Translation of Tree-processing Programs into Stream-processing Programs based on Ordered Linear Type ∗

KOICHI KODAMA
Tokyo Institute of Technology

KOHEI SUENAGA
University of Tokyo

NAOKI KOBAYASHI
Tohoku University

Abstract

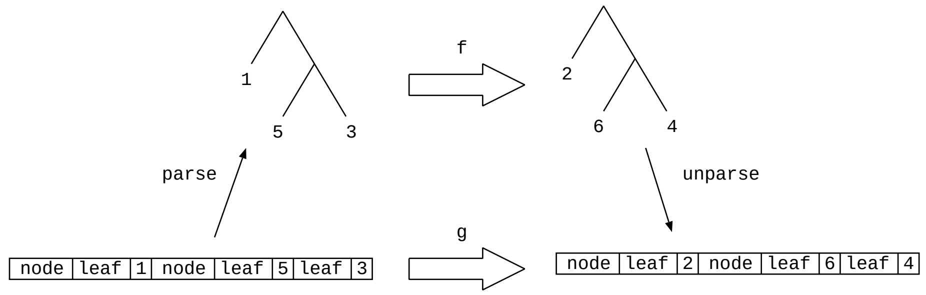
There are two ways to write a program for manipulating tree-structured data such as XML documents: One is to write a tree-processing program focusing on the logical structure of the data and the other is to write a stream-processing program focusing on the physical structure. While tree-processing programs are easier to write than stream-processing programs, tree-processing programs are less efficient in memory usage since they use trees as intermediate data. Our aim is to establish a method for automatically translating a tree-processing program to a stream-processing one in order to take the best of both worlds. We first define a programming language for processing binary trees and a type system based on ordered linear type, and show that every well-typed program can be translated to an equivalent stream-processing program. We then extend the language and the type system to deal with XML documents. We have implemented an XML stream processor generator based on our algorithm, and obtained promising experimental results.

1 Introduction

There are two ways to write a program for manipulating tree-structured data such as XML documents (Bray *et al.*, 2000): One is to write a tree-processing program focusing on the logical structure of the data and the other is to write a stream-processing program focusing on the physical structure. For example, as for XML processing, DOM (Document Object Model) API and programming language XDuce (Hosoya & Pierce, 2003) are used for tree-processing, while SAX (Simple API for XML) is for stream-processing.

Figure 1 illustrates what tree-processing and stream-processing programs look like for the case of binary trees. The tree-processing program f takes a binary

∗ This article is a revised and extended version of the paper presented in The Second ASIAN Symposium on Programming Languages and Systems (APLAS 2004).



Tree-processing program f:
$\textbf{fix}\ f.\lambda t.(\textbf{case}\ t\ \textbf{of}\ \textbf{leaf}\ x \Rightarrow \textbf{leaf}\ (x+1) \mid \textbf{node}\ x_1\ x_2 \Rightarrow \textbf{node}\ (f\ x_1)\ (f\ x_2))$
Stream-processing program g:
$\textbf{fix}\ g.\lambda t.(\textbf{case}\ \textbf{read}()\ \textbf{of}\ \textbf{leaf} \ \Rightarrow\ \textbf{let}\ x = \textbf{read}()\ \textbf{in}\ \textbf{write}(\textbf{leaf}); \textbf{write}(x+1)$
$\mid\ \textbf{node}\ \Rightarrow\ \textbf{write}(\textbf{node}); g\ (); g\ ())$

Fig. 1. Tree-processing and stream-processing.

tree t as an input, and performs case analysis on t. If t is a leaf, it increments the value of the leaf. If t is a branch, f recursively processes the left and right subtrees. If actual tree data are represented as a sequence of tokens (as is often the case for XML documents), f must be combined with the function *parse* for parsing the input sequence, and the function *unparse* for unparsing the result tree into the output sequence, as shown in the figure. The stream-processing program g directly reads/writes data from/to streams. It reads an element from the input stream using the **read** primitive and performs case-analysis on the element. If the input is the **leaf** tag, g outputs **leaf** to the output stream with the **write** primitive, reads another element, adds 1 to it, and outputs it. If the input is the **node** tag, g outputs **node** to the output stream and recursively calls the function g twice with the argument ().

Both of the approaches explained above have advantages and disadvantages. Tree-processing programs are written based on the logical structure of data, so that it is easier to write, read, and manipulate (e.g., apply program transformation like deforestation (Wadler, 1988)) than stream-processing programs. On the other hand, stream-processing programs have their own advantage that intermediate tree structures are not needed, so that they often run faster than the corresponding tree-processing programs if input/output trees are physically represented as streams, as in the case of XML.

The goal of the present paper is to achieve the best of both approaches, by allowing a programmer to write a tree-processing program and automatically translating the program to an equivalent stream-processing program. To clarify the essence, we first use a λ-calculus with primitives on binary trees, and show how the translation works. We then extend the language for XML processing.

The key observation is that: (1) stream processing is most effective *when trees are traversed and constructed from left to right in the depth-first preorder* and (2) in that case, we can obtain from the tree-processing program the corresponding stream-

Tree-processing program:
$\mathbf{fix}\ f.\lambda t.(\mathbf{case}\ t\ \mathbf{of\ leaf}\ x \Rightarrow \mathbf{leaf}\ x \mid \mathbf{node}\ x_1\ x_2 \Rightarrow \mathbf{node}\ (f\ x_2)\ (f\ x_1))$

Fig. 2. A program that swaps children of every node.

processing program simply by replacing case analysis on an input tree with case analysis on input tokens, and replacing tree constructions with stream outputs. In fact, the stream-processing program in Figure 1, which satisfies the above criterion, is obtained from the tree-processing program in that way.

In order to check that a program satisfies the criterion, we use the idea of ordered linear types (Petersen *et al.*, 2003; Polakow, 2001). Ordered linear types, which are linear types (Baker, 1992; N.Turner *et al.*, 1995) extended with order constraints, describe not only how often but also *in which order* data is used. Our type system designed based on the ordered linear types guarantees that a well-typed program traverses and constructs trees from left to right and in the depth-first preorder. Thus, every well-typed program can be translated to an equivalent stream-processing program. The tree-processing program f in Figure 1 is well-typed in our type system, so that it can automatically be translated to the stream-processing program g. On the other hand, the program in Figure 2 is not well-typed in our type system since it accesses the right sub-tree of an input before accessing the left sub-tree. In fact, we would obtain a wrong stream-processing program if we simply applied the above-mentioned translation to the program in Figure 2.

Our contributions can be summarized as follows.

- Formalization of the type system and the translation algorithm mentioned above for an extension of λ-calculus with binary trees.
- Proof of correctness of the translation algorithm.
- Extension of the type system and the translation algorithm for XML documents.
- Implementation and experiments. We implemented the type checking algorithm and the translation algorithm for XML documents. With that implementation, we performed experiments using four micro-benchmarks in XSLTMark (DataPower Technology, 2001) and one our own micro-benchmark.

The rest of the paper is organized as follows: To clarify the essence, we first focus on a minimal calculus which deals with only binary trees in Sections 2–4. In Section 2, we define the source language and the target language of the translation. We define an ordered linear type system of the source language in Section 3. As mentioned above, the type system guarantees that every node of input trees is accessed exactly once in left-to-right depth-first preorder. Section 4 presents a translation algorithm, proves its correctness and discusses the improvement gained by the translation. The minimal caclulus is not so expressive; especially, one can only write a program that does not store input/output trees on memory at all. Section 5 describes an extension that allows selective buffering of input/output trees. Section 6 shows how our framework can be applied to XML documents. Section 7 reports implementation and experiment. After discussing related work in Section 8, we conclude in Section 9.

Terms, values and evaluation contexts:

$$
\begin{array}{lcl}
M \text{ (terms)} & ::= & i \mid \lambda x.M \mid x \mid M_1\ M_2 \mid M_1 + M_2 \mid \mathbf{fix}\ f.M \\
 & \mid & \mathbf{leaf}\ M \mid \mathbf{node}\ M_1\ M_2 \\
 & \mid & (\mathbf{case}\ M\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2) \\
V \text{ (tree values)} & ::= & \mathbf{leaf}\ i \mid \mathbf{node}\ V_1\ V_2 \\
W \text{ (values)} & ::= & i \mid \lambda x.M \mid V \\
E_s \text{ (evaluation contexts)} & ::= & [\,] \mid E_s\ M \mid (\lambda x.M)\ E_s \mid E_s + M \mid i + E_s \\
 & \mid & \mathbf{leaf}\ E_s \mid \mathbf{node}\ E_s\ M \mid \mathbf{node}\ V\ E_s \\
 & \mid & (\mathbf{case}\ E_s\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2)
\end{array}
$$

Reduction rules:

$$E_s[i_1 + i_2] \to E_s[plus(i_1, i_2)] \qquad \text{(Es-Plus)}$$

$$E_s[(\lambda x.M)W] \to E_s[[W/x]M] \qquad \text{(Es-App)}$$

$$E_s[\mathbf{fix}\ f.M] \to E_s[[\mathbf{fix}\ f.M/f]M] \qquad \text{(Es-Fix)}$$

$$E_s[\mathbf{case}\ \mathbf{leaf}\ i\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2] \to E_s[[i/x]M_1] \qquad \text{(Es-Case1)}$$

$$E_s[\mathbf{case}\ \mathbf{node}\ V_1\ V_2\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2] \to E_s[[V_1/x_1, V_2/x_2]M_2] \qquad \text{(Es-Case2)}$$

Fig. 3. The syntax, evaluation context and reduction rules of the source language. $plus(i_1, i_2)$ is the sum of i_1 and i_2.

2 Language Definitions

We define the source and target languages in this section. The source language is a call-by-value functional language with primitives for manipulating binary trees. The target language is a call-by-value, impure functional language that uses imperative streams for input and output.

2.1 Source Language

The syntax and operational semantics of the source language is summarized in Figure 3.

The meta-variables x and i range over the sets of variables and integers respectively. The meta-variable W ranges over the set of values, which consists of integers i, lambda-abstractions $\lambda x.M$, and binary-trees V. A binary tree V is either a leaf labeled with an integer or a tree with two children. ($\mathbf{case}\ M\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2$) performs case analysis on a tree. If M is a leaf, x is bound to its label and M_1 is evaluated. Otherwise, x_1 and x_2 are bound to the left and right children respectively and M_2 is evaluated. $\mathbf{fix}\ f.M$ is a recursive function that satisfies $f = M$. Bound and free variables are defined as usual. We assume that α-conversion is implicitly applied so that bound variables are always different from each other and free variables.

We write $\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2$ for $(\lambda x.M_2)\ M_1$. Especially, if M_2 contains no free occurrence of x, we write $M_1; M_2$ for it.

Terms, values and evaluation contexts:

$$
\begin{array}{rcl}
e \text{ (terms)} & ::= & v \mid x \mid e_1\ e_2 \mid e_1 + e_2 \mid \textbf{fix}\ f.e \\
& \mid & \textbf{read}\ e \mid \textbf{write}\ e \\
& \mid & (\textbf{case}\ e\ \textbf{of leaf} \Rightarrow e_1 \mid \textbf{node} \Rightarrow e_2) \\
v \text{ (values)} & ::= & i \mid \textbf{leaf} \mid \textbf{node} \mid \lambda x.e \mid () \\
E_t \text{ (evaluation contexts)} & ::= & [\,] \mid E_t\ e \mid (\lambda x.e)\ E_t \mid E_t + e \mid i + E_t \\
& \mid & \textbf{read}\ E_t \mid \textbf{write}\ E_t \\
& \mid & (\textbf{case}\ E_t\ \textbf{of leaf} \Rightarrow e_1 \mid \textbf{node} \Rightarrow e_2)
\end{array}
$$

Reduction rules:

$$
\begin{array}{rr}
(E_t[v_1 + v_2], S_i, S_o) \rightarrow (E_t[plus(v_1, v_2)], S_i, S_o) & (\text{ET-PLUS}) \\
(E_t[(\lambda x.M)v], S_i, S_o) \rightarrow (E_t[[v/x]M], S_i, S_o) & (\text{ET-APP}) \\
(E_t[\textbf{fix}\ f.e], S_i, S_o) \rightarrow (E_t[[\textbf{fix}\ f.e/f]e], S_i, S_o) & (\text{ET-FIX}) \\
(E_t[\textbf{read}()], v; S_i, S_o) \rightarrow (E_t[v], S_i, S_o) & (\text{ET-READ}) \\
(E_t[\textbf{write}\ v], S_i, S_o) \rightarrow (E_t[()], S_i, S_o; v) & (\text{ET-WRITE}) \\
(\text{when } v \text{ is an integer, } \textbf{leaf} \text{ or } \textbf{node}) & \\
(E_t[\textbf{case leaf of leaf} \Rightarrow e_1 \mid \textbf{node} \Rightarrow e_2], S_i, S_o) \rightarrow (E_t[e_1], S_i, S_o) & (\text{ET-CASE1}) \\
(E_t[\textbf{case node of leaf} \Rightarrow e_1 \mid \textbf{node} \Rightarrow e_2], S_i, S_o) \rightarrow (E_t[e_2], S_i, S_o) & (\text{ET-CASE2})
\end{array}
$$

Fig. 4. The reduction rules of the target language.

A source program:
$\textbf{fix}\ \mathit{sumtree}.\lambda t.(\textbf{case}\ t\ \textbf{of leaf}\ x \Rightarrow x \mid \textbf{node}\ x_1\ x_2 \Rightarrow (\mathit{sumtree}\ x_1) + (\mathit{sumtree}\ x_2))$
A target program:
$\textbf{fix}\ \mathit{sumtree}.\lambda t.(\textbf{case read}()\ \textbf{of leaf} \Rightarrow \textbf{read}() \mid \textbf{node} \Rightarrow \mathit{sumtree}\ () + \mathit{sumtree}\ ())$

Fig. 5. Programs that calculate the sum of leaf elements of a binary tree.

2.2 Target Language

The syntax and operational semantics of the target language is summarized in Figure 4. A stream, represented by the meta variable S, is a sequence consisting of **leaf**, **node** and integers. We write $\emptyset$ for the empty sequence and write $S_1; S_2$ for the concatenation of the sequences S_1 and S_2.

read is a primitive for reading a token (**leaf**, **node**, or an integer) from the input stream. **write** is a primitive for writing a value to the output stream. The term ($\textbf{case}\ e\ \textbf{of leaf} \Rightarrow e_1 \mid \textbf{node} \Rightarrow e_2$) performs a case analysis on the value of e. If e evaluates to **leaf**, e_1 is evaluated and if e evaluates to **node**, e_2 is evaluated. $\textbf{fix}\ f.e$ is a recursive function that satisfies $f = e$. Bound and free variables are defined as usual.

We write $\textbf{let}\ x = e_1\ \textbf{in}\ e_2$ for $(\lambda x.e_2)\ e_1$. Especially, if e_2 does not contain x as a free variable, we write $e_1; e_2$ for it.

Figure 5 shows programs that take a tree as an input and calculate the sum of leaf elements. The source program takes a tree t as an argument of the function, and performs a case analysis on t. If t is a leaf, the program binds x to the element and returns it. If t is a branch node, the program recursively applies f to the left and right children and returns the sum of the results. The target program reads a tree (as a sequence of tokens) from the input stream, performs a case analysis on tokens, and returns the sum of leaf elements. Here, we assume that the input stream represents a valid tree. If the input stream is in a wrong format (e.g., when the stream is $\mathbf{node}; 1; 2$), the execution gets stuck.

3 Type System

In this section, we present a type system of the source language, which guarantees that a well-typed program reads every node of an input tree exactly once from left to right in the left-to-right depth-first preorder. Thanks to this guarantee, any well-typed program can be translated to an equivalent, stream-processing program without changing the structure of the program, as shown in the next section. To enforce the correct access order on input trees, we use ordered linear types (Polakow, 2001; Petersen *et al.*, 2003).

3.1 Type and Type Environment

Definition 3.1 (*Type*)
The set of *types*, ranged over by τ, is defined by:

$$\begin{array}{rcl} \tau \text{ (type)} & ::= & \mathbf{Int} \mid \mathbf{Tree}^d \mid \tau_1 \to \tau_2 \\ d \text{ (mode)} & ::= & - \mid +. \end{array}$$

Int is the type of integers. For a technical reason, we distinguish between input trees and output trees by types. We write $\mathbf{Tree}^-$ for the type of input trees, and write $\mathbf{Tree}^+$ for the type of output trees. $\tau_1 \to \tau_2$ is the type of functions from τ_1 to τ_2.

We introduce two kinds of type environments for our type system: ordered linear type environments and (non-ordered) type environments.

Definition 3.2 (*Ordered Linear Type Environment*)
An *ordered linear type environment*, ranged over by the meta-variable Δ, is a sequence of the form $x_1 : \mathbf{Tree}^-, \ldots, x_n : \mathbf{Tree}^-$, where $x_1, \ldots, x_n$ are different from each other. We write Δ_1, Δ_2 for the concatenation of Δ_1 and Δ_2.

An ordered linear type environment $x_1 : \mathbf{Tree}^-, \ldots, x_n : \mathbf{Tree}^-$ specifies not only that $x_1, \ldots, x_n$ are bound to trees, but also that each of $x_1, \ldots, x_n$ must be accessed exactly once in this order and that each of the trees bound to $x_1, \ldots, x_n$ must be accessed in the left-to-right, depth-first preorder.

Definition 3.3 (*Non-Ordered Type Environment*)
A *(non-ordered) type environment*, ranged over by a meta variable Γ, is a set of the form $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ where $x_1, \ldots, x_n$ are different from each other and $\{\tau_1, \ldots, \tau_n\}$ does not contain $\mathbf{Tree}^d$.

Note that a non-ordered type environment must not contain variables of tree types. $\mathbf{Tree}^-$ is excluded since input trees must be accessed in the specific order.

$\mathbf{Tree}^{+}$ is excluded in order to forbid output trees from being bound to variables. For example, we will exclude a program like **let** $x_1 = t_1$ **in let** $x_2 = t_2$ **in node** $x_1\ x_2$ when t_1 and t_2 have type $\mathbf{Tree}^{+}$. This restriction is convenient for ensuring that trees are constructed in the specific (from left to right, and in the depth-first pre-) order. Note that a non-ordered type environment can contain types that use tree types, like $\mathbf{Tree}^{-} \to \mathbf{Tree}^{+}$. See the first example in Section 3.2.

3.2 Type Judgment

A type judgement is of the form $\Gamma \mid \Delta \vdash M : \tau$, where Γ is a non-ordered type environment, Δ is an ordered linear type environment and Γ and Δ do not share the same variables. The judgment means "If M evaluates to a value under an environment described by Γ and Δ, the value has type τ and the variables in Δ are accessed in the order specified by Δ." For example, if $\Gamma = \{f : \mathbf{Tree}^{-} \to \mathbf{Tree}^{+}\}$ and $\Delta = x_1 : \mathbf{Tree}^{-}, x_2 : \mathbf{Tree}^{-}$,

$$\Gamma \mid \Delta \vdash \mathbf{node}\ (f\ x_1)\ (f\ x_2) : \mathbf{Tree}^{+}$$

holds, while

$$\Gamma \mid \Delta \vdash \mathbf{node}\ (f\ x_2)\ (f\ x_1) : \mathbf{Tree}^{+}$$

does not. The latter program violates the restriction specified by Δ that x_1 and x_2 must be accessed in this order.

$\Gamma \mid \Delta \vdash M : \tau$ is the least relation that is closed under the rules in Figure 6. Note that in the typing rules, there is an implicit assumption that the conclustion must be a well-formed judgment in order for a rule to be applied. Therefore, in T-App, T-Plus and T-Node, it is implicitly assumed that the variables in Δ_1 and Δ_2 are disjoint.

T-Var1, T-Var2 and T-Int are the rules for variables and integer constants. As in ordinary linear type systems, these rules prohibit variables that do not occur in a term from occurring in the ordered linear type environment. (In other words, weakening is not allowed on an ordered linear type environment.) That restriction is necessary to guarantee that each variable in an ordered linear type environment is accessed exactly once.

T-Abs1 and T-Abs2 are rules for lambda abstraction. Note that the ordered type environments of the conclusions of these rules must be empty. This restriction prevents input trees from being stored in function closures. That makes it easy to enforce the access order on input trees. For example, without this restriction, the function

$$\lambda t.\mathbf{let}\ g = \lambda f.(f\ t)\ \mathbf{in}\ (g\ \mathit{sumtree}) + (g\ \mathit{sumtree})$$

would be well-typed where *sumtree* is the function given in Figure 5. However, when a tree is passed to this function, its nodes are accessed twice because the function g is called twice. The program above is actually rejected by our type system since the closure $\lambda f.(f\ t)$ is not well-typed due to the restriction of T-Abs2.[1]

[1] We can relax the restriction by controlling usage of not only trees but also functions, as in

$$\Gamma \mid x : \textbf{Tree}^- \vdash x : \textbf{Tree}^- \qquad \text{(T-VAR1)}$$

$$\Gamma, x : \tau \mid \emptyset \vdash x : \tau \qquad \text{(T-VAR2)}$$

$$\Gamma \mid \emptyset \vdash i : \textbf{Int} \qquad \text{(T-INT)}$$

$$\frac{\Gamma \mid x : \textbf{Tree}^- \vdash M : \tau}{\Gamma \mid \emptyset \vdash \lambda x.M : \textbf{Tree}^- \to \tau} \qquad \text{(T-ABS1)}$$

$$\frac{\Gamma, x : \tau_1 \mid \emptyset \vdash M : \tau_2}{\Gamma \mid \emptyset \vdash \lambda x.M : \tau_1 \to \tau_2} \qquad \text{(T-ABS2)}$$

$$\frac{\Gamma \mid \Delta_1 \vdash M_1 : \tau_2 \to \tau_1 \qquad \Gamma \mid \Delta_2 \vdash M_2 : \tau_2}{\Gamma \mid \Delta_1, \Delta_2 \vdash M_1 M_2 : \tau_1} \qquad \text{(T-APP)}$$

$$\frac{\Gamma \mid \Delta_1 \vdash M_1 : \textbf{Int} \qquad \Gamma \mid \Delta_2 \vdash M_2 : \textbf{Int}}{\Gamma \mid \Delta_1, \Delta_2 \vdash M_1 + M_2 : \textbf{Int}} \qquad \text{(T-PLUS)}$$

$$\frac{\Gamma, f : \tau_1 \to \tau_2 \mid \emptyset \vdash M : \tau_1 \to \tau_2}{\Gamma \mid \emptyset \vdash \textbf{fix } f.M : \tau_1 \to \tau_2} \qquad \text{(T-FIX)}$$

$$\frac{\Gamma \mid \Delta \vdash M : \textbf{Int}}{\Gamma \mid \Delta \vdash \textbf{leaf } M : \textbf{Tree}^+} \qquad \text{(T-LEAF)}$$

$$\frac{\Gamma \mid \Delta_1 \vdash M_1 : \textbf{Tree}^+ \qquad \Gamma \mid \Delta_2 \vdash M_2 : \textbf{Tree}^+}{\Gamma \mid \Delta_1, \Delta_2 \vdash \textbf{node } M_1\ M_2 : \textbf{Tree}^+} \qquad \text{(T-NODE)}$$

$$\frac{\begin{array}{c}\Gamma \mid \Delta_1 \vdash M : \textbf{Tree}^- \qquad \Gamma, x : \textbf{Int} \mid \Delta_2 \vdash M_1 : \tau \\ \Gamma \mid x_1 : \textbf{Tree}^-, x_2 : \textbf{Tree}^-, \Delta_2 \vdash M_2 : \tau\end{array}}{\Gamma \mid \Delta_1, \Delta_2 \vdash \textbf{case } M \textbf{ of leaf } x \Rightarrow M_1 \mid \textbf{node } x_1\ x_2 \Rightarrow M_2 : \tau} \qquad \text{(T-CASE)}$$

Fig. 6. Typing Rules.

T-APP is the rule for function application. The ordered linear type environments of M_1 and M_2, Δ_1 and Δ_2 respectively, are concatenated in this order because when $M_1\ M_2$ is evaluated, (1) M_1 is first evaluated, (2) M_2 is then evaluated, and (3) M_1 is finally applied to M_2. In the first step, the variables in Δ_1 are accessed in the order specified by Δ_1. In the second and third steps, the variables in Δ_2 are accessed in the order specified by Δ_2. On the other hand, because there is no restriction on usage of the variables in a non-ordered type environment, the same type environment (Γ) is used for both subterms.

T-LEAF and T-NODE are rules for tree construction. We concatenate the ordered type environments of M_1 and M_2, Δ_1 and Δ_2, in this order as we did in T-APP. Also as in the rule T-APP, Δ_1 and Δ_2 cannot share the same variables in T-NODE.

T-CASE is the rule for case expressions. If M matches **node** $x_1\ x_2$, subtrees x_1 and x_2 have to be accessed in this order after that. This restriction is expressed by $x_1 : \textbf{Tree}^-, x_2 : \textbf{Tree}^-, \Delta_2$, the ordered linear type environment of M_2.

T-FIX is the rule for recursion. Note that the ordered type environment must be empty as in T-ABS2.

The program in Figure 1 is typed as shown in Figure 7. On the other hand, the

the resource usage analysis (Igarashi & Kobayashi, 2002). The resulting type system would, however, become very complex.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \mid t : \mathbf{Tree}^- \vdash t : \mathbf{Tree}^-} \qquad \dfrac{\vdots}{\Gamma' \mid \emptyset \vdash M : \mathbf{Tree}^+} \qquad \dfrac{\vdots}{\Gamma \mid \Delta \vdash \mathbf{node}\ (f\ x_1)\ (f\ x_2) : \mathbf{Tree}^+}}{\Gamma \mid t : \mathbf{Tree}^- \vdash \mathbf{case}\ t\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M \mid \mathbf{node}\ x_1\ x_2 \Rightarrow \mathbf{node}\ (f\ x_1)\ (f\ x_2) : \mathbf{Tree}^+}}{\Gamma \mid \emptyset \vdash \lambda t.\mathbf{case}\ t\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M \mid \mathbf{node}\ x_1\ x_2 \Rightarrow \mathbf{node}\ (f\ x_1)\ (f\ x_2) : \mathbf{Tree}^- \to \mathbf{Tree}^+}}{\emptyset \mid \emptyset \vdash \mathbf{fix}\ f.\lambda t.\mathbf{case}\ t\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M \mid \mathbf{node}\ x_1\ x_2 \Rightarrow \mathbf{node}\ (f\ x_1)\ (f\ x_2) : \mathbf{Tree}^- \to \mathbf{Tree}^+}$$

Fig. 7. An example of typing derivation. $\Gamma = \{f : \mathbf{Tree}^- \to \mathbf{Tree}^+\}$, $M = \mathbf{leaf}\ (x+1)$, $\Gamma' = \{f : \mathbf{Tree}^- \to \mathbf{Tree}^+, x : \mathbf{Int}\}$ and $\Delta = x_1 : \mathbf{Tree}^-, x_2 : \mathbf{Tree}^-$.

program in Figure 2 is ill-typed: $\Gamma \mid x_1 : \mathbf{Tree}^-, x_2 : \mathbf{Tree}^- \vdash \mathbf{node}\ (f\ x_2)\ (f\ x_1) : \mathbf{Tree}^+$ must hold for the program to be well-typed, but it cannot be derived by using T-Node.

3.3 Examples of Programs

Figure 8 shows more examples of source programs. The first and second programs (or the catamorphism (Meijer *et al.*, 1991)) apply the same operation on every node of the input tree. (The return value of the function *tree_fold* cannot, however, be a tree because a return value of *tree_fold* is passed to g. Note that functions cannot take a value of type $\mathbf{Tree}^+$ as argument because binding variables to $\mathbf{Tree}^+$ values is forbidden in our type system as we mentioned in Section 3.1.) The third program returns a tree obtained by incrementing all the leaf values of odd depth by one. In this way, one can also write functions that process nodes in a non-uniform manner. The fourth program is an example of a well-typed program in which variables of type $\mathbf{Tree}^-$ (y_1, y_2 and x_2) are not bound to sibling nodes in the input tree.

The fifth program in Figure 8 takes a tree as an input and returns the right subtree. Due to the restriction imposed by the type system, the program uses sub-functions *copy_tree* and *skip_tree* for explicitly copying and skipping trees.[2] (See Section 9 for a method for automatically inserting those functions.)

The last program in Figure 8 takes a tree as an input, and returns a tree consisting of a copy of the input tree and the number of leaves of the input. Here, we assume that the source language has been extended with ML-style reference cells. The type system can be easily extended: The only restriction is that input/output trees cannot be stored in reference cells.

Remark 3.1
The reader may think that our type system is too restrictive. The following two functions, which are obtained by modifying the first and third programs in Figure 8,

[2] Due to the restriction that lambda abstractions cannot contain variables of type $\mathbf{Tree}^d$, we need to introduce **let** expression as a primitive and extend typing rules with the following rule:

$$\frac{\Gamma \mid \Delta_1 \vdash M_1 : \tau' \qquad \Gamma \cup \{x : \tau'\} \mid \Delta_2 \vdash M_2 : \tau \qquad \tau' \neq \mathbf{Tree}^d}{\Gamma \mid \Delta_1, \Delta_2 \vdash \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : \tau.} \quad \text{(T-Let)}$$

```
fix tree_map.λf.λt.
    case t of
        leaf x  ⇒  leaf (f x)
      | node x1 x2  ⇒ node (tree_map f x1) (tree_map f x2))

fix tree_fold.λf.λg.λt.
    case t of
        leaf n  ⇒  (f n)
      | node t1 t2  ⇒  (g (tree_fold f g t1) (tree_fold f g t2))

fix inc_alt.
    let inc_alt_even =
        λt.case t of
            Leafx ⇒ leaf x + 1
          | node x1 x2 ⇒ node (inc_alt x1) (inc_alt x2)
    in
        λt.case t of
            Leafx ⇒ leaf x
          | node x1 x2 ⇒ node (inc_alt_even x1) (inc_alt_even x2)

fix inc_left.λt.
    case t of
        leaf x ⇒ leaf x
      | node x1 x2 ⇒
        node
            (case x1 of leaf y ⇒ leaf (y + 1)
                | node y1 y2 ⇒ node (inc_left y1) (inc_left y2))
            (inc_left x2)

let copy_tree =
    fix copy_tree.λt.
        case t of
            leaf x  ⇒  leaf x
          | node x1 x2  ⇒  node (copy_tree x1) (copy_tree x2)) in
let skip_tree =
    fix skip_tree.λt.
        case t of
            leaf x  ⇒  0
          | node x1 x2  ⇒  (skip_tree x1); (copy_tree x2) in
    λt.(case t of leaf x ⇒ leaf x | node x1 x2 ⇒ (skip_tree x1); (copy_tree x2))

let copy_and_count_aux = fix f.λr.λt.
        case t of
            leaf x  ⇒  (r :=!r + 1; leaf x)
          | node x1 x2  ⇒  node (f r x1) (f r x2)) in
let copy_and_count =
    λt.let x = ref 0 in node (copy_and_count_aux x t) (leaf !x)
```

Fig. 8. Program Examples.

$$\boxed{\textbf{fix } f.\lambda t.\textbf{case } t \textbf{ of leaf } x \Rightarrow \textbf{leaf } x \mid \textbf{node } x_1\ x_2 \Rightarrow (\lambda t'.\textbf{node } (f\ t')\ (f\ x_2))\ x_1}$$

Fig. 9. Program that is not typed due to the restriction on closures though the access order is correct.

highlight the main limitations of our type system.

$$
\begin{array}{l}
\textbf{fix } \mathit{tree\_map}'.\lambda t.\lambda f. \\
\quad \textbf{case } t \textbf{ of} \\
\qquad \textbf{leaf } x \ \Rightarrow\ \textbf{leaf } (f\ x) \\
\quad \mid\ \textbf{node } x_1\ x_2\ \Rightarrow \textbf{node } (\mathit{tree\_map}'\ x_1\ f)\ (\mathit{tree\_map}'\ x_2\ f))
\end{array}
$$

$$
\begin{array}{l}
\textbf{fix } \mathit{tree\_map}''.\lambda f.\lambda t. \\
\quad \textbf{case } t \textbf{ of} \\
\qquad \textbf{leaf } x \ \Rightarrow\ \textbf{leaf } (f\ x) \\
\quad \mid\ \textbf{node } x_1\ x_2\ \Rightarrow \textbf{let } t_1 = \mathit{tree\_map}''\ f\ x_1 \textbf{ in node } t_1\ (\mathit{tree\_map}''\ f\ x_2))
\end{array}
$$

They are operationally equivalent to the first program in Figure 8, but are *not* well-typed. The sub-expression $\lambda f.\textbf{case } t \textbf{ of } \cdots$ of the first program above violates the rule T-ABS2, which says that a variable of type $\textbf{Tree}^-$ must not occur free in a function. The sub-expression $\textbf{let } t_1 = \mathit{tree\_map}''\ f\ x_1 \textbf{ in } \cdots$ violates the condition that no variable can be bound to a tree of type $\textbf{Tree}^+$.

It would be possible to remove the limitations to some extent by introducing a more complex type system.[3] We do not do so, however. We believe that in many cases, the above problems can be avoided by adding a simple pre-processing phase; The first problem can often be avoided by uncurrying transformation, and the second problem can often be avoided by inlining arguments of the **node** contructor. Thus, it is unclear how much benefit is obtained in practice by introducing a more complex type system. Moreover, the simplicity of the type system is beneficial for various extensions of the framework, especially for automatic insertion of buffering primitives discussed elsewhere (Suenaga *et al.*, 2005).

3.4 Type Checking Algorithm

We sketch a type checking algorithm that takes a type-annotated term M and a non-ordered type environment Γ as input, and outputs τ and Δ such that $\Gamma \mid \Delta \vdash M : \tau$, or reports failure if such τ or Δ does not exist. Here, by a type-annotated term, we mean a term whose bound variables are annotated with types.

The rules in Figure 6 can be interpreted as such an algorithm. Consider Γ and M in each rule as input and Δ and τ in each rule as output. Then, read each rule from bottom to top. For example, T-APP can be read as follows. For input $(\Gamma, M_1\ M_2)$, do the following:

[3] For the second problem, we also have to modify the translation algorithm discussed in the next section.

$$
\begin{array}{l}
\mathcal{A}(x) = x \\
\mathcal{A}(i) = i \\
\mathcal{A}(\lambda x.M) = \lambda x.\mathcal{A}(M) \\
\mathcal{A}(M_1 M_2) = \mathcal{A}(M_1)\ \mathcal{A}(M_2) \\
\mathcal{A}(M_1 + M_2) = \mathcal{A}(M_1) + \mathcal{A}(M_2) \\
\mathcal{A}(\mathbf{fix}\ f.M) = \mathbf{fix}\ f.\mathcal{A}(M) \\
\mathcal{A}(\mathbf{leaf}\ M) = \mathbf{write}(\mathbf{leaf}); \mathbf{write}(\mathcal{A}(M)) \\
\mathcal{A}(\mathbf{node}\ M_1\ M_2) = \mathbf{write}(\mathbf{node}); \mathcal{A}(M_1); \mathcal{A}(M_2) \\
\mathcal{A}(\mathbf{case}\ M\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2) = \\
\qquad\qquad\qquad \mathbf{case}\ \mathcal{A}(M); \mathbf{read}()\ \mathbf{of}\ \mathbf{leaf} \Rightarrow \mathbf{let}\ x = \mathbf{read}()\ \mathbf{in}\ \mathcal{A}(M_1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}(M_2)
\end{array}
$$

Fig. 10. Translation Algorithm.

1. Execute the algorithm with input (Γ, M_1) and obtain the result $(\Delta_1, \tau_2 \to \tau_1)$.
2. Execute the algorithm with input (Γ, M_2) and obtain the result (Δ_2, τ_2').
3. Check (Δ_1, Δ_2) is well-formed (i.e., does not share the same variables) and $\tau_2 = \tau_2'$.
4. Output $((\Delta_1, \Delta_2), \tau_1)$.

4 Translation Algorithm

Definition 4.1 (*Translation Algorithm*)
The translation function $\mathcal{A}$ from a source program into the target program is defined in Figure 10.

$\mathcal{A}$ maps a source program to a target program, preserving the structure of the source program and replacing operations on trees with operations on streams. For example, the tree construction primitives ($\mathbf{leaf}\ M$ and $\mathbf{node}\ M_1\ M_2$) are translated into writing operations to the output stream. The pattern matching primitive ($\mathbf{case}\ M\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2$) is translated into the case analysis on the head of the input stream. As we have observed in Section 1, this simple translation works because the type system guarantees that well-typed programs access an input tree in left-to-right, depth-first preorder.

In the rest of this section, we will prove the correctness of the translation algorithm $\mathcal{A}$ and discuss the efficiency of the translated program.

4.1 Correctness of Translation Algorithm

The correctness of the translation algorithm $\mathcal{A}$ is stated as follows.

Definition 4.2
A function $[\![\cdot]\!]$ from the set of trees to the set of streams is defined by:

$$
\begin{array}{rcl}
[\![\mathbf{leaf}\ i]\!] & = & \mathbf{leaf}; i \\
[\![\mathbf{node}\ V_1\ V_2]\!] & = & \mathbf{node}; [\![V_1]\!]; [\![V_2]\!].
\end{array}
$$

Theorem 4.1 (*Correctness of Translation*)

If $\emptyset \mid \emptyset \vdash M : \mathbf{Tree}^{-} \to \tau$ and τ is **Int** or $\mathbf{Tree}^{+}$, the following properties hold for any tree value V:

(i) $M\ V \ \to^{*} i$ if and only if $(\mathcal{A}(M)(), [\![V]\!], \emptyset) \to^{*} (i, \emptyset, \emptyset)$
(ii) $M\ V \ \to^{*} V'$ if and only if $(\mathcal{A}(M)(), [\![V]\!], \emptyset) \to^{*} ((), \emptyset, [\![V']\!])$.

The above theorem means that a source program and the corresponding target program evaluate to the same value. The clause (i) is for the case where the result is an integer, and (ii) is for the case where the result is a tree.

We give an outline of the proof of Theorem 4.1 below. The basic idea of the proof is to show a correspondence between reduction steps of a source program and those of the target program. However, the reduction semantics given in Section 2 is not convenient for showing the correspondence because the target language semantics imposes a restriction on access to stream input/outputs, while the source language semantics in Section 2 does not. So, we introduce another semantics of the source language that has the corresponding order restriction on environments holding input/output trees, and prove: (1) for well-typed programs, the new semantics is equivalent to the one in Section 2 (Corollary 4.3 below), and (2) evaluation of the source program based on the new semantics agrees with evaluation of the corresponding target program (Theorem 4.4 below). First, we define the new operational semantics of the source language. The semantics takes the access order of input trees into account.

Definition 4.3 (*Ordered Environment*)
An ordered environment is a sequence of the form $x_1 \mapsto V_1, \ldots, x_n \mapsto V_n$, where $x_1, \ldots, x_n$ are distinct from each other.

We use a meta-variable δ to represent an ordered environment. Given an ordered environment $x_1 \mapsto V_1, \ldots, x_n \mapsto V_n$, a program must access variables $x_1, \ldots, x_n$ in this order.

Definition 4.4 (*New Reduction Semantics*)
The reduction relation $(M, \delta) \to (M', \delta')$ is the least relation that satisfies the rules in Figure 11.

The meta-variable U in Figure 11 ranges over the set of variables (bound to tree values) and non-tree values.

The differences between the new reduction semantics above and the original one in Section 2 are: (1) input trees are substituted in the original semantics while they are held in ordered environments in the new semantics (compare Es-Case2 with Es2-Case2), and (2) input trees must be accessed in the order specified by δ in the new semantics (note that variable y that is being referred to must be at the head of the ordered environment in Es2-Case1 and Es2-Case2). Thus, evaluation based on the new semantics can differ from the one in Section 2 only when the latter one succeeds while the former one gets stuck due to the restriction on access to input trees. As the following theorem (Theorem 4.2) states, that cannot happen if the program is well-typed, so that both semantics are equivalent for well-typed programs (Corollary 4.3).

$$U \quad ::= \quad x \mid i \mid \lambda x.M$$

$$(E_s[i_1 + i_2], \delta) \rightarrow (E_s[plus(i_1, i_2)], \delta) \qquad \text{(Es2-Plus)}$$

$$(E_s[(\lambda x.M)U], \delta) \rightarrow (E_s[[U/x]M], \delta) \qquad \text{(Es2-App)}$$

$$(E_s[\mathbf{case}\ y\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2], (y \mapsto \mathbf{leaf}\ i, \delta)) \rightarrow (E_s[[i/x]M_1], \delta) \qquad \text{(Es2-Case1)}$$

$$(E_s[\mathbf{case}\ y\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2], (y \mapsto \mathbf{node}\ V_1\ V_2, \delta)) \rightarrow (E_s[M_2], (x_1 \mapsto V_1, x_2 \mapsto V_2, \delta)) \qquad \text{(Es2-Case2)}$$

$$(E_s[\mathbf{fix}\ f.M], \delta) \rightarrow (E_s[[\mathbf{fix}\ f.M/f]M], \delta) \qquad \text{(Es2-Fix)}$$

Fig. 11. The new reduction semantics of the source language.

Definition 4.5
A function $\langle\langle\cdot\rangle\rangle$ from the set of ordered environments to the set of ordered linear type environments is defined by:

$$\begin{array}{rcl} \langle\langle\emptyset\rangle\rangle & = & \emptyset \\ \langle\langle x \mapsto V, \delta\rangle\rangle & = & x : \mathbf{Tree}^-, \langle\langle\delta\rangle\rangle \end{array}$$

Theorem 4.2
Suppose $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash M : \tau$. Then the following conditions hold.

- M is a value or a variable, or $(M, \delta) \rightarrow (M', \delta')$ holds for some M' and δ'.
- If $(M, \delta) \rightarrow (M', \delta')$ holds, then $\emptyset \mid \langle\langle\delta'\rangle\rangle \vdash M' : \tau$.

Proof
See Appendix A. □

Corollary 4.3
If $\emptyset \mid \emptyset \vdash M : \mathbf{Tree}^- \rightarrow \tau$ and if $\tau \in \{\mathbf{Int}, \mathbf{Tree}^+\}$, $MV \rightarrow^* W$ if and only if $(Mx, x \mapsto V) \rightarrow^* (W, \emptyset)$ for any tree value V.

Proof
See Appendix B. □

The following theorem states that the evaluation of a source program under the new rules agrees with the evaluation of the target program.

Theorem 4.4
If $\emptyset \mid \emptyset \vdash M : \mathbf{Tree}^- \rightarrow \tau$ and $\tau \in \{\mathbf{Int}, \mathbf{Tree}^+\}$ holds, the following statements hold for any tree value V.

(i) $(Mx, x \mapsto V) \rightarrow^* (i, \emptyset)$ holds if and only if $(\mathcal{A}(M)(), [\![V]\!], \emptyset) \rightarrow^* (i, \emptyset, \emptyset))$
(ii) If $(Mx, x \mapsto V) \rightarrow^* (V', \emptyset)$ holds if and only if $(\mathcal{A}(M)(), [\![V]\!], \emptyset) \rightarrow^* ((), \emptyset, [\![V']\!]))$

We hereafter give an outline of the proof of Theorem 4.4. Figure 13 illustrates the idea of the proof (for the case where the result is a tree). The relation $\sim$ (defined later in Definition 4.7) in the diagram expresses the correspondence between an evaluation state of a source program (M, δ) and a state of a target program (e, S_i, S_o). We shall show that the target program $\mathcal{A}(M)$ can always be reduced to a state corresponding to the inital state of the source program M (Lemma 4.5 below) and that reductions and the correspondence relation commute (Lemma 4.6). Those imply that the whole diagram in Figure 13 commutes, i.e., the second statement of Theorem 4.4 holds.

To define the correspondence $(M, \delta) \sim (e, S_i, S_o)$ between states, we use the following function $\langle\cdot\rangle$, which maps an ordered environment to the corresponding stream.

Definition 4.6
A function $\langle\cdot\rangle$ from the set of ordered environments to the set of streams is defined by:

$$\begin{array}{rcl} \langle\emptyset\rangle & = & \emptyset \\ \langle x \mapsto V, \delta\rangle & = & [\![V]\!]; \langle\delta\rangle \end{array}$$

Definition 4.7 (*Correspondence between States*)
The relations $(M, \delta) \sim (e, S_i, S_o)$ and $M \sim_\gamma (e, S_o)$ are the least relations closed under the rules in Figure 12.

In the figure, the meta-variable γ denotes a set of variables. $\mathbf{FV}(M)$ is the set of free variables in M. $\mathcal{A}_\gamma(M)$ is the term obtained from $\mathcal{A}(M)$ by replacing every occurrence of variables in γ with $()$. The meta-variable I represents the term that is being reduced. Note that any term M can be written as $E_s[I]$ if it is reducible.

In the relation $(M, \delta) \sim (e, S_i, S_o)$, e represents the rest of computation, S_i is the input stream, and S_o is the already generated output streams. For example, $(\mathbf{node}(\mathbf{leaf}\ 1)(\mathbf{leaf}\ (2+3)), \emptyset)$ corresponds to $(2+3, \emptyset, \mathbf{node}; \mathbf{leaf}; 1; \mathbf{leaf})$.

We explain some of the rules in Figure 12 below.

- C-Tree: A source program V represents a state where the tree V has been constructed. Thus, it corresponds to $((), [\![V]\!])$, where there is nothing to be computed and V has been written to the output stream.
- C-Node1: A source program $\mathbf{node}\ E_s[I]\ M$ represents a state where the left subtree is being computed. Thus, the rest computation of the target program is $(e; \mathcal{A}_\gamma(M))$ where e is the rest computation in $E_s[I]$, and $\mathcal{A}_\gamma(M)$ represents the computation for constructing the right subtree. The corresponding output stream is $\mathbf{node}; S_o$ because $\mathbf{node}$ represents the root of the tree being constructed, and S_o represents the part of the left subtree that has been already constructed.

Lemmas 4.5 and 4.6 below imply that the whole diagram in Figure 12 commutes, which completes the proof of Theorem 4.4.

Lemma 4.5
Suppose $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash M : \tau$. Then, there exist e and S_o that satisfy

- $(M, \delta) \sim (e, \langle\delta\rangle, S_o)$
- $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \to^* (e, \langle\delta\rangle, S_o)$

$$I \quad ::= \quad (\lambda x.M)U \mid i_1 + i_2 \mid (\textbf{case } y \textbf{ of leaf } x \Rightarrow M_1 \mid \textbf{node } x_1\ x_2 \Rightarrow M_2) \mid \textbf{fix } f.M$$

$$\frac{M \sim_{FV(M)} (e, S_o) \qquad S_i = \langle\delta\rangle}{(M,\delta) \sim (e, S_i, S_o)}$$

$$\frac{}{U \sim_\gamma (\mathcal{A}_\gamma(U), \emptyset)} \quad \text{(C-Value)}$$

$$\frac{}{V \sim_\gamma ((), [\![V]\!])} \quad \text{(C-Tree)}$$

$$\frac{}{I \sim_\gamma (\mathcal{A}_\gamma(I), \emptyset)} \quad \text{(C-Inst)}$$

$$\frac{E_s[I] \sim_\gamma (e, S_o)}{E_s[I]\ M \sim_\gamma (e\ \mathcal{A}_\gamma(M), S_o)} \quad \text{(C-App1)}$$

$$\frac{E_s[I] \sim_\gamma (e, S_o)}{(\lambda x.M)\ E_s[I] \sim_\gamma ((\lambda x.\mathcal{A}_\gamma(M))\ e, S_o)} \quad \text{(C-App2)}$$

$$\frac{E_s[I] \sim_\gamma (e, S_o)}{E_s[I] + M \sim_\gamma (e + \mathcal{A}_\gamma(M), S_o)} \quad \text{(C-Plus1)}$$

$$\frac{E_s[I] \sim_\gamma (e, S_o)}{i + E_s[I] \sim_\gamma (i + e, S_o)} \quad \text{(C-Plus2)}$$

$$\frac{E_s[I] \sim_\gamma (e, S_o)}{\textbf{leaf } E_s[I] \sim_\gamma (\textbf{write}(e), \textbf{leaf}; S_o)} \quad \text{(C-Leaf)}$$

$$\frac{E_s[I] \sim_\gamma (e, S_o)}{\textbf{node } E_s[I]\ M \sim_\gamma (e; \mathcal{A}_\gamma(M), \textbf{node}; S_o)} \quad \text{(C-Node1)}$$

$$\frac{E_s[I] \sim_\gamma (e, S_o)}{\textbf{node } V\ E_s[I] \sim_\gamma (e, \textbf{node}; [\![V]\!]; S_o)} \quad \text{(C-Node2)}$$

$$\frac{E_s[I] \sim_\gamma (e, S_o)}{\begin{array}{c}\textbf{case } E_s[I] \textbf{ of leaf } x \Rightarrow M_1 \mid \textbf{node } x_1\ x_2 \Rightarrow M_2 \sim_\gamma \\ \left(\begin{array}{lll}\textbf{case } e; \textbf{read } () \textbf{ of} & \textbf{leaf} \Rightarrow & \textbf{let } x = \textbf{read } () \textbf{ in } \mathcal{A}_\gamma(M_1) \\ & \textbf{node} \Rightarrow & [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2)\end{array}, S_o\right)\end{array}} \quad \text{(C-Case)}$$

Fig. 12. Correspondence between run-time states of source and target programs.

Proof
See Appendix C. □

Lemma 4.6
If $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash M : \tau$ and $(M,\delta) \sim (e, S_i, S_o)$, the following conditions hold:

- $S_i = \langle\delta\rangle$.
- If $(M,\delta) \to (M',\delta')$, then there exist e' and S'_i and S'_o that satisfy $(e, \langle\delta\rangle, S_o) \to^+ (e', \langle\delta'\rangle, S'_o)$ and $(M',\delta') \sim (e', S'_i, S'_o)$.
- If $(e, \langle\delta\rangle, S_o)$ is reducible,there exist M' and δ' that satisfy $(M,\delta) \to (M',\delta')$.

Proof
The first condition is obvious. For the proof of the other conditions, see Appendix D.
□

$$\begin{array}{ccccccccc}
 & & (Mx, x \mapsto V) & \to & (M', \delta') & \to & \cdots & \to & (V', \emptyset) \\
 & & \text{Lemma 4.5} \wr & & \text{Lemma 4.6} \wr & & & & \wr \\
(\mathcal{A}(M), [\![V]\!], \emptyset) & \to^* & (e, [\![V]\!], S_o) & \to^+ & (e', S_i', S_o') & \to^+ & \cdots & \to^+ & ((), \emptyset, [\![V']\!])
\end{array}$$

Fig. 13. Evaluation of a source and the target program.

4.2 Efficiency of Translated Programs

Let M be a source program of type $\mathbf{Tree}^- \to \mathbf{Tree}^+$. We argue below that the target program $\mathcal{A}(M)$ runs more efficiently than the source program $\mathit{unparse} \circ M \circ \mathit{parse}$, where *parse* is a function that parses the input stream and returns a binary tree, and *unparse* is a function that takes a binary tree as an input and writes it to the output stream. Note that the fact that the target program is a stream-processing program does not necessarily imply that it is more efficient than the source program: In fact, if the translation $\mathcal{A}$ were defined by $\mathcal{A}(M) = \mathit{unparse} \circ M \circ \mathit{parse}$, obviously there would be no improvement.

The target program being more efficient follows from the fact that the translation function $\mathcal{A}$ preserves the structure of the source program, with only replacing tree constructions with stream outputs, and case analyses on trees with stream inputs and case analyses on input tokens. More precisely, by inspecting the proof of soundness of the translation, we can observe:[4]

- When a closure is allocated in the execution of M (so that the heap space is consumed), the corresponding closure is allocated in the corresponding reduction step of $\mathcal{A}(M)$, and vice versa.
- When a function is called in the execution of M (so that the stack space is consumed), the corresponding function is called in the corresponding reduction step of $\mathcal{A}(M)$, and vice versa.
- When a case analysis on an input tree is performed in the execution of M, a token is read from the input stream and a case analysis on the token is performed in the corresponding reduction step of $\mathcal{A}(M)$.
- When a tree is constructed in the execution of M, the corresponding sequence of tokens is written on the output stream in the corresponding reduction steps of $A(M)$.

By the observation above, we can conclude:

- The memory space allocated by $\mathcal{A}(M)$ is less than the one allocated by $\mathit{unparse} \circ M \circ \mathit{parse}$, by the amount of the space for storing the intermediate trees output by *parse* and M (except for an implementation-dependent constant factor).
- The number of computation steps for running $\mathcal{A}(M)$ is the same as the one for running $\mathit{unparse} \circ M \circ \mathit{parse}$ (up to an implementation-dependent constant factor).

[4] To completely formalize these observations, we need to define another operational semantics that makes the heap and the stack explicit.

$$
\begin{array}{rcl}
M & ::= & i \mid \lambda x.M \mid x \mid M_1\ M_2 \mid M_1 + M_2 \\
& \mid & \mathbf{leaf}\ M \mid \mathbf{node}\ M_1\ M_2 \mid \mathbf{mleaf}\ M \mid \mathbf{mnode}\ M_1\ M_2 \\
& \mid & (\mathbf{case}\ M\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2) \\
& \mid & (\mathbf{mcase}\ M\ \mathbf{of}\ \mathbf{mleaf}\ x \Rightarrow M_1 \mid \mathbf{mnode}\ x_1\ x_2 \Rightarrow M_2) \\
& \mid & \mathbf{fix}\ f.M \\
e & ::= & v \mid x \mid e_1\ e_2 \mid e_1 + e_2 \\
& \mid & \mathbf{read}\ e \mid \mathbf{write}\ e \\
& \mid & \mathbf{mleaf}\ M \mid \mathbf{mnode}\ M_1\ M_2 \\
& \mid & (\mathbf{mcase}\ M\ \mathbf{of}\ \mathbf{mleaf}\ x \Rightarrow M_1 \mid \mathbf{mnode}\ x_1\ x_2 \Rightarrow M_2) \\
& \mid & (\mathbf{case}\ e\ \mathbf{of}\ \mathbf{leaf} \Rightarrow e_1 \mid \mathbf{node} \Rightarrow e_2) \\
& \mid & \mathbf{fix}\ f.e
\end{array}
$$

$$
\frac{\Gamma \mid \Delta \vdash M : \mathbf{Int}}{\Gamma \mid \Delta \vdash \mathbf{mleaf}\ M : \mathbf{Tree}^{\omega}} \quad (\text{T-MLeaf})
$$

$$
\frac{\Gamma \mid \Delta_1 \vdash M_1 : \mathbf{Tree}^{\omega} \qquad \Gamma \mid \Delta_2 \vdash M_2 : \mathbf{Tree}^{\omega}}{\Gamma \mid \Delta_1, \Delta_2 \vdash \mathbf{mnode}\ M_1\ M_2 : \mathbf{Tree}^{\omega}} \quad (\text{T-MNode})
$$

$$
\frac{\begin{array}{c}\Gamma \mid \Delta_1 \vdash M : \mathbf{Tree}^{\omega} \qquad \Gamma, x : \mathbf{Int} \mid \Delta_2 \vdash M_1 : \tau \\ \Gamma, x_1 : \mathbf{Tree}^{\omega}, x_2 : \mathbf{Tree}^{\omega} \mid \Delta_2 \vdash M_2 : \tau\end{array}}{\Gamma \mid \Delta_1, \Delta_2 \vdash \mathbf{mcase}\ M\ \mathbf{of}\ \mathbf{mleaf}\ x \Rightarrow M_1 \mid \mathbf{mnode}\ x_1\ x_2 \Rightarrow M_2 : \tau} \quad (\text{T-MCase})
$$

$$
\begin{array}{l}
\mathcal{A}(\mathbf{mleaf}\ M) = \mathbf{mleaf}\ \mathcal{A}(M) \\
\mathcal{A}(\mathbf{mnode}\ M_1\ M_2) = \mathbf{mnode}\ \mathcal{A}(M_1)\ \mathcal{A}(M_2) \\
\mathcal{A}(\mathbf{mcase}\ M\ \mathbf{of}\ \mathbf{mleaf}\ x \Rightarrow M_1 \mid \mathbf{mnode}\ x_1\ x_2 \Rightarrow M_2) = \\
\qquad \mathbf{mcase}\ \mathcal{A}(M)\ \mathbf{of}\ \mathbf{mleaf}\ x \Rightarrow \mathcal{A}(M_1) \mid \mathbf{mnode}\ x_1\ x_2 \Rightarrow \mathcal{A}(M_2)
\end{array}
$$

Fig. 14. Extended languages, type system and translation algorithm.

Thus, our translation is effective especially when the space for evaluating M is much smaller than the space for storing input and output trees.

5 Constructs for Storing Trees on Memory

By adding primitives for constructing and destructing trees on memory, we can allow programmers to selectively buffer input/output trees at the cost of efficiency of target programs. Let us extend the syntax of the source and target languages as follows:

$$
\begin{array}{rcl}
M & ::= & \cdots \mid \mathbf{mleaf}\ M \mid \mathbf{mnode}\ M_1\ M_2 \\
& \mid & (\mathbf{mcase}\ M\ \mathbf{of}\ \mathbf{mleaf}\ x \Rightarrow M_1 \mid \mathbf{mnode}\ x_1\ x_2 \Rightarrow M_2) \\
e & ::= & \cdots \mid \mathbf{mleaf}\ e \mid \mathbf{mnode}\ e_1\ e_2 \\
& \mid & (\mathbf{mcase}\ e\ \mathbf{of}\ \mathbf{mleaf}\ x \Rightarrow e_1 \mid \mathbf{mnode}\ x_1\ x_2 \Rightarrow e_2)\ .
\end{array}
$$

Here, **mleaf** M and **mnode** $M_1\ M_2$ are constructors of trees on memory and **mcase** $\cdots$ is a destructor.

We also add type $\mathbf{Tree}^{\omega}$, the type of trees stored on memory. The type system imposes no restriction on access order between variables of type $\mathbf{Tree}^{\omega}$ like type **Int** (so $\mathbf{Tree}^{\omega}$ is put in the ordinary type environment, not the ordered linear type environment). The translation algorithm $\mathcal{A}$ simply translates a source program, preserving the structure:

$$
\begin{array}{l}
\mathbf{fix}\ \mathit{strm\_to\_mem}.\\
\quad \lambda t.\mathbf{case}\ t\ \mathbf{of}\ \mathbf{leaf}\ x\ \Rightarrow\ \mathbf{mleaf}\ x\\
\qquad\qquad\quad \mid\ \mathbf{node}\ x_1\ x_2\ \Rightarrow\ \mathbf{mnode}\ (\mathit{strm\_to\_mem}\ x_1)\ (\mathit{strm\_to\_mem}\ x_2)\\
\mathbf{fix}\ \mathit{mem\_to\_strm}.\\
\quad \lambda t.\mathbf{mcase}\ t\ \mathbf{of}\ \mathbf{mleaf}\ x\ \Rightarrow\ \mathbf{leaf}\ x\\
\qquad\qquad\quad \mid \mathbf{mnode}\ x_1\ x_2\ \Rightarrow\ \mathbf{node}\ (\mathit{mem\_to\_strm}\ x_1)\ (\mathit{mem\_to\_strm}\ x_2)
\end{array}
$$

Fig. 15. Definition of *strm_to_mem* and *mem_to_strm*.

$$
\begin{array}{l}
\mathbf{let}\ \mathit{mswap}\ =\\
\quad \mathbf{fix}\ f.\lambda\ t.\mathbf{mcase}\ t\ \mathbf{of}\ \mathbf{mleaf}\ x\ \Rightarrow\ \mathbf{leaf}\ x\\
\qquad\qquad\qquad\quad \mid\ \mathbf{mnode}\ x_1\ x_2\ \Rightarrow\ \mathbf{node}\ (f\ x_2)\ (f\ x_1)\ \mathbf{in}\\
\quad \mathbf{fix}\ \mathit{swap\_deep}.\lambda n.\lambda t.\\
\qquad \mathbf{if}\ n = 0\ \mathbf{then}\ \mathit{mswap}\ (\mathit{strm\_to\_mem}\ t)\\
\qquad \mathbf{else}\\
\qquad\quad \mathbf{case}\ t\ \mathbf{of}\\
\qquad\qquad \mathbf{leaf}\ x \Rightarrow \mathbf{leaf}\ x\\
\qquad\quad \mid\ \mathbf{node}\ x_1\ x_2 \Rightarrow \mathbf{node}\ (\mathit{swap\_deep}\ (n-1)\ x_1)\ (\mathit{swap\_deep}\ (n-1)\ x_2)
\end{array}
$$

Fig. 16. A program which swaps children of nodes whose depth is more than n.

$$
\begin{array}{rcl}
\mathcal{A}(\mathbf{mleaf}\ M) & = & \mathbf{mleaf}\ \mathcal{A}(M)\\
\mathcal{A}(\mathbf{mnode}\ M_1\ M_2) & = & \mathbf{mnode}\ \mathcal{A}(M_1)\ \mathcal{A}(M_2)\\
\ldots & &
\end{array}
$$

These extensions are summarized in Figure 14.

With these primitives, a function *strm_to_mem*, which copies a tree from the input stream to memory, and *mem_to_strm*, which writes a tree on memory to the output stream, can be defined as shown in Figure 15.

Using the functions above, one can write a program that selectively buffers only a part of the input tree, while the type system guarantees that the selective buffering is correctly performed. For example, the program in Figure 16, which swaps children of nodes whose depth is more than n, only buffers the nodes whose depth is more than n. In that example, we assume that we have booleans, comparison operators and **if** M **then** M_1 **else** M_2 in our source and target language syntax. The typing rule for **if** − **then** − **else** expressions is as follows:

$$\frac{\Gamma \mid \Delta_1 \vdash M : \mathbf{bool} \qquad \Gamma \mid \Delta_2 \vdash M_1 : \tau \qquad \Gamma \mid \Delta_2 \vdash M_2 : \tau}{\Gamma \mid \Delta_1, \Delta_2 \vdash \mathbf{if}\ M\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 : \tau.} \qquad (\text{T-If})$$

The proof of Theorem 4.1 can be easily adapted for the extended language.

One drawback of the framework described in this section is that programmers need to write tree buffering operations. We report elsewhere an algorithm that automatically inserts *strm_to_mem* and *mem_to_strm* (Suenaga *et al.*, 2005).

6 Extension for Dealing with XML

We extend our framework to deal with XML documents in this section. Note that we do not consider schemata for XML in this paper (i.e., XML documents are considered as untyped labeled unranked trees.) Using schemata in our framework is left as future work.

Remove **leaf**, **node** and **case** expressions from the definition of M and add the following constructs.

$$\begin{array}{rcl} M & ::= & \ldots \\ & | & \textbf{leaf}\ M \mid \textbf{node}\ M_l\ M_a\ M \\ & | & \textbf{nil} \mid \textbf{cons}(M_1, M_2) \\ & | & \textbf{caseElem}\ M\ \textbf{of leaf}\ x\ \Rightarrow\ M_1\ \mid\ \textbf{node}(l, \mathit{attr}, \mathit{tl})\ \Rightarrow\ M_2 \\ & | & \textbf{caseElem}\ M\ \textbf{of nil}\ \Rightarrow\ M_1\ \mid\ \textbf{cons}(x, \mathit{xs})\ \Rightarrow\ M_2 \end{array}$$

Fig. 17. Change to the syntax of the source language to deal with XML documents.

The difference between binary trees and (untyped) XML documents is that the latter ones (i) are unranked trees and (ii) contain end tags that mark the end of sequences in the stream format. The first point can be captured as the difference between the type `binarytree` and the type `xmltree` in the following ML-style type declarations.

```
type binarytree = Leaf of int | Node of tree * tree;

type xmltree = Leaf of pcdata
             | Node of label * attribute * treelist
and treelist = Nil | Cons of xmltree * treelist;
```

While the type `binarytree` represents binary trees, `xmltree` represents unranked trees. Based on the difference between these types, we can replace the **case**-construct of the source language with the following two pattern-matching primitives.

```
caseElem ... of Leaf x -> ...
              | Node(l, attr, tl) -> ...

caseSeq ... of Nil -> ...
             | Cons(x, xs) -> ...
```

The above changes to the syntax are summarized in Figure 6.

We have not incorporated regular expression patterns (Hosoya & Pierce, 2003) into our framework because type checking regular pattern matching requires document type information. This makes our current framework not satisfactory as a high-level programming language for XML processing. We plan to address that by designing a translation algorithm from a language with regular pattern matching into our language.

Typing rules can also be naturally extended. For example, the typing rules for the **leaf** and **node** constructs are:

$$\frac{\Gamma \mid \Delta \vdash M : \textbf{pcdata}}{\Gamma \mid \Delta \vdash \textbf{leaf}\ M : \textbf{xmltree}^{+}}$$

$$\frac{\Gamma \mid \Delta_1 \vdash M_l : \textbf{label} \qquad \Gamma \mid \Delta_2 \vdash M_a : \textbf{attribute} \qquad \Gamma \mid \Delta_3 \vdash M_3 : \textbf{treelist}^{+}}{\Gamma \mid \Delta_1, \Delta_2, \Delta_3 \vdash \textbf{node}(M_l, M_a, M)}$$

and the typing rule for the **caseSeq** construct is:

$$\frac{\Gamma \mid \Delta \vdash M : \mathbf{pcdata}}{\Gamma \mid \Delta \vdash \mathbf{leaf}\ M : \mathbf{xmltree}^{+}} \quad \text{(T-Leaf)}$$

$$\frac{\Gamma \mid \Delta_1 \vdash M_l : \mathbf{label} \qquad \Gamma \mid \Delta_2 \vdash M_a : \mathbf{attribute} \qquad \Gamma \mid \Delta_3 \vdash M_3 : \mathbf{treelist}^{+}}{\Gamma \mid \Delta_1, \Delta_2, \Delta_3 \vdash \mathbf{node}(M_l, M_a, M)} \quad \text{(T-Node)}$$

$$\frac{\begin{array}{c}\Gamma \mid \Delta_1 \vdash M : \mathbf{xmltree} \qquad \Gamma, x : \mathbf{pcdata} \mid \Delta_2 \vdash M_2 : \tau \\ \Gamma, l : \mathbf{label}, attr : \mathbf{attribute} \mid tl : \mathbf{treelist}, \Delta_2 \vdash M_1 : \tau\end{array}}{\Gamma \mid \Delta_1, \Delta_2 \vdash \mathbf{caseElem}\ M\ \mathbf{of}\ \mathbf{node}(l, attr, tl) \ \Rightarrow\ M_1 \ \mid\ \mathbf{leaf}(x) \ \Rightarrow\ M_2 : \tau} \quad \text{(T-CaseElem)}$$

$$\frac{\begin{array}{c}\Gamma \mid \Delta_1 \vdash tl : \mathbf{treelist} \\ \Gamma \mid \Delta_2 \vdash M_1 : \tau \qquad \Gamma \mid x : \mathbf{xmltree}, xl : \mathbf{treelist}, \Delta_2 \vdash M_2 : \tau\end{array}}{\Gamma \mid \Delta_1, \Delta_2 \vdash \mathbf{caseSeq}\ tl\ \mathbf{of}\ \mathbf{nil} \Rightarrow M_1 \mid \mathbf{cons}(x, xl) \Rightarrow M_2 : \tau} \quad \text{(T-CaseSeq)}$$

Fig. 18. Typing rules for XML extension.

$$\begin{array}{l}
\mathcal{A}(\mathbf{caseElem}\ M\ \mathbf{of}\ \mathbf{node}(l, attr, tl) \ \Rightarrow\ M_1 \ \mid\ \mathbf{leaf}(x) \ \Rightarrow\ M_2) = \\
\quad \mathbf{case}\ \mathbf{readtoken}()\ \mathbf{of} \\
\qquad \quad \mathbf{start_tag}(l, attr) \ \Rightarrow\ \mathbf{push}(l); [()/tl]\mathcal{A}(M_1) \\
\qquad \mid\ \mathbf{pcdata}(x) \ \Rightarrow\ \mathcal{A}(M_2) \\
\qquad \mid\ _ \ \Rightarrow\ \mathbf{raise}\ \mathit{IllFormedException}\ \text{(* end_tag or end of file *)} \\
\\
\mathcal{A}(\mathbf{caseSeq}\ M\ \mathbf{of}\ \mathbf{nil} \ \Rightarrow\ M_1 \ \mid\ \mathbf{cons}(x, xs) \ \Rightarrow\ M_2) = \\
\quad \mathbf{case}\ \mathbf{peektoken}()\ \mathbf{of} \\
\qquad \quad \mathbf{end_tag}(l) \ \Rightarrow\ \mathbf{let}\ l' = \mathbf{pop}()\ \mathbf{in} \\
\qquad \qquad \qquad \qquad \mathbf{if}\ l = l'\ \mathbf{then}\ \mathbf{readtoken}(); \mathcal{A}(M_1) \\
\qquad \qquad \qquad \qquad \mathbf{else\ raise}\ \mathit{IllFormedException} \\
\qquad \mid\ _ \ \Rightarrow\ [()/x, ()/xs]\mathcal{A}(M_2)
\end{array}$$

Fig. 19. Definition of $\mathcal{A}$ for XML processing constructs.

$$\frac{\begin{array}{c}\Gamma \mid \Delta_1 \vdash tl : \mathbf{treelist}^{-} \qquad \Gamma \mid \Delta_2 \vdash M_1 : \tau \\ \Gamma \mid x : \mathbf{xmltree}^{-}, xl : \mathbf{treelist}^{-}, \Delta_2 \vdash M_2 : \tau\end{array}}{\Gamma \mid \Delta_1, \Delta_2 \vdash \mathbf{caseSeq}\ tl\ \mathbf{of}\ \mathbf{nil} \Rightarrow M_1 \mid \mathbf{cons}(x, xl) \Rightarrow M_2 : \tau.}$$

The restriction on the access order is expressed by $x : \mathbf{xmltree}^{-}, xl : \mathbf{treelist}^{-}, \Delta_2$ as in T-Node. Figure 18 shows the typing rules for newly added constructs.

Figure 19 shows the definition of $\mathcal{A}$ for XML processing constructs. **readtoken**() reads an element from the input stream where the element is either **start_tag**($l, attr$), **end_tag**(l), **pcdata**(x) or **end_of_file**. The primitive **peektoken**() peeks the top of the input stream without deleting the element. Values of type **attribute** are stored on memory like values of type **pcdata**, because attributes can occur in an arbitrary order (e.g., `<a b=''foo'' c=''baa''>` and `<a c=''baa'' b=''foo''>` have the same meaning). Thus, type **attribute** is put in the ordinary type environment, not in the ordered linear type environment, in the rule T-CaseElem.

In target programs, a stack is used to check well-formedness of input documents. When a start tag is read, the tag is pushed on the stack. When an end tag is read, it is compared with the tag stored on the top of the stack.

Note that the pattern **nil** in the source language is translated to the pattern for closing tags.

Remark 6.1
The data representation and pattern matching constructs for XML are essentially the same as those used in *general-purpose* programming languages that provide no primitive support for XML processing (except that there is an additional restriction on access order). Programs written with such data representation and pattern matching are certainly more awkward than those written in XML-centric programming languages (Hosoya & Pierce, 2003; Benzaken *et al.*, 2003), which provide a number of useful features such as regular expression pattern matching. (For example, compare the program in Figure 20 with a program using regular expression pattern matching.) We, however, believe that writing a program in our language is still much easier than writing it using a low-level stream processing language (like the target language of our framework).

Programming can be made much easier by introduction of additional layers of transformations. For example, we have already implemented a translator that takes an ordinary XML processing program as an input, and automatically inserts buffering primitives. so that the output is well-typed in our type system (Suenaga *et al.*, 2005; Sato, 2007). The result of the transformation is then given as an input of the transformation described in the present paper and transformed into a stream-processing program. Thus, the source language discussed in this section actually serves as an *intermediate* language of the entire transformation framework. We also plan to add an additional layer of transformation for compiling regular expression pattern matching.

7 Implementation and Experiments

We have implemented a translator X-P based on the type system and the translation algorithm we presented. X-P can deal with both tree buffering in Section 5 and XML processing in Section 6.

Figure 20 shows a part of a sample X-P program. That program extracts `firstname` and `lastname` fields from input data, by filtering `id`, `street`, `city`, `state` and `zip` fields out. An example of input and the output is shown in Figure 21.

The current implementation of X-P translator adopts ML-like syntax. Here is an overview of the syntax of X-P language.

- a tree expressed with an XML document

  ```
  <row sex=''male''><firstname>Kohei</firstname>
                    <lastname>Suenaga</lastname></row>
  ```

  is written like

  ```
  <row>@{sex = ``male''}[
    <firstname>@{}[PC ``Kohei''];
    <lastname>@{}[PC ``Suenaga'']
  ]
  ```

```
...
let rec process_row = fun /t/ ->
  match t with
    /id/::/tl/ -> skip_tree id;
      match tl with /firstname/::/tl/ ->
        let firstname = get_firstname firstname in
          match tl with /lastname/::/tl/ ->
            let lastname = get_lastname lastname in
              match tl with /street/::/tl/ -> skip_tree street;
                match tl with /city/::/tl/ -> skip_tree city;
                  match tl with /state/::/tl/ -> skip_tree state;
                    match tl with /zip/::/tl/ -> skip_tree zip;
                      match tl with [] ->
                        <row>@{}[
                          <firstname>@{}[PC firstname];
                          <lastname>@{}[PC lastname]
                        ]
...
```

Fig. 20. An X-P program that extracts first name and last names fields from an input document.

```
Input:                            Output:
<table>                           <table>
  <row>                             <row>
    <id>0001</id>                     <firstname>Al</firstname>
    <firstname>Al</firstname>         <lastname>Aranow</lastname>
    <lastname>Aranow</lastname>     </row>
    <street>1 Any St.</street>      ...
    <city>Anytown</city>          </table>
    <state>AL</state>
    <zip>22000</zip>
  </row>
  ...
</table>
```

Fig. 21. A sample input and the intended output for programs in Figure 20 and Figure 22.

- **case** M **of nil** $\Rightarrow M_1 \mid$ **cons**$(x, y) \Rightarrow M_2$ is written like `match M with [] -> M1 | /x/::/y/ -> M2`. We mark bound variables of type **Tree**$^-$ with surrounding slashes. These marks are used in the type checking algorithm described in Section 3.4.
- In the current implementation, we allow only two patterns for matching lists: `[]` and `x::y`.

Remark 7.1

The program in Figure 20 is unnecessarily involved because of the limited pattern matching constructs and the need for automatic insertion of the `skip_tree` instructions. By using the usual ML-like pattern matching constructs and the translator for automatic insertion of buffering (recall Remark 6.1), the program in Figure 20 can be actually simplified to:

```
let rec _process_row29 = (fun _t30 ->
  match _t30;peek_token () with
    StartTag(_, _) | Pcdata _ ->
      skip_tree (); match ();peek_token () with
        StartTag(_, _) | Pcdata _ ->
          let _firstname35 = _get_firstname7 () in
            match ();peek_token () with
              StartTag(_, _) | Pcdata _ ->
                let _lastname38 = _get_lastname8 () in
                  match ();peek_token () with
                    StartTag(_, _) | Pcdata _ ->
                      skip_tree (); match ();peek_token () with
                        StartTag(_, _) | Pcdata _ ->
                          skip_tree (); match ();peek_token () with
                            StartTag(_, _) | Pcdata _ ->
                              skip_tree (); match ();peek_token () with
                                StartTag(_, _) | Pcdata _ ->
                                  skip_tree (); match ();peek_token () with
                                    EndTag __65 ->
                                      read_token ();
                                      assert (pop ()) = Pcdata(__65);
                                      write (StartTag("row",[]));
                                      write (StartTag("first",[]));
                                      write (Pcdata(_firstname35));
                                      write (EndTag("first"));
                                      write (StartTag("last",[]));
                                      write (Pcdata(_lastname38));
                                      write (EndTag("last"));
                                      write (EndTag("row"))
...
```

Fig. 22. The result of translation of the program in Figure 20.

```
  let rec process_row = fun /t/ ->
  match _::/firstname/::tl with
     let firstname = get_firstname firstname in
        match tl with /lastname/::_ ->
            let lastname = get_lastname lastname in
                   <row>@{}[
                      <firstname>@{}[PC firstname];
                      <lastname>@{}[PC lastname]
                     ]
```

Figure 22 shows the result of translation. Based on the framework so far, the system first type-checks the source program and then transforms it into a stream processing program written in Objective Caml.

We measured the maximum heap size during the execution for four micro-benchmarks in XSLTMark (DataPower Technology, 2001) (`dbonerow`, `dbtail`, `avts` and `stringsort`), one in our own micro-benchmark (`evensort`), and two in XMark (Schmidt *et al.*, 2001) (Q_1 and Q_8). We also measured the execution time of benchmarks Q_1 and Q_8.

```
<site>
 ...
 <people>
  <person id=''person0''><name>Kohei Suenaga</name> ... </person>
  ...
 </people>
 ...
 <closed_auctions>
  <closed_auction> ... <buyer person=''person0''/> ... </closed_auction>
  ...
 </closed_auctions>
</site>
```

Fig. 23. Sample input of benchmarks Q_1 and Q_8.

Benchmarks `dbonerow`, `dbtail`, `avts`, `stringsort` and `evensort` use XML documents of the form shown in Figure 21 as input. The following is the description of those benchmarks.

- `dbonerow`: Scans the whole input and extracts the `row` record whose `id` field is equal to 0432 in the form of an HTML document.
- `dbtail`: Scans the whole input and extracts `firstname` field and `lastname` field of each `row` record (A program presented in part in Figure 20).
- `avts`: Scans the whole input and converts each `row` record to an `address` tag whose attributes have the information the `row` record has.
- `stringsort`: Sorts all the `row` records in input documents by values of `firstname` field.
- `evensort`: Sorts `row` records whose `id` field is an even number by values of `firstname` field.

Benchmarks Q_1 and Q_8 use auction data of the form shown in Figure 23 as input. The followings are the description of each benchmark.

- Q_1: Extracts the name of the person whose ID is `person0`.
- Q_8: Extracts the names of persons and the number of items bought by each person from the join of `people` and `closed_auctions`.

For each benchmark, we prepared (1) an OCaml program generated by our implementation from a manually written tree-processing program in our language and (2) a manually written OCaml tree-processing program. All the OCaml programs were compiled by ocamlopt 3.08.4. The experiments were performed on Intel(R) CPU 1.06GHz with 1GB memory. We executed each of `dbonerow`, `dbtail`, `avts`, `stringsort` and `evensort` with 12 input documents each of which contains 10, 50, 100, 250, 500, 750, 1000, 2000, 4000, 6000, 8000 and 10000 records. Benchmarks Q_1 and Q_8 were executed with 5, 10, 50 and 100 MB documents.

Figure 24 shows the memory consumption of `dbonerow`, `dbtail`, `avts`, `stringsort` and `evensort`. In those graphs, the horizontal axes show the number of `row` records. Figure 25 shows the memory consumption of Q_1 and Q_8. In those graphs, the horizontal axes show the size of input documents. In both figures, the vertical axes are the maximum heap size during the execution (measured in kilobytes.)

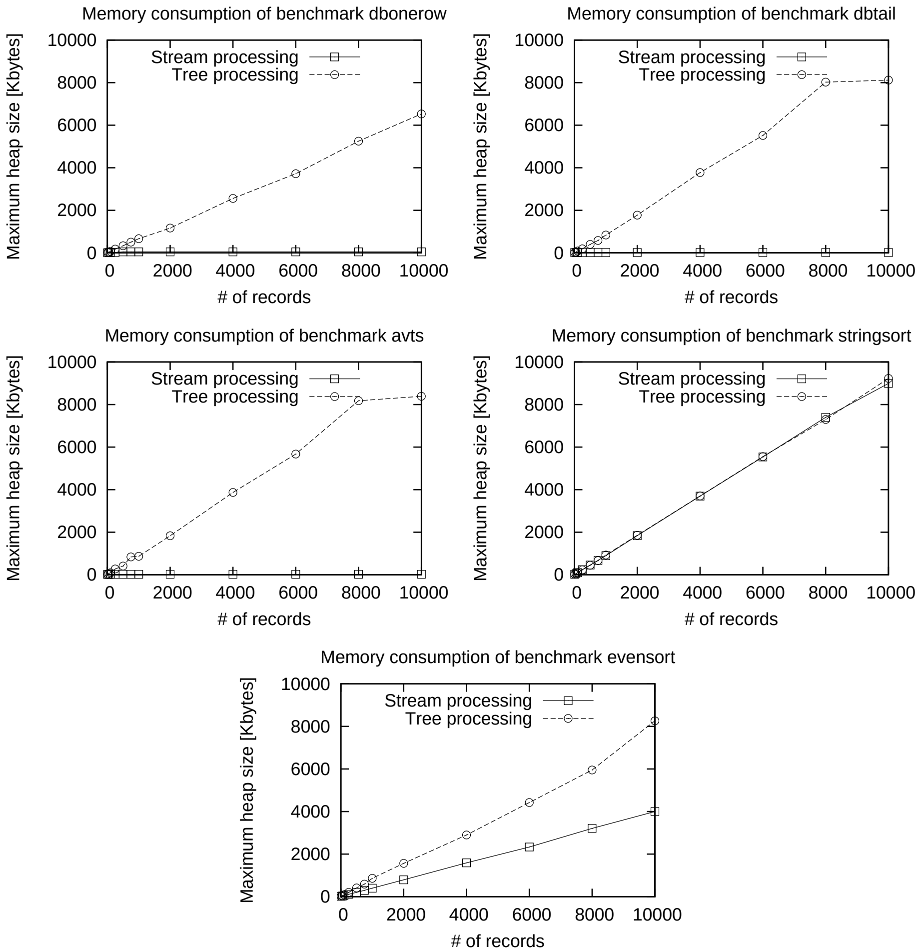


Fig. 24. Memory consumption of benchmarks `dbonerow`, `dbtail`, `avts`, `stringsort` and `evensort`.

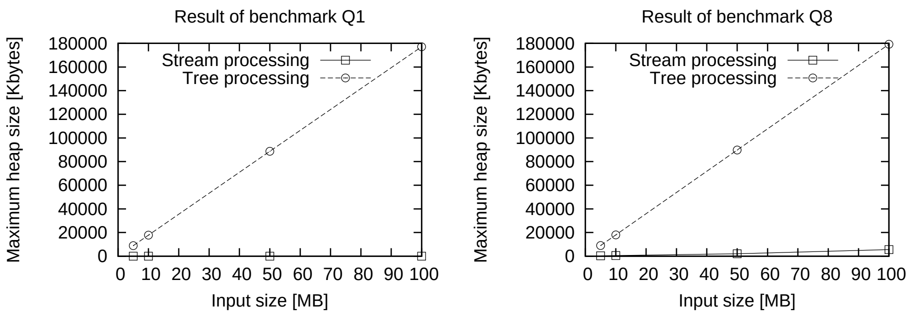


Fig. 25. Memory consumption of benchmarks Q_1 and Q_8.

In each benchmark, memory consumption of tree-processing programs is proportional to the number of the size of input. In `dbonerow`, `dbtail`, `avts` and Q_1

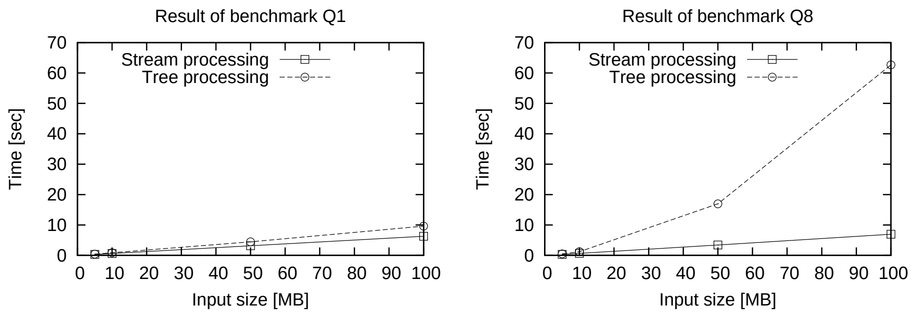


Fig. 26. Time consumption of benchmarks Q_1 and Q_8.

benchmarks, in which all the tree variables have ordered linear type, the heap memory consumption of stream-processing programs is constant w.r.t. the number of records. (On the other hand, the stack memory consumption is linear in the depth of the input document. The input documents used in our experiments, however, have the constant depth, so that the stack memory consumption is also constant.) In `stringsort` benchmark, which requires all the records to be stored on memory in processing, so that all the tree variables in the program have buffered tree type, the program consumes as much memory as the tree-processing version does. Benchmark `evensort` requires records with even `id` field values to be stored on memory. The memory consumption of the stream-processing version is almost a half of that of the tree-processing version as expected. In benchmark Q_8, which performs a join operation and hence requires a part of input to be on heap, the memory consumption of the stream-processing program is about 3% of that of tree-processing one. For example, for 100MB input, memory consumption is 5576KB in the stream-processing program, while 179232KB in the tree-processing program. Note that we cannot simply compare this result with that in (Koch & Scherzinger, 2003), because our tree-processing program is optimized by hand to produce a memory-efficient stream-processing program,[5] while (Koch & Scherzinger, 2003) automatically derives stream-processing programs from XMark queries.

Figure 26 shows the execution time of benchmark Q_1 and Q_8. In both benchmarks, the stream-processing program is faster than the tree-processing program.

8 Related Work

Attribute grammars are widely used for specifying XML processing. Nakano and Nishimura (Nakano & Nishimura, 2001; Nakano, 2004) proposed a method for translating tree-processing attribute grammars into stream-processing ones by descriptional composition (Ganzinger & Giegerich, 1984) with parsing and unparsing grammars. *Quasi-SSUR condition* in (Nakano & Nishimura, 2001) and *single use requirement* in (Nakano, 2004), which force attributes of non-terminal symbols to

[5] Specifically, we took advantage of the following facts in translating Q_8 into our language: (1) `people` comes before `closed_auctions` (2) in `people` tree, only the name and the ID of each person are needed and thus should be buffered.

$$
\begin{array}{l}
N \rightarrow \textbf{node}\ N_1\ N_2 \\
\qquad N_1.inh = f_1\ N.inh; N_2.inh = f_2\ N.inh\ N_1.syn\ N_1.inh \\
\qquad N.syn = f_3\ N.inh\ N_1.syn\ N_1.inh\ N_2.syn\ N_2.inh \\
N \rightarrow \textbf{leaf}\ i \\
\qquad N.syn = f_4\ N.inh\ i \\
\\
\textbf{fix}\ f.\lambda inh.\lambda t.\textbf{case}\ t\ \textbf{of} \\
\qquad \textbf{leaf}\ x \Rightarrow f_4\ inh\ x \\
\qquad \textbf{node}\ x_1\ x_2 \Rightarrow \quad \textbf{let}\ N_1.inh = f_1\ inh\ \textbf{in} \\
\qquad\qquad\qquad\qquad\quad \textbf{let}\ N_1.syn = f\ N_1.inh\ x_1\ \textbf{in} \\
\qquad\qquad\qquad\qquad\quad \textbf{let}\ N_2.inh = f_2\ N.inh\ N_1.syn\ N_1.inh\ \textbf{in} \\
\qquad\qquad\qquad\qquad\quad \textbf{let}\ N_2.syn = f\ N_2.inh\ x_2\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\quad f_3\ N.inh\ N_1.syn\ N_1.inh\ N_2.syn\ N_2.inh
\end{array}
$$

Fig. 27. L-attributed grammar over binary trees and corresponding program.

be used at most once, seems to correspond to our linearity restriction on variables of tree types, but there seems to be no restriction that corresponds to our order restriction. As a result, their method can deal with programs (written as attribute grammars) that violate the order restriction of our type system, although in that case, generated stream-processing programs store parts of trees in memory, so that the translation may not improve the efficiency. On the other hand, an advantage of our method is that programs are easier to read and write since one can write programs as ordinary functional programs except for the restriction imposed by the type system, rather than as attribute grammars. Another advantage of our method is that we can deal with source programs that involve side-effects (e.g. programs that print the value of every leaf) while that seems difficult in their method based on attribute grammars (since the order is important for side effects).

The class of well-typed programs in our language seems to be closely related to the class of L-attributed grammars (Aho *et al.*, 1986). The upper half of Figure 27 shows the general form of L-attributed grammar over binary trees. In the grammar, *syn* represents a synthesized attribute and *inh* represents an inherited attribute. $f_1, \ldots, f_4$ are functions that calculate values of attributes. For example, the inherited attribute of N_2 is calculated from the inherited attributes of N and N_1 and the synthesized attribute of N_1 with the function f_2. (Note that $N_2.inh$ depends only on $N.inh, N_1.syn$ and $N_1.inh$ from the definition of L-attributed grammar.) That L-attributed grammar can be expressed as a program as shown in the lower half of Figure 27. If output trees are not used in attributes, the program is well-typed. Conversely, any program that is well-typed in our language seems to be definable as an L-attribute grammar. The corresponding attribute grammar may, however, be awkward, since one has to encode control information into attributes.

Koch and Scherzinger (Koch & Scherzinger, 2003) proposed XML Stream Attribute Grammars (XSAGs), which are extended regular tree grammars (Neven, 2005) with several primitives for manipulating the output stream. By limiting the grammars to L-attributed ones, they ensure that any XSAGs are evaluated as stream-processing programs that read the input stream from head to tail with bounded memory consumption, and thus programmers need not care about the input stream. However, their framework does not ensure that output is well-formed. We guess well-typed XML processing programs in our framework are at least as expressive as XSAGs that output well-formed documents by the following observation. According to (Koch & Scherzinger, 2003), expressive power of XSAGs is equivalent to that of XML-DPDTs, deterministic pushdown transducers with an auxiliary stack for checking well-formedness of input documents. XML-DPDTs are essentially XML processing programs that (1) read well-formed input documents in left-to-right, depth-first preorder, and (2) generate an output in document order, which are also the properties of well-typed programs of our framework. Thus, the expressive power of XSAGs is equivalent to that of our well-typed programs.

There are other studies on translation of tree-processing programs into stream-processing programs. Some of them (Green *et al.*, 2001; Gupta & Suciu, 2003; Avila-Campillo *et al.*, 2002; Bar-Yossef *et al.*, 2005; Olteanu *et al.*, 2002) deal with XPath expressions (Berglund *et al.*, 2003; Scardina & Fernandez, 2003) and others (Ludäscher *et al.*, 2002) deal with XQuery (Boag *et al.*, 2003; Chamberlin *et al.*, 2003). Those translations are more aggressive than ours in the sense that the structure of source programs is changed so that input trees can be processed in one pass. On the other hand, their target languages (XPath and XQuery languages) are restricted in the sense that they do not contain functions and side-effects [6].

There are many studies on program transformation (Wadler, 1988; Meijer *et al.*, 1991) for eliminating intermediate data structures of functional programs, known as deforestation or fusion. Although the goal of our translation is also to remove intermediate data structures from $\mathit{unparse} \circ f \circ \mathit{parse}$, the previous methods are not directly applicable since those methods do not guarantee that transformed programs access inputs in a stream-processing manner. In fact, *swap* in Figure 2, which violates the access order, can be expressed as a treeless program (Wadler, 1988) or a catamorphism (Meijer *et al.*, 1991), but the result of deforestation is not an expected stream-processing program.

Actually, there are many similarities between the restriction of treeless program (Wadler, 1988) and that of our type system. In treeless programs, (1) variables have to occur only once, and (2) only variables can be passed to functions. (1) corresponds to the linearity restriction of our type system. (2) is the restriction for prohibiting trees generated in programs to be passed to functions, which corresponds to the restriction that functions cannot take values of type $\mathbf{Tree}^+$ in our type system. The main differences are:

- Our type system additionally imposes a restriction on the access order. This is required to guarantee that translated programs read input streams sequentially.
- We restrict programs with a type system, while the restriction on treeless programs is syntactic. Our type-based approach enables us to deal with higher-

[6] Though XQuery can deal with user-defined functions, the fragment dealt with in (Ludäscher *et al.*, 2002) excludes user-defined functions.

order functions. The type-based approach is also useful for automatic inference of selective buffering of trees, as discussed in Section 9.

The type system we used in this paper is based on the ordered linear logic proposed by Polakow (Polakow, 2001). He proposed a logic programming language Olli, logical framework OLF and ordered lambda calculus based on the logic. There are many similarities between our typing rules and his derivation rules for the ordered linear logic. For example, our type judgment $\Gamma \mid \Delta \vdash M : \tau$ corresponds to the judgment $\Gamma; \cdot; \Delta \vdash A$ of ordered linear logic. The rule T-ABS1 corresponds to a combination of the rules for an ordered linear implication and the modality (!). However, we cannot use ordered linear logic directly since it would make our type system unsound.

Petersen et al. (Petersen *et al.*, 2003) used ordered linear types to guarantee correctness of memory allocation and data layout. While they used an ordered linear type environment to express a spatial order, we used it to express a temporal order.

As we stated in Section 3 and 5, one can write tree-processing programs that selectively skip and/or buffer trees by using *skip_tree*, *copy_tree*, *strm_to_mem* and *mem_to_strm*. However, inserting those functions by hand is sometimes tedious. To solve that problem, we have developed a type-directed source-to-source translation for automatically inserting these functions (Suenaga *et al.*, 2005).

9 Conclusion

We have proposed a type system based on ordered linear types to enable translation of tree-processing programs into stream-processing programs, and proved the correctness of the translation.

Since our translation algorithm preserves the structure of source programs, the translation works in the presence of side effects other than stream inputs/outputs. Our framework can also be easily extended to deal with multiple input trees.

In addition to application to XML processing, our translation framework may also be useful for optimization of distributed programs that process and communicate complex data structures. Serialization/unserialization of data corresponds to unparsing/parsing in Figure 1, so that our translation framework can be used for eliminating intermediate data structures and processing serialized data directly.

Acknowledgement

We thank Keisuke Nakano for his comment on this research. We also thank the anonymous referees for their fruitful comments.

References

Aho, Alfred V., Sethi, Ravi, & Ullman, Jeffrey D. (1986). *Compilers.* Addison-Wesley Pub Co.

Avila-Campillo, Iliana, Green, Todd J., Gupta, Ashish, Onizuka, Makoto, Raven, Demian, & Suciu, Dan. 2002 (Oct.). XMLTK: An XML toolkit for scalable XML stream processing. *Proceedings of PLAN-X*.

Baker, Henry G. (1992). Lively Linear Lisp – Look Ma, No Garbage! *ACM SIGPLAN notices*, **27**(8), 89–98.

Bar-Yossef, Ziv, Fontoura, Marcus, & Josifovski, Vanja. (2005). Buffering in query evaluation over XML streams. *Pages 216–227 of: Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART symposium on priciples of database systems (PODS2005)*.

Benzaken, Véronique, Castagna, Giuseppe, & Frisch, Alain. 2003 (Aug.). CDuce: An XML-centric general-purpose language. *Pages 51–63 of: Proceedings of 8th ACM SIGPLAN international conference on functional programming (ICFP 2003)*.

Berglund, Anders, Boag, Scott, Chamberlin, Don, Fernández, Mary F., Kay, Michael, Robie, Jonathan, & Simeon, Jérôme. 2003 (Nov.). *XML path language (XPath) 2.0*. World Wide Web Consortium. `http://www.w3.org/TR/xpath20/`.

Boag, Scott, Chamberlin, Don, Fernández, Mary F., Florescu, Daniela, Robie, Jonathan, & Siméon, Jérôme. 2003 (Nov.). *XQuery 1.0: An XML Query Language*. World Wide Web Consortium. `http://www.w3.org/TR/xquery/`.

Bray, Tim, Paoli, Jean, C.M.Sperberg-McQueen, & Maler, Eve. 2000 (Oct.). *Extensible Markup Language (XML) 1.0 (Second Edition)*. Tech. rept. World Wide Web Consortium. `http://www.w3.org/TR/REC-xml`.

Chamberlin, Don, Frankhauser, Peter, Marchiori, Massimo, & Robie, Jonathan. 2003 (Nov.). *XML query (XQuery) requirements*. World Wide Web Consortium. `http://www.w3.org/TR/xquery-requirements/`.

DataPower Technology, Inc. 2001 (Mar.). *XSLTMark*. `http://www.datapower.com/xmldev/xsltmark.html`.

Ganzinger, Harald, & Giegerich, Robert. (1984). Attribute Coupled Grammars. *Proceedings of the ACM SIGPLAN '84 symposium on compiler construction*.

Green, T., Onizuka, M., & Suciu, D. (2001). *Processing XML Streams with Deterministic Automata and Stream Indexes*. Tech. rept. University of Washington.

Gupta, Ashish Kumar, & Suciu, Dan. 2003 (June). Stream Processing of XPath Queries with Predicates. *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*.

Hosoya, Haruo, & Pierce, Benjamin C. (2003). XDuce: A Typed XML Processing Language. *ACM Transactions on Internet Technology (TOIT)*, **3**(2), 117–148.

Igarashi, Atsushi, & Kobayashi, Naoki. (2002). Resource Usage Analysis. *Pages 331–342 of: Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Koch, Christoph, & Scherzinger, Stefanie. 2003 (Sept.). Attribute grammars for scalable query processing on XML streams. *Proceedings of the 9th international conference on data base programming language (DBPL)*.

Ludäscher, Bertram, Mukhopadhyay, Pratik, & Papakonstantinou, Yannis. (2002). A Transducer-Based XML Query Processor. *Proceedings of the 28th International Conference on Very Large Data Bases*.

Meijer, Erik, Fokkinga, Maarten, & Paterson, Ross. (1991). Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. *Pages 124 – 144 of: Proceedings of the 5th ACM conference on functional programming languages and computer architecture*.

Nakano, Keisuke. (2004). *Composing Stack-Attributed Tree Transducers*. Tech. rept. METR–2004–01. Department of Mathematical Informatics, University of Tokyo, Japan.

Nakano, Keisuke, & Nishimura, Susumu. (2001). Deriving Event-Based Document Transformers from Tree-Based Specifications. *Electronic Notes in Theoretical Computer Science*, **44**(2).

Neven, Frank. (2005). Attribute grammars for unranked trees as a query language for structured documents. *Journal of computer and system sciences*, **70**(2), 221–257.

N.Turner, David, Wadler, Philip, & Mossin, Christian. (1995). Once Upon A Type. *Pages 1–11 of: Proceedings of Functional Programming Languages and Computer Architecture*.

Olteanu, Dan, Meuss, Holger, Furche, Tim, & François. (2002). XPath: Looking Forward. *Pages 109–127 of: Proceedings of the EDBT workshop on XML data management (XMLDM)*.

Petersen, Leaf, Harper, Robert, Crary, Karl, & Pfenning, Frank. (2003). A Type Theory for Memory Allocation and Data Layout. *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Polakow, Jeff. 2001 (June). *Ordered Linear Logic and Applications*. Ph.D. thesis, Carnegie Mellon University. Available as Technical Report CMU-CS-01-152.

Sato, Shuji. 2007 (Mar.). *Automatic insertion of buffering primitives for generating XML stream processor (in Japanese)*. M.Phil. thesis, Tohoku University.

Scardina, Mark, & Fernandez, Mary. 2003 (Aug.). *XPath Requirements Version 2.0*. World Wide Web Consortium. `http://www.w3.org/TR/xpath20req/`.

Schmidt, A. R., Waas, F., Kersten, M. L., Florescu, D., Manolescu, I., Carey, M. J., & Busse, R. 2001 (Apr.). *The XML benchmark project*. Tech. rept. Centrum voor Wiskunde en Informatica.

Suenaga, Kohei, Kobayashi, Naoki, & Yonezawa, Akinori. 2005 (Sept.). Extension of Type-Based Approach to Generation of Stream-Processing Programs by Automatic Insertion of Buffering Primitives. *Pages 98–114 of: International Symposium on Logic-based Program Synthesis and Translformation (LOPSTR 2005)*.

Wadler, P. (1988). Deforestation: Transforming Programs to Eliminate Trees. *Pages 344–358 of: ESOP '88. European Symposium on Programming, Nancy, France, 1988 (Lecture Notes in Computer Science, vol. 300)*. Berlin: Springer-Verlag.

A Proof of Theorem 4.2

We prepare the following lemma to prove the theorem.

Lemma A.1 (*type substitution*)
If $\Gamma \cup \{x : \tau'\} \mid \Delta \vdash M : \tau$ and $\Gamma \mid \emptyset \vdash N : \tau'$ hold, $\Gamma \mid \Delta \vdash [N/x]M : \tau$.

Proof
Straightforward. □

Proof of Theorem 4.2. The first condition can be easily proved by induction on the derivation tree of $\Gamma \mid \langle\langle\delta\rangle\rangle \vdash M : \tau$. Here we only show the proof of the second condition.

From the assumption $(M, \delta) \to (M', \delta')$, there exist E_s and I that satisfy $M = E_s[I]$. We use structural induction on E_s.

- Case $E_s = [\,]$.
  Case analysis on I.
  - Case $I = (\lambda x.N)\ U$.
    First, suppose $U = i$ or $U = \lambda y.N'$ for some i or y and N'. Then, $\delta = \emptyset$

and $\Gamma \mid \emptyset \vdash U : \tau'$ and $\Gamma \cup \{x : \tau'\} \mid \emptyset \vdash N : \tau$ follow from the assumption $\Gamma \mid \langle\langle \delta \rangle\rangle \vdash M : \tau$ and inversion of T-App and T-Abs2. $M' = [U/x]N$ and $\delta' = \delta$ (and thus, $\delta' = \emptyset$) follow from Es2-App. Thus, $\Gamma \mid \langle\langle \delta' \rangle\rangle \vdash M' : \tau$ follows from Lemma A.1 as required.
Next, suppose $U = y$ for some y. Then, $M' = [y/x]N$ and $\delta' = \delta$ follow from Es2-App. $\langle\langle \delta \rangle\rangle = y : \mathbf{Tree}^-$ and $\Gamma \mid x : \mathbf{Tree}^- \vdash N : \tau$ follow from the assumption $\Gamma \mid \langle\langle \delta \rangle\rangle \vdash M : \tau$ and inversion of T-App and T-Abs1. Thus, as easily seen, $\Gamma \mid y : \mathbf{Tree}^- \vdash [y/x]N : \tau$. Thus, $\Gamma \mid \langle\langle \delta' \rangle\rangle \vdash M' : \tau$ follows as required.

— Case $I = (\mathbf{case}\ y\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2)$ with $\delta = (y \mapsto \mathbf{leaf}\ i, \delta'')$.
$\Gamma \mid \langle\langle \delta \rangle\rangle \vdash M : \tau$ must have been derived by using T-Case. Thus, we have $\Gamma \cup \{x : \mathbf{Int}\} \mid \langle\langle \delta'' \rangle\rangle \vdash M_1 : \tau$. Because $\Gamma \mid \emptyset \vdash i : \mathbf{Int}$, we have $\Gamma \mid \langle\langle \delta'' \rangle\rangle \vdash [i/x]M_1 : \tau$ from Lemma A.1. Because $M' = [i/x]M_1$ and $\delta' = \delta''$ follow from Es2-Case1, we have $\Gamma \mid \langle\langle \delta' \rangle\rangle \vdash M' : \tau$ as required.

— Case $I = (\mathbf{case}\ y\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2)$ with $\delta = (y \mapsto (\mathbf{node}\ V_1\ V_2), \delta'')$.
$\Gamma \mid \langle\langle \delta \rangle\rangle \vdash M : \tau$ must have been derived by using T-Case. Thus, we have $\Gamma \mid x_1 : \mathbf{Tree}^-, x_2 : \mathbf{Tree}^-, \langle\langle \delta'' \rangle\rangle \vdash M_2 : \tau$. Because $M' = M_2$ and $\delta' = x_1 \mapsto V_1, x_2 \mapsto V_2, \delta''$ follow from Es2-Case2, we have $\Gamma \mid \langle\langle \delta' \rangle\rangle \vdash M' : \tau$ as required.

- Case $E_s = (\mathbf{case}\ E'_s\ \mathbf{of}\ \mathbf{leaf} x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2)$.
$\Gamma \mid \langle\langle \delta \rangle\rangle \vdash M : \tau$ must have been derived by using T-Case. Thus, we have $\Gamma \mid \langle\langle \delta_1 \rangle\rangle \vdash E'_s[I] : \mathbf{Tree}^-$ and $\Gamma \cup \{x : \mathbf{Int}\} \mid \langle\langle \delta_2 \rangle\rangle \vdash M_1 : \tau$ and $\Gamma \mid x_1 : \mathbf{Tree}^-, x_2 : \mathbf{Tree}^-, \langle\langle \delta_2 \rangle\rangle \vdash M_2 : \tau$ and $\delta = \delta_1, \delta_2$ for some δ_1 and δ_2. By the induction hypothesis, there exist δ'_1 and M'' that satisfy $\Gamma \mid \langle\langle \delta'_1 \rangle\rangle \vdash M'' : \mathbf{Tree}^-$ and $(E'_s[I], \delta_1) \to (M'', \delta'_1)$. Because $M' = (\mathbf{case}\ M''\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2)$ and $\delta' = \delta'_1, \delta_2$, we have $\Gamma \mid \langle\langle \delta' \rangle\rangle \vdash M' : \tau$ from T-Case as required.

B Proof of Corollary 4.3

The following lemma guarantees that the reducibility in the original source language semantics and that in the revised semantics are equivalent *for well-typed programs.*

Lemma B.1
If $\emptyset \mid \langle\langle \delta \rangle\rangle \vdash M : \tau$ and if $\tau \in \{\mathbf{Int}, \mathbf{Tree}^+\}$, then $(M, \delta) \to^* (M', \delta')$ if and only if $\{\!|\delta|\!\}M \to \{\!|\delta'|\!\}M'$ where $\{\!|\cdot|\!\}$ is the function from ordered environments to substitutions defined by

$$\begin{array}{rcl} \{\!|\emptyset|\!\} & = & \text{(identity function)} \\ \{\!|x \mapsto V, \delta|\!\} & = & [V/x] \circ \{\!|\delta|\!\}. \end{array}$$

Proof
$(\Longrightarrow)$ is obvious. We only show $(\Longleftarrow)$ by induction on the length of $\{\!|\delta|\!\}M \to^* \{\!|\delta'|\!\}M'$ The case where $M = M'$ and $\delta = \delta'$ is obvious. Suppose that $\{\!|\delta|\!\}M \to$

$\{\!|\delta''|\!\}M'' \to^* \{\!|\delta'|\!\}M'$. From Theorem 4.2, we have $(M, \delta) \to (M''', \delta''')$ and $\emptyset \mid \langle\langle \delta''' \rangle\rangle \vdash M''' : \tau$ for some M''' and δ'''. From the ($\Longrightarrow$) direction of this Lemma, and from that the first semantics is deterministic, we have $M''' = M''$ and $\delta''' = \delta''$. Then, we have $(M'', \delta'') \to^* (M', \delta')$ from the induction hypothesis. $\square$

Lemma B.2
If $\emptyset \mid \Delta \vdash W : \tau$, then $\Delta = \emptyset$.

Proof
Case analysis on the last rule that derives $\emptyset \mid \Delta \vdash W : \tau$. $\square$

Proof of Corollary 4.3. From $\emptyset \mid \emptyset \vdash M : \mathbf{Tree}^- \to \tau$, we have $\emptyset \mid x : \mathbf{Tree}^- \vdash M\ x : \tau$. Thus, $(M\ x, x \mapsto V) \to^* (W, \delta)$ if and only if $M\ V \to^* \{\!|\delta|\!\}W$. (Note that $\{\!|x \mapsto V|\!\}(M\ x) = M\ V$.) It suffices to show that $\delta = \emptyset$. From Theorem 4.2, we have $\emptyset \mid \langle\langle \delta \rangle\rangle \vdash W : \tau$. Thus, we have $\delta = \emptyset$ from Lemma B.2.

C Proof of Lemma 4.5

For the proof of Lemma 4.5, we prepare the following lemma.

Lemma C.1
If $x \notin \gamma$, then $\mathcal{A}_\gamma([M_1/x]M_2) = [\mathcal{A}_\gamma(M_1)/x]\mathcal{A}_\gamma(M_2)$.

Proof
Induction on the structure of M_2. $\square$

Proof of Lemma 4.5. We prove $M \sim_{\mathbf{FV}(M)} (e, S_o)$. We hereafter write γ for $\mathbf{FV}(M)$ and S for S_o.

First, suppose that M is not reducible. Then, $M = U$ or $M = V$.

- Case $M = U$.
  Let $e = \mathcal{A}_\gamma(M)$ and $S = \emptyset$. Then $M \sim_\gamma (e, S)$ follows from C-Value. $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \to^* (e, \langle\delta\rangle, S)$ is obvious.
- Case $M = V$.
  Let $e = ()$ and $S = [\![V]\!]$. Then $M \sim_\gamma (e, S)$ follows from C-Tree.
  $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \to^* (e, \langle\delta\rangle, S)$ follows from the structural induction on V below:
  - Case $V = \mathbf{leaf}\ i$.
    In this case, $\mathcal{A}_\gamma(M) = (\mathbf{write}(\mathbf{leaf}); \mathbf{write}(i))$. Thus,
    $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \to^* (e, \langle\delta\rangle, S)$ holds.
  - Case $V = \mathbf{node}\ V_1\ V_2$.
    In this case, $\mathcal{A}_\gamma(M) = (\mathbf{write}(\mathbf{node}); \mathcal{A}_\gamma(V_1); \mathcal{A}_\gamma(V_2))$ and
    $S = \mathbf{node}; [\![V_1]\!]; [\![V_2]\!]$. Because $\emptyset \mid \emptyset \vdash V_1 : \mathbf{Tree}^+$ and $\emptyset \mid \emptyset \vdash V_2 : \mathbf{Tree}^+$, we have
    $(\mathcal{A}_\gamma(V_1), \emptyset, \emptyset) \to^* ((), \emptyset, [\![V_1]\!])$ and $(\mathcal{A}_\gamma(V_2), \emptyset, \emptyset) \to^* ((), \emptyset, [\![V_2]\!])$ from the induction hypothesis. Thus, $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \to^* (e, \langle\delta\rangle, S)$. (Note that $\delta = \emptyset$ because $\emptyset \mid \emptyset \vdash V : \mathbf{Tree}^+$.)

Next, suppose that M is reducible. Then, there exist E_s and I such as $M = E_s[I]$. We use structural induction on E_s. We show only important cases.

- Case $E_s = [\,]$.
  In this case, $M = I$. Let $e = \mathcal{A}_\gamma(M)$ and $S = \emptyset$. Then, $M \sim_\gamma (e, S)$ follows from C-Inst. $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \rightarrow^* (e, \langle\delta\rangle, S)$ is obvious.
- Case $E_s = \textbf{node}\ E_s'\ M'$.
  In this case, $M = \textbf{node}\ E_s'[I]\ M'$. We have $\emptyset \mid \langle\langle\delta_1\rangle\rangle \vdash E_s'[I] : \textbf{Tree}^+$ and $\emptyset \mid \langle\langle\delta_2\rangle\rangle \vdash M' : \textbf{Tree}^+$ and $\delta = \delta_1, \delta_2$ for some δ_1 and δ_2 from the assumption $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash M : \tau$. Thus, $E_s'[I] \sim_\gamma (e', S')$ and $(\mathcal{A}_\gamma(E_s'[I]), \langle\delta_1\rangle, \emptyset) \rightarrow^* (e', \langle\delta_1\rangle, S')$ follows for some e' and S' from the induction hypothesis. Let e be $e'; \mathcal{A}_\gamma(M')$ and S be $\textbf{node}; S'$. Then, $M \sim_\gamma (e, S)$ follows from C-Node1. Because $\mathcal{A}_\gamma(M) = \textbf{write}(\textbf{node}); \mathcal{A}_\gamma(E_s'[I]); \mathcal{A}_\gamma(M')$, we have $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \rightarrow^* (e, \langle\delta\rangle, S)$.
- Case $E_s = \textbf{node}\ V\ E_s'$.
  In this case, $M = \textbf{node}\ V\ E_s'[I]$. We have $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash E_s'[I] : \textbf{Tree}^+$ from the assumption $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash M{:}\tau$. Thus, $E_s'[I] \sim_\gamma (e', S')$ and $(\mathcal{A}_\gamma(E_s'[I]), \langle\delta\rangle, \emptyset) \rightarrow^* (e', \langle\delta\rangle, S')$ follow for some e' and S' from the induction hypothesis. Let e be e' and S be $\textbf{node}; [\![\, V\,]\!]; S'$. Then, $M \sim_\gamma (e, S)$ follows from C-Node2. Because $\mathcal{A}_\gamma(M) = \textbf{write}(\textbf{node});$
  $\mathcal{A}_\gamma(V); \mathcal{A}_\gamma(E_s'[I])$, we have $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \rightarrow^* (e, \langle\delta\rangle, S)$.
- Case $E_s = (\textbf{case}\ E_s'\ \textbf{of leaf}\ x \Rightarrow M_1 \mid \textbf{node}\ x_1\ x_2 \Rightarrow M_2)$.
  In this case, $M = (\textbf{case}\ E_s'[I]\ \textbf{of leaf}\ x \Rightarrow M_1 \mid \textbf{node}\ x_1\ x_2 \Rightarrow M_2)$. We have $\emptyset \mid \langle\langle\delta_1\rangle\rangle \vdash E_s'[I] : \textbf{Tree}^-$ and $x : \textbf{Int} \mid \langle\langle\delta_2\rangle\rangle \vdash M_1 : \tau$ and $\emptyset \mid x_1 : \textbf{Tree}^-, x_2 : \textbf{Tree}^-, \langle\langle\delta_2\rangle\rangle \vdash M_1 : \tau$ and $\delta = \delta_1, \delta_2$ for some δ_1 and δ_2 from the assumption $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash M : \tau$. Thus, $E_s'[I] \sim_\gamma (e', S')$ and $(\mathcal{A}_\gamma(E_s'[I]), \langle\delta_1\rangle, \emptyset) \rightarrow^* (e', \langle\delta_1\rangle, S')$ follows for some e' and S' from the induction hypothesis. Let e be
  $$\begin{array}{ll}\textbf{case}\ e'; \textbf{read}() & \textbf{leaf} \Rightarrow \textbf{let}\ x = \textbf{read}()\ \textbf{in}\ \mathcal{A}_\gamma(M_1) \\ & \textbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2)\end{array}$$
  and S be S'. Then, $M \sim_\gamma (e, S)$ follows from C-Case. Because $\mathcal{A}_\gamma(M) = \textbf{case}\ \mathcal{A}_\gamma(E_s'[I]); \textbf{read}()\ \textbf{of leaf} \Rightarrow \textbf{let}\ x = \textbf{read}()\ \textbf{in}\ \mathcal{A}_\gamma(M_1) \mid \textbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2)$. $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \rightarrow^* (e, \langle\delta\rangle, S)$ holds.

D Proof of Lemma 4.6

We prove Lemma 4.6 in this section.

Proof
The second property follows immediately from the definition of $M \sim_\gamma (e, S)$. (If M is irreducible, then $M \sim_\gamma (e, S)$ must follow either from C-Value or C-Tree, which implies that e is irreducible too.)

We prove the first property below. We prove $M' \sim_{\textbf{FV}(M)} (e', S_o')$. We hereafter write γ for $\textbf{FV}(M)$ and S' for S_o'.

Suppose $(M, \delta) \rightarrow (M', \delta')$. Then, $M = E_s[I]$ for some E_s and I. We use structural induction on E_s. We show only important cases.

- Case $E_s = [\,]$. Case analysis on I. We show only important cases.
  - Case $I = \textbf{case}\ y\ \textbf{of leaf}\ x \Rightarrow M_1 \mid \textbf{node}\ x_1\ x_2 \Rightarrow M_2$ with $\delta = (y \mapsto \textbf{leaf}\ i, \delta_1)$.
    $(M, \delta) \to (M', \delta')$ must have been derived by using Es2-Case1. So, it must be the case that $M' = [i/x]M_1$ and $\delta' = \delta_1$. $M \sim_\gamma (e, S)$ implies

    $$e = \mathcal{A}_\gamma(I) = \begin{array}{ll} \textbf{case}\ (); \textbf{read}() & \textbf{leaf} \Rightarrow \textbf{let}\ x = \textbf{read}()\ \textbf{in}\ \mathcal{A}_\gamma(M_1) \\ & \textbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2) \end{array}$$

    and $S = \emptyset$. $(e, \textbf{leaf}; i; \langle\delta_1\rangle, \emptyset) \to^+ (\mathcal{A}_\gamma(M'), \langle\delta_1\rangle, \emptyset)$ follows from Lemma C.1. By Lemma 4.5, there exist e'' and S'' that satisfy $(\mathcal{A}_\gamma(M'), \langle\delta_1\rangle, \emptyset) \to (e'', \langle\delta_1\rangle, S'')$ and $M' \sim_\gamma (e'', S'')$. Let e' be e'' and S' be S''. Then, we have $(e, \langle\delta\rangle, S) \to^+ (e', \langle\delta'\rangle, S')$ and $M' \sim_\gamma (e', S')$ as required.
  - Case $I = \textbf{case}\ y\ \textbf{of leaf}\ x \Rightarrow M_1 \mid \textbf{node}\ x_1\ x_2 \Rightarrow M_2$ with $\delta = (y \mapsto \textbf{node}\ V_1\ V_2, \delta_1)$.
    $(M, \delta) \to (M', \delta')$ must have been derived by using Es2-Case2. So, it must be the case that $M' = M_2$ and $\delta' = x_1 \mapsto V_1, x_2 \mapsto V_2, \delta_1$. $M \sim_\gamma (e, S)$ implies

    $$e = \mathcal{A}_\gamma(I) = \begin{array}{ll} \textbf{case}\ (); \textbf{read}() & \textbf{leaf} \Rightarrow \textbf{let}\ x = \textbf{read}()\ \textbf{in}\ \mathcal{A}_\gamma(M_1) \\ & \textbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2) \end{array}$$

    and $S = \emptyset$. As easily seen, $[()/x_1, ()/x2]\mathcal{A}_\gamma(M_2) = \mathcal{A}_{\gamma\cup\{x_1,x_2\}}(M_2)$.Thus, $(e, \textbf{node}; [\![V_1]\!]; [\![V_2]\!]; \langle\delta_1\rangle, \emptyset) \to^+ (\mathcal{A}_{\gamma\cup\{x_1,x_2\}}(M_2), [\![V_1]\!]; [\![V_2]\!]; \langle\delta_1\rangle, \emptyset)$. By Lemma4.5, there exist e'' and S'' that satisfy
    $(\mathcal{A}_{\gamma\cup\{x_1,x_2\}}(M_2), [\![V_1]\!]; [\![V_2]\!]; \langle\delta_1\rangle, \emptyset) \to^*$
    $(e'', [\![V_1]\!]; [\![V_2]\!]; \langle\delta_1\rangle, S'')$ and $M_2 \sim_\gamma (e'', S'')$. Let $e' = e''$ and $S' = S''$. Then, we have $(e, \langle\delta\rangle, S) \to^+ (e', \langle\delta'\rangle, S')$ and $M' \sim_\gamma (e', S')$ as required.
- Case $E_s = \textbf{leaf}\ E_1$.
  There exists M_1' that satisfies $M' = \textbf{leaf}\ M_1'$ and $(E_1[I], \delta) \to (M_1', \delta')$. $M \sim_\gamma (e, S)$ must have been derived from C-Leaf. Thus, there exist e_1 and S_1 that satisfies $e = \textbf{write}(e_1)$ and $E_1[I] \sim_\gamma (e_1, S_1)$ and $S = \textbf{leaf}; S_1$. Because M is well-typed, $E_1[I]$ is also well-typed. Thus, from the induction hypothesis, we have $(e_1, \langle\delta\rangle, S_1) \to^+ (e_1', \langle\delta'\rangle, S_1')$ and $M_1' \sim_\gamma (e_1', S_1')$ for some e_1' and S_1'.
  First, suppose that M_1' is reducible. Let e' be $\textbf{write}(e_1')$ and S' be $\textbf{leaf}; S_1$. Then, $M' \sim_\gamma (e', S')$ follows from C-Leaf as required.
  Next, suppose that M_1' is a value. Because M is well-typed, M_1' is an integer (let the integer be i_1') and $M_1' \sim_\gamma (e_1', S_1')$ must have been derived from C-Value. Thus, $e_1' = i_1'$ and $S_1' = \emptyset$. Let e' be $()$ and S' be $\textbf{leaf}; i_1'$. Then, $M' \sim_\gamma (e', S')$ follows from C-Tree and $(\textbf{write}(e_1'), \langle\delta'\rangle, \textbf{leaf}; S_1') \to (e', \langle\delta'\rangle, S')$ holds.
- Case $E_s = \textbf{node}\ E_1\ M_2$.
  There exists M_1' that satisfies $M' = \textbf{node}\ M_1'\ M_2$ and $(E_1[I], \delta) \to (M_1', \delta')$. $M \sim_\gamma (e, S)$ must have been derived from C-Node1. Thus, there exist e_1 and S_1 that satisfies $e = e_1; \mathcal{A}_\gamma(M_2)$ and $E_1[I] \sim_\gamma (e_1, S_1)$ and $S = \textbf{node}; S_1$. Because we assume that M is well-typed, $E_1[I]$ is also well-typed. Thus, from

the induction hypothesis, we have $(e_1;\mathcal{A}_\gamma(M_2), \langle\delta\rangle, \mathbf{node}; S_1) \to^+ (e'_1;\mathcal{A}_\gamma(M_2), \langle\delta'\rangle, \mathbf{node}; S'_1)$ and $M'_1 \sim_\gamma (e'_1, S'_1)$ for some e'_1 and S'_1.
First, suppose that M'_1 is reducible. Let e' be $e'_1;\mathcal{A}_\gamma(M_2)$ and S' be $\mathbf{node}; S'_1$. Then, $M' \sim_\gamma (e', S')$ follows from C-Node1 as required.
Next, suppose that M'_1 is a value (let the value be V'_1). $M'_1 \sim_\gamma (e'_1, S_1)$ must have been derived from C-Tree. Thus, $e'_1 = ()$ and $S_1 = [\![V'_1]\!]$. Thus, $(e, \langle\delta\rangle, S) \to^+ (\mathcal{A}_\gamma(M_2), \langle\delta'\rangle, \mathbf{node}; [\![V'_1]\!])$. From Lemma 4.5, there exist e_2 and S_2 that satisfy $M_2 \sim_\gamma (e_2, S_2)$ and $(\mathcal{A}_\gamma(M_2), \langle\delta'\rangle, \mathbf{node}; [\![V'_1]\!]) \to^* (e_2, \langle\delta'\rangle, \mathbf{node}; [\![V'_1]\!]; S_2)$. Let e' be e_2 and S' be $\mathbf{node}; [\![V'_1]\!]; S_2$. Then, $M' \sim_\gamma (e', S')$ follows from C-Node2 and $(e, \langle\delta\rangle, S) \to^+ (e', \langle\delta'\rangle, S')$ holds.

- Case $E_s = \mathbf{node}\ V_2\ E_1$.
There exists M'_1 that satisfies $M' = \mathbf{node}\ V_2\ M'_1$ and $(E_1[I], \delta) \to (M'_1, \delta')$. $M \sim_\gamma (e, S)$ must have been derived from C-Node2. Thus, there exist e_1 and S_1 that satisfies $e = e_1$ and $E_1[I] \sim_\gamma (e_1, S_1)$ and $S = \mathbf{node}; [\![V_2]\!]; S_1$. Because M is well-typed, $E_1[I]$ is also well-typed. Thus, from the induction hypothesis, we have $(e_1, \langle\delta\rangle, \mathbf{node}; [\![V_2]\!]; S_1) \to^+ (e'_1, \langle\delta'\rangle, \mathbf{node}; [\![V_2]\!]; S'_1)$ and $M'_1 \sim_\gamma (e'_1, S'_1)$ for some e'_1 and S'_1.
First, suppose that M'_1 is reducible. Let e' be e'_1 and S' be $\mathbf{node}; [\![V_2]\!]; S'_1$. Then, $M' \sim_\gamma (e', S')$ follows from C-Node1 as required.
Next, suppose that M'_1 is a value (let the value be V'_1). $M'_1 \sim_\gamma (e'_1, S_1)$ must have been derived from C-Tree. Thus, $e'_1 = ()$ and $S_1 = [\![V'_1]\!]$, and thus, $(e, \langle\delta\rangle, S) \to^+ ((), \langle\delta'\rangle, \mathbf{node}; [\![V_2]\!]; [\![V'_1]\!])$. Let e' be $()$ and S' be $\mathbf{node}; [\![V_2]\!]; [\![V'_1]\!]$. Then, $M' \sim_\gamma (e', S')$ follows from C-Tree as required.
- Case $E_s = (\mathbf{case}\ E_1\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2)$.
There exists M'_1 that satisfies

$$M' = \begin{array}{lll} \mathbf{case}\ M'_1 & \mathbf{of} & \mathbf{leaf}\ x \Rightarrow M_1 \\ & \mid & \mathbf{node}\ x_1\ x_2 \Rightarrow M_2 \end{array}$$

and $(E_1[I], \delta) \to (M'_1, \delta')$. $M \sim_\gamma (e, S)$ must have been derived from C-Case. Thus, there exists e_1 that satisfies

$$e = \begin{array}{lll} \mathbf{case}\ e_1; \mathbf{read}() & \mathbf{of} & \mathbf{leaf} \Rightarrow \mathbf{let}\ x = \mathbf{read}()\ \mathbf{in}\ M_1 \\ & \mid & \mathbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2) \end{array}$$

and $E_1[I] \sim_\gamma (e_1, S)$. Because M is well-typed, $E_1[I]$ is also well-typed. Thus, from the induction hypothesis, we have

$$\begin{array}{l} (\begin{array}{lll} \mathbf{case}\ e_1; \mathbf{read}() & \mathbf{of} & \mathbf{leaf} \Rightarrow \mathbf{let}\ x = \mathbf{read}()\ \mathbf{in}\ M_1 \\ & \mid & \mathbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2) \end{array}, \langle\delta\rangle, S) \to^+ \\ (\begin{array}{lll} \mathbf{case}\ e'_1; \mathbf{read}() & \mathbf{of} & \mathbf{leaf} \Rightarrow \mathbf{let}\ x = \mathbf{read}()\ \mathbf{in}\ M_1 \\ & \mid & \mathbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2) \end{array}, \langle\delta'\rangle, S'') \end{array}$$

and $M'_1 \sim_\gamma (e'_1, S'')$ for some e'_1 and S''.
First, suppose that M'_1 is reducible. Let e' be

$$\begin{array}{lll} \mathbf{case}\ e'_1; \mathbf{read}() & \mathbf{of} & \mathbf{leaf} \Rightarrow \mathbf{let}\ x = \mathbf{read}()\ \mathbf{in}\ M_1 \\ & \mid & \mathbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2) \end{array}$$

and S' be S''. Then, $M' \sim_\gamma (e', S')$ follows from C-Case.
Next, suppose that M'_1 is not reducible. Because M'_1 is a tree-typed term, M'_1 is a variable (let it be y'_1). Because $M'_1 \sim_\gamma (e'_1, S'')$ must have been derived from C-Value, $e'_1 = \mathcal{A}_\gamma(y'_1)$. Thus, $(e, \langle\delta\rangle, S) \to^+ (\mathcal{A}_\gamma(M'), \langle\delta'\rangle, S'')$. From Lemma 4.5, there exist e'' and S_1 that satisfy $M' \sim_\gamma (e'', S_1)$ and $(\mathcal{A}_\gamma(M'), \langle\delta'\rangle, S'') \to^* (e'', \langle\delta'\rangle, S_1)$. Let e' be e'' and S' be S_1. Then, $(e, \langle\delta\rangle, S) \to^+ (e', \langle\delta'\rangle, S')$ and $M' \sim_\gamma (e', S')$ hold as required.

□

E Proof of Theorem 4.4

This section proves Theorem 4.4.

Proof
First of all, note that $\emptyset \mid x : \mathbf{Tree}^- \vdash M\ x : \tau$ follows an assumption $\emptyset \mid \emptyset \vdash M : \mathbf{Tree}^- \to \tau$ and $\emptyset \mid x : \mathbf{Tree}^- \vdash x : \mathbf{Tree}^-$.

We prove only (ii) hereafter. (i) can be proved in the same way.

Assume $((M\ x), x \mapsto V) \to^* (V', \emptyset)$. Because $\emptyset \mid x{:}\mathbf{Tree}^- \vdash (M\ x) : \tau$ holds, there exist e, S_i and S_o such that $((M\ x), x \mapsto V) \sim (e, S_i, S_o)$ and $(\mathcal{A}(M)(), S_i, \emptyset) \to^* (e, S_i, S_o)$ from Lemma 4.5 [7]. From the definition of $\sim$, $S_i = [\![V]\!]$. Because of Theorem 4.2 and Lemma 4.6, there exists a sequence of reduction $(e, [\![V]\!], S_o) \to^* (e', \emptyset, S'_o)$ that satisfies $(V', \emptyset) \sim (e', \emptyset, S'_o)$. From the definition of $\sim$, $e' = ()$ and $S'_o = [\![V']\!]$. Thus, $(\mathcal{A}(M)(), [\![V]\!], \emptyset) \to^* ((), \emptyset, [\![V']\!])$ holds.

Next, assume $(\mathcal{A}(M)(), [\![V]\!], \emptyset) \to^* ((), \emptyset, [\![V']\!])$. As we stated above, there exist e, S_i and S_o such that $((M\ x), x \mapsto V) \sim (e, [\![V]\!], S_o)$ and $(\mathcal{A}(M)(), [\![V]\!], \emptyset) \to^* (e, [\![V]\!], S_o)$. Because applicable reduction rule can be uniquely determined at each step of reduction, $(e, [\![V]\!], S_o) \to^* ((), \emptyset, [\![V']\!])$ holds.

In the following, we prove "if $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash M'{:}\mathbf{Tree}^+$ and $(e, \langle\delta\rangle, S'_o) \to^* ((), \emptyset, [\![V']\!]))$ and $(M', \delta) \sim (e, S'_i, S'_o)$ hold, $(M', \delta) \to^* (V', \emptyset)$ holds". We use mathematical induction on the number of reduction step of $(e, \langle\delta\rangle, S'_o) \to^* ((), \emptyset, [\![V']\!]))$. With this fact, by letting M' be $M\ x$ and δ be $x \mapsto V$, $((M\ x), x \mapsto V) \to^* (V', \emptyset)$ holds because $\emptyset \mid x : \mathbf{Tree}^- \vdash (M\ x) : \mathbf{Tree}^+$ follows $((M\ x), x \mapsto V) \to^* (V, \emptyset)$ and Theorem 4.2.

- In the case of $n = 0$, $M' = V'$ and $\delta = \emptyset$ hold because $e = (), \langle\delta\rangle = \emptyset, S'_o = [\![V']\!]$ and $(M', \emptyset) \sim (e, \emptyset, S'_o)$ hold. Thus, $(M', \delta) \to^* (V', \emptyset)$ holds.
- In the case of $n \geq 1$, there exist e', S_i, S''_o that satisfies

$$(e, \langle\delta\rangle, S'_o) \to (e', S_i, S''_o) \to^* ((), \emptyset, [\![V']\!]))$$

From Lemma 4.6, there exist M'', δ', S'''_o that satisfies

— $(M', \delta) \to (M'', \delta')$
— $(M'', \delta') \sim (e'', \langle\delta'\rangle, S'''_o)$
— $(e, \langle\delta\rangle, S'_o) \to^+ (e'', \langle\delta'\rangle, S'''_o)$

[7] Because $\emptyset \mid \emptyset \vdash M : \mathbf{Tree}^- \to \tau$ holds, $\mathbf{FV}(M) = \emptyset$. Thus, $\mathcal{A}_{\mathbf{FV}(M)\cup\{x\}}(M\ x) = \mathcal{A}(M)()$.

Since the reduction is deterministic, $(e'', \langle\delta'\rangle, S_o''') \rightarrow^* ((), \emptyset, [\![V']\!]))$ holds. From the induction hypothesis, $(M'', \delta') \rightarrow^* (V', \emptyset)$. Thus, $(M', \delta) \rightarrow^* (V', \emptyset)$ holds.

□