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 [32]. 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 $G_0$ is an order-1 recursion scheme, where the non-terminal $F$ takes a tree as an argument.

$$S \rightarrow F \ c \quad F \ x \rightarrow a \ x \ (F \ (b \ x))$$

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

$$S \rightarrow F \ c \rightarrow a \ c \ (F \ (b \ c)) \rightarrow \ldots,$$

we obtain the infinite tree shown in Figure 1. Ong [32] has shown that modal mu-calculus model-checking of recursion schemes (“Given a recursion scheme $G$ and a modal mu-calculus formula $\varphi$, does the tree generated by $G$ satisfy $\varphi$?”) is $n$-EXPTIME-complete (where $n$ is the order of the recursion scheme $G$). 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 $\ast$ denotes a random boolean value):

```plaintext
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 $G_1$:

$$S \rightarrow F \ e \quad F \ k \rightarrow \text{br}(c\ k) \ (\tau (F \ k))$$

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 lambda-calculus with recursion and tree constructors, so that they are already as expressive as the source language (of the simply-typed lambda-calculus with resources), and no information is lost by the translation. Although the

---

1 © ACM, 2010. This is the author’s version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version will be published in Proceedings of POPL’10.
model-checking of recursion schemes has extremely high worst-case time complexity (n-EXPTIME-complete for the full modal μ-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. For example, consider the following function merge:

\[
\text{let rec merge } x \ y =
\]
\[
\begin{align*}
\text{case } x \text{ of } (\mathit{[]} & \Rightarrow \text{copy } y) \\
\quad | a : x' & \Rightarrow \text{merge}_a x' y \\
\quad | b : y' & \Rightarrow \text{merge}_b x' y \\
\text{and } \text{merge}_a \ x \ y =
\end{align*}
\]
\[
\begin{align*}
\text{case } y \text{ of } (\mathit{[]} & \Rightarrow a :: (\text{copy } x) \\
\quad | a : y' & \Rightarrow a :: a :: (\text{merge } x y') \\
\quad | b : y' & \Rightarrow a :: (\text{merge}_b y' x) \\
\text{and } \text{merge}_b \ x \ y =
\end{align*}
\]
\[
\begin{align*}
\text{case } y \text{ of } (\mathit{[]} & \Rightarrow b :: (\text{copy } x) \\
\quad | a : y' & \Rightarrow b :: (\text{merge}_a y' x) \\
\quad | b : y' & \Rightarrow b :: b :: (\text{merge } x y') \\
\text{and } \text{copy } x =
\end{align*}
\]
\[
\begin{align*}
\text{case } x \text{ of } (\mathit{[]} & \Rightarrow \mathit{[]} \\
\quad | a : x' & \Rightarrow a :: (\text{copy } x') \\
\quad | b : y' & \Rightarrow b :: (\text{copy } y')
\end{align*}
\]

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 \( T_2 \):

\[
\text{Merge } x \ y \Rightarrow
\]

\[
\begin{align*}
\text{case}(x, \text{Copy } y, x'. \text{Merge}_a x' y, x'. \text{Merge}_b x' y). \\
\text{Merge}_a x \ y \Rightarrow \\
\quad \text{case}(y, \\
\quad \quad \text{a(Copy } x), \\
\quad \quad y'. \text{a(a(Merge } x y'))), \\
\quad \quad y'. \text{a(Merge}_b y' x)). \\
\text{...}
\end{align*}
\]

Here, strings are expressed as linear trees (consisting of only terminal symbols of arity 1 or 0). case(e, \( \overline{x}_1.e_1, \ldots, \overline{x}_n.e_n \)) matches a tree e with a pattern \( \overline{x} \), and reduces to \( \overline{\overline{e}_1.e} \). 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 \((T, M_1, \ldots, M_k, M)\): “Given an HMTT T that takes k (possibly infinite) input trees, and Büchi tree automata with a trivial acceptance condition (where all the states are final) \( M_1, \ldots, M_k, M \), does \( T \) always output a (possibly infinite) tree accepted by \( M \), given trees accepted by \( M_1, \ldots, M_k \) as inputs?” For example, let \( M \) be a tree automaton that accepts linear trees labeled by elements of \( a^*b^*e + a^* + a^*b^* \) (where a, b, and e are terminal symbols of arity 1, 1, and 0 respectively). Then, the verification problem \((T, M, M, M)\) is the problem of deciding whether \( T \) produces a linear tree labeled by an element of \( a^*b^*e + a^* + a^*b^* \) (thus, a tree like \( b(a(e)) \) is excluded), given two input trees labeled by elements of \( a^*b^*e + a^* + a^*b^* \).

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 \( T_2 \) (corresponding to the merge function). The automaton \( M \) has two states \( q_0 \) and \( q_1 \), with transition \( \delta(q_0, a) = q_0, \delta(q_0, b) = q_1, \delta(q_1, e) = e, \delta(q_1, b) = q_1, \delta(q_1, e) = e \). Thus, the output of \( T_2 \) applied to linear trees in \( L(M) \) is approximated by the following recursion scheme with finite data domains \( \{ q_0, q_1 \} \):

\[
S \Rightarrow \text{Merge } q_0 \ q_0.
\]

Merge x y \Rightarrow

\[
\begin{align*}
\text{case}(x, \\
\quad \text{br}_3 \text{ (Copy } y) \text{ (Merge}_a q_0 y) \text{ (Merge}_b q_1 y), \\
\quad \text{br} \text{ (Copy } y) \text{ (Merge}_b q_1 y)). \\
\text{Merge}_a x \ y \Rightarrow \\
\quad \text{case}(y, \\
\quad \quad \text{br}_3 \text{ (a(Copy } x) \text{ (a(a(Merge } x q_0)))) \text{ (a(Merge}_b q_1 x)) \\
\quad \quad \text{br} \text{ (a(Copy } x) \text{ (a(Merge}_b q_1 x))). \\
\text{...}
\end{align*}
\]

Here, the non-terminal Merge now takes elements of \( \{ q_0, q_1 \} \) as arguments instead of trees, and the case statement case(e, \( c_0, c_1 \)) reduces to \( c_0 \) if the value of \( c_0 \) is \( q_0 \), and reduces to \( c_1 \) otherwise. The three branches of \( \text{br}_3 \text{ (Copy } y) \text{ (Merge}_a q_0 y) \text{ (Merge}_b q_1 y) \) 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 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 Appendix C).

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 [25, 27, 28, 34]. 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 [34]. Other applications include: (iii) string analysis [3, 29], 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 [29]. Previous approaches [3, 29] 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 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.

## 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:

$\kappa ::= 1 \mid o \mid \kappa_1 \rightarrow \kappa_2$

The sort 1 describes input trees, while o describes output trees. The sort $\kappa_1 \rightarrow \kappa_2$ describes a function that takes an element of sort $\kappa_1$ and returns an element of sort $\kappa_2$.

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

- $\Sigma$ 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.
- $R$ is a set of rewriting rules of the form $F_1 \cdots F_n \rightarrow t$, where $F \in \text{dom}(\mathcal{N})$ is a non-terminal and $t$ is a term. Here, the set of terms is defined by:

$t ::= a \mid x \mid F \mid t_1 t_2 \mid \text{case}(x, \bar{y}_1.t_1, \ldots, \bar{y}_n.t_n)$,

where $\bar{y}_i$ 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 $1 \rightarrow \cdots \rightarrow 1 \rightarrow o$.

Furthermore, each rule $F_1 \cdots F_n \rightarrow t$ must be well-sorted, i.e. $\Gamma F_1 \cdots F_n \rightarrow t$ must be derivable by using the following sort assignment rules:

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

$K \vdash t : \Gamma$ 

$K, x : \kappa \vdash x : \kappa$

$K, t_1 : \kappa_1 \rightarrow \kappa_2 \vdash t_2 : \kappa_1$

$K, t_1 : \kappa_1 \rightarrow \kappa_2 \vdash t_2 : \kappa_2$

$K \vdash \text{case}(x, \bar{y}_1, t_1, \ldots, \bar{y}_n, t_n) : o$

$\mathcal{N}(F) \rightarrow \kappa_1 \rightarrow \cdots \rightarrow \kappa_m \rightarrow o$

$x_1 : \kappa_1, \ldots, x_m : \kappa_m \vdash t : o$

$\Gamma F_1 \cdots F_n \rightarrow t$

Note that in the sort assignment rule for case-expressions, $x$ must be of sort 1; thus, pattern matching on trees are only allowed on input trees (of sort 1).

### Remark 2.1. Each terminal symbol is used as a constructor for input trees (of sort 1) as well as for output trees (of sort o). In the sort assignment rule above, however, only the sort $o \rightarrow \cdots \rightarrow o \rightarrow o$ is assigned, as input trees cannot be constructed by an HMTT. In an environment $\rho$ introduced below, a terminal of arity $k$ has sort $1 \rightarrow \cdots \rightarrow 1 \rightarrow k$.

A $\Sigma$-labeled ranked tree, written $T$, is a mapping from $\{1, \ldots, A\}^*$ (where $A$ is the largest arity of symbols in $\Sigma$) to $\text{dom}(\Sigma)$, such that (i) $\text{dom}(T)$ is closed under the prefix operation, and (ii) if $T(x) = a$, then $\{i \mid xi \in \text{dom}(T)\} = \{1, \ldots, \Sigma(a)\}$.

The set of evaluation contexts is defined by:

$E ::= [ ] \mid a t_1 \cdots t_{j-1} E t_{j+1} \cdots t_{S(a)}$
Let \( \rho \) be a mapping from a finite set of variables (of sort \( \mathbb{1} \)) to a set of pairs consisting of a \( \Sigma \)-labeled ranked tree and a non-negative integer. (The second element of such a pair, called \( \textit{uses} \), will later be used for defining the linearity condition.) Let \( T = (\Sigma, N', R, S) \). The reduction relation \((t, \rho) \rightarrow (t', \rho')\) is defined by:

\[
F \cdot x_1 \ldots x_m \rightarrow t \in R \\
(\text{if } t_1 \ldots t_m), \rho \rightarrow (E[t_1/x_1, \ldots, t_m/x_m], \rho)
\]

\[
\rho(x) = (a_i, \bar{T}, j) \quad \rho' = \rho[x \mapsto (a_i, \bar{T}, j + 1), y' \mapsto (\bar{T}, 0)]
\]

\[
(E, \text{case}(x, \bar{y}_1, \ldots, \bar{y}_N I.N), \rho) \rightarrow (E, \text{case}(\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_N I.N), \rho')
\]

Here, \( [t_1][t_2] \) denotes the term obtained from \( t_1 \) by simultaneously replacing \( x \) with \( t_1 \) in the rule above, \( \bar{T} \) denotes a sequence of (possibly infinite) trees, and \( a_i \bar{T} \) denotes a tree whose root is labeled by \( a_i \) and children are \( T \).

Suppose that \( t \) is a term of type \( 1 \rightarrow \ldots \rightarrow 1 \rightarrow 0 \). We write \( [T, t, I_1, \ldots, I_k] \) for the tree obtained by infinitary rewriting of \((t_1 \ldots t_m, \{x_1 \mapsto (I_1, 0), \ldots, x_m \mapsto (I_k, 0)\})\). More precisely, it is defined as follows. First, we term \( t' \) for the tree inducively defined by (i) \((a_1 \cdots a_n) = (a_1 \cdots a_n) \) (where \( n \geq 0 \)), and (ii) \( t = \top \) if \( t \) is from \( F \cdot s_1 \cdots s_m \) or \( \text{case}(x, y_1, t_1, \ldots, y_N t_N) \). For example, \((a \ b) \subseteq (a \ b) \). We define \([T, t_1, \ldots, I_k] \) as the (possibly infinite) \((\Sigma \cup \{\bot \})\)-labeled tree \( ([t_1], \ldots, [t_k]) \rightarrow ([t_1]) \). Then, \([T, t] \) is defined as \( ([T, t_1], \ldots, [T, t_N]) \) for every \( \pi \in \{I_1, \ldots, A\}^* \) (where \( A \) is the largest arity), with \( \bot \cup \bot = a \) for every \( a \in \text{dom}(\Sigma) \).

**Example 2.1.** Consider the HMTT \( T = (\Sigma, N, R, Rev) \) where:

\[
\Sigma = \{a_1 \rightarrow 1, a_2 \rightarrow 1, a_3 \rightarrow 0\} \\
N = \{Rev \rightarrow \{, RevSub \rightarrow 1 \rightarrow 0 \} \}
\]

\[
R = \{Rev \rightarrow RevSub x a_3, \\
RevSub x y \rightarrow case(x, x'.RevSub x' (a_1 y), x'.RevSub x' (a_2 y), y)\}
\]

\( T \) computes the reverse of the (tree representation of) a string over \( \{a_1, a_2\} \). If \( u_1, \ldots, u_n \in \{a_1, a_2\} \), then \([T, Rev, u_1 u_2 \cdots u_n\{a_1 a_2\}]\) is:

\[
u_{a_1 a_2}(u_n, \cdots u_2, u_1 a_3)\]

Here, the terminal symbol \( a_3 \) is used to denote the end of a string. For example, \((Rev \rightarrow (a_1(a_2a_3), 0))\) is reduced as follows:

\[
(Rev x, \{x \mapsto (a_1(a_2a_3), 0)\}) \rightarrow (RevSub x a_3, \rho) \rightarrow (case(x, x'.RevSub x' (a_1 a_2 a_3)), \rho) \rightarrow (RevSub x' (a_1 a_2 a_3), \{x \mapsto (a_1 a_2 a_3, 0)\}) \rightarrow (a_1 a_2 a_3, \rho)\]

**Example 2.2.** Recall the merge function in Section 1. It is expressed as the following HMTT \( T = (\Sigma, N, R, \text{Merge}) \):

\[
\Sigma = \{a \rightarrow 1, b \rightarrow 1, e \rightarrow 0\} \\
R = \{\text{Merge} x y \rightarrow case(x, x'.\text{Merge} x y, x', \text{Merge} x y, y, \text{Copy} y)\
\]

\[
\text{Merge} x y \rightarrow case(y, y'.a(\text{Merge} x y', y'.a(\text{Merge} x y, y', \text{Copy} x)))
\]

\[
\text{Merge} x y \rightarrow case(y, y'.b(\text{Merge} x y', y'.b(\text{Merge} x y, y', \text{Copy} x)))
\]

\[
\text{Copy} x \rightarrow case(x, x'.a(\text{Copy} x), x'.b(\text{Copy} x), e, \{\})
\]

We introduce an important subclass of HMTTs.

**Definition 2.3** (linear HMTT). An HMTT \( T = (\Sigma, N, R, S) \) is linear if, for all \( \Sigma \)-labeled ranked trees \( I_1, \ldots, I_k \) (where \( k \) is the arity of \( S \)) and for all \( \rho \) and \( t, (S x_1 \cdots x_k, x_1 \mapsto (I_1, 0), \ldots, x_k \mapsto (I_k, 0)) \rightarrow (T, \rho) \) and \( \rho(y) = (I, j) \) implies \( j \leq 1 \), for every \( y \in \text{dom}(\rho) \).

Note that the linearity is a semantic condition. It is possible to construct a sound (but incomplete) 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. [27]. For higher-order cases, one can define a linear type system (conservatively) guaranteeing the linearity condition.

**Remark 2.2.** The reader may think that the linearity condition 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 [28] and stay macro tree transducers [27]) studied in the literature can be expressed by compositions of macro tree transducers (see [6] for the case of pebble tree transducers). Second, by any composition of deterministic, total macro tree transducers can be expressed by a single deterministic high-level tree transducer. Third, as discussed in Appendix C, 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 \textit{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) \( M \) is a quadruple:

\[
(\Sigma, Q, \Delta, q_0)
\]

where \( \Sigma \) is a ranked alphabet, \( Q \) is a set of states, \( \Delta \) called a transition function, is a finite subset of \( \Sigma \times \text{dom}(\Sigma) \times Q \) such that if \((q, a, q_1 \ldots q_k) \in \Delta \), then \( k = \text{arity}(a) \). A \( \Sigma \)-labeled tree \( T \) is accepted by \( M \) if there is a \( Q \)-labeled tree \( R \) such that \((i) \text{dom}(T) = \text{dom}(R); (ii) For every \( x \in \text{dom}(R), (R(x), T(x), R(x_1) \ldots R(x_n)) \in \Delta \) where \( m = \text{arity}(T(x)) \). \( R \) is a run tree of \( M \) over \( T \). A trivial automaton is deterministic if the set \( \{s \in Q \mid (q, a, s) \in \Delta \} \) is empty or a singleton set for every \( q \in Q \) and \( a \in \text{dom}(\Sigma) \).

(assuming that \( \bot \) does not occur in the alphabet of \( M \)) we write \( M^* \) for the trivial automaton obtained from \( M \) by adding the special symbol \( \bot \) (of arity 0) to the alphabet, and replacing the transition relation \( \Delta \) with \( \Delta \cup \{(q, \bot, \bot) \mid q \in Q \} \).

**Example 2.3.** Consider a trivial automaton \( M = (\Sigma, \{q_0, q_1\}, \Delta, q_0) \) where:

\[
\Sigma = \{a_1 \rightarrow 2, a_2 \rightarrow 1, a_3 \rightarrow 0\} \\
\Delta = \{(q_0, q_1, q_0 q_1), (q_0, a_2, q_1), (q_0, a_3, \epsilon), (q_1, a_2, a_3)\}
\]

\( M \) accepts \( \Sigma \)-ranked trees whose paths are labeled by elements of \( a_1 a_2 a_3 + a_1 a_2 + a_1^* \).

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 \( T \) be an HMTT with \( \text{dom}(N(S)) = k \). We write \( \models (T, M_1, \ldots, M_k, M) \) if
The problem of model-checking higher-order recursion schemes [32] (where properties are restricted to those described by trivial automata) is a special case of the HMTT verification problem (where $\kappa = 0$).

**Remark 2.3.** Note that the above definition allows $\bot$ to occur in $[T,S,I_1,\ldots,I_k]$. Given finite trees as input, an HMTT may produce $\bot$ when it does not terminate. Thus, the above problem is slightly different from the problem of exact type checking of $\bot$ separate problem.

We consider in $\Sigma$ schemes [32] (where properties are restricted to those described by labelled ranked tree obtained by infinitary rewriting of $\bot$).

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

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

$$t \ ::= \ a | d_i | x | F | t_1 t_2 | \text{case}(t_1, \ldots, t_i)$$

The set of (RSFD) sorts is $\Sigma$.

The rewriting relation $\rightarrow^G$ is defined by:

$$F x_1 \cdots x_m \rightarrow t \in R$$

where $F \in \text{dom}(N), t \in \Sigma$, and $x_1, \ldots, x_m$. Then,

$$E[t_1] \rightarrow^G E[\text{case}(d_i, t_1, \ldots, t_i)]$$

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

$$E ::= [] | a t_1 \cdots t_{m} E[t_j+1] \cdots E[t_2(a)]$$

The value tree of $G$, written by $[G]$, is the $(\Sigma \cup \{\bot \rightarrow 1\}$-labelled ranked tree obtained by infinitary rewriting of $S$, i.e.,

$$\exists \{t' : S \rightarrow^G t'\}$$

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

$$N = \{\emptyset : F : d \rightarrow \emptyset\}$$

$$R = \{S \rightarrow F d_1, F x \rightarrow \text{case}(a(F d_2), b(F d_1))\}$$

The value tree $[G]$ is the infinite tree $a(b(a(b(\cdots))))$.

**Definition 3.3 (RSFD model checking problem).** Let $G$ be an RSFD and $M$ be a trivial automaton. The RSFD model checking problem $(G, M)$ is the problem of deciding whether $[G] \in L(M^+)$. We write $\models (G, M)$ if $[G] \in L(M^+)$ 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 $\delta_i$ of a finite base type $\{d_1, \ldots, d_k\}$ into a function of sort $\delta \rightarrow \delta \rightarrow \delta \rightarrow 0 : \delta x_1, \ldots, \delta x_k \rightarrow \delta$. Then, $\text{case}(x_1, \ldots, x_k)$ can be encoded into $x_1 \cdots x_k$. In Section 5, we give 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 [32], we allow $[G]$ to generate a tree containing $\bot$ for a technical convenience. The problem of checking $[G] \in \Sigma$ is also decidable.

### 4. From HMTTT Verification to RSFD

This section reduces the HMTTT verification problem to the model-checking problem for RSFD. The reduction is sound but incomplete in general. For any HMTTT verification problem $P_i$, if the corresponding model checking problem $P_i$ for RSFD has a positive answer, then so does $P_i$, 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 $\delta_i$). Let $(T_1, M_1, \ldots, M_k, M)$ be an HMTTT verification problem. One can construct a trivial automaton $M_1, \ldots, k = (\Sigma_1, Q_1, \Delta_1, q_1)$ such that $L(M_1, \ldots, k) = L(M_1)$. Here, $L(M_1, \ldots, k, q) \neq 0$, i.e., there is no garbage state (from which no tree can be accepted).

For an HMTTT verification problem $(T_1, M_1, \ldots, M_k, M)$ where $T = (\Sigma, N, R, S)$ and $M = (\Sigma, Q, \Delta, q_0)$, we write $(T_1, M_1, \ldots, M_k, M)^\#$ for the pair $(G, M^\#)$, where:

$$G = (\Sigma \cup \{\text{br} \mapsto 2\}, N^\# \cup \{q_1, Q_1, R^\#, S^\#\})$$

$$R^\# = \{S^\# \rightarrow S q_1 \cdots q_k\}$$

$$\cup\{F \text{ br} \rightarrow \text{t}^\# \mid F \text{ br} \rightarrow \text{t} \in R\}$$

$$M^\# = (\Sigma \cup \{\text{br} \mapsto 2\}, \Delta, q_0)$$

$$\Delta^\# = \Delta \cup \{q, \text{ br}, q_1 \mid q \in Q\}.$$
We can show by induction on the derivation that the translation \[ \Gamma \rightarrow \Delta \] is well-formed.

**Theorem 4.3 (well-formedness of the recursion scheme).** Let \((T, M_1, \ldots, M_k, M)\) be an HMTT verification problem. If \(\Gamma = (T, M_1, \ldots, M_k, M)\) is well-formed, then \(\Delta = (T, M_1, \ldots, M_k, M)\) is well-formed.

**Proof** It suffices to show that each rule of \(\Gamma\) is well-sorted. We define a translation of sorts of HMTT into those of RSFD by:

\[ \text{sorts: } \Gamma \\ \text{rule: } t \\ \text{env: } \Delta \]

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

The following theorem guarantees the soundness of our transformation.

**Theorem 4.2 (soundness).** Let \((T, M_1, \ldots, M_k, M)\) be an HMTT verification problem. If \(\Gamma = (T, M_1, \ldots, M_k, M)\) is well-formed, then \(\Delta = (T, M_1, \ldots, M_k, M)\) is well-formed.

**Proof Sketch** Let \((\Gamma, M')\) be \((T, M_1, \ldots, M_k, M)\). We first note that by the definitions of \([G]\) and \([T, S, I_1, \ldots, I_k]\), we have:

1. \([G] \in L(M^+)\) and, if only if, \(\Gamma \in L(M^+\Delta)\)
2. \([T, S, I_1, \ldots, I_k] \in L(M^+)\)

We define the relation \(u \sim u'\) on RSFD terms of sort \(o\) inductively by:

1. \(u_1 \cdots u_n \sim u_1' \cdots u_n'\) if \(u_1 \sim u_1'\) and \(u_2 \sim u_2'\) if \(u_1 \sim u_1'\) for each \(i \in [1, n]\),
2. \(\Delta \sim \Gamma\) if \(\Delta \vdash \Gamma\), and
3. \(\Delta \sim \Gamma\) if \(\Delta \vdash \Gamma\)

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

**Example 4.2.** Consider an HMTT \(T = (\Sigma, N, R, S)\) where:

\[ \Sigma = \{a_1 \rightarrow 0, a_2 \rightarrow 0, a_3 \rightarrow 2\} \]
\[ N = \{S \rightarrow x \rightarrow o\} \]
\[ R = \{S \rightarrow \text{case}(x, a_3, C(x), a_2, \ldots)\} \]
\[ C(x) \rightarrow \text{case}(x, a_1, a_2, \ldots)\]

We have omitted the case for \(a_3\) since it does not matter. Let \(M_1\) be the trivial automaton \(\{q_0, q_1, \Delta_1, q_0\}\) where \(\Delta_1 = \{(q_0, a_1, \epsilon), (q_0, a_2, \epsilon)\}\). Let \(\Delta(\Gamma_1)\) be another trivial automaton \(\{q_0, q_1, \Delta_0, q_0\}\), where \(\Delta_0 = \{(q_0, a_2, \epsilon), (q_0, a_3, q_0), (q_1, a_1, \epsilon)\}\), which accepts \(a_3 a_1 a_2\). Obviously, \(\Delta(\Gamma_1)\) holds. However, \((T, M_1, \ldots, M_k, M)^\#\) is \((\Gamma, M')\) where:

\[ G = (\Sigma, N, \{q_0, q_1, \Delta, q_0\} \]
\[ R = \{S \rightarrow S \rightarrow o, S \rightarrow \text{case}(x, br(a_3, a_1, C(q_0)), a_2)\} \]
\[ C(x) \rightarrow \text{case}(x, br(a_1, a_2)) \]
\[ M' = (\Sigma, \{q_0, q_1, \Delta', q_0\} \]
\[ \Delta' = \{(q_0, br(q_0, q_1), (q_1, br(q_1, q_1)), \Delta) \}

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 cases 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.
If \( T \) is linear and \( \vdash (T, M_1, \ldots, M_k, M) \), then \( \vdash (T, M_1, \ldots, M_k, M) \).

Proof Sketch Suppose that \( \vdash (T, M_1, \ldots, M_k, M) \) and \( S' \to^* u \). It suffices to show \( u' \in \mathcal{L}(M') \) (recall the first paragraph of the proof sketch of Theorem 4.2).

By the assumption that \( T \) is linear, one can prove that \( S' \to^* u \) implies \( (S' x_1 \cdot \ldots \cdot x_k) \to (I_0, O), \ldots, x_k \to (I_k, O)) \) for every \( i \in \{1, \ldots, k\} \). (This intuitively means that \( G \) is an abstraction of \( T \) that is precise enough to capture only the reductions possible in \( T \).) For such \( u \), we have \( t^1 \in \mathcal{L}(M') \) by the assumption \( \vdash (T, M_1, \ldots, M_k, M) \). By the construction of \( M' \) and the fact \( t^1 = t^1 \), we have \( u' \in \mathcal{L}(M') \).

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

See Appendix B 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 \( T = (\Sigma, \mathcal{N}, R, S) \) where:

\[
\begin{align*}
\Sigma &\equiv \{a_1 \to 1, a_2 \to 0, a_3 \to 0\} \\
\mathcal{N} &\equiv \{S \to a \to o\} \\
\mathcal{R} &\equiv \{s x \to case(x, x', a_1(S' x'), a_2, a_3)\}
\end{align*}
\]

It just generates a copy of a given input tree. Let us consider (non-deterministic) automata \( M_1 = (\Sigma, \{q_1, q_2\}, \Delta_1, q_1) \) and \( M = (\Sigma, \{q_1, q_2\}, \Delta, q_1) \) where:

\[
\Delta_1 = \{(q_1, a_1, q_2), (q_2, a_2, e), (q_2, a_3, e)\}
\]

Since both automata accept \( \{a_1 a_2, a_1 a_3\} \), \( \vdash (T, M_1, M) \) holds. The transformation algorithm generates the recursion scheme \( G' = (\Sigma \cup \{br \to 2\}, \mathcal{N}', \{q_1, q_2\}, R, S') \) and the automaton \( M' = (\Sigma \cup \{br \to 2\}, \Delta' \cup \{(q_1, br, q_1, q_1), (q_2, br, q_2, q_1)\}, q_1) \).

Thus, the HMTT outputs \( yes \) if the sequence is a solution for the Post correspondence problem and no otherwise. Let \( M_1 \) and \( M_2 \) be trivial automata:

\[
\begin{align*}
M_1 &\equiv (\Sigma, \{q_0, q_1\}, \Delta_1, q_0) \\
\Delta_1 &\equiv \{(q_0, a_1, q_1), (q_1, a_1, q_1) | a_1 \in A\} \\
M_2 &\equiv (\Sigma, \{q_0, q_1\}, \Delta_2, q_2) \\
\Delta_2 &\equiv \{(q_0, n_0.o)\}
\end{align*}
\]

Then, \( \vdash (T, M_1, M_2) \) if and only if the Post correspondence problem \( \{(u_1, v_1), \ldots, (u_n, v_n)\} \) has no solution. Since the 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 \( M \)), such that an RSFD \( G \) is well-typed in the type system if, and only if, \( \vdash (G, M) \) 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 \( M \) be a trivial automaton \((\Sigma, Q, \Delta, q_0)\), and \( D \) be a finite set \( \{d_1, \ldots, d_k\} \). The set of types is given by:

\[
\tau ::= d_i | q_i | \bigwedge_{i=1}^n t_i \to \tau
\]

Intuitively, \( d_i \in D \) is a singleton type, describing the value \( d_i \). The type \( q_i \in Q \) describes trees accepted from \( q_i \) (i.e. elements of \( \mathcal{L}(M, q_i) \)). The type \( \bigwedge_{i=1}^n t_i \to \tau \) describes functions that take an element having types \( t_1, \ldots, t_n \) and return an element of type \( \tau \).

A type judgment for terms (where a non-terminal is treated as a variable) is of the form \( \Gamma \vdash t : \tau \), where \( \Gamma \) is a type environment, is a finite set of bindings of the form \( x : \tau \). The \( \Gamma \) may contain more than one bindings for each variable.

The typing rules are given by:

\[
\begin{align*}
\Gamma, x : \tau &\vdash M x : \tau \\
\end{align*}
\]
Let $G$ be a recursion scheme with finite domains $(\Sigma, \mathcal{N}, D, \mathcal{R}, \mathcal{S})$. We write $\vdash G \in M \if\tau$ if $\Gamma \vdash R(F) \if\tau$ holds for every $F : \tau \in \Gamma$. A recursion scheme $G$ is well-typed, written $\vdash G \in M$, just if there exists $\Gamma$ such that (i) $\vdash G \in \Gamma$, (ii) $S : q_0 \in \Gamma$, and (iii) for each $F : \tau \in \Gamma$, $\vdash \mathcal{N}(F)$ holds (i.e. the sorts declared in $\mathcal{N}$ are respected). Here, the relation $\tau :: \kappa$, which means that $\tau$ is a type of sort $\kappa$, is defined by: (i) $q :: q$, (ii) $q :: d$, and (iii) $\sum_{i=1}^{k} \tau_i \rightarrow \tau : \tau' :: \kappa'$ if $\tau :: \kappa$ and $\tau_i :: \kappa_i$ 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 $M$ be a trivial automaton, and $G$ be a recursion scheme with finite domains, if $[G]$ is well-defined, then $\vdash G \in M$ if and only if $\vdash G \in (G, M)$.

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

**Proof.** This follows from Theorem 5.1 and the decidability of the typability $\vdash G \in M$. The latter follows from the fact that, for each non-terminal, there are only finitely many candidates of types, because of the condition on well-sorting (the condition (iii)).

### 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 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:

```plaintext
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 $T$ (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

\[ O(|G|^{\exp_n((A(|Q_0| + |Q'|))^{1+\epsilon}))} \]
case expressions of HMTT. We also write \( = \) instead of \( \rightarrow \). To verify that if the list of commands only consists of \( x \), then the file pointer \( fp \) is only read and then closed, it suffices to consider the HMTT verification problem \((T, M_1, M)\), where \( M_1 \) accepts the language consisting of linear trees labeled by elements of \( r^*(e) \) and \( M \) accepts the language consisting of linear trees labeled by elements of \( \text{read}^*(\text{close}) + \text{read}^* \).

### 7.2 Verification of XML-Processing Programs

Another application domain of our HMTT verification algorithm is exact type checking of XML-processing programs [25, 27, 28, 34]. 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 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.

### 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 [29] or detect security vulnerabilities such as SQL-injection or cross-site scripting [12, 15, 38]. They are also applicable to other application do-

<table>
<thead>
<tr>
<th>M x = case x of doc x1 =&gt; M x1</th>
<th>( \Delta = {(q_0, doc, q_1), (q_1, addr, q_2, q_1), (q_2, pc, e), (q_1, e, t)} )</th>
</tr>
</thead>
</table>
| MergeAddr x y = case x of addr x1 x2 => addr (M x1) (M x2) | \( M \) accepts (the binary tree representation of) trees of the form \( \text{doc}(\text{addr}(\text{pc}^*)) \), i.e. documents containing a sequence of address data \( \text{addr}(\text{pc}) \) (the terminal \( e \) represents the end of a sequence). Consider the following HMTT:

### 7.2 Verification of XML-Processing Programs

Another application domain of our HMTT verification algorithm is exact type checking of XML-processing programs [25, 27, 28, 34]. 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 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.

### 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 [29] or detect security vulnerabilities such as SQL-injection or cross-site scripting [12, 15, 38]. They are also applicable to other application do-

### 7.2 Verification of XML-Processing Programs

Another application domain of our HMTT verification algorithm is exact type checking of XML-processing programs [25, 27, 28, 34]. 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 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.

### 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 [29] or detect security vulnerabilities such as SQL-injection or cross-site scripting [12, 15, 38]. They are also applicable to other application do-

<table>
<thead>
<tr>
<th>UpgradeAddr x y = case x of addr x1 x2 =&gt; addr (M x1) (M x2)</th>
<th>( \Delta = {(q_0, doc, q_1), (q_1, addr, q_2, q_1), (q_2, pc, e), (q_1, e, t)} )</th>
</tr>
</thead>
</table>
| MergeAddr x y = case x of addr x1 x2 => addr (M x1) (M x2) | \( M \) accepts (the binary tree representation of) trees of the form \( \text{doc}(\text{addr}(\text{pc}^*)) \), i.e. documents containing a sequence of address data \( \text{addr}(\text{pc}) \) (the terminal \( e \) represents the end of a sequence). Consider the following HMTT:

### 7.2 Verification of XML-Processing Programs

Another application domain of our HMTT verification algorithm is exact type checking of XML-processing programs [25, 27, 28, 34]. 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 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.

### 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 [29] or detect security vulnerabilities such as SQL-injection or cross-site scripting [12, 15, 38]. They are also applicable to other application do-

### 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 [29] or detect security vulnerabilities such as SQL-injection or cross-site scripting [12, 15, 38]. They are also applicable to other application do-

### 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 [29] or detect security vulnerabilities such as SQL-injection or cross-site scripting [12, 15, 38]. They are also applicable to other application do-
The sort of each non-terminal is given by:

\[
\begin{align*}
A, B, \text{Empty} &: \text{str} \\
\text{Concat} &: \text{str} \rightarrow \text{str} \rightarrow \text{str} \\
\text{I2Str} &: \text{i} \rightarrow \text{str} \\
\text{Str20} &: \text{str} \rightarrow \text{0} \\
\text{Hom} &: \text{str} \rightarrow \text{str} \rightarrow \text{str} \rightarrow \text{str}
\end{align*}
\]

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 Str20 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:

\[
\text{HomRep } n \ s = F \ n \ (\text{Hom } (\text{Concat } B \ B) \ A) \ (\text{I2Str } s).
\]

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 \(a^* (b^* (a))\), 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.

\[
\begin{align*}
\text{Mult } x \ y &= \text{case } x \ of \ zero \Rightarrow \zero \\
& \quad \cup \text{succ}(z) \Rightarrow \text{Add } y (\text{Mult } z \ y).
\end{align*}
\]

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.

\[
\begin{align*}
\text{Flatten } x &= F \ x \ n11. \\
F \ x \ z &= \text{case } x \ of \ n11 \Rightarrow z \\
& \quad | \ cons \ x1 \ x2 \Rightarrow F \ x1 \ (F \ x2 \ z) \\
& \quad | \ a \Rightarrow cons \ a \ z \\
& \quad | \ b \Rightarrow cons \ b \ z.
\end{align*}
\]

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 implemented the transformation of TREC\$S [18, 19], a model checker for recursion schemes, to handle recursion schemes with finite data domains. (Kobayashi [20] has shown that recursion schemes with finite data domains can be transformed into pure recursion schemes, but as the transformation suffers from an exponential increase the size of the recursion schemes, we have extended the type-based model checking algorithm of TREC\$S to directly handle 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 programs used in the experiments below are available at http://www.kb.ecei.tohoku.ac.jp/~tabee/hmtt-experiments.html. 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:

\[
\text{order}(i) = \text{order}(a) = 0
\]

\[
\text{order}(\kappa_1 \rightarrow \kappa_2) = \max(\text{order}(\kappa_1) + 1, \text{order}(\kappa_2))
\]

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 \(M_{\text{input}}\) and the output specification \(M_\text{output}\). 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 TREC\$S (thus, excluding the time for transformation from HMTT to RSFD, which is currently manual), measured in milliseconds.

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 \text{HomRep} (the last example in Section 7.3).

Remark 8.1. Note that \text{Mult} is non-linear, but it is verified correctly. If the property of \text{Mult} were “Given an even number \(x\) and a number \(y, \text{Mult } x \ y\) 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 \text{Mult} into a linear HMTT.

The programs in the second group have been taken from Tozawa’s experiments [34] (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 \text{app}_1\text{fo} takes a document of the following DTD:

\[
\begin{align*}
\text{type } \text{Input} &= \text{doc}[\text{Preface}, (\text{Div}|\text{P}|\text{Note})*] \\
\text{type } \text{Preface} &= \text{preface}[\text{Header}, \text{P}]* \\
\text{type } \text{Header} &= \text{header}[] \\
\text{type } \text{P} &= \text{p}[] \\
\text{type } \text{Div} &= \text{div}[(\text{Div}|\text{P}|\text{Note})*] \\
\text{type } \text{Note} &= \text{note}[\text{P}]*
\end{align*}
\]

Then, it inserts an appendix node to the end of the document, and moves \text{preface} and \text{note} nodes in the original document to the appendix. The program \text{app}_2\text{fo} works almost the same as \text{app}_1\text{fo} except that it transforms the document into an XHTML document. The program \text{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 \text{xml}_1\text{rep1} and \text{xml}_2\text{rep2} are the same, and differ only in their output specifications. They take a document of the same type as \text{gapid} and a tree with a hole, and then replace each \text{div} node of the document with a tree obtained from the input tree with a hole by inserting the child of \text{div} node to the hole. The output specification of \text{xml}_1\text{rep2} is not satisfied, so the model checker (correctly) rejected it with a counterexample. All the programs in the second group have been correctly verified (or rejected) in less than 100 milliseconds. Tozawa [34] reported that it took from 600 milliseconds to
2.2 seconds for his system to check programs of the same functionality.\footnote{This is according to Tozawa’s slides presented at the conference, the paper [34] does not report experimental results.}

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>. The DTD definitions for XhtmlS and XhtmlM are given in Appendix D. 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), *_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 has been extensively studied [1, 2, 11, 16, 17, 32]. Higher-order grammars (where non-terminals can generate “functions” rather than words or trees) related to higher-order recursion schemes have been introduced in early 70’s [35, 37], 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 [32] 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 [25, 27, 28, 34]. 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 \( T(L_1) \subseteq L_2 \) (where \( L_1 \) and \( L_2 \) are regular languages), one can reduce it to the inclusion problem \( L_1 \subseteq T^{-1}(L_2) \) between the regular languages \( T^{-1}(L_2) \) 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, 34]). 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.\footnote{Interestingly, many of the previous papers on inverse type inference (e.g. [26]) 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 Appendix C, 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).}

Maneth et al. [27] 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 [34]. For (ii), the inverse inference approach suffers from extremely high time complexity for the higher-order transducers [34] 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 intermediate 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 [36] 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, 29] 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, 29] 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:

\[
\begin{align*}
\text{let rec insert} & \ x \ y = \ldots \\
\text{let rec isort} & \ x = \\
& \quad \text{case x of}
& \quad \quad \text{nil} \Rightarrow \text{nil} \\
& \quad \quad \text{cons} \ x1 \ x2 \Rightarrow \text{insert} \ x1 \ (\text{coerce}_{\text{a} \rightarrow \text{b} \rightarrow \text{c}}(\text{isort} \ x2))
\end{align*}
\]

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 sort 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):

\[
\begin{align*}
isort1 \ x \ z & = \text{case} \ x \ \text{of} \ \text{nil} \Rightarrow \text{nil} \\
& | \ \text{cons} \ x1 \ x2 \Rightarrow \text{insert} \ x1 \ z
\end{align*}
\]

\[
\begin{align*}
isort2 \ x \ z & = \text{case} \ x \ \text{of} \ \text{cons} \ x1 \ x2 \Rightarrow \text{isort} \ x1 \ x2 \ z
\end{align*}
\]

isort1 is obtained from the original isort function by replacing the part \((\text{coerce}_{\text{a} \rightarrow \text{b} \rightarrow \text{c}}(\text{isort} x2))\) with z. isort2 corresponds to the part isort x2. Then, it suffices to check that isort1 and isort2 both conform to the specification \((a + b + c)^* \rightarrow a'b'c' \rightarrow a''b''c''\). The formalization of this idea is left for future work.

Intersection type systems equivalent to model-checking have been studied by Naik and Palsberg [30, 31]. They considered an intersection type system for 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 TREC5.

Acknowledgments

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

References


Appendix

A. Proof of Soundness of Transformation (Theorem 4.2)

In the sequel, we fix a HMTT \( (T, M_1, \ldots, M_k, M) \) and a RSFD \( (G, M') \) such that

\[ (T, M_1, \ldots, M_k, M, M') = (G, M'). \]

(Thus, \( M_1, \ldots, k = (\Sigma, Q, \Delta, q_0) \) is also fixed.)

We first note the following lemmas hold by the definitions of \( [G] \) and \([T, S, I_1, \ldots, I_k]\).

**Lemma A.1.** \( [G] \in \mathcal{L}(M') \) if, and only if, \( u^+ \in \mathcal{L}(M') \) for every \( u \) such that \( S^i \rightarrow^* u \).

**Lemma A.2.** \( [T, S, I_1, \ldots, I_k] \in \mathcal{L}(M') \) if, and only if, \( t^+ \in \mathcal{L}(M') \) for every \( t \) such that \( \exists p, \{ (S x_1 \cdots x_k, \{ x_i \mapsto (I_i, 0) \} \mid t = 1, \ldots, k) \} \rightarrow^* (t, p) \).

Intuitively, an RSFD term \( u \) represents a set of terms (where \( \oplus \) denotes a union). We define the following relation \( \sim u' \), which means that \( u' \) is an element of the set represented by \( u \).

**Definition A.1.** The relation \( u \sim u' \) between RSFD terms is the least relation closed under the following rules:

\[
\begin{align*}
&\text{for some } 1 \leq i \leq n, \quad \text{br } u_1 \cdots u_n \sim u_i' \\
&\sum_{1 \leq i \leq n} a u_1 \cdots u_n \sim a u_1' \cdots u_n' \\
&F \, \sim u \Rightarrow F \, \sim u \\
&\text{case}(u_1, \ldots, u_l) \sim \text{case}(u_1, \ldots, u_l)
\end{align*}
\]

As stated in the proof sketch of the soundness, given \( \models ([G], M') \) and \( (S x_1 \cdots x_k, \{ x_i \mapsto (I_i, 0) \} \mid i = 1, \ldots, k) \) \( \rightarrow^* (t, p) \) with \( t \in \mathcal{L}(M_1), \ldots, k \in \mathcal{L}(M_k) \), it suffices to show \( t^+ \in \mathcal{L}(M') \). For that purpose, we prove that we can construct \( u \) such that \( S^i \rightarrow^* \sim u \) and \( t^+ = u^+ \) (Corollary A.5).

The idea of the proof is to show that each HMTT reduction \( \rightarrow^* \) can be simulated by a RSFD reduction \( \sim \) followed by \( \rightarrow^* \).

For the simulation, we use the following relation \( (t, p) \sim u \), which means that \( u \) is an approximation of the HMTT configuration \( (t, p) \).

**Definition A.2.** Let \( \sim \) be a mapping from variables to \( Q \). We write \( p \sim \sigma \) if \( \rho(x) = (T, i) \Rightarrow T \in \mathcal{L}(M_1, \ldots, M_k, \sigma(x)) \) for all \( x \in \text{dom}(\rho) \). Further, we write \( (t, p) \sim u \) if \( p \sim \sigma \) and \( u = \sigma(t^+) \), where \( \sigma() \) is the standard substitution. We write \( (t, p) \sim u \) if \( (t, p) \sim u \) for some \( \sigma \).

**Lemma A.3 (Substitution).** If \( (t, p) \sim u_1 \) and \( (t, p) \sim u_1 \) for each \( i \in \{1, \ldots, m\} \), then \( ([I_1, \ldots, I_m/x_1, \ldots, x_m] t, p) \sim_u [u_1, \ldots, u_m/x_1, \ldots, x_m] u \).

**Proof** By induction on the structure of \( t \). The cases \( t = a \) and \( t = F \) are trivial.

Case \( t = x \): Immediately follows from the definition of \( (t, p) \sim_u u \).

Case \( t = t'_1 t'_2 \): Assume \( (t, p) \sim_u u \) and \( (t_i, p) \sim_u u_i \) \((i = 1, \ldots, m) \). By \( \sigma(t') = u \), we have \( u = \sigma(t'_1 t'_2) \sigma(t'_1) \sigma(t'_2) \). Then, by the induction hypothesis, it follows that \( ([I_1, \ldots, I_m/x_1, \ldots, x_m] t'_i, p) \sim_u [u_1, \ldots, u_m/x_1, \ldots, x_m] t'_i \) for each \( i = 1, 2 \). Therefore, \( ([I_1, \ldots, I_m/x_1, \ldots, x_m] t, p) \sim_u [u_1, \ldots, u_m/x_1, \ldots, x_m] u \).

Case \( t = (x_1 \cdot \ldots \cdot x_n.t'_n) \): Assume \( (t, p) \sim_u u \) and \( (t_i, p) \sim_u u_i \) \((i \in \{1, \ldots, n\}) \). By \( \sigma(t') = u \), we have \( u = \sigma(t'_1, t'_2) \sigma(t'_1) \sigma(t'_2) \). Then, by the induction hypothesis, it follows that \( ([I_1, \ldots, I_m/x_1, \ldots, x_m] t'_i, p) \sim_u [u_1, \ldots, u_m/x_1, \ldots, x_m] t'_i \) for each \( i \in \{1, \ldots, n\} \). Therefore, we get \( ([I_1, \ldots, I_m/x_1, \ldots, x_m] t, p) \sim_u [u_1, \ldots, u_m/x_1, \ldots, x_m] u \).

The following lemma establishes a one-step simulation:

**Lemma A.4.** If \( (t, p) \sim u \) and \( (t, p) \rightarrow_T (t', p') \), then \( u \sim u' \) and \( (t', p') \sim u' \) for some \( u' \).

**Proof** By case analysis on the redex of \( t \).

Case \( t = E[t] \): Follows from Lemma A.3.

Case \( t = E[x] \cdot \ldots \cdot y_n.t_n \cdot \ldots \cdot y_1.t_1 \): Suppose, without loss of generality, that \( \rho(x) = (a_1(T_1 \cdots T_n), j) \) and \( (t, p) \rightarrow_T (t', p') \). By \( \rho \) and \( \rho(x) = (a_1(T_1 \cdots T_n), j + 1) \), \( y_j \mapsto (T_1, 0), \ldots, y_n \mapsto (T_0, 0) \) must be the case. Because \( (t, p) \sim_u u \), there exists \( \sigma \) such that \( u = \sigma(t'_{\#}) \) and \( \Delta(a_1, \sigma(x)) = q_1 \cdots q_n \) for some \( q_1, \ldots, q_n \) such that \( T_i \in \mathcal{L}(M_{1, \ldots, k}, q_i) \) for each \( i \). We can assume without loss of generality that \( u' \) is of the form \( \text{br}(u') \) where \( u = \sigma(t'_{\#}) \) and \( \Delta(a_1, \sigma(x)) = q_1 \cdots q_n \). Therefore, \( \sigma(E'[u']) \sim \sigma(E'[t'_1]) \). We can show that \( \rho' \sim \sigma' \). Thus, \( (E'[t'_1], \rho') \sim \sigma'(E'[t'_1]) \).

**Corollary A.5.** If \( (t, p) \sim u \) and \( (t, p) \rightarrow_T (t', p') \), then \( u \sim u' \) with \( (t', p') \sim u' \) for some \( u' \).

**Proof** By Lemma A.4, we get \( u \sim u' \) with \( (t', p') \sim u' \) for some \( u' \). Since \( u \sim u' \) implies \( u \sim u' \) we obtain \( u \sim u' \).

We now prove the soundness.

**Proof of Theorem 4.2** Suppose that we have \( \models ([G], M') \) and \( (S x_1 \cdots x_k, \{ x_i \mapsto (I_i, 0) \} \mid i = 1, \ldots, k) \) \( \rightarrow^* (t, p) \) for some \( I_1 \in \mathcal{L}(M_1), \ldots, k \in \mathcal{L}(M_k) \). Then, by Corollary A.5, there exist some \( u \) such that \( S^i \rightarrow^* \sim u \sim u \). By Lemma A.1 and \( \models ([G], M') \), we have \( u^+ \in \mathcal{L}(M') \). By the definition of \( \sim \) and the construction of \( M' \), we obtain \( u^+ \in \mathcal{L}(M') \). By \( (t, p) \sim u \), we get \( t^+ = u^+ \). Thus, we have \( t^+ \in \mathcal{L}(M') \). By Lemma A.2, we get \( [T, S, I_1, \ldots, I_k] \in \mathcal{L}(M) \).
B. Proof of Completeness of Transformation for Linear HMTT (Theorem 4.3)

Below, we fix a HMTT \((T, M_1, \ldots, M_k, M)\) and a RSFD \((G, M')\) such that
\[
(T, M_1, \ldots, M_k, M)^\# = (G, M').
\]
(Thus, \(M_1, \ldots, M_k = (\Sigma, Q, \Delta, q_0)\) is also fixed). We also assume that \(T\) is a linear HMTT.

**Lemma B.1.** For any \(u\) such that \(S' \xrightarrow{\sigma} u\), there exist \((t, \rho)\) and \(I_1 \in \mathcal{L}(M_1), \ldots, I_k \in \mathcal{L}(M_k)\) such that \((S \mid x_i \mapsto (I_i, 0) \mid i = 1, \ldots, k)) \xrightarrow{\sigma} (t, \rho)\) and
\[
u' = t\]
Proof: Given in Section B.1. □

**Lemma B.2.** Suppose one of the following conditions holds:

1. The output specification \(M\) is deterministic.
2. The symbols \(\mathbf{br}\) only occur at the top-level of \([G]\).

For any \(u\) such that \(S' \xrightarrow{\sigma} u\), if \(u' \in \mathcal{L}(M')\) for every \(u'\) with \(u \sim u'\), then \(u \in \mathcal{L}(M')\).

Proof: We prove the lemma by contradiction on the structure of \(u\): For any \(q\) and \(x\), if \(u' \in \mathcal{L}(M')\), for every \(u'\) with \(u \sim u'\), then \(u\) is trivial since \(u' = \bot\).

(2) By induction on the structure of \(u\). We perform case analysis on the root of \(u\).

The case \(u = \mathbf{br} u_1 \cdots u_m\) is trivial by the induction hypothesis, the construction of \(M\), and the definition of \(\sim\).

The case \(u = a u_1 \cdots u_m\): We assume that \(u' \in \mathcal{L}(M', q)\) for every \(u'\) with \(u \sim u'\). Let \(u' = a u'_1 \cdots u'_m\) for some \(u'_1, \ldots, u'_m\). Since \(M'\) is deterministic, there is a unique sequence \(q_1 \cdots q_m = q(a, q).\) Therefore, we have \(u' \in \mathcal{L}(M', q)\) for any \(i \in [1, m]\) and \(u'_i\) such that \(u_i \sim u'_i\).

By the induction hypothesis, \(u_i \in \mathcal{L}(M', q_i)\). Thus, \((a u_1 \cdots u_m)^{\sim} \in \mathcal{L}(M').\)

The cases \(u = F\overline{u}\) and \(u = \text{case}(x, u_1, \ldots, u_i)\) are trivial since \(u' = \bot\).

□

**Proof of Theorem 4.3** Suppose one of the following conditions holds:

1. The output specification \(M\) is deterministic.
2. The symbols \(\mathbf{br}\) only occur at the top-level of \([G]\).

Suppose that we have \(\mid (T, M_1, \ldots, M_k, M) \mid S' \xrightarrow{\sigma} u\). It suffices to show \(u \in \mathcal{L}(M')\). By Lemma B.1, \(u \sim u'\) implies \((S \mid x_i \mapsto (I_i, 0) \mid i = 1, \ldots, k) \xrightarrow{\sigma} (t, \rho)\) and
\[
u' = t\]
for some \(t, \rho, I_1 \in \mathcal{L}(M_1), \ldots, I_k \in \mathcal{L}(M_k)\). By Lemma A.2 and \((T, M_1, \ldots, M_k, M)\), we get \(t' \in \mathcal{L}(M')\). Thus, we have \(u' \in \mathcal{L}(M')\). By the construction of \(M'\), we get \(u' \in \mathcal{L}(M')\).

**B.1 Proof of Lemma B.1**

The idea of the proof is to show that every RSFD reduction \(\xrightarrow{\sigma} \xrightarrow{\sigma} u\) can be simulated by a linear HMTT reduction \(\xrightarrow{\sigma} (t, \rho)\) that is defined as follows:

**Definition B.1.** We write \((t, \rho) \xrightarrow{\sigma} \xrightarrow{\sigma} (t', \rho')\) if \((t, \rho) \xrightarrow{\sigma} (t', \rho')\) and all the uses of \((t, \rho)\) are at most 1.

For the simulation, we use the following relation \((t, \rho) \approx u\), which means that \(u\) is an approximation of \((t, \rho)\):

**Definition B.2.** We write \((t, \rho) \approx u\) if \((t, \rho) = (T, 0) \Rightarrow T \in \mathcal{L}(M_1, \ldots, M_k, \sigma(x))\) holds for all \(x \in \text{dom}(\rho) \cap \text{FV}(t)\). Further we write \((t, \rho) \approx u\) if \((t, \rho) \approx (t', \rho')\) and for every \(x \in \text{dom}(\rho) \cap \text{FV}(t)\) and \(p(x) = (T, 0)\) if only if \(p(x) = (T, 0)\). We also write \((t, \rho) \approx (t', \rho')\) if \((t, \rho) \approx (t', \rho')\) and all the uses of \((t', \rho')\) are the same as those of \(\rho'\).

Note here that if the use of a variable \(x \in \text{dom}(\rho)\) is 0 or \(x \notin \text{FV}(t)\), then \(x\) is dead. Namely \(x\) will never be used in a linear HMTT reduction of \((t, \rho)\). Thus, we ignore whatever tree \(\rho, \rho'\) assign to dead variables in the definitions of \(\approx\) and \(\approx\).

**Lemma B.3 (Substitution).** Suppose that \((t, \rho) \approx u\) and \((t, \rho) \approx u_i\) for some \(i \in \{1, \ldots, m\}\). Then, \((\{x_1, \ldots, x_m|x_1, \ldots, x_m\}) \approx u_i\).

**Proof** Similar to the proof of Lemma A.3. □

The following lemma establishes a one-step \(\approx_{\text{use}} \longrightarrow_{\text{lin}}\) simulation of \(\longrightarrow_{\approx}\):

**Lemma B.4.** For any \(u, \rho\),
\[
\phi \leq u \longrightarrow \psi \text{ lin}
\]
then there exist \(\phi, \rho, \psi\) such that
\[
u_1 \xrightarrow{\approx} \nu_2
\]
Proof: We perform case analysis on the redex of \(u_1\).

Case \(u_1 = E[F\overline{u}]\): Since \(u_1 \approx \sigma(t, \rho)\) and \((t, \rho) \approx \sigma\) for some \(\sigma\), we have \(t_1 = E[F\overline{u}]\) for some \(E'T\) such that \(\overline{u} \approx \sigma(t, \rho)\) and \(E = \sigma(E')\). Suppose that \(F \rightarrow t \in \text{R}\). We have \(u_2 = E[u]\). Let \(\phi_1 = \rho_1, \phi_2 = u_1, \text{ and } (t_1, \rho_1) \longrightarrow_{\text{lin}} (E'[\overline{E'[t]}]\) when \(\phi_1 \approx \phi_2\).

By Lemma B.3, we obtain \(\sigma(E'[\overline{E'[t]})\approx u_2\). Therefore, we obtain \(E'[\overline{E'[t]})\approx u_2\).

Case \(u_1 = E[E'(\text{case}(q, u_1, \ldots, u_i))\]: Since \(u_1 \approx \sigma(t, \rho)\) and \((t, \rho) \approx \sigma\) for some \(\sigma\), we have \(t_1 = E[E'(\text{case}(x, y_1, \ldots, y_n)\). For some \(\overline{u}_j = \sigma(t, \rho)\) and \((t_1, \rho_1) \longrightarrow_{\text{lin}} (E'[\overline{E'[t]})\.\)

Since \(\sigma(t, \rho) \approx (t_1, \rho_1) \approx u_1\) and \((t_1, \rho_1) \longrightarrow (t_1, \rho_1) \) use \(\rho_1\), we get \((t_1, \rho_1) \longrightarrow_{\text{lin}} (E'[\overline{E'[t]})\.\)

□
The following lemma obtains a one-step \( \approx_{\text{use}} \rightarrow_{\text{lin}} \approx_{0} \) simulation of \( \rightarrow_{\text{sim}} \approx_{\text{use}} \) simulation of \( \rightarrow_{\omega} \):

**Lemma B.5.** If

\[
\begin{align*}
    u_1 &\rightarrow_{\text{lin}} u_2, \\
    (t_1, \rho_1) &\rightarrow_{\text{lin}} (t_2, \rho_2) \approx_{\text{use}} (t_2, \rho_2') \\
    \text{then there exists } \rho_1' \text{ such that}
    &u_1 \rightarrow_{\omega} u_2, \\
    &\rho_1' \approx_{\text{use}} (t_1, \rho_1') \rightarrow_{\text{lin}} (t_2, \rho_2').
\end{align*}
\]

**Proof:** We perform case analysis on the redex of \( t_1 \).

Case \( t_1 = E[F \tilde{t}] \): We have \( \rho_2 = \rho_1 \). Let \( \rho_1' = \rho_2' \). Then, we get \( (t_1, \rho_1') \approx_{\text{use}} (t_2, \rho_2') \approx_{\text{use}} (t_2, \rho_2') \). We have \( (t_1, \rho_1') \approx_{\text{use}} u_1 \) since \( \sigma(t_1') = u_1 \) and \( (t_1, \rho_1') \approx_{\sigma} \sigma(t_2) \approx_{\omega} u_2 \).

Case \( t_1 = E[\text{case}(x, \tilde{y}_1, \ldots, \tilde{y}_n, \tilde{t}_0)] \): We have \( p_1(x) = (a_1 \ldots T_{m-1}, y_1 \rightarrow (T_1, 0), \ldots, y_m \rightarrow (T, 0)) \) and \( p_2 = p_1(x) \rightarrow (a_1 \ldots T_{m-1}, y_1 \rightarrow (T_1, 0), \ldots, y_m \rightarrow (T, 0)) \). We can assume that \( p_1' = p_1(x) \rightarrow (a_1' \ldots T_{m-1}', y_1 \rightarrow (T_1', 0), \ldots, y_m \rightarrow (T, 0)) \) for some \( a_1, \ldots, T_{m-1}, a_1' \ldots T_{m-1}' \). Therefore, we get \( (t_1, \rho_1') \approx_{\text{lin}} (t_1, \rho_1') \) and \( (t_2, \rho_2') \approx_{\text{lin}} (t_2, \rho_2') \). We obtain \( (t_2, \rho_2') \approx_{\text{lin}} (t_2, \rho_2') \) by the definition of \( \approx_{\text{lin}} \).

Let us consider such that \( (t_2, \rho_2') \approx_{\omega} \sigma(t_2') \approx_{\omega} u_2 \).

We have \( (t, \rho) \approx_{\text{lin}} (t, \rho) \). Then, we get \( (t, \rho) \approx_{\text{lin}} (t, \rho) \) and \( (t, \rho) \approx_{\text{lin}} (t, \rho) \). We obtain \( (t, \rho) \approx_{\text{lin}} (t, \rho) \) and \( (t, \rho) \approx_{\text{lin}} (t, \rho) \). We know \( (t, \rho) \approx_{\text{lin}} (t, \rho) \).

**Lemma B.6.** If

\( (t, \rho) \approx_{\text{lin}} (t', \rho') \)

then

\( (t, \rho) \rightarrow_{\text{lin}} (t', \rho') \).

**Proof:** We assume that \( (t, \rho) \approx_{\text{lin}} (t, \rho) \) and \( (t, \rho) \rightarrow_{\text{lin}} (t', \rho') \) hold, and show that \( (t, \rho) \rightarrow_{\text{lin}} (t', \rho') \) for some \( p_2 \) such that \( (t', \rho') \approx_{\text{lin}} (t', \rho') \). We perform case analysis on \( t \).

Case \( t = E[F \tilde{t}] \): We have \( \rho = \rho_1 \). Then, we get \( (t, \rho) \approx_{\text{lin}} (t', \rho') \).

Case \( t = E[\text{case}(x, \tilde{y}_1, \ldots, \tilde{y}_n, \tilde{t}_0)] \): We have \( p_1(x) = p_1(x) \rightarrow (T, 0) \) for some \( T \). We obtain \( \rho = \rho_1 \) for some \( y \rightarrow (T, 0) \). We get \( (t, \rho) \rightarrow_{\text{lin}} (t', \rho') \).

**Lemma B.7.** If \( S \tilde{x}_1 \cdots \tilde{x}_k = u_0 (\rightarrow_{\omega})^* u_m, \) then there exist \( t_0, \ldots, t_m, \rho_0, \ldots, \rho_m, \tilde{x}, I \) such that

\[
    (S \tilde{x}, \tilde{x} \rightarrow (T, 0)) \rightarrow_{\text{lin}} (t_0, \rho_0).
\]

**Proof:** We prove the lemma by induction on \( m \). The case \( m = 0 \) is trivial. Suppose \( m \geq 1 \). Then we have

\( (t_0, \rho) \approx_{\text{lin}} (u_0 (\rightarrow_{\omega})^* u_{m-1}) \rightarrow_{\omega} u_m. \)

By the induction hypothesis, we have

\( (t_0, \rho) \approx_{\text{lin}} (u_0 (\rightarrow_{\omega})^* u_{m-1}) \rightarrow_{\omega} u_m. \)
denotes the term obtained by replacing \( F_j x_{1,j} \) in \( t_{1,j} \) 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 \( F_1 x, \ldots, F_m x \) 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 \( a_i t_i \), \( H_i \) is called. \( H_i t_i k \) first computes the values of \( F_1, \ldots, F_m \) for \( t_{1,i}, \ldots, t_{m,i} \). It then computes \( F_1 x, \ldots, F_m x \) (i.e., \( t_{1,i}, \ldots, t_{m,i} \)), 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.)

**Example C.1.** Consider the following HTT (of order 0)

\[
F : \quad \text{case}(x, \\
\quad x' \cdot \text{br} (a(Copy x')) (\text{Take}_b x'), \\
\quad x' \cdot \text{br} (b(Copy x')) (b(\text{Take}_b x')), \\
\quad c)
\]

\[
\text{Copy} \ x \ \rightarrow \ \text{case}(x, x' \cdot a(Copy x'), x' \cdot b(Copy x'), c)
\]

\[
\text{Take}_b \ x \ \rightarrow \ \text{case}(x, x' \cdot \text{Take}_b x'), x' \cdot b(\text{Take}_b x'), c)
\]

The start symbol is \( F \). It can be transformed into the following linear HMTT.

\[
S \ x \rightarrow \ G x \ \text{Proj}
\]

\[
\quad \text{Proj} x_1 x_2 x_3 \rightarrow x_1
\]

\[
G x k \rightarrow \ \text{case}(x, x' . H_a x' \ k, x' . H_b x' \ k, c \ c \ c)
\]

\[
H_a x k \rightarrow \ G x (C_a k)
\]

\[
H_b x k \rightarrow \ G x (C_b k)
\]

\[
C_a k z : z_1 z_2 z_3 \rightarrow k (\text{br} (a z_2) z_3)(a z_2) z_3
\]

\[
C_b k z : z_1 z_2 z_3 \rightarrow k (\text{br} (b z_2) (b z_2))(b z_2)(b z_3)
\]

Here, the sorts of non-terminals are given by:

\[
S : i \rightarrow o
\]

\[
\text{Proj} : o \rightarrow o \rightarrow o
\]

\[
G, H_a, H_b : i \rightarrow (o \rightarrow o \rightarrow o \rightarrow o) \rightarrow o
\]

\[
C_a, C_b : (o \rightarrow o \rightarrow o \rightarrow o) \rightarrow o \rightarrow o \rightarrow o \rightarrow o
\]

**Remark C.1.** The above transformation increases the order of HMTT by 2, which may be disappointing from the viewpoint of time complexity, as the verification of order-\( n \) HMTT is \( \mu \)-EXPTIME in general (see Section 6). However, as the continuation argument is used linearly, one can optimize the model checking algorithm for recursion schemes so that the increase of the time complexity caused by the above transformation is only single-exponential.

**D. DTD of XHTML Subset Used in the Experiment**

This section shows the document type definitions used in the experiments in Section 8. Figure 4 and Figure 5 are the DTDs for Xhtml\( S_\ast \) and Xhtml\( M_\ast \) respectively.

---

**Figure 4.** DTD for Xhtml\( S_\ast \)
Figure 5. DTD for XhtmlM_