Verification of Tree-Processing Programs via Higher-Order Model Checking

Hiroshi Unno, Naoshi Tabuchi, and Naoki Kobayashi

Tohoku University

Abstract. 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 [20], or higher-order model checking for short, has been extensively studied recently. Ong [20] has shown the decidability of higher-order model checking. Kobayashi [12, 13] 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 λ-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 [13] 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 [15], we have introduced higher-order multi-parameter tree transducers (HMTTs) as an extension of recursion schemes with tree destructors. 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.

In this paper, we propose a verification method for an extension of HMTTs called EHMTTs. In essence, EHMTTs are higher-order, simply-typed functional

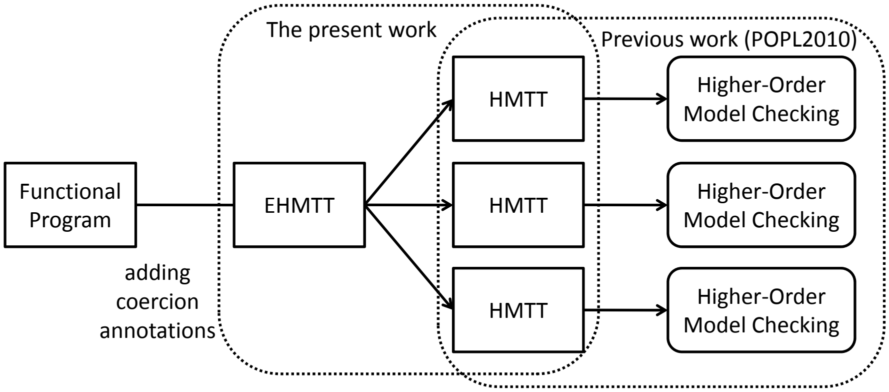
The present work
Previous work (POPL2010)
Functional Program
adding coercion annotations
EHMTT
HMTT
HMTT
HMTT
Higher-Order Model Checking
Higher-Order Model Checking
Higher-Order Model Checking

Fig. 1. Overall Structure of EHMTT Verification Method

programs with recursion and tree primitives. Unlike our previous HMTTs [15], 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 an HMTT verification method presented in our previous work [15]. 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 [12]. In this paper, we have formalized the reduction from an EHMTT verification problem to HMTT verification problems, and proved 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 paper is organized as follows. Section 2 presents some preliminary definitions and notations. In Section 3, we introduce 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 paper 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 ϵ for the empty sequence, and $v_1 \cdots v_n$ 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 $v_1 \cdots v_n$ is often abbreviated to $\widetilde{v}$.

A *ranked alphabet* Σ 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. We write A_Σ to denote the largest arity of the symbols in $\mathrm{dom}(\Sigma)$. A *Σ-labeled ranked tree* T is a map from a subset of $\{1, \cdots, A_\Sigma\}^*$ to $\mathrm{dom}(\Sigma)$ such that:

- $\mathrm{dom}(T)$ is prefix-closed, i.e. if $\pi \cdot i \in \mathrm{dom}(T)$, then $\pi \in \mathrm{dom}(T)$; and
- if $T(\pi) = a$, then $\{i \mid \pi \cdot i \in \mathrm{dom}(T)\} = \{1, \ldots, \Sigma(a)\}$.

3 Extended HMTTs

In this section, we introduce extended HMTTs (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 differences 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. Special annotations ($\mathbf{coerce}^L(\cdot)$ 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 Σ. We call elements of $\mathrm{dom}(\Sigma)$ *terminal symbols*, and use the meta-variable a for them.

Definition 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 a function name. 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 $\mathcal{K}$ 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 κ_1 and return a tree or tree function of sort κ_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.

Syntax of Sorts:

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

Sort Assignment Rules:

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

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

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

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

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

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

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

$$\frac{\mathcal{K} = \{F_1 : \widetilde{\kappa_1} \to \mathbf{o}, \ldots, F_n : \widetilde{\kappa_1} \to \mathbf{o}\} \qquad \mathcal{K}, \widetilde{x_i} : \widetilde{\kappa_i} \vdash t_i : \mathbf{o} \text{ (for each } i)}{\vdash \{F_1\ \widetilde{x_1} = t_1, \ldots, F_n\ \widetilde{x_n} = t_n\} : \mathcal{K}} \quad \text{(T-Def)}$$

Operational Semantics

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

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

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

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

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

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

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

$$E[\mathbf{o2i}(a_i\ \widetilde{t})] \longrightarrow_{\mathcal{P}} E[\underline{a_i}\ \mathbf{o2i}(\widetilde{t})] \qquad \text{(E-Input)}$$

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

Fig. 2. Sort Assignment Rules and Call-by-Name Operational Semantics

The term $\mathbf{case}\ t\ \mathbf{of}\ \{a_i\ \widetilde{y}_i \Rightarrow t_i\}_{i=1}^n$ 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 `fail`.[1] The term $\mathbf{coerce}^L(t)$ asserts that the tree generated by t belongs to a set L of trees, and converts the tree to an input tree. $\mathbf{gen}^L$ generates an element of L non-deterministically.

Example 1. Consider the EHMTT $\mathcal{P}_{rev} = (D, \mathit{Reverse})$, where D consists of:

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

P_{rev} takes a tree of the form $\mathtt{a}^m(\mathtt{b}^n(\mathtt{e}))$ as input, and outputs a 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 $\{\mathtt{b}^m(\mathtt{a}^n(\mathtt{e})) \mid m, n \geq 0\}$ and $\{\mathtt{b}^m(\mathtt{e}) \mid m \geq 0\}$ 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, by the restriction of EHMTT, 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 belong to L. In the rule E-Assert-Error, $t^\perp$ is a finite $(\Sigma \cup \{\perp \mapsto 0\})$-labeled 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 nodes of T' with $\perp$.

Remark 1. Note that EHMTT is call-by-name. This is because our verification method is based on the model checking of higher-oder 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). Secondly, we would like to model a

[1] 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 `fail`, which is an instance of EHMTT verification problems considered below.

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.

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* (which are Büchi tree automata with a trivial acceptance condition) as specifications (as well as for annotations L in $\mathbf{coerce}^L(\cdot)$ and $\mathbf{gen}^L$).

Definition 2 (trivial automaton). *A* trivial automaton *$\mathcal{M}$ is a quadruple (Σ, Q, Δ, q_0), where:*

- *Σ is a ranked alphabet.*
- *Q is a finite set of states.*
- *Δ 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 Σ-labeled ranked tree T is accepted *by $\mathcal{M}$ if there is a Q-labeled 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 Σ-labeled 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 2. Recall Example 1. A trivial automaton for accepting $\mathtt{b}^*\mathtt{a}^*\mathtt{e}$ is defined by $(\Sigma, \{q_0, q_1\}, \Delta, q_0)$, where:

$$\Sigma = \{\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 1, \mathtt{e} \mapsto 0\}$$
$$\Delta = \{(q_0, \mathtt{b}, q_0), (q_0, \mathtt{a}, q_1), (q_0, \mathtt{e}, \epsilon), (q_1, \mathtt{a}, q_1), (q_1, \mathtt{e}, \epsilon)\} \qquad \Box$$

We now formalize our verification problem:

Definition 3. *Given an EHMTT $\mathcal{P} = (D, S)$ and trivial automata $\mathcal{M}_1, \dots, \mathcal{M}_k$, $\mathcal{M} = (\Sigma, Q, \Delta, q_0)$, we write $\models (\mathcal{P}, \mathcal{M}_1, \dots, \mathcal{M}_k, \mathcal{M})$ if for all $T_1 \in \mathcal{L}(\mathcal{M}_1), \dots, T_k \in \mathcal{L}(\mathcal{M}_k)$,*

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

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 $(\mathcal{P}, \mathcal{M}_1, \dots, \mathcal{M}_k, \mathcal{M})$ is the problem to check that $\models (\mathcal{P}, \mathcal{M}_1, \dots, \mathcal{M}_k, \mathcal{M})$.

The first condition of an EHMTT verification problem says that given input trees that conform to the input specification, the EHMTT generates a valid tree.[2] The second condition means that the EHMTT in fact never causes a coercion error.

In [15], we have presented a (sound but incomplete) 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.

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 [15]). The idea is to reduce each of the two conditions in Definition 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, the verification problem $(\mathcal{P}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})$ 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.

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.

Theorem 1. *Let $\mathcal{P} = (D, S)$ be an EHMTT such 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 $S\ T_1 \cdots T_k \not\longrightarrow^*_{\mathcal{P}}$* **Error** *and $S\ T_1 \cdots T_k \longrightarrow^*_{\mathcal{P}} t$, then $t^{\perp} \in \mathcal{L}(\mathcal{M}^{\perp})$.*

A proof is given in Appendix A.

[2] 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 $\mathcal{L}(\mathcal{M})$.

Example 3. Recall $\mathcal{P}_{rev}$ in Example 1. $\mathcal{P}_{rev}^{\mathcal{A}}$ is $(D, \mathit{Reverse}^{\mathcal{A}})$ where D is given by:

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

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

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 $\textbf{coerce}^{L_i}(\cdot)$ 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 $\textbf{coerce}^{L_i}(\cdot)$ 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. Recall $\mathcal{P}_{rev}$ in Example 1. For the first coercion $\textbf{coerce}^{L_1}(\mathit{Reverse}\ x')$ (where $L_1 = \mathtt{b}^*\mathtt{a}^*\mathtt{e}$), we construct the following HMTT $\mathcal{P}_{rev}^{\mathcal{B}_1}$:

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

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 $\mathtt{a}(x'^{\mathcal{A}})$, $\mathit{Append}^{\mathcal{B}}$ 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 $(\mathtt{br}\ (\mathit{Reverse}^{\mathcal{A}}\ x'^{\mathcal{A}})\ (\mathit{Reverse}^{\mathcal{B}}\ x'^{\mathcal{A}}\ \mathtt{emp}))$ approximates the set of trees passed to $\mathbf{coerce}^{L_1}(\cdot)$ during the computation of $\mathbf{coerce}^{L_1}(\mathit{Reverse}\ x')$. 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 $(\mathit{Reverse}^{\mathcal{A}}\ x'^{\mathcal{A}})$ and $(\mathit{Reverse}^{\mathcal{B}}\ x'^{\mathcal{A}}\ \mathtt{emp})$ cover the former and the latter cases respectively. In the latter, the second parameter of $(\mathit{Reverse}^{\mathcal{B}}$ is an empty set, as the trees passed to $\mathbf{coerce}^{L_1}(\cdot)$ during the computation of x' are already covered by $x^{\mathcal{B}}$.

□

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

Example 5. Let us consider a higher-order version of the list reverse program:

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

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

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

Here, $\mathit{Append}^{\mathcal{A}}$ is the one obtained in Example 3. Note that $\mathit{Reverseh}^{\mathcal{B}}$ requires an additional argument $f^{\mathcal{B}}$, which generates all the trees passed to the coercion by f. □

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

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

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

Here, $\mathcal{A}(t)$ is the term obtained by replacing every coercion $\mathbf{coerce}^{L}(\cdot)$, variable x, and function name F in t with $\mathbf{gen}^{L}$, $x^{\mathcal{A}}$, and $F^{\mathcal{A}}$ respectively. $\mathtt{br}\ t_1 \cdots t_n$ stands for $\mathtt{br}\ t_1\ (\mathtt{br}\ t_2\ (\mathtt{br} \cdots (\mathtt{br}\ t_{n-1}\ t_n)))$ if $n \geq 2$, t_1 if $n = 1$, and $\mathtt{emp}$ if $n = 0$. For each terminal $a \in \mathrm{dom}(\Sigma)$, we obtain 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:

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

The following theorem states the correctness of the construction of $\mathcal{P}^{\mathcal{B}_i}$. See Appendix B for the proof.

Theorem 2. *Let $\mathcal{P} = (D, S)$ be an EHMTT and suppose that the coercions in $\mathcal{P}$ are $\mathbf{coerce}^{L_1}(\cdot), \ldots, \mathbf{coerce}^{L_m}(\cdot)$. If $\models (\mathcal{P}^{\mathcal{B}_i}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{B}(L_i))$ holds for each $i \in \{1, \ldots, m\}$, for any $T_1 \in \mathcal{L}(\mathcal{M}_1), \ldots, T_k \in \mathcal{L}(\mathcal{M}_k)$, $S\ T_1 \cdots T_k \not\longrightarrow_{\mathcal{P}}^{*}$* **Error**.

Our reduction from EHMTT to HMTT verification problems is incomplete, however, i.e. 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 $\emptyset$, 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 $x \in \{\mathtt{s}^n\ \mathtt{z} \mid n \geq 0\}$, we need the coercion annotation $L = \{\mathtt{pair}\ (\mathtt{s}^n\ \mathtt{z})\ (\mathtt{s}^n\ \mathtt{z}) \mid n \geq 0\}$, 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 of sequences 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. This problem can easily be remedied, however, by using a more elaborate construction of $\mathcal{B}(L_i)$, hence not a fundamental limitation.

Our overall method is also incomplete because of the incompleteness of the HMTT verification method [15] (unsurprisingly, as 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 [15] and Kobayashi's higher-order model checker TRecS [12].

Table 1 shows the results of 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:

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

Programs	O	C	R	S	$\mathrm{Sum_R}$	$\mathrm{Sum_S}$	$\mathrm{Q_I}$	$\mathrm{Q_O}$	$\mathrm{T_{Red}}$	Y/N	$\mathrm{T_{MC}}$
`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

Table 1. Experimental Results

The column "C" shows the number of coercions in each EHMTT. The columns "R" and "S" are the number of rules 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 rewriting rules. "$\mathrm{Sum_R}$" and "$\mathrm{Sum_S}$" respectively are the sum of the numbers of the rules 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). The column "$\mathrm{T_{MC}}$" 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 paper. `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 number n 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 [2], a higher-order XML-oriented functional language. `Split` takes a list of persons, and splits it into two lists of men and women. `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 [21] simulate simple XQuery queries. The program `Gapid-Html` is a composition of Tozawa's high-level tree transducers [22]. It takes a 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. It checks whether the children of each node of the document are empty, and if so, replaces the empty 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 rule for a function T with an argument x:

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

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 involve wrong coercions which lead to an **Error** to see that programs with wrong coercions are rejected. These 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 (thus the number of annotations required by our method) is much smaller than the number of rules (functions) 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 [2, 4, 9], which require type annotations for every function definition. Further 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.

```
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. `Split`

6 Related Work

As shown in Section 1, our verification method is based on recent advances on higher-order model checking [1, 11, 14, 20]. Ong [20] has proven the decidability of the model checking problem for recursion schemes, and Kobayashi has developed and implemented a type-based model checking algorithm [14].

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 [4, 7] can be used for verification of functional programs that manipulate user-defined recursive data structures. The original refinement type system [7] 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 [4] requires users to write type annotations for each function. For example, for $\mathcal{P}_{rev}$ in Example 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 at the end of Section 4, however, it may be useful to combine both approaches.

Several research groups have proposed typed XML processing languages [2, 9]. 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 [8] and regular expression pattern matching [10]. 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 [17, 18]. They are not Turing-complete, however, and thus less expressive than our EHMTTs. As shown in [16], ordinary macro and high-level tree transducers [5, 6] are subsumed by linear HMTTs, for which our EHMTT verification method is sound and complete.

String analysis [3, 19] 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.

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. We plan to investigate techniques for inferring coercion annotations, which would enable fully automatic verification of tree-processing programs. If the derived HMTTs are ordinary macro or high-level tree transducers [5, 6], annotations can indeed be inferred by the inverse inference technique. For general EHMTTs, we plan to apply techniques of machine learning. Addressing the limitations discussed at the end of Section 4 is also left for future work.

Acknowledgment

We would like to thank anonymous referees for useful comments.

References

1. Aehlig, K., de Miranda, J.G., Ong, C.H.L.: The monadic second order theory of trees given by arbitrary level-two recursion schemes is decidable. In: TLCA '05. LNCS, vol. 3461, pp. 39–54. Springer (2005)

2. Benzaken, V., Castagna, G., Frisch, A.: CDuce: an XML-centric general-purpose language. In: ICFP '03. pp. 51–63. ACM (2003)
3. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: SAS '03. LNCS, vol. 2694, pp. 1–18. Springer (2003)
4. Davies, R.: Practical refinement-type checking. Ph.D. thesis, Carnegie Mellon University (2005), chair-Pfenning, Frank
5. Engelfriet, J., Vogler, H.: Macro tree transducers. Journal of Computer and System Sciences 31(1), 71–146 (1985)
6. Engelfriet, J., Vogler, H.: High level tree transducers and iterated pushdown tree transducers. Acta Informatica 26(1/2), 131–192 (1988)
7. Freeman, T., Pfenning, F.: Refinement types for ML. In: PLDI '91. pp. 268–277. ACM (1991)
8. Hosoya, H., Frisch, A., Castagna, G.: Parametric polymorphism for XML. ACM Transactions on Programming Languages and Systems 32(1), 1–56 (2009)
9. Hosoya, H., Pierce, B.C.: XDuce: A statically typed XML processing language. ACM Transactions on Internet Technology 3(2), 117–148 (2003)
10. Hosoya, H., Vouillon, J., Pierce, B.C.: Regular expression types for XML. In: ICFP '00. pp. 11–22. ACM (2000)
11. Knapik, T., Niwinski, D., Urzyczyn, P.: Higher-order pushdown trees are easy. In: FoSSaCS '02. LNCS, vol. 2303, pp. 205–222. Springer (2002)
12. Kobayashi, N.: Model-checking higher-order functions. In: PPDP '09. pp. 25–36. ACM (2009)
13. Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs. In: POPL '09. pp. 416–428. ACM (2009)
14. Kobayashi, N., Ong, C.H.L.: A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In: LICS '09. pp. 179–188. IEEE (2009)
15. Kobayashi, N., Tabuchi, N., Unno, H.: Higher-order multi-parameter tree transducers and recursion schemes for program verification. In: POPL '10. pp. 495–508. ACM (2010)
16. Kobayashi, N., Tabuchi, N., Unno, H.: Higher-order multi-parameter tree transducers and recursion schemes for program verification. An extended version, available from `http://www.kb.ecei.tohoku.ac.jp/~koba/papers/hmtt.pdf` (2010)
17. Maneth, S., Berlea, A., Perst, T., Seidl, H.: XML type checking with macro tree transducers. In: PODS '05. pp. 283–294. ACM (2005)
18. Milo, T., Suciu, D., Vianu, V.: Typechecking for XML transformers. Journal of Computer and System Sciences 66(1), 66–97 (2003)
19. Minamide, Y.: Static approximation of dynamically generated web pages. In: WWW '05. pp. 432–441. ACM (2005)
20. Ong, C.H.L.: On model-checking trees generated by higher-order recursion schemes. In: LICS '06. pp. 81–90. IEEE (2006)
21. Schmidt, A., Waas, F., Kersten, M., Carey, M.J., Manolescu, I., Busse, R.: XMark: a benchmark for XML data management. In: VLDB '02. pp. 974–985. VLDB Endowment (2002)
22. Tozawa, A.: XML type checking using high-level tree transducer. In: FLOPS '06. LNCS, vol. 3945, pp. 81–96. Springer (2006)

Appendix

A Proof of Theorem 1

To prove Theorem 1, we use a variant $\Longrightarrow$ of the evaluation relation $\longrightarrow$ in Figure 2 which adopts the following rule instead of E-Coerce-Input and E-Input:

$$\frac{t \Longrightarrow^*_{\mathcal{P}} t' \qquad t'^{\perp} \in L^{\perp}}{E[\mathbf{coerce}^L(t)] \Longrightarrow_{\mathcal{P}} E[t'^{\perp}]} \qquad \text{(E-Coerce)}$$

Lemma 1. *If $S\ T_1 \cdots T_k \not\longrightarrow^*_{\mathcal{P}}$ **Error** and $S\ T_1 \cdots T_k \longrightarrow^* t$, then $S\ T_1 \cdots T_k \Longrightarrow^* t'$ is derivable for some t' such that $t^{\perp} = t'^{\perp}$.*

The term translation $\mathcal{A}(t)$ is formally defined by:

$$\begin{array}{l}
\mathcal{A}(a) = a \quad \mathcal{A}(x) = x^{\mathcal{A}} \quad \mathcal{A}(F) = F^{\mathcal{A}} \quad \mathcal{A}(t_1\ t_2) = \mathcal{A}(t_1)\ \mathcal{A}(t_2) \\
\mathcal{A}(\mathbf{case}\ t\ \mathbf{of}\ \{a_i\ \widetilde{y_i} \Rightarrow t_i\}_{i=1}^n) = \mathbf{case}\ \mathcal{A}(t)\ \mathbf{of}\ \{a_i\ \widetilde{y_i}^{\mathcal{A}} \Rightarrow \mathcal{A}(t_i)\}_{i=1}^n \\
\mathcal{A}(\mathbf{gen}^L) = \mathbf{gen}^L \quad \mathcal{A}(\mathbf{coerce}^L(t)) = \mathbf{gen}^L \quad \mathcal{A}(\underline{a}) = \underline{a}
\end{array}$$

We prepare lemmas for Theorem 1.

Lemma 2. $\mathcal{A}([t'/x]t) = [\mathcal{A}(t')/x^{\mathcal{A}}]\mathcal{A}(t)$

Lemma 3. *If $t \Longrightarrow t'$ is derivable without using* E-Coerce-Assert*, then $\mathcal{A}(t) \Longrightarrow^* \mathcal{A}(t')$ holds.*

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

E-App

We have

$$\begin{array}{l}
t = E[F\ \widetilde{t}] \\
t' = E[[\widetilde{t}/\widetilde{x}]t''] \\
F\ \widetilde{x} = t'' \in \mathcal{D}
\end{array}$$

We get $F^{\mathcal{A}}\ \widetilde{x}^{\mathcal{A}} = \mathcal{A}(t'') \in \mathcal{A}(D)$. We have

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

E-Case

We have

$$\begin{aligned}
&t = E[\textbf{case}\ \underline{a_i}\ \widetilde{t}\ \textbf{of}\ \{a_i\ \widetilde{y}_i \Rightarrow t_i\}_{i=1}^{n}]\\
&t' = E[[\widetilde{t}/\widetilde{y}_i]t_i]
\end{aligned}$$

We obtain

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

E-Case-Fail

We have

$$\begin{aligned}
&t = E[\textbf{case}\ \underline{a}\ \widetilde{t}\ \textbf{of}\ \{a_i\ \widetilde{y}_i \Rightarrow t_i\}_{i=1}^{n}]\\
&t' = E[\texttt{fail}]\\
&a \notin \{a_1, \ldots, a_n\}
\end{aligned}$$

We obtain

$$\begin{aligned}
\mathcal{A}(t) &= \mathcal{A}(E)[\mathcal{A}(\textbf{case}\ \underline{a}\ \widetilde{t}\ \textbf{of}\ \{a_i\ \widetilde{y}_i \Rightarrow t_i\}_{i=1}^{n})]\\
&= \mathcal{A}(E)[\textbf{case}\ \underline{a}\ \mathcal{A}(\widetilde{t})\ \textbf{of}\ \{a_i\ \widetilde{y}_i^{\mathcal{A}} \Rightarrow \mathcal{A}(t_i)\}_{i=1}^{n}]\\
&\Longrightarrow \mathcal{A}(E)[\texttt{fail}] \qquad \text{(By E-Case-Fail)}\\
&= \mathcal{A}(E[\texttt{fail}])\\
&= \mathcal{A}(t')
\end{aligned}$$

E-Gen

We have

$$\begin{aligned}
&t = E[\textbf{gen}^{L}]\\
&t' = E[\underline{a}\ \textbf{gen}^{L_1} \ldots \textbf{gen}^{L_n}]\\
&a\ L_1 \cdots L_n \subseteq L
\end{aligned}$$

$$\begin{aligned}
\mathcal{A}(t) &= \mathcal{A}(E)[\mathcal{A}(\textbf{gen}^{L})]\\
&= \mathcal{A}(E)[\textbf{gen}^{L}]\\
&\Longrightarrow \mathcal{A}(E)[\underline{a}\ \textbf{gen}^{L_1} \ldots \textbf{gen}^{L_n}] \qquad \text{(By E-Gen)}\\
&= \mathcal{A}(E[\underline{a}\ \textbf{gen}^{L_1} \ldots \textbf{gen}^{L_n}])\\
&= \mathcal{A}(t')
\end{aligned}$$

E-Coerce

We have

$$\begin{aligned}
&t = E[\mathbf{coerce}^L(t_1)]\\
&t' = E[t_2^\perp]\\
&t_1 \Longrightarrow^* t_2\\
&t_2^\perp \in L^\perp
\end{aligned}$$

$$\begin{aligned}
\mathcal{A}(t) \quad &= \quad \mathcal{A}(E)[\mathcal{A}(\mathbf{coerce}^L(t''))]\\
&= \quad \mathcal{A}(E)[\mathbf{gen}^L]\\
&\Longrightarrow^* \mathcal{A}(E)[t_2^\perp] \qquad (\text{By E-Gen and } \mathbf{gen}^\emptyset = \perp)\\
&= \quad \mathcal{A}(E[t_2^\perp])\\
&= \quad \mathcal{A}(t')
\end{aligned}$$

E-Coerce-Assert

This case is impossible

E-Assert-Error

This case is impossible

We now prove Theorem 1.

Proof of Theorem 1 Assume that $S\ T_1 \cdots T_k \not\longrightarrow^* \mathbf{Error}$ and $S\ T_1 \cdots T_k \longrightarrow^* t$. By Lemma 1, $S\ T_1 \cdots T_k \Longrightarrow^* t'$ is derivable for some t' such that $t^\perp = t'^\perp$. If the derivation of $S\ T_1 \cdots T_k \Longrightarrow^* t'$ uses T-Coerce-Assert, we have $t^\perp = t'^\perp = \perp \in \mathcal{L}(\mathcal{M}^\perp)$. Otherwise, by Lemma 3, we have $S^{\mathcal{A}}\ T_1 \cdots T_k \Longrightarrow^* \mathcal{A}(t')$. Thus, we get $t^\perp = t'^\perp = \mathcal{A}(t')^\perp \in \mathcal{L}(\mathcal{M}^\perp)$. □

B Proof of Theorem 2

Lemma 4. *If* $S\ T_1 \cdots T_k \not\Longrightarrow_{\mathcal{P}} \mathbf{Error}$*, then* $S\ T_1 \cdots T_k \not\longrightarrow^*_{\mathcal{P}} \mathbf{Error}$.

The term translation $\mathcal{B}_i(t)$ is extended with:

$$\mathcal{B}_i(\underline{a}) = a^{\mathcal{B}_i} \qquad \mathcal{B}_i(\mathbf{assert}^L(t)) = \mathcal{B}_i(t)$$

Lemma 5. $\mathcal{B}_i([t'/x]t) = [\mathcal{A}(t')/x^{\mathcal{A}}, \mathcal{B}_i(t')/x^{\mathcal{B}_i}]\mathcal{B}_i(t)$.

We write $t_1 \subseteq t_2$ if the set of trees generated by t_1 is a subset of that of t_2.

Lemma 6. $\mathtt{br}\ t\ [\mathtt{emp}/x]t' \supseteq [t/x]t'$.

Lemma 7. *If* $t_1 \supseteq\!\Longrightarrow t_2$*, then* $t_1 \Longrightarrow\!\supseteq t_2$.

We define $\mathcal{B}_i(E)[t^{\mathcal{A}}][t^{\mathcal{B}_i}]$ by:

$$\begin{array}{l}
\mathcal{B}_i([\ ])[t^{\mathcal{A}}][t^{\mathcal{B}_i}] = t^{\mathcal{B}_i}\\
\mathcal{B}_i(a\ t_1 \cdots t_{j-1}\ E\ t_{j+1} \cdots t_{\Sigma(a)})[t^{\mathcal{A}}][t^{\mathcal{B}_i}]\\
\quad = a^{\mathcal{B}_i}\ \mathcal{A}(t_1)\ \mathcal{B}_i(t_1) \cdots \mathcal{A}(t_{j-1})\ \mathcal{B}_i(t_{j-1})\ t^{\mathcal{A}}\ \mathcal{B}_i(E)[t^{\mathcal{A}}][t^{\mathcal{B}_i}]\ \mathcal{A}(t_{j+1})\ \mathcal{B}_i(t_{j+1}) \cdots \mathcal{A}(t_{\Sigma(a)})\ \mathcal{B}_i(t_{\Sigma(a)})\\
\mathcal{B}_i(\textbf{case}\ E\ \textbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow t_j\}_{j=1}^{n})[t^{\mathcal{A}}][t^{\mathcal{B}_i}]\\
\quad = \texttt{br}\ \mathcal{B}_i(E)[t^{\mathcal{A}}][t^{\mathcal{B}_i}]\ (\textbf{case}\ \mathcal{A}(E)[t^{\mathcal{A}}]\ \textbf{of}\ \{a_j\ \widetilde{y}_j^{\mathcal{A}} \Rightarrow [\widetilde{\texttt{emp}}/\widetilde{y}_j^{\mathcal{B}_i}]\mathcal{B}_i(t_j)\}_{j=1}^{n})\\
\mathcal{B}_i(\textbf{assert}^{L}(E)) = \mathcal{B}_i(E)[t^{\mathcal{A}}][t^{\mathcal{B}_i}]
\end{array}$$

Lemma 8. *If $t \Longrightarrow t'$, then $\mathcal{B}_i(t) \Longrightarrow^{*}\supseteq \mathcal{B}_i(t')$.*

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

E-App

We have

$$\begin{array}{l}
t = E[F\ t_1 \ldots t_n]\\
t' = E[[t_1/x_1, \ldots, t_n/x_n]t'']\\
F\ x_1 \ldots x_n = t'' \in \mathcal{D}
\end{array}$$

We get $F^{\mathcal{A}}\ x_1^{\mathcal{A}} \ldots x_n^{\mathcal{A}} = \mathcal{A}(t'') \in \mathcal{B}_i(D)$. $F^{\mathcal{B}_i}\ x_1^{\mathcal{A}}\ x_1^{\mathcal{B}_i} \ldots x_n^{\mathcal{A}}\ x_n^{\mathcal{B}_i} = \mathcal{B}_i(t'') \in \mathcal{B}_i(D)$. We have

$$\begin{array}{rcl}
\mathcal{B}_i(t) & = & \mathcal{B}_i(E)[\mathcal{A}(F\ t_1 \ldots t_n)][\mathcal{B}_i(F\ t_1 \ldots t_n)]\\
& = & \mathcal{B}_i(E)[\mathcal{A}(F\ t_1 \ldots t_n)][F^{\mathcal{B}_i}\ \mathcal{A}(t_1)\ \mathcal{B}_i(t_1) \ldots \mathcal{A}(t_n)\ \mathcal{B}_i(t_n)]\\
& \Longrightarrow & \mathcal{B}_i(E)[\mathcal{A}(F\ t_1 \ldots t_n)][[\mathcal{A}(t_1)/x_1^{\mathcal{A}}, \mathcal{B}_i(t_1)/x_1^{\mathcal{B}_i}, \ldots, \mathcal{A}(t_n)/x_n^{\mathcal{A}}, \mathcal{B}_i(t_n)/x_n^{\mathcal{B}_i}]\mathcal{B}_i(t'')] \quad \text{(By E-App)}\\
& = & \mathcal{B}_i(E)[\mathcal{A}(F\ t_1 \ldots t_n)][\mathcal{B}_i([t_1/x_1, \ldots t_n/x_n]t'')] \qquad \text{(By Lemma 5)}\\
& \Longrightarrow & \mathcal{B}_i(E)[\mathcal{A}([t_1/x_1, \ldots, t_n/x_n]t'')][\mathcal{B}_i([t_1/x_1, \ldots t_n/x_n]t'')] \qquad \text{(By E-App)}\\
& = & \mathcal{B}_i(E[[t_1/x_1, \ldots t_n/x_n]t''])\\
& = & \mathcal{B}_i(t')
\end{array}$$

E-Case

We have

$$\begin{array}{l}
t = E[\textbf{case}\ \underline{a_j}\ \widetilde{t}\ \textbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow t_j\}_{j=1}^{n}]\\
t' = E[[\widetilde{t}/\widetilde{y}_j]t_j]
\end{array}$$

We obtain

$$\begin{array}{rcl}
\mathcal{B}_i(t) & = & \mathcal{B}_i(E)[\mathcal{A}(\textbf{case}\ \underline{a_j}\ \widetilde{t}\ \textbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow t_j\}_{j=1}^{n})][\mathcal{B}_i(\textbf{case}\ \underline{a_j}\ \widetilde{t}\ \textbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow t_j\}_{j=1}^{n})]\\
& \Longrightarrow & \mathcal{B}_i(E)[\mathcal{A}([\widetilde{t}/\widetilde{y}_j]t_j)][\mathcal{B}_i(\textbf{case}\ \underline{a_j}\ \widetilde{t}\ \textbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow t_j\}_{j=1}^{n})] \qquad \text{(By E-Case)}\\
& = & \mathcal{B}_i(E)[\mathcal{A}([\widetilde{t}/\widetilde{y}_j]t_j)][\texttt{br}\ \mathcal{B}_i(\underline{a_j}\ \widetilde{t})\ (\textbf{case}\ \underline{a_j}\ \mathcal{A}(\widetilde{t})\ \textbf{of}\ \{a_j\ \widetilde{y}_j^{\mathcal{A}} \Rightarrow [\widetilde{\texttt{emp}}/\widetilde{y}_j^{\mathcal{B}_i}]\mathcal{B}_i(t_j)\}_{j=1}^{n})]
\end{array}$$

$$\begin{aligned}
&\Longrightarrow \mathcal{B}_i(E)[\mathcal{A}([\widetilde{t}/\widetilde{y}_j]t_j)][\texttt{br}\ \mathcal{B}_i(\underline{a_j}\ \widetilde{t})\ [\mathcal{A}(\widetilde{t})/\widetilde{y}_j^{\mathcal{A}}, \widetilde{\texttt{emp}}/\widetilde{y}_j^{\mathcal{B}_i}]\mathcal{B}_i(t_j)] && \text{(By E-Case)}\\
&\supseteq\ \mathcal{B}_i(E)[\mathcal{A}([\widetilde{t}/\widetilde{y}_j]t_j)][[\mathcal{A}(\widetilde{t})/\widetilde{y}_j^{\mathcal{A}}, \mathcal{B}_i(\widetilde{t})/\widetilde{y}_j^{\mathcal{B}_i}]\mathcal{B}_i(t_j)] && \text{(By Lemma 6)}\\
&=\ \mathcal{B}_i(E)[\mathcal{A}([\widetilde{t}/\widetilde{y}_j]t_j)][\mathcal{B}_i([\widetilde{t}/\widetilde{y}_j]t_j)] && \text{(By Lemma 5)}\\
&=\ \mathcal{B}_i(E[[\widetilde{t}/\widetilde{y}_j]t_j])\\
&=\ \mathcal{B}_i(t')
\end{aligned}$$

E-Case-Fail

We have

$$\begin{aligned}
&t = E[\mathbf{case}\ \underline{a}\ \widetilde{t}\ \mathbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow t_j\}_{j=1}^{n}]\\
&t' = E[\texttt{fail}]\\
&a \notin \{a_1, \ldots, a_n\}
\end{aligned}$$

We obtain

$$\begin{aligned}
\mathcal{B}_i(t)\ &=\ \mathcal{B}_i(E)[\mathcal{A}(\mathbf{case}\ \underline{a}\ \widetilde{t}\ \mathbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow t_j\}_{j=1}^{n})][\mathcal{B}_i(\mathbf{case}\ \underline{a}\ \widetilde{t}\ \mathbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow t_j\}_{j=1}^{n})]\\
&\Longrightarrow \mathcal{B}_i(E)[\texttt{fail}][\mathcal{B}_i(\mathbf{case}\ \underline{a}\ \widetilde{t}\ \mathbf{of}\ \{a_j\ \widetilde{y}_j \Rightarrow t_j\}_{j=1}^{n})] \qquad \text{(By E-Case-Fail)}\\
&\supseteq\ \mathcal{B}_i(E)[\texttt{fail}][\texttt{emp}]\\
&=\ \mathcal{B}_i(E)[\mathcal{A}(\texttt{fail})][\mathcal{B}_i(\texttt{fail})]\\
&=\ \mathcal{B}_i(E[\texttt{fail}])\\
&=\ \mathcal{B}_i(t')
\end{aligned}$$

E-Gen

We have

$$\begin{aligned}
&t = E[\mathbf{gen}^{L}]\\
&t' = E[\underline{a}\ \mathbf{gen}^{L_1} \ldots \mathbf{gen}^{L_n}]\\
&a\ L_1 \cdots L_n \subseteq L
\end{aligned}$$

$$\begin{aligned}
\mathcal{B}_i(t)\ &=\ \mathcal{B}_i(E)[\mathcal{A}(\mathbf{gen}^{L})][\mathcal{B}_i(\mathbf{gen}^{L})]\\
&\Longrightarrow \mathcal{B}_i(E)[\mathcal{A}(\underline{a}\ \mathbf{gen}^{L_1} \ldots \mathbf{gen}^{L_n})][\mathcal{B}_i(\mathbf{gen}^{L})] \qquad \text{(By E-Gen)}\\
&=\ \mathcal{B}_i(E)[\mathcal{A}(\underline{a}\ \mathbf{gen}^{L_1} \ldots \mathbf{gen}^{L_n})][\texttt{emp}]\\
&=\ \mathcal{B}_i(E)[\mathcal{A}(\underline{a}\ \mathbf{gen}^{L_1} \ldots \mathbf{gen}^{L_n})][\mathcal{B}_i(\underline{a}\ \mathbf{gen}^{L_1} \ldots \mathbf{gen}^{L_n})]\\
&=\ \mathcal{B}_i(E[\underline{a}\ \mathbf{gen}^{L_1} \ldots \mathbf{gen}^{L_n}])\\
&=\ \mathcal{B}_i(t')
\end{aligned}$$

E-Coerce

We have

$$\begin{aligned}
&t = E[\mathbf{coerce}^{L_j}(t_1)]\\
&t' = E[t_2^{\perp}]\\
&t_1 \Longrightarrow^{*} t_2\\
&t_2^{\perp} \in L^{\perp}
\end{aligned}$$

$$\begin{array}{rcll}
\mathcal{B}_i(t) & = & \mathcal{B}_i(E)[\mathcal{A}(\mathbf{coerce}^{L_j}(t_1))][\mathcal{B}_i(\mathbf{coerce}^{L_j}(t_1))] & \\
 & \Longrightarrow^* & \mathcal{B}_i(E)[\mathcal{A}(t_2^{\perp})][\mathcal{B}_i(\mathbf{coerce}^{L_j}(t_1))] & \text{(By E-G{\scriptsize EN})} \\
 & \supseteq & \mathcal{B}_i(E)[\mathcal{A}(t_2^{\perp})][\texttt{emp}] & \\
 & = & \mathcal{B}_i(E)[\mathcal{A}(t_2^{\perp})][\mathcal{B}_i(t_2^{\perp})] & \\
 & = & \mathcal{B}_i(E[t_2^{\perp}]) & \\
 & = & \mathcal{B}_i(t') &
\end{array}$$

E-Coerce-Assert

We have

$$\begin{array}{l}
t = E[\mathbf{coerce}^{L_j}(t'')] \\
t' = \mathbf{assert}^{L_j}(t'')
\end{array}$$

$$\begin{array}{rl}
\mathcal{B}_i(t) & = \mathcal{B}_i(E)[\mathcal{A}(\mathbf{coerce}^{L_j}(t''))][\mathcal{B}_i(\mathbf{coerce}^{L_j}(t''))] \\
 & \supseteq \mathcal{B}_i(E)[\mathcal{A}(\mathbf{coerce}^{L_j}(t''))][\mathcal{B}_i(t'')] \\
 & \supseteq \mathcal{B}_i(t'') \\
 & = \mathcal{B}_i(\mathbf{assert}^{L_j}(t'')) \\
 & = \mathcal{B}_i(t')
\end{array}$$

E-Assert-Error

This case is impossible

Proof of Theorem 2 Assume 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 $S\ T_1 \cdots T_k \Longrightarrow^* E[\mathbf{coerce}^{L_j}(t)]$ for some $T_1 \in \mathcal{L}(\mathcal{M}_1), \ldots, T_k \in \mathcal{L}(\mathcal{M}_k)$. By Lemma 8, we obtain $S^{\mathcal{B}_j}\ T_1\ \texttt{emp} \cdots T_k\ \texttt{emp} (\Longrightarrow^* \supseteq)^* \mathcal{B}_j(E[\mathbf{coerce}^{L_j}(t)])$ By Lemma 7, we get $S^{\mathcal{B}_j}\ T_1\ \texttt{emp} \cdots T_k\ \texttt{emp} \Longrightarrow^* \supseteq \mathcal{B}_j(E[\mathbf{coerce}^{L_j}(t)])$. Thus, for any t' such that $t \Longrightarrow^* t'$, $t'^{\perp} \in \mathcal{B}(L_j) = L_j^{\perp}$. Therefore, $E[\mathbf{coerce}^{L_j}(t)] \not\Longrightarrow^*$ **Error**. By Lemma 4, we get $E[\mathbf{coerce}^{L_j}(t)] \not\longrightarrow^*$ **Error**. □