Ordered Types for Stream Processing of Tree-Structured Data*

Ryosuke Sato† Kohei Suenaga† Naoki Kobayashi†

ABSTRACT

Suenaga et al. have developed a type-based framework for automatically translating tree-processing programs into stream-processing ones. The key ingredient of the framework was the use of ordered linear types to guarantee that a tree-processing program traverses an input tree just once in the depth-first, left-to-right order (so that the input tree can be read from a stream). Their translation, however, sometimes introduces redundant buffering of input data. This paper extends their framework by introducing ordered, *non-linear* types in addition to ordered linear types. The resulting transformation framework reduces the redundant buffering, generating more efficient stream-processing programs.

1. INTRODUCTION

Suenaga et al. [3, 11] have proposed a framework for automatically translating tree-processing programs into stream-processing ones. By using the framework, a user can write a tree-manipulating program in an ordinary functional language, and then the program is translated into a stream-processing program and executed. The framework allows efficient processing of tree-structured data (as they are usually stored in a text or stream format), while keeping the readability and maintainability of functional programs. Based on the framework, they have implemented an XML stream-processing program generator `X-P` [12].

The key ingredient of their framework was an *ordered linear type system*. The type system classifies tree data into those of *ordered linear types* (which model trees stored in streams) and those of non-linear types called *buffered trees* (which model trees stored in memory), and ensures that trees of ordered linear types are accessed only once, in the left-to-right, depth-first preorder, so that they can be read from a stream. By performing a kind of type inference [11], one can automatically transform an ordinary functional, tree-processing program into another tree-processing program that is well-typed in the ordered linear type system. The latter program can then be further transformed into a stream processing program in a straightforward manner.

Figure 1 shows an example of the two-step transformations. The source program deals with binary trees which stores an integer value at each leaf. The program takes a binary tree t as input, conducts pattern matching to the tree and returns $\mathbf{node}(t_2, f\ t_1)$ if the tree is a branch. The program accesses t_2 before t_1, so that the access order restriction mentioned above is violated. (We assume the call-by-value, left-to-right evaluation order.) Suenaga et al.'s framework automatically finds the violation and inserts *buffering primitives* to the program. In this case, t_1 is converted to a *buffered tree* by the buffering primitive **s2m**. Buffered trees can be freely accessed, so that the translated program conforms to the access order restriction. Then, the program is translated into the stream-processing program by replacing tree operations with stream operations.

Source program:
$\quad \mathbf{let\ rec}\ f\ t\ = \mathbf{case}\ t\ \mathbf{of}$
$\qquad \mathbf{leaf}\ n \Rightarrow \mathbf{leaf}\ n$
$\qquad \mid \mathbf{node}(t_1, t_2) \Rightarrow \mathbf{node}(t_2,\ f\ t_1)$

Intermediate program:
$\quad \mathbf{let\ rec}\ f\ t\ = \mathbf{case}\ t\ \mathbf{of}$
$\qquad \mathbf{leaf}\ n \Rightarrow \mathbf{leaf}\ n$
$\qquad \mid \mathbf{node}(t_1, t_2) \Rightarrow \mathbf{let}\ t_1 = \mathbf{s2m}(t_1)\ \mathbf{in}\ \mathbf{node}(t_2,\ f\ t_1)$

Target program:
$\quad \mathbf{let\ rec}\ f\ ()\ = \mathbf{case}\ \mathbf{read}()\ \mathbf{of}$
$\qquad \mathbf{leaf} \Rightarrow \mathbf{write\ leaf};\ \mathbf{write}\ (\mathbf{read}())$
$\qquad \mid \mathbf{node} \Rightarrow \mathbf{let}\ t_1 = \mathbf{s2m}()\ \mathbf{in}$
$\qquad\quad \mathbf{write\ node};\ \mathbf{copy}\ ();\ \mathbf{copymem}\ (f\ t_1)$

Figure 1: Suenaga et al.'s translation framework.

A shortcoming of the framework of Suenaga et al. [3] was that too many buffering commands were sometimes inserted in the first step of the transformation, resulting in less efficient stream-processing programs than hand-optimized code. That is mainly due to the severe restriction on the access order imposed by the ordered linear type system. For example, consider the following function, which takes an XML tree data representing a record of a person as an input, and returns the first and last names.

```
fun name(t) = (get_firstname(t), get_lastname(t))
```

Since the function `name` accesses `t` twice, a buffering command is inserted in the first step of the transformation, as follows.

```
fun name(t) =
  let t' = s2m(t) in
    (get_firstname(t'), get_lastname(t'))
```

The stream processing program generated from the intermediate program is not so efficient as it could be, because

*This paper is a revised version of the paper presented in Seventh Workshop on Programming Language Technologies for XML (PLAN-X 2009). Note for the editor and referees: PLAN-X is a workshop without formal proceedings. Informal proceedings is available on the web (http://db.ucsd.edu/planx2009/program.html), but the copyright is retained by the authors. The present submission is almost the same as the paper presented at PLAN-X, except addition of an experiment and minor adjustment. If this is a problem, please let us know.

†Tohoku University

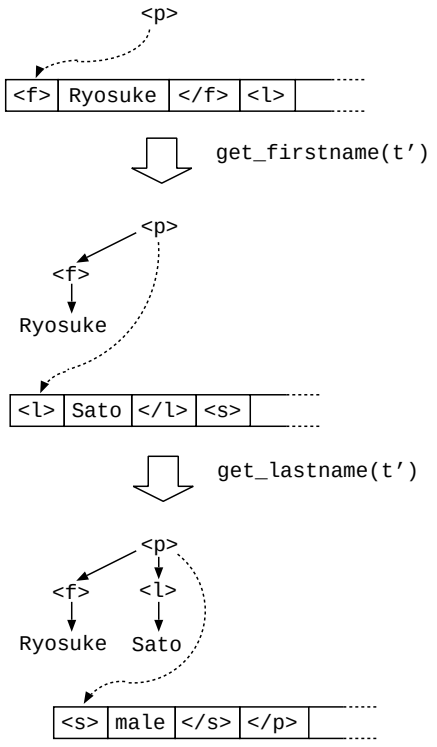

Figure 2: Hybrid tree during execution

(i) the whole tree t is copied to memory, despite that the only used data are the first and last names in t, and (ii) the memory space for t' can be reclaimed only by garbage collection.

We overcome the shortcoming mentioned above, by extending the ordered linear type system with *ordered, non-linear types* (which will be just called *ordered types* below). Trees of ordered types can be accessed more than once, but have to conform to a certain restriction on the access order. We use ordered types for describing *hybrid trees*, trees that are currently being read from a stream. A program stores a part of a hybrid tree on memory and the rest in a stream. By using ordered types and hybrid trees, `s2m` in the program above is repalced by `s2h`:

```
fun name(t) = let t' = s2h(t) in ...
```

The tree `t'` is now a hybrid one. Figure 2 illustrates how the state of `t'` changes. In that figure, `<f>` and `<l>` stand for `<firstname>` and `<lastname>`. The tree `t` is copied to memory only lazily, when needed by `get_firstname` and `get_lastname`. The hybrid tree `t'` is automatically deallocated after the execution of `get_lastname(t)`. Thus, unlike in the previous framework, the part `s` shown in Figure 2 is never copied to memory, and the memory space for the hybrid tree `t'` can be immediately reclaimed after being used.

In the rest of this paper, we first formalize the intermediate language and the new ordered linear type system (which has unlimited types, ordered types, and ordered linear types as mentioned above) and discuss its soundness in Section 2. Once the intermediate language and its type system are defined, then the translations into/from this language can be formalized by extending the authors' previous work [3, 11]

$d(modes)$	::=	$1 \mid \sharp \mid \omega \mid +$
M (terms)	::=	$n \mid x \mid \mathbf{fix}(f,x,M) \mid M_1\ M_2$
	$\mid$	$M_1 + M_2 \mid \mathbf{m2s}(x)$
	$\mid$	$\mathbf{let}\ x = \mathbf{s2m}(y)\ \mathbf{in}\ M$
	$\mid$	$\mathbf{let}\ x = \mathbf{s2h}(y)\ \mathbf{in}\ M$
	$\mid$	$\mathbf{leaf}^d\ M \mid \mathbf{node}^d(M_1,M_2)$
	$\mid$	$\mathbf{case}^d\ x\ \mathbf{of}\ \mathbf{leaf}\ y \Rightarrow M_1$
		$\mid \mathbf{node}(x_1,x_2) \Rightarrow M_2$
V (trees)	::=	$\mathbf{leaf}^d\ n \mid \mathbf{node}^d(V,V)$
v (values)	::=	$n \mid \mathbf{fix}(f,x,M) \mid V$
E (eval. ctx.)	::=	$[\,] \mid E\ M \mid v\ E \mid E + M$
	$\mid$	$v + E \mid \mathbf{m2s}(E) \mid \mathbf{leaf}^+\ E$
	$\mid$	$\mathbf{node}^+(E,M) \mid \mathbf{node}^+(v,E)$
τ (types)	::=	$\mathbf{int} \mid \tau_1 \to \tau_2 \mid \mathbf{tree}^d$

Figure 3: The syntax of $\mathcal{L}_I$ and types

with ordered types. We briefly sketch those translations in Section 3. Section 4 reports preliminary experiments. Section 5 discusses related work and Section 6 concludes.

2. INTERMEDIATE LANGUAGE $\mathcal{L}_I$ AND TYPE SYSTEM

This section introduces a functional tree-processing language $\mathcal{L}_I$, equipped with an ordered type system. The language makes distinction among four kinds of trees: (i) *ordered linear trees*, which can be accessed only once in the depth-first preorder, (ii) *hybrid trees*, which can be accessed more than once, but only until an ordered linear tree is accessed, (iii) *buffered trees*, which can be accessed without any order or linearity restrictions, and (iv) *output trees*, which are the result of a program and can never be read. The ordered linear type system guarantees that well-typed programs conform to such access restrictions on trees.

The language $\mathcal{L}_I$ serves as the intermediate language of the transformation framework sketched in Section 1. As discussed in Section 3, once the ordered type system for this language has been set up, the first step of the transformation can be achieved through a kind of type inference for the ordered type system, and the second step can be achieved by replacing (functional) tree operations with the corresponding stream operations in a rather straightforward manner.

2.1 Language

Figure 3 shows the syntax of our language. The language is a functional programming language extended with primitives for binary trees. The meta-variables n and x range over the sets of integers and variables, respectively. $\mathbf{fix}(f,x,M)$ is a recursive function that takes an argument x. f is bound to the function itself inside M.

$\mathbf{leaf}^d$ and $\mathbf{node}^d$ are constructors for binary trees. Here, d, called a *mode*, is either $1, \sharp, \omega$ or $+$, which describes *ordered-linear*, *hybrid*, *buffered* or *output* trees respectively. Each tree has the different restrictions on access order as mentioned before.

The term $\mathbf{let}\ x = \mathbf{s2m}(y)\ \mathbf{in}\ M$ copies the ordered linear tree y into a buffered one, binds x to it and evaluates M. $\mathbf{m2s}(M)$ converts a buffered tree into an output tree. $\mathbf{let}\ x = \mathbf{s2h}(y)\ \mathbf{in}\ M$ converts an ordered linear tree y into a hybrid tree, binds x to it and evaluates M. The $\mathbf{case}^d$ expression performs case analysis for each kind of trees.

$$(\mathbf{fix}(f,x,M)\ v, B, H, S) \longrightarrow ([f \mapsto \mathbf{fix}(f,x,M), x \mapsto v]M, B, H, S) \quad \text{(E-App)}$$

$$(n_1 + n_2, B, H, S) \longrightarrow (plus(n_1, n_2), B, H, S) \quad \text{(E-Plus)}$$

$$(\mathbf{let}\ x = \mathbf{s2m}(y)\ \mathbf{in}\ M, B, H, (y \mapsto V; S)) \longrightarrow ([x \mapsto z]M, B[z \mapsto V^{\omega}], \emptyset, S) \quad (z \text{ is fresh}) \quad \text{(E-StoM)}$$

$$(\mathbf{let}\ x = \mathbf{s2h}(y)\ \mathbf{in}\ M, B, H, (y \mapsto V; S)) \longrightarrow ([x \mapsto x']M, B, \{x' \mapsto V^{\sharp}\}, S) \quad (x' \text{ is fresh}) \quad \text{(E-StoH)}$$

$$(\mathbf{m2s}(x), B[x \mapsto V], H, S) \longrightarrow (V^{+}, B[x \mapsto V], H, S) \quad \text{(E-MtoS)}$$

$$(\mathbf{case}^1\ x\ \mathbf{of\ leaf}\ x_1 \Rightarrow M_1 \mid \mathbf{node}(x_1, x_2) \Rightarrow M_2, B, H, (x \mapsto \mathbf{leaf}^1\ n; S)) \longrightarrow ([x_1 \mapsto n]M_1, B, \emptyset, S) \quad \text{(E-Case1)}$$

$$(\mathbf{case}^1\ x\ \mathbf{of\ leaf}\ x_1 \Rightarrow M_1 \mid \mathbf{node}^{x_1}(x_2, \Rightarrow)M_2, B, H, (x \mapsto \mathbf{node}^1(V_1, V_2); S)) \longrightarrow (M_2, B, \emptyset, (x_1 \mapsto V_1; x_2 \mapsto V_2; S)) \quad \text{(E-Case2)}$$

$$(\mathbf{case}^{\omega}\ y\ \mathbf{of\ leaf}\ x \Rightarrow M \mid \mathbf{node}(x_1, x_2) \Rightarrow M', B[y \mapsto \mathbf{leaf}^{\omega}\ n], H, S) \longrightarrow ([x \mapsto n]M, B, H, S) \quad \text{(E-MCase1)}$$

$$(\mathbf{case}^{\omega}\ y\ \mathbf{of\ leaf}\ x \Rightarrow M \mid \mathbf{node}(x_1, x_2) \Rightarrow M', B[y \mapsto \mathbf{node}^{\omega}(V_1, V_2), H, S) \longrightarrow ([x_1 \mapsto x_1', x_2 \mapsto x_2']M', B[y \mapsto \mathbf{node}^{\omega}(V_1, V_2), x_1' \mapsto V_1, x_2' \mapsto V_2], H, S)) \quad (x_1' \text{ and } x_2' \text{ are fresh}) \quad \text{(E-MCase2)}$$

$$(\mathbf{case}^{\sharp}\ y\ \mathbf{of\ leaf}\ x \Rightarrow M \mid \mathbf{node}(x_1, x_2) \Rightarrow M', B, H[y \mapsto \mathbf{leaf}^{\sharp}\ n], S) \longrightarrow ([x_1 \mapsto n]M, B, H, S) \quad \text{(E-HCase1)}$$

$$(\mathbf{case}^{\sharp}\ y\ \mathbf{of\ leaf}\ x \Rightarrow M \mid \mathbf{node}(x_1, x_2) \Rightarrow M', B, H[y \mapsto \mathbf{node}^{\sharp}(V_1 \mapsto V_2,]), S) \longrightarrow ([x_1 \mapsto x_1', x_2 \mapsto x_2']M', B, H \cup \{x_1' \mapsto V_1, x_2' \mapsto V_2\}, S)(x_1' \text{ and } x_2' \text{ are fresh}) \quad \text{(E-HCase2)}$$

$$\frac{(M, B, H, S) \longrightarrow (M', B', H', S')}{(E[M], B, H, S) \longrightarrow (E[M'], B', H', S')} \quad \text{(E-Context)}$$

Figure 4: Operational semantics of $\mathcal{L}_I$

Example 1. *The following program takes a tree as an input, and returns a list of integers obtained by replacing each tree with the sum of its leftmost and second leftmost elements. Here,* `leftmost` *and* `leftmostsecond` *are functions which take a hybrid tree and return its leftmost and second leftmost elements.*

```
fix(f, t,
  case t of
      leaf n  -> leaf 0
    | node(t1,t2) ->
       let t1' = s2h(t1) in
       let n = leftmost t1' +
               leftmostsecond t1'
       in
         node(leaf n, f t2)
```

Figure 4 shows the operational semantics of the language. The semantics is expressed as a rewriting relation of configurations of the form (M, B, H, S). Here, B is a map from variables to buffered trees. H is a map from variables to hybrid trees. S is a *sequence* of bindings from variables to ordered linear trees (therefore the order of bindings matters). In Figure 4, V^d represents the tree obtained by replacing every mode annotation in V with d. For example, $(\mathbf{leaf}^1\ 1)^{\omega}$ represents the tree $\mathbf{leaf}^{\omega}\ 1$.

Note that we use the three tree environments in order to express the difference on access restrictions among the different kinds of trees. In the rules E-StoM and E-Case1, hybrid trees in H are discarded because a variable in S is accessed. In the rules E-StoH, E-StoM and E-Case1, in which a variable in S is accessed, the variable has to be at the head of S. Those restrictions reflect the intuition of the intermediate language explained in Section 1.

2.2 Ordered type system

We next introduce an ordered type system for the language introduced in the previous section. The type system guarantees that well-typed programs access trees in a valid order.

Figure 3 gives the syntax of types. The type **int** describes integers and $\tau_1 \to \tau_2$ describes functions from τ_1 to τ_2. We have four kinds of tree types. $\mathbf{tree}^{\omega}$ is the type of buffered trees. $\mathbf{tree}^{\sharp}$ is the type of hybrid trees. $\mathbf{tree}^1$ and $\mathbf{tree}^+$ are the types of input trees and output trees respectively.

Notice the constraints imposed on trees of each type. Trees of type $\mathbf{tree}^1$ must be accessed in the left-to-right, depth-first manner by traversing each node exactly once. Trees of type $\mathbf{tree}^{\omega}$ can be accessed in arbitrary manner. Though trees of type $\mathbf{tree}^{\sharp}$ can be accessed any number of times, they cannot be accessed after another tree of type $\mathbf{tree}^1$ is accessed.

A type judgment is of the form $\Gamma \mid \Psi \mid \Delta \vdash M{:}\tau$. Here, Γ is a *non-ordered type environment*, Ψ is an *ordered type environment* and Δ is an *ordered linear 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 $\mathbf{tree}^d \in \{\tau_1, \ldots, \tau_n\}$ implies $d = \omega$. An ordered type environment is a set of the form $\{x_1 : \mathbf{tree}^{\sharp}, \ldots, x_n : \mathbf{tree}^{\sharp}\}$, where $x_1, \ldots, x_n$ are different from each other. An ordered linear type environment is a *sequence* of the form $x_1 : \mathbf{tree}^1, \ldots, x_n : \mathbf{tree}^1$, where $x_1, \ldots, x_n$ are different from each other. We assume that Γ,Ψ, and Δ do not share identical variables.

In that judgment, the type environments express how trees are accessed during the evaluation of M. The ordered linear type environment $x_1 : \mathbf{tree}^1, \ldots, x_n : \mathbf{tree}^1$ specifies not only $x_1, \ldots, x_n$ 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. The ordered type environment $x_1{:}\mathbf{tree}^{\sharp}, \ldots, x_n{:}\mathbf{tree}^{\sharp}$ specifies that $x_1, \ldots, x_n$ can be accessed several times and there is no restriction on

access order among $x_1, \ldots, x_n$. However, if a variable in Δ is accessed, none of $x_1, \ldots, x_n$ can be accessed anymore. For example, if $\Psi = x_1 : \mathbf{tree}^\sharp, x_2 : \mathbf{tree}^\sharp$ and $\Delta = y : \mathbf{tree}^1$, then both accessing x_1, x_1 and y and accessing x_2, x_2, x_1 and y in these orders are legitimate, while x_1, y and x_2 is illegal.

DEFINITION 1 (CONCATENATION). *An operation* $(\Psi_1 \mid \Delta_1); (\Psi_2 \mid \Delta_2)$ *is defined as follows.*

$$(\Psi_1 \mid \Delta_1); (\Psi_2 \mid \Delta_2) \quad = \quad \begin{cases} (\Psi_1 \cup \Psi_2 \mid \Delta_2) & (\textit{if } \Delta_1 = \emptyset) \\ (\Psi_1 \mid (\Delta_1, \Delta_2)) & (\textit{if } \Psi_2 = \emptyset) \end{cases}$$

Intuitively, $(\Psi \mid \Delta) = (\Psi_1 \mid \Delta_1); (\Psi_2 \mid \Delta_2)$ are environments that allow trees to be accessed according to $\Psi_1 \mid \Delta_1$ and then to $\Psi_2 \mid \Delta_2$ sequentially. $(\Psi_1 \mid \Delta_1); (\Psi_2 \mid \Delta_2)$ is defined only when $\Delta_1 = \emptyset$ or $\Psi_2 = \emptyset$ because variables in Ψ_2 cannot be accessed after an ordered linear tree is accessed.

Figure 5 shows the typing rules. We explain important rules below.

- In the rules T-STOM, T-STOH and T-CASE, the ordered type environment in the conclusion has to be empty because an ordered linear tree is being accessed, so that a program is not allowed to access hybrid trees. Note also that the ordered linear tree variable that is being used has to be at the head of the ordered linear type environment to ensure the order condition.
- In the rule T-STOH for **let** $x = \mathbf{s2h}(y)$ **in** M, x is in the ordered type environment in the premise because y is converted to a hybrid tree, named x and used in M.
- T-HCASE is for $\mathbf{case}^\sharp$ expressions. Because a hybrid tree can be freely accessed until another variable in the ordered linear type environment is accessed, the variable x in the ordered type environment in the conclusion part also can be used as a hybrid tree in M_1 and M_2. In M_2, the children of x (x_2 and x_3) can also be used as hybrid trees.
- In the rules T-FIX1 and T-FIX2, both the ordered type environment and the ordered type environment have to be empty to avoid hybrid trees and ordered linear trees being captured in the closure.
- In the rules T-APP, T-PLUS, T-NODE, and T-MNODE, the ordered linear and the ordered type environments of M_1 and M_2 are concatenated in this order in the conclusion. On the other hand, M_1 and M_2 share the same non-ordered type environment since there is no restriction on usage of the variables in a non-ordered type environment.
- T-CASE is the rule for destructors for ordered linear trees. If x matches $\mathbf{node}^1(x_2, x_3)$, subtrees x_2 and x_3 have to be accessed in this order to enforce the left-to-right depth-first order restriction. This is expressed by $x_1 : \mathbf{tree}^1, x_2 : \mathbf{tree}^1, \Delta$, the ordered linear type environment of M_2.

Figure 6 shows a part of a typing example of the program presented in Section 2. Thanks to the primitive **s2h**, tree on stream t_1 is shared in **leftmost** t_1' and **leftmostsecond** t_1' as hybrid tree t'.

$$\begin{array}{rcl} M \text{ (terms)} & ::= & n \mid x \mid \mathbf{fix}(f, x, M) \mid M_1\ M_2 \\ & \mid & M_1 + M_2 \mid \mathbf{leaf}\ M \mid \mathbf{node}(M_1, M_2) \\ & \mid & \mathbf{case}\ x\ \mathbf{of\ leaf}\ x_1 \Rightarrow M_1 \\ & & \quad \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2 \\ \tau \text{ (types)} & ::= & \mathbf{int} \mid \tau_1 \to \tau_2 \mid \mathbf{tree} \end{array}$$

Figure 7: The syntax of $\mathcal{L}_S$ and types

2.3 Type soundness

We state soundness of the type system in this section. The soundness theorem guarantees that, well-typed programs access trees in a valid order. As an illegal access order leads to a stuck state in our operational semantics, it is sufficient to state that well-typed programs never get stuck.

THEOREM 1 (TYPE SOUNDNESS). *If* $\emptyset \mid \emptyset \mid x : \mathbf{tree}^1 \vdash M : \mathbf{tree}^+$ *and* $(M, \emptyset, \emptyset, x \mapsto V) \longrightarrow^* (M', B', H', S')$ *then* M' *is a tree value and* $S' = \emptyset$*, or there exist* M'', B'', H'' *and* S'' *such that* $(M', B', H', S') \longrightarrow (M'', B'', H'', S'')$.

3. TRANSLATION

This section introduces the source language $\mathcal{L}_S$ and the target language $\mathcal{L}_T$, and describes how a source program is translated into a well-typed intermediate program, and then translated into a target program. Because a source program given to the translation algorithm may not respect the order restriction on an input tree, the algorithm first inserts buffering primitives **s2m**, **s2h** and **m2s** and make the program a well-typed intermediate program. This step is conducted by performing a kind of type inference for the type system introduced in Section 2. Then, the algorithm replaces each tree-manipulating primitives with stream-manipulating primitives.

3.1 Translation from $\mathcal{L}_S$ to $\mathcal{L}_I$.

Figure 7 gives the syntax of the source language $\mathcal{L}_S$. The language differs from $\mathcal{L}_I$ in Section 2 in that $\mathcal{L}_S$ has neither buffering primitives nor the distinction among $\mathbf{leaf}^1/\mathbf{node}^1$, $\mathbf{leaf}^\omega/\mathbf{node}^\omega$, $\mathbf{leaf}^\sharp/\mathbf{node}^\sharp$ and $\mathbf{leaf}^+/\mathbf{node}^+$. A user can write a source program without considering the order and linearity restrictions. Such a source program is translated into a well-typed intermediate program by inserting buffering and hybridization primitives.

We describe an algorithm for translating a source program into a well-typed intermediate program by inserting buffering primitives to the program. Following Suenaga et al. [11], we introduce a type-based, non-deterministic translation rules. Then the translation algorithm is obtained as a kind of type inference algorithm in a manner similar to [11].

The non-deterministic translation is given by a judgment $\Gamma \mid \Psi \mid \Delta \vdash M \rightsquigarrow M' : \tau$. The judgment means:

1. M and M' are equivalent except for the representation of trees, and
2. $\Gamma \mid \Psi \mid \Delta \vdash M' : \tau$ holds.

Figure 9 shows a part of rules for the judgment $\Gamma \mid \Psi \mid \Delta \vdash M' : \tau$. The rules are non-deterministic in the sense that there may be more than one valid transformations for each source program M. For example, there are three rules for the

$$\Gamma \mid \Psi \mid \emptyset \vdash n : \mathbf{int} \quad \text{(T-Int)} \qquad \Gamma \mid \Psi \mid x : \mathbf{tree}^1 \vdash x : \mathbf{tree}^1 \quad \text{(T-Var1)}$$

$$\Gamma \mid x : \mathbf{tree}^\sharp, \Psi \mid \emptyset \vdash x : \mathbf{tree}^\sharp \quad \text{(T-Var2)} \qquad \Gamma, x : \tau \mid \Psi \mid \emptyset \vdash x : \tau \quad \text{(T-Var3)}$$

$$\frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \mid \emptyset \mid \emptyset \vdash M : \tau_2}{\Gamma \mid \emptyset \mid \emptyset \vdash \mathbf{fix}(f, x, M) : \tau_1 \to \tau_2} \text{(T-Fix1)} \qquad \frac{\Gamma, f : \mathbf{tree}^\sharp \to \tau_2 \mid x : \mathbf{tree}^\sharp \mid \emptyset \vdash M : \tau_2}{\Gamma \mid \emptyset \mid \emptyset \vdash \mathbf{fix}(f, x, M) : \mathbf{tree}^\sharp \to \tau_2} \text{(T-Fix2)}$$

$$\frac{\Gamma, f : \mathbf{tree}^1 \to \tau_2 \mid \emptyset \mid x : \mathbf{tree}^1 \vdash M : \tau_2}{\Gamma \mid \emptyset \mid \emptyset \vdash \mathbf{fix}(f, x, M) : \mathbf{tree}^1 \to \tau_2} \text{(T-Fix3)} \qquad \frac{\Gamma \mid \Psi_1 \mid \Delta_1 \vdash M_1 : \tau_1 \to \tau_2 \qquad \Gamma \mid \Psi_2 \mid \Delta_2 \vdash M_2 : \tau_1}{\Gamma \mid (\Psi_1 \mid \Delta_1); (\Psi_2 \mid \Delta_2) \vdash M_1 M_2 : \tau_2} \text{(T-App)}$$

$$\frac{\Gamma \mid \Psi_1 \mid \Delta_1 \vdash M_1 : \mathbf{int} \qquad \Gamma \mid \Psi_2 \mid \Delta_2 \vdash M_2 : \mathbf{int}}{\Gamma \mid (\Psi_1 \mid \Delta_1); (\Psi_2 \mid \Delta_2) \vdash M_1 + M_2 : \mathbf{int}} \text{(T-Plus)} \qquad \frac{\Gamma, x : \mathbf{tree}^\omega \mid \emptyset \mid \Delta \vdash M : \tau}{\Gamma \mid \emptyset \mid y : \mathbf{tree}^1, \Delta \vdash \mathbf{let}\ x = \mathbf{s2m}(y)\ \mathbf{in}\ M : \tau} \text{(T-StoM)}$$

$$\frac{\Gamma \mid x : \mathbf{tree}^\sharp \mid \Delta \vdash M : \tau}{\Gamma \mid \emptyset \mid y : \mathbf{tree}^1, \Delta \vdash \mathbf{let}\ x = \mathbf{s2h}(y)\ \mathbf{in}\ M : \tau} \text{(T-StoH)} \qquad \frac{\Gamma \mid \Psi \mid \Delta \vdash M : \mathbf{tree}^\omega}{\Gamma \mid \Psi \mid \Delta \vdash \mathbf{m2s}(M) : \mathbf{tree}^+} \text{(T-MtoS)}$$

$$\frac{\Gamma \mid \Psi \mid \Delta \vdash M : \mathbf{int} \qquad d \in \{\omega, +\}}{\Gamma \mid \Psi \mid \Delta \vdash \mathbf{leaf}^d\ M : \mathbf{tree}^d} \text{(T-Leaf)}$$

$$\frac{\Gamma \mid \Psi_1 \mid \Delta_1 \vdash M_1 : \mathbf{tree}^d \qquad \Gamma \mid \Psi_2 \mid \Delta_2 \vdash M_2 : \mathbf{tree}^d \qquad d \in \{\omega, +\}}{\Gamma \mid (\Psi_1 \mid \Delta_1); (\Psi_2 \mid \Delta_2) \vdash \mathbf{node}^d(M_1, M_2) : \mathbf{tree}^d} \text{(T-Node)}$$

$$\frac{\Gamma, x_1 : \mathbf{int} \mid \emptyset \mid \Delta \vdash M_1 : \tau \qquad \Gamma \mid \emptyset \mid x_2 : \mathbf{tree}^1, x_3 : \mathbf{tree}^1, \Delta \vdash M_2 : \tau}{\Gamma \mid \emptyset \mid x : \mathbf{tree}^1, \Delta \vdash \mathbf{case}^1\ x\ \mathbf{of\ leaf}\ x_1 \Rightarrow M_1 \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2 : \tau} \text{(T-Case)}$$

$$\frac{\Gamma, x : \mathbf{tree}^\omega, x_1 : \mathbf{int} \mid \Psi \mid \Delta \vdash M_1 : \tau \qquad \Gamma, x : \mathbf{tree}^\omega, x_2 : \mathbf{tree}^\omega, x_3 : \mathbf{tree}^\omega \mid \Psi \mid \Delta \vdash M_2 : \tau}{\Gamma, x : \mathbf{tree}^\omega \mid \Psi \mid \Delta \vdash \mathbf{case}^\omega\ x\ \mathbf{of\ leaf}\ x_1 \Rightarrow M_1 \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2 : \tau} \text{(T-MCase)}$$

$$\frac{\Gamma, x_1 : \mathbf{int} \mid x : \mathbf{tree}^\sharp, \Psi \mid \Delta \vdash M_1 : \tau \qquad \Gamma \mid x : \mathbf{tree}^\sharp, x_2 : \mathbf{tree}^\sharp, x_3 : \mathbf{tree}^\sharp, \Psi \mid \Delta \vdash M_2 : \tau}{\Gamma \mid x : \mathbf{tree}^\sharp, \Psi \mid \Delta \vdash \ \mathbf{case}^\sharp\ x\ \mathbf{of\ leaf}\ x_1 \Rightarrow M_1 \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2\ : \tau} \text{(T-HCase)}$$

Figure 5: The typing rules

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma' \mid t_1' : \mathbf{tree}^\sharp \mid \emptyset \vdash \mathbf{leftmost}\ t_1' : \mathbf{int}} \quad \dfrac{\vdots}{\Gamma' \mid t_1' : \mathbf{tree}^\sharp \mid \emptyset \vdash \mathbf{leftmostsecond}\ t_1' : \mathbf{int}}}{\Gamma' \mid t_1' : \mathbf{tree}^\sharp \mid \emptyset \vdash \mathbf{leftmost}\ t_1' + \mathbf{leftmostsecond}\ t_1' : \mathbf{int}} \quad \dfrac{\dfrac{\vdots}{\Gamma' \mid t_1 : \mathbf{tree}^\sharp \mid t_2 : \mathbf{tree}^1 \vdash M : \mathbf{tree}^+}}{\Gamma' \mid \emptyset \mid t_1 : \mathbf{tree}^\sharp, t_2 : \mathbf{tree}^1 \vdash \mathbf{let}\ t_1' = \mathbf{s2h}(t_1)\ \mathbf{in}\ M : \mathbf{tree}^+}}{\Gamma' \mid \emptyset \mid t : \mathbf{tree}^1 \mid \mathbf{case}\ t\ \mathbf{of\ leaf}\ n \Rightarrow \mathbf{leaf}\ n \mid \mathbf{node}(t_1, t_2) \Rightarrow \mathbf{let}\ t_1' = \mathbf{s2h}(t_1)\ \mathbf{in}\ M : \mathbf{tree}^+}}{\Gamma \mid \emptyset \mid \emptyset \vdash \mathbf{fix}(f, t, \mathbf{case}\ t\ \mathbf{of\ leaf}\ n \Rightarrow \mathbf{leaf}\ n \mid \mathbf{node}(t_1, t_2) \Rightarrow \mathbf{let}\ t_1' = \mathbf{s2h}(t_1)\ \mathbf{in}\ M) : \mathbf{tree}^1 \to \mathbf{tree}^+}$$

Figure 6: A part of a typing example. $\Gamma = \{\mathbf{leftmost} : \mathbf{tree}^\sharp \to \mathbf{int}, \mathbf{leftmostsecond} : \mathbf{tree}^\sharp \to \mathbf{int}\}$**,** $\Gamma' = \Gamma, f : \mathbf{tree}^1 \to \mathbf{tree}^+$**.**

term **case** x **of** ... depending on whether the matched tree is translated into an ordered linear, a hybrid or a buffered one. The highlight of the rules is Tr-StoM and Tr-StoH which insert **s2m** and **s2h** to source programs. For example, the rule Tr-StoH says that, if a variable x is bound to an ordered linear tree before the evaluation of M, and if x can be used as a hybrid tree in M', the result of translation of M, then one can convert x to a hybrid tree here by the **s2h** primitive. The rule Tr-StoM is similar.

The transformation rules presented above are non-deterministic in the sense that there may be more than one possible M' and τ that satisfy $\Gamma \mid \Psi \mid \Delta \vdash M \rightsquigarrow M' : \tau$. A deterministic algorithm is obtained as a kind of type inference algorithm as in Suenaga et al.'s work [11]. By merging three (unordered, ordered, and ordered linear) type environments into one, we can construct syntax-directed program transformation rules. The transformation algorithm is then obtained as a constraint-based algorithm, which first extracts constraints on modes based on the transformation rules and solves them. We omit a detailed description of the algorithm in this paper.

3.2 Translation from $\mathcal{L}_I$ to $\mathcal{L}_T$

Figures 8 shows the syntax of the target language $\mathcal{L}_T$, which is a stream-processing impure functional language. **read** is a primitive for reading a token (**leaf**, **node**, or an integer) from the input stream. **write** is a primitive for writing a

$$\frac{\Gamma \mid \Psi_1 \mid \Delta_1 \vdash M_1 \rightsquigarrow M_1' : \mathbf{tree}^{+} \qquad \Gamma \mid \Psi_2 \mid \Delta_2 \vdash M_2 \rightsquigarrow M_2' : \mathbf{tree}^{+}}{\Gamma \mid (\Psi_1 \mid \Delta_1); (\Psi_2 \mid \Delta_2) \vdash \mathbf{node}(M_1, M_2) \rightsquigarrow \mathbf{node}^{+}(M_1', M_2') : \mathbf{tree}^{+}} \quad (\text{Tr-Node1})$$

$$\frac{\Gamma \mid \Psi_1 \mid \Delta_2 \vdash M_1 \rightsquigarrow M_1' : \mathbf{tree}^{\omega} \qquad \Gamma \mid \Psi_2 \mid \Delta_2 \vdash M_2 \rightsquigarrow M_2' : \mathbf{tree}^{\omega}}{\Gamma \mid (\Psi_1 \mid \Delta_1); (\Psi_2 \mid \Delta_2) \vdash \mathbf{node}(M_1, M_2) \rightsquigarrow \mathbf{node}^{\omega}(M_1', M_2') : \mathbf{tree}^{\omega}} \quad (\text{Tr-Node2})$$

$$\frac{\Gamma, x_1 : \mathbf{int} \mid \emptyset \mid \Delta \vdash M_1 \rightsquigarrow M_1' : \tau \qquad \Gamma \mid \emptyset \mid x_2 : \mathbf{tree}^{1}, x_3 : \mathbf{tree}^{1}, \Delta \vdash M_2 \rightsquigarrow M_2' : \tau}{\begin{array}{r}\Gamma \mid \Psi \mid x : \mathbf{tree}^{1}, \Delta \vdash \mathbf{case}\ x\ \mathbf{of\ leaf}\ x_1 \Rightarrow M_1 \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2 \\ \rightsquigarrow \mathbf{case}^{1}\ x\ \mathbf{of\ leaf}\ x_1 \Rightarrow M_1' \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2' : \tau\end{array}} \quad (\text{Tr-Case1})$$

$$\frac{\Gamma, x : \mathbf{tree}^{\omega}, x_1 : \mathbf{int} \mid \Psi \mid \Delta \vdash M_1 \rightsquigarrow M_1' : \tau \qquad \Gamma, x : \mathbf{tree}^{\omega}, x_2 : \mathbf{tree}^{\omega}, x_3 : \mathbf{tree}^{\omega} \mid \Psi \mid \Delta \vdash M_2 \rightsquigarrow M_2' : \tau}{\begin{array}{r}\Gamma, x : \mathbf{tree}^{\omega} \mid \Psi \mid \Delta \vdash \mathbf{case}\ x\ \mathbf{of\ leaf}\ x_1 \Rightarrow M_1 \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2 \\ \rightsquigarrow \mathbf{case}^{\omega}\ x\ \mathbf{of\ leaf}\ x_1 \Rightarrow M_1' \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2' : \tau\end{array}} \quad (\text{Tr-Case2})$$

$$\frac{\Gamma, x_1 : \mathbf{int} \mid x : \mathbf{tree}^{\sharp}, \Psi \mid \Delta \vdash M_1 \rightsquigarrow M_1' : \tau \qquad \Gamma, \mid x : \mathbf{tree}^{\sharp}, x_2 : \mathbf{tree}^{\sharp}, x_3 : \mathbf{tree}^{\sharp}, \Psi \mid \Delta \vdash M_2 \rightsquigarrow M_2'' : \tau}{\begin{array}{r}\Gamma \mid x : \mathbf{tree}^{\sharp}, \Psi \mid \Delta \vdash \mathbf{case}\ x\ \mathbf{of\ leaf}\ x_1 \Rightarrow M_1 \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2 \\ \rightsquigarrow \mathbf{case}^{\sharp}\ x\ \mathbf{of\ leaf}\ x_1 \Rightarrow M_1' \mid \mathbf{node}(x_2, x_3) : \tau\end{array}} \quad (\text{Tr-Case3})$$

$$\frac{\Gamma, x : \mathbf{tree}^{\omega} \mid \emptyset \mid \Delta \vdash M \rightsquigarrow M' : \tau}{\Gamma \mid \Psi \mid x : \mathbf{tree}^{1}, \Delta \vdash M \rightsquigarrow \mathbf{let}\ x = \mathbf{s2m}(x)\ \mathbf{in}\ M' : \tau} \quad (\text{Tr-StoM})$$

$$\frac{\Gamma \mid x : \mathbf{tree}^{\sharp} \mid \Delta \vdash M \rightsquigarrow M' : \tau}{\Gamma \mid \Psi \mid x : \mathbf{tree}^{1}, \Delta \vdash M \rightsquigarrow \mathbf{let}\ x = \mathbf{s2h}(x)\ \mathbf{in}\ M' : \tau} \quad (\text{Tr-StoH})$$

Figure 9: A part of the rules for the judgment $\Gamma \mid \Psi \mid \Delta \vdash M \rightsquigarrow M' : \tau$

e (terms)	::=	$n \mid x \mid l \mid \mathbf{leaf} \mid \mathbf{node}$
		$\mid\ () \mid \mathbf{fix}(f, x, e) \mid e_1\ e_2$
		$\mid\ e_1 + e_2 \mid \mathbf{read}\ () \mid \mathbf{write}\ e$
		$\mid\ \mathbf{leaf}^{\omega}\ e \mid \mathbf{node}^{\omega}(e_1, e_2)$
		$\mid\ \mathbf{let}\ x = \mathbf{s2m}()\ \mathbf{in}\ M$
		$\mid\ \mathbf{let}\ x = \mathbf{s2h}()\ \mathbf{in}\ M$
		$\mid\ \mathbf{flush}()$
		$\mid\ \mathbf{case}\ e\ \mathbf{of\ leaf} \Rightarrow e_1$
		$\quad \mid \mathbf{node} \Rightarrow e_2$
		$\mid\ \mathbf{case}^{\omega}\ e\ \mathbf{of\ leaf}\ x_1 \Rightarrow e_1$
		$\quad \mid \mathbf{node}(x_2, x_3) \Rightarrow e_2$
		$\mid\ \mathbf{case}^{\sharp}\ e\ \mathbf{of\ leaf}\ x_1 \Rightarrow e_1$
		$\quad \mid \mathbf{node}(x_2, x_3) \Rightarrow e_2$
V^{ω} (trees on mem.)	::=	$\mathbf{leaf}^{\omega}\ n \mid \mathbf{node}^{\omega}(V_1^{\omega}, V_2^{\omega})$
$V^{\sharp}$ (hybrid trees)	::=	$l \mid \mathbf{leaf}^{\omega}\ n \mid \mathbf{node}^{\omega}(V_1^{\sharp}, V_2^{\sharp})$
v (values)	::=	$n \mid \mathbf{fix}(f, x, e) \mid V^{\omega} \mid V^{\sharp}$
E (eval. ctx.)	::=	$[\,] \mid E\ M \mid \mathbf{fix}(f, x, e)\ E$
		$\mid\ E + e \mid n + E \mid \mathbf{read}\ E$
		$\mid\ \mathbf{write}\ E \mid \mathbf{leaf}^{\omega}\ E$
		$\mid\ \mathbf{node}^{\omega}(E, e) \mid \mathbf{node}^{\omega}(V^{\omega}, E)$
		$\mid\ \mathbf{h2m}(E)$
		$\mid\ \mathbf{case}\ E\ \mathbf{of\ leaf} \Rightarrow e_1$
		$\quad \mid \mathbf{node} \Rightarrow e_2$
		$\mid\ \mathbf{case}^{\omega}\ E\ \mathbf{of\ leaf}\ x_1 \Rightarrow e_1$
		$\quad \mid \mathbf{node}(x_2, x_3) \Rightarrow e_2$
		$\mid\ \mathbf{case}^{\sharp}\ E\ \mathbf{of\ leaf}\ x_1 \Rightarrow e_1$
		$\quad \mid \mathbf{node}(x_2, x_3) \Rightarrow e_2$

Figure 8: The syntax of $\mathcal{L}_T$

token to the output stream. $\mathbf{leaf}^{\omega}\ e$ and $\mathbf{node}^{\omega}(e_1, e_2)$ are trees constructed on memory. The term $\mathbf{case}\ e\ \mathbf{of\ leaf} \Rightarrow e_1 \mid \mathbf{node} \Rightarrow e_2$ performs a case analysis on the value of e. To express lazily read hybrid trees, we use *locations* which are ranged over by a meta variable l. A location is a dummy pointer for a tree that has not been accessed and thus has not been constructed yet. Such a tree is constructed when the location is accessed. A hybrid tree is expressed as a location, a leaf on memory or a branch whose children are hybrid trees. **flush** is a primitive for discarding hybrid trees that is currently kept. $\mathbf{case}^{\omega}\ e\ \mathbf{of}\ \ldots$ and $\mathbf{case}^{\sharp}\ e\ \mathbf{of}\ \ldots$ are pattern matching for buffered and hybrid trees.

A well-typed $\mathcal{L}_I$ program can be translated into an equivalent stream-processing program using the algorithm $\mathcal{A}$ defined in Figure 10. The algorithm $\mathcal{A}$ converts output tree constructions into stream output operations and case analysis for ordered linear trees into stream input operations. Note that an instruction **flush** is inserted before $\mathbf{s2m}, \mathbf{s2h}$ and $\mathbf{case}^{1}\ x\ \mathbf{of}$. This instruction ensures that hybrid trees are actually discarded before another ordered linear tree is accessed.

4. PRELIMINARY EXPERIMENTS

To evaluate the effectiveness of the new transformation framework, we have implemented a prototype translator from the intermediate language $\mathcal{L}_I$ to the stream-processing language $\mathcal{L}_T$. The current translator supports only binary trees having integers or strings as leaves. An extension for dealing with XML documents, as well as implementation of a translator from the source to the intermediate language are currently under development.

As a benchmark program for preliminary experiments, we used the following programs:

- (ex_leftmost) a program in Example 1, which takes a list of binary trees and returns a list of integers obtained by replacing each tree with thesum of its leftmost and second leftmost elements, and
- (ex_bib) a program which takes a bibliography database and returns a list of title and authors where the title contains a specific word.

$$
\begin{aligned}
&\mathcal{A}(n) = n\\
&\mathcal{A}(x) = x\\
&\mathcal{A}(\mathbf{fix}(f, x, M)) = \mathbf{fix}(f, x, \mathcal{A}(M))\\
&\mathcal{A}(M_1\ M_2) = \mathcal{A}(M_1)\ \mathcal{A}(M_2)\\
&\mathcal{A}(M_1 + M_2) = \mathcal{A}(M_1) + \mathcal{A}(M_2)\\
&\mathcal{A}(\mathbf{let}\ x = \mathbf{s2m}(y)\ \mathbf{in}\ M) = \mathbf{flush}(); \mathbf{let}\ x = \mathbf{s2m}()\ \mathbf{in}\ \mathcal{A}(M)\\
&\mathcal{A}(\mathbf{let}\ x = \mathbf{s2h}(y)\ \mathbf{in}\ M) = \mathbf{flush}(); \mathbf{let}\ x = \mathbf{s2h}()\ \mathbf{in}\ \mathcal{A}(M)\\
&\mathcal{A}\big(\mathbf{leaf}^{+}\ M\big) = \mathbf{write\ leaf}; \mathbf{write}\ \mathcal{A}(M)\\
&\mathcal{A}\big(\mathbf{node}^{+}(M_1, M_2)\big) = \mathbf{write\ node}; \mathcal{A}(M_1); \mathcal{A}(M_2)\\
&\mathcal{A}(\mathbf{leaf}^{\omega}\ M) = \mathbf{leaf}^{\omega}\ \mathcal{A}(M)\\
&\mathcal{A}(\mathbf{node}^{\omega}(M_1, M_2)) = \mathbf{node}^{\omega}(\mathcal{A}(M_1), \mathcal{A}(M_2))\\
&\mathcal{A}\big(\mathbf{case}^{1}\ x\ \mathbf{of\ leaf}\ x_1 \Rightarrow M_1 \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2\big) =\\
&\qquad \mathbf{case\ flush}(); \mathbf{read}()\ \mathbf{of\ leaf} \Rightarrow \mathbf{let}\ x_1 = \mathbf{read}()\ \mathbf{in}\ \mathcal{A}(M_1) \mid \mathbf{node} \Rightarrow [()/x_2, ()/x_3]\mathcal{A}(M_2)\\
&\mathcal{A}(\mathbf{case}^{\omega}\ x\ \mathbf{of\ leaf}\ x_1 \Rightarrow M_1 \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2) = \mathbf{case}^{\omega}\ x\ \mathbf{of\ leaf}\ x_1 \Rightarrow \mathcal{A}(M_1) \mid \mathbf{node}(x_2, x_3) \Rightarrow \mathcal{A}(M_2)\\
&\mathcal{A}\big(\mathbf{case}^{\sharp}\ x\ \mathbf{of\ leaf}\ x_1 \Rightarrow M_1 \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2\big) = \mathbf{case}^{\sharp}\ x\ \mathbf{of\ leaf}\ x_1 \Rightarrow \mathcal{A}(M_1) \mid \mathbf{node}(x_2, x_3) \Rightarrow \mathcal{A}(M_2)
\end{aligned}
$$

Figure 10: Translation algorithm

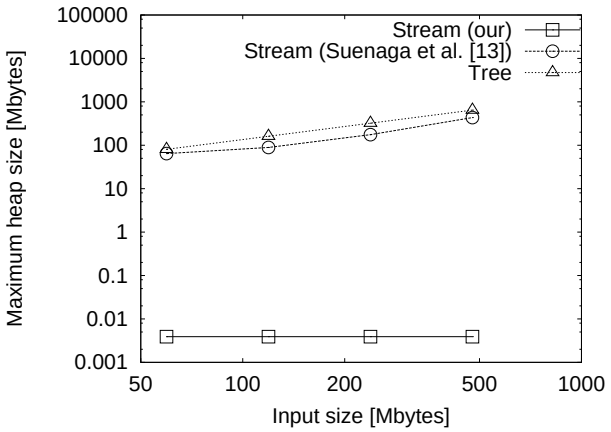

Figure 11: Memory consumption (ex_leftmost)

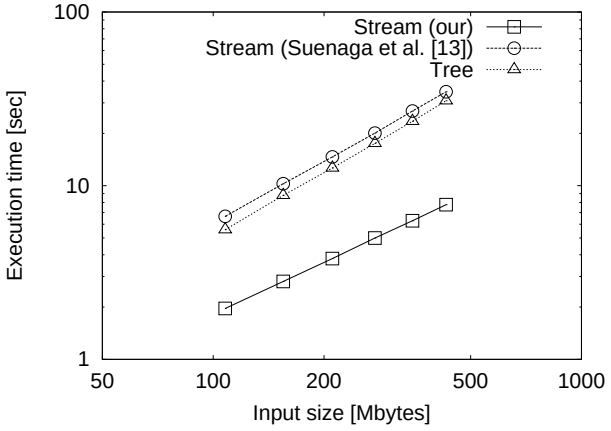

Figure 12: Memory consumption (ex_bib)

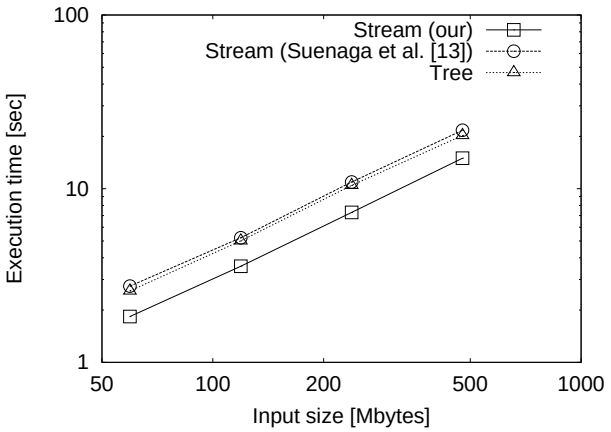

Figure 13: Execution time (ex_leftmost)

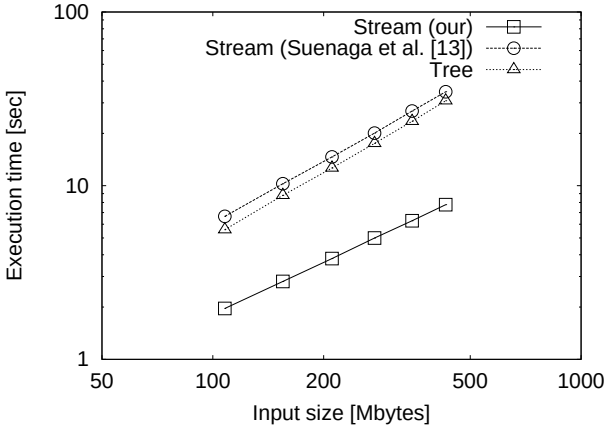

Figure 14: Execution time (ex_bib)

Figures 11–14 show the result of the experiment. Figure 11 and 12 compare the maximum memory consumption of the stream-processing programs generated by our new translator with those of a naive tree-processing programs (which copy the whole input tree to memory) and the stream-processing programs generated by the previous framework [12]. Figure 13 and 14 show the running times for the same programs. The experiment is conducted on Intel Xeon 5150 CPU with 4 MB cache and 8 GB memory.

As shown in the figures, the stream-processing program generated by the new translator is more efficient than the one generated by `X-P`. The improvement was mainly gained by the lazy construction of hybrid trees, which avoids copying the unnecessary part of the input to memory.

As mentioned in Section 1, our transformation framework has another advantage that the memory space for a hybrid tree can be immediately deallocated when the next tree is

read from the stream. That advantage is, however, not exploited in the current implementation; since the target language of our current translator is Objective Caml, we cannot control memory deallocation. It is left for future work to replace the target language with a lower-level language (so that hybrid trees can be explicitly deallocated) and conduct more experiments to evaluate the advantage of deallocating hybrid trees.

5. RELATED WORK

Besides Suenaga et al.'s framework [3, 11], there are other approaches to automatic transformation of tree-processing programs into stream-processing programs [1, 2, 6, 5, 7, 8]. In those approaches, the source languages for describing tree-processing programs are more restricted than ordinary programming languages (term rewriting [1], query language [2], and attribute grammars [6, 5, 7, 8]). On the other hand, the source language in our framework is an ordinary functional programming language. There are also differences in how and when trees are buffered in memory between our framework and other frameworks. A detailed comparison on this point is left for future work.

Ordered linear type systems have been first studied by Polakow [10], and later by Petersen et al [9] and ourselves [3, 11]. To the authors' knowledge, this is the first application of ordered but *non-linear* types in the context of program transformation. In a different area, non-commutative logic has been studied by Lambek and applied to computational linguistic [4]. It is not clear whether our type system has some connection (in the spirit of Curry-Howard isomorphism) to a non-commutative logic.

6. CONCLUSION

We have introduced an ordered type system to extend Suenaga et al.'s type-based framework [3, 11] for transforming tree-processing programs into stream-processing ones. The use of ordered but non-linear types enabled a more flexible buffering (and hence more efficient stream-processing) of tree-structured data than the previous framework. We have carried out very preliminary experiments and confirmed the effectiveness of the new transformation framework. It is left for future work to fully implement the proposed framework (as a new version of `X-P`) and to carry out more serious experiments.

7. REFERENCES

[1] A. Frisch and K. Nakano. Streaming XML transformation using term rewriting. In *ACM SIGPLAN Workshop on Programming Language Technologies for XML (PLAN-X 2007)*, pages 2–13, 2007.

[2] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. In *Proceedings of the Thirtieth international conference on Very large data bases (VLDB 2004)*, pages 228–239. VLDB Endowment, 2004.

[3] K. Kodama, K. Suenaga, and N. Kobayashi. Translation of tree-processing programs into stream-processing programs based on ordered linear type. *Journal of Functional Programming*, 18(3):333–371, 2008.

[4] J. Lambek. The mathematics of sentence structure. *The American Mathematical Monthly*, 65(3):154–170, 1958.

[5] K. Nakano. An implementation scheme for XML transformation languages through derivation of stream processors. In *the Second Asian Symposium on Programming Languages and Systems (APLAS 2004)*, volume 3302 of *Lecture Notes in Computer Science*, pages 74–90. Springer-Verlag, 2004.

[6] K. Nakano. Composing stack-attributed tree transducers. *Theory of Computing Systems*, 44(1):1–38, 2009.

[7] K. Nakano and S. Nishimura. Deriving event-based document transformers from tree-based specifications. *Electronic Notes in Theoretical Computer Science*, 44(2):181–205, 2001.

[8] S. Nishimura and K. Nakano. XML stream transformer generation through program composition and dependency analysis. *Science of Computer Programming*, 54(2-3):257–290, 2005.

[9] L. Petersen, R. Harper, K. Crary, and F. Pfenning. A type theory for memory allocation and data layout. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2003)*, pages 172–184, New York, NY, USA, 2003. ACM.

[10] J. Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Carnegie Mellon University, June 2001. Available as Technical Report CMU-CS-01-152.

[11] K. Suenaga, N. Kobayashi, and A. Yonezawa. Extension of type-based approach to generation of stream-processing programs by automatic insertion of buffering primitives. In *International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2005)*, volume 3901 of *Lecture Notes in Computer Science*, pages 98–114. Springer-Verlag, 2005.

[12] K. Suenaga, S. Sato, R. Sato, and N. Kobayashi. X-P: XML stream processing program generator. `http://www.kb.ecei.tohoku.ac.jp/~suenaga/x-p/`.