Higher-Order Multi-Parameter Tree Transducers and Recursion Schemes for Program Verification

Naoki Kobayashi
Tohoku University
koba@ecei.tohoku.ac.jp

Naoshi Tabuchi
Tohoku University
tabee@kb.ecei.tohoku.ac.jp

Hiroshi Unno
Tohoku University
uhiro@kb.ecei.tohoku.ac.jp

Abstract

We introduce higher-order, multi-parameter, tree transducers (HMTTs, for short), which are kinds of higher-order tree transducers that take input trees and output a (possibly infinite) tree. We study the problem of checking whether the tree generated by a given HMTT conforms to a given output specification, provided that the input trees conform to input specifications (where both input/output specifications are regular tree languages). HMTTs subsume higher-order recursion schemes and ordinary tree transducers, so that their verification has a number of potential applications to verification of functional programs using recursive data structures, including resource usage verification, string analysis, and exact type-checking of XML-processing programs.

We propose a sound but incomplete verification algorithm for the HMTT verification problem: the algorithm reduces the verification problem to a model-checking problem for higher-order recursion schemes extended with finite data domains, and then uses (an extension of) Kobayashi's algorithm for model-checking recursion schemes. While the algorithm is incomplete (indeed, as we show in the paper, the verification problem is undecidable in general), it is sound and complete for a subclass of HMTTs called *linear HMTTs*. We have applied our HMTT verification algorithm to various program verification problems and obtained promising results.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meaning of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Verification

1. Introduction

Kobayashi [20] has recently proposed a verification method for higher-order functional programs based on Ong's decidability result on model-checking recursion schemes [31]. A higher-order recursion scheme (recursion scheme, for short) is a grammar for generating a (possibly infinite) tree. It is an extension of regular tree grammars, where non-terminal symbols can take trees and higher-order functions on trees as parameters. For example, the following grammar $\mathcal{G}_0$ is an order-1 recursion scheme, where the non-terminal F takes a tree as an argument.

$$S \to F\,\mathtt{c} \qquad F\,x \to \mathtt{a}\,x\,(F\,(\mathtt{b}\,x))$$

Here, each non-terminal has exactly one rewrite rule. By infinitary rewriting of the start symbol S:

$$S \longrightarrow F\,\mathtt{c} \longrightarrow \mathtt{a}\,\mathtt{c}\,(F\,(\mathtt{b}\,\mathtt{c})) \longrightarrow \cdots,$$

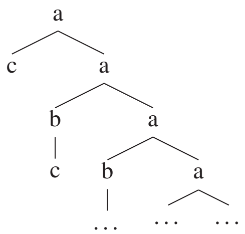

Figure 1. The tree generated by $\mathcal{G}_0$.

we obtain the infinite tree shown in Figure 1. Ong [31] has shown that modal mu-calculus model-checking of recursion schemes ("Given a recursion scheme $\mathcal{G}$ and a modal mu-calculus formula φ, does the tree generated by $\mathcal{G}$ satisfy φ?") is n-EXPTIME-complete (where n is the order of the recursion scheme $\mathcal{G}$). The idea of Kobayashi's verification method [20] is to translate a functional program into a recursion scheme that generates a tree whose paths represent all the possible event sequences of the program, so that temporal properties of the functional program can be verified by model-checking the recursion scheme. For example, consider the following program that accesses a file (where * denotes a random boolean value):

```
let x = open_in "foo" in
let rec f() = if * then close(x) else read(x); f()
in f()
```

It can be translated into the following recursion scheme $\mathcal{G}_1$:

$$S \to F\,\mathtt{e} \qquad F\,k \to \mathtt{br}\,(\mathtt{c}\,k)\,(\mathtt{r}\,(F\,k))$$

Here, r, c, br and e denote a read operation, a close operation, a non-deterministic branch, and program termination respectively. The recursion scheme generates the tree shown in Figure 2, which represents all the possible event (i.e. read, write, branch, and termination) sequences of the program. Kobayashi [20] applied the verification method to resource usage verification [14] (the problem of checking whether a program accesses resources such as files in a valid manner), and showed that it is sound and complete for the simply-typed λ-calculus extended with recursion, resource creation/access primitives, and booleans. The completeness follows intuitively because recursion schemes are essentially terms of the simply-typed λ-calculus with recursion and

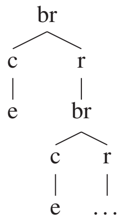

Figure 2. The tree generated by $\mathcal{G}_1$.

tree constructors, so that they are already almost as expressive as the source language (of the simply-typed λ-calculus with resources), and no information is lost by the translation. Although the model-checking of recursion schemes has extremely high worst-case time complexity (n-EXPTIME-complete for the full modal mu-calculus), Kobayashi [18] constructed a model-checker for recursion schemes, and showed that it works well for realistic inputs. Thus, the verification method based on recursion schemes seems to be one of the promising methods for higher-order program verification.

The main limitation of recursion schemes from a programming language point of view is that there are tree constructors but not destructors. Because of this limitation, one cannot naturally model programs operating over infinite data domains such as integers, lists, and trees.[1] For example, consider the following function `merge`:

```
let rec merge x y =
  case x of [] => copy y
          | a::x' => merge_a x' y
          | b::y' => merge_b x' y
and merge_a x y =
  case y of [] => a::(copy x)
          | a::y' => a::a::(merge x y')
          | b::y' => a::(merge_b y' x)
and merge_b x y =
  case y of [] => b::(copy x)
          | a::y' => b::(merge_a y' x)
          | b::y' => b::b::(merge x y')
and copy x =
  case x of [] => []
          | a::x' => a::(copy x')
          | b::y' => b::(copy y')
```

The function `merge` merges two lists consisting of `a` and `b`. It recursively destructs `x` and `y`, which cannot be expressed by a recursion scheme. Functions operating over recursive data structures are ubiquitous in functional programming, so that not being able to handle them is a great limitation for the approach of model-checking functional programs by modeling them as recursion schemes.

To relax the limitation above, we introduce an extension of recursion schemes called *higher-order, multi-parameter tree transducers* (HMTTs, for short). As in other (top-down) tree transducers [7, 8], we classify trees into input and output trees: only constructors can be applied to output trees, and only destructors can be applied to input trees. Unlike in ordinary tree transducers, however, each function symbol (non-terminal) can take multiple input trees as arguments (as in the function `merge` above). Furthermore, as in recursion schemes (and high-level tree transducers [8]), higher-order functions can be used. The function `merge` above is expressed as the following HMTT $\mathcal{T}_2$:

```
Merge x y ->
 case(x,Copy y, x'.Merge_a x' y, x'.Merge_b x' y).
Merge_a x y ->
 case(y,
      a(Copy x),
      y'.a(a(Merge x y')),
      y'.a(Merge_b y' x)).
...
```

Here, strings are expressed as linear trees (consisting of only terminal symbols of arity 1 or 0). $\mathit{case}(e, \widetilde{x}_1.e_1, \ldots, \widetilde{x}_n.e_n)$ matches a tree e with a pattern $\mathtt{a}_i\,\widetilde{t}$, and reduces to $[\widetilde{t}/\widetilde{x}_i]e_i$. HMTTs subsume both higher-order recursion schemes and various kinds of tree transducers (such as macro tree transducers and high-level tree transducers [7, 8]).

For HMTTs, we consider the following verification problem $(\mathcal{T}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})$: "Given an HMTT $\mathcal{T}$ that takes k (possibly infinite) input trees, and Büchi tree automata with a trivial acceptance condition (where all the states are final) $\mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M}$, does $\mathcal{T}$ always output a (possibly infinite) tree accepted by $\mathcal{M}$, given trees accepted by $\mathcal{M}_1, \ldots, \mathcal{M}_k$ as inputs?" For example, let $\mathcal{M}$ be a tree automaton that accepts linear trees labeled by elements of $a^*b^*e + a^\omega + a^*b^\omega$ (where a, b and e are terminal symbols of arity 1, 1, and 0 respectively). Then, the verification problem $(\mathcal{T}, \mathcal{M}, \mathcal{M}, \mathcal{M})$ is the problem of deciding whether $\mathcal{T}$ produces a linear tree labeled by an element of $a^*b^*e + a^\omega + a^*b^\omega$ (thus, a tree like $b(a(e))$ is excluded), given two input trees labeled by elements of $a^*b^*e + a^\omega + a^*b^\omega$.

In the present paper, we give a sound but *incomplete* algorithm for the HMTT verification problem. The algorithm consists of two phases. First, we transform the HMTT verification problem into a model-checking problem for recursion schemes extended with constructors and destructors for finite data domains (such as booleans). We then solve the model-checking problem for the recursion scheme by using an extension of Kobayashi's model checking algorithm for recursion schemes [18]. The idea of the first phase is to approximate an input tree by an automaton state. For example, recall the HMTT $\mathcal{T}_2$ (corresponding to the merge function). The automaton $\mathcal{M}$ has two states q_0 and q_1, with transition $\delta(q_0, a) = q_0, \delta(q_0, b) = q_1, \delta(q_0, e) = \epsilon, \delta(q_1, b) = q_1, \delta(q_1, e) = \epsilon$. Thus, the output of $\mathcal{T}_2$ applied to linear trees in $\mathcal{L}(\mathcal{M})$ is approximated by the following recursion scheme with finite data domains $\{q_0, q_1\}$:

```
S -> Merge q0 q0.
Merge x y ->
 case(x,
      br3 (Copy y) (Merge_a q0 y) (Merge_b q1 y),
      br (Copy y) (Merge_b q1 y)).
Merge_a x y ->
 case(y,
      br3 (a(Copy x)) (a(a(Merge x q0)))
          (a(Merge_b q1 x))
      br (a(Copy x)) (a(Merge_b q1 x))).
...
```

Here, the non-terminal `Merge` now takes elements of $\{q_0, q_1\}$ as arguments instead of trees, and the case statement $\mathtt{case}(e, e_0, e_1)$ reduces to e_0 if the value of e_0 is q_0, and reduces to e_1 otherwise. The three branches of `br3 (Copy y) (Merge_a q0 y) (Merge_b q1 y)` for the case $x = q_0$ simulates the cases where the input tree x is of the form e, $a\,t_1$, and $b\,t_2$ respectively. The tree generated by the recursion scheme (with finite data domains) constructed in this manner contains all the possible outputs of the HMTT (for valid

[1] It is possible to encode trees as functions instead of using primitive tree constructors. The tree operations realized in this way is however limited, as recursion schemes must be simply-typed.

inputs). Thus, if the recursion scheme generates only valid trees, so does the HMTT.

As mentioned above, our verification algorithm is sound but incomplete: it does not accept an invalid HMTT, but may reject a valid HMTT. As we show in the paper, the HMTT verification problem is undecidable in general, so that we cannot hope to find a complete algorithm. We show, however, that the proposed algorithm is complete for a subclass of HMTTs called *linear* HMTTs (with certain additional assumptions). Intuitively, an HMTT is linear if it traverses each input tree at most once. (For example, the merge function above is linear since it traverses `x` and `y` just once.) Note that, even for the decidable class of linear HMTTs, our verification problem subsumes the model checking problem for recursion schemes (where properties are restricted to those described by Büchi tree automata with a trivial acceptance condition). Linear HMTTs also subsume (deterministic and total) high-level tree transducers [8] in the sense that any high-level tree transducer can be transformed into a linear HMTT by using a standard program transformation technique (see [**?**]).

There are many potential applications of our HMTT verification method. As HMTTs subsume recursion schemes and high-level tree transducers, immediate applications include (i) the resource usage verification considered by Kobayashi [20], and (ii) exact type-checking of XML-processing programs [24, 26, 27, 33]. For (i), while Kobayashi [18, 20] considered closed programs, we can now deal with programs that take recursive data structures as arguments. For (ii), unlike previous approaches to using macro or high-level tree transducers, we can verify programs that take multiple XML documents as inputs. Furthermore, thanks to the efficient model-checking algorithm for recursion schemes, our HMTT verification algorithm is reasonably fast even for higher-order cases; on the other hand, previous approaches based on transducers do not seem to scale well for higher-order cases [33]. Other applications include: (iii) string analysis [3, 28], and (iv) verification of programs that use other recursive data types (e.g. the list-ness of the output of the function “flatten” converting nested lists into flat lists). For (iii), the goal is to check that a program always generates a valid string as output. Web applications often generate HTML files, and it is important to check that there is no cross-site scripting vulnerabilities and also that the output conforms to the HTML format [28]. Previous approaches [3, 28] use a kind of control flow analysis to approximate the output string as a regular or context-free language, and then compare it with the output specification. By using HMTT, we can more precisely approximate the output string (and indeed, no approximation is required in certain cases).

The rest of this paper is structured as follows. Section 2 introduces HMTTs and defines their verification problems. Section 3 introduces higher-order recursion schemes extended with finite data domains. Section 4 gives a transformation of HMTT verification problems into model-checking problems for recursion schemes with finite data domains. Section 5 extends Kobayashi’s type-based method for model-checking recursion schemes [18, 20], to deal with finite data domains. Section 6 discusses the complexity of our HMTT verification algorithm. Section 7 discusses applications of our verification method, and Section 8 reports preliminary experiments. Section 9 discusses related work, and Section 10 concludes the paper. A longer version of this paper [**?**], containing proofs and more details on the experiments, is available from `http://www.kb.ecei.tohoku.ac.jp/~koba/papers/hmtt.pdf`.

2. Higher-Order, Multi-Parameter Tree Transducer

This section defines higher-order, multi-parameter tree transducers (HMTTs) and their verification problems. From a programming language point of view, an HMTT is a term of the simply-typed λ-calculus extended with recursion and tree constructors/destructors. Trees are classified into *input trees*, which can only be destructed, and *output trees*, which can only be constructed.

DEFINITION 2.1 (sorts). *The set of sorts is given by:*

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

The sort `i` describes input trees, while `o` describes output trees. The sort $\kappa_1 \to \kappa_2$ describes a function that takes an element of sort κ_1 and returns an element of sort κ_2

DEFINITION 2.2. A *higher-order, multi-parameter tree transducer* (HMTT for short) $\mathcal{T}$ is a quadruple $(\Sigma, \mathcal{N}, \mathcal{R}, S)$, where:

- Σ is a *ranked alphabet*. i.e. a map from a finite set $\{a_1, \ldots, a_N\}$ of symbols called *terminals* to non-negative integers.
- $\mathcal{N}$ is a map from a finite set of symbols called *non-terminals* to sorts.
- $\mathcal{R}$ is a set of rewriting rules of the form $F\,x_1\,\cdots\,x_n \to t$, where $F \in \mathit{dom}(\mathcal{N})$ is a non-terminal and t is a term. Here, the set of terms is defined by:

$$t ::= a \mid x \mid F \mid t_1\,t_2 \mid \mathbf{case}(x, \widetilde{y}_1.t_1, \ldots, \widetilde{y}_N.t_N),$$

where $\widetilde{y}_i$ abbreviates a sequence of variables, whose length must coincide with the arity of a_i.
- S is a non-terminal called the *start symbol*. $\mathcal{N}(S)$ must be of the form $\mathtt{i} \to \cdots \to \mathtt{i} \to \mathtt{o}$.

Furthermore, each rule $F\,x_1 \cdots x_n \to t$ must be well-sorted, i.e. $\vdash F\,x_1\,\cdots\,x_m \to t$ must be derivable by using the following sort assignment rules:

$$\mathcal{K} \vdash F : \mathcal{N}(F)$$

$$\mathcal{K} \vdash a : \underbrace{\mathtt{o} \to \cdots \to \mathtt{o}}_{\Sigma(a)} \to \mathtt{o}$$

$$\mathcal{K}, x : \kappa \vdash x : \kappa$$

$$\frac{\mathcal{K} \vdash t_1 : \kappa_1 \to \kappa_2 \qquad \mathcal{K} \vdash t_2 : \kappa_1}{\mathcal{K} \vdash t_1 t_2 : \kappa_2}$$

$$\frac{\begin{array}{c}\mathcal{K} \vdash x : \mathtt{i} \\ \mathcal{K}, \widetilde{y}_i : \widetilde{\mathtt{i}} \vdash t_i : \mathtt{o}\end{array}}{\mathcal{K} \vdash \mathbf{case}(x, \widetilde{y}_1.t_1, \ldots, \widetilde{y}_N.t_N) : \mathtt{o}}$$

$$\frac{\begin{array}{c}\mathcal{N}(F) = \kappa_1 \to \cdots \to \kappa_m \to \mathtt{o} \\ x_1 : \kappa_1, \ldots, x_m : \kappa_m \vdash t : \mathtt{o}\end{array}}{\vdash F\,x_1\,\cdots\,x_m \to t}$$

Note that in the sort assignment rule for case-expressions, x must be of sort `i`; thus, pattern matching on trees are only allowed on input trees (of sort `i`).

REMARK 2.1. Each terminal symbol is used as a constructor for input trees (of sort `i`) as well as for output trees (of sort `o`). In the sort assignment rule above, however, only the sort $\mathtt{o} \to \cdots \to \mathtt{o} \to \mathtt{o}$ is assigned, as input trees cannot be constructed in a recursion scheme. In an environment ρ introduced below, a terminal of arity k has sort $\underbrace{\mathtt{i} \to \cdots \to \mathtt{i}}_{k} \to \mathtt{i}$. □

A Σ-labeled ranked tree, written T, is a mapping from $\{1, \ldots, A\}^*$ (where A is the largest arity of symbols in Σ) to

$dom(\Sigma)$, such that (i) $dom(T)$ is closed under the prefix operation, and (ii) if $T(x) = a$, then $\{i \mid xi \in dom(T)\} = \{1, \ldots, \Sigma(a)\}$.

The set of evaluation contexts is defined by:

$$E ::= [\,] \mid a\, t_1 \cdots t_{j-1}\, [\,]\, t_{j+1} \cdots t_{\Sigma(a)}$$

Let ρ be a mapping from a finite set of variables (of sort $\texttt{i}$) to a set of pairs consisting of a Σ-labeled ranked tree and a non-negative integer. (The second element of such a pair, called *uses*, will later be used for defining the linearity condition.) Let $\mathcal{T} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$. The reduction relation $(t, \rho) \longrightarrow_{\mathcal{T}} (t', \rho')$ is defined by:

$$\frac{F\, x_1 \cdots x_m \to t \in \mathcal{R}}{(E[F\, t_1 \cdots t_m], \rho) \longrightarrow_{\mathcal{T}} (E[[t_1/x_1, \ldots, t_m/x_m]t], \rho)}$$

$$\frac{\rho(x) = (a_i\, \widetilde{T}, j) \qquad \rho' = \rho\{x \mapsto (a_i\, \widetilde{T}, j+1), \widetilde{y}' \mapsto (\widetilde{T}, \widetilde{0})\} \quad (\widetilde{y}' \text{ are fresh})}{(E[\mathbf{case}(x, \widetilde{y}_1.t_1, \ldots, \widetilde{y}_N.t_N)], \rho) \longrightarrow_{\mathcal{T}} (E[[\widetilde{y}'/\widetilde{y}_i]t_i], \rho')}$$

Here, $[\widetilde{t}/\widetilde{x}]t'$ denotes the term obtained from t' by simultaneously replacing $\widetilde{x}$ with $\widetilde{t}$. In the rule above, $\widetilde{T}$ denotes a sequence of (possibly infinite) trees, and $a_i\, \widetilde{T}$ denotes a tree whose root is labeled by a_i and children are $\widetilde{T}$.

Suppose that t is a term of type $\underbrace{\texttt{i} \to \cdots \to \texttt{i}}_{k} \to \texttt{o}$. We write $\llbracket \mathcal{T}, t, I_1, \ldots, I_k \rrbracket$ for the tree obtained by infinitary rewriting of $(t\, x_1 \cdots x_k, \{x_1 \mapsto (I_1, 0), \ldots, x_k \mapsto (I_k, 0)\})$. More precisely, it is defined as follows. Given a term t, we write $t^{\perp}$ for the finite tree inductively defined by (i) $(a s_1 \cdots s_n)^{\perp} = a({s_1}^{\perp}) \cdots ({s_n}^{\perp})$ (where $n \geq 0$), and (ii) $t^{\perp} = \perp$ if t is of the from $F s_1 \cdots s_n$ or $\mathbf{case}(x, \widetilde{y}.t_1, \ldots, \widetilde{y}.t_n)$. For example, $(\texttt{a}\,\texttt{c}\,(F\,(\texttt{b}\,\texttt{c})))^{\perp} = \texttt{a}\,\texttt{c}\,\perp$. We define $\llbracket \mathcal{T}, t, I_1, \ldots, I_k \rrbracket$ as the (possibly infinite) $(\Sigma \cup \{\perp \mapsto 0\})$-labeled tree $\bigsqcup\{{t'}^{\perp} \mid (t\, x_1 \cdots x_k, \{x_1 \mapsto (I_1, 0), \ldots, x_k \mapsto (I_k, 0)\}) \longrightarrow^{*}_{\mathcal{T}} (t', \rho')\}$, where $\bigsqcup_i T_i$ is defined by $(\bigsqcup_i T_i)(\pi) = \bigsqcup_i (T_i(\pi))$ for every $\pi \in \{1, \ldots, A\}^{*}$ (where A is the largest arity), with $\perp \sqcup a = a$ for every $a \in dom(\Sigma)$.

EXAMPLE 2.1. Consider the HMTT $\mathcal{T} = (\Sigma, \mathcal{N}, \mathcal{R}, \mathit{Rev})$ where:

$$\begin{array}{l}
\Sigma = \{\texttt{a}_1 \mapsto 1, \texttt{a}_2 \mapsto 1, \texttt{a}_3 \mapsto 0\} \\
\mathcal{N} = \{\mathit{Rev} \mapsto \texttt{i} \to \texttt{o}, \mathit{RevSub} \mapsto \texttt{i} \to \texttt{o} \to \texttt{o}\} \\
\mathcal{R} = \{\mathit{Rev}\, x \to \mathit{RevSub}\, x\, \texttt{a}_3, \\
\qquad \mathit{RevSub}\, x\, y \to \\
\qquad\quad \mathbf{case}(x, x'.\mathit{RevSub}\, x'\, (\texttt{a}_1\, y), x'.\mathit{RevSub}\, x'\, (\texttt{a}_2\, y), y)\}
\end{array}$$

$\mathcal{T}$ computes the reverse of (the tree representation of) a string over $\{\texttt{a}_1, \texttt{a}_2\}$. If $u_1, \ldots, u_n \in \{\texttt{a}_1, \texttt{a}_2\}$, then $\llbracket \mathcal{T}, \mathit{Rev}, u_1(u_2 \cdots (u_{n-1}(u_n \texttt{a}_3)) \cdots)) \rrbracket$ is:

$$u_n(u_{n-1}(\cdots(u_2(u_1 \texttt{a}_3)) \cdots)).$$

Here, the terminal symbol $\texttt{a}_3$ is used to denote the end of a string. For example, $(\mathit{Rev}\, x, \{x \mapsto (\texttt{a}_1(\texttt{a}_2 \texttt{a}_3), 0)\})$ is reduced as follows:

$$\begin{array}{l}
(\mathit{Rev}\, x, \rho = \{x \mapsto (\texttt{a}_1(\texttt{a}_2\texttt{a}_3), 0)\}) \\
\longrightarrow_{\mathcal{T}} (\mathit{RevSub}\, x\, \texttt{a}_3, \rho) \\
\longrightarrow_{\mathcal{T}} (\mathbf{case}(x, \\
\qquad\qquad x'.\mathit{RevSub}\, x'\, (\texttt{a}_1\texttt{a}_3), x'.\mathit{RevSub}\, x'\, (\texttt{a}_2\texttt{a}_3), \texttt{a}_3)), \rho) \\
\longrightarrow_{\mathcal{T}} (\mathit{RevSub}\, x''\, (\texttt{a}_1\texttt{a}_3),\{x \mapsto (\texttt{a}_1(\texttt{a}_2\texttt{a}_3),1), x'' \mapsto (\texttt{a}_2\texttt{a}_3,0)\}) \\
\longrightarrow^{*}_{\mathcal{T}} (\texttt{a}_2(\texttt{a}_1\texttt{a}_3), \rho') \qquad \square
\end{array}$$

EXAMPLE 2.2. Recall the `merge` function in Section 1. It is expressed as the following HMTT $\mathcal{T} = (\Sigma, \mathcal{N}, \mathcal{R}, \mathit{Merge})$.

$$\begin{array}{l}
\Sigma = \{\texttt{a} \mapsto 1, \texttt{b} \mapsto 1, \texttt{e} \mapsto 0\} \\
\mathcal{R} = \{\mathit{Merge}\ x\ y \to \\
\qquad\qquad \mathbf{case}(x, x'.\mathit{Merge}_a\, x'\, y, x'.\mathit{Merge}_b\, x'\, y, \mathit{Copy}\, y) \\
\mathit{Merge}_a\ x\ y \to \mathbf{case}(y, \\
\quad y'.\texttt{a}(\texttt{a}(\mathit{Merge}\, x\, y')), y'.\texttt{a}(\mathit{Merge}_b\, y'\, x), \texttt{a}(\mathit{Copy}\, x)), \\
\mathit{Merge}_b\ x\ y \to \mathbf{case}(y, \\
\quad y'.\texttt{b}(\mathit{Merge}_a\ y'\ x), y'.\texttt{b}(\texttt{b}(\mathit{Merge}\ x\ y')), \texttt{b}(\mathit{Copy}\ x)), \\
\mathit{Copy}\ x \to \mathbf{case}(x, x'.\texttt{a}(\mathit{Copy}\ x'), x'.\texttt{b}(\mathit{Copy}\ x'), \texttt{e}), \} \ \square
\end{array}$$

We introduce an important subclass of HMTTs.

DEFINITION 2.3 (linear HMTT). *An HMTT $\mathcal{T} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ is* linear *if, for all Σ-labeled ranked trees $I_1, \ldots, I_k$ (where k is the arity of S) and for all ρ and t, $(S\, x_1 \cdots x_k, \{x_1 \mapsto (I_1, 0), \ldots, x_k \mapsto (I_k, 0)\}) \longrightarrow^{*}_{\mathcal{T}} (t, \rho)$ and $\rho(y) = (I, j)$ imply $j \leq 1$, for every $y \in dom(\rho)$.*

Note that the linearity is a semantic condition. It is possible to construct a sound (but incomplete) type system for guaranteeing the linearity of HMTTs, but the semantic condition is sufficient (and actually more convenient) for our purpose. For order-1 case, the linearity condition is almost the same as the syntactic condition of linearity (or, 1-bounded copying) introduced by Maneth et al. [26]. For higher-order cases, one can define a linear type system for (conservatively) guaranteeing the linearity condition.

REMARK 2.2. The reader may think that the linearity condition is too restrictive. Actually, however, the class of tree transformations expressed by linear HMTTs is at least as large as those expressed by variations of deterministic macro/high-level tree transducers studied in the literature. That follows from the following facts. First, to our knowledge, all the variations of macro tree transducers (including pebble tree transducers [27] and stay macro tree transducers [26]) studied in the literature can be expressed by compositions of macro tree transducers (see [6] for the case of pebble tree transducers). Secondly, by the result of Engelfriet [8], any composition of deterministic, total macro tree transducers can be expressed by a single deterministic high-level tree transducer. Third, as discussed in Appendix A, any deterministic high-level tree transducer can be translated to a linear HMTT by using the tupling transformation [13].

We use a variant of tree automaton called a *trivial automaton* [2, 20] for describing properties on (possibly infinite) input and output trees of HMTT.

DEFINITION 2.4 (trivial automaton). A *Büchi automaton with a trivial acceptance condition* (a *trivial automaton*, for short) $\mathcal{M}$ is a quadruple:

$$(\Sigma, Q, \Delta, q_0)$$

where Σ is a ranked alphabet, Q is a set of states, Δ, called a transition function, is a finite subset of $Q \times dom(\Sigma) \times Q^{*}$ such that if $(q, a, q_1 \cdots q_k) \in \Delta$, then $k = arity(a)$. A $dom(\Sigma)$-labeled tree T is *accepted* by $\mathcal{M}$ if there is a Q-labeled tree R such that (i) $dom(T) = dom(R)$; (ii) For every $x \in dom(R)$, $(R(x), T(x), R(x1) \cdots R(xm)) \in \Delta$ where $m = arity(T(x))$. R is called a *run tree* of $\mathcal{M}$ over T. A trivial automaton is *deterministic* if the set $\{s \mid (q, a, s) \in \Delta\}$ is empty or a singleton set for every $q \in Q$ and $a \in dom(\Sigma)$.

(Assuming that $\perp$ does not occur in the alphabet of $\mathcal{M}$) we write $\mathcal{M}^{\perp}$ for the trivial automaton obtained from $\mathcal{M}$ by adding the special symbol $\perp$ (of arity 0) to the alphabet, and replacing the transition relation Δ with $\Delta \cup \{(q, \perp, \epsilon) | q \in Q\}$.

EXAMPLE 2.3. Consider a trivial automaton $\mathcal{M} = (\Sigma, \{q_0, q_1\}, \Delta, q_0)$ where

$$\begin{aligned}&\Sigma = \{a_1 \mapsto 2, a_2 \mapsto 1, a_3 \mapsto 0\}\\ &\Delta = \{(q_0, a_1, q_0q_0), (q_0, a_2, q_1q_1),\\ &\qquad (q_0, a_2, q_1), (q_1, a_2, q_1), (q_0, a_3, \epsilon), (q_1, a_3, \epsilon)\}\end{aligned}$$

$\mathcal{M}$ accepts Σ-ranked trees whose paths are labeled by elements of $a_1^*a_2^*a_3 + a_1^*a_2^\omega + a_1^\omega$. □

We now define the HMTT verification problem, which is the main subject of the rest of this paper.

DEFINITION 2.5 (HMTT verification problem). *Let $\mathcal{T}$ be an HMTT with $\mathcal{N}(S) = k$. We write $\models (\mathcal{T}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})$ if $[\![\mathcal{T}, S, I_1, \ldots, I_k]\!] \in \mathcal{L}(\mathcal{M}^\perp)$ holds for every $I_1 \in \mathcal{L}(\mathcal{M}_1), \ldots, I_k \in \mathcal{L}(\mathcal{M}_k)$. An* HMTT verification problem *$(\mathcal{T}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})$ is the problem of deciding whether $\models (\mathcal{T}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})$ holds.*

The problem of model-checking higher-order recursion schemes [31] (where properties are restricted to those described by trivial automata) is a special case of the HMTT verification problem (where $k = 0$).

REMARK 2.3. Note that the above definition allows $\perp$ to occur in $[\![\mathcal{T}, S, I_1, \ldots, I_k]\!]$. Given finite trees as input, an HMTT may produce $\perp$ when it does not terminate. Thus, the above problem is slightly different from the problem of exact type checking of tree transducers studied in the literature. They usually check that, given an input tree in $\mathcal{L}(\mathcal{M}_1)$, a transducer does terminate and produces a tree in $\mathcal{L}(\mathcal{M})$, while our definition allows the case where the program does not terminate. This difference is analogous to partial vs total correctness in program verification. We consider only partial correctness, leaving the termination verification as a separate problem.

3. Recursion Schemes with Finite Data Domains

In this section, we introduce an extension of higher-order recursion schemes with finite data domains (RSFD, for short), and define a model-checking problem for RSFD, into which the HMTT verification problem in the previous section will be transformed.

DEFINITION 3.1 (sorts for RSFD). *The set of (RSFD) sorts is given by:*

$$\kappa ::= \mathtt{d} \mid \mathtt{o} \mid \kappa_1 \to \kappa_2$$

DEFINITION 3.2. *A higher-order recursion scheme with finite data domain (RSFD for short) $\mathcal{G}$ is a quintuple $(\Sigma, \mathcal{N}, D, \mathcal{R}, S)$, where:*

- *Σ is a* ranked alphabet.
- *$\mathcal{N}$ is a map from a finite set of symbols called* non-terminals *to sorts.*
- *D is a finite set $\{d_1, \ldots, d_l\}$.*
- *$\mathcal{R}$ is a set of rewriting rules of the form $F\,x_1\,\cdots\,x_n \to t$, where $F \in dom(\mathcal{N})$ is a non-terminal and t is a term. Here, the set of terms is defined by:*

$$t ::= a \mid d_i \mid x \mid F \mid t_1t_2 \mid \mathbf{case}(t, t_1, \ldots, t_l)$$

The sort assignment rules for terms are the same as those of HMTT, except the following rules.

$$\mathcal{K} \vdash d : \mathtt{d}$$

$$\frac{\mathcal{K} \vdash t : \mathtt{d} \qquad \mathcal{K} \vdash t_i : \mathtt{o}\ \textit{(for each i)}}{\mathcal{K} \vdash \mathbf{case}(t, t_1, \ldots, t_l) : \mathtt{o}}$$

- *S is a non-terminal called the* start symbol*, with $\mathcal{N}(S) = \mathtt{o}$.*

The rewriting relation $t \longrightarrow_{\mathcal{G}} t'$ is defined by:

$$\frac{F\,x_1\,\cdots\,x_m \to t \in \mathcal{R}}{E[F\,t_1\,\cdots\,t_m] \longrightarrow_{\mathcal{G}} E[[t_1/x_1, \ldots, t_m/x_m]t]}$$

$$E[\mathbf{case}(d_i, t_1, \ldots, t_l)] \longrightarrow_{\mathcal{G}} E[t_i]$$

Here, E denotes an evaluation context, whose syntax is given by:

$$E ::= [\,] \mid a\,t_1\,\cdots\,t_{j-1}\,[\,]\,t_{j+1}\,\cdots\,t_{\Sigma(a)}$$

The value tree of $\mathcal{G}$, written by $[\![\mathcal{G}]\!]$, is the $(\Sigma \cup \{\perp \mapsto 1\})$-labelled ranked tree obtained by infinitary rewriting of S, i.e. $\bigsqcup\{t'^{\perp} \mid S \longrightarrow_{\mathcal{G}}^* t'\}$.

EXAMPLE 3.1. Consider the following RSFD $\mathcal{G} = (\{\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 1\}, \mathcal{N}, \{d_1, d_2\}, \mathcal{R}, S)$, where:

$$\begin{aligned}&\mathcal{N} = \{S : \mathtt{o}, F : \mathtt{d} \to \mathtt{o}\}\\ &\mathcal{R} = \{S \to F\,d_1,\\ &\qquad F\,x \to \mathbf{case}(x, \mathtt{a}(F\,d_2), \mathtt{b}(F\,d_1))\}\end{aligned}$$

The value tree $[\![\mathcal{G}]\!]$ is the infinite tree $\mathtt{a}(\mathtt{b}(\mathtt{a}(\mathtt{b}(\cdots))))$.

DEFINITION 3.3 (RSFD model checking problem). *Let $\mathcal{G}$ be an RSFD and $\mathcal{M}$ be a trivial automaton. The RSFD model checking problem $(\mathcal{G}, \mathcal{M})$ is the problem of deciding whether $[\![\mathcal{G}]\!] \in \mathcal{L}(\mathcal{M}^\perp)$ holds. We write $\models (\mathcal{G}, \mathcal{M})$ if $[\![\mathcal{G}]\!] \in \mathcal{L}(\mathcal{M}^\perp)$ holds.*

The RSFD model checking problem is decidable: as sketched in [20], any recursion scheme with finite data domains can be encoded into an ordinary recursion scheme, and the model checking problem for ordinary recursion schemes is decidable. Instead, it is also possible to encode an element d_i of a finite base type $\{d_1, \ldots, d_k\}$ into a function of sort $\underbrace{\mathtt{o} \to \cdots \to \mathtt{o}}_{k} \to \mathtt{o}$: $\lambda x_1. \cdots .\lambda x_k.x_i$. Then, $\mathbf{case}(x, t_1, \ldots, t_k)$ can be encoded into $x\,t_1\,\cdots\,t_k$. In Section 5, we given an alternative, direct proof of the decidability, which reduces the RSFD model checking problem into a type checking problem.

REMARK 3.1. *Unlike the original definition of the model checking problem for recursion schemes [31], we here allow $[\![\mathcal{G}]\!]$ to generate a tree containing $\perp$ for a technical convenience. The problem of checking $[\![\mathcal{G}]\!] \in \mathcal{L}(\mathcal{M})$ is also decidable.*

4. From HMTT Verification to RSFD Model-Checking

This section reduces the HMTT verification problem to the model-checking problem for RSFD. The reduction is sound but incomplete in general: For any HMTT verification problem P_1, if the corresponding model checking problem P_2 for RSFD has a positive answer, then so does P_1, but not vice versa. For *linear* HMTTs, however, the reduction is sound and complete.

As mentioned in Section 1, the idea of the transformation is to use the state of a trivial automaton as an abstraction of an input tree (of sort $\mathtt{i}$). Let $(\mathcal{T}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})$ be an HMTT verification problem. One can construct a trivial automaton $\mathcal{M}_{1,\ldots,k} = (\Sigma_I, Q_I, \Delta_I, q_1)$ such that $\mathcal{L}(\mathcal{M}_{1,\ldots,k}, q_i) = \mathcal{L}(\mathcal{M}_i)$. Here, $\mathcal{L}(\mathcal{M}_{1,\ldots,k}, q_i)$ is the set of trees accepted by $(\Sigma_I, Q_I, \Delta_I, q_i)$. Let Q_I be $\{q_1, \ldots, q_l\}$. We also assume that for every $q \in Q_I$, $\mathcal{L}(\mathcal{M}_{1,\ldots,k}, q) \neq \emptyset$, i.e. there is no garbage state (from which no tree can be accepted).

For an HMTT verification problem $(\mathcal{T}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})$ where $\mathcal{T} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ and $\mathcal{M} = (\Sigma, Q, \Delta, q_0)$, we write

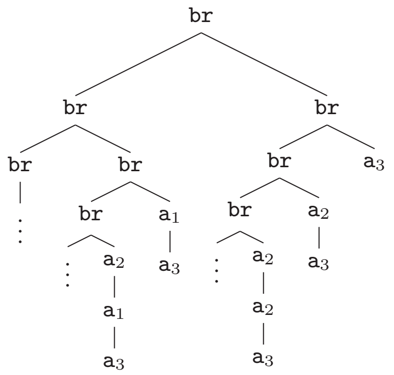

Figure 3. The tree generated by $\mathcal{G}$ of Example 4.1

$(\mathcal{T}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})^\#$ for the pair $(\mathcal{G}, \mathcal{M}')$, where:

$$\begin{array}{l}
\mathcal{G} = (\Sigma \,\cup \{\mathtt{br} \mapsto 2\}, \mathcal{N}^\# \cup \{S' \mapsto \mathtt{o}\}, Q_I, \mathcal{R}', S') \\
\mathcal{R}' = \{S' \to S\, q_1 \,\cdots\, q_k\} \\
\qquad \cup\{F\,\widetilde{x} \to t^\# \mid F\,\widetilde{x} \to t \in \mathcal{R}\} \\
\mathcal{M}' = (\Sigma \cup \{\mathtt{br} \mapsto 2\}, Q, \Delta', q_0) \\
\Delta' = \Delta \cup \{(q, \mathtt{br}, qq) \mid q \in Q\}.
\end{array}$$

Here, $\mathcal{N}^\#$ just replaces each occurrence of sort $\mathtt{i}$ in $\mathcal{N}$ with $\mathtt{d}$. The translation $t^\#$ of a term t is defined by:

$$\begin{array}{l}
a^\# = a \qquad x^\# = x \qquad F^\# = F \qquad (t_1\, t_2)^\# = t_1{}^\#\, t_2{}^\# \\
\mathbf{case}(x, \widetilde{y}_1.t_1, \ldots, \widetilde{y}_N.t_N)^\# = \mathbf{case}(x, u'_1, \ldots, u'_l) \\
\text{where } u'_i = \mathtt{br}\, ([\widetilde{q}_{1,1}/\widetilde{y}_1]t_1{}^\#) \,\cdots\, ([\widetilde{q}_{1,k_{i,1}}/\widetilde{y}_1]t_1{}^\#) \\
\qquad\qquad \cdots\, ([\widetilde{q}_{N,1}/\widetilde{y}_N]t_N{}^\#) \,\cdots\, ([\widetilde{q}_{N,k_{i,N}}/\widetilde{y}_N]t_N{}^\#) \\
\qquad \Delta(q_i, a_j) = \{\widetilde{q}_{j,1}, \ldots, \widetilde{q}_{j,k_{i,j}}\} \\
\text{(Here } \Delta(q, a) \text{ denotes } \{\widetilde{q} \mid (q, a, \widetilde{q}) \in \Delta\})
\end{array}$$

In the definition above, $\mathtt{br}\, t_1 \,\cdots\, t_n$ is an abbreviated form of:

$$\mathtt{br}\, t_1\, (\mathtt{br}\, t_2\, (\mathtt{br} \cdots (\mathtt{br}\, t_{n-1}\, t_n) \cdots))$$

Note that case analysis on an input tree x is replaced by case analysis on the corresponding automaton state x. For each case $q_i \in Q$ of x, the case expression reduces to a term of the form $\mathtt{br}\, t_1 \,\cdots\, t_m$, where $t_1, \ldots, t_m$ are reducts for all the possible trees abstracted by q_i.

EXAMPLE 4.1. Recall Example 2.1. Let $\mathcal{M}_1$ and $\mathcal{M}$ be trivial automata $(\Sigma, Q, \Delta_1, q_1)$ and (Σ, Q, Δ, q_1) where:

$$\begin{array}{l}
\Sigma = \{a_1 \mapsto 1, a_2 \mapsto 1, a_3 \mapsto 0\} \\
Q = \{q_1, q_2\} \\
\Delta_1 = \{(q_1,a_1,q_1), (q_1,a_2,q_2), (q_2,a_2,q_2), (q_1,a_3,\epsilon), (q_2,a_3,\epsilon)\} \\
\Delta = \{(q_1,a_2,q_1), (q_1,a_1,q_2), (q_2,a_1,q_2), (q_1,a_3,\epsilon), (q_2,a_3,\epsilon)\}
\end{array}$$

Then, $(\mathcal{T}, \mathcal{M}_1, \mathcal{M}_2)^\# = (\mathcal{G}, \mathcal{M}')$ where

$$\begin{array}{l}
\mathcal{G} = (\Sigma \cup \{\mathtt{br}\}, \mathcal{N}, \{q_1, q_2\}, \mathcal{R}, S') \\
\mathcal{R} = \{S' \to S\, q_1, \quad S\, x \to \mathit{RevSub}\, x\, \mathtt{a}_3, \\
\qquad \mathit{RevSub}\, x\, y \to \mathbf{case}(x, t_1, t_2)\} \\
t_1 = \mathtt{br}\, (\mathit{RevSub}\, q_1\, (\mathtt{a}_1\, y))\, (\mathtt{br}\, (\mathit{RevSub}\, q_2\, (\mathtt{a}_2\, y))\, y) \\
t_2 = \mathtt{br}\, (\mathit{RevSub}\, q_2\, (\mathtt{a}_2\, y))\, y \\
\mathcal{M}' = (\Sigma \cup \{\mathtt{br} \mapsto 2\}, \{q_1, q_2\}, \Delta, q_1) \\
\Delta = \Delta \cup \{(q_1, \mathtt{br}, q_1 q_1), (q_2, \mathtt{br}, q_2 q_2)\}
\end{array}$$

Note that input trees have been replaced by states of $\mathcal{M}_1$, and case analyses on an input tree have been replaced by case analyses on the corresponding state of $\mathcal{M}_1$. The value tree of $\mathcal{G}$ is illustrated in Figure 3. □

We now discuss the correctness of the transformation. The following theorem guarantees that (the first element of) the output of the transformation is indeed a recursion scheme.

THEOREM 4.1 (well-formedness of the recursion scheme). *Let $(\mathcal{T}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})$ be an HMTT verification problem. If $(\mathcal{T}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})^\# = (\mathcal{G}, \mathcal{M}')$, then $\mathcal{G}$ is a recursion scheme.*

Proof It suffices to show that each rule of $\mathcal{G}$ is well-sorted. We define a translation of sorts of HMTT into those of RSFD by:

$$\mathtt{i}^\# = \mathtt{d} \quad \mathtt{o}^\# = \mathtt{o} \quad (\kappa_1 \to \kappa_2)^\# = \kappa_1{}^\# \to \kappa_2{}^\#$$

The translation $(\cdot)^\#$ is pointwise extended to sort environments. We can show by induction on the derivation that $\mathcal{K} \vdash t : \kappa$ implies $\mathcal{K}^\# \vdash t^\# : \kappa^\#$. Thus, any well-sorted rule of $\mathcal{T}$ is transformed into a well-sorted rule of $\mathcal{G}$. □

The following theorem guarantees the soundness of our transformation.

THEOREM 4.2 (soundness). *Let $(\mathcal{T}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})$ be an HMTT verification problem. If $\models (\mathcal{T}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})^\#$, then $\models (\mathcal{T}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})$.*

Proof Sketch Let $(\mathcal{G}, \mathcal{M}')$ be $(\mathcal{T}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})^\#$. We first note that by the definitions of $[\![\mathcal{G}]\!]$ and $[\![\mathcal{T}, S, I_1, \ldots, I_k]\!]$, we have: (I) $[\![\mathcal{G}]\!] \in \mathcal{L}(\mathcal{M}'^\perp)$ if, and only if, $u^\perp \in \mathcal{L}(\mathcal{M}'^\perp)$ for every u such that $S' \longrightarrow^*_\mathcal{G} u$, and (II) $[\![\mathcal{T}, S, I_1, \ldots, I_k]\!] \in \mathcal{L}(\mathcal{M}^\perp)$ if, and only if, $t^\perp \in \mathcal{L}(\mathcal{M}^\perp)$ for every t such that $\exists \rho.((S\, x_1 \,\cdots\, x_k, \rho_0) \longrightarrow^*_\mathcal{T} (t, \rho))$, where $\rho_0 = \{x_1 \mapsto (I_1, 0), \ldots, x_k \mapsto (I_k, 0)\}$.

We define the relation $u \rightsquigarrow u'$ on RSFD terms of sort $\mathtt{o}$ inductively by: (i) $\mathtt{br}\, u_1 \cdots u_n \rightsquigarrow u'_i$ if $u_i \rightsquigarrow u'_i$, (ii) $a u_1 \cdots u_n \rightsquigarrow a u'_1 \cdots u'_n$ if $u_i \rightsquigarrow u'_i$ for each $i \in \{1, \ldots, n\}$, (iii)$F\,\widetilde{u} \rightsquigarrow F\,\widetilde{u}$, and (iv) $\mathbf{case}(u, u_1, \ldots, u_l) \rightsquigarrow \mathbf{case}(u, u_1, \ldots, u_l)$. Intuitively, an RSFD term u represents a set of terms (where $\mathtt{br}$ denotes a union), and $u \rightsquigarrow u'$ means that u' is an element of the set represented by u.

Suppose that $\models (\mathcal{G}, \mathcal{M}')$ and $(S\, x_1 \,\cdots\, x_k, \rho_0) \longrightarrow^*_\mathcal{T} (t, \rho)$ with $I_i \in \mathcal{L}(\mathcal{M}_i)$ for each $i \in \{1, \ldots, k\}$. It suffices to show that $t^\perp \in \mathcal{L}(\mathcal{M}^\perp)$. One can prove that $(S\, x_1 \,\cdots\, x_k, \rho_0) \longrightarrow^*_\mathcal{T} (t, \rho)$ implies $S' \longrightarrow^*_\mathcal{G} u \rightsquigarrow u'$ and $t^\perp = u'^\perp$ for some u and u' (which intuitively means that $\mathcal{G}$ is a correct abstraction of $\mathcal{T}$). By the assumption $\models (\mathcal{G}, \mathcal{M}')$, we have $u^\perp \in \mathcal{L}(\mathcal{M}'^\perp)$. By the construction of $\mathcal{M}'$, we have $u'^\perp \in \mathcal{L}(\mathcal{M}^\perp)$ (note that $u'^\perp$ does not contain $\mathtt{br}$), which implies $t^\perp \in \mathcal{L}(\mathcal{M}^\perp)$ as required. See [**?**] for more details. □

As shown by the following example, the converse of the above theorem does not hold in general.

EXAMPLE 4.2. Consider an HMTT $\mathcal{T} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ where:

$$\begin{array}{l}
\Sigma = \{a_1 \mapsto 0, a_2 \mapsto 0, a_3 \mapsto 2\} \\
\mathcal{N} = \{S \mapsto \mathtt{i} \to \mathtt{o}\} \\
\mathcal{R} = \{S\, x \to \mathbf{case}(x, a_3\, a_1\, (C\, x), a_2, \cdots), \\
\qquad C\, x \to \mathbf{case}(x, a_1, a_2, \cdots)\}
\end{array}$$

We have omitted the case for a_3 since it does not matter. Let $\mathcal{M}_1$ be the trivial automaton $(\Sigma, \{q_0\}, \Delta_1, q_0)$ where $\Delta_1 = \{(q_0, a_1, \epsilon), (q_0, a_2, \epsilon)\}$. Note that $\mathcal{L}(\mathcal{M}_1)$ accepts $\{a_1, a_2\}$. Let $\mathcal{M}$ be another trivial automaton $(\Sigma, \{q_0, q_1\}, \Delta, q_0)$, where $\Delta = \{(q_0, a_2, \epsilon), (q_0, a_3, q_1 q_1), (q_1, a_1, \epsilon)\}$, which accepts $\{a_3\, a_1\, a_1, a_2\}$. Obviously, $\models (\mathcal{T}, \mathcal{M}_1, \mathcal{M})$ holds. However,

$(\mathcal{T}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})^{\#}$ is $(\mathcal{G}, \mathcal{M}')$ where:

$$\begin{array}{l}
\mathcal{G} = (\Sigma, \mathcal{N}, \{q_0\}, \mathcal{R}, S') \\
\mathcal{R} = \{S' \to S\, q_0, \quad S\, x \to \mathbf{case}(x, \mathtt{br}\,(a_3\, a_1\, (C\, q_0))\, a_2), \\
\qquad C\, x \to \mathbf{case}(x, \mathtt{br}\, a_1\, a_2)\} \\
\mathcal{M}' = (\Sigma, \{q_0, q_1\}, \Delta', q_0) \\
\Delta' = \{(q_0, \mathtt{br}, q_0 q_0), (q_1, \mathtt{br}, q_1 q_1)\} \cup \Delta
\end{array}$$

$[\![\mathcal{G}]\!] = \mathtt{br}\,(a_3\, a_1\, (\mathtt{br}\, a_1\, a_2))\, a_2$ is not accepted by $\mathcal{M}'$.

The reason why the transformation above does not preserve the validity is that both of the input trees a_1 and a_2 are abstracted to q_0, and independent choices between a_1 and a_2 are made in the two case analyses on x of the recursion scheme (once in the rule for S, and the other time in the rule for C). □

If we restrict ourselves to *linear* HMTTs and make certain additional assumptions, then our transformation is complete.

THEOREM 4.3 (completeness for linear HMTT).
Let $(\mathcal{T}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})$ be an HMTT verification problem. Suppose also that one of the following conditions holds.

1. *$\mathcal{M} = (\Sigma, Q, \Delta, q_0)$ is deterministic.*
2. *If $(\mathcal{T}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})^{\#} = (\mathcal{G}, \mathcal{M}')$, the symbols $\mathtt{br}$ only occur at the top-level of $[\![\mathcal{G}]\!]$, i.e. in every path of $[\![\mathcal{G}]\!]$ from the root, $\mathtt{br}$ does not occur after other terminal symbols occur.*

If $\mathcal{T}$ is linear and $\models (\mathcal{T}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})$,
then $\models (\mathcal{T}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})^{\#}$.

Proof Sketch Suppose that $\models (\mathcal{T}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})$ and $S' \longrightarrow^*_{\mathcal{G}} u$. It suffices to show $u^{\perp} \in \mathcal{L}(\mathcal{M}'^{\perp})$ (recall the first paragraph of the proof sketch of Theorem 4.2).

By the assumption that $\mathcal{T}$ is linear, one can prove that $S' \longrightarrow^*_{\mathcal{G}} u \rightsquigarrow u'$ implies $(S\, x_1 \cdots x_k, \{x_1 \mapsto (I_1, 0), \ldots, x_k \mapsto (I_k, 0)\}) \longrightarrow^*_{\mathcal{T}} (t, \rho)$ and $u'^{\perp} = t^{\perp}$ for some t, ρ, and $I_1, \ldots, I_k$ such that $I_i \in \mathcal{L}(\mathcal{M}_i)$ for every $i \in \{1, \ldots, k\}$. (This intuitively means that $\mathcal{G}$ is an abstraction of $\mathcal{T}$ that is precise enough to capture only the reductions possible in $\mathcal{T}$.) For such t, we have $t^{\perp} \in \mathcal{L}(\mathcal{M}^{\perp})$ by the assumption $\models (\mathcal{T}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})$. By the construction of $\mathcal{M}'$ and the fact $u'^{\perp} = t^{\perp}$, we have $u'^{\perp} \in \mathcal{L}(\mathcal{M}'^{\perp})$.

Thus, we have $u'^{\perp} \in \mathcal{L}(\mathcal{M}'^{\perp})$ for every u' such that $u \rightsquigarrow u'$. We therefore have $u^{\perp} \in \mathcal{L}(\mathcal{M}'^{\perp})$ if one of the two conditions in the statement of the theorem holds.

See [**?**] for more details. □

REMARK 4.1. If neither of the two conditions in Theorem 4.3 holds, the last step of the above proof does not go through. In that case, the completeness does not hold indeed. Consider the HMTT $\mathcal{T} = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ where:

$$\begin{array}{l}
\Sigma = \{a_1 \mapsto 1, a_2 \mapsto 0, a_3 \mapsto 0\} \\
\mathcal{N} = \{S \mapsto \mathtt{i} \to \mathtt{o}\} \\
\mathcal{R} = \{S\, x \to \mathbf{case}(x, x'.a_1(S\, x'), a_2, a_3)\}
\end{array}$$

It just generates a copy of a given input tree. Let us consider (non-deterministic) automata $\mathcal{M}_1 = (\Sigma, \{q_1, q_2\}, \Delta_1, q_1)$ and $\mathcal{M} = (\Sigma, \{q_1, q_2\}, \Delta, q_1)$ where

$$\begin{array}{l}
\Delta_1 = \{(q_1, a_1, q_2), (q_2, a_2, \epsilon), (q_2, a_3, \epsilon)\} \\
\Delta = \{(q_1, a_1, q_2), (q_1, a_1, q_3), (q_2, a_2, \epsilon), (q_3, a_3, \epsilon)\}.
\end{array}$$

Since both automata accept $\{a_1\, a_2, a_1\, a_3\}$, $\models (\mathcal{T}, \mathcal{M}_1, \mathcal{M})$ holds. Our transformation algorithm generates the recursion scheme $\mathcal{G} = (\Sigma \cup \{\mathtt{br} \mapsto 2\}, \mathcal{N}, \{q_1, q_2\}, \mathcal{R}, S')$ and the automaton $\mathcal{M}' = (\Sigma \cup \{\mathtt{br} \mapsto 2\}, Q, \Delta \cup \{(q_1, \mathtt{br}, q_1, q_1), (q_2, \mathtt{br}, q_2, q_2)\}, q_1)$ where $\mathcal{R}$ consists of:

$$S' \to S\, q_1 \qquad S\, x \to \mathbf{case}(x, a_1(S\, q_2), \mathtt{br}\, a_2\, a_3)$$

It generates a finite tree $a_1(\mathtt{br}\, a_2\, a_3)$. The tree is NOT accepted by $\mathcal{M}'$: note that no state can be assigned to the subtree $\mathtt{br}\, a_2\, a_3$.

The second condition of Theorem 4.3 can be guaranteed by applying the CPS transformation, which ensures that an output tree is returned only after all the non-deterministic branches have been made. For example, the HMTT can be transformed into the following equivalent HMTT (where only the rewriting rules are shown).

$$\begin{array}{l}
S\, x \to S_1\, x\, I \qquad I\, x \to x \qquad C\, k\, x\, y \to k(x(y)) \\
S_1\, x\, k \to \mathbf{case}(x, x'.S_1\, x'\, (C\, k\, a_1), k\, a_2, k\, a_3)
\end{array}$$

It generates the tree $\mathtt{br}\,(a_1\, a_2)\,(a_1\, a_3)$, which is accepted by $\mathcal{M}'$.

The HMTT verification problem is undecidable in general, since we can encode Post correspondence problem.

THEOREM 4.4. *The HMTT verification problem is undecidable.*

Proof Let $\mathcal{A}$ be $\{a_1, ..., a_n\}$ and consider a Post correspondence problem

$$(u_1, v_1), ..., (u_m, v_m) \in \mathcal{A}^* \times \mathcal{A}^*$$

Without loss of generality, we may assume that $m = n$, since if $m < n$, we can add dummy pairs (ϵ, ϵ) as (u_k, v_k) for $m < k \leq n$. The Post correspondence problem is the problem of deciding whether there exists a sequence $i_1 \cdots i_\ell$ such that

$$u_{i_1} u_{i_2} \cdots u_{i_\ell} = v_{i_1} v_{i_2} \cdots v_{i_\ell}.$$

We shall construct an HMTT that takes a candidate of such a sequence, and checks whether the candidate satisfies the condition above. Let $\mathcal{T}$ be an HMTT $(\Sigma, \mathcal{N}, \mathcal{R}, S)$ where:

$$\begin{array}{l}
\Sigma = \{a_1 \mapsto 1, \ldots, a_n \mapsto 1, \mathtt{e} \mapsto 0, \mathtt{yes} \mapsto 0, \mathtt{no} \mapsto 0\} \\
\mathcal{N} = \{S \mapsto \mathtt{i} \to \mathtt{o}\} \\
\qquad \cup\{\mathtt{Sub}_{u,v} \mapsto \mathtt{i} \to \mathtt{i} \to \mathtt{o} \mid \\
\qquad\quad u, v \text{ are suffixes of } u_i \text{ and } v_j \text{ for some } i \text{ and } j\} \\
\qquad \cup\{\mathtt{IsNull}_u \mapsto \mathtt{i} \to \mathtt{o} \mid u \text{ is a suffix of } u_i \text{ for some } i\} \\
\mathcal{R} = \{S\, x \to \mathtt{Sub}_{\varepsilon,\varepsilon}\, x\, x\} \\
\quad \cup\{\mathtt{Sub}_{\varepsilon,s}\, x\, y \to \\
\qquad \mathbf{case}(x, x'.\mathtt{Sub}_{u_1,s}\, x'\, y, \ldots, x'.\mathtt{Sub}_{u_n,s}\, x'\, y, \mathtt{IsNull}_s\, y) \\
\qquad \mid s \in \{a_1, \ldots, a_n\}^*\} \\
\quad \cup\{\mathtt{Sub}_{s,\epsilon}\, x\, y \to \\
\qquad \mathbf{case}(y, y'.\mathtt{Sub}_{s,v_1}\, x\, y', \ldots, x.\mathtt{Sub}_{s,v_n}\, x\, y', \mathtt{IsNull}_s\, x) \\
\qquad \mid s \in \{a_1, \ldots, a_n\}^+\} \\
\quad \cup\{\mathtt{Sub}_{au',av'},\, x\, y \to \mathtt{Sub}_{u',v'},\, x\, y \mid a \in \mathcal{A}, u', v' \in \mathcal{A}^*\} \\
\quad \cup\{\mathtt{Sub}_{au',bv'},\, x\, y \to \mathtt{no} \mid a, b \in \mathcal{A}, u', v' \in \mathcal{A}^*, a \neq b\} \\
\quad \cup\{\mathtt{IsNull}_\epsilon\, x \to \mathbf{case}(x, x'.\mathtt{no}, \cdots, x'.\mathtt{no}, \mathtt{yes})\} \\
\quad \cup\{\mathtt{IsNull}_s\, x \to \mathtt{no} \mid s \in \mathcal{A}^+\}
\end{array}$$

Here, we assume that $\mathtt{yes}$ and $\mathtt{no}$ do not occur in input trees, and omit cases for $\mathtt{yes}$ and $\mathtt{no}$ in case expressions.

The above HMTT takes as input (the tree representation of) a sequence $a_{i_1} a_{i_2} \cdots a_{i_\ell} \in \mathcal{A}^*$, and checks whether the strings $u_{i_1} u_{i_2} \cdots u_{i_\ell}$ and $v_{i_1} v_{i_2} \cdots v_{i_\ell}$ are the same, by comparing their elements one by one. In $\mathtt{Sub}_{u,v},\, x\, y$, x and y are bound to suffixes of $a_{i_1} a_{i_2} \cdots a_{i_\ell}$, and it is checked whether $u u_{i_{\ell+1-|x|}} \cdots u_{i_\ell}$ and $v v_{i_{\ell+1-|y|}} \cdots v_{i_\ell}$ are the same. Thus, the HMTT outputs $\mathtt{yes}$ if the sequence is a solution for the Post correspondence problem and $\mathtt{no}$ otherwise. Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be trivial automata:

$$\begin{array}{l}
\mathcal{M}_1 = (\Sigma, \{q_0\}, \Delta_1, q_0) \\
\Delta_1 = \{(q_0, a_i, q_0) \mid a_i \in \mathcal{A}\} \cup \{(q_0, b, \epsilon)\} \\
\mathcal{M}_2 = (\Sigma, \{q_0\}, \Delta_2, q_0) \\
\Delta_2 = \{(q_0, \mathtt{no}, \epsilon)\}
\end{array}$$

Then, $\models (\mathcal{T}, \mathcal{M}_1, \mathcal{M}_2)$ if and only if the Post correspondence problem $\{(u_1, v_1), \ldots, (u_n, v_n)\}$ has no solution. Since Post correspondence problem is undecidable, the HMTT verification problem is also undecidable. □

5. Type System for Model-Checking Recursion Schemes with Finite Data Domains

This section gives a type system for recursion schemes with finite data domains (which is parameterized by a trivial automaton $\mathcal{M}$), such that an RSFD $\mathcal{G}$ is well-typed in the type system if, and only if, $\models (\mathcal{G}, \mathcal{M})$ holds. Thus, the RSFD model checking problem is reduced to a type checking problem, which can be solved by extending Kobayashi's type inference algorithms [18, 20].

Let $\mathcal{M}$ be a trivial automaton (Σ, Q, Δ, q_0), and D be a finite set $\{d_1, \ldots, d_k\}$. The set of types is given by:

$$\tau ::= d_i \mid q_j \mid \textstyle\bigwedge_{i=1}^{n} \tau_i \to \tau$$

Intuitively, $d_i \in D$ is a singleton type, describing the value d_i. The type $q_j \in Q$ describes trees accepted from q_j (i.e. elements of $\mathcal{L}(\mathcal{M}, q_j)$). The type $\bigwedge_{i=1}^{n} \tau_n \to \tau$ describes functions that take an element having types $\tau_1, \ldots, \tau_n$ and return an element of type τ.

A type judgment for terms (where a non-terminal is treated as a variable) is of the form $\Gamma \vdash_{\mathcal{M}} t : \tau$, where Γ, called a *type environment*, is a finite set of bindings of the form $x : \tau$. Γ may contain more than one bindings for each variable.

The typing rules are given by:

$$\frac{}{\Gamma, x : \tau \vdash_{\mathcal{M}} x : \tau}$$

$$\frac{}{\Gamma \vdash_{\mathcal{M}} d_i : d_i}$$

$$\frac{(q, a, q_1 \cdots q_n) \in \Delta}{\Gamma \vdash_{\mathcal{M}} a : q_1 \to \cdots \to q_n \to q}$$

$$\frac{\Gamma \vdash_{\mathcal{M}} t_0 : \bigwedge_{i=1}^{n} \tau_i \to \tau \qquad \Gamma \vdash_{\mathcal{M}} t_1 : \tau_i \text{ (for each } i = 1, \ldots, n)}{\Gamma \vdash_{\mathcal{M}} t_0 t_1 : \tau}$$

$$\frac{\Gamma \vdash_{\mathcal{M}} t : d_i \qquad \Gamma \vdash_{\mathcal{M}} t_i : q \quad \text{(for some } i)}{\Gamma \vdash_{\mathcal{M}} \mathbf{case}(t, t_1, \ldots, t_k) : q}$$

$$\frac{\Gamma, x : \tau_1, \ldots, x : \tau_n \vdash_{\mathcal{M}} t : \tau \qquad x \text{ not occur in } \Gamma}{\Gamma \vdash_{\mathcal{M}} \lambda x.t : \bigwedge_{i=1}^{n} \tau_i \to \tau}$$

Let $\mathcal{G}$ be a recursion scheme with finite domains $(\Sigma, \mathcal{N}, D, \mathcal{R}, S)$. We write $\vdash_{\mathcal{M}} \mathcal{G} : \Gamma$ if $\Gamma \vdash \mathcal{R}(F) : \tau$ holds for every $F : \tau \in \Gamma$. A recursion scheme $\mathcal{G}$ is well-typed, written $\vdash_{\mathcal{M}} \mathcal{G}$, just if there exists Γ such that (i) $\vdash_{\mathcal{M}} \mathcal{G} : \Gamma$, (ii)$S : q_0 \in \Gamma$, and (iii) for each $F : \tau \in \Gamma$, $\tau :: \mathcal{N}(F)$ holds (i.e. the sorts declared in $\mathcal{N}$ are respected). Here, the relation $\tau :: \kappa$, which means that τ is a type of sort κ, is defined by: (i) $q :: \mathtt{o}$, (ii) $q :: \mathtt{d}$, and (iii) $\bigwedge_{i=1}^{k} \tau_i \to \tau :: \kappa' \to \kappa$ if $\tau :: \kappa$ and $\tau_i :: \kappa'$ for each $i \in \{1, \ldots, k\}$.

The following theorem is an extension of the result of Kobayashi [20]. The proof is almost the same as that of the soundness and completeness of the type system for recursion schemes (without finite data domains) [20], hence is omitted.

Theorem 5.1. *Let $\mathcal{M}$ be a trivial automaton, and $\mathcal{G}$ be a recursion scheme with finite domains, If $[\![\mathcal{G}]\!]$ is well-defined, then $\vdash_{\mathcal{M}} \mathcal{G}$ if and only if $\models (\mathcal{G}, \mathcal{M})$.*

Corollary 5.2. *The RSFD model-checking problem is decidable.*

6. Complexity of the Verification Algorithm

We briefly discuss the complexity of our HMTT verification algorithm, which consists of two phases: transformation into a recursion scheme (with finite data domains) and model checking of the recursion scheme.

Let $(\mathcal{T}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})$ be an HMTT verification problem. We may assume that $\mathcal{M}_i = (\Sigma, Q_0, \Delta_0, q_i)$ for each i (i.e. $\mathcal{M}_1, \ldots, \mathcal{M}_k$ differ only in their initial states). Let $|\mathcal{T}|$ be the size of the rewriting rules of $\mathcal{T}$. We also assume that the rewriting rules of $\mathcal{T}$ are normalized, so that each body of a rewriting rule contains at most one case-expression. Let $(\mathcal{T}, \mathcal{M}_1, \ldots, \mathcal{M}_k, \mathcal{M})^{\#}$ be $(\mathcal{G}, \mathcal{M}')$. The size $|\mathcal{G}|$ of the rewriting rules of $\mathcal{G}$ is $O(|Q_0|^{A_\Sigma + 1}|\mathcal{T}|)$ where A_Σ is the largest arity of terminal symbols: note that only the sizes of case expressions may increase, and for each body of a case expression, at most $|\{(q, \widetilde{q}) \mid (q, a, \widetilde{q}) \in \Delta\}|$ (which are bounded by $|Q|^{A+1}$) copies are created.

As for the time complexity of model-checking $\mathcal{G}$, we can apply the same argument as [20, 22] to obtain an upper-bound $O(|\mathcal{G}|\mathbf{exp}_n((A(|Q_0| + |Q'|))^{1+\epsilon}))$ for arbitrary $\epsilon > 0$ for $n \geq 2$, where n is the order of the recursion scheme $\mathcal{G}$ and $|Q'|$ is the number of states of $\mathcal{M}'$, and A is the largest arity of symbols in $\mathcal{G}$. Here, $\mathbf{exp}_n(x)$ is defined by: $\mathbf{exp}_0(x) = x$ and $\mathbf{exp}_{n+1}(x) = 2^{\mathbf{exp}_n(x)}$.

Thus, the time complexity of our HMTT verification algorithm is $O(|\mathcal{T}|\mathbf{exp}_n((A(|Q_0| + |Q|))^{1+\epsilon}))$ for $n \geq 2$, where $|Q|$ is the number of states of $\mathcal{M}$. If the largest arity and the number of automaton states are fixed, it is linear in the size of HMTT. (Note, however, that the constant factor is huge.)

Theorem 4.3 implies that if the HMTT is linear and the automaton $\mathcal{M}$ is deterministic, then the HMTT verification problem is decidable. For this decidable fragment, the verification problem itself is $(n-1)$-EXPTIME complete for $n \geq 2$ (in the combined size of a recursion scheme and an automaton), as described below. First, from a deterministic trivial automaton $\mathcal{M}$, one can construct a disjunctive alternating parity tree automaton (disjunctive APT) [21] that accepts the complement of $\mathcal{L}(\mathcal{M})$ and the size of the disjunctive APT is linear in that of $\mathcal{M}$. Thus, the model checking problem for the class of deterministic trivial automaton can be reduced to that for the class of disjunctive APT [21], which gives the upper-bound of $(n-1)$-EXPTIME. Secondly, it follows from the result of [21] (more precisely, from the $(n-1)$-EXPTIME hardness of the reachability problem) that the problem of model-checking recursion schemes for the class of *deterministic* trivial automata is $(n-1)$-EXPTIME hard. For $n = 1$, by a similar argument, if we assume that the largest arity of terminals and non-terminals is fixed, the HMTT verification problem is polynomial-time.[2]

7. Applications

This section discusses applications of our HMTT verification framework.

7.1 Resource Usage Verification

As mentioned in Section 1, the goal of resource usage verification [14] is to check whether a given program accesses each resource in a valid manner. In our previous work [20], we considered

[2] This result is similar to, but should not be confused with the result of [26], which shows that the exact type checking of linear MTT is polynomial time under a similar assumption about arities and the assumption that the specification is given by a deterministic bottom-up tree automaton. The class of languages of finite trees accepted by deterministic trivial automata is strictly less expressive than the class of those accepted by deterministic bottom-up tree automata. Thus, our result is weaker in this sense. On the other hand, we allow multiple input trees as arguments of non-terminals.

resource usage verification of a closed program. Our HMTT verification framework allows us to perform resource usage verification of an open program, which is a function that takes some parameters as input.

For example, consider the following program:

```
let rec accfile cmds =  match cmds with
      [] -> close(fp)
    | r::cmds' -> read(fp); accfile cmds' fp
    | w::cmds' -> write(fp); accfile cmds' fp
```

It takes a list of "commands" consisting of `r` and `w`, and accesses the resource x according to the list of commands, and then closes it.

We can covert the above program into the following HMTT $\mathcal{T}$ (only the rewriting rules are shown):

```
AccFile cmds = case cmds of e => close
           | r(cmds') => read(AccFile cmds')
           | w(cmds') => write(AccFile cmds').
```

For the sake of readability, throughout this section, we use ordinary pattern matching constructs, which can be easily translated into case expressions of HMTT. We also write `=` instead of `->`. To verify that if the list of commands only consists of `r`, then the file pointer `fp` is only read and then closed, it suffices to consider the HMTT verification problem $(\mathcal{T}, \mathcal{M}_1, \mathcal{M})$, where $\mathcal{M}_1$ accepts the language consisting of linear trees labeled by elements of $r^*(e) + r^\omega$ and $\mathcal{M}$ accepts the language consisting of linear trees labeled by elements of $\mathtt{read}^*(\mathtt{close}) + \mathtt{read}^\omega$.

7.2 Verification of XML-Processing Programs

Another application domain of our HMTT verification algorithm is exact type checking of XML-processing programs [24, 26, 27, 33].

We assume below that XML documents (which are unranked trees) are represented as binary trees in the standard manner: the left and right children of a node in the binary tree representation stand for the leftmost child and the leftmost sibling respectively.

An obvious advantage of our approach over previous work based on macro/high-level tree transducers is that we can handle programs that take multiple XML documents. Let $\Sigma = \{\mathtt{addr} \mapsto 2, \mathtt{doc} \mapsto 1, \mathtt{e}, \mathtt{pc} \mapsto 0\}$ and $\mathcal{M} = (\Sigma, \{q_0, q_1, q_2\}, \Delta, q_0)$ where

$$\Delta = \{(q_0, \mathtt{doc}, q_1), (q_1, \mathtt{addr}, q_2 q_1), (q_2, \mathtt{pc}, \epsilon), (q_1, \mathtt{e}, \epsilon)\}.$$

$\mathcal{M}$ accepts (the binary tree representation of) trees of the form $\mathtt{doc}(\mathtt{addr}(\mathtt{pc})^*)$, i.e. documents containing a sequence of address data $\mathtt{addr}(\mathtt{pc})$ (the terminal `e` represents the end of a sequence). Consider the following HMTT:

```
MergeAddr x y = case x of
        doc x1 => doc(MergeAddr x1 y)
      | addr x1 x2 => addr (M x1) (MergeAddr x2 y)
      | e => M y | pc => pc.
M x = case x of doc x1 => M x1
      | addr x1 x2 => addr (M x1) (M x2)
      | e => e | pc => pc.
```

It takes two documents of the form $\mathtt{doc}(\mathtt{addr}(\mathtt{pc})^*)$ as input, and merges them into one document. One can verify that, given two valid documents (of type $\mathtt{doc}(\mathtt{addr}(\mathtt{pc})^*)$), $\mathcal{T}$ produces another valid document, by checking that $\models (\mathcal{T}, \mathcal{M}, \mathcal{M}, \mathcal{M})$ holds.

For another example, consider the following HMTT $\mathcal{T}$, which removes all the `b` nodes occurring as descendants of `a` nodes.

```
RemoveB x = F x G.
F x g = case x of doc y => doc (F y g)
          | a x y => a (g x) (F y g)
          | b x y => b (F x g) (F y g)
          | pc => pc | e => e.
G x = case x of doc y => doc (G y)
        | b x y => (G y)
        | a x y => a (G x) (G y)
        | pc => pc | e => e.
```

The non-terminal `F` represents a higher-order function, which takes a tree x and a function g, and applies g to the child of each a node. Let $\mathcal{M}_1$ be an automaton that accepts binary tree representation of trees of the form $\mathtt{doc}(t^*)$ (where `doc` does not occur in t), and $\mathcal{M}$ be an automaton that accepts only trees in which `b` does not occur below `a` (in the corresponding unranked representation). Then, it can be verified that $\models (\mathcal{T}, \mathcal{M}_1, \mathcal{M})$ holds.

7.3 String Analysis

Our HMTT verification algorithm is also applicable to string analysis, whose goal is to statically check that, given valid strings as input, a program generates a valid string. String analyses have been extensively studied in the context of Web applications, to guarantee the well-formedness of output HTMLs [28] or detect security vulnerabilities such as SQL-injection or cross-site scripting [12, 15, 37]. They are also applicable to other application domains such as static resolution of dynamically specified resource names [23, 32].

A (regular) string analysis problem can be encoded to an HMTT verification problem by representing strings as linear trees. For example, the function `replace` that takes three strings s_a, s_b and x and returns the string obtained by replacing each occurrence of a and b in x with s_a and s_b respectively, can be represented as the following HMTT:

```
Replace sa sb x = Conv sa (C1 sb x).
C1 sb x u = Conv sb (Repl x u).
Repl x u v =  case x of
    a(y) => u(Repl y u v)
  | b(y) => v(Repl y u v)
  | e => e.
Conv x k = case x of a(y) => Conv y (C2 k a)
  | b(y) => Conv y (C2 k b)
  | e => k I.
C2 k y z = k(Concat y z).
Concat x y z = x(y z).
I x = x.
```

Here, `Conv` and `Concat` are generic functions to express a string-processing program as an HMTT. `Conv` converts an input string to a function of type $\mathtt{o} \to \mathtt{o}$, which is used as an internal representation of strings. For example, a string ab (represented as $\mathtt{a}(\mathtt{b}\,\mathtt{e})$) is converted to a function $\lambda x.\mathtt{a}(\mathtt{b}\,x)$. By using the internal representation, string concatenation is expressed as `Concat` as defined above. Let A and B be regular word languages. Then one can verify that $\mathcal{T} : A \to B \to a^*b^*e \to A^*B^*e$ (i.e. given elements of A and B, and an element of of a^*b^*e, $\mathcal{T}$ produces an element of A^*B^*e).

The function `Replace` s_a s_b is equivalent to the word homomorphism $h(a) = s_a, h(b) = s_b$.

We can also encode an arbitrary non-deterministic finite state transducer (FST) by linear HMTT. Homomorphisms and FSTs subsume a large class of practical *sanitization*; For example, let $\Sigma = \{a \mapsto 1, b \mapsto 1, \sharp \mapsto 1, e \mapsto 0\}$ and suppose $\sharp$ stands for a "dangerous" meta-character (such as tag-markers < and > of XHTML). Then, `replace` $s_\sharp$ x, replacement of $\sharp$ by $s_\sharp$, represents the sanitization of $\sharp$. One can verify the image $(\mathtt{replace}\,\mathtt{a}\,\mathtt{b}\,s_\sharp)(\mathcal{A}^*)$ does not contain an occurrence of $\sharp$ as long as $s_\sharp$ does not. Sanitization of *sequences* (such as the `<script>` tag of XHTML) can also be verified by encoding FSTs.

Interestingly, we can allow arbitrary use of string homomorphisms in a string-processing program (even inside recursion), by choosing an appropriate internal representation of strings. Consider strings over $\{\mathtt{a},\mathtt{b}\}^*$. Then, we can use a function of type $\mathtt{str} = (\mathtt{o} \to \mathtt{o}) \to (\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o}$ as the internal representation of a string. Intuitively, the first two parameters of type $\mathtt{o} \to \mathtt{o}$ represent the homomorphism images of a and b respectively, while the last parameter is the suffix of a string. (Thus, it is similar to the Church encoding of natural numbers.)

We can define primitive functions on strings as follows.

```
A xa xb z = xa z.
B xa xb z = xb z.
Empty xa xb z = z.
Concat s1 s2 xa xb z = s1 xa xb (s2 xa xb z).
I2Str x = case x of e => Empty
   |  a(y) => Concat A (I2Str y)
   | b(y) => Concat B (I2Str y).
Str2O s = s a b e.
Hom sa sb s xa xb z= s (sa xa xb) (sb xa xb) z.
```

A, B, and Empty are internal representations of a, b, and an empty string respectively. The non-terminal Concat represents the concatenation of (internal representations of) two strings. I2Str and Str2O convert an input string to the internal representation, and the internal representation to an output string, respectively. The non-terminal Hom represents a generic homomorphic function, such that Hom s_a s_b is a string homomorphism that replaces a and b with s_a and s_b respectively. Consider the following HMTT:

```
HomRep n s = F n (Hom (Concat B B) A) (I2Str s).
F n h s =  case n of zero => (Str2O s)
    | succ(m) => F m h (h s).
```

It takes a natural number n and a string s as an argument, and applies the homomorphism $\{a \mapsto bb, b \mapsto a\}$ n times. We can verify, for example, that if n is even and s is an element of $\mathtt{a}^*(\mathtt{b}^*(\mathtt{e}))$, then so is the output.

7.4 Verification of Programs Manipulating Other Algebraic Data Structures

Our verification framework is also applicable to functional programs manipulating other algebraic data structures.

The following HMTT takes two natural numbers as input and returns the multiplication.

```
Mult x y = case x of zero => zero
     | succ(z) => Add y (Mult z y).
Add x y = case x of zero => y
     | succ(z) => succ(Add z y).
```

By using our algorithm, we can verify that, given an even number and an odd number, the HMTT returns an even number.

The following HMTT takes nested lists (of elements a and b) as inputs, and returns a flat list.

```
Flatten x = F x nil.
F x z =  case x of nil => z
   |  cons x1 x2 => F x1 (F x2 z)
   | a => cons a z
   | b => cons b z.
```

We can verify that, given an arbitrary nested list, the HMTT returns a flat list.

8. Preliminary Experiments

To evaluate the effectiveness of our verification framework, we have extended the implementation of TRecS [18, 19], a model checker for recursion schemes, to handle recursion schemes with finite data domains.

The transformation from HMTT to recursion schemes with finite data domains has not been implemented yet, so that we have manually translated HMTTs in the experiments below. For the experiments on XML processing programs, automata have been automatically generated from DTD-like definitions.

Table 1 shows the result of preliminary experiments. The experiments were conducted on a machine with Intel(R) Xeon(R) CPU 3GHz and 2GB memory. The columns "O", "R", "S" show the order, the number of rules, and the size of HMTTs respectively. Here, the order of an HMTT is the largest order of the sorts of non-terminals. The order of a sort is defined by:

$$\begin{aligned}&\mathit{order}(\mathtt{i}) = \mathit{order}(\mathtt{o}) = 0\\&\mathit{order}(\kappa_1 \to \kappa_2) = \mathit{max}(\mathit{order}(\kappa_1) + 1, \mathit{order}(\kappa_2))\end{aligned}$$

The size of an HMTT is measured by the number of symbols occurring in the righthand side of the rewriting rules. The column "L" shows whether the HMTT is linear (L) or not (NL). The column "Q" shows the sum of the numbers of the states of automata for the input specification ($\mathcal{M}_{1,\dots,k}$) and the output specification ($\mathcal{M}$). All the automata used for the experiments are deterministic. The column "Y/N" shows whether the HMTT was verified (Y) or rejected (N) (note that because of the incompleteness for non-linear HMTTs, a valid HMTT verification problem may be rejected). The column "T" shows the running time of TRecS (thus, excluding the time for transformation from HMTT to RSFD, which is currently manual), measured in milli-seconds.

The programs in the first group (from Rev to Flatten) have been taken from the examples already shown in the paper; the name of each program indicates the start symbol of the corresponding example in the paper. All of them have been correctly verified in a few milliseconds, except that it took 29 milliseconds for HomRep (the last example in Section 7.3).

Remark 8.1. Note that Mult is non-linear, but it is verified correctly. If the property of Mult were "Given an even number x and a number y, Mult $x\,y$ returns an even number", then our verifier would report a false alarm. To avoid the false alarm, we need to either swap the arguments x and y, or transform Mult into a linear HMTT. □

The programs in the second group have been taken from Tozawa's experiments [33] (which are the only experimental results on exact type checking of *high-level* tree transducers in the literature, to our knowledge) and rewritten in HMTTs. The program app_fo takes a document of the following DTD:

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

Then, it inserts an appendix node to the end of the document, and moves preface and note nodes in the original document to the appendix. The program appx_fo works almost the same as appendix except that it transforms the document into an XHTML document. The program gapid takes a document of the DTD of appendix extended with a nodes, as well as another tree as arguments. It checks whether the children of each node are empty, and if so, replaces the empty children with a hole. The program then inserts a given tree to the holes. The programs xml_rep1 and xml_rep2 are the same, and differ only in their output specifications. They take a document of the same type as gapid and a tree with a hole, and then replace each div node of the document with a tree obtained from the input tree with a hole by inserting the child of div node to the hole. The output specification of xml_rep2 is not satisfied, so the model checker (correctly) rejected it with a counterexample. All the programs in the second group

Programs	O	R	S	L	Q	Y/N	T
Rev	1	2	14	L	4	Y	1
Merge	1	5	47	L	3	Y	1
AccFile	1	2	9	L	4	Y	1
MergeAddr	1	3	26	L	6	Y	1
RemoveB	2	4	36	L	7	Y	1
Replace	2	8	45	L	4	Y	1
HomRep	4	10	53	L	4	Y	29
Mult	1	3	18	NL	4	Y	1
Flatten	1	3	17	L	4	Y	1
app_fo	1	6	171	NL	21	Y	5
appx_fo	1	5	174	NL	19	Y	6
gapid	3	10	94	L	30	Y	87
xml_rep1	3	8	84	L	23	Y	3
xml_rep2	3	8	84	L	23	N	1
XhtmlS_id	1	1	113	L	24	Y	11
XhtmlS_div	1	1	110	L	24	N	2
XhtmlS_m	1	1	110	L	24	Y	18
XhtmlS_div'	1	2	161	L	24	N	2
XhtmlS_a	1	2	161	L	24	Y	17
XhtmlM_id	1	1	168	L	64	Y	395
XhtmlM_div	1	1	165	L	64	N	4
XhtmlM_m	1	1	165	L	64	Y	138
XhtmlM_div'	1	2	232	L	64	N	5
XhtmlM_a	1	2	232	L	64	Y	291
XhtmlF_id	1	1	398	L	100	Y	13,889

Table 1. Experimental Results

have been correctly verified (or rejected) in less than 100 milliseconds. Tozawa [33] reported that it took from 600 milliseconds to 2.2 seconds for his system to check programs of the same functionality.[3]

The programs in the third group are those manipulating XHTML documents. We have used two subsets of XHTML specification (indicated by XhtmlS and XhtmlM in the table), and the full XHTML specification (indicated by XhtmlF in the table). The subset XhtmlS consists of the tags <div>, <p>, <a>, <img/>, <br/> and <h1> to <h6>, which are most commonly used. XhtmlM additionally has the tags <table> <ol>,<ul>,<li>,<dl>,<dt>, <dd>, and <form>. We have tested five operations: *_id just outputs a copy of a given input document, *_div removes all the nodes labeled by div (hence the output is an invalid document), and *_m removes all the meta nodes in the header. The program *_div' removes only the tags div, instead of removing all the nodes under div, and *_a removes the tags a.

For the subsets of XHTML (XhtmlS and XhtmlM), all the examples have been verified or rejected correctly in less than a second. For the full XHTML, it took more than 10 seconds even for the identity function. This indicates that there is a problem in the scalability of our verification technique to large input/output specifications. For MTTs, Frisch and Hosoya [10] report that they could verify MTTs on the full XHTML in about a second.

9. Related Work

From the viewpoint of program verification, there are at least three threads of related work: model checking of higher-order recursion schemes, exact type-checking for macro/high-level tree transducers, and string analysis.

Model checking of recursion schemes have been extensively studied [1, 2, 11, 16, 17, 31]. Higher-order grammars (where nonterminals can generate "functions" rather than words or trees) related to higher-order recursion schemes have been introduced in early 70's [34, 36], and actively studied in 80's. [4]. Knapik et al. [17] studied the model checking problem for recursion schemes in the present form, and showed that the modal mu-calculus model checking of *safe* recursion schemes is decidable. Ong [31] extended the decidability result to arbitrary recursion schemes (without the safety condition). Kobayashi [20] showed that the resource usage verification of functional programs [14] (which subsumes other standard verification problems such as reachability and control flow analysis) can be reduced to model-checking problems for recursion schemes. Kobayashi [18] then constructed a model-checker for recursion schemes, demonstrating that the program verification method based on recursion schemes may be feasible in practice, despite the extremely high worst-case time complexity. The present work is built on these results, and extend them to deal with higher-order, multi-parameter tree transducers (while recursion schemes are just tree generators, instead of tree transformers). The extension significantly widens application domains of the verification framework, as discussed in the present paper.

In the context of exact type-checking of XML processing programs, a lot of variations of macro/high-level tree transducers have been studied [24, 26, 27, 33]. A nice feature of those tree transducers is that the class of regular tree languages is closed under the inverse of transducers, and that the inverse image is indeed computable. Thus, given a transducer verification problem $\mathcal{T}(L_1) \overset{?}{\subseteq} L_2$ (where L_1 and L_2 are regular languages), one can reduce it to the inclusion problem $L_1 \overset{?}{\subseteq} \mathcal{T}^{-1}(L_2)$ between the regular languages $\mathcal{T}^{-1}(L_2)$ and L_1. This inverse inference approach is applicable to composition of transducers, and also to the higher-order case (called high-level transducers [8, 33]). On the other hand, our approach can be considered a *forward* inference approach. Given an input language and an HMTT, our transformation essentially approximates (or precisely models, in the case of linear HMTTs) the output language by using a recursion scheme. We then check that the language represented by the recursion scheme conforms to the output specification.[4] Maneth et al. [26] take a forward inference approach for exact type checking of linear macro tree transducers, and uses a context-free tree grammar to approximate the output language. Our approach may be considered a generalization of that approach to higher-order, multi-parameter transducers. Advantages of our approach using HMTTs are: (i) HMTTs can take multiple input trees, and there is no such restriction that the first argument must be an input tree, and (ii) our verification method seems more efficient than the previous approach based on the inverse inference for higher-order transducers [33]. For (ii), the inverse inference approach suffers from extremely high time complexity for the higher-order transducers [33] or compositions of multiple transducers. On the other hand, limitations of our verification method are: (i) we have to impose the linearity restriction to ensure completeness, and (ii) the method is not directly applicable to composition of HMTTs (unless specification of interme-

[3] This is according to Tozawa's slides presented at the conference;the paper [33] does not report experimental results.

[4] Interestingly, many of the previous papers on inverse type inference (e.g. [25]) argue that forward type inference does not work because the image of the transformation is not a *regular* language. For the purpose of *checking* types, however, it is actually unnecessary to compute the image L as a regular language; it is sufficient that $L \subseteq R$ is decidable for any regular language R. Combined with the tupling transformation discussed in [**?**], we can exactly express the image of a high-level tree transducer as a recursion scheme, and since the model checking problem for recursion schemes is decidable, the forward inference approach actually works for high-level transducers (which subsume most of the other transducers in the literature)!

diate trees are given). As discussed in Remark 2.2, however, linear HMTTs are already at least as expressive as compositions of (deterministic) macro/high-level tree transducers studied in the literature. Furthermore, for (ii), one can sometimes use fusion (or deforestation) transformation [35] to compose multiple HMTTs into a single HMTT, and then apply our verification method.

As already mentioned, our method can also be applied to string analysis, by representing strings as linear trees. Previous approaches to string analysis [3, 28] extract a context-free grammar from a flow graph or SSA form of a program, and then matches it with an output specification. In such approaches, information about the output string is approximated at join points of the flow graph, especially at function calls. Our approach instead models a program as an HMTT, which naturally models higher-order functions and is more expressive than context free grammars (indeed, order-1 recursion schemes are already as expressive as context-free grammars). Thus, our verification technique would be more precise for analyzing higher-order programs. Another interesting advantage of our approach is that string homomorphisms can be expressed and freely used inside recursion (recall Section 7.3). On the other hand, the previous approaches [3, 28] return very conservative approximations when string homomorphisms are used in a loop.

Refinement types [5, 9] have been used for verification of programs manipulating algebraic data structures (the application domain discussed in Section 7.4). To our knowledge, most of the previous studies on refinement types rely on explicit type annotations (except perhaps the first work on refinement types [9], which uses a naive fixedpoint algorithm for type inference and does not seem to scale for higher-order functions). A limitation of our approach is that data structures must be strictly classified into sorts i and o. One way to overcome the limitation is to use predicate abstraction for intermediate data structures, as suggested in [20]. Another way would be to introduce an explicit coercion operation from trees of sort o to sort i, and force a programmer to annotate each coercion with a refinement type specification. Then, refinement type checking of such a program can be reduced to multiple HMTT verification problems. For example, let us consider a type checking problem of the following insertion sort:

```
let rec insert x y = ...
let rec isort x =
  case x of
    nil => nil
  | cons x1 x2 => insert x1 (coerce$^{a^*b^*c^*}_{o \to i}$(isort x2))
```

Here, suppose that we want to verify that isort takes a sequence consisting of a, b, c and returns a sequence of the form $a^*b^*c^*$. The coercion converts the result of isort x2 to an input tree. Thanks to the annotation, we should be able to split the above verification problem into the verification problems of the following two HMTTs (Isort1 and Isort2):

```
Isort1 x z = case x of nil => nil
          | cons x1 x2 => Insert x1 z.
Isort2 x z = case x of cons x1 x2 => Isort1 x2 z.
```

Isort1 is obtained from the original isort function by replacing the part $(\mathtt{coerce}^{a^*b^*c^*}_{o \to i}(\mathtt{isortx2}))$ with z. Isort2 corresponds to the part isort x2. Then, it suffices to check that Isort1 and Isort2 both conform to the specification $(a + b + c)^* \to a^*b^*c^* \to a^*b^*c^*$. The formalization of this idea is left for future work.

Intersection type systems equivalent to model-checking have been studied by Naik and Palsberg [29, 30]. They considered intersection an imperative language and did not treat higher-order programs.

10. Conclusion

We have introduced a new class of tree transducers called higher-order multi-parameter tree transducers, and proposed a verification algorithm for them. Compared with our previous verification framework based on recursion schemes [20], our new approach significantly increases application domains. The result of preliminary experiments is promising, although there is still a problem in scalability (especially with respect to the size of specifications). It is left for future work to investigate whether it is a fundamental limitation of our verification framework, or it is just a limitation of the current implementation of the underlying model checker TRecS.

References

[1] K. Aehlig. A finite semantics of simply-typed lambda terms for infinite runs of automata. *Logical Methods in Computer Science*, 3(3), 2007.

[2] K. Aehlig, J. G. de Miranda, and C.-H. L. Ong. The monadic second order theory of trees given by arbitrary level-two recursion schemes is decidable. In *TLCA 2005*, volume 3461 of *Lecture Notes in Computer Science*, pages 39–54. Springer-Verlag, 2005.

[3] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Static Analysis, 10th International Symposium, SAS 2003*, volume 2694 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2003.

[4] W. Dam. The io- and oi-hierarchies. *Theoretical Computer Science*, 20:95–207, 1982.

[5] R. Davies. *Practical refinement-type checking*. PhD thesis, Pittsburgh, PA, USA, 2005.

[6] J. Engelfriet and S. Maneth. A comparison of pebble tree transducers with macro tree transducers. *Acta Inf.*, 39(9):613–698, 2003.

[7] J. Engelfriet and H. Vogler. Macro tree transducers. *J. Comput. Syst. Sci.*, 31(1):71–146, 1985.

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

[9] T. Freeman and F. Pfenning. Refinement types for ML. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 268–277. ACM Press, 1991.

[10] A. Frisch and H. Hosoya. Towards practical typechecking for macro tree transducers. In *Database Programming Languages, 11th International Symposium (DBPL 2007)*, volume 4797 of *Lecture Notes in Computer Science*, pages 246–260. Springer-Verlag, 2007.

[11] M. Hague, A. Murawski, C.-H. L. Ong, and O. Serre. Collapsible pushdown automata and recursion schemes. In *Proceedings of 23rd Annual IEEE Symposium on Logic in Computer Science*, pages 452–461. IEEE Computer Society, 2008.

[12] W. G. J. Halfond and A. Orso. Amnesia: analysis and monitoring for neutralizing sql-injection attacks. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 174–183, New York, NY, USA, 2005. ACM.

[13] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In *Proceedings of International Conference on Functional Programming*, pages 164–175, 1997.

[14] A. Igarashi and N. Kobayashi. Resource usage analysis. *ACM Transactions on Programming Languages and Systems*, 27(2):264–313, 2005.

[15] N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 27–36, New York, NY, USA, 2006. ACM.

[16] T. Knapik, D. Niwinski, and P. Urzyczyn. Deciding monadic theories of hyperalgebraic trees. In *TLCA 2001*, volume 2044 of *Lecture Notes in Computer Science*, pages 253–267. Springer-Verlag, 2001.

[17] T. Knapik, D. Niwinski, and P. Urzyczyn. Higher-order pushdown trees are easy. In *FoSSaCS 2002*, volume 2303 of *Lecture Notes in Computer Science*, pages 205–222. Springer-Verlag, 2002.

[18] N. Kobayashi. Model-checking higher-order functions. In *Proceedings of PPDP 2009*. ACM Press, 2009.

[19] N. Kobayashi. TRecS. `http://www.kb.ecei.tohoku.ac.jp/~koba/trecs/`, 2009.

[20] N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 416–428, 2009.

[21] N. Kobayashi and C.-H. L. Ong. Complexity of model checking recursion schemes for fragments of the modal mu-calculus. In *Proceedings of ICALP 2009*, volume 5556 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009.

[22] N. Kobayashi and C.-H. L. Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *Proceedings of LICS 2009*, pages 179–188. IEEE Computer Society Press, 2009.

[23] N. Kobayashi, N. Tabuchi, and H. Unno. Higher-order multi-parameter tree transducers and recursion schemes for program verification. A longer version, available from `http://www.kb.ecei.tohoku.ac.jp/~koba/papers/hmtt.pdf`, 2009.

[24] M. Koganeyama, N. Tabuchi, and T. Tateishi. Reducing unnecessary conservativeness in access rights analysis with string analysis. In *APSEC '07: Proceedings of the 14th Asia-Pacific Software Engineering Conference*, pages 438–445, Washington, DC, USA, 2007. IEEE Computer Society.

[25] S. Maneth, A. Berlea, T. Perst, and H. Seidl. XML type checking with macro tree transducers. In *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2005)*, pages 283–294, 2005.

[26] S. Maneth and K. Nakano. Xml type checking for macro tree transducers with holes. In *Programming Language Technologies for XML (PLAN-X)*, 2008.

[27] S. Maneth, T. Perst, and H. Seidl. Exact XML type checking in polynomial time. In *ICDT 2007*, volume 4353 of *Lecture Notes in Computer Science*, pages 254–268. Springer-Verlag, 2007.

[28] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. *J. Comput. Syst. Sci.*, 66(1):66–97, 2003.

[29] Y. Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th international conference on World Wide Web (WWW 2005)*, pages 432–441. ACM Press, 2005.

[30] M. Naik. A type system equivalent to a model checker. Master Thesis, Purdue University.

[31] M. Naik and J. Palsberg. A type system equivalent to a model checker. In *ESOP 2005*, volume 3444 of *Lecture Notes in Computer Science*, pages 374–388. Springer-Verlag, 2005.

[32] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS 2006*, pages 81–90. IEEE Computer Society Press, 2006.

[33] J. Sawin and A. Rountev. Improving static resolution of dynamic class loading in java using dynamically gathered environment information. *Automated Software Engg.*, 16(2):357–381, 2009.

[34] A. Tozawa. XML type checking using high-level tree transducer. In *Functional and Logic Programming, 8th International Symposium (FLOPS 2006)*, volume 3945 of *Lecture Notes in Computer Science*, pages 81–96. Springer-Verlag, 2006.

[35] R. Turner. An infinite hierarchy of term languages - an approach to mathematical complexity. In *Proceedings of ICALP*, pages 593–608, 1972.

[36] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.

[37] M. Wand. An algebraic formulation of the chomsky hierarchy. In *Category Theory Applied to Computation and Control*, volume 25 of *Lecture Notes in Computer Science*, pages 209–213. Springer-Verlag, 1974.

[38] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 171–180, New York, NY, USA, 2008. ACM.

Acknowledgments

We would like thank Akihiko Tozawa and Yasuhiko Minamde for discussions and information about their work. We would also like to thank Luke Ong and anonymous reviewers for useful comments.

A. High-Level Tree Transducers and Linear HMTTs

This section shows that high-level tree transducers [8] (which subsume macro tree transducers) can be transformed into a linear HMTT. In essence, a (deterministic) high-level tree transducer is a higher-order function on trees (of sort `o`) defined by induction on input trees (of sort `i`).

Let us fix the input alphabet $\Sigma_I = \{a_1 \mapsto k_1, \dots, a_n \mapsto k_n\}$. A high-level tree transducer $(\mathcal{N}, \Sigma_I, \Sigma_O, \mathcal{R}, F_1)$ is a restriction of HMTT, where the rewriting rules are only of the form:

$$\begin{array}{l} F_1\, x\, \widetilde{y}_1 \rightarrow \mathbf{case}(x, \widetilde{x}_1.t_{1,1}, \dots, \widetilde{x}_n.t_{1,n}) \\ \cdots \\ F_m\, x\, \widetilde{y}_m \rightarrow \mathbf{case}(x, \widetilde{x}_1.t_{m,1}, \dots, \widetilde{x}_n.t_{m,n}) \end{array}$$

Here, x has sort `i` and it may not occur in $t_{1,1}, \dots, t_{1,n}, \dots, t_{m,1}, \dots, t_{m,n}$. The sort of each non-terminal must be of the form $\mathtt{i} \rightarrow \kappa_1 \rightarrow \cdots \rightarrow \kappa_l \rightarrow \mathtt{o}$, where `i` does not occur in $\kappa_1, \dots, \kappa_l$. F_1 has sort $\mathtt{i} \rightarrow \mathtt{o}$ (so, $\widetilde{y}_1$ is the empty sequence). Furthermore, input trees are restricted to finite ones. In the original definition of higher-level tree transducers, there is a further restriction that $\kappa_1 \rightarrow \cdots \rightarrow \kappa_l \rightarrow \mathtt{o}$ must be "derived types" [8]; We do not impose that restriction.

By using the tupling transformation [13], we can transform the above HTT into the following linear HMTT:

$$\begin{array}{l} S\, x \rightarrow G\, x\, \mathit{Proj} \\ \mathit{Proj}\, y_1 \cdots y_m \rightarrow y_1 \\ G\, x\, k \rightarrow \mathbf{case}(x, \widetilde{x}_1.H_1\, \widetilde{x}_1\, k, \dots, \widetilde{x}_n.H_n\, \widetilde{x}_n\, k) \\ H_i\, \widetilde{x}_i\, k \rightarrow G\, x_{i,1}\, (\lambda y_{1,1}, \dots, y_{1,m}. \\ \qquad\qquad G\, x_{i,2}\, (\lambda y_{2,1}, \dots, y_{2,m}. \cdots \\ \qquad\qquad G\, x_{i,k_i}\, (\lambda y_{k_i,1}, \dots, y_{k_i,m}. \\ \qquad\qquad k\, [y_{1,1}/F_1\, x_{1,1}, \dots, y_{k_i,m}/F_m\, x_{k_i,m}]t_{1,i} \\ \qquad\qquad \cdots [y_{1,1}/F_1\, x_{1,1}, \dots, y_{k_i,m}/F_m\, x_{k_i,m}]t_{m,i}) \cdots)) \\ \hfill (i = 1, \dots, n) \end{array}$$

Here, we have λ-abstractions for clarity; they can be removed by lambda-lifting. The term $[y_{1,1}/F_1\, x_{1,1}, \dots, y_{k_i,m}/F_m\, x_{k_i,m}]t_{1,i}$ denotes the term obtained by replacing $F_j\, x_{1,j}$ in $t_{1,i}$ with $y_{1,j}$. (Note that by the restriction on the sorts of F_j, $x_{1,j}$ may occur only as the first argument of F_j.) In the above rules, $G\, x\, k$ computes a sequence of values of $F_1\, x, \dots, F_m\, x$ and applies k to them (so, $G\, x\, k$ is intuitively the same as $k\, (F_1\, x) \cdots (F_m\, x)$). To compute $G\, x\, k$, it first performs a case analysis on x. If x is $a_i\, \widetilde{t_i}$, H_i is called. $H_i\, \widetilde{t_i}\, k$ first computes the values of $F_1, \dots, F_m$ for $t_{1,1}, \dots, t_{i,k_i}$. It then computes $F_1\, x, \dots, F_m\, x$ (i.e., $t_{1,i}, \dots, t_{m,i}$), and passes it to k.

The HMTT obtained by the above transformation is linear, and outputs the same tree as the original HTT, given a *finite* tree as input. (That is not always the case if an input tree is infinite, since the evaluation order has been changed by the tupling transformation.)