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 a 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 node(t1, f t1) if the tree is a branch. The program accesses t2 before t1, 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, t1 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.

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.

\[
\text{fun name(t) = (get\_firstname(t), get\_lastname(t))}
\]

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

\[
\text{fun name(t) =}
\]

\[
\text{let t' = s2m(t) in}
\]

\[
\text{(get\_firstname(t'), get\_lastname(t'))}
\]

The stream processing program generated from the intermediate program is not so efficient as it could be, because (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...
Figure 2: Hybrid tree during execution

 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 replaced 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, <p> and <s> 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 dealloca-
 ted 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 hy-
 brid tree t' can be immediately reclaimed after being used.

 In the rest of this paper, we first formalize the interme-
 diate 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 de-
 fined, then the translations into/from this language can be
 formalized by extending the authors’ previous work [3, 11]
 with ordered types. We briefly sketch those translations in
 Section 3. Section 4 reports preliminary experiments. Sec-
 tion 5 discusses related work and Section 6 concludes the
 paper. A longer version of this paper is available from http://

 2. INTERMEDIATE LANGUAGE L_I AND
 TYPE SYSTEM

 This section introduces a functional tree-processing language
 L_I, equipped with an ordered type system. The language
 makes distinction among four kinds of trees: (i) ordered lin-
 ear 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 is never read. The ordered linear
type system guarantees that well-typed programs conform
 to such access restrictions on trees.

 The language L_I serves as the intermediate language of
 the transformation framework sketched in Section 1. As dis-
 cussed in Section 3, once the ordered type system for this
 language has been set up, the first step of the transforma-
tion 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. fix(f, x, M) is a
 recursive function that takes an argument x. f is bound to
 the function itself inside M.

 leaf^d and node^d are constructors for binary trees. Here, d,
called a mode, is either 1, 2, or 3, which describes ordered-
 linear, hybrid, buffered, or output trees, respectively. Each
 tree has the different restrictions on access order as men-
tioned before.

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

 EXAMPLE 1. The following program takes a tree as an in-
 put, and returns a list of integers obtained by replacing each
 tree with the sum of its leftmost and second leftmost ele-
 ments. Here, leftmost and leftmostsecond are functions
 which take a hybrid tree and return its leftmost and second
 leftmost elements. Here, let n^M in N is syntaxic sugar
 for fix(f, n, N).

 fix(f, t, case t of
     leaf n -> leaf 0
     node(t1, t2) ->

\[
(\text{fix}(f, x, M) \ triangleright v, B, H, S) \longrightarrow (f \triangleright \text{fix}(f, x, M), x \triangleright v)B, H, S \quad (v \text{ is an integer or of the form fix}(f', x', M'))
\] (E-APP1)

\[
(\text{fix}(f, x, M) \triangleright y, B, H, S) \longrightarrow (f \triangleright \text{fix}(f, x, M), x \triangleright y)B, H, S \quad (E-APP2)
\]

\[
(\text{fix}(f, x, M) \triangleright V, B, H, S) \longrightarrow (f \triangleright \text{fix}(f, x, M), x \triangleright y)B, [y \triangleright V], H, S \quad (y \text{ is fresh}) \quad (E-APP3)
\]

\[
(n_1 + n_2, B, H, S) \longrightarrow (n_1B, H, S) \quad (n_1 \text{ is the sum of } n_1 \text{ and } n_2) \quad (E-PLUS)
\]

\[
(\text{let } x = s2m(y) \text{ in } B, H, S) \longrightarrow (x \triangleright z)B, [z \triangleright V], H, S \quad (z \text{ is fresh}) \quad (E-SToM)
\]

\[
(\text{let } x = s2h(y) \text{ in } B, H, S) \longrightarrow (x \triangleright z)B, [z \triangleright V^4], H, S \quad (z \text{ is fresh}) \quad (E-SToH)
\]

\[
\text{(m2s}(z), B[x \triangleright V], H, S) \longrightarrow (V^+, B[x \triangleright V], H, S) \quad (E-MToS)
\]

\[
(\text{case } x \triangleright \text{leaf } M \mid \text{node}(x_1, x_2) \triangleright M_2, B, H, (x \triangleright \text{leaf } n; S)) \longrightarrow (x_1 \triangleright n)M_1, B, H, S \quad (E-Case1)
\]

\[
(\text{case } x \triangleright \text{node}(x_1, x_2) \triangleright M_2, B, H, (x \triangleright \text{node}(V_1, V_2); S)) \longrightarrow (M_2, B, H, (x_1 \triangleright V_1; x_2 \triangleright V_2; S)) \quad (E-Case2)
\]

\[
(\text{case } y \triangleright \text{leaf } x \triangleright M \mid \text{node}(x_1, x_2) \triangleright M', B[y \triangleright \text{leaf } n], H, S) \longrightarrow (x \triangleright n)M, B[y \triangleright \text{leaf } n], H, S \quad (E-MCase1)
\]

\[
(\text{case } y \triangleright \text{node}(x_1, x_2) \triangleright M', B[y \triangleright \text{node}(V_1, V_2)], H, S) \longrightarrow
\]

\[
(x_1 \triangleright x_1, x_2 \triangleright x_2)M', B[y \triangleright \text{node}(V_1, V_2)], H, S \quad (x_1 \text{ and } x_2 \text{ are fresh}) \quad (E-MCase2)
\]

\[
(\text{case } y \triangleright \text{leaf } x \triangleright M \mid \text{node}(x_1, x_2) \triangleright M', B, H[y \triangleright \text{leaf } n], S) \longrightarrow (x_1 \triangleright n)M, B, H[y \triangleright \text{leaf } n], S \quad (E-Case1)
\]

\[
(\text{case } y \triangleright \text{node}(x_1, x_2) \triangleright M', B, H[y \triangleright \text{node}(V_1, V_2)], S) \longrightarrow
\]

\[
(x_1 \triangleright x_1, x_2 \triangleright x_2)M', B, H[y \triangleright \text{node}(V_1, V_2)], S \quad (x_1 \text{ and } x_2 \text{ are fresh}) \quad (E-HCase1)
\]

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

---

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

```
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 binding matters). In Figure 4, $V^+$ represents the tree obtained by replacing every node annotation in $V$ with $d$. For example, $(\text{leaf } 1)^2$ represents the tree $\text{leaf } 1^2$.

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 $E\text{-SToM}$ and $E\text{-Case1}$, hybrid trees in $H$ are discarded because a variable in $S$ is accessed. In the rules $E\text{-SToH}$, $E\text{-SToM}$ and $E\text{-Case1}$, 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 4 gives the syntax of types. The type $\text{int}$ describes integers and $\tau_1 \rightarrow \tau_2$ describes functions from $\tau_1$ to $\tau_2$. We have four kinds of tree types. $\text{tree}^\omega$ is the type of buffered trees. $\text{tree}^1$ is the type of hybrid trees. $\text{tree}^1$ and $\text{tree}^+$ are the types of input trees and output trees respectively.

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

A type judgment is of the form $\Gamma, \Psi \vdash M : T$. Here, $\Gamma$ is a non-ordered type environment, $\Psi$ is an ordered type environment and $\Delta$ 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 $\text{tree}^e \in \{T_1, \ldots, T_n\}$ implies $d = \omega$. An ordered type environment is a set of the form $\{x_1 : \text{tree}^1, \ldots, x_n : \text{tree}^1\}$, where $x_1, \ldots, x_n$ are different from each other. An ordered linear type environment is a sequence of the form $x_1 : \text{tree}^1, \ldots, x_n : \text{tree}^1$, where $x_1, \ldots, x_n$ are different from each other. We assume that $\Gamma, \Psi, \Delta$ do not share identical variables.

In that judgment, the types express how trees are accessed during the evaluation of $M$. The ordered linear type environment $x_1 : \text{tree}^1, \ldots, x_n : \text{tree}^1$ 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 $x_1 : \text{tree}^1, \ldots, x_n : \text{tree}^1$ 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 $\Delta$ is accessed, none of $x_1, \ldots, x_n$ can be accessed anymore (i.e., ordered). For example, if $\Psi = x_1 : \text{tree}^1, x_2 : \text{tree}^1$ and $\Delta = y : \text{tree}^1$, then both accessing $x_1$, $x_2$ and $y$ and accessing $x_2, x_1, x_1$ and $y$ in these orders are legitimate, while $x_1$, $y$ and $x_2$ is illegal.
**Definition 1.** (Concatenation). An operation \((\Psi_1 \mid \Delta_1) \circ (\Psi_2 \mid \Delta_2)\) is defined as follows.

\[
(\Psi_1 \mid \Delta_1) \circ (\Psi_2 \mid \Delta_2) = \begin{cases} 
(\Psi_1 \cup \Psi_2 \mid \Delta_2) & (\text{if } \Delta_1 = \emptyset) \\
(\Psi_1 \mid (\Delta_1, \Delta_2)) & (\text{if } \Psi_2 = \emptyset) 
\end{cases}
\]

Intuitively, \((\Psi \mid \Delta) = (\Psi_1 \mid \Delta_1) \circ (\Psi_2 \mid \Delta_2)\) are environments that allow trees to be accessed according to \(\Psi_1\) or \(\Delta_1\) and then to \(\Psi_2\) or \(\Delta_2\) sequentially. \((\Psi_1 \mid \Delta_1) \circ (\Psi_2 \mid \Delta_2)\) is defined only when \(\Delta_1 = \emptyset\) or \(\Psi_2 = \emptyset\) because variables in \(\Psi_2\) 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 \(z = s2h(y)\) in \(M\), \(x\) is in the ordered type environment in the premise because \(y\) is converted to a hybrid tree, named \(z\) and used in \(M\).

- T-HCASE is for \(\text{case}\) expressions. Because a hybrid tree can be freely accessed until another variable in the ordered 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 \(x\) \((x_2, x_3)\) can also be used as hybrid trees.

- In the rules T-FiX1 and T-FiX2, both the ordered linear 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 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 \(\text{node}\) \((x_2, x_3)\), 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 \(z_1 = \text{tree}^1, z_2 = \text{tree}^1, \Delta\), 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 \(\text{leftmost} t_1\) and \(\text{leftmost second} t_1\) as hybrid tree \(t'\).

### 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.

\[
\begin{align*}
M & \text{ (terms)} := n \mid x \mid \text{fix}(f, x, M) \mid M_1 \cdot M_2 \\
& \mid M_1 + M_2 \mid \text{leaf} M \mid \text{node}(M_1, M_2) \\
& \mid \text{case} x \text{ of } x_1 \Rightarrow M_1 \\
& \mid \text{node}(x_2, x_3) \Rightarrow M_2
\end{align*}
\]

\(\tau\) (types) := \(\text{int} \mid \tau_1 \Rightarrow \tau_2 \mid \text{tree}\)

**Figure 7:** The syntax of \(\mathcal{L}_S\) and types

**Theorem 1.** (Type soundness). If \(\emptyset \neq \emptyset \mid x : \text{tree}^1 + M : \text{tree}^1\) then \(M' : \text{tree}^1\) and \((M', B', H', S') \rightarrow (M'', B'', H'', S'')\) then \(M'\) is a tree value and \(S' = \emptyset\), or there exist \(M''\), \(B''\), \(H''\), and \(S''\) such that \((M', B', H', S') \rightarrow (M'', B'', H'', S'')\).

### 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 \(s2m\) and \(s2h\) and makes 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 \(\mathcal{L}_T\)

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 \(\text{leaf}\)/\(\text{node}\), \(\text{leaf}/\text{node}^+\), \(\text{leaf}/\text{node}^\ast\), and \(\text{leaf}^+\)/\(\text{node}^\ast\). 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 \(\Gamma \mid \Psi \mid \Delta \vdash M \Rightarrow M' ; \tau\). The judgment means:

- \(M\) and \(M'\) are equivalent except for the representation of trees, and

- \(\Gamma \mid \Psi \mid \Delta \vdash M' ; \tau\).

Figure 10 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 term \(\text{case} x \text{ of } \ldots\). depending on whether the matched tree is translated into an ordered linear, a hybrid or a buffered one. The highlight of the rules is \(\text{Tr-StoM}\) and \(\text{Tr-StoH}\) which insert \(s2m\) and \(s2h\) to source programs. For example, the rule \(\text{Tr-StoH}\) 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 \(M'\), then translation of \(M\) can then convert \(x\) to a hybrid tree here by the \(s2h\) primitive. The rule \(\text{Tr-StoM}\) is similar.
The transformation rules presented above are non-deterministic in the sense that there may be more than one possible $M'$ and $\tau$ that satisfy $\Gamma \vdash \Psi \vdash M \rightarrow M' : \tau$. 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 as in [3, 11]. 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 $L_2$ to $L_T$

Figures 8 shows the syntax of the target language $L_T$, 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 token to the output stream. leaf$^e$ e and node$^e$ (e1, e2) are trees constructed on memory. The term case e of leaf$^e$ e1 | node$^e$ e2 performs a case analysis on the value of e.

Figure 9 presents the semantics of $L_T$. The semantics is expressed as a rewriting of configuration (e, H, L, S1, S2). $H$ is a mapping from locations to hybrid trees. $S_1$ and $S_2$ are the input and the output streams. A stream is a sequence consisting of leaf, node, and integers. L is a sequence of...
locations which have not been accessed yet.

A well-typed $L_1$ program can be translated into an equivalent stream-processing program using the algorithm $A$ defined in Figure 11. The algorithm $A$ converts output tree constructions into stream output operations and case analysis for ordered linear trees into stream input operations. Note that an instruction $\text{flush}$ is inserted before $\text{s2m}$ and $\text{s2h}$ and $\text{case}^1 \ x$ 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 $L_1$ to the stream-processing language $L_2$ in Objective Caml. The current translator supports only binary trees having integers or strings as leaves. An exten-
\[ A(n) = n \]
\[ A(x) = x \]
\[ A(\text{fix}(f, x, M)) = \text{fix}(f, x, A(M)) \]
\[ A(M_1 + M_2) = A(M_1) + A(M_2) \]
\[ A(\text{let } x = \text{s2m}(y) \text{ in } M) = \text{flush}(); \text{let } x = \text{s2m}(y) \text{ in } A(M) \]
\[ A(\text{let } x = \text{s2h}(y) \text{ in } M) = \text{flush}(); \text{let } x = \text{s2h}(y) \text{ in } A(M) \]
\[ A(\text{leaf}^+ M) = \text{write } \text{leaf}; \text{write } A(M) \]
\[ A(\text{node}^+(M_1, M_2)) = \text{node};; A(M_1); A(M_2) \]
\[ A(\text{leaf}^M) = \text{leaf}^M = A(M) \]
\[ A(\text{node}^M(M_1, M_2)) = \text{node}^M(A(M_1), A(M_2)) \]
\[ A(\text{case}^1 x \text{ of } \text{leaf } x_1 \Rightarrow M_1 \mid \text{node}(x_2, x_3) \Rightarrow M_2) = \]
\[ \text{case flush}(); \text{read}(x) \text{ if } \text{leaf } \Rightarrow \text{let } x_1 = \text{read}(x) \text{ in } A(M_1); \text{node}(x_2, x_3) \Rightarrow A(M_2) \]
\[ A(\text{case}^2 x \text{ of } \text{leaf } x_1 \Rightarrow M_1 \mid \text{node}(x_2, x_3) \Rightarrow M_2) = \text{case}^2 x \text{ of } \text{leaf } x_1 \Rightarrow A(M_1); \text{node}(x_2, x_3) \Rightarrow A(M_2) \]
\[ A(\text{case}^3 x \text{ of } \text{leaf } x_1 \Rightarrow M_1 \mid \text{node}(x_2, x_3) \Rightarrow M_2) = \text{case}^3 x \text{ of } \text{leaf } x_1 \Rightarrow A(M_1); \text{node}(x_2, x_3) \Rightarrow A(M_2) \]

Figure 11: Translation algorithm

\[ e \text{ (terms)} ::= n \mid x \mid l \mid \text{leaf} \mid \text{node} \]
\[ (\text{fix}(f, x, e) \mid e_1 \cdot e_2) \mid \text{read } () \mid \text{write } e \]
\[ \text{leaf}^e \mid \text{node}^e(e_1, e_2) \]
\[ \text{let } x = \text{s2m}(y) \text{ in } M \]
\[ \text{let } x = \text{s2h}(y) \text{ in } M \]
\[ \text{flush}() \]
\[ \text{case } e \text{ of } \text{leaf } \Rightarrow e_1 \]
\[ \text{node} \Rightarrow e_2 \]
\[ \text{case}^e \text{ of } \text{leaf } x_1 \Rightarrow e_1 \]
\[ \text{node}(x_2, x_3) \Rightarrow e_2 \]
\[ \text{case}^3 e \text{ of } \text{leaf } x_1 \Rightarrow e_1 \]
\[ \text{node}(x_2, x_3) \Rightarrow e_2 \]

\[ V^w \text{ (trees on mem.)} ::= \text{leaf}^w n \mid \text{node}^w(V^w, V^w) \]
\[ V^d \text{ (hybrid trees)} ::= l \mid \text{leaf}^d n \mid \text{node}^d(V^d, V^d) \]
\[ v \text{ (values)} ::= n \mid \text{leaf} \mid \text{node} \mid \text{fix}(f, x, e) \]
\[ V^w \mid V^d \]

\[ E \text{ (eval. ctx.)} ::= [] \mid E \text{ M} \mid \text{fix}(f, x, e) E \]
\[ E \mid E + e \mid n + E \mid \text{read } E \]
\[ \text{write } E \mid \text{leaf}^e E \]
\[ \text{node}^e(E, e) \mid \text{node}^w(V^w, E) \]
\[ \text{h2m}(E) \]
\[ \text{case } E \text{ of } \text{leaf } \Rightarrow e_1 \]
\[ \text{node} \Rightarrow e_2 \]
\[ \text{case}^E \text{ of } \text{leaf } x_1 \Rightarrow e_1 \]
\[ \text{node}(x_2, x_3) \Rightarrow e_2 \]
\[ \text{case}^3 E \text{ of } \text{leaf } x_1 \Rightarrow e_1 \]
\[ \text{node}(x_2, x_3) \Rightarrow e_2 \]

Figure 8: The syntax of $L_T$

The generation 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 test program for preliminary experiments, we used the following programs written in $L_I$:

- (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 the sum 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.

Here, we used input data generated by a program.

Figures 12–15 show the result of the experiment. Figure 12 and 13 compare the maximum memory consumption of the stream-processing programs generated by our new translator with those of naive tree-processing programs (which copy the whole input tree to memory) and the stream-processing programs generated by the previous framework [12]. Figure 14 and 15 show the running times for the same programs. The experiments are 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