calculus with recursion and tree constructors (but not destructors). One can also encode finite data domains (such as booleans) by using Church encoding. Based on this observation, Kobayashi (2009a) applied model checking to resource usage verification of simply typed functional programs with recursion, booleans and resource primitives. The limitation of this approach was that programs manipulating infinite data domains such as lists and trees could not be handled. To relax this limitation, in our previous work (Kobayashi *et al.* 2010), we have introduced higher-order multi-parameter tree transducers (HMTTs) as an extension of recursion schemes with tree destructors.

† Revised and extended version of (Unno *et al.* 2010).

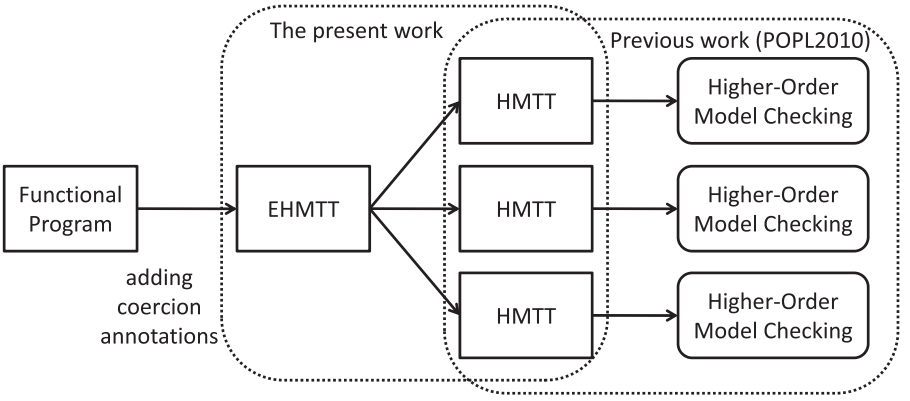


Fig. 1. Overall structure of EHMTT verification method.

HMTTs are a kind of tree transducers that take (possibly infinite) input trees, which can be destructed, and outputs a (possibly infinite) tree. However, there still remains a gap between HMTTs and ordinary functional programs that use recursive data structures since HMTTs do not support intermediate data structures: an HMTT cannot destruct trees constructed by the HMTT itself. For example, let us consider the following reverse function on lists:

$$\begin{aligned} &\mathit{Reverse}\ x = \mathbf{case}\ x\ \mathbf{of}\ \mathtt{nil} \Rightarrow \mathtt{nil} \\ &\qquad\qquad\qquad\qquad \mid \mathtt{cons}\ z\ x' \Rightarrow \mathit{Append}\ (\mathit{Reverse}\ x')\ (\mathtt{cons}\ z\ \mathtt{nil}) \\ &\mathit{Append}\ x\ y = \mathbf{case}\ x\ \mathbf{of}\ \mathtt{nil} \Rightarrow y \\ &\qquad\qquad\qquad\qquad \mid \mathtt{cons}\ z\ x' \Rightarrow \mathtt{cons}\ z\ (\mathit{Append}\ x'\ y). \end{aligned}$$

Here, the list constructed by $\mathit{Reverse}\ x'$ is passed to the first argument of *Append*, and then destructed. Thus, this reverse function cannot be modelled as an HMTT.

In this article, we propose a verification method for an extension of HMTTs called extended HMTTs (EHMTTs). In essence, EHMTTs are higher-order, simply-typed functional programs with recursion and tree primitives. Unlike our previous HMTTs (Kobayashi *et al.* 2010), there is no fundamental restriction on tree constructors/destructors, except that special annotations (called *coercion*) are required for destructing trees constructed in a program. Our method can check whether the output trees generated by a given EHMTT conform to a given output specification whenever the input trees conform to given input specifications. We can apply our method to verification of ordinary functional programs that manipulate algebraic data structures by encoding them as trees and adding annotations to the programs.

The overall structure of our method is shown in Figure 1. A given EHMTT verification problem is reduced to multiple HMTT verification problems, which are then solved by the HMTT verification method presented in our previous work (Kobayashi *et al.* 2010). The HMTT verification method further reduces the HMTT verification problems to model checking problems of recursion schemes, which are finally solved by Kobayashi's higher-order model checker TRecS (Kobayashi 2009b). In this article, we formalize the reduction from an EHMTT verification problem to HMTT verification problems, and prove the soundness of the reduction. Our verification method is not complete, however, since the

verification problem is undecidable in general. We have implemented a prototype verifier and verified functional programs that manipulate XML and user-defined recursive data structures.

The rest of the article is organized as follows. Section 2 presents some preliminary definitions and notations. Section 3 introduces EHMTTs. Section 4 formalizes our verification method for EHMTTs. Section 5 reports on the experimental results. We compare our method with related work in Section 6, and conclude the article with some remarks on future work in Section 7.

## 2. Preliminaries

We write $\mathrm{dom}(f)$ for the domain of a map $f$, and $f\{x \mapsto v\}$ for the map $f'$ such that $\mathrm{dom}(f') = \mathrm{dom}(f) \cup \{x\}$, $f'(x) = v$ and $f'(y) = f(y)$ for $y \in \mathrm{dom}(f) \setminus \{x\}$.

We write $X^*$ for the set of sequences of elements of $X$. We write $\epsilon$ for the empty sequence, and $v_1, \ldots, v_n$ for the sequence consisting of $v_1, \ldots, v_n$. We write $s_1 \cdot s_2$ for the concatenation of sequences $s_1$ and $s_2$. A sequence $v_1, \ldots, v_n$ is often abbreviated to $\widetilde{v}$. Let $\mathbb{P}$ denote the set of positive integers. *$X$-labelled tree* $R$ is a map from a subset of $\mathbb{P}^*$ to $X$ such that:

— $\mathrm{dom}(R)$ is prefix closed, i.e. if $\pi \cdot i \in \mathrm{dom}(R)$, then $\pi \in \mathrm{dom}(R)$ and
— if $\pi \cdot i \in \mathrm{dom}(R)$, then $\pi \cdot j \in \mathrm{dom}(R)$ for any $j \in \{1, \ldots, i\}$.

A *ranked alphabet* $\Sigma$ is a map from a finite set of symbols to non-negative integers. For each symbol $a \in \mathrm{dom}(\Sigma)$, $\Sigma(a)$ denotes the arity of $a$. A *$\Sigma$-labelled ranked tree* $T$ is a $(\mathrm{dom}(\Sigma))$-labelled tree such that if $T(\pi) = a$, then $\{i \mid \pi \cdot i \in \mathrm{dom}(T)\} = \{1, \ldots, \Sigma(a)\}$. Note that, a $\Sigma$-labelled ranked tree $T$ is possibly infinite and $\mathrm{dom}(T) \subseteq \{1, \cdots, A_\Sigma\}^*$, where $A_\Sigma$ denotes the largest arity of the symbols in $\mathrm{dom}(\Sigma)$. Let $T_1, \ldots, T_n$ be $\Sigma$-labelled ranked trees. We write $a\ T_1 \ldots T_n$ for a $\Sigma$-labelled tree such that $(a\ T_1 \ldots T_n)(\epsilon) = a$ and $(a\ T_1 \ldots T_n)(i \cdot \pi) = T_i(\pi)$. Let $L_1, \ldots, L_n$ be sets of $\Sigma$-labelled ranked trees. We write $a\ L_1 \ldots L_n$ for $\{a\ T_1 \ldots T_n \mid T_1 \in L_1, \ldots, T_n \in L_n\}$. We say that a $\Sigma$-labelled ranked tree $T$ is *linear* if $\mathrm{dom}(T) = \{1\}^*$, and use a regular expression for a set of linear trees. For example, $a^*c$ represents $\{c, a\ c, a\ (a\ c), \ldots\}$. We sometimes identify a linear tree $a_1\ (\cdots (a_n\ c) \cdots)$ with the sequence $a_1 \cdots a_n \cdot c$.

## 3. Extended HMTTs

In this section, we introduce EHMTTs. From a programming language point of view, an EHMTT is a simply-typed, call-by-name, higher-order functional program that takes possibly infinite trees as input and outputs a possibly infinite tree. The main difference from ordinary functional programs is that trees are classified into input and output trees. Input trees can only be destructed, and output trees can only be constructed in a program, as in other tree transducers (Engelfriet and Vogler 1985, 1988). Special primitives ($\mathbf{coerce}^L(\cdot)$ introduced below) are however provided to convert output trees to input trees, so that, unlike ordinary tree transducers, trees constructed in a program can be destructed again in the same program. Thus, the class of EHMTTs is actually Turing complete.

We fix below a ranked alphabet $\Sigma$. We call elements of $\mathrm{dom}(\Sigma)$ *terminal symbols*, and use the meta-variable $a$ for them.

**Definition 3.1 (EHMTT).** An EHMTT $\mathcal{P}$ is a pair $(D, S)$, where $D$ is a set of function definitions of the form $\{F_1\,\widetilde{x}_1 = t_1, \ldots, F_n\,\widetilde{x}_n = t_n\}$, and $S$ is the name of the main function. Here, $t$ ranges over the set of terms, given by

$$t \ ::= \ a \mid x \mid F \mid t_1\ t_2 \mid \mathbf{case}\ t\ \mathbf{of}\ \{a_i\ \widetilde{y}_i \Rightarrow t_i\}_{i=1}^{n} \mid \mathbf{coerce}^{L}(t) \mid \mathbf{gen}^{L}.$$

Here, $L$ denotes a set of trees. An EHMTT $(D, S)$ is *well sorted* under a sort environment $\mathcal{K}$, if $S : \mathbf{i} \to \cdots \to \mathbf{i} \to \mathbf{o} \in \mathcal{K}$ and $\vdash D : \mathcal{K}$ is derivable by using the sort assignment rules in Figure 2. An HMTT is an EHMTT that does not contain $\mathbf{coerce}^{L}(t)$. □

In the figure, the sorts $\mathbf{i}$ and $\mathbf{o}$ describe input and output trees, respectively. The sort $\kappa_1 \to \kappa_2$ denotes functions that take a tree or tree function of sort $\kappa_1$ and return a tree or tree function of sort $\kappa_2$. $\widetilde{\kappa} \to \mathbf{o}$, and $\widetilde{x} : \widetilde{\kappa}$ are shorthand forms of $\kappa_1 \to \cdots \to \kappa_k \to \mathbf{o}$ and $x_1 : \kappa_1, \ldots, x_k : \kappa_k$, respectively. We consider only well-sorted EHMTTs below.

The tree destructor $\mathbf{case}\ t\ \mathbf{of}\ \{a_i\ \widetilde{y}_i \Rightarrow t_i\}_{i=1}^{n}$ is reduced to $[\widetilde{u}_i/\widetilde{y}_i]t_i$ if $t$ evaluates to $a_i\ \widetilde{u}_i$. If $t$ does not match any pattern, the term evaluates to a special terminal symbol $\mathtt{fail}$.[†] Note that each branch $t_i$ is assumed to have the sort $\mathbf{o}$ for simplicity. The condition can always be enforced by applying CPS transformation. The term $\mathbf{coerce}^{L}(t)$ asserts that the tree generated by $t$ belongs to the set $L$ of trees, and converts the tree to an input tree. $\mathbf{gen}^{L}$ generates an element of $L$ non-deterministically.

**Example 3.1.** Consider the EHMTT $\mathcal{P}_{rev} = (D, \mathit{Reverse})$, where $D$ consists of the following:

$$\begin{aligned}
\mathit{Reverse}\ x = \mathbf{case}\ x\ \mathbf{of}\ & \mathtt{e} \Rightarrow \mathtt{e} \\
& \mid \mathtt{a}\ x' \Rightarrow \mathit{Append}\ (\mathbf{coerce}^{\mathtt{b}^*\mathtt{a}^*\mathtt{e}}(\mathit{Reverse}\ x'))\ (\mathtt{a}\ \mathtt{e}) \\
& \mid \mathtt{b}\ x' \Rightarrow \mathit{Append}\ (\mathbf{coerce}^{\mathtt{b}^*\mathtt{e}}(\mathit{Reverse}\ x'))\ (\mathtt{b}\ \mathtt{e}) \\
\mathit{Append}\ x\ y = \mathbf{case}\ x\ \mathbf{of}\ & \mathtt{e} \Rightarrow y \\
& \mid \mathtt{a}\ x' \Rightarrow \mathtt{a}\ (\mathit{Append}\ x'\ y) \\
& \mid \mathtt{b}\ x' \Rightarrow \mathtt{b}\ (\mathit{Append}\ x'\ y)
\end{aligned}$$

$P_{rev}$ takes a tree of the form $\mathtt{a}^m(\mathtt{b}^n(\mathtt{e}))$ as input, and outputs the tree $\mathtt{b}^n(\mathtt{a}^m(\mathtt{e}))$. The coercions $\mathbf{coerce}^{\mathtt{b}^*\mathtt{a}^*\mathtt{e}}(\cdot)$ and $\mathbf{coerce}^{\mathtt{b}^*\mathtt{e}}(\cdot)$ assert that their arguments belong to $\{\mathtt{b}^m(\mathtt{a}^n(\mathtt{e})) \mid m, n \geqslant 0\}$ and $\{\mathtt{b}^m(\mathtt{e}) \mid m \geqslant 0\}$ respectively, and convert them to input trees. Note that, *Reverse* and *Append* have sorts $\mathbf{i} \to \mathbf{o}$ and $\mathbf{i} \to \mathbf{o} \to \mathbf{o}$ respectively, so that $\mathit{Reverse}\ x'$ returns an output tree. □

Figure 2 shows the formal semantics of the language. In the semantics, the set of terms are extended as follows. An underlined symbol $\underline{a}$ denotes an input tree constructor (which occurs only at run-time, not in source programs). $\mathbf{o2i}(t)$ and $\mathbf{assert}^{L}(t)$ are used to define the semantics of $\mathbf{coerce}^{L}(t)$: the former converts an output tree to an input tree, and the latter asserts that the tree generated by $t$ belongs to $L$. In the semantics, $\mathbf{coerce}^{L}(t)$ evaluates to

[†] Thus, verification of the absence of pattern match errors can be encoded as a problem of checking that the tree generated by EHMTT does not contain $\mathtt{fail}$, which is an instance of EHMTT verification problems considered below.

Syntax of Sorts:

$$\kappa ::= \mathbf{i} \mid \mathbf{o} \mid \kappa_1 \to \kappa_2$$

Sort Assignment Rules:

$$\mathcal{K} \vdash F : \mathcal{K}(F) \quad \text{(T-Fun)}$$

$$\mathcal{K} \vdash a : \underbrace{\mathbf{o} \to \cdots \to \mathbf{o}}_{\Sigma(a)} \to \mathbf{o} \quad \text{(T-Con)}$$

$$\mathcal{K} \vdash x : \mathcal{K}(x) \quad \text{(T-Var)}$$

$$\frac{\mathcal{K} \vdash t_1 : \kappa_1 \to \kappa_2 \quad \mathcal{K} \vdash t_2 : \kappa_1}{\mathcal{K} \vdash t_1\ t_2 : \kappa_2} \quad \text{(T-App)}$$

$$\frac{\mathcal{K} \vdash t : \mathbf{i} \qquad \mathcal{K}, \widetilde{y_i} : \widetilde{\mathbf{i}} \vdash t_i : \mathbf{o} \ (\text{for all } i = 1, \ldots, N)}{\mathcal{K} \vdash \mathbf{case}\ t\ \mathbf{of}\ \{a_i\ \widetilde{y_i} \Rightarrow t_i\}_{i=1}^n : \mathbf{o}} \quad \text{(T-Case)}$$

$$\mathcal{K} \vdash \mathbf{gen}^L : \mathbf{i} \quad \text{(T-Gen)}$$

$$\frac{\mathcal{K} \vdash t : \mathbf{o}}{\mathcal{K} \vdash \mathbf{coerce}^L(t) : \mathbf{i}} \quad \text{(T-Coerce)}$$

$$\frac{\mathcal{K} = \{F_1 : \widetilde{\kappa}_1 \to \mathbf{o}, \ldots, F_n : \widetilde{\kappa}_n \to \mathbf{o}\} \quad \mathcal{K}, \widetilde{x_i} : \widetilde{\kappa_i} \vdash t_i : \mathbf{o} \ (\text{for each } i)}{\vdash \{F_1\ \widetilde{x}_1 = t_1, \ldots, F_n\ \widetilde{x}_n = t_n\} : \mathcal{K}} \quad \text{(T-Def)}$$

Operational Semantics

$$\begin{aligned} t \ (\text{extended terms}) \quad &::= \quad \cdots \mid \underline{a} \mid \mathbf{o2i}(t) \mid \mathbf{assert}^L(t) \\ E \ (\text{evaluation contexts}) \quad &::= \quad [\,] \mid a\ t_1 \cdots t_{j-1}\ E\ t_{j+1} \cdots t_{\Sigma(a)} \\ &\quad\ \mid \mathbf{case}\ E\ \mathbf{of}\ \{a_i\ \widetilde{y_i} \Rightarrow t_i\}_{i=1}^n \mid \mathbf{o2i}(E) \mid \mathbf{assert}^L(E) \end{aligned}$$

$$\frac{F\ \widetilde{x} = t \in D}{E[F\ \widetilde{t}] \longrightarrow_{\mathcal{P}} E[[\widetilde{t}/\widetilde{x}]t]} \quad \text{(E-App)}$$

$$E[\mathbf{case}\ \underline{a_i}\ \widetilde{t}\ \mathbf{of}\ \{a_i\ \widetilde{y_i} \Rightarrow t_i\}_{i=1}^n] \longrightarrow_{\mathcal{P}} E[[\widetilde{t}/\widetilde{y_i}]t_i] \quad \text{(E-Case)}$$

$$\frac{a \notin \{a_1, \ldots, a_n\}}{E[\mathbf{case}\ \underline{a}\ \widetilde{t}\ \mathbf{of}\ \{a_i\ \widetilde{y_i} \Rightarrow t_i\}_{i=1}^n] \longrightarrow_{\mathcal{P}} E[\mathtt{fail}]} \quad \text{(E-Case-Fail)}$$

$$\frac{a\ L_1 \cdots L_n \subseteq L \qquad L_1, \ldots, L_n \neq \emptyset}{E[\mathbf{gen}^L] \longrightarrow_{\mathcal{P}} E[\underline{a}\ \mathbf{gen}^{L_1} \ldots \mathbf{gen}^{L_n}]} \quad \text{(E-Gen)}$$

$$E[\mathbf{coerce}^L(t)] \longrightarrow_{\mathcal{P}} \mathbf{assert}^L(t) \quad \text{(E-Coerce-Assert)}$$

$$E[\mathbf{coerce}^L(t)] \longrightarrow_{\mathcal{P}} E[\mathbf{o2i}(t)] \quad \text{(E-Coerce-Input)}$$

$$E[\mathbf{o2i}(a\ t_1 \cdots t_n)] \longrightarrow_{\mathcal{P}} E[\underline{a}\ \mathbf{o2i}(t_1) \cdots \mathbf{o2i}(t_n)] \quad \text{(E-Input)}$$

$$\frac{t^\perp \notin L^\perp}{\mathbf{assert}^L(t) \longrightarrow_{\mathcal{P}} \mathbf{Error}} \quad \text{(E-Assert-Error)}$$

Fig. 2. Sort assignment rules and call-by-name operational semantics.

**o2i**$(t)$ or **assert**$^L(t)$ non-deterministically by the rule E-Coerce-Input or E-Coerce-Assert, respectively. In the rule E-Assert-Error, $t^\perp$ is a finite $(\Sigma \cup \{\perp \mapsto 0\})$-labelled ranked tree, defined by

$$t^\perp = \begin{cases} a\ t_1^\perp \cdots t_n^\perp & (\text{if } t = a\ t_1\ \cdots\ t_n) \\ \perp & (\text{otherwise}) \end{cases}$$

$L^\perp$ is the set $\{T \mid T \preceq T' \in L\}$, where $T \preceq T'$ means that $T$ is obtained from $T'$ by replacing some subtrees of $T'$ with $\perp$.

Note that, the semantics is non-deterministic in three ways:

1. E-Gen generates a tree in $L$ non-deterministically,
2. E-Coerce-Assert and E-Coerce-Input may be applied non-deterministically to **coerce**$^L(t)$, and
3. a term may have more than one redexes (e.g., **o2i**$(a\ t)$ has redexes **o2i**$(a\ t)$ and $t$).

**Example 3.2.** Recall $\mathcal{P}_{rev}$ in Example 3.1. *Reverse* (a (a (b e))) evaluates as follows:

$$\begin{aligned}
& \textit{Reverse}\ (\mathtt{a}\ (\mathtt{a}\ (\mathtt{b}\ \mathtt{e}))) \\
\longrightarrow\ & \textbf{case}\ \mathtt{a}\ (\mathtt{a}\ (\mathtt{b}\ \mathtt{e}))\ \textbf{of}\ \cdots && \text{(By E-App)} \\
\longrightarrow\ & \textit{Append}\ (\textbf{coerce}^{\mathtt{b}^*\mathtt{a}^*\mathtt{e}}(\textit{Reverse}\ (\mathtt{a}\ (\mathtt{b}\ \mathtt{e}))))\ (\mathtt{a}\ \mathtt{e}) && \text{(By E-Case)} \\
\longrightarrow\ & \textbf{case}\ \textbf{coerce}^{\mathtt{b}^*\mathtt{a}^*\mathtt{e}}(\textit{Reverse}\ (\mathtt{a}\ (\mathtt{b}\ \mathtt{e})))\ \textbf{of}\ \cdots && \text{(By E-App)} \\
\longrightarrow\ & \textbf{case}\ \textbf{o2i}(\textit{Reverse}\ (\mathtt{a}\ (\mathtt{b}\ \mathtt{e})))\ \textbf{of}\ \cdots && \text{(By E-Coerce-Input)} \\
\longrightarrow^*\ & \textbf{case}\ \textbf{o2i}(\mathtt{b}\ (\mathtt{a}\ \mathtt{e}))\ \textbf{of}\ \cdots \\
\longrightarrow\ & \textbf{case}\ \underline{\mathtt{b}}\ \textbf{o2i}(\mathtt{a}\ \mathtt{e})\ \textbf{of}\ \cdots && \text{(By E-Input)} \\
\longrightarrow\ & \mathtt{b}\ (\textit{Append}\ \textbf{o2i}(\mathtt{a}\ \mathtt{e})\ (\mathtt{a}\ \mathtt{e})) && \text{(By E-Case)} \\
\longrightarrow^*\ & \mathtt{b}\ (\mathtt{a}\ (\mathtt{a}\ \mathtt{e}))
\end{aligned}$$

□

The goal of our verification is to check that a given EHMTT conforms to a given specification on input and output. As EHMTTs manipulate infinite trees, we use top-down tree automata called *trivial automata* (Aehlig *et al.* 2005) (which are Büchi tree automata with a trivial acceptance condition) as specifications (as well as for annotations $L$ in **coerce**$^L(\cdot)$ and **gen**$^L$).

**Definition 3.2 (trivial automaton).** A *trivial automaton* $\mathcal{M}$ is a quadruple $(\Sigma, Q, \Delta, q_0)$, where

— $\Sigma$ is a ranked alphabet.
— $Q$ is a finite set of states.
— $\Delta$ is a finite subset of $Q \times \mathrm{dom}(\Sigma) \times Q^*$ called a *transition relation* such that if $(q, a, \widetilde{q}) \in \Delta$, then the length of the sequence $\widetilde{q}$ is $\Sigma(a)$.
— $q_0$ is a state called an *initial state*.

A $\Sigma$-labelled ranked tree $T$ is *accepted* by $\mathcal{M}$ if there is a $Q$-labelled tree $R$ such that:

— $\mathrm{dom}(T) = \mathrm{dom}(R)$.
— For any $\pi \in \mathrm{dom}(R)$, $(R(\pi), T(\pi), R(\pi \cdot 1) \cdots R(\pi \cdot \Sigma(T(\pi)))) \in \Delta$.
— $R(\epsilon) = q_0$.

We write $\mathcal{L}(\mathcal{M})$ for the set of $\Sigma$-labelled ranked trees accepted by $\mathcal{M}$. □

When restricted to finite trees, the class of languages recognized by trivial automata is equivalent to the class of regular tree languages.

**Example 3.3.** Recall Example 3.1. A trivial automaton for accepting $\mathtt{b}^*\mathtt{a}^*\mathtt{e}$ is defined by $(\Sigma, \{q_0, q_1\}, \Delta, q_0)$, where

$$\begin{aligned}\Sigma &= \{\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 1, \mathtt{e} \mapsto 0\}\\ \Delta &= \{(q_0, \mathtt{b}, q_0), (q_0, \mathtt{a}, q_1), (q_0, \mathtt{e}, \epsilon), (q_1, \mathtt{a}, q_1), (q_1, \mathtt{e}, \epsilon)\}.\end{aligned}$$

□

We now formalize our verification problem.

**Definition 3.3.** Given an EHMTT $\mathcal{P} = (D, S)$ and trivial automata $\mathcal{M}_1, \ldots, \mathcal{M}_k$, $\mathcal{M} = (\Sigma, Q, \Delta, q_0)$, we write $\models (\mathcal{P}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})$ if for all $T_1 \in \mathcal{L}(\mathcal{M}_1), \ldots, T_k \in \mathcal{L}(\mathcal{M}_k)$,

1 $S\ T_1 \cdots T_k \longrightarrow^*_{\mathcal{P}} t$ implies $t^\perp \in \mathcal{L}(\mathcal{M}^\perp)$, and
2 $S\ T_1 \cdots T_k \not\longrightarrow^*_{\mathcal{P}}$ **Error**.

Here, $\mathcal{M}^\perp$ is the trivial automaton $(\Sigma \cup \{\perp \mapsto 0\}, Q, \Delta \cup \{(q, \perp, \epsilon) \mid q \in Q\}, q_0)$. An *EHMTT verification problem* $(\mathcal{P}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})$ is the problem to check that $\models (\mathcal{P}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})$. □

The first condition of an EHMTT verification problem says that given input trees that conform to the input specification, the EHMTT generates a tree that conforms to the output specification†. The second condition means that the EHMTT never causes a coercion error. Let $L_1, \ldots, L_k, L$ be tree languages accepted respectively by trivial automata $\mathcal{M}_1, \ldots \mathcal{M}_k, \mathcal{M}$. We shall often write $(\mathcal{P}, L_1, \ldots, L_k, L)$ instead of $(\mathcal{P}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})$.

**Remark 1.** Note that EHMTT is call-by-name. This is because our verification method is based on the model checking of higher-order recursion schemes, whose semantics is call-by-name. To deal with call-by-value programs, it suffices to apply CPS transformation before applying our verification method. The reasons why we allow infinite trees as inputs and outputs for EHMTTs are as follows. First, we would like to verify programs that manipulate not only finite but also infinite data structures (such as streams). Second, we would like to model a program that contains non-deterministic branches (which is typically obtained by abstracting branching information of a user program) as an EHMTT that generates a single tree describing all the possible outputs of the program. In that case, even if a program manipulates only finite data structures, the output of the EHMTT can be an infinite tree.

In Kobayashi *et al.* (2010), we have presented a (sound but incomplete) verification method for the restricted case (which we call *HMTT verification problems*), where $\mathcal{P}$ is an HMTT (i.e., for the case, where $\mathcal{P}$ does not contain $\mathbf{coerce}^L(\cdot)$). In the next section, we reduce an EHMTT verification problem to HMTT verification problems.

**Example 3.4.** Recall $\mathcal{P}_{rev}$ in Example 3.1. $(\mathcal{P}_{rev}, \mathtt{a}^*\mathtt{b}^*\mathtt{e}, \mathtt{b}^*\mathtt{a}^*\mathtt{e})$ is an EHMTT verification problem. We use this as a running example in the next section. □

† Because of the presence of $\perp$, only safety properties are guaranteed; there is no guarantee that the EHMTT eventually generates a tree that belongs to $\mathcal{L}(\mathcal{M})$.

## 4. Verification method for EHMTTs

We now present a method for reducing an EHMTT verification problem to HMTT verification problems, which can then be solved by the previous method (Kobayashi *et al.* 2010). The idea is to reduce each of the two conditions in Definition 3.3 to HMTT verification problems.

Let $(\mathcal{P}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})$ be a given EHMTT verification problem, and suppose that $\mathcal{P}$ contains $m$ occurrences of coercions: $\mathbf{coerce}^{L_1}(\cdot), \ldots, \mathbf{coerce}^{L_m}(\cdot)$. We construct HMTTs $\mathcal{P}^{\mathcal{A}}, \mathcal{P}^{\mathcal{B}_1}, \ldots, \mathcal{P}^{\mathcal{B}_m}$ such that:

— $\mathcal{P}^{\mathcal{A}}$ approximates the output of $\mathcal{P}$, by assuming that coercions never fail.
— $\mathcal{P}^{\mathcal{B}_i}$ approximates all the possible arguments of $\mathbf{coerce}^{L_i}(\cdot)$.

Then, $(\mathcal{P}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})$ can be reduced to $m+1$ HMTT verification problems:

$$(\mathcal{P}^{\mathcal{A}}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M}), (\mathcal{P}^{\mathcal{B}_1}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{B}(L_1)), \ldots, (\mathcal{P}^{\mathcal{B}_m}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{B}(L_m))$$

(where $\mathcal{B}(L)$ is an automaton for accepting trees representing subsets of $L$; see Section 4.2 below). Sections 4.1 and 4.2 below show the constructions of $\mathcal{P}^{\mathcal{A}}$ and $\mathcal{P}^{\mathcal{B}_i}$, respectively. Sections 4.3 and 4.4 discuss properties of our EHMTT verification method.

### 4.1. *Construction of* $\mathcal{P}^{\mathcal{A}}$

Let $\mathcal{P}^{\mathcal{A}}$ be the HMTT obtained by just replacing every occurrence of $\mathbf{coerce}^{L_i}(\cdot)$ in $\mathcal{P}$ with $\mathbf{gen}^{L_i}$. Then, $\mathcal{P}^{\mathcal{A}}$ approximates the output of $\mathcal{P}$, assuming that no coercion error occurs.

**Example 4.1.** Recall $\mathcal{P}_{rev}$ in Example 3.1. ${\mathcal{P}_{rev}}^{\mathcal{A}}$ is $(D, \mathit{Reverse}^{\mathcal{A}})$ where $D$ is given by

$$\begin{aligned}
\mathit{Reverse}^{\mathcal{A}}\ x^{\mathcal{A}} \ =\ & \mathbf{case}\ x^{\mathcal{A}}\ \mathbf{of}\ \mathtt{e} \Rightarrow \mathtt{e} \\
& \mid \mathtt{a}\ x'^{\mathcal{A}} \Rightarrow \mathit{Append}^{\mathcal{A}}\ \mathbf{gen}^{\mathtt{b}^*\mathtt{a}^*\mathtt{e}}\ (\mathtt{a}\ \mathtt{e}) \\
& \mid \mathtt{b}\ x'^{\mathcal{A}} \Rightarrow \mathit{Append}^{\mathcal{A}}\ \mathbf{gen}^{\mathtt{b}^*\mathtt{e}}\ (\mathtt{b}\ \mathtt{e}) \\
\mathit{Append}^{\mathcal{A}}\ x^{\mathcal{A}}\ y^{\mathcal{A}} \ =\ & \mathbf{case}\ x^{\mathcal{A}}\ \mathbf{of}\ \mathtt{e} \Rightarrow y^{\mathcal{A}} \\
& \mid \mathtt{a}\ x'^{\mathcal{A}} \Rightarrow \mathtt{a}\ (\mathit{Append}^{\mathcal{A}}\ x'^{\mathcal{A}}\ y^{\mathcal{A}}) \\
& \mid \mathtt{b}\ x'^{\mathcal{A}} \Rightarrow \mathtt{b}\ (\mathit{Append}^{\mathcal{A}}\ x'^{\mathcal{A}}\ y^{\mathcal{A}}).
\end{aligned}$$

□

Formally, given an EHMTT $\mathcal{P} = (D, S)$, $\mathcal{P}^{\mathcal{A}}$ is $(\mathcal{A}(D), S^{\mathcal{A}})$ where

$$\begin{aligned}
&\mathcal{A}(D) = \{F^{\mathcal{A}}\ \widetilde{x}^{\mathcal{A}} = \mathcal{A}(t) \mid F\ \widetilde{x} = t \in D\} \\
&\mathcal{A}(a) = a \qquad \mathcal{A}(x) = x^{\mathcal{A}} \\
&\mathcal{A}(F) = F^{\mathcal{A}} \qquad \mathcal{A}(t_1\ t_2) = \mathcal{A}(t_1)\ \mathcal{A}(t_2) \\
&\mathcal{A}(\mathbf{case}\ t\ \mathbf{of}\ \{a_i\ \widetilde{y}_i \Rightarrow t_i\}_{i=1}^n) = \mathbf{case}\ \mathcal{A}(t)\ \mathbf{of}\ \{a_i\ \widetilde{y}_i^{\mathcal{A}} \Rightarrow \mathcal{A}(t_i)\}_{i=1}^n \\
&\mathcal{A}(\mathbf{gen}^L) = \mathbf{gen}^L \qquad \mathcal{A}(\mathbf{coerce}^L(t)) = \mathbf{gen}^L.
\end{aligned}$$

### 4.2. *Construction of* $\mathcal{P}^{\mathcal{B}_i}$

The construction of $\mathcal{P}^{\mathcal{B}_i}$ is more involved, for the following reasons.

1. Given an input, $\mathcal{P}$ may invoke $\mathbf{coerce}^{L_i}(\cdot)$ more than once. For example, given $\mathtt{b(b(e))}$ as input, $\mathcal{P}_{rev}$ in Example 3.1 invoke $\mathbf{coerce}^{\mathtt{b}^*\mathtt{e}}(\cdot)$ twice, with different parameters $\mathtt{e}$ and $\mathtt{b(e)}$. Thus, $\mathcal{P}^{\mathcal{B}_i}$ should approximate the *set* of trees that are passed to $\mathbf{coerce}^{L_i}(\cdot)$.
2. How a function invokes $\mathbf{coerce}^{L_i}(\cdot)$ may depend on its arguments. For example, consider a higher-order function $F$ defined by $F\ g\ x = g(x)$. Obviously, how $\mathbf{coerce}^{L_i}(\cdot)$ is invoked during evaluation of $F\ t_1\ t_2$ depends on $t_1$ and $t_2$.

To address the first issue, we represent a (possibly infinite) set of trees by a single (possibly infinite) tree. We use special terminal symbols $\mathtt{br}$ and $\mathtt{emp}$, which represent the set union and an empty set, respectively. For example, the set $\{\mathtt{e}, \mathtt{b(e)}\}$ is represented by $\mathtt{br\ e\ (b(e))}$. $\mathcal{P}^{\mathcal{B}_i}$ outputs such a tree representation of (an over-approximation of) the set of trees passed to $\mathbf{coerce}^{L_i}(\cdot)$.

To address the second issue, we duplicate each parameter $x$ of a function into $x^{\mathcal{A}}$ and $x^{\mathcal{B}}$. The parameter $x^{\mathcal{A}}$ is used to compute (an approximation of) the original value of $x$, while the parameter $x^{\mathcal{B}}$ computes (an approximation of) the set of trees passed to $\mathbf{coerce}^{L_i}(\cdot)$ during evaluation of $x$. For example, $F\ g\ x = g(x)$ is transformed to $F^{\mathcal{B}}\ g^{\mathcal{A}}\ g^{\mathcal{B}}\ x^{\mathcal{A}}\ x^{\mathcal{B}} = g^{\mathcal{B}}\ x^{\mathcal{A}}\ x^{\mathcal{B}}$. Here, $F^{\mathcal{B}}$ computes an approximation of the set of trees passed to $\mathbf{coerce}^{L_i}(\cdot)$ by calling $g^{\mathcal{B}}$ with duplicated parameters $x^{\mathcal{A}}$ and $x^{\mathcal{B}}$.

We give below more concrete examples to explain the construction of $\mathcal{P}^{\mathcal{B}_i}$.

**Example 4.2.** Recall $\mathcal{P}_{rev}$ in Example 3.1. For the first coercion $\mathbf{coerce}^{L_1}(\mathit{Reverse}\ x')$ (where $L_1 = \mathtt{b}^*\mathtt{a}^*\mathtt{e}$), we construct the following HMTT ${\mathcal{P}_{rev}}^{\mathcal{B}}$:

$$
\begin{array}{l}
S\ x = \mathit{Reverse}^{\mathcal{B}}\ x\ \mathtt{emp} \\
\mathit{Reverse}^{\mathcal{B}}\ x^{\mathcal{A}}\ x^{\mathcal{B}} = \\
\quad \mathtt{br}\ x^{\mathcal{B}}\ (\mathbf{case}\ x^{\mathcal{A}}\ \mathbf{of}\ \mathtt{e} \Rightarrow \mathtt{emp} \\
\qquad\qquad\qquad\qquad \mid \mathtt{a}\ x'^{\mathcal{A}} \Rightarrow \mathit{Append}^{\mathcal{B}}\ \mathbf{gen}^{\mathtt{b}^*\mathtt{a}^*\mathtt{e}}\ (\mathtt{br}\ (\mathit{Reverse}^{\mathcal{A}}\ x'^{\mathcal{A}}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Reverse}^{\mathcal{B}}\ x'^{\mathcal{A}}\ x^{\mathcal{B}})) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathtt{a}\ \mathtt{e})\ \mathtt{emp} \\
\qquad\qquad\qquad\qquad \mid \mathtt{b}\ x'^{\mathcal{A}} \Rightarrow \mathit{Append}^{\mathcal{B}}\ \mathbf{gen}^{\mathtt{b}^*\mathtt{e}}\ (\mathit{Reverse}^{\mathcal{B}}\ x'^{\mathcal{A}}\ x^{\mathcal{B}}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathtt{b}\ \mathtt{e})\ \mathtt{emp}) \\
\mathit{Append}^{\mathcal{B}}\ x^{\mathcal{A}}\ x^{\mathcal{B}}\ y^{\mathcal{A}}\ y^{\mathcal{B}} = \\
\quad \mathtt{br}\ x^{\mathcal{B}}\ (\mathbf{case}\ x^{\mathcal{A}}\ \mathbf{of}\ \mathtt{e} \Rightarrow y^{\mathcal{B}} \\
\qquad\qquad\qquad\qquad \mid \mathtt{a}\ x'^{\mathcal{A}} \Rightarrow \mathit{Append}^{\mathcal{B}}\ x'^{\mathcal{A}}\ x^{\mathcal{B}}\ y^{\mathcal{A}}\ y^{\mathcal{B}} \\
\qquad\qquad\qquad\qquad \mid \mathtt{b}\ x'^{\mathcal{A}} \Rightarrow \mathit{Append}^{\mathcal{B}}\ x'^{\mathcal{A}}\ x^{\mathcal{B}}\ y^{\mathcal{A}}\ y^{\mathcal{B}}).
\end{array}
$$

Here, $S$ is the main function of ${\mathcal{P}_{rev}}^{\mathcal{B}}$. As mentioned above, the parameters of *Reverse* and *Append* have been duplicated. When *Reverse* $t$ is called in $\mathcal{P}_{rev}$, there are two cases where $\mathbf{coerce}^{L_1}(\cdot)$ may be called: the case where $t$ contains $\mathbf{coerce}^{L_1}(\cdot)$ and it is called when $t$ is evaluated by the case statement (note that our language is call-by-name); and the case where $\mathbf{coerce}^{L_1}(\cdot)$ is called in a case branch. In the body of the definition of $\mathit{Reverse}^{\mathcal{B}}$, the

part $x^{\mathcal{B}}$ approximates the set of trees passed to $\mathbf{coerce}^{L_1}(\cdot)$ in the former case, and the part $\mathbf{case}\ x^{\mathcal{A}}\ \mathbf{of}\ \cdots$ approximates the set of trees for the latter case.

In the clause for $\mathtt{a}\ x'^{\mathcal{A}}$, $Append^{\mathcal{B}}$ is used to compute an approximation of the set of trees passed to $\mathbf{coerce}^{L_1}(\cdot)$. The first and third parameters approximate the values of the original parameters of *Append*. The second parameter $(\mathtt{br}\ (Reverse^{\mathcal{A}}\ x'^{\mathcal{A}})\ (Reverse^{\mathcal{B}}\ x'^{\mathcal{A}}\ x^{\mathcal{B}}))$ of $Append^{\mathcal{B}}$ approximates the set of trees passed to $\mathbf{coerce}^{L_1}(\cdot)$ during the computation of $\mathbf{coerce}^{L_1}(Reverse\ x')$. Here, there are two cases where coercion can occur: (i) the value of $Reverse\ x'$ is computed and passed to $\mathbf{coerce}^{L_1}(\cdot)$ and (ii) $\mathbf{coerce}^{L_1}(\cdot)$ is invoked during the computation of $Reverse\ x'$. The parts $(Reverse^{\mathcal{A}}\ x'^{\mathcal{A}})$ and $(Reverse^{\mathcal{B}}\ x'^{\mathcal{A}}\ x^{\mathcal{B}})$ cover the former and the latter cases, respectively. In the latter, the second parameter of $Reverse^{\mathcal{B}}$ is $x^{\mathcal{B}}$,[†] as the trees passed to $\mathbf{coerce}^{L_1}(\cdot)$ during the computation of $x'$ are covered by $x^{\mathcal{B}}$.

□

The reduction works similarly for EHMTTs with higher-order functions.

**Example 4.3.** Let us consider a higher-order version of the list reverse program:

$$
\begin{aligned}
Reverse\ x &= Reverseh\ Append\ x\\
Reverseh\ f\ x &= \mathbf{case}\ x\ \mathbf{of}\ \mathtt{e} \Rightarrow \mathtt{e}\\
&\qquad \mid \mathtt{a}\ x' \Rightarrow f\ (\mathbf{coerce}^{\mathtt{b}^*\mathtt{a}^*\mathtt{e}}(Reverseh\ f\ x'))\ (\mathtt{a}\ \mathtt{e})\\
&\qquad \mid \mathtt{b}\ x' \Rightarrow f\ (\mathbf{coerce}^{\mathtt{b}^*\mathtt{e}}(Reverseh\ f\ x'))\ (\mathtt{b}\ \mathtt{e}).
\end{aligned}
$$

We get the following HMTT for the first coercion $\mathbf{coerce}^{\mathtt{b}^*\mathtt{a}^*\mathtt{e}}(Reverseh\ f\ x')$:

$$
\begin{aligned}
&S\ x = Reverse^{\mathcal{B}}\ x\ \mathtt{emp}\\
&Reverse^{\mathcal{B}}\ x^{\mathcal{A}}\ x^{\mathcal{B}} = Reverseh^{\mathcal{B}}\ Append^{\mathcal{A}}\ Append^{\mathcal{B}}\ x^{\mathcal{A}}\ x^{\mathcal{B}}\\
&Reverseh^{\mathcal{B}}\ f^{\mathcal{A}}\ f^{\mathcal{B}}\ x^{\mathcal{A}}\ x^{\mathcal{B}} =\\
&\quad \mathtt{br}\ x^{\mathcal{B}}\ (\mathbf{case}\ x^{\mathcal{A}}\ \mathbf{of}\ \mathtt{e} \Rightarrow \mathtt{emp}\\
&\qquad\qquad \mid \mathtt{a}\ x'^{\mathcal{A}} \Rightarrow f^{\mathcal{B}}\ \mathbf{gen}^{\mathtt{b}^*\mathtt{a}^*\mathtt{e}}\ (\mathtt{br}\ (Reverseh^{\mathcal{A}}\ f^{\mathcal{A}}\ x'^{\mathcal{A}})\\
&\qquad\qquad\qquad\qquad\qquad\qquad (Reverseh^{\mathcal{B}}\ f^{\mathcal{A}}\ f^{\mathcal{B}}\ x'^{\mathcal{A}}\ x^{\mathcal{B}}))\\
&\qquad\qquad\qquad (\mathtt{a}\ \mathtt{e})\ \mathtt{emp}\\
&\qquad\qquad \mid \mathtt{b}\ x'^{\mathcal{A}} \Rightarrow f^{\mathcal{B}}\ \mathbf{gen}^{\mathtt{b}^*\mathtt{e}}\ (Reverseh^{\mathcal{B}}\ f^{\mathcal{A}}\ f^{\mathcal{B}}\ x'^{\mathcal{A}}\ x^{\mathcal{B}})\\
&\qquad\qquad\qquad (\mathtt{b}\ \mathtt{e})\ \mathtt{emp}).
\end{aligned}
$$

Here, $S$ is the main function and $Append^{\mathcal{A}}$ is the one obtained in Example 4.1. Note that, $Reverseh^{\mathcal{B}}$ requires an additional argument $f^{\mathcal{B}}$, which generates all the trees passed to the coercion by $f$. □

[†] Actually, the second parameter of $Reverse^{\mathcal{B}}$ can be replaced with an empty set, since we have the part $\mathtt{br}\ x^{\mathcal{B}}(\cdots)$ in the definition of $Reverse^{\mathcal{B}}$.

Formally, given an EHMTT $\mathcal{P} = (D, S)$, $\mathcal{P}^{\mathcal{B}_i}$ is $(\mathcal{B}_i(D), S)$ where

$$\begin{aligned}\mathcal{B}_i(D) = {} & \{S\ x_1 \cdots x_k = S^{\mathcal{B}_i}\ x_1\ \mathtt{emp} \cdots x_k\ \mathtt{emp}\} \cup \\ & \{a^{\mathcal{B}_i}\ x_1{}^{\mathcal{A}}\ x_1{}^{\mathcal{B}_i} \cdots x_{\Sigma(a)}{}^{\mathcal{A}}\ x_{\Sigma(a)}{}^{\mathcal{B}_i} = \mathtt{br}\ x_1{}^{\mathcal{B}_i} \cdots x_{\Sigma(a)}{}^{\mathcal{B}_i} \mid a \in \mathrm{dom}(\Sigma)\} \cup \\ & \{F^{\mathcal{A}}\ x_1{}^{\mathcal{A}} \cdots x_n{}^{\mathcal{A}} = \mathcal{A}(t) \mid F\ x_1 \cdots x_n = t \in D\} \cup \\ & \{F^{\mathcal{B}_i}\ x_1{}^{\mathcal{A}}\ x_1{}^{\mathcal{B}_i} \cdots x_n{}^{\mathcal{A}}\ x_n{}^{\mathcal{B}_i} = \mathcal{B}_i(t) \mid F\ x_1 \cdots x_n = t \in D\}\end{aligned}$$

$$\begin{aligned}&\mathcal{B}_i(a) = a^{\mathcal{B}_i} \qquad \mathcal{B}_i(x) = x^{\mathcal{B}_i} \\ &\mathcal{B}_i(F) = F^{\mathcal{B}_i} \qquad \mathcal{B}_i(t_1\ t_2) = \mathcal{B}_i(t_1)\ \mathcal{A}(t_2)\ \mathcal{B}_i(t_2) \\ &\mathcal{B}_i(\mathbf{case}\ t\ \mathbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow t_j\}_{j=1}^n) = \\ &\qquad \mathtt{br}\ \mathcal{B}_i(t)\ (\mathbf{case}\ \mathcal{A}(t)\ \mathbf{of}\ \{a_j\ \widetilde{y}_j^{\mathcal{A}} \Rightarrow [\mathcal{B}_i(t), \ldots, \mathcal{B}_i(t)/\widetilde{y}_j^{\mathcal{B}_i}]\mathcal{B}_i(t_j)\}_{j=1}^n) \\ &\mathcal{B}_i(\mathbf{gen}^L) = \mathtt{emp} \qquad \mathcal{B}_i(\mathbf{coerce}^{L_j}(t)) = \begin{cases} \mathtt{br}\ \mathcal{A}(t)\ \mathcal{B}_i(t) & (\text{if } i = j) \\ \mathcal{B}_i(t) & (\text{otherwise}). \end{cases}\end{aligned}$$

Here, $\mathtt{br}\ t_1 \cdots t_n$ stands for $\mathtt{br}\ t_1\ (\mathtt{br}\ t_2\ (\mathtt{br} \cdots (\mathtt{br}\ t_{n-1}\ t_n)))$ if $n \geqslant 2$, $t_1$ if $n = 1$, and $\mathtt{emp}$ if $n = 0$. For each terminal $a \in \mathrm{dom}(\Sigma)$, we prepare the new function $a^{\mathcal{B}_i}$ that generates all the trees passed to the $i$th coercion by the actual arguments of $a$.

Given a trivial automaton $\mathcal{M}(L_i) = (\Sigma, Q, \Delta, q_0)$ for accepting $L_i$, the output specification for $\mathcal{P}^{\mathcal{B}_i}$ is the trivial automaton $\mathcal{B}(L_i) = (\Sigma', Q, \Delta', q_0)$, where

$$\begin{aligned}&\Sigma' = \Sigma \cup \{\mathtt{br} \mapsto 2, \mathtt{emp} \mapsto 0\} \\ &\Delta' = \Delta \cup \{(q, \mathtt{br}, q \cdot q), (q, \mathtt{emp}, \epsilon) \mid q \in Q\}.\end{aligned}$$

### 4.3. *Soundness*

Our overall EHMTT verification method, which consists of the reduction from EHMTT to HMTTs given above and the HMTT verification method (Kobayashi *et al.* 2010), is sound. The soundness of the latter step is shown by Kobayashi *et al.* (2010). We prove the soundness of the former step below.

Let $\mathcal{P} = (D, S)$ be an EHMTT and $\mathbf{coerce}^{L_1}(\cdot), \ldots, \mathbf{coerce}^{L_m}(\cdot)$ be the occurrences of coercions in $\mathcal{P}$. Then, we prove the following theorems stating the correctness of the constructions of $\mathcal{P}^{\mathcal{A}}$ and $\mathcal{P}^{\mathcal{B}_i}$, respectively.

**Theorem 4.1.** Suppose that $\models (\mathcal{P}^{\mathcal{A}}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})$ holds. For any $T_1 \in \mathcal{L}(\mathcal{M}_1), \ldots, T_k \in \mathcal{L}(\mathcal{M}_k)$, if $S\ T_1 \cdots T_k \not\longrightarrow_{\mathcal{P}}^* \mathbf{Error}$ and $S\ T_1 \cdots T_k \longrightarrow_{\mathcal{P}}^* t$, then $t^{\perp} \in \mathcal{L}(\mathcal{M}^{\perp})$.

**Theorem 4.2.** Suppose that $\models (\mathcal{P}^{\mathcal{B}_i}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{B}(L_i))$ holds for each $i \in \{1, \ldots, m\}$. Then, for any $T_1 \in \mathcal{L}(\mathcal{M}_1), \ldots, T_k \in \mathcal{L}(\mathcal{M}_k)$, $S\ T_1 \cdots T_k \not\longrightarrow_{\mathcal{P}}^* \mathbf{Error}$.

The soundness of our EHMTT verification method follows immediately from Theorems 4.1 and 4.2.

**Corollary 4.3.** If $\models (\mathcal{P}^{\mathcal{A}}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})$ and $\models (\mathcal{P}^{\mathcal{B}_i}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{B}(L_i))$ hold for each $i \in \{1, \ldots, m\}$, then $\models (\mathcal{P}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})$ holds.

4.3.1. *Proof of Theorem 4.1.* The idea of the proof is to show that if coercions in $\mathcal{P}$ never fail, an evaluation $S\ T_1 \cdots T_k \longrightarrow_{\mathcal{P}}^* t$ in $\mathcal{P}$ (where $t$ is not of the form $\mathbf{assert}^L(t'')$) can be simulated by some evaluation $S^{\mathcal{A}}\ T_1 \cdots T_k \longrightarrow_{\mathcal{P}^{\mathcal{A}}}^* t'$ in $\mathcal{P}^{\mathcal{A}}$ for some $t'$ such that $t'^{\perp} = t^{\perp}$.

For establishing the simulation, we use annotated terms $\mathbf{o2i}^L(t)$ instead of $\mathbf{o2i}(t)$, and accordingly introduce a variant $\Longrightarrow$ of the evaluation relation $\longrightarrow$, which adopts the following rules instead of the rules E-Coerce-Input and E-Input for $\longrightarrow$:

$$E[\mathbf{coerce}^L(t)] \Longrightarrow_{\mathcal{P}} E[\mathbf{o2i}^L(t)] \qquad \text{(E-Coerce-InputL)}$$

$$\frac{a\ L_1 \cdots L_n \subseteq L \qquad L_1, \ldots, L_n \neq \varnothing}{E[\mathbf{o2i}^L(a\ t_1 \cdots t_n)] \Longrightarrow_{\mathcal{P}} E[\underline{a}\ \mathbf{o2i}^{L_1}(t_1) \cdots \mathbf{o2i}^{L_n}(t_n)]} \qquad \text{(E-InputL)}$$

Then, it is easy to show that an evaluation in $\Longrightarrow$ can be simulated by some evaluation in $\longrightarrow$.

**Lemma 4.4.** For any $T_1 \in \mathcal{L}(\mathcal{M}_1), \ldots, T_k \in \mathcal{L}(\mathcal{M}_k)$, if $S\ T_1 \cdots T_k \Longrightarrow^* t$, then $S\ T_1 \cdots T_k \longrightarrow^* \mathit{Strip}(t)$.

Here, $\mathit{Strip}(t)$ is the term obtained from $t$ by replacing each occurrence of $\mathbf{o2i}^L(t')$ with $\mathbf{o2i}(\mathit{Strip}(t'))$. Conversely, an evaluation in $\longrightarrow$ can be simulated by some evaluation in $\Longrightarrow$ if coercions in $\mathcal{P}$ never fail.

**Lemma 4.5.** For any $T_1 \in \mathcal{L}(\mathcal{M}_1), \ldots, T_k \in \mathcal{L}(\mathcal{M}_k)$, if $S\ T_1 \cdots T_k \not\longrightarrow^* \mathbf{Error}$ and $S\ T_1 \cdots\ T_k \longrightarrow^* t_1$, then $S\ T_1 \cdots\ T_k \Longrightarrow^* t_2$ for some $t_2$ such that $t_1 = \mathit{Strip}(t_2)$.

*Proof.* We prove the lemma by induction on the derivation of $S\ T_1 \cdots T_k \longrightarrow^* t_1$. If $t_1 = S\ T_1 \cdots T_k$, let $t_2 = S\ T_1 \cdots T_k$. Then, we obtain $S\ T_1 \cdots T_k \Longrightarrow^* t_2$ and $t_1 = \mathit{Strip}(t_2)$. Otherwise, suppose that $S\ T_1 \cdots T_k \longrightarrow^* t_1' \longrightarrow t_1$. By the induction hypothesis, there is $t_2'$ such that $S\ T_1 \cdots T_k \Longrightarrow^* t_2'$ and $t_1' = \mathit{Strip}(t_2')$. We perform case analysis on the rule used to derive $t_1' \longrightarrow t_1$. We consider only the case for E-Input below since the other cases are straightforward. We have $t_1' = E[\mathbf{o2i}(a\ \widetilde{t})]$ and $t_1 = E[\underline{a}\ \mathbf{o2i}(\widetilde{t})]$. Thus, it follows that $t_2' = E'[\mathbf{o2i}^L(a\ \widetilde{t'})]$ for some $L$, $E'$, and $\widetilde{t'}$ such that $\mathit{Strip}(E') = E$ and $\mathit{Strip}(\widetilde{t'}) = \widetilde{t}$. Note that, $L$ may not contain a tree of the form $a\ \widetilde{T}$ and we may not be able to apply E-InputL to $t_2'$. However, we can show that $S\ T_1 \cdots T_k \Longrightarrow^* E'[\mathbf{o2i}^{(a\ \widetilde{L})}(a\ \widetilde{t'})]$ for some $\widetilde{L} \neq \widetilde{\varnothing}$ as follows. By $S\ T_1 \cdots T_k \Longrightarrow^* t_2'$, there exist $E''$, $L'$, $t'$, and context $C$ such that:

— $S\ T_1 \cdots T_k \Longrightarrow^* E''[\mathbf{coerce}^{L'}(t')] \Longrightarrow E''[\mathbf{o2i}^{L'}(t')] \Longrightarrow^* E'[\mathbf{o2i}^L(a\ \widetilde{t'})] = t_2'$,
— every node in the path of $C$ from the root to the hole [ ] is a terminal symbol, and
— $S\ T_1 \cdots T_k \Longrightarrow^* E''[\mathbf{coerce}^{L'}(t')] \Longrightarrow \mathbf{assert}^{L'}(t') \Longrightarrow^* \mathbf{assert}^{L'}(C[a\ \widetilde{t'}])$.

Here, $\mathbf{o2i}^L(a\ \widetilde{t'})$ is obtained from $\mathbf{o2i}^{L'}(t')$ by repeatedly applying E-InputL to it. Thus, by $S\ T_1 \cdots T_k \not\Longrightarrow^* \mathbf{Error}$ (which follows from Lemma 4.4 and $S\ T_1 \cdots T_k \not\longrightarrow^* \mathbf{Error}$), we get some $\widetilde{L} \neq \widetilde{\varnothing}$ such that:

— $\varnothing \subsetneq C^{\perp}[a\ \widetilde{L}] \subseteq L'^{\perp}$ and
— $S\ T_1 \cdots T_k \Longrightarrow^* E'[\mathbf{o2i}^{(a\ \widetilde{L})}(a\ \widetilde{t'})]$.

Let $t_2 = E'[\underline{a}\ \mathbf{o2i}^{\widetilde{L}}(\widetilde{t'})]$. Then, $t_1 = \mathit{Strip}(t_2)$ and $S\ T_1 \cdots T_k \Longrightarrow^* E'[\mathbf{o2i}^{(a\ \widetilde{L})}(a\ \widetilde{t'})] \Longrightarrow t_2$ by E-InputL. □

We extend $\mathcal{A}(t)$ to the run-time terms as follows:

$$\mathcal{A}(\underline{a}) = \underline{a} \qquad \mathcal{A}(\mathbf{o2i}^L(t)) = \mathbf{gen}^L.$$

We can naturally extend the definition of $\mathcal{A}(t)$ to evaluation contexts $E$ as follows:

$$\begin{aligned}
\mathcal{A}([\,]) &= [\,] \\
\mathcal{A}(a\ t_1 \cdots t_{i-1}\ E\ t_{i+1} \cdots t_{\Sigma(a)}) &= a\ \mathcal{A}(t_1) \cdots \mathcal{A}(t_{i-1})\ \mathcal{A}(E)\ \mathcal{A}(t_{i+1}) \cdots \mathcal{A}(t_{\Sigma(a)}) \\
\mathcal{A}(\mathbf{case}\ E\ \mathbf{of}\ \{a_i\ \widetilde{y}_i \Rightarrow t_i\}_{i=1}^n) &= \mathbf{case}\ \mathcal{A}(E)\ \mathbf{of}\ \{a_i\ \widetilde{y_i^{\mathcal{A}}} \Rightarrow \mathcal{A}(t_i)\}_{i=1}^n \\
\mathcal{A}(\mathbf{o2i}^L(E)) &= \mathbf{gen}^L.
\end{aligned}$$

We can show the following properties of $\mathcal{A}(t)$ and $\mathcal{A}(E)$ by a straightforward induction on the structure of $t$ or $E$.

**Lemma 4.6.**

1. $\mathcal{A}(t)^{\perp} = t^{\perp}$,
2. $\mathcal{A}([t'/x]t) = [\mathcal{A}(t')/x^{\mathcal{A}}]\mathcal{A}(t)$ and
3. $\mathcal{A}(E[t]) = \mathcal{A}(E)[\mathcal{A}(t)]$.

We now show that a one-step evaluation in $\mathcal{P}$ can be simulated by some evaluation in $\mathcal{P}^{\mathcal{A}}$.

**Lemma 4.7.** Suppose that $t \Longrightarrow t'$ where A-Coerce-Assert and A-Assert-Error are not applied. Then, $\mathcal{A}(t) \Longrightarrow^* \mathcal{A}(t')$ holds.

*Proof.* We prove the lemma by case analysis on the rule used to derive $t \Longrightarrow t'$:

— E-App: we have $t = E[F\ \widetilde{t}]$ and $t' = E[[\widetilde{t}/\widetilde{x}]t'']$, where $F\ \widetilde{x} = t'' \in \mathcal{D}$. We have $F^{\mathcal{A}}\ \widetilde{x}^{\mathcal{A}} = \mathcal{A}(t'') \in \mathcal{A}(D)$. Thus, we obtain:

$$\begin{aligned}
\mathcal{A}(t) &= \mathcal{A}(E)[F^{\mathcal{A}}\ \mathcal{A}(\widetilde{t})] && \text{(By Lemma 4.6(3))} \\
&\Longrightarrow^* \mathcal{A}(E)[[\mathcal{A}(\widetilde{t})/\widetilde{x}^{\mathcal{A}}]\mathcal{A}(t'')] && \text{(By E-App)} \\
&= \mathcal{A}(E)[\mathcal{A}([\widetilde{t}/\widetilde{x}]t'')] && \text{(By Lemma 4.6(2))} \\
&= \mathcal{A}(t') && \text{(By Lemma 4.6(3))}
\end{aligned}$$

— E-Case: we have $t = E[\mathbf{case}\ \underline{a_i}\ \widetilde{t}\ \mathbf{of}\ \{a_i\ \widetilde{y}_i \Rightarrow t_i\}_{i=1}^n]$ and $t' = E[[\widetilde{t}/\widetilde{y}_i]t_i]$. We obtain:

$$\begin{aligned}
\mathcal{A}(t) &= \mathcal{A}(E)[\mathbf{case}\ \underline{a_i}\ \mathcal{A}(\widetilde{t})\ \mathbf{of}\ \{a_i\ \widetilde{y_i^{\mathcal{A}}} \Rightarrow \mathcal{A}(t_i)\}_{i=1}^n] && \text{(By Lemma 4.6(3))} \\
&\Longrightarrow^* \mathcal{A}(E)[[\mathcal{A}(\widetilde{t})/\widetilde{y_i^{\mathcal{A}}}]\mathcal{A}(t_i)] && \text{(By E-Case)} \\
&= \mathcal{A}(E)[\mathcal{A}([\widetilde{t}/\widetilde{y}_i]t_i)] && \text{(By Lemma 4.6(2))} \\
&= \mathcal{A}(t') && \text{(By Lemma 4.6(3))}
\end{aligned}$$

— E-Case-Fail: we have $t = E[\mathbf{case}\ \underline{a}\ \widetilde{t}\ \mathbf{of}\ \{a_i\ \widetilde{y}_i \Rightarrow t_i\}_{i=1}^n]$ and $t' = E[\mathtt{fail}]$, where $a \notin \{a_1, \ldots, a_n\}$. We obtain:

$$\begin{aligned}
\mathcal{A}(t) &= \mathcal{A}(E)[\mathbf{case}\ \underline{a}\ \mathcal{A}(\widetilde{t})\ \mathbf{of}\ \{a_i\ \widetilde{y_i^{\mathcal{A}}} \Rightarrow \mathcal{A}(t_i)\}_{i=1}^n] && \text{(By Lemma 4.6(3))} \\
&\Longrightarrow^* \mathcal{A}(E)[\mathtt{fail}] && \text{(By E-Case-Fail)} \\
&= \mathcal{A}(t') && \text{(By Lemma 4.6(3))}
\end{aligned}$$

— E-GEN: we have $t = E[\mathbf{gen}^L]$ and $t' = E[\underline{a}\ \mathbf{gen}^{L_1} \dots \mathbf{gen}^{L_n}]$, where $a\ L_1 \cdots L_n \subseteq L$ and $L_1, \dots, L_n \neq \varnothing$. We get:

$$\begin{array}{rcll} \mathcal{A}(t) & = & \mathcal{A}(E)[\mathbf{gen}^L] & \text{(By Lemma 4.6(3))} \\ & \Longrightarrow^* & \mathcal{A}(E)[\underline{a}\ \mathbf{gen}^{L_1} \dots \mathbf{gen}^{L_n}] & \text{(By E-GEN)} \\ & = & \mathcal{A}(t') & \text{(By Lemma 4.6(3))} \end{array}$$

— E-COERCE-INPUTL: we have $t = E[\mathbf{coerce}^L(t'')]$ and $t' = E[\mathbf{o2i}^L(t'')]$. By Lemma 4.6(3), we obtain $\mathcal{A}(t) = \mathcal{A}(E)[\mathbf{gen}^L] = \mathcal{A}(t')$.

— E-INPUTL: we have $t = E[\mathbf{o2i}^L(a\ t_1 \cdots t_n)]$ and $t' = E[\underline{a}\ \mathbf{o2i}^{L_1}(t_1) \cdots \mathbf{o2i}^{L_n}(t_n)]$, where $a\ L_1 \cdots L_n \subseteq L$ and $L_1, \dots, L_n \neq \varnothing$. We obtain:

$$\begin{array}{rcll} \mathcal{A}(t) & = & \mathcal{A}(E)[\mathbf{gen}^L] & \text{(By Lemma 4.6(3))} \\ & \Longrightarrow^* & \mathcal{A}(E)[\underline{a}\ \mathbf{gen}^{L_1} \dots \mathbf{gen}^{L_n}] & \text{(By E-GEN)} \\ & = & \mathcal{A}(t') & \text{(By Lemma 4.6(3)).} \end{array}$$

□

We now prove Theorem 4.1.

Proof of Theorem 4.1. Suppose that $S\ T_1 \cdots T_k \not\longrightarrow^*$ **Error** and $S\ T_1 \cdots T_k \longrightarrow^* t$ for some $T_1 \in \mathcal{L}(\mathcal{M}_1), \dots, T_k \in \mathcal{L}(\mathcal{M}_k)$. If $t$ is of the form $\mathbf{assert}^L(t'')$, $t^\perp = \perp \in \mathcal{L}(\mathcal{M}^\perp)$ follows immediately. Otherwise, by Lemma 4.5, $S\ T_1 \cdots T_k \Longrightarrow^* t'$ is derivable for some $t'$ such that $t = \mathit{Strip}(t')$. Since $t'$ is neither of the form $\mathbf{assert}^L(t'')$ nor **Error**, E-COERCE-ASSERT and E-ASSERT-ERROR are not applied in the derivation of $S\ T_1 \cdots T_k \Longrightarrow^* t'$. By Lemma 4.7, we have $S^{\mathcal{A}}\ T_1 \cdots T_k \Longrightarrow^* \mathcal{A}(t')$. Thus, by Lemma 4.4, we get $S^{\mathcal{A}}\ T_1 \cdots T_k \longrightarrow^* \mathit{Strip}(\mathcal{A}(t'))$. Since $\models (\mathcal{P}^{\mathcal{A}}, \mathcal{M}_1, \dots, \mathcal{M}_k, \mathcal{M})$ holds, we get $\mathcal{A}(t')^\perp = \mathit{Strip}(\mathcal{A}(t'))^\perp \in \mathcal{L}(\mathcal{M}^\perp)$. Thus, by Lemma 4.6(1), we get $t^\perp = t'^\perp \in \mathcal{L}(\mathcal{M}^\perp)$. □

4.3.2. *Proof of Theorem 4.2.* The idea of the proof is to show that an evaluation $S\ T_1 \cdots T_k \longrightarrow^*_{\mathcal{P}} \mathbf{assert}^{L_j}(t)$ in $\mathcal{P}$ can be simulated by some evaluation $S\ T_1 \cdots T_k \longrightarrow^*_{\mathcal{P}^{\mathcal{B}_j}} t'$ in $\mathcal{P}^{\mathcal{B}_j}$ for some $t'$ such that the set of trees represented by $t'^\perp$ contains $t^\perp$.

To establish the simulation, in what follows, we first introduce a relation $t_1 \sqsubseteq t_2$, which means that the set of trees generated by $t_1$ is a subset of that of $t_2$, and prove some properties of $\sqsubseteq$. Especially, we prove the monotonicity of HMTTs in Lemma 4.9: $t_1 \sqsubseteq t_2$ implies $[t_1/x]t \sqsubseteq [t_2/x]t$ for any HMTT term $t$ with a free variable $x$. We also show some properties of the reduction $\mathcal{B}_i$. Second, we use the properties of $\sqsubseteq$ and $\mathcal{B}_i$ to prove that a one-step evaluation $t_1 \Longrightarrow t_2$ can be simulated by $\mathcal{B}_i(t_1) \sqsupseteq \mathcal{B}_i(t_2)$ in Lemma 4.11. Third, in Lemma 4.13, we show that an evaluation $S\ T_1 \cdots T_k \longrightarrow^*_{\mathcal{P}} \mathbf{assert}^{L_j}(t)$ can be simulated by some evaluation $S\ T_1 \cdots T_k \Longrightarrow^*_{\mathcal{P}} \mathbf{assert}^{L_j}(t'')$ for some $t''$ that satisfies $\mathit{Strip}(t'') = t$ under the assumption that $\models (\mathcal{P}^{\mathcal{B}_i}, \mathcal{M}_1, \dots, \mathcal{M}_k, \mathcal{B}(L_i))$ holds for each $i \in \{1, \dots, m\}$. Finally, we use the simulation of $\Longrightarrow$ by $\sqsupseteq$ in Lemma 4.11, the definition of $\sqsupseteq$, and the simulation of $\Longrightarrow$ by $\longrightarrow$ in Lemma 4.4 to show that $S\ T_1 \cdots T_k \longrightarrow^*_{\mathcal{P}^{\mathcal{B}_j}} t'$ is derivable for some $t'$ such that the set of trees represented by $t'^\perp$ contains $t^\perp$ in Lemma 4.12.

We write $t_1 \sqsubseteq_{\mathcal{P}} t_2$ if $GenBy_{\mathcal{P}}(t_1) \subseteq GenBy_{\mathcal{P}}(t_2)$, where $GenBy_{\mathcal{P}}(t)$ represents the set of trees generated by $t$. Formally, $GenBy_{\mathcal{P}}(t)$ is defined by $\{T \mid t \Longrightarrow_{\mathcal{P}}^{*} t', T \in Trees(t'^{\perp})\}$, where $Trees(T)$ denotes the set of trees represented by $T$, defined by

$$Trees(a\ T_1 \cdots T_n) = \begin{cases} Trees(T_1) \cup \cdots \cup Trees(T_n) & (\text{if } a = \mathtt{br}) \\ \varnothing & (\text{if } a = \mathtt{emp}) \\ \{a\ T_1' \cdots T_n' \mid T_1' \in Trees(T_1), \ldots, T_n' \in Trees(T_n)\} & (\text{otherwise}). \end{cases}$$

The following properties follow immediately.

**Lemma 4.8.**

1. If $t_1 \Longrightarrow t_2$, then $t_1 \sqsupseteq t_2$.
2. $\sqsubseteq$ is reflexive and transitive.

The following lemma states the monotonicity of HMTTs with respect to $\sqsubseteq$, and follows from the fact that HMTTs cannot destruct output trees. See Appendix A for the proof.

**Lemma 4.9.** Let $\mathcal{P}'$ be an HMTT, $t$ be an HMTT term with a free variable $x$ of the sort **o**, and $t_1$ and $t_2$ be closed terms of the sort **o** such that $t_1 \sqsubseteq t_2$. Then, $[t_1/x]t \sqsubseteq [t_2/x]t$ holds.

We extend $\mathcal{B}_i(t)$ for run-time terms as follows:

$$\mathcal{B}_i(\underline{a}) = a^{\mathcal{B}_i} \qquad \mathcal{B}_i(\mathbf{Error}) = \mathtt{emp}$$

$$\mathcal{B}_i(\mathbf{o2i}^L(t)) = \mathcal{B}_i(t) \qquad \mathcal{B}_i(\mathbf{assert}^{L_j}(t)) = \begin{cases} \mathtt{br}\ \mathcal{A}(t)\ \mathcal{B}_i(t) & (\text{if } i = j) \\ \mathcal{B}_i(t) & (\text{otherwise}). \end{cases}$$

We can naturally extend the definition of $\mathcal{B}_i(t)$ to evaluation contexts $E$ as follows:

$$\begin{aligned} \mathcal{B}_i([\,]) &= [\,]_2 \\ \mathcal{B}_i(a\ t_1 \cdots t_{j-1}\ E\ t_{j+1} \cdots t_{\Sigma(a)}) &= a^{\mathcal{B}_i}\ \mathcal{A}(t_1)\ \mathcal{B}_i(t_1) \cdots \mathcal{A}(t_{j-1})\ \mathcal{B}_i(t_{j-1}) \\ &\quad\ \mathcal{A}(E)[[\,]_1]\ \mathcal{B}_i(E) \\ &\quad\ \mathcal{A}(t_{j+1})\ \mathcal{B}_i(t_{j+1}) \cdots \mathcal{A}(t_{\Sigma(a)})\ \mathcal{B}_i(t_{\Sigma(a)}) \\ \mathcal{B}_i(\mathbf{case}\ E\ \mathbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow t_j\}_{j=1}^n) &= \mathtt{br}\ \mathcal{B}_i(E) \\ &\quad\ (\mathbf{case}\ \mathcal{A}(E)[[\,]_1]\ \mathbf{of} \\ &\quad\ \ \{a_j\ \widetilde{y}_j^{\mathcal{A}} \Rightarrow [\mathcal{B}_i(E), \ldots, \mathcal{B}_i(E)/\widetilde{y}_j^{\mathcal{B}_i}]\mathcal{B}_i(t_j)\}_{j=1}^n) \\ \mathcal{B}_i(\mathbf{o2i}^L(E)) &= \mathcal{B}_i(E) \\ \mathcal{B}_i(\mathbf{assert}^{L_j}(E)) &= \begin{cases} \mathtt{br}\ \mathcal{A}(E)\ \mathcal{B}_i(E) & (\text{if } i = j) \\ \mathcal{B}_i(E) & (\text{otherwise}). \end{cases} \end{aligned}$$

Note here that $\mathcal{B}_i(E)$ may have two holes $[\,]_1$ and $[\,]_2$. We can prove the following properties of $\mathcal{B}_i(t)$ and $\mathcal{B}_i(E)$ by a straightforward induction on the structure of $t$ or $E$.

**Lemma 4.10.**

1. $\mathcal{B}_i([t'/x]t) = [\mathcal{A}(t')/x^{\mathcal{A}}, \mathcal{B}_i(t')/x^{\mathcal{B}_i}]\mathcal{B}_i(t)$ and
2. $\mathcal{B}_i(E[t]) = \mathcal{B}_i(E)[\mathcal{A}(t)]_1[\mathcal{B}_i(t)]_2$.

We now show that a one-step evaluation in $\mathcal{P}$ can be simulated by some evaluation in $\mathcal{P}^{\mathcal{B}_i}$.

**Lemma 4.11.** If $t \Longrightarrow_{\mathcal{P}} t'$, then $\mathcal{B}_i(t) \sqsupseteq_{\mathcal{P}^{\mathcal{B}_i}} \mathcal{B}_i(t')$.

*Proof.* We prove the lemma by case analysis on the rule used to derive $t \Longrightarrow t'$:

— E-App: We have $t = E[F\ t_1 \ldots t_n]$ and $t' = E[[t_1/x_1, \ldots, t_n/x_n]t'']$, where $F\ x_1 \ldots x_n = t'' \in \mathcal{D}$. We have $F^{\mathcal{A}}\ x_1{}^{\mathcal{A}} \ldots x_n{}^{\mathcal{A}} = \mathcal{A}(t'') \in \mathcal{B}_i(D)$ and $F^{\mathcal{B}_i}\ x_1{}^{\mathcal{A}}\ x_1{}^{\mathcal{B}_i} \ldots x_n{}^{\mathcal{A}}\ x_n{}^{\mathcal{B}_i} = \mathcal{B}_i(t'') \in \mathcal{B}_i(D)$. Thus, we obtain:

$$
\begin{array}{rcll}
\mathcal{B}_i(t) & = & \mathcal{B}_i(E)[F^{\mathcal{A}}\ \mathcal{A}(t_1) \cdots \mathcal{A}(t_n)]_1 & \\
 & & \quad [F^{\mathcal{B}_i}\ \mathcal{A}(t_1)\ \mathcal{B}_i(t_1) \ldots \mathcal{A}(t_n)\ \mathcal{B}_i(t_n)]_2 & \text{(By Lemma 4.10(2))} \\
 & \Longrightarrow^* & \mathcal{B}_i(E)[F^{\mathcal{A}}\ \mathcal{A}(t_1) \cdots \mathcal{A}(t_n)]_1 & \\
 & & \quad [[\mathcal{A}(t_1)/x_1{}^{\mathcal{A}}, \ldots, \mathcal{A}(t_n)/x_n{}^{\mathcal{A}}] & \\
 & & \quad [\mathcal{B}_i(t_1)/x_1{}^{\mathcal{B}_i}, \ldots, \mathcal{B}_i(t_n)/x_n{}^{\mathcal{B}_i}]\mathcal{B}_i(t'')]_2 & \text{(By E-App)} \\
 & = & \mathcal{B}_i(E)[F^{\mathcal{A}}\ \mathcal{A}(t_1) \cdots \mathcal{A}(t_n)]_1 & \\
 & & \quad [\mathcal{B}_i([t_1/x_1, \ldots t_n/x_n]t'')]_2 & \text{(By Lemma 4.10(1))} \\
 & \Longrightarrow^* & \mathcal{B}_i(E)[[\mathcal{A}(t_1)/x_1{}^{\mathcal{A}}, \ldots, \mathcal{A}(t_n)/x_n{}^{\mathcal{A}}]\mathcal{A}(t'')]_1 & \\
 & & \quad [\mathcal{B}_i([t_1/x_1, \ldots t_n/x_n]t'')]_2 & \text{(By E-App)} \\
 & = & \mathcal{B}_i(E)[\mathcal{A}([t_1/x_1, \ldots, t_n/x_n]t'')]_1 & \\
 & & \quad [\mathcal{B}_i([t_1/x_1, \ldots t_n/x_n]t'')]_2 & \text{(By Lemma 4.6(2))} \\
 & = & \mathcal{B}_i(t'). & \text{(By Lemma 4.10(2))}
\end{array}
$$

The result follows by Lemma 4.8.

— E-Case: We have $t = E[\mathbf{case}\ \underline{a_j}\ \widetilde{t}\ \mathbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow t_j\}_{j=1}^n]$ and $t' = E[[\widetilde{t}/\widetilde{y}_j]t_j]$. We obtain:

$$
\begin{array}{rcll}
\mathcal{B}_i(t) & = & \mathcal{B}_i(E)[\mathbf{case}\ \underline{a_j}\ \mathcal{A}(\widetilde{t})\ \mathbf{of}\ \{a_j\ \widetilde{y}_j^{\mathcal{A}} \Rightarrow \mathcal{A}(t_j)\}_{j=1}^n]_1 & \\
 & & \quad [\mathcal{B}_i(\mathbf{case}\ \underline{a_j}\ \widetilde{t}\ \mathbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow t_j\}_{j=1}^n)]_2 & \text{(By Lemma 4.10(2))} \\
 & \Longrightarrow^* & \mathcal{B}_i(E)[[\mathcal{A}(\widetilde{t})/\widetilde{y}_j^{\mathcal{A}}]\mathcal{A}(t_j)]_1 & \\
 & & \quad [\mathcal{B}_i(\mathbf{case}\ \underline{a_j}\ \widetilde{t}\ \mathbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow t_j\}_{j=1}^n)]_2 & \text{(By E-Case)} \\
 & = & \mathcal{B}_i(E)[\mathcal{A}([\widetilde{t}/\widetilde{y}_j]t_j)]_1 & \\
 & & \quad [\mathcal{B}_i(\mathbf{case}\ \underline{a_j}\ \widetilde{t}\ \mathbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow t_j\}_{j=1}^n)]_2 & \text{(By Lemma 4.6(2))} \\
 & = & \mathcal{B}_i(E)[\mathcal{A}([\widetilde{t}/\widetilde{y}_j]t_j)]_1 & \\
 & & \quad [\mathtt{br}\ \mathcal{B}_i(\underline{a_j}\ \widetilde{t})\ (\mathbf{case}\ \underline{a_j}\ \mathcal{A}(\widetilde{t})\ \mathbf{of} & \\
 & & \qquad \{a_j\ \widetilde{y}_j^{\mathcal{A}} \Rightarrow \theta\mathcal{B}_i(t_j)\}_{j=1}^n)]_2 & \\
 & \Longrightarrow^* & \mathcal{B}_i(E)[\mathcal{A}([\widetilde{t}/\widetilde{y}_j]t_j)]_1 & \\
 & & \quad [\mathtt{br}\ \mathcal{B}_i(\underline{a_j}\ \widetilde{t})\ [\mathcal{A}(\widetilde{t})/\widetilde{y}_j^{\mathcal{A}}]\theta\mathcal{B}_i(t_j)]_2 & \text{(By E-Case)} \\
 & \sqsupseteq & \mathcal{B}_i(E)[\mathcal{A}([\widetilde{t}/\widetilde{y}_j]t_j)]_1 & \\
 & & \quad [[\mathcal{A}(\widetilde{t})/\widetilde{y}_j^{\mathcal{A}}, \mathcal{B}_i(\widetilde{t})/\widetilde{y}_j^{\mathcal{B}_i}]\mathcal{B}_i(t_j)]_2 & \text{(By Lemma 4.9)} \\
 & = & \mathcal{B}_i(E)[\mathcal{A}([\widetilde{t}/\widetilde{y}_j]t_j)]_1[\mathcal{B}_i([\widetilde{t}/\widetilde{y}_j]t_j)]_2 & \text{(By Lemma 4.10(1))} \\
 & = & \mathcal{B}_i(t'). & \text{(By Lemma 4.10(2))}
\end{array}
$$

Here, $\theta = [(\mathcal{B}_i(\underline{a_j}\ \widetilde{t}), \ldots, \mathcal{B}_i(\underline{a_j}\ \widetilde{t}))/\widetilde{y}_j^{\mathcal{B}_i}]$. The result follows by Lemma 4.8.

— E-CASE-FAIL: we have $t = E[\mathbf{case}\ \underline{a}\ \widetilde{t}\ \mathbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow t_j\}_{j=1}^n]$ and $t' = E[\mathtt{fail}]$, where $a \notin \{a_1, \ldots, a_n\}$. We obtain:

$$\begin{array}{rcll}
\mathcal{B}_i(t) & = & \mathcal{B}_i(E)[\mathbf{case}\ \underline{a}\ \mathcal{A}(\widetilde{t})\ \mathbf{of}\ \{a_j\ \widetilde{y_j^{\mathcal{A}}} \Rightarrow \mathcal{A}(t_j)\}_{j=1}^n]_1 & \\
 & & \quad [\mathcal{B}_i(\mathbf{case}\ \underline{a}\ \widetilde{t}\ \mathbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow t_j\}_{j=1}^n)]_2 & \text{(By Lemma 4.10(2))} \\
 & \Longrightarrow^* & \mathcal{B}_i(E)[\mathtt{fail}]_1[\mathcal{B}_i(\mathbf{case}\ \underline{a}\ \widetilde{t}\ \mathbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow t_j\}_{j=1}^n)]_2 & \text{(By E-CASE-FAIL)} \\
 & \sqsupseteq & \mathcal{B}_i(E)[\mathtt{fail}]_1[\mathtt{fail}^{\mathcal{B}_i}]_2 & \\
 & = & \mathcal{B}_i(t'). & \text{(By Lemma 4.10(2))}
\end{array}$$

The result follows by Lemma 4.8.

— E-GEN: we have $t = E[\mathbf{gen}^L]$ and $t' = E[\underline{a}\ \mathbf{gen}^{L_1} \ldots \mathbf{gen}^{L_n}]$, where $a\ L_1 \cdots L_n \subseteq L$ and $L_1, \ldots, L_n \neq \varnothing$. We obtain:

$$\begin{array}{rcll}
\mathcal{B}_i(t) & = & \mathcal{B}_i(E)[\mathbf{gen}^L]_1[\mathtt{emp}]_2 & \text{(By Lemma 4.10(2))} \\
 & \Longrightarrow^* & \mathcal{B}_i(E)[\underline{a}\ \mathbf{gen}^{L_1} \ldots \mathbf{gen}^{L_n}]_1[\mathtt{emp}]_2 & \text{(By E-GEN)} \\
 & \sqsupseteq & \mathcal{B}_i(E)[\underline{a}\ \mathbf{gen}^{L_1} \ldots \mathbf{gen}^{L_n}]_1 & \\
 & & \quad [a^{\mathcal{B}_i}\ \mathbf{gen}^{L_1}\ \mathtt{emp} \ldots \mathbf{gen}^{L_n}\ \mathtt{emp}]_2 & \text{(By Lemma 4.9)} \\
 & = & \mathcal{B}_i(t'). & \text{(By Lemma 4.10(2))}
\end{array}$$

The result follows by Lemma 4.8.

— E-COERCE-INPUTL: we have $t = E[\mathbf{coerce}^{L_j}(t'')]$ and $t' = E[\mathbf{o2i}^{L_j}(t'')]$. We obtain:

$$\begin{array}{rcll}
\mathcal{B}_i(t) & = & \mathcal{B}_i(E)[\mathbf{gen}^{L_j}]_1[\mathcal{B}_i(\mathbf{coerce}^{L_j}(t''))]_2 & \text{(By Lemma 4.10(2))} \\
 & \sqsupseteq & \mathcal{B}_i(E)[\mathbf{gen}^{L_j}]_1[\mathcal{B}_i(t'')]_2 & \text{(By Lemma 4.9)} \\
 & = & \mathcal{B}_i(t'). & \text{(By Lemma 4.10(2))}
\end{array}$$

— E-INPUTL: we have $t = E[\mathbf{o2i}^L(a\ t_1 \cdots t_n)]$ and $t' = E[\underline{a}\ \mathbf{o2i}^{L_1}(t_1) \cdots \mathbf{o2i}^{L_n}(t_n)]$, where $a\ L_1 \cdots L_n \subseteq L$ and $L_1, \ldots, L_n \neq \varnothing$. We obtain:

$$\begin{array}{rcll}
\mathcal{B}_i(t) & = & \mathcal{B}_i(E)[\mathbf{gen}^L]_1[a^{\mathcal{B}_i}\ \mathcal{A}(t_1)\ \mathcal{B}_i(t_1) \cdots \mathcal{A}(t_n)\ \mathcal{B}_i(t_n)]_2 & \text{(By Lemma 4.10(2))} \\
 & \Longrightarrow^* & \mathcal{B}_i(E)[\underline{a}\ \mathbf{gen}^{L_1} \cdots \mathbf{gen}^{L_n}]_1 & \\
 & & \quad [a^{\mathcal{B}_i}\ \mathcal{A}(t_1)\ \mathcal{B}_i(t_1) \cdots \mathcal{A}(t_n)\ \mathcal{B}_i(t_n)]_2 & \text{(By E-GEN)} \\
 & \sqsupseteq & \mathcal{B}_i(E)[\underline{a}\ \mathbf{gen}^{L_1} \cdots \mathbf{gen}^{L_n}]_1 & \\
 & & \quad [a^{\mathcal{B}_i}\ \mathbf{gen}^{L_1}\ \mathcal{B}_i(t_1) \cdots \mathbf{gen}^{L_n}\ \mathcal{B}_i(t_n)]_2 & \text{(By Lemma 4.9)} \\
 & = & \mathcal{B}_i(t'). & \text{(By Lemma 4.10(2))}
\end{array}$$

The result follows by Lemma 4.8.

— E-COERCE-ASSERT: we have $t = E[\mathbf{coerce}^{L_j}(t'')]$ and $t' = \mathbf{assert}^{L_j}(t'')$. If $i = j$, we get $\mathcal{B}_i(t) \sqsupseteq \mathtt{br}\ \mathcal{A}(t'')\ \mathcal{B}_i(t'') = \mathcal{B}_i(t')$. Otherwise, we obtain $\mathcal{B}_i(t) \sqsupseteq \mathcal{B}_i(t'') = \mathcal{B}_i(t')$.

— E-ASSERT-ERROR: we have $t = \mathbf{assert}^L(t'')$ and $t' = \mathbf{Error}$. We get $\mathcal{B}_i(t) \sqsupseteq \mathtt{emp} = \mathcal{B}_i(t')$.

□

Theorem 4.2 follows immediately from Lemmas 4.12 and 4.13 below.

**Lemma 4.12.** Suppose that $\models (\mathcal{P}^{\mathcal{B}_i}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{B}(L_i))$ holds for each $i \in \{1, \ldots, m\}$. For any $T_1 \in \mathcal{L}(\mathcal{M}_1), \ldots, T_k \in \mathcal{L}(\mathcal{M}_k)$, if $S\ T_1 \cdots T_k \Longrightarrow_{\mathcal{P}}^* \mathbf{assert}^{L_j}(t)$, then $\mathbf{assert}^{L_j}(\mathit{Strip}(t)) \not\longrightarrow_{\mathcal{P}} \mathbf{Error}$.

*Proof.* Suppose that $\models (\mathcal{P}^{\mathcal{B}_i}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{B}(L_i))$ holds for each $i \in \{1, \ldots, m\}$ and $S\ T_1 \cdots T_k \Longrightarrow^* \mathbf{assert}^{L_j}(t)$ for some $T_1 \in \mathcal{L}(\mathcal{M}_1), \ldots, T_k \in \mathcal{L}(\mathcal{M}_k)$. By Lemma 4.11,

we obtain $S^{\mathcal{B}_j}\ T_1\ \mathtt{emp} \cdots T_k\ \mathtt{emp} \sqsupseteq \mathcal{B}_j(\mathbf{assert}^{L_j}(t)) \sqsupseteq \mathcal{A}(t)$. By the definition of $\sqsubseteq$, $S^{\mathcal{B}_j}\ T_1\ \mathtt{emp} \cdots T_k\ \mathtt{emp} \Longrightarrow^* t'$ for some $t'$ such that $\mathcal{A}(t)^\perp \in \mathit{Trees}(t'^\perp)$. By Lemma 4.4, we obtain $S^{\mathcal{B}_j}\ T_1\ \mathtt{emp} \cdots T_k\ \mathtt{emp} \longrightarrow^* t'$. Since $\models (\mathcal{P}^{\mathcal{B}_j}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{B}(L_j))$ holds, $t'^\perp \in \mathcal{B}(L_j)^\perp$. Thus, by the definition of $\mathcal{B}(L_j)$, we get $\mathit{Trees}(t'^\perp) \subseteq L_j^\perp$. It follows that $\mathcal{A}(t)^\perp \in L_j^\perp$. By Lemma 4.6(1), we get $\mathit{Strip}(t)^\perp = t^\perp = \mathcal{A}(t)^\perp \in L_j^\perp$. Therefore, $\mathbf{assert}^{L_j}(\mathit{Strip}(t)) \not\longrightarrow \mathbf{Error}$. □

**Lemma 4.13.** Suppose that $\models (\mathcal{P}^{\mathcal{B}_i}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{B}(L_i))$ holds for each $i \in \{1, \ldots, m\}$. For any $T_1 \in \mathcal{L}(\mathcal{M}_1), \ldots, T_k \in \mathcal{L}(\mathcal{M}_k)$, if $S\ T_1 \cdots T_k \longrightarrow_{\mathcal{P}}^* t_1$, then $S\ T_1 \cdots T_k \Longrightarrow_{\mathcal{P}}^* t_2$ for some $t_2$ such that $t_1 = \mathit{Strip}(t_2)$.

*Proof.* The same as that of Lemma 4.5 except that $\mathbf{assert}^{L'}(\mathit{Strip}(C[a\ \widetilde{t'}])) \not\longrightarrow \mathbf{Error}$ is obtained from Lemma 4.12 and the assumption that $\models (\mathcal{P}^{\mathcal{B}_i}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{B}(L_i))$ holds for each $i \in \{1, \ldots, m\}$. □

Proof of Theorem 4.2. Suppose that $\models (\mathcal{P}^{\mathcal{B}_i}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{B}(L_i))$ holds for each $i \in \{1, \ldots, m\}$ and $S\ T_1 \cdots T_k \longrightarrow^* \mathbf{assert}^{L_j}(t)$ for some $T_1 \in \mathcal{L}(\mathcal{M}_1), \ldots, T_k \in \mathcal{L}(\mathcal{M}_k)$. By Lemma 4.13, we get $S\ T_1 \cdots T_k \Longrightarrow^* \mathbf{assert}^{L_j}(t')$ for some $t'$ such that $\mathit{Strip}(t') = t$. Then, by Lemma 4.12, we obtain $\mathbf{assert}^{L_j}(\mathit{Strip}(t')) \not\longrightarrow_{\mathcal{P}} \mathbf{Error}$. Therefore, $S\ T_1 \cdots T_k \not\longrightarrow_{\mathcal{P}} \mathbf{Error}$. □

### 4.4. *Incompleteness*

Our reduction from EHMTT to HMTT verification problems is incomplete: there is a case that an EHMTT satisfies a given specification, but the generated HMTTs do not satisfy the required properties. There are two main reasons for this.

— Coercion annotations may not be good enough. For example, if coercions are annotated with the empty language $\varnothing$, the derived HMTTs obviously do not satisfy the property. Actually, there may be no good way to annotate coercions. For example, consider the EHMTT $S\ x = \mathit{Zip}\ (\mathbf{coerce}^L(\mathit{Unzip}\ x))$, where *Unzip* takes an input $\mathtt{s}^n\ \mathtt{z}$ that encodes a natural number $n$ and returns an output tree $\mathtt{pair}\ (\mathtt{s}^n\ \mathtt{z})\ (\mathtt{s}^n\ \mathtt{z})$, and *Zip* takes an input tree of the form $\mathtt{pair}\ (\mathtt{s}^{n_1}\ \mathtt{z})\ (\mathtt{s}^{n_2}\ \mathtt{z})$ and outputs `fail` if and only if $n_1 \neq n_2$. To verify that $S\ x$ never outputs `fail` for any $x \in \{\mathtt{s}^n\ \mathtt{z} \mid n \geqslant 0\}$, we need the coercion annotation $L = \{\mathtt{pair}\ (\mathtt{s}^n\ \mathtt{z})\ (\mathtt{s}^n\ \mathtt{z}) \mid n \geqslant 0\}$, which cannot be expressed by a trivial automaton or a regular language. As another example, consider the following variant of a reverse function:

$$\begin{aligned}&\mathit{Reverse}\ x = \mathbf{case}\ x\ \mathbf{of}\ \mathtt{e} \Rightarrow \mathtt{e}\\ &\qquad\qquad\qquad\qquad \mid \mathtt{cons}\ z\ x' \Rightarrow \mathit{Append}\ (\mathbf{coerce}^L(\mathit{Reverse}\ x'))\ (\mathtt{cons}\ z\ \mathtt{e})\\ &\mathit{Append}\ x\ y = \cdots.\end{aligned}$$

Here, we have used a list-like representation for sequences consisting of $\mathtt{a}, \mathtt{b}$. In this case, the appropriate annotation depends on the value of $z$ ($L$ should be $\mathtt{b}^*\mathtt{a}^*\mathtt{e}$ if $z$ is $\mathtt{a}$ while $\mathtt{b}^*\mathtt{e}$ if $z$ is $\mathtt{b}$), which cannot be expressed in our language. One way to avoid this problem is to duplicate a part of the code so that appropriate annotations can be inserted.

— The output specification $\mathcal{B}(L_i)$ for $\mathcal{P}^{\mathcal{B}_i}$ is too restrictive. When the automaton for accepting $L_i$ is non-deterministic, $\mathcal{B}(L_i)$ does not accept all the tree representations of subsets of $L_i$. For example, let us consider a non-deterministic trivial automaton $\mathcal{M} = (\Sigma, \{q_0, q_1, q_2\}, \Delta, q_0)$, where

$$\begin{aligned}\Sigma &= \{\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 0, \mathtt{c} \mapsto 0\}\\ \Delta &= \{(q_0, \mathtt{a}, q_1), (q_0, \mathtt{a}, q_2), (q_1, \mathtt{b}, \epsilon), (q_2, \mathtt{c}, \epsilon)\}.\end{aligned}$$

Note that, $\mathcal{B}(\mathcal{M})$ does not accept `a (br b c)`. This problem can easily be remedied, however, by using a more elaborate construction of $\mathcal{B}(L_i)$, hence not a fundamental limitation. For the above example, we can construct $\mathcal{B}(\mathcal{M})$ as $(\Sigma', \{q_0, q_{1\vee 2}\}, \Delta', q_0)$, where

$$\begin{aligned}\Sigma' &= \Sigma \cup \{\mathtt{br} \mapsto 2, \mathtt{emp} \mapsto 0\}\\ \Delta' &= \{(q_0, \mathtt{a}, q_{1\vee 2}), (q_{1\vee 2}, \mathtt{b}, \epsilon), (q_{1\vee 2}, \mathtt{c}, \epsilon)\} \cup\\ &\quad \{(q_0, \mathtt{br}, q_0 \cdot q_0), (q_{1\vee 2}, \mathtt{br}, q_{1\vee 2} \cdot q_{1\vee 2})\} \cup\\ &\quad \{(q_0, \mathtt{emp}, \epsilon), (q_{1\vee 2}, \mathtt{emp}, \epsilon)\}.\end{aligned}$$

Our overall method is incomplete also because of the incompleteness of the HMTT verification method (Kobayashi *et al.* 2010). Note that the HMTT verification problem is undecidable in general.

## 5. Experiments

We have implemented the reduction method from an EHMTT verification problem to HMTT verification problems presented in Section 4. For solving the HMTT verification problems, we adopted an HMTT verification method in (Kobayashi *et al.* 2010) and Kobayashi's higher-order model checker TRecS (Kobayashi 2009b). A web interface of our prototype verifier and all the programs used in preliminary experiments reported in this section are available from `http://www.kb.is.s.u-tokyo.ac.jp/~uhiro/ehmtt/`.

Table 1 shows the results of the preliminary experiments. The column 'O' shows the order of each EHMTT, which is the largest order of the sorts of the functions. The order of a sort is defined by

$$order(\mathbf{i}) = order(\mathbf{o}) = 0 \qquad order(\kappa_1 \to \kappa_2) = max(order(\kappa_1) + 1, order(\kappa_2)).$$

The column 'C' shows the number of coercions in each EHMTT. The columns 'F' and 'S' are the number of functions and the size of each EHMTT, respectively. The size of an EHMTT is measured by the number of symbols occurring in the right-hand side of the function definitions. '$\mathrm{Sum_F}$' and '$\mathrm{Sum_S}$', respectively are the sum of the numbers of the functions and the sum of the sizes of all HMTTs derived from each EHMTT. '$\mathrm{Q_I}$' and '$\mathrm{Q_O}$', respectively, show the numbers of the states of trivial automata for the input and output specifications. The column '$\mathrm{T_{Red}}$' shows the elapsed time, in milliseconds, of reduction from an EHMTT verification problem to HMTT verification problems. The column 'Y/N' indicates whether each EHMTT was proved correct (Y) or rejected (N).

Table 1. *Experimental results.*

| Programs | O | C | F | S | $\text{Sum}_\text{F}$ | $\text{Sum}_\text{S}$ | $\text{Q}_\text{I}$ | $\text{Q}_\text{O}$ | $\text{T}_\text{Red}$ | Y/N | $\text{T}_\text{MC}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| `Reverse` | 1 | 2 | 3 | 32 | 23 | 222 | 4 | 2 | 1 | Y | 4 |
| `Isort` | 1 | 1 | 4 | 29 | 16 | 115 | 3 | 2 | 1 | Y | 3 |
| `Msort` | 2 | 4 | 8 | 131 | 88 | 1,731 | 3 | 2 | 2 | Y | 224 |
| `HomRep-Rev` | 4 | 1 | 12 | 90 | 43 | 362 | 6 | 2 | 1 | Y | 31 |
| `Split` | 2 | 1 | 6 | 126 | 33 | 572 | 23 | 9 | 3 | Y | 132 |
| `Bib2Html` | 2 | 1 | 13 | 493 | 126 | 2,303 | 59 | 50 | 52 | Y | 52 |
| `XMarkQ1` | 2 | 1 | 12 | 454 | 118 | 2,136 | 99 | 23 | 29 | Y | 168 |
| `XMarkQ2` | 1 | 2 | 9 | 461 | 207 | 3,797 | 99 | 4 | 77 | Y | 92 |
| `Gapid-Html` | 3 | 1 | 17 | 374 | 75 | 1,642 | 16 | 7 | 2 | Y | 112 |
| `JWIG-guess` | 2 | 1 | 6 | 465 | 98 | 2,331 | 64 | 50 | 588 | Y | 50 |
| `JWIG-cal` | 1 | 2 | 12 | 475 | 222 | 4,045 | 60 | 50 | 72 | Y | 73 |
| `MinCaml-K` | 2 | 8 | 19 | 605 | 563 | 16,117 | 5 | 3 | 5 | Y | 647 |
| `Split'` | 2 | 1 | 6 | 126 | 33 | 572 | 23 | 9 | 3 | N | 27 |
| `JWIG-guess'` | 2 | 1 | 6 | 465 | 98 | 2,331 | 64 | 50 | 586 | N | 49 |
| `JWIG-cal'` | 1 | 2 | 12 | 475 | 222 | 4,045 | 60 | 50 | 2 | N | 55 |

The column '$\text{T}_\text{MC}$' shows the total running time of the higher-order model checker TRecS to solve all the HMTT verification problems derived from each EHMTT.

The program `Reverse` is the same as the one presented in the article. `Isort` performs insertion sort on the lists encoded as linear trees over $\Sigma = \{\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 1, \mathtt{e} \mapsto 0\}$. `Msort` performs merge sort instead of insertion sort on the same linear trees. `HomRep-Rev` takes a word homomorphism $h$ over linear trees $(\mathtt{a}+\mathtt{b})^*\mathtt{e}$, a natural number $n$ encoded by $\mathtt{s}^n\,\mathtt{z}$ and a word $w \in (\mathtt{a}+\mathtt{b})^*\mathtt{e}$, 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 program `Split` presented in Figure 3 is taken from sample programs of CDuce (Benzaken *et al.* 2003), a higher-order XML-oriented functional language. `Split` takes a family tree and transforms it into another representation by distinguishing children into sons and daughters. `Bib2Html` also simulates a CDuce program that transforms a list of bibliography into an XHTML. `XMarkQ1` and `XMarkQ2` taken from Q1 and Q2 of XMark benchmark suite (Schmidt *et al.* 2002) simulate simple XQuery queries. The program `Gapid-Html` is a composition of Tozawa's high-level tree transducers (Tozawa 2006). It takes an XML document of the following DTD:

```
type Doc      = doc[Preface, (Div|P|Note)*]
type Preface  = preface[Header, P*]
type Header   = header[A*]
type P        = p[A*]
type Div      = div[(Div|P|Note|A)*]
type Note     = note[(P|A)*]
type A        = a[A*]
```

and another tree as inputs. The program replaces each node of the document that has no children with a 'hole'. The program then inserts the given tree into the holes. The program finally transforms the result to an XHTML. The programs `JWIG-guess` and `JWIG-cal` are taken from sample programs of JWIG (`http://www.brics.dk/JWIG/`), a programming language for interactive web services. A main feature of JWIG is document templates. For example, the following document template represents an HTML document with a hole named `x`:

```
<html>
  <head><title> ... </title></head>
  <body><[x]></body>
</html>
```

We can instantiate the template by substituting another document or template for `x`. In EHMTTs, the template can be encoded as the following function $T$ with an argument $x$:

$$T\ x = \texttt{html}\ (\texttt{head}\ (\texttt{title}\ (\texttt{text}\ \texttt{leaf}\ \texttt{leaf})\ \texttt{leaf})\ (\texttt{body}\ x\ \texttt{leaf}))\ \texttt{leaf}.$$

Note here that the original unranked HTML tree is encoded as a binary tree. The program `JWIG-guess` is a number guessing game. The program `JWIG-cal` is a web-based calendar service. `MinCaml-K` simulates the K-normalization routine of the MinCaml compiler (`http://min-caml.sourceforge.net/index-e.html`). Finally, `Split'`, `JWIG-guess'` and `JWIG-cal'` are respectively the same as `Split`, `JWIG-guess` and `JWIG-cal` except that they have wrong coercion annotations leading to an **Error**. We use them to confirm that our method can reject programs with wrong annotations. All the programs (except for `Reverse`, `Isort`, `Msort` and `HomRep-Rev`) are manually translated from original source codes to EHMTTs.

All the valid programs have been proved correct by our verification method despite its incompleteness, while wrong programs are correctly rejected. The number of coercions shown in the column 'C' (thus, the number of annotations required by our method) is much smaller than the number of functions shown in the column 'F' in all cases. Though these numbers depend on the particular encoding, this result witnesses that our verification method usually requires fewer annotations than existing verification methods (Benzaken *et al.* 2003; Davies 2005; Hosoya and Pierce 2003), which require type annotations for every function definition. Furthermore, comparison with the existing methods on this point is given in Section 6.

All the programs were proved correct within 1 second. From this, we can expect that our method can verify non-trivial programs reasonably fast despite the high time complexity ($n$-EXPTIME complete, where $n$ is the order) of higher-order model checking.

## 6. Related work

As shown in Section 1, our verification method is based on recent advances on higher-order model checking (Aehlig *et al.* 2005; Knapik *et al.* 2002; Kobayashi 2009a,b; Kobayashi and Ong 2009; Ong 2006). Ong (2006) has proven the decidability of the model checking problem for recursion schemes, and Kobayashi (2009a,b) has developed and implemented a type-based model checking algorithm.

```
Split x = case x of person g n c =>
  case c of children cs => case g of gender gend =>
    Let gend n (coerce qPair (MakePair cs nil nil))
Let gend n x = case gend of
  m => Make man (copy n) x
| f => Make woman (copy n) x
MakePair ps ms fs = case ps of nil => pair ms fs
| cons p sib => case p of person g n c => case g of gender gend =>
  case gend of
   m => MakePair sib (cons (person (gender m) (copy n) (copy c)) ms) fs
  |f => MakePair sib ms (cons (person (gender f) (copy n) (copy c)) fs)
Make tag name sdpair = case sdpair of pair s d =>
  tag name (sons (RevMap Split s nil))
           (daughters (RevMap Split d nil))
RevMap f l ac =
  case l of nil => ac | cons x xs => RevMap f xs (cons (f x) ac)
```

Fig. 3. Program `Split` taken from sample programs of CDuce (Benzaken *et al.* 2003), where `copy` is a primitive function that converts input trees to output trees, and `qPair` in the coerce annotation represents an externally defined regular set of pairs of males and females.

Ong and Ramsay (2011) proposed a verification method for tree-processing programs based on higher-order model checking. Their method can handle as expressive programs as EHMTTs but does not require coercion annotations unlike in our method. Their method uses the so-called binding analysis to infer such invariants on tree data structures. Their method also tackles the incompleteness of the HMTT verification method (Kobayashi *et al.* 2010) by using a counterexample guided abstraction refinement technique. In contrast, our method is more precise for functions that take an input tree. In fact, our method can conclude that the function *Append* in Example 3.1 has the following context-sensitive type:

$$\mathit{Append} \ : \ (\mathtt{b}^*\mathtt{a}^*\mathtt{e} \to \mathtt{a}^*\mathtt{e} \to \mathtt{b}^*\mathtt{a}^*\mathtt{e}) \wedge (\mathtt{b}^*\mathtt{e} \to \mathtt{b}^*\mathtt{e} \to \mathtt{b}^*\mathtt{e}).$$

However, their method cannot infer the above type since the method analyse the function context-insensitively by assuming that the first argument is in $(\mathtt{b}^*\mathtt{a}^*\mathtt{e} \mid \mathtt{b}^*\mathtt{e}^*)$.

As we have shown in Section 5, our EHMTT verification method can be applied to verification of functional programs that manipulate various data structures such as strings, lists, trees, XML, and user-defined recursive data structures by encoding them as trees and adding coercion annotations to the programs. We compare our method with existing verification methods below.

Refinement types (Davies 2005; Freeman and Pfenning 1991) can be used for verification of functional programs that manipulate user-defined recursive data structures. The original refinement type system (Freeman and Pfenning 1991) uses a naïve least fixed-point algorithm to infer the most precise refinement types of functions, and does not seem to scale for higher-order functions. Another refinement type system proposed by Davies (2005) requires users to write type annotations for each function. For example, for $\mathcal{P}_{rev}$

in Example 3.1, *Reverse* and *Append* need to be annotated with the following intersection types:

$$\begin{aligned} \mathit{Reverse} &: (\mathtt{a}^*\mathtt{b}^*\mathtt{e} \to \mathtt{b}^*\mathtt{a}^*\mathtt{e}) \wedge (\mathtt{b}^*\mathtt{e} \to \mathtt{b}^*\mathtt{e}). \\ \mathit{Append} &: (\mathtt{b}^*\mathtt{a}^*\mathtt{e} \to \mathtt{a}^*\mathtt{e} \to \mathtt{b}^*\mathtt{a}^*\mathtt{e}) \wedge (\mathtt{b}^*\mathtt{e} \to \mathtt{b}^*\mathtt{e} \to \mathtt{b}^*\mathtt{e}). \end{aligned}$$

In contrast, our method requires only the annotations $\mathbf{coerce}^{\mathtt{b}^*\mathtt{a}^*\mathtt{e}}(\mathit{Reverse}\ x')$ and $\mathbf{coerce}^{\mathtt{b}^*\mathtt{e}}(\mathit{Reverse}\ x')$ in the definition of *Reverse*. As in this case, we expect that coercion annotations required in our approach tends to be simpler than refinement type declarations. Because of the limitation of our approach discussed in Section 4.4, however, it may be useful to combine both approaches.

Several research groups have proposed typed XML processing languages (Benzaken *et al.* 2003; Hosoya and Pierce 2003). Their type systems can be used for verification of XML processing programs. As in the Davies's refinement type system, these type systems require type annotations for each function for type checking. Thus, our method can be used as an alternative for a verification purpose. Meanwhile their type systems support advanced programming features such as parametric polymorphism (Hosoya *et al.* 2009) and regular expression pattern matching (Hosoya *et al.* 2000). Extensions of our method with these features are left for future work. While the type checking of the XML processing languages are incomplete, extensive work has been done on complete type checking of various tree transducers (Maneth *et al.* 2005; Milo *et al.* 2003; Tozawa 2006). They are not Turing-complete, however, and thus less expressive than our EHMTTs. As shown in Kobayashi *et al.* (2010), ordinary macro and high-level tree transducers (Engelfriet and Vogler 1985, 1988) are subsumed by linear HMTTs, for which our EHMTT verification method is sound and complete.

String analysis (Christensen *et al.* 2003; Minamide 2005) can verify programs that manipulate strings by approximating a string-processing program as a regular or a context-free grammar. In contrast, our method is more precise since we can naturally model programs as EHMTTs, which are strictly more expressive than context-free grammars.

Our approach of reducing EHMTT verification to simpler verification problems based on coercion annotations is a reminiscent of program verification techniques for imperative languages based on verification condition generation from loop invariants: coercion annotations are invariants, and generated HMTT verification problems can be considered verification conditions. The main differences are that our target is a higher-order functional language and that not all recursions (or loops) need to be annotated with invariants.

To enable fully automatic verification of tree-processing programs using our method, we can apply existing techniques to the inference of coercion annotations. If the derived HMTTs are ordinary macro or high-level tree transducers (Engelfriet and Vogler 1985, 1988), annotations can indeed be inferred by the inverse inference technique. For general EHMTTs, we may use forward invariant inference techniques (Jones and Andersen 2007; Kochems and Ong 2011; Minamide 2005; Ong and Ramsay 2011). We have also proposed a context-sensitive invariant inference method based on a machine learning technique (Tabuchi *et al.* 2011).

## 7. Conclusion

We have proposed a verification method for tree-processing programs based on reduction to higher-order model checking, and shown its effectiveness through experiments. Addressing the limitations discussed in Section 4.4 is left for future work.

### Acknowledgements

We would like to thank anonymous referees for useful comments. This work was partially supported by Kakenhi 20240001 and 23220001.

## Appendix A. Proof of Lemma 4.9

We first prepare Lemmas A.1 and A.2 to prove Lemma 4.9.

**Lemma A.1.** Let $\mathcal{P}'$ be an HMTT, $t$ be an HMTT term, $\widetilde{x}$ be free variables of the sort $\mathbf{o}$ in $t$, and $\widetilde{t_1}$ and $\widetilde{t_2}$ be closed terms of the sort $\mathbf{o}$ such that $\widetilde{t_1} \sqsubseteq \widetilde{t_2}$. Then, $\mathit{Trees}((\widetilde{[t_1/x]}t)^{\perp}) \subseteq \mathit{GenBy}(\widetilde{[t_2/x]}t)$ holds.

*Proof.* This follows by induction on the structure of $t$.

— Case $t = a\ t'_1 \cdots t'_n$: By the induction hypothesis, it follows that $\mathit{Trees}((\widetilde{[t_1/x]}t'_i)^{\perp}) \subseteq \mathit{GenBy}(\widetilde{[t_2/x]}t'_i)$ for each $i \in \{1,\ldots,n\}$. Thus, $\mathit{Trees}((\widetilde{[t_1/x]}t)^{\perp}) \subseteq \mathit{GenBy}(\widetilde{[t_2/x]}t)$.

— Case $t = y$: If $y \in \{\widetilde{x}\}$, $\mathit{Trees}((\widetilde{[t_1/x]}y)^{\perp}) \subseteq \mathit{GenBy}(\widetilde{[t_2/x]}y)$ follows from $\widetilde{t_1} \sqsubseteq \widetilde{t_2}$. Otherwise, $\mathit{Trees}((\widetilde{[t_1/x]}y)^{\perp}) = \varnothing \subseteq \mathit{GenBy}(\widetilde{[t_2/x]}y)$ follows.

— Cases $t = F\ t'_1 \cdots t'_n$, $t = \mathbf{case}\ t'\ \mathbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow t'_j\}_{j=1}^{n}$, $t = \mathbf{gen}^L$: It follows that $\mathit{Trees}((\widetilde{[t_1/x]}t)^{\perp}) = \varnothing \subseteq \mathit{GenBy}(\widetilde{[t_2/x]}t)$.

□

**Lemma A.2.** Let $\mathcal{P}' = (\mathcal{D}, S)$ be an HMTT, $t$ be an HMTT term, $x$ be a free variable of the sort $\mathbf{o}$ in $t$, and $t_0$ be a closed term of the sort $\mathbf{o}$. Let $t_1$ be a term obtained from $t$ by replacing each occurrence of $x$ with some term in $X = \{t \mid t_0 \Longrightarrow^* t\}$. If $t_1 \Longrightarrow t'_1$, then there exists $t'$ such that $t \Longrightarrow^* t'$ and $t'_1$ can be obtained from $t'$ by replacing each occurrence of $x$ with some term in $X$.

*Proof.* We prove the lemma by induction on the structure of $t$.

— Case $t = a\ u_1 \cdots u_n$: Let $t_1 = a\ u'_1 \cdots u'_n$. Suppose that $u'_i \Longrightarrow u'$ and $t'_1 = a\ u'_1 \cdots u'_{i-1}\ u'\ u'_{i+1} \cdots u'_n$. By the induction hypothesis, we obtain $u_i \Longrightarrow^* u$ and $u'$ is obtained from $u$ by replacing each occurrence of $x$ with some term in $X$. Let $t' = a\ u_1 \cdots u_{i-1}\ u\ u_{i+1} \cdots u_n$. Then, we get $t = a\ u_1 \cdots u_n \Longrightarrow^* a\ u_1 \cdots u_{i-1}\ u\ u_{i+1} \cdots u_n = t'$. $t'_1$ can be obtained from $t'$ by replacing each occurrence of $x$ with some term in $X$.

— Case $t = F\ u_1 \cdots u_n$: Let $t_1 = F\ u'_1 \cdots u'_n$. We can apply only A-App. Let $F\ x_1 \ldots x_n = t'' \in \mathcal{D}$. Then, we get $t'_1 = [u'_1/x_1,\ldots,u'_n/x_n]t''$. Let $t' = [u_1/x_1,\ldots,u_n/x_n]t''$. Then, we get $t = F\ u_1 \cdots u_n \Longrightarrow [u_1/x_1,\ldots,u_n/x_n]t'' = t'$. $t'_1$ can be obtained from $t'$ by replacing each occurrence of $x$ with some term in $X$.

— Case $t = x$: Let $t' = x$. Then, $t \Longrightarrow^* t'$ follows immediately. Since $t_1 \in X$, we get $t'_1 \in X$. Thus, $t'_1$ can be obtained from $t'$ by replacing each occurrence of $x$ with some term in $X$.

— Case $t = \mathbf{case}\ u\ \mathbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow u_j\}_{j=1}^n$: Let $t_1 = \mathbf{case}\ u'\ \mathbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow u'_j\}_{j=1}^n$
  - If $u' \Longrightarrow u'_0$ and $t'_1 = \mathbf{case}\ u'_0\ \mathbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow u'_j\}_{j=1}^n$, by the induction hypothesis, there exists some $u_0$ such that $u \Longrightarrow^* u_0$ and $u'_0$ is obtained from $u_0$ by replacing each occurrence of $x$ with some term in $X$. Let $t' = \mathbf{case}\ u_0\ \mathbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow u_j\}_{j=1}^n$. Then, we get $t = \mathbf{case}\ u\ \mathbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow u_j\}_{j=1}^n \Longrightarrow^* \mathbf{case}\ u_0\ \mathbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow u_j\}_{j=1}^n = t'$. $t'_1$ can be obtained from $t'$ by replacing each occurrence of $x$ with some term in $X$.
  - Otherwise, $u = \underline{a_j}\ \widetilde{u}$, $u' = \underline{a_j}\ \widetilde{u}'$, and $t'_1 = [\widetilde{u}'/\widetilde{y}_j]u'_j$. Let $t' = [\widetilde{u}/\widetilde{y}_j]u_j$. Then, we get $t = \mathbf{case}\ \underline{a_j}\ \widetilde{u}\ \mathbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow \overline{u_j}\}_{j=1}^n \Longrightarrow [\widetilde{u}/\widetilde{y}_j]u_j = t'$. $t'_1$ can be obtained from $t'$ by replacing each occurrence of $x$ with some term in $X$.

— Case $t = \mathbf{gen}^L$: it must be the case that $t_1 = t$. We can apply only E-GEN. Thus, we get $t'_1 = \underline{a}\ \mathbf{gen}^{L_1} \cdots \mathbf{gen}^{L_n}$, where $a\ \mathbf{gen}^{L_1} \cdots \mathbf{gen}^{L_n} \subseteq L$ and $L_1, \ldots, L_n \neq \emptyset$. Let $t' = \underline{a}\ \mathbf{gen}^{L_1} \cdots \mathbf{gen}^{L_n}$. Then, we get $t = \mathbf{gen}^L \Longrightarrow \underline{a}\ \mathbf{gen}^{L_1} \cdots \mathbf{gen}^{L_n} = t'$. $x$ does not occur in $t'_1$ and $t'_1 = t'$.

□

Proof of Lemma 4.9. Suppose that $[t_1/x]t \Longrightarrow^*_{\mathcal{P}'} t'_1$. By Lemma A.2, $t \Longrightarrow^*_{\mathcal{P}'} t'$ for some $t'$ such that $t'_1$ can be obtained from $t'$ by replacing each occurrence of $x$ with some term in $\{t \mid t_1 \Longrightarrow^* t\}$. Thus, we get $[t_2/x]t \Longrightarrow^*_{\mathcal{P}'} [t_2/x]t'$. Let $t''$ be the term obtained from $t'$ by renaming each occurrence of $x$ to unique names. Then, we have substitutions $\theta_1$ and $\theta_2$ such that $\mathrm{dom}(\theta_1)$ and $\mathrm{dom}(\theta_2)$ are the set of the free variables in $t''$, $\theta_1 t'' = t'_1$, and $\theta_2 t'' = [t_2/x]t'$. From $t_1 \sqsubseteq t_2$ and the fact that $\theta_1 x \in \{t \mid t_1 \Longrightarrow^* t\}$ for all $x \in \mathrm{dom}(\theta_1)$, we obtain $\theta_1 x \sqsubseteq \theta_2 x$ for all $x \in \mathrm{dom}(\theta_1)$. By Lemma A.1, $\mathit{Trees}((t'_1)^\perp) = \mathit{Trees}((\theta_1 t'')^\perp) \subseteq \mathit{GenBy}(\theta_2 t'') = \mathit{GenBy}([t_2/x]t')$ holds. Thus, by the definition of $\sqsubseteq$, we obtain $[t_1/x]t \sqsubseteq [t_2/x]t$. □

## References


Aehlig, K., de Miranda, J. G. and C.-H. Luke Ong. (2005) The monadic second order theory of trees given by arbitrary level-two recursion schemes is decidable. In:TLCA'05, *Springer Lecture Notes in Computer Science*, **3461** 39–54.

Benzaken, V., Castagna, G. and Frisch, A. (2003) CDuce: An XML-centric general-purpose language. In: *ICFP '03*, ACM 51–63.

Christensen, A. S., Møller, A. and Schwartzbach, M. I. (2003) Precise analysis of string expressions. In: SAS'03, *Springer Lecture Notes in Computer Science* **2694** 1–18.

Davies, R. (2005) *Practical Refinement-Type Checking*, Ph.D. thesis, Carnegie Mellon University, Chair-Pfenning, Frank.

Engelfriet, J. and Vogler, H. (1985) Macro tree transducers. *Journal of Computer and System Sciences* **31** (1) 71–146.

Engelfriet, J. and Vogler, H. (1988) High level tree transducers and iterated pushdown tree transducers. *Acta Informatica* 26 (1/2) 131–192.

Freeman, T. and Pfenning, F. (1991) Refinement types for ML. In: *PLDI '91*, ACM 268–277.

Hosoya, H. and Pierce, B. C. (2003) XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology* **3** (2) 117–148.

Hosoya, H., Vouillon, J. and Pierce, B. C. (2000) Regular expression types for XML. In: *ICFP '00*, ACM 11–22.

Hosoya, H., Frisch, A. and Castagna, G. (2009) Parametric polymorphism for XML. *ACM Transactions on Programming Languages and Systems* **32** (1) 1–56.

Jones, N. D. and Andersen, N. (2007) Flow analysis of lazy higher-order functional programs. *Theoretical Computer Science* **375** (1–3) 120–136.

Knapik, T., Niwinski, D. and Urzyczyn, P. (2002) Higher-order pushdown trees are easy. In: FoSSaCS'02, *Springer Lecture Notes in Computer Science* **2303** 205–222.

Kobayashi, N. and Ong, C.-H. L. (2009) A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In: *LICS '09*, IEEE 179–188.

Kobayashi, N., Tabuchi, N. and Unno, H. (2010) Higher-order multi-parameter tree transducers and recursion schemes for program verification. In: *POPL '10*, ACM 495–508.

Kobayashi, N. (2009a) Types and higher-order recursion schemes for verification of higher-order programs. In: *POPL '09*, ACM 416–428.

Kobayashi, N. (2009b) Model-checking higher-order functions. In: *PPDP '09*, ACM 25–36.

Kochems, J. and Ong, C.-H. L. (2011) Improved functional flow and reachability analyses using indexed linear tree grammars. In: *RTA2011*.

Maneth, S., Berlea, A., Perst, T. and Seidl, H. (2005) XML type checking with macro tree transducers. In: *PODS '05*, ACM 283–294.

Milo, T., Suciu, D. and Vianu, V. (2003) Typechecking for XML transformers. *Journal of Computer and System Sciences* **66** (1) 66–97.

Minamide, Y. (2005) Static approximation of dynamically generated web pages. In: *WWW '05*, ACM 432–441.

Ong C.-H. L. and Ramsay S. J. (2011) Verifying higher-order functional programs with pattern-matching algebraic data types. In: *POPL '11*, ACM 587–598.

Ong, C.-H. L. (2006) On model-checking trees generated by higher-order recursion schemes. In: *LICS '06*, IEEE 81–90.

Schmidt, A., Waas, F., Kersten, M., Carey M. J., Manolescu, I. and Busse, R. (2002) XMark: A benchmark for XML data management. In: *VLDB '02*, VLDB Endowment 974–985.

Tabuchi, N., Kobayashi, N. and Unno, H. (2011) Inference of tree data structure invariant based on language identification from samples. (Available at `http://www.kb.is.s.u-tokyo.ac.jp/~uhiro/`.)

Tozawa, A. (2006) XML type checking using high-level tree transducer. In: FLOPS'06, *Springer Lecture Notes in Computer Science* **3945** 81–96.

Unno, H., Tabuchi, N. and Kobayashi, N. (2010) Verification of tree-processing programs via higher-order model checking. In: APLAS'10, *Springer Lecture Notes in Computer Science* **6461** 312–327.