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

Koichi Kodama[*], Kohei Suenaga[**], and Naoki Kobayashi[***]

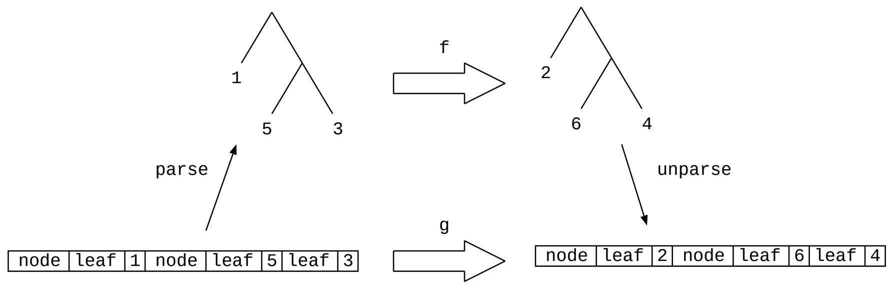
[*]Tokyo Institute of Technology, `kodama@kb.cs.titech.ac.jp`
[**]University of Tokyo, `kohei@yl.is.s.u-tokyo.ac.jp`
[***]Tokyo Institute of Technology, `kobayasi@kb.cs.titech.ac.jp`

Abstract. There are two ways to write a program for manipulating tree-structured data such as XML documents and S-expressions: 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 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.

1 Introduction

There are two ways to write a program for manipulating tree-structured data such as XML documents [3] and S-expressions: 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 Mode) API and programming language XDuce [5] 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 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


f
1
5
3
2
6
4
parse
unparse
g
node leaf 1 node leaf 5 leaf 3
node leaf 2 node leaf 6 leaf 4

Tree-processing program f:
$\mathbf{fix}\ f.\lambda t.(\mathbf{case}\ t\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow \mathbf{leaf}\ (x+1) \mid \mathbf{node}\ x_1\ x_2 \Rightarrow \mathbf{node}\ (f\ x_1)\ (f\ x_2))$
Stream-processing program: g

```
fix g.λt.(case read() of leaf ⇒ leaf (read() + 1)
                       | node ⇒ write(node); g (); g ())
```

Fig. 1. Tree-processing and stream-processing

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 [14]) 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 use a λ-calculus with primitives on binary trees, and show how the translation works.

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 manner* and (2) in that case, we can obtain from the tree-processing program the corresponding stream-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 [11, 12]. Ordered linear types, which are an extension of linear types [2, 13], describe not only how often but also *in which order* data are used. Our type system designed based on the ordered linear types guaran-

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

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

tees that a well-typed program traverses and constructs trees from left to right and in the depth-first order. 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 apply the above-mentioned translation to the program in Figure 2.

The rest of the paper is organized as follows: To clarify the essence, we first focus on a minimal calculus in Section 2–4. In Section 2, we define the source language and the target language of the translation. We define a type system of the source language in Section 3. Section 4 presents a translation algorithm, shows its correctness and discuss 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. (Strictly speaking, one can still store some information about trees by encoding it into lambda-terms.) Section 5 describes several extensions to recover the expressive power. With the extensions, one can write a program that selectively buffers input/output trees on memory, while the type system guarantees that the buffering is correctly performed. After discussing related work in Section 6, we conclude in Section 7.

For the restriction of space, proofs are omitted in this paper. They are found in the full version [7].

2 Language

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.

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. ($\textbf{case}\ M\ \textbf{of leaf}\ x \Rightarrow M_1 \mid \textbf{node}\ x_1\ x_2 \Rightarrow M_2$) 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.

Terms, values and evaluation contexts:

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

Reduction rules:

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

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

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

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

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

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

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 **let** $x = M_1$ **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.

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 (**case** e **of leaf** $\Rightarrow e_1 \mid$ **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. **fix** $f.e$ is a recursive function that satisfies $f = e$. Bound and free variables are defined as usual.

We write **let** $x = e_1$ **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 target program assumes that the input stream represents a valid tree. If the input stream is in a wrong format (e.g., when the stream is **node**; 1; 2), the execution gets stuck.

Terms, values and evaluation contexts:

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

Reduction rules:

$$(E_t[v_1 + v_2], S_i, S_o) \longrightarrow (E_t[\mathit{plus}(v_1, v_2)], S_i, S_o) \quad \text{(ET-PLUS)}$$

$$(E_t[(\lambda x.M)v], S_i, S_o) \longrightarrow (E_t[[v/x]M], S_i, S_o) \quad \text{(ET-APP)}$$

$$(E_t[\mathbf{fix}\ f.e], S_i, S_o) \longrightarrow (E_t[[\mathbf{fix}\ f.e/f]e], S_i, S_o) \quad \text{(ET-FIX)}$$

$$(E_t[\mathbf{read}()], v; S_i, S_o) \longrightarrow (E_t[v], S_i, S_o) \quad \text{(ET-READ)}$$

$$(E_t[\mathbf{write}\ v], S_i, S_o) \longrightarrow (E_t[()], S_i, S_o; v) \quad \text{(when } v \text{ is an integer, } \mathbf{leaf} \text{ or } \mathbf{node}\text{)} \quad \text{(ET-WRITE)}$$

$$(E_t[\mathbf{case\ leaf\ of\ leaf} \Rightarrow e_1 \mid \mathbf{node} \Rightarrow e_2], S_i, S_o) \longrightarrow (E_t[e_1], S_i, S_o) \quad \text{(ET-CASE1)}$$

$$(E_t[\mathbf{case\ node\ of\ leaf} \Rightarrow e_1 \mid \mathbf{node} \Rightarrow e_2], S_i, S_o) \longrightarrow (E_t[e_2], S_i, S_o) \quad \text{(ET-CASE2)}$$

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

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

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

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 depth-first order. 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 depth-first access order on input trees, we use ordered linear types [11, 12].

3.1 Type and Type Environment

Definition 1 (Type). The set of *types*, ranged over by τ, is defined by:

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

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 2 (Ordered Linear Type Environment). An *ordered linear type environment* 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 subtrees bound to $x_1, \ldots, x_n$ must be accessed in the left-to-right, depth-first order.

Definition 3 (Non-Ordered Type Environment). *A* (non-ordered) type environment *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* $\mathbf{Tree}^d$.

We use the meta-variable Γ for non-ordered type environments. We often write $\Gamma, x : \tau$ for $\Gamma \cup \{x : \tau\}$, and write $x_1 : \tau_1, \ldots, x_n : \tau_n$ for $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$.

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 manner) order.

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 and Δ is an ordered linear type environment. 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. We explain only main rules below. T-Var1, T-Var2 and T-Int are the rules for

$$\Gamma \mid x : \mathbf{Tree}^- \vdash x : \mathbf{Tree}^- \qquad \text{(T-Var1)}$$

$$\Gamma, x : \tau \mid \emptyset \vdash x : \tau \qquad \text{(T-Var2)}$$

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

$$\frac{\Gamma \mid x : \mathbf{Tree}^- \vdash M : \tau}{\Gamma \mid \emptyset \vdash \lambda x.M : \mathbf{Tree}^- \to \tau} \qquad \text{(T-Abs1)}$$

$$\frac{\Gamma, x : \tau_1 \mid \emptyset \vdash M : \tau_2}{\Gamma \mid \emptyset \vdash \lambda x.M : \tau_1 \to \tau_2} \qquad \text{(T-Abs2)}$$

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

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

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

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

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

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

Fig. 6. Rules of typing judgment

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]

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-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 $x_1 : \mathbf{Tree}^-, x_2 : \mathbf{Tree}^-, \Delta_2$, the ordered linear type environment of M_2.

3.3 Examples of Well-typed Programs

Figure 7 shows more examples of well-typed source programs. The first and second programs (or the catamorphism [8]) 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 the value is passed to g.) One can also write functions that process nodes in a non-uniform manner, like the third program in Figure 7 (which increments the value of each leaf whose depth is odd).

The fourth program takes a tree as an input and returns the right subtree. Due to the restriction imposed by the type system, the program uses subfunctions *copy_tree* and *skip_tree* for explicitly copying and skipping trees.[2] (See Section 7 for a method for automatically inserting those functions.)

4 Translation Algorithm

In this section, we define a translation algorithm for well-typed source programs and prove its correctness.

4.1 Definition and Correctness of Translation

The translation algorithm $\mathcal{A}$ is shown in Figure 8. $\mathcal{A}$ maps a source program to a target program, preserving the structure of the source program and replacing operations on trees with operations on streams.

[1] We can relax the restriction by controlling usage of not only trees but also functions, as in the resource usage analysis [6]. The resulting type system would, however, become very complex.

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

$$\frac{\Gamma \mid \Delta_1 \vdash M_1 : \tau' \qquad \Gamma \mid \Delta_2 \vdash M_2 : \tau \qquad \tau' \neq \mathbf{Tree}^d}{\Gamma \mid \Delta_1, \Delta_2 \vdash M_1; M_2 : \tau.} \qquad \text{(T-Seq)}$$

```
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.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_alt y1) (inc_alt y2))
        (case x2 of leaf z ⇒ leaf (z + 1)
                | node z1 z2 ⇒ node (inc_alt z1) (inc_alt z2))

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))
```

Fig. 7. Examples of well-typed programs.

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

Definition 4. A function $[\![\,\cdot\,]\!]$ from the set of trees to the set of streams is defined by:

$$[\![\,\textbf{leaf}\ i\,]\!] = \textbf{leaf}; i$$
$$[\![\,\textbf{node}\ V_1\ V_2\,]\!] = \textbf{node}; [\![\,V_1\,]\!]; [\![\,V_2\,]\!]\,.$$

Theorem 1 (Correctness of Translation).
If $\emptyset \mid \emptyset \vdash M : \textbf{Tree}^- \rightarrow \tau$ *and* τ *is* **Int** *or* $\textbf{Tree}^+$*, the following properties hold for any tree value* V*:*

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

The above theorem means that a source program and the corresponding target program evaluates 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 1 below. Full proofs are found in the full version of this paper [7]. We define another reduction semantics of the source language and prove that (1) for well-typed programs, the new semantics is equivalent to the one in Section 2 and (2) each reduction step based on the new semantics has the corresponding one in the target program.

The new reduction relation is of the form $(M, \delta) \longrightarrow (M', \delta')$ where δ is a sequence of binding of the form $x \mapsto V$. The formal definition is given in the full version [7]. The only difference from the one defined in Section 2 is that input trees are bound in δ and must be accessed in the order specified by δ. So, evaluation based on the new rules can differ from the one in Section 2 only when

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

Fig. 8. Translation Algorithm

$$\begin{tikzcd}
 & (Mx, x \mapsto V) \arrow[r] \arrow[dl, bend right] \arrow[d, phantom, "\wr"] & (M', \delta') \arrow[r] \arrow[d, phantom, "\wr"] & \cdots \arrow[r] & (V', \emptyset) \arrow[d, phantom, "\wr"] \\
(\mathcal{A}(M), [\![V]\!], \emptyset) \arrow[r, "*"] & (e, [\![V]\!], S_o) \arrow[r, "+"] & (e', S_i', S_o') \arrow[r, "+"] & \cdots \arrow[r, "+"] & ((), \emptyset, [\![V']\!])
\end{tikzcd}$$

Fig. 9. Evaluation of a source and the target program.

the latter one succeeds while the former one gets stuck due to the restriction on access to input trees. The following theorem guarantees that this does not happen if the program is well-typed.

Theorem 2 (soundness of the type system). *If* $\emptyset \mid x : \mathbf{Tree}^- \vdash M : \tau$ *and* $(M, x \mapsto V) \longrightarrow^* (M', \delta')$ *hold, then* M' *is a value or a variable, or* $(M', \delta') \longrightarrow (M'', \delta'')$ *holds for some* M'', δ''*.*

Using the new semantics, we can prove that each reduction step of a source program has the corresponding one in the target program. Figure 9 illustrates the idea of the proof (for the case where the result is a tree). 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 output streams. For example, $(\mathbf{node}(\mathbf{leaf}\ 1)(\mathbf{leaf}\ (2+3)), \emptyset)$ corresponds to $(2+3, \emptyset, \mathbf{node}; \mathbf{leaf}; 1; \mathbf{leaf})$. The formal definition of $\sim$ is found in the full version [7].) We can show that (1) the target program $\mathcal{A}(M)$ can always be reduced to a state corresponding to the inital state of the source program M and that (2) reductions and the correspondence relation commute. Those imply that the whole diagram in Figure 9 commutes, so that the source program and the target program evaluates to the same value.

4.2 Efficiency of Translated Programs

Let M be a source program of type $\mathbf{Tree}^- \rightarrow \mathbf{Tree}^+$. We argue below that the target program $\mathcal{A}(M)$ runs more efficiently than the source program *unparse* $\circ$

$M \circ \textit{parse}$, 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 $\mathcal{A}(M) = \textit{unparse} \circ M \circ \textit{parse}$, obviously there would be no improvement.

Intuitively, 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.

In fact, by looking at the proof of Theorem 1, we know (see the full version for the reason):

– The memory space allocated by $\mathcal{A}(M)$ is less than the one allocated by $\textit{unparse} \circ M \circ \textit{parse}$, 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 $\textit{unparse} \circ M \circ \textit{parse}$ (up to an implementation-dependent constant factor).

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 Extensions

So far, we have focused on a minimal calculus to clarify the essence of our framework. This section briefly shows how to extend the framework to be used in practice. More details are found in the full version [7]

5.1 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 prgorams. Let us extend the syntax of the source and target languages as follows:

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

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 **MTree**, the type of trees stored on memory. The type system imposes no restriction on access order between variables of type **MTree** like type **Int** (so **MTree** is put in the ordinary type environment, not the ordered

```
fix $strm\_to\_mem$.
    $\lambda t.$case $t$ of leaf $x$ $\Rightarrow$ mleaf $x$
                 | node $x_1$ $x_2$ $\Rightarrow$ mnode ($strm\_to\_mem$ $x_1$) ($strm\_to\_mem$ $x_2$)
fix $mem\_to\_strm$.
    $\lambda t.$mcase $t$ of mleaf $x$ $\Rightarrow$ leaf $x$
                 | mnode $x_1$ $x_2$ $\Rightarrow$ node ($mem\_to\_strm$ $x_1$) ($mem\_to\_strm$ $x_2$)
```

Fig. 10. Definition of *strm_to_mem* and *mem_to_strm*

```
let $mswap$ =
    fix $f.\lambda$ $t.$mcase $t$ of mleaf $x$ $\Rightarrow$ leaf $x$
                        | mnode $x_1$ $x_2$ $\Rightarrow$ node ($f$ $x_2$) ($f$ $x_1$) in
    fix $swap\_deep.\lambda n.\lambda t.$
        if $n = 0$ then $mswap$ ($strm\_to\_mem$ $t$)
        else
            case $t$ of
                leaf $x \Rightarrow$ leaf $x$
            | node $x_1$ $x_2 \Rightarrow$ node ($swap\_deep$ $(n-1)$ $x_1$) ($swap\_deep$ $(n-1)$ $x_2$)
```

Fig. 11. A program which swaps children of nodes whose depth is more than n

linear type environment). The translation algorithm $\mathcal{A}$ simply translates a source program, preserving the structure:

$$\begin{array}{c}\mathcal{A}(\textbf{mleaf}\ M) = \textbf{mleaf}\ \mathcal{A}(M)\\ \mathcal{A}(\textbf{mnode}\ M_1\ M_2) = \textbf{mnode}\ \mathcal{A}(M_1)\ \mathcal{A}(M_2)\\ \cdots\end{array}$$

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 10.

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 11, which swaps children of nodes whose depth is more than n, only buffers the nodes whose depth is more than n.

The proof of Theorem 1 can be easily adapted for the extended language.

5.2 Side Effects and Multiple Input Trees

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, by introducing pair constructors and refining the type judgment form to $\Gamma \mid \{s_1 : \Delta_1, \ldots, s_n : \Delta_n\} \vdash M : \tau$ where $s_1, \ldots, s_n$ are the names of input streams and each of $\Delta_1, \ldots, \Delta$ is an ordered linear type environment.

5.3 Extention for Dealing with XML

We discuss below how to extend our method to deal with XML documents.

The difference between binary trees and XML documents is that the latter ones (i) are rose 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 following types (we use ML-style type declarations):

```
datatype tree = leaf of int | node of tree*tree;
datatype xmltree = leaf of pcdata
                 | node of label * attribute * treelist
and treelist = nil | cons of xmltree * treelist;
```

While the type `tree` represents binary trees, `xmltree` represents rose trees. Based on the difference between these types, we can replace the **case**-construct of the source language with the following two **case**-constructs.

$$\begin{array}{l}\mathbf{caseElem}\ t\ \mathbf{of}\ \mathbf{leaf}(x) \Rightarrow M_1 \mid \mathbf{node}(l, \mathit{attr}, \mathit{tl}) \Rightarrow M_2 \\ \mathbf{caseSeq}\ \mathit{tl}\ \mathbf{of}\ \mathbf{nil} \Rightarrow M_1 \mid \mathbf{cons}(x, \mathit{xl}) \Rightarrow M_2.\end{array}$$

Typing rules can also be naturally extended. For example, the typing rule for the latter construct is:

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

The restriction on the access order is expressed by $x : \mathbf{xmltree}, \mathit{xl} : \mathbf{treelist}, \Delta_2$ as in T-Node.

The translation algorithm (1) maps the pattern **nil** in the source language to the pattern for closing tags. (2) prepares a stack and confirms well-formedness of input documents.

6 Related Work

Nakano and Nishimura [9, 10] proposed a method for translating tree-processing programs to stream-processing programs using attribute grammars. In their method, programmers write XML processing with an attribute grammar. Then, the grammar is composed with parsing and unparsing grammars by using the descriptional composition [4] and translated to a grammar that directly deals with streams. *Quasi-SSUR condition* in [10] and *single use requirement* in [9], which force attributes of non-terminal symbols to 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 a part of trees in memory, so that the translation may not improve the efficiency. On the other hand, an advantage of our method is

```
$N \to$ node $N_1$ $N_2$
        $N_1.inh = f_1\ N.inh; N_2.inh = f_2\ N.inh\ N_1.syn\ N_1.inh$
        $N.syn = f_3\ N.inh\ N_1.syn\ N_1.inh\ N_2.syn\ N_2.inh$
$N \to$ leaf $i$
        $N.syn = f_4\ N.inh\ i$

fix $f.\lambda inh.\lambda t.$case $t$ of
        leaf $x \Rightarrow f_4\ inh\ x$
        node $x_1\ x_2 \Rightarrow$   let $N_1.inh = f_1\ inh$ in
                             let $N_1.syn = f\ N_1.inh\ x_1$ in
                             let $N_2.inh = f_2\ N.inh\ N_1.syn\ N_1.inh$ in
                             let $N_2.syn = f\ N_2.inh\ x_2$ in
                                 $f_3\ N.inh\ N_1.syn\ N_1.inh\ N_2.syn\ N_2.inh$
```

Fig. 12. L-attributed grammar over binary trees and corresponding program.

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 [1]. In fact, any L-attributed grammar over the binary tree can be expressed as a program as shown in Figure 12. 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.

There are many studies on program transformation [8, 14] 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 $unparse \circ f \circ parse$, 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 [14] or a catamorphism [8], but the result of deforestation is not an expected stream-processing program.

Actually, there are many similarities between the restriction of treeless program [14] 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-order functions. The type-based approach is also useful for automatic inference of selective buffering of trees, as discussed in Section 7.

The type system we used in this paper is based on the ordered linear logic proposed by Polakow [12]. 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. [11] 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.

7 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.

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. We are currently studying a type-directed, source-to-source translation for automatically inserting these functions.

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 correspond 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 members of “Programming Language Principles” group at University of Tokyo and Tokyo Institute of Technology.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers.* Addison-Wesley Pub Co, 1986.

2. Henry G. Baker. Lively linear lisp – look ma, no garbage! *ACM SIGPLAN Notices*, 27(8):89–98, 1992.
3. Tim Bray, Jean Paoli, C.M.Sperberg-McQueen, and Eve Maler. Extensible markup language (XML) 1.0 (second edition). Technical report, World Wide Web Consortium, October 2000. `http://www.w3.org/TR/REC-xml`.
4. Harald Ganzinger and Robert Giegerich. Attribute coupled grammars. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, 1984.
5. Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology (TOIT)*, 3(2):117–148, 2003.
6. Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 331–342, 2002.
7. Koichi Kodama, Kohei Suenaga, and Naoki Kobayashi. Translation of tree-processing programs into stream-processing programs based on ordered linear type. Full paper. Available from `http://www.yl.is.s.u-tokyo.ac.jp/~kohei/doc/paper/translation.pdf`.
8. Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 124 – 144, 1991.
9. Keisuke Nakano. Composing stack-attributed tree transducers. Technical Report METR–2004–01, Department of Mathematical Informatics, University of Tokyo, Japan, 2004.
10. Keisuke Nakano and Susumu Nishimura. Deriving event-based document transformers from tree-based specifications. In Mark van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001.
11. Leaf Petersen, Robert Harper, Karl Crary, and Frank Pfenning. A type theory for memory allocation and data layout. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2003.
12. Jeff Polakow. *Ordered linear logic and applications*. PhD thesis, Carnegie Mellon University, June 2001. Available as Technical Report CMU-CS-01-152.
13. David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Proceedings of Functional Programming Languages and Computer Architecture*, pages 1–11, San Diego, California, 1995.
14. P. Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP '88. European Symposium on Programming, Nancy, France, 1988 (Lecture Notes in Computer Science, vol. 300)*, pages 344–358. Berlin: Springer-Verlag, 1988.