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 \( \text{node}(t_2, f t_1) \) 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 \( \text{s2m} \). 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.

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

Since the function \( \text{name} \) accesses \( t \) twice, a buffering command is inserted in the first step of the transformation, as follows.

\[
\text{fun name}(t) = \text{let } t' = \text{s2m}(t) \text{ in } (\text{get_firstname}(t'), \text{get_lastname}(t'))
\]

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

\[\text{fun name}(t) = \text{let } t' = \text{s2m}(t) \text{ in } (\text{get_firstname}(t'), \text{get_lastname}(t'))\]

\[\text{fun name}(t) = \text{let } t' = \text{s2m}(t) \text{ in } (\text{get_firstname}(t'), \text{get_lastname}(t'))\]

\[\text{fun name}(t) = \text{let } t' = \text{s2m}(t) \text{ in } (\text{get_firstname}(t'), \text{get_lastname}(t'))\]

\[\text{fun name}(t) = \text{let } t' = \text{s2m}(t) \text{ in } (\text{get_firstname}(t'), \text{get_lastname}(t'))\]

\[\text{fun name}(t) = \text{let } t' = \text{s2m}(t) \text{ in } (\text{get_firstname}(t'), \text{get_lastname}(t'))\]

\[\text{fun name}(t) = \text{let } t' = \text{s2m}(t) \text{ in } (\text{get_firstname}(t'), \text{get_lastname}(t'))\]

\[\text{fun name}(t) = \text{let } t' = \text{s2m}(t) \text{ in } (\text{get_firstname}(t'), \text{get_lastname}(t'))\]

\[\text{fun name}(t) = \text{let } t' = \text{s2m}(t) \text{ in } (\text{get_firstname}(t'), \text{get_lastname}(t'))\]

\[\text{fun name}(t) = \text{let } t' = \text{s2m}(t) \text{ in } (\text{get_firstname}(t'), \text{get_lastname}(t'))\]

\[\text{fun name}(t) = \text{let } t' = \text{s2m}(t) \text{ in } (\text{get_firstname}(t'), \text{get_lastname}(t'))\]

\[\text{fun name}(t) = \text{let } t' = \text{s2m}(t) \text{ in } (\text{get_firstname}(t'), \text{get_lastname}(t'))\]

\[\text{fun name}(t) = \text{let } t' = \text{s2m}(t) \text{ in } (\text{get_firstname}(t'), \text{get_lastname}(t'))\]
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, \texttt{s2m} in the program above is replacled by \texttt{s2h}:

\[
\text{fun name}(t) = \text{let } t' = \text{s2h}(t) \text{ in } \ldots
\]

The tree \( t' \) is now a hybrid one. Figure 2 illustrates how the state of \( t' \) changes. That is to say, \(<\ell>\) and \(<\ell>\) stand for \texttt{firstname} and \texttt{lastname}. The tree \( t \) is copied to memory only lazily, when needed by \texttt{get\_firstname} and \texttt{get\_lastname}. The hybrid tree \( t' \) is automatically deallocated after the execution of \texttt{get\_lastname}(t'). Thus, unlike in the previous framework, the part \( t' \) 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] 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 the paper. A longer version of this paper is available from \url{http://www.kb.ecei.tohoku.ac.jp/~ryosuke/papers/FIT2009.pdf}.

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) \textit{ordered linear trees}, which can be accessed only once in the depth-first preorder, (ii) \textit{hybrid trees}, which can be accessed more than once, but only until an ordered linear tree is accessed, (iii) \textit{buffered trees}, which can be accessed without any order or linearity restrictions, and (iv) \textit{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 \( \mathcal{L}_I \) 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. \texttt{fix}(f,x,M) is a recursive function that takes an argument \( x \). \( f \) is bound to the function itself inside \( M \).

\[
\begin{align*}
\text{d (modes)} & : = \{ 1, \# \} | + \\
\text{M (terms)} & : = n | x \mid \text{fix}(f,x,M) \mid M_1 \cdot M_2 \\
& \mid M_1 + M_2 \mid \text{m2s}(x) \\
& \mid \text{let } x = \text{s2m}(y) \text{ in } M \\
& \mid \text{let } x = \text{s2h}(y) \text{ in } M \\
& \mid \text{leaf}^x(M_1, M_2) \\
& \mid \text{case}^x(y) \text{ of } M_1 \Rightarrow M_1 \\
& \mid \text{node}^x(E,M_1) \mid \text{node}^y(E,M) \\
\text{V (trees)} & : = \text{leaf}^x n | \text{node}^x(V, V) \\
\text{v (values)} & : = n | \text{fix}(f,x,M) | V \\
\text{E (eval. ctx.)} & : = [ ] | E \mid [ v ] | E + M \\
& \mid v + E \mid \text{m2s}(E) | \text{leaf}^x E \\
& \mid \text{node}^x(E,M) \mid \text{node}^x(v,E) \\
\text{\tau (types)} & : = \text{int} \mid \tau_1 \rightarrow \tau_2 \mid \text{tree}^d
\end{align*}
\]

Figure 3: The syntax of \( \mathcal{L}_I \) and types
We next introduce an ordered type system for the language. The semantics is expressed as a rewriting relation of configurations of the form $(\Gamma, \Psi, \Delta)$. Here, $\Gamma$ 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, $V$ represents the tree obtained by replacing every node annotation in $V$ with $d$. For example, $(\text{leaf}^1)\tau$ represents the tree $\text{leaf}^1 1$.

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 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. $\text{tree}^\tau$ is the type of buffered trees. $\text{tree}^\tau$ is the type of hybrid trees. $\text{tree}^1$ and $\text{tree}^e$ are the types of input trees and output trees respectively.
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{definition}[Concatenation] An operation \((\Psi_1 | \Delta_1); (\Psi_2 | \Delta_2)\) is defined as follows.

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

Intuitively, \((\Psi | \Delta) = (\Psi_1 | \Delta_1); (\Psi_2 | \Delta_2)\) are environments that allow trees to be accessed according to \(\Psi_1 | \Delta_1\) and then to \(\Psi_2 | \Delta_2\) sequentially. \((\Psi_1 | \Delta_1); (\Psi_2 | \Delta_2)\) 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 \(x = s2h(y)\) 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 case 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 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 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 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 \(x_1: \text{tree}, x_2: \text{tree}, \Delta\), 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 \(t_1\).
3.2 Translation from $L_L$ 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$^e$ and node$^e(e_1, e_2)$ are trees constructed on memory. The term case of leaf 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 pointer to a tree that has been accessed and thus has been constructed. The transformation algorithm is then obtained as a constraint-based algorithm, which first extracts constraints on nodes based on the transformation rules and solves them. We omit a detailed description of the algorithm in this paper.

Figure 9 presents the semantics of $L_T$. The semantics is expressed as a rewriting of configuration (e, H, L, S_i, S_o). $H$ is a mapping from locations to hybrid trees. $S_i$ and $S_o$ are the input and the output streams. A stream is a sequence consisting of leaf, node, and integers. $L$ is a sequence of
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 \texttt{flush} is inserted before \texttt{s2m}, \texttt{s2h} and \texttt{case} \( x \) 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 \( L_T \) in Objective Caml. The current translator supports only binary trees having integers or strings as leaves. An exten-

Figure 9: A part of the operational semantics of the target language

Figure 10: A part of the rules for the judgment \( \Gamma \vdash \Psi \vdash M \leadsto M' \vdash \tau \)
Here, we used input data generated by a program. As a test program for preliminary experiments, we used the language are currently under development.

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

\begin{align*}
\mathcal{A}(n) &= n \\
\mathcal{A}(x) &= x \\
\mathcal{A}(\text{fix}(f, x, M)) &= \text{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}(\text{let } x = \text{s2m}(y) \text{ in } M) &= \text{flush}(); \text{let } x = \text{s2m}() \text{ in } \mathcal{A}(M) \\
\mathcal{A}(\text{let } x = \text{s2h}(y) \text{ in } M) &= \text{flush}(); \text{let } x = \text{s2h}() \text{ in } \mathcal{A}(M) \\
\mathcal{A}(\text{leaf} + M) &= \text{write leaf}; \text{write } \mathcal{A}(M) \\
\mathcal{A}(\text{node}^t(M_1, M_2)) &= \text{write node}; \mathcal{A}(M_1) ; \mathcal{A}(M_2) \\
\mathcal{A}(\text{leaf}^t M) &= \text{leaf}^t \mathcal{A}(M) \\
\mathcal{A}(\text{node}^e(M_1, M_2)) &= \text{node}^e(\mathcal{A}(M_1), \mathcal{A}(M_2)) \\
\mathcal{A}(\text{case}^1 x \text{ of leaf } x_1 \Rightarrow M_1 \text{ node}(x_2, x_3) \Rightarrow M_2) &= \text{case flush}(); \text{read}() \text{ of leaf } \Rightarrow \text{let } x_1 = \text{read}() \text{ in } \mathcal{A}(M_1) | \text{node} \Rightarrow [()] / x_2, (/) / x_3 \mathcal{A}(M_2) \\
\mathcal{A}(\text{case}^e x \text{ of leaf } x_1 \Rightarrow M_1 \text{ node}(x_2, x_3) \Rightarrow M_2) &= \text{case}^e x \text{ of leaf } x_1 \Rightarrow \mathcal{A}(M_1) | \text{node}(x_2, x_3) \Rightarrow \mathcal{A}(M_2) \\
\mathcal{A}(\text{case}^t x \text{ of leaf } x_1 \Rightarrow M_1 \text{ node}(x_2, x_3) \Rightarrow M_2) &= \text{case}^t x \text{ of leaf } x_1 \Rightarrow \mathcal{A}(M_1) | \text{node}(x_2, x_3) \Rightarrow \mathcal{A}(M_2)
\end{align*}

Figure 11: Translation algorithm

$s_2m$ and $s_2h$ are procedures. As for the intermediate language $\mathcal{L}_T$, sentence boundary values +1 and -1 are used to indicate the end of leaf nodes.

$V^\omega$ (trees on mem.) := leaf$^\omega$ n | node$^\omega(V^\omega_1, V^\omega_2)$
$V^t$ (hybrid trees) := l | leaf$^\omega$ n | node$^e(V^\omega_1, V^\omega_2)$
$v$ (values) := n | leaf | node | fix(f, x, e)
$E$ (eval. ctx.) := $E \mid M \mid \text{fix}(f, x, e) \mid \text{write } e \mid n + E \mid \text{read } E \\
\text{write } E \mid \text{leaf}^\omega E \mid \text{node}^e(E, e) \mid \text{node}^e(V^\omega, E) \mid \text{h2m}(E) \\
\text{case } E \text{ of leaf } \Rightarrow e_1 \mid \text{node} \Rightarrow e_2 \\
\text{case}^e E \text{ of leaf } x_1 \Rightarrow e_1 \mid \text{node}(x_2, x_3) \Rightarrow e_2 \\
\text{case}^t E \text{ of leaf } x_1 \Rightarrow e_1 \mid \text{node}(x_2, x_3) \Rightarrow e_2$

In order to avoid copying, the hybrid trees are constructed lazily.

Figure 9: Example 1

(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.

As a test program for preliminary experiments, we used the following programs written in $\mathcal{L}_T$:

- (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 a naive tree-processing programs (which copy the whole input tree to memory) and the stream-processing programs generated by the previous framework [12]. Figures 14 and 15 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 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

http://www.kb.ecei.tohoku.ac.jp/~suenaga/x-p/.