Recommended Paper

Ordered Types for Stream Processing of Tree-Structured Data

Ryosuke Sato,[†1] Kohei Suenaga[†1]
and Naoki Kobayashi[†1]

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.[1),2)] 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, while keeping the readability and maintainability of functional programs. Based on the framework, they have implemented an XML stream-processing program generator `X-P`[3)].

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 type inference[2)], 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 store an integer value at each leaf. The type of the binary trees can be described by the following **datatype** definition of ML.

$$\textbf{datatype tree} = \textbf{leaf of int} \mid \textbf{node of tree} * \textbf{tree}.$$

The program takes a binary tree t as input, conducts pattern matching on the tree and returns $\textbf{node}(t_2, f\ t_1)$ 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* into 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.

A shortcoming of the framework of Suenaga, et al.[1)] 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 programs. 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 representing a record of a person as an input, and returns the first and last names.

$$\mathit{name}(t) \overset{\text{def}}{\equiv} (\mathit{get\_firstname}\ t, \mathit{get\_lastname}\ t)$$

Here, we assume that a person record has the following structure:

[†1] Tohoku University
The initial version of this paper was presented at FIT2009, the 8th Forum on Information Technology held on Sep. 2009. This paper was recommended to be submitted to IPSJ Journal by the program chair of FIT2009.

```
Source program:
  let rec $f\ t$ = case $t$ of
      leaf $n \Rightarrow$ leaf $n$
    | node$(t_1, t_2) \Rightarrow$ node$(t_2,\ f\ t_1)$
Intermediate program:
  let rec $f\ t$ = case $t$ of
      leaf $n \Rightarrow$ leaf $n$
    | node$(t_1, t_2) \Rightarrow$ let $t_1 =$ s2m$(t_1)$ in node$(t_2,\ f\ t_1)$
Target program:
  let rec $f$ () = case read() of
      leaf $\Rightarrow$ write leaf; write (read())
    | node $\Rightarrow$ let $t_1 =$ s2m() in
        write node; copy (); copymem $(f\ t_1)$
```

Fig. 1 Suenaga, et al.'s translation framework.

`<p><f>Ryosuke</f><l>Sato</l><s>...</s>...</p>`

which has various fields other than first and last names. Since the function *name* accesses t twice, a buffering command is inserted in the first step of the transformation, as follows.

$$name(t) \overset{\text{def}}{\equiv} \textbf{let } t' = \textbf{s2m}(t) \textbf{ in } (get\_firstname\ t', get\_lastname\ t')$$

The stream processing program generated from the intermediate program is not so efficient as it could be, because (i) the whole tree t is copied on memory, though only the first and the last names of t are used later, 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 replaced by **s2h**:

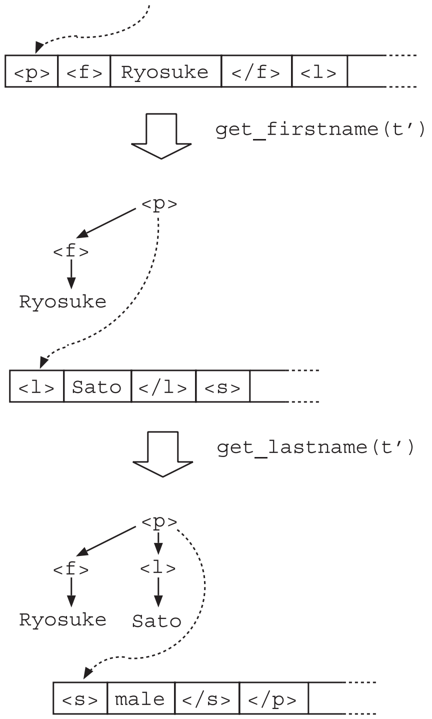


Fig. 2 Hybrid tree during execution.

$$name(t) \overset{\text{def}}{\equiv} \textbf{let } t' = \textbf{s2h}(t) \textbf{ in } (get\_firstname\ t', get\_lastname\ t')$$

The tree t' is now a hybrid one. **Figure 2** illustrates how the state of t' changes. In this figure, the solid arrows represent pointers to nodes of the binary tree constructed on memory and the dotted arrows represent pointers to the head of the input stream. In the figure, `<f>` and `<l>` stand for `<firstname>` and `<lastname>`. The tree t' is constructed on memory only lazily, when requested.

For example, just after the evaluation of **s2h**, a pointer to the input stream is created and t' is bound to the pointer. Then, when *get_firstname*(t') is evaluated, the part `<p><f>Ryosuke</f>` is read from the input stream and constructed on memory. The evaluation of *get_lastname*(t') proceeds similarly. The hybrid tree t' is automatically deallocated after the execution of *get_lastname*. Thus, unlike in the previous framework, the part `<s>...</s>` shown in Fig. 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 consists of unrestricted types, ordered types and ordered linear types that we mentioned above) and discuss soundness of the type system 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 [1),2)] 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.

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 are never destructed in the program. 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 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 straightforward manner.

$$
\begin{array}{lcl}
d \text{ (modes)} & ::= & 1 \mid \sharp \mid \omega \mid + \\
M \text{ (terms)} & ::= & n \mid x \mid \mathbf{fix}(f,x,M) \mid M_1\ M_2 \mid M_1 + M_2 \mid \mathbf{m2s}(x) \\
 & \mid & \mathbf{let}\ x = \mathbf{s2m}(y)\ \mathbf{in}\ M \mid \mathbf{let}\ x = \mathbf{s2h}(y)\ \mathbf{in}\ M \\
 & \mid & \mathbf{leaf}^d\ M \mid \mathbf{node}^d(M_1, M_2) \\
 & \mid & \mathbf{case}^d\ x\ \mathbf{of\ leaf}\ y \Rightarrow M_1 \mid \mathbf{node}(x_1,x_2) \Rightarrow M_2 \\
V \text{ (trees)} & ::= & \mathbf{leaf}^\sharp\ n \mid \mathbf{leaf}^\omega\ n \mid \mathbf{leaf}^+\ n \\
 & \mid & \mathbf{node}^\sharp(x_1,x_2) \mid \mathbf{node}^\omega(x_1,x_2) \mid \mathbf{node}^+(V_1,V_2) \\
v \text{ (values)} & ::= & x \mid n \mid \mathbf{fix}(f,x,M) \mid V \\
E \text{ (evalčtx.)} & ::= & [\,] \mid E\ M \mid v\ E \mid E + M \mid v + E \mid \mathbf{m2s}(E) \\
 & \mid & \mathbf{leaf}^+\ E \mid \mathbf{node}^+(E,M) \mid \mathbf{node}^+(v,E) \\
 & \mid & \mathbf{leaf}^\omega\ E \mid \mathbf{node}^\omega(E,M) \mid \mathbf{node}^\omega(v,E) \\
\tau \text{ (types)} & ::= & \mathbf{int} \mid \tau_1 \to \tau_2 \mid \mathbf{tree}^d
\end{array}
$$

Fig. 3 The syntax of $\mathcal{L}_I$ and types.

2.1 Language

Figure 3 shows the syntax of the language $\mathcal{L}_I$. 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 to a buffered one, binds x to it and evaluates M. $\mathbf{m2s}(x)$ converts a buffered tree to an output tree. $\mathbf{let}\ x = \mathbf{s2h}(y)\ \mathbf{in}\ M$ converts an ordered linear tree y to a hybrid tree, binds x to it and evaluates M. For the sake of simplicity, we allow only variables as arguments of **s2m**, **m2s**, and **s2h**. The $\mathbf{case}^d$ expression performs case analysis on each kind of trees. Here, d must not be $+$, as output trees cannot be destructed.

Example 1. *The following program takes a tree as an input, and returns (the* *tree representation of) a list obtained by replacing the left subtree of each node with the sum of the values of the leftmost and second leftmost leaves.*

```
$\mathbf{fix}(f, t,$
    $\mathbf{case}\ t\ \mathbf{of}$
        $\mathbf{leaf}\ n\ \Rightarrow\ \mathbf{leaf}\ 0$
      $\mid \mathbf{node}(t_1, t_2)\ \Rightarrow$
        $\mathbf{let}\ t_1' = \mathbf{s2h}\ t_1\ \mathbf{in}$
        $\mathbf{let}\ n = \mathit{leftmost}\ t_1' + \mathit{secondleftmost}\ t_1'\ \mathbf{in}$
          $\mathbf{node}(\mathbf{leaf}\ n, f\ t_2)$
```

Here, leftmost and secondleftmost take a hybrid tree and return its leftmost and second leftmost elements, respectively. The expression **let** $n = M$ **in** N *is a syntax sugar for* $\mathbf{fix}(f, n, N)\ M$.

Figure 4 shows the operational semantics of the language, defined as a rewriting relation of configurations of the form (M, B, H, S). Here, M is the term that is being evaluated. B is a heap on which buffered trees are constructed. 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). S represents trees stored in the input stream, where the leftmost binding corresponds to the tree stored at the head of the input stream. In Fig. 4, $\mathit{Alloc}^{\omega}(B, x, V)$ allocates a tree V on a heap B and let x refer to the tree V. $\mathit{Alloc}^{\sharp}(H, x, V)$ is defined in a similar manner. $\mathit{ToTree}(B, x)$ takes a heap B and a variable x and returns a tree referred to by x. V^d represents the tree obtained by replacing every mode annotation in V with d. For example, $(\mathbf{leaf}^1\ 1)^+$ represents the tree $\mathbf{leaf}^+\ 1$.

Note that we use the three tree environments B, H and S in order to express the difference on access restrictions among the different kinds of trees. When a variable in S is accessed (in rules E-StoM, E-StoH, E-Case1, and E-Case2), the variable must be at the head of S; furthermore, the hybrid trees stored in H are discarded. 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 $\mathcal{L}_I$. The type system guarantees that well-typed programs access trees in a valid order.

$$
\begin{array}{lcl}
\mathit{Alloc}^{\omega}(B, x, \mathbf{leaf}^{\omega}\ n) & = & B[x \mapsto \mathbf{leaf}^{\omega}\ n] \\
\mathit{Alloc}^{\omega}(B, x, \mathbf{node}^{\omega}(V_1, V_2)) & = & B[x \mapsto \mathbf{node}^{\omega}(x_1, x_2)] \cup \mathit{Alloc}^{\omega}(\emptyset, x_1, V_1) \cup \mathit{Alloc}^{\omega}(\emptyset, x_2, V_2) \\
 & & \qquad (x_1 \text{ and } x_2 \text{ are fresh.}) \\
\mathit{Alloc}^{\sharp}(H, x, \mathbf{leaf}^{\sharp}\ n) & = & H[x \mapsto \mathbf{leaf}^{\sharp}\ n] \\
\mathit{Alloc}^{\sharp}(H, x, \mathbf{node}^{\sharp}(V_1, V_2)) & = & H[x \mapsto \mathbf{node}^{\sharp}(x_1, x_2)] \cup \mathit{Alloc}^{\sharp}(\emptyset, x_1, V_1) \cup \mathit{Alloc}^{\sharp}(\emptyset, x_2, V_2) \\
 & & \qquad (x_1 \text{ and } x_2 \text{ are fresh.}) \\
\mathit{ToTree}(B[x \mapsto \mathbf{leaf}^{\omega}\ n], x) & = & \mathbf{leaf}^{\omega}\ n \\
\mathit{ToTree}(B[x \mapsto \mathbf{node}^{\omega}(x_1, x_2)], x) & = & \mathbf{node}^{\omega}(\mathit{ToTree}(B, x_1), \mathit{ToTree}(B, x_2))
\end{array}
$$

$$
\begin{array}{c}
(\mathbf{fix}(f, x, M)\ v, B, H, S) \longrightarrow \\
([f \mapsto \mathbf{fix}(f, x, M), x \mapsto v]M, B, H, S)
\end{array}
\ \text{(E-App)}
\qquad
\frac{n \text{ is the sum of } n_1 \text{ and } n_2}{(n_1 + n_2, B, H, S) \longrightarrow (n, B, H, S)}\ \text{(E-Plus)}
$$

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

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

$$
\frac{\mathit{ToTree}(B, x) = V}{(\mathbf{m2s}(x), B, H, S) \longrightarrow (V^{+}, B, H, S)}\ \text{(E-MtoS)}
$$

$$
\frac{x' \text{ is fresh}}{(\mathbf{leaf}^{\omega}\ n, B, H, S) \longrightarrow (x', B[x' \mapsto \mathbf{leaf}^{\omega}\ n], H, S)}\ \text{(E-Leaf)}
$$

$$
\frac{x' \text{ is fresh}}{(\mathbf{node}^{\omega}(x_1, x_2), B, H, S) \longrightarrow (x', B[x' \mapsto \mathbf{node}^{\omega}(x_1, x_2)], H, S)}\ \text{(E-Node)}
$$

$$
\begin{array}{c}
(\mathbf{case}^1\ x\ \mathbf{of}\ \mathbf{leaf}\ x_1 \Rightarrow M_1 \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2, B, H, (x \mapsto \mathbf{leaf}^1\ n; S)) \longrightarrow \\
([x_1 \mapsto n]M_1, B, \emptyset, S)
\end{array}
\ \text{(E-Case1)}
$$

$$
\begin{array}{c}
(\mathbf{case}^1\ x\ \mathbf{of}\ \mathbf{leaf}\ x_1 \Rightarrow M_1 \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2, B, H, (x \mapsto \mathbf{node}^1(V_1, V_2); S)) \longrightarrow \\
(M_2, B, \emptyset, (x_2 \mapsto V_1; x_3 \mapsto V_2; S))
\end{array}
\ \text{(E-Case2)}
$$

$$
\frac{B(x) = \mathbf{leaf}^{\omega}\ n}{(\mathbf{case}^{\omega}\ x\ \mathbf{of}\ \mathbf{leaf}\ x_1 \Rightarrow M_1 \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2, B, H, S) \longrightarrow ([x_1 \mapsto n]M_1, B, H, S)}\ \text{(E-MCase1)}
$$

$$
\frac{B(x) = \mathbf{node}^{\omega}(x_2', x_3')}{(\mathbf{case}^{\omega}\ x\ \mathbf{of}\ \mathbf{leaf}\ x_1 \Rightarrow M_1 \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2, B, H, S) \longrightarrow ([x_2 \mapsto x_2', x_3 \mapsto x_3']M_2, B, H, S))}\ \text{(E-MCase2)}
$$

$$
\frac{H(x) = \mathbf{leaf}^{\sharp}\ n}{(\mathbf{case}^{\sharp}\ x\ \mathbf{of}\ \mathbf{leaf}\ x_1 \Rightarrow M_1 \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2, B, H, S) \longrightarrow ([x_1 \mapsto n]M_1, B, H, S)}\ \text{(E-HCase1)}
$$

$$
\frac{H(x) = \mathbf{node}^{\sharp}(x_2', x_3')}{\begin{array}{c}(\mathbf{case}^{\sharp}\ x\ \mathbf{of}\ \mathbf{leaf}\ x_1 \Rightarrow M_1 \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2, B, H, S) \longrightarrow \\ ([x_2 \mapsto x_2', x_3 \mapsto x_3']M_2, B, H, S)\end{array}}\ \text{(E-HCase2)}
$$

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

Fig. 4 Operational semantics of $\mathcal{L}_I$.

Figure 3 gives the syntax of types. The type **int** describes integers and $\tau_1 \rightarrow \tau_2$ 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.

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 an arbitrary manner. Though trees of type $\mathbf{tree}^\sharp$ can be accessed an arbitrary 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 variables.

In the judgment $\Gamma \mid \Psi \mid \Delta \vdash M : \tau$, 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 that $x_1, \ldots, x_n$ are bound to trees, but also that each of $x_1, \ldots, x_n$ must be accessed exactly once in this order and that each of the trees bound to $x_1, \ldots, x_n$ must be accessed in the left-to-right, depth-first preorder. 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 the access order among $x_1, \ldots, x_n$. However, if a variable in Δ is accessed, the variables in Ψ can no longer be accessed (i.e., ordered). For example, if $\Psi = x_1 : \mathbf{tree}^\sharp, x_2 : \mathbf{tree}^\sharp$ and $\Delta = y : \mathbf{tree}^1$, then the access sequences $x_1; x_1; y$ and $x_2; x_2; x_1; y$ are legitimate, while $x_1; y; x_2$ is not.

Definition 1 (Concatenation)**.** *A partial operation* $(\Psi_1 \mid \Delta_1); (\Psi_2 \mid \Delta_2)$ *is defined as follows.*

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

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 program is accessing an ordered linear tree, so that it is not allowed to access hybrid trees, thus the ordered type environment in the conclusion has to be empty. 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, T-Fix2, and T-Fix3, 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, and T-Node, 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.

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

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

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

$$\dfrac{\Gamma, f : \mathbf{tree}^1 \to \tau_2 \mid \emptyset \mid x : \mathbf{tree}^1 \vdash M : \tau_2}{\Gamma \mid \emptyset \mid \emptyset \vdash \mathbf{fix}(f, x, M) : \mathbf{tree}^1 \to \tau_2} \ \text{(T-}\textsc{Fix3}\text{)}$$

$$\dfrac{\Gamma \mid \Psi_1 \mid \Delta_1 \vdash M_1 : \tau_1 \to \tau_2 \qquad \Gamma \mid \Psi_2 \mid \Delta_2 \vdash M_2 : \tau_1}{\Gamma \mid (\Psi_1 \mid \Delta_1); (\Psi_2 \mid \Delta_2) \vdash M_1 M_2 : \tau_2} \ \text{(T-}\textsc{App}\text{)}$$

$$\dfrac{\Gamma \mid \Psi_1 \mid \Delta_1 \vdash M_1 : \mathbf{int} \qquad \Gamma \mid \Psi_2 \mid \Delta_2 \vdash M_2 : \mathbf{int}}{\Gamma \mid (\Psi_1 \mid \Delta_1); (\Psi_2 \mid \Delta_2) \vdash M_1 + M_2 : \mathbf{int}} \ \text{(T-}\textsc{Plus}\text{)}$$

$$\dfrac{\Gamma, x : \mathbf{tree}^\omega \mid \emptyset \mid \Delta \vdash M : \tau}{\Gamma \mid \emptyset \mid y : \mathbf{tree}^1, \Delta \vdash \mathbf{let}\ x = \mathbf{s2m}(y)\ \mathbf{in}\ M : \tau} \ \text{(T-}\textsc{StoM}\text{)}$$

$$\dfrac{\Gamma \mid x : \mathbf{tree}^\sharp \mid \Delta \vdash M : \tau}{\Gamma \mid \emptyset \mid y : \mathbf{tree}^1, \Delta \vdash \mathbf{let}\ x = \mathbf{s2h}(y)\ \mathbf{in}\ M : \tau} \ \text{(T-}\textsc{StoH}\text{)}$$

$$\dfrac{\Gamma \mid \Psi \mid \Delta \vdash M : \mathbf{tree}^\omega}{\Gamma \mid \Psi \mid \Delta \vdash \mathbf{m2s}(M) : \mathbf{tree}^+} \ \text{(T-}\textsc{MtoS}\text{)} \qquad \dfrac{\Gamma \mid \Psi \mid \Delta \vdash M : \mathbf{int} \qquad d \in \{\omega, +\}}{\Gamma \mid \Psi \mid \Delta \vdash \mathbf{leaf}^d\ M : \mathbf{tree}^d} \ \text{(T-}\textsc{Leaf}\text{)}$$

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

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

$$\dfrac{\begin{array}{c}\Gamma, x : \mathbf{tree}^\omega, x_1 : \mathbf{int} \mid \Psi \mid \Delta \vdash M_1 : \tau \\ \Gamma, x : \mathbf{tree}^\omega, x_2 : \mathbf{tree}^\omega, x_3 : \mathbf{tree}^\omega \mid \Psi \mid \Delta \vdash M_2 : \tau\end{array}}{\Gamma, x : \mathbf{tree}^\omega \mid \Psi \mid \Delta \vdash \mathbf{case}^\omega\ x\ \mathbf{of\ leaf}\ x_1 \Rightarrow M_1 \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2 : \tau} \ \text{(T-MC}\textsc{ase}\text{)}$$

$$\dfrac{\begin{array}{c}\Gamma, x_1 : \mathbf{int} \mid x : \mathbf{tree}^\sharp, \Psi \mid \Delta \vdash M_1 : \tau \\ \Gamma \mid x : \mathbf{tree}^\sharp, x_2 : \mathbf{tree}^\sharp, x_3 : \mathbf{tree}^\sharp, \Psi \mid \Delta \vdash M_2 : \tau\end{array}}{\Gamma \mid x : \mathbf{tree}^\sharp, \Psi \mid \Delta \vdash \mathbf{case}^\sharp\ x\ \mathbf{of\ leaf}\ x_1 \Rightarrow M_1 \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2 : \tau} \ \text{(T-HC}\textsc{ase}\text{)}$$

Fig. 5 The typing rules.

$$\underline{\dfrac{\dfrac{\vdots}{\Gamma' \mid t_1' : \mathbf{tree}^\sharp \mid \emptyset \vdash \mathbf{leftmost}\ t_1' : \mathbf{int}} \quad \dfrac{\vdots}{\Gamma' \mid t_1' : \mathbf{tree}^\sharp \mid \emptyset \vdash \mathbf{leftmostsecond}\ t_1' : \mathbf{int}}}{\Gamma' \mid t_1' : \mathbf{tree}^\sharp \mid \emptyset \vdash \mathbf{leftmost}\ t_1' + \mathbf{leftmostsecond}\ t_1' : \mathbf{int}}}$$

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

Fig. 6 A type derivation tree. $\Gamma = \{\mathbf{leftmost} : \mathbf{tree}^\sharp \to \mathbf{int}, \mathbf{leftmostsecond} : \mathbf{tree}^\sharp \to \mathbf{int}\}$, $\Gamma' = \Gamma, f : \mathbf{tree}^1 \to \mathbf{tree}^+$.

Figure 6 shows a type derivation tree for the program presented in Section 2.

2.3 Type Soundness

We state the 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 may not respect the order restriction on an input tree, the algorithm first inserts buffering primitives **s2m**, **s2h** and **m2s** to make the program well-typed. This step is conducted by performing type inference for the type system introduced in Section 2. Then, the algorithm generates a stream-processing program by replacing each tree primitive with a corresponding stream primitive.

Figure 7 gives the syntax of the source language $\mathcal{L}_S$. The language differs from $\mathcal{L}_I$ in Section 2 in that $\mathcal{L}_S$ does not have buffering primitives, and does not make the distinction among $\mathbf{leaf}^1/\mathbf{node}^1$, $\mathbf{leaf}^\omega/\mathbf{node}^\omega$, $\mathbf{leaf}^\sharp/\mathbf{node}^\sharp$ and

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

Fig. 7 The syntax of $\mathcal{L}_S$ and types.

$\mathbf{leaf}^{+}/\mathbf{node}^{+}$. Similarly, $\mathcal{L}_S$ makes no distinction among $\mathbf{tree}^1$, $\mathbf{tree}^{\sharp}$, $\mathbf{tree}^{\omega}$, $\mathbf{tree}^{+}$. Thus, a user can write a source program without considering the order and linearity restrictions. We assume that programs in $\mathcal{L}_S$ are well-typed in the standard type system for the simply-typed lambda-calculus, having int and tree as base types. The types of tree primitives are given by:

$$
\begin{array}{l}
\mathbf{leaf} : \mathbf{int} \to \mathbf{tree} \\
\mathbf{node} : \mathbf{tree} \to \mathbf{tree} \to \mathbf{tree} \\
\mathbf{case} : \forall\alpha.\mathbf{tree} \to (\mathbf{int} \to \alpha) \to (\mathbf{tree} \to \mathbf{tree} \to \alpha) \to \alpha
\end{array}
$$

(where $\mathbf{case}\ x\ \mathbf{of}\ \mathbf{leaf}\ x_1 \Rightarrow M_1 \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2$ is interpreted as $\mathbf{case}\ x\ (\lambda x_1.M_1)\ (\lambda x_2.\lambda x_3.M_2)$.)

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

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. [2], we introduce type-based, non-deterministic translation rules. Then the translation algorithm is obtained as a type inference algorithm in a manner similar to Suenaga, et al. [2]

The non-deterministic translation is given by a judgment $\Gamma \mid \Psi \mid \Delta \vdash M \leadsto M' : \tau$. 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 8 shows the rules for the judgment $\Gamma \mid \Psi \mid \Delta \vdash M \leadsto M' : \tau$. The rules are non-deterministic in the sense that there may be more than one valid transformation for each source program M. For example, there are three rules for the term $\mathbf{case}\ x\ \mathbf{of}\ \ldots$ depending on whether the matched tree is translated into an ordered linear, a hybrid or a buffered one. The key rules are Tr-StoM and

$$\Gamma \mid \Psi \mid \emptyset \vdash n \leadsto n : \mathbf{int} \qquad (\textsc{Tr-Int})$$

$$\Gamma \mid \Psi \mid x : \mathbf{tree}^1 \vdash x \leadsto x : \mathbf{tree}^1 \qquad (\textsc{Tr-Var1})$$

$$\Gamma \mid x : \mathbf{tree}^{\sharp}, \Psi \mid \emptyset \vdash x \leadsto x : \mathbf{tree}^{\sharp} \qquad (\textsc{Tr-Var2})$$

$$\Gamma, x : \tau \mid \Psi \mid \emptyset \vdash x \leadsto x : \tau \qquad (\textsc{Tr-Var3})$$

$$\frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \mid \emptyset \mid \emptyset \vdash M \leadsto M' : \tau_2}{\Gamma \mid \emptyset \mid \emptyset \vdash \mathbf{fix}(f, x, M) \leadsto \mathbf{fix}(f, x, M') : \tau_1 \to \tau_2} \qquad (\textsc{Tr-Fix1})$$

$$\frac{\Gamma, f : \mathbf{tree}^{\sharp} \to \tau \mid x : \mathbf{tree}^{\sharp} \mid \emptyset \vdash M \leadsto M' : \tau}{\Gamma \mid \emptyset \mid \emptyset \vdash \mathbf{fix}(f, x, M) \leadsto \mathbf{fix}(f, x, M') : \mathbf{tree}^{\sharp} \to \tau} \qquad (\textsc{Tr-Fix2})$$

$$\frac{\Gamma, f : \mathbf{tree}^1 \to \tau \mid \emptyset \mid x : \mathbf{tree}^1 \vdash M \leadsto M' : \tau}{\Gamma \mid \emptyset \mid \emptyset \vdash \mathbf{fix}(f, x, M) \leadsto \mathbf{fix}(f, x, M') : \mathbf{tree}^1 \to \tau} \qquad (\textsc{Tr-Fix3})$$

$$\frac{\Gamma \mid \Psi_1 \mid \Delta_1 \vdash M_1 \leadsto M_1' : \tau_1 \to \tau_2 \qquad \Gamma \mid \Psi_2 \mid \Delta_2 \vdash M_2 \leadsto M_2' : \tau_1}{\Gamma \mid (\Psi_1 \mid \Delta_1); (\Psi_2 \mid \Delta_2) \vdash M_1 M_2 \leadsto M_1' M_2' : \tau_2} \qquad (\textsc{Tr-App})$$

$$\frac{\Gamma \mid \Psi_1 \mid \Delta_1 \vdash M_1 \leadsto M_1' : \mathbf{int} \qquad \Gamma \mid \Psi_2 \mid \Delta_2 \vdash M_2 \leadsto M_2' : \mathbf{int}}{\Gamma \mid (\Psi_1 \mid \Delta_1); (\Psi_2 \mid \Delta_2) \vdash M_1 + M_2 \leadsto M_1' + M_2' : \mathbf{int}} \qquad (\textsc{Tr-Plus})$$

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

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

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

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

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

$$\frac{\Gamma, x : \mathbf{tree}^{\omega} \mid \emptyset \mid \Delta \vdash M \leadsto M' : \tau \qquad y \text{ is fresh}}{\Gamma \mid \emptyset \mid x : \mathbf{tree}^1, \Delta \vdash M \leadsto \mathbf{let}\ y = \mathbf{s2m}(x)\ \mathbf{in}\ [x \mapsto y]M' : \tau} \qquad (\textsc{Tr-StoM})$$

$$\frac{\Gamma \mid x : \mathbf{tree}^{\sharp} \mid \Delta \vdash M \leadsto M' : \tau \qquad y \text{ is fresh}}{\Gamma \mid \emptyset \mid x : \mathbf{tree}^1, \Delta \vdash M \leadsto \mathbf{let}\ y = \mathbf{s2h}(x)\ \mathbf{in}\ [x \mapsto y]M' : \tau} \qquad (\textsc{Tr-StoH})$$

$$\frac{\Gamma \mid \Psi \mid \Delta \vdash M \leadsto M' : \mathbf{tree}^{\omega}}{\Gamma \mid \Psi \mid \Delta \vdash M \leadsto \mathbf{m2s}(M') : \mathbf{tree}^{+}} \qquad (\textsc{Tr-MtoS})$$

Fig. 8 The rules for the judgment $\Gamma \mid \Psi \mid \Delta \vdash M \leadsto M' : \tau$.

Tr-StoH, which insert **s2m** and **s2h** into source programs. 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 the translation of M, then one can convert x to a hybrid tree here by the **s2h** primitive. The rule Tr-StoM can be read in a similar manner.

Note that there is at least one valid transformation for every simply-typed program: every program in $\mathcal{L}_S$ can be translated into one in $\mathcal{L}_I$ that first buffers every input tree on heap by **s2m**. A deterministic algorithm is obtained as a type inference algorithm as in Suenaga, et al.'s work[2]. By merging the 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.

We suppose the transformation rules are correct in the sense that a valid transformation always exists, and that the source program evaluates to a value if and only if the translated program evaluates to the same value. The proof would be similar to the correctness proof of the transformation of our previous work[1].

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

Figure 9 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 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 lazy reading of 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 are currently kept. $\mathbf{case}^\omega\ e\ \mathbf{of}\ \ldots$ and $\mathbf{case}^\sharp\ e\ \mathbf{of}\ \ldots$ are pattern matching on buffered and hybrid trees, respectively. We write $e_1; e_2$ for $\mathbf{fix}(f, x, e_2)e_1$ where f and x are not free in e_2.

$$
\begin{array}{lcl}
e \text{ (terms)} & ::= & n \mid x \mid l \mid \mathbf{leaf} \mid \mathbf{node} \mid () \mid \mathbf{fix}(f,x,e) \mid e_1\ e_2 \\
 & \mid & e_1 + e_2 \mid \mathbf{read}() \mid \mathbf{write}\ e \\
 & \mid & \mathbf{leaf}^\omega\ e \mid \mathbf{node}^\omega(e_1,e_2) \\
 & \mid & \mathbf{let}\ x = \mathbf{s2m}()\ \mathbf{in}\ e \\
 & \mid & \mathbf{let}\ x = \mathbf{s2h}()\ \mathbf{in}\ e \\
 & \mid & \mathbf{flush}() \\
 & \mid & \mathbf{case}\ e\ \mathbf{of\ leaf} \Rightarrow e_1 \mid \mathbf{node} \Rightarrow e_2 \\
 & \mid & \mathbf{case}^\omega\ e\ \mathbf{of\ leaf}\ x_1 \Rightarrow e_1 \mid \mathbf{node}(x_2,x_3) \Rightarrow e_2 \\
 & \mid & \mathbf{case}^\sharp\ e\ \mathbf{of\ leaf}\ x_1 \Rightarrow e_1 \mid \mathbf{node}(x_2,x_3) \Rightarrow e_2 \\
V^\omega \text{ (trees on mem.)} & ::= & \mathbf{leaf}^\omega\ n \mid \mathbf{node}^\omega(x_1,x_2) \\
V^\sharp \text{ (hybrid trees)} & ::= & l \mid \mathbf{leaf}^\omega\ n \mid \mathbf{node}^\omega(l_1,l_2) \\
v \text{ (values)} & ::= & n \mid \mathbf{leaf} \mid \mathbf{node} \mid \mathbf{fix}(f,x,e) \mid V^\omega \mid V^\sharp \\
E \text{ (eval. ctx.)} & ::= & [\,] \mid E\ M \mid \mathbf{fix}(f,x,e)\ E \mid E + e \mid n + E \\
 & \mid & \mathbf{read}()E \mid \mathbf{write}\ E \mid \mathbf{leaf}^\omega\ E \\
 & \mid & \mathbf{node}^\omega(E,e) \mid \mathbf{node}^\omega(V^\omega,E) \mid \mathbf{h2m}(E) \\
 & \mid & \mathbf{case}\ E\ \mathbf{of\ leaf} \Rightarrow e_1 \mid \mathbf{node} \Rightarrow e_2 \\
 & \mid & \mathbf{case}^\omega\ E\ \mathbf{of\ leaf}\ x_1 \Rightarrow e_1 \mid \mathbf{node}(x_2,x_3) \Rightarrow e_2 \\
 & \mid & \mathbf{case}^\sharp\ E\ \mathbf{of\ leaf}\ x_1 \Rightarrow e_1 \mid \mathbf{node}(x_2,x_3) \Rightarrow e_2
\end{array}
$$

Fig. 9 The syntax of $\mathcal{L}_T$.

The semantics of $\mathcal{L}_T$ is expressed as a rewriting of configuration (e, B, H, L, S_i, S_o). e is the term that is being evaluated. B is a *memory heap*, a mapping from variables to buffered trees. H is a *hybrid heap*, 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 locations which have not been accessed yet.

We introduce an auxiliary function to define the semantics of $\mathcal{L}_T$.

Definition 2. $[\![V]\!]$ *is defined as follows.*

$$[\![\mathbf{leaf}^\omega n]\!] = \mathbf{leaf}; n$$
$$[\![\mathbf{node}^\omega(V_1, V_2)]\!] = \mathbf{node}; [\![V_1]\!]; [\![V_2]\!].$$

$[\![V]\!]$ represents a part of a stream that encodes a buffered tree V.

Figure 10 presents the semantics of $\mathcal{L}_T$. The rules for hybrid trees (E-Flush1,

$$(\mathbf{fix}(f,x,M)\ v, B, H, L, S_i, S_o) \longrightarrow ([f \mapsto \mathbf{fix}(f,x,M), x \mapsto v]M, B, H, L, S_i, S_o) \qquad \text{(E-}\textsc{App}\text{)}$$

$$\frac{(n \text{ is the sum of } n_1 \text{ and } n_2)}{(n_1 + n_2, B, H, L, S_i, S_o) \longrightarrow (n, B, H, L, S_i, S_o)} \qquad \text{(E-}\textsc{Plus}\text{)}$$

$$(\mathbf{read}(), B, \emptyset, \emptyset, (v; S_i), S_o) \longrightarrow (v, B, \emptyset, \emptyset, S_i, S_o) \qquad \text{(E-}\textsc{Read}\text{)}$$

$$\frac{v \text{ is an integer, } \mathbf{leaf}\text{, or } \mathbf{node}}{(\mathbf{write}\ v, B, H, L, S_i, S_o) \longrightarrow ((), B, H, L, S_i, (S_o; v))} \qquad \text{(E-}\textsc{Write}\text{)}$$

$$(\mathbf{flush}(), B, H, \emptyset, S_i, S_o) \longrightarrow ((), B, \emptyset, \emptyset, S_i, S_o) \qquad \text{(E-}\textsc{Flush1}\text{)}$$

$$(\mathbf{flush}(), B, H, (l; L), ([\![V]\!]; S_i), S_o) \longrightarrow (\mathbf{flush}(), B, \emptyset, L, S_i, S_o) \qquad \text{(E-}\textsc{Flush2}\text{)}$$

$$\frac{x' \text{ is fresh}}{(\mathbf{let}\ x = \mathbf{s2m}()\ \mathbf{in}\ e, B, \emptyset, \emptyset, ([\![V]\!]; S_i), S_o) \longrightarrow ([x \mapsto x']e, \mathit{Alloc}^{\omega}(B, x', V), \emptyset, \emptyset, S_i, S_o)} \qquad \text{(E-}\textsc{StoM}\text{)}$$

$$(\mathbf{let}\ x = \mathbf{s2h}()\ \mathbf{in}\ e, B, \emptyset, \emptyset, S_i, S_o) \longrightarrow ([x \mapsto l]e, B, \emptyset, l, S_i, S_o) \quad (l \text{ is fresh}) \qquad \text{(E-}\textsc{StoH}\text{)}$$

$$(\mathbf{m2s}(x), B, H, L, S_i, S_o) \longrightarrow ((), B, H, L, S_i, (S_o; [\![\mathit{ToTree}(B, x)]\!])) \qquad \text{(E-}\textsc{MtoS}\text{)}$$

$$\frac{x \text{ is fresh}}{(\mathbf{leaf}^{\omega}\ n, B, H, L, S_i, S_o) \longrightarrow (x, \mathit{Alloc}^{\omega}(B, x, \mathbf{leaf}^{\omega}\ n), H, L, S_i, S_o)} \qquad \text{(E-}\textsc{Leaf}\text{)}$$

$$\frac{x \text{ is fresh}}{(\mathbf{node}^{\omega}(x_1, x_2), B, H, L, S_i, S_o) \longrightarrow (x, \mathit{Alloc}^{\omega}(B, x, \mathbf{node}^{\omega}(x_1, x_2)), H, L, S_i, S_o)} \qquad \text{(E-}\textsc{Node}\text{)}$$

$$(\mathbf{case}^1\ \mathbf{leaf}\ \mathbf{of}\ \mathbf{leaf} \Rightarrow e_1 \mid \mathbf{node} \Rightarrow e_2, B, H, L, S_i, S_o) \longrightarrow (e_1, B, H, L, S_i, S_o) \qquad \text{(E-}\textsc{Case1}\text{)}$$

$$(\mathbf{case}^1\ \mathbf{node}\ \mathbf{of}\ \mathbf{leaf} \Rightarrow e_1 \mid \mathbf{node} \Rightarrow e_2, B, H, L, S_i, S_o) \longrightarrow (e_2, B, H, L, S_i, S_o) \qquad \text{(E-}\textsc{Case2}\text{)}$$

$$\frac{B(x) = \mathbf{leaf}^{\omega}\ n}{\begin{array}{c}(\mathbf{case}^{\omega}\ x\ \mathbf{of}\ \mathbf{leaf}\ x_1 \Rightarrow e_1 \mid \mathbf{node}(x_2, x_3) \Rightarrow e_2, B, H, L, S_i, S_o) \longrightarrow \\ ([x_1 \mapsto n]e_1, B, H, L, S_i, S_o)\end{array}} \qquad \text{(E-M}\textsc{Case1}\text{)}$$

$$\frac{B(x) = \mathbf{node}^{\omega}(x'_2, x'_3)}{\begin{array}{c}(\mathbf{case}^{\omega}\ x\ \mathbf{of}\ \mathbf{leaf}\ x_1 \Rightarrow e_1 \mid \mathbf{node}(x_2, x_3) \Rightarrow e_2, B, H, L, S_i, S_o) \longrightarrow \\ ([x_2 \mapsto x'_2, x_3 \mapsto x'_3]e_2, B, H, L, S_i, S_o)\end{array}} \qquad \text{(E-M}\textsc{Case2}\text{)}$$

$$\frac{l \notin \mathit{dom}(H)}{\begin{array}{c}(\mathbf{case}^{\sharp}\ l\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow e_1 \mid \mathbf{node}(x_1, x_2) \Rightarrow e_2, B, H, (l'; L), (\mathbf{leaf}; n; S_i), S_o) \longrightarrow \\ (\mathbf{case}^{\sharp}\ l\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow e_1 \mid \mathbf{node}(x_1, x_2) \Rightarrow e_2, B, H[l' \mapsto \mathbf{leaf}^{\omega}\ n], L, S_i, S_o)\end{array}} \qquad \text{(E-H}\textsc{Case1}\text{)}$$

$$\frac{l \notin \mathit{dom}(H) \qquad l_1 \text{ and } l_2 \text{ are fresh}}{\begin{array}{c}(\mathbf{case}^{\sharp}\ l\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow e_1 \mid \mathbf{node}(x_1, x_2) \Rightarrow e_2, B, H, (l'; L), (\mathbf{node}; S_i), S_o) \longrightarrow \\ (\mathbf{case}^{\sharp}\ l\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow e_1 \mid \mathbf{node}(x_1, x_2) \Rightarrow e_2, B, H[l' \mapsto \mathbf{node}^{\omega}(l_1, l_2)], (l_1; l_2; L), S_i, S_o)\end{array}} \qquad \text{(E-H}\textsc{Case2}\text{)}$$

$$\begin{array}{c}(\mathbf{case}^{\sharp}\ l\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow e_1 \mid \mathbf{node}(x_1, x_2) \Rightarrow e_2, B, H[l \mapsto \mathbf{leaf}^{\omega}\ n], L, S_i, S_o) \longrightarrow \\ ([x \mapsto n]e_1, B, H[l \mapsto \mathbf{leaf}^{\omega}\ n], L, S_i, S_o)\end{array} \qquad \text{(E-H}\textsc{Case3}\text{)}$$

$$\begin{array}{c}(\mathbf{case}^{\sharp}\ l\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow e_1 \mid \mathbf{node}(x_1, x_2) \Rightarrow e_2, B, H[l \mapsto \mathbf{node}^{\omega}(l_1, l_2)], L, S_i, S_o) \longrightarrow \\ ([x_1 \mapsto l_1, x_2 \mapsto l_2]e_2, B, H[l \mapsto \mathbf{node}^{\omega}(l_1, l_2)], L, S_i, S_o)\end{array} \qquad \text{(E-H}\textsc{Case4}\text{)}$$

$$\frac{(e, B, H, L, S_i, S_o) \longrightarrow (e', B', H', L', S'_i, S'_o)}{(E[e], B, H, L, S_i, S_o) \longrightarrow (E[e'], B', H', L', S'_i, S'_o)} \qquad \text{(E-}\textsc{Context}\text{)}$$

Fig. 10 The operational semantics of the target language.

$$\begin{array}{l}
\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{m2s}(M)) = \mathbf{m2s}(\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}(\mathbf{leaf}^{+}\ M) = \mathbf{write}\ \mathbf{leaf}; \mathbf{write}\ \mathcal{A}(M) \\
\mathcal{A}(\mathbf{node}^{+}(M_1, M_2)) = \mathbf{write}\ \mathbf{node}; \mathcal{A}(M_1); \mathcal{A}(M_2) \\
\mathcal{A}(\mathbf{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}(\mathbf{case}^1\ x\ \mathbf{of}\ \mathbf{leaf}\ x_1 \Rightarrow M_1 \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2) = \\
\qquad \mathbf{case}\ \mathbf{flush}(); \mathbf{read}()\ \mathbf{of} \\
\qquad\qquad \mathbf{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}\ \mathbf{leaf}\ x_1 \Rightarrow M_1 \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2) = \\
\qquad \mathbf{case}^{\omega}\ x\ \mathbf{of}\ \mathbf{leaf}\ x_1 \Rightarrow \mathcal{A}(M_1) \mid \mathbf{node}(x_2, x_3) \Rightarrow \mathcal{A}(M_2) \\
\mathcal{A}(\mathbf{case}^{\sharp}\ x\ \mathbf{of}\ \mathbf{leaf}\ x_1 \Rightarrow M_1 \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2) = \\
\qquad \mathbf{case}^{\sharp}\ x\ \mathbf{of}\ \mathbf{leaf}\ x_1 \Rightarrow \mathcal{A}(M_1) \mid \mathbf{node}(x_2, x_3) \Rightarrow \mathcal{A}(M_2)
\end{array}$$

Fig. 11 Translation algorithm.

E-Flush2, E-StoH, E-HCase1, ..., E-HCase4) would require an explanation. E-StoH is the rule for turning an input tree into a hybrid tree. Instead of immediately copying a tree from S_i to H, however, we create only a pointer (l in the rule) to the first tree of S_i, and record the pointer in L. The hybrid tree is actually copied to H only when it is accessed by a case expression, as described in the rules E-HCase1 and E-HCase2. In E-HCase2, only the root node is copied, and the subtrees are kept in S_i, with pointers (l_1 and l_2) to them being added to L. E-Flush1 and E-Flush2 are the rules for discarding the current hybrid tree. E-Flush2 discards a part of the hybrid tree that has not been copied to H.

A well-typed $\mathcal{L}_I$ program can be translated into an equivalent stream-processing program using the algorithm $\mathcal{A}$ defined in **Fig. 11**. The algorithm $\mathcal{A}$ converts output tree constructions into stream output operations and a case analysis for ordered linear trees into stream input operations. Note that an instruction **flush** is inserted before **s2m**, **s2h** and **case**^{1} x **of**. This instruction ensures that hybrid trees are indeed discarded before another ordered linear tree is accessed.

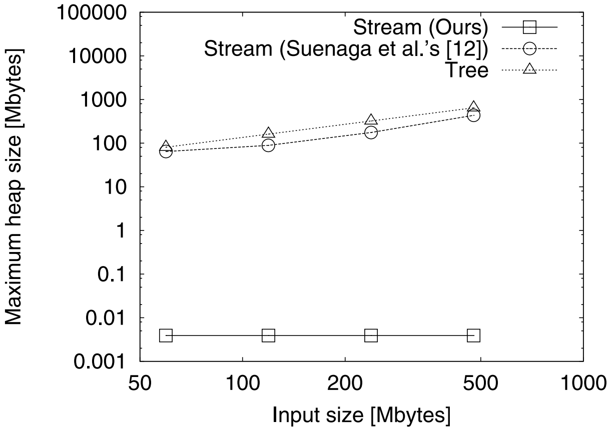


Fig. 12 Memory consumption (ex_leftmost).

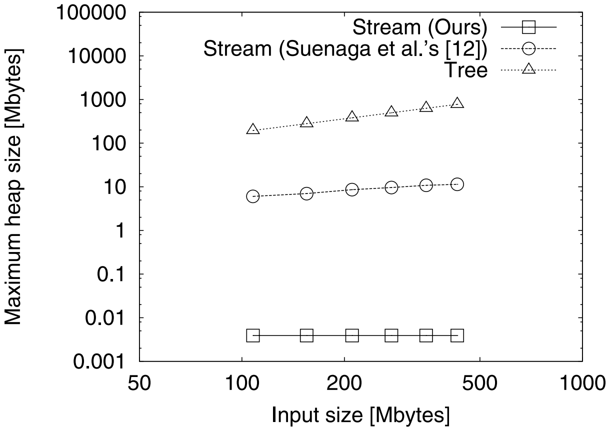


Fig. 13 Memory consumption (ex_bib).

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 $\mathcal{L}_T$ in Objective Caml. The current translator supports only binary trees that have integers or strings in leaves. An extension for dealing with XML documents, as well as the implementation of a translator from the source to the intermediate language are currently under development.

In this experiment, we wrote the following programs in $\mathcal{L}_I$, translated them into $\mathcal{L}_T$ by using our translator and gave automatically-generated well-formed XML documents as input to each $\mathcal{L}_T$ program.

- **ex_leftmost:** a program in Example 1 which takes a tree as an input, and returns (the tree representation of) a list obtained by replacing the left subtree of each node with the sum of the values of the leftmost and second leftmost leaves.
- **ex_bib:** a program which takes a bibliography database and returns a list of title and authors where the title contains a specific word.

Figures 12, **13**, **14** and **15** show the result of the experiment. Figures 12 and 13 compare the maximum memory consumption of the stream-processing programs generated by our new translator with (1) naive tree-processing programs (which construct the whole input tree on memory) and (2) the stream-processing programs generated by the previous framework [3]. Figs. 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.

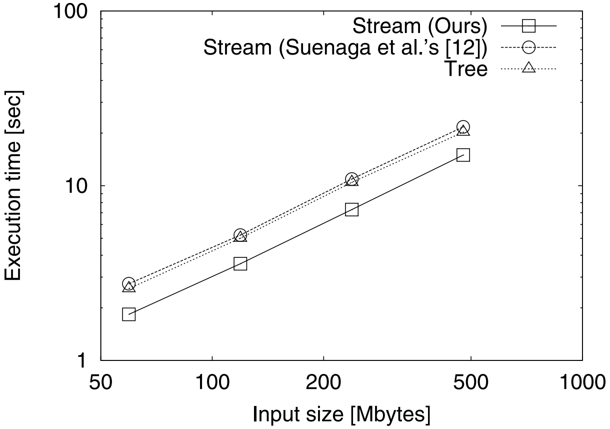


Fig. 14 Execution time (ex_leftmost).

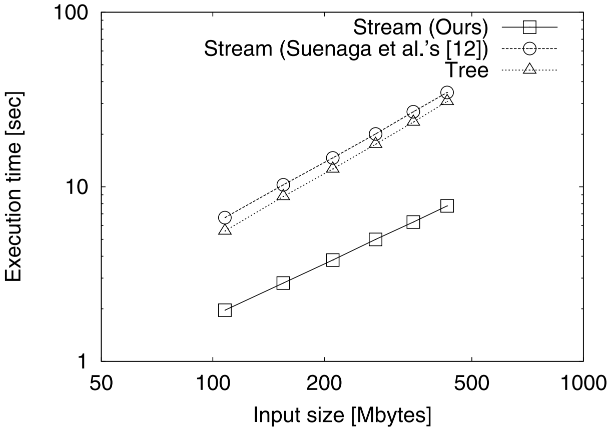


Fig. 15 Execution time (ex_bib).

As shown in the figures, the stream-processing programs generated by the new translator are more efficient than the ones generated by X-P, the translator developed in our previous work[1]. The improvement was mainly gained by the lazy construction of hybrid trees, which avoids copying the unnecessary part of the input on 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[1),2)], there are other approaches to automatic transformation of tree-processing programs into stream-processing programs[4)–8)]. (An exception is Frisch and Nakano's work[9)], which will be addressed later.) In those approaches, the source languages for describing tree-processing programs are more restricted than ordinary programming languages (query language[4)], and attribute grammars[5)–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.

Frisch and Nakano[9)] proposed a framework that achieves stream processing for a Turing-complete language based on term rewriting. The hybrid trees in our approach are somewhat similar to input trees in their approach, in that tree data are incrementally constructed, and discarded when they become unnecessary. In fact, the L component of the run-time state of our target language $\mathcal{L}_T$ is a reminiscent of the parsing stack in Frisch and Nakano's system. The main differences are: (i) The evaluation order of their language is input-driven, while that of our language is call-by-value. As discussed in Section 9 of their paper[9)], there is a trade-off in the choice of an evaluation order, and if necessary, one can often rewrite a program to avoid some of the limitations of a particular evaluation order. An obvious advantage of the call-by-value evaluation is, however, that it is easy to extend the language with side effects. (ii) Our approach supports ordered linear trees (which need no buffering at all) besides hybrid trees, while Frisch and Nakano's approach supports only a single kind of input trees that are similar to our hybrid trees as mentioned above. Instead, they use sophisticated run-time techniques to enable more memory-efficient buffering than our hybrid trees. For example, in our approach, hybrid trees are discarded (by the operation **flush** in $\mathcal{L}_T$) only when all the current hybrid trees become unnecessary, while Frisch and Nakano's system eagerly inspects the run-time state and discards a part of input trees. (iii) Our technique statically decides when and how a tree is buffered and discarded) by using a static type system, while Frisch and Nakano's system decides it at run-time. There are obvious trade-offs: On one hand, in the former approach, the run-time system is simple and easy to implement, and the run-time overhead is kept small. On the other hand, the latter approach is more flexible, potentially enabling better memory usage. Thus, a combination of the static and dynamic approaches may be useful.

Ordered linear type systems have been first studied by Polakow[10)], and later by

Petersen, et al.[11] and ourselves[1),2)]. 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[12]. 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[1),2)] 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.

Acknowledgments This work was supported by Kakenhi 20240001. We would like to thank anonymous referees for useful comments.

References

1) Kodama, K., Suenaga, K. and Kobayashi, N.: Translation of Tree-Processing Programs into Stream-Processing Programs Based on Ordered Linear Type, *Journal of Functional Programming*, Vol.18, No.3, pp.333–371 (2008).

2) Suenaga, K., Kobayashi, N. and Yonezawa, A.: Extension of Type-Based Approach to Generation of Stream-Processing Programs by Automatic Insertion of Buffering Primitives, *International Symposium on Logic-based Program Synthesis and Translformation* (*LOPSTR 2005*), Lecture Notes in Computer Science, Vol.3901, Springer-Verlag, pp.98–114 (2005).

3) Suenaga, K., Sato, S., Sato, R. and Kobayashi, N.: X-P: XML stream processing program generator. http://www.kb.ecei.tohoku.ac.jp/˜suenaga/x-p/

4) Koch, C., Scherzinger, S., Schweikardt, N. and Stegmaier, B.: Schema-based scheduling of event processors and buffer minimization for queries on structured data streams, *Proc. 30th International Conference on Very Large Data Bases* (*VLDB 2004*), VLDB Endowment, pp.228–239 (2004).

5) Nakano, K.: An Implementation Scheme for XML Transformation Languages Through Derivation of Stream Processors, *The Second Asian Symposium on Programming Languages and Systems* (*APLAS 2004*), Lecture Notes in Computer Science, Vol.3302, Springer-Verlag, pp.74–90 (2004).

6) Nakano, K.: Composing Stack-Attributed Tree Transducers, *Theory of Computing Systems*, Vol.44, No.1, pp.1–38 (2009).

7) Nakano, K. and Nishimura, S.: Deriving Event-Based Document Transformers from Tree-Based Specifications, *Electronic Notes in Theoretical Computer Science*, Vol.44, No.2, pp.181–205 (2001).

8) Nishimura, S. and Nakano, K.: XML stream transformer generation through program composition and dependency analysis, *Science of Computer Programming*, Vol.54, No.2-3, pp.257–290 (2005).

9) Frisch, A. and Nakano, K.: Streaming XML Transformation Using Term Rewriting, *ACM SIGPLAN Workshop on Programming Language Technologies for XML* (*PLAN-X 2007*), pp.2–13 (2007).

10) Polakow, J.: Ordered Linear Logic and Applications, PhD Thesis, Carnegie Mellon University (2001). Available as Technical Report CMU-CS-01-152.

11) Petersen, L., Harper, R., Crary, K. and Pfenning, F.: A type theory for memory allocation and data layout, *Proc. 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL 2003*), New York, NY, USA, ACM, pp.172–184 (2003).

12) Lambek, J.: The Mathematics of Sentence Structure, *The American Mathematical Monthly*, Vol.65, No.3, pp.154–170 (1958).

Appendix

A.1 Proof of Theorem 1

Theorem 1 is a corollary of Lemma 1, Theorem 2 (progress), and Theorem 3 (preservation). We first define the type judgment for the environments.

Definition 3. $\Gamma \mid \Psi \mid \Delta \vdash (B, H, S)$ *holds if and only if the following conditions hold:*

- $dom(\Gamma) = dom(B)$.
- $dom(\Psi) = dom(H)$.
- *If* $\Delta = x_1 : \tau_1, \ldots, x_n : \tau_n$ *and* $S = (y_1 \mapsto V_1); \ldots; (y_m \mapsto V_m)$*, then* $n = m$ *and* $x_i = y_i$ *for each* i.
- *For each* $x \in dom(B)$, $\Gamma \mid \emptyset \mid \emptyset \vdash B(x) : \Gamma(x)$.

Lemma 1 (Canonical form). *If* $\Gamma \mid \Psi \mid \Delta \vdash v{:}\tau$*, then exactly one of the following holds.*

(1) $\tau = \mathbf{int}$ *and* $v = n$ *for some* n *and* $\Delta = \emptyset$.

(2) $\tau = \tau_1 \to \tau_2$ *for some* τ_1 *and* τ_2 *and* $v = \mathbf{fix}(f, x, M)$ *for some* f, x *and* M,

and $\Delta = \emptyset$.

(3) $\tau = \mathbf{tree}^\omega$ *and* $v = x \in dom(\Gamma)$ *and* $\Delta = \emptyset$ *for some* x

(4) $\tau = \mathbf{tree}^\sharp$ *and* $v = x \in dom(\Psi)$ *and* $\Delta = \emptyset$ *for some* x

(5) $\tau = \mathbf{tree}^1$ *and* $v = x$ *and* $\Delta = x : \mathbf{tree}^1$ *for some* x

(6) $\tau = \mathbf{tree}^+$ *and* (1) $v = \mathbf{leaf}^+\ n$ *and* $\Delta = \emptyset$ *for some* n*, or* (2) $v = \mathbf{node}^+(V_1, V_2)$ *and* $\Delta = \emptyset$ *for some* V_1 *and* V_2.

Proof. Case analysis on the derivation of $\Gamma \mid \Psi \mid \Delta \vdash v : \tau$. □

Theorem 2 (Progress)**.** *If* $\Gamma \mid \Psi \mid \Delta \vdash M : \tau$ *and* $\Gamma \mid \Psi \mid \Delta \vdash (B, H, S)$*, then one of the following holds.*

(1) M *is a value.*

(2) $(M, B, H, S) \longrightarrow (M', B', H', S')$ *for some* M', B', H', S'.

Proof. By induction on the derivation of $\Gamma \mid \Psi \mid \Delta \vdash M : \tau$. We show only important cases.

- Case T-App: $\Gamma \mid \Psi_1 \mid \Delta_1 \vdash M_1 : \tau_1 \to \tau$ and $\Gamma \mid \Psi_2 \mid \Delta_2 \vdash M_2 : \tau_1$ for some $\Psi_1, \Psi_2, \Delta_1, \Delta_2, M_1, M_2$ and τ_1. From the I.H., M_1 and M_2 satisfy one of the conditions stated in the theorem.
  - If M_1 or M_2 is reducible, then M is also reducible from E-Context.
  - If M_1 and M_2 are values, then, from Lemma 1, $M_1 = \mathbf{fix}(f, x, M_1')$ for some f, x and M_1'. Thus, by applying E-App, M is reducible.
- Case T-HCase: $M = \mathbf{case}^\sharp\ x\ \mathbf{of\ leaf}\ x_1 \Rightarrow M_1 \mid \mathbf{node}(x_2, x_3) \Rightarrow M_2$ and $(x{:}\mathbf{tree}^\sharp) \in \Psi$ for some x, x_1, M_1, x_2, x_3 and M_2. From $\Gamma \mid \Psi \mid \Delta \vdash (B, H, S)$, we have $x \in dom(H)$. Thus, M is reducible from E-HCase1 or E-HCase2.

□

The following lemmas are used to prove Theorem 3.

Lemma 2 (Substitution)**.** *If* $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \mid \Psi \mid \Delta \vdash M : \tau$ *and* $\Gamma \mid \Psi \mid \emptyset \vdash v_i : \tau_i$ *for* $i \in \{1, \ldots, n\}$ *then* $\Gamma \mid \Psi \mid \Delta \vdash [x_1 \mapsto v_1 \ldots x_n \mapsto v_n]M : \tau$.

Proof. By induction on the derivation of $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \mid \Psi \mid \Delta \vdash M : \tau$. □

Lemma 3 (Renaming)**.** *Suppose* $x \notin dom(\Gamma) \cup dom(\Psi) \cup dom(\Delta)$.

- *If* $\Gamma, y : \tau \mid \Psi \mid \Delta \vdash M : \tau$ *then* $\Gamma, x : \tau \mid \Psi \mid \Delta \vdash [y \mapsto x]M : \tau$.
- *If* $\Gamma \mid y : \mathbf{tree}^\sharp, \Psi \mid \Delta \vdash M : \tau$ *then* $\Gamma \mid x : \mathbf{tree}^\sharp, \Psi \mid \Delta \vdash [y \mapsto x]M : \tau$.
- *If* $\Gamma \mid \Psi \mid \Delta_1, y{:}\mathbf{tree}^1, \Delta_2 \vdash M{:}\tau$ *then* $\Gamma \mid \Psi \mid \Delta_1, x{:}\mathbf{tree}^1, \Delta_2 \vdash [y \mapsto x]M{:}\tau$.

Proof. By induction on the derivations of typing judgments. □

Lemma 4 (Weakening)**.** *If* $\Gamma \mid \Psi \mid \Delta \vdash M{:}\tau$ *and* $dom(\Psi') \cap (dom(\Gamma) \cup dom(\Psi) \cup dom(\Delta)) = \emptyset$*, then* $\Gamma \mid \Psi, \Psi' \mid \Delta \vdash M : \tau$

Proof. Induction on the derivation of $\Gamma \mid \Psi \mid \Delta \vdash M : \tau$. □

Theorem 3 (Preservation)**.** *If* $\Gamma \mid \Psi \mid \Delta \vdash M : \tau$ *and* $\Gamma \mid \Psi \mid \Delta \vdash (B, H, S)$ *and* $(M, B, H, S) \longrightarrow (M', B', H', S')$*, then* $\Gamma' \mid \Psi' \mid \Delta' \vdash M' : \tau$ *and* $\Gamma' \mid \Psi' \mid \Delta' \vdash (B', H', S')$*, for some* Γ', Ψ' *and* Δ'.

Proof. By induction on the derivation of $(M, B, H, S) \longrightarrow (M', B', H', S')$. We show only Case E-App. The other cases immediately follow from the induction hypothesis and the renaming lemma.

- Case E-App. By inversion of T-App, $\Gamma \mid \Psi_1 \mid \Delta_1 \vdash \mathbf{fix}(f, x, M_1) : \tau_1 \to \tau$ and $\Gamma \mid \Psi_2 \mid \Delta_2 \vdash v : \tau_1$ hold, where $(\Psi \mid \Delta) = (\Psi_1 \mid \Delta_1); (\Psi_2 \mid \Delta_2)$. The last rule that derives $\Gamma \mid \Psi_1 \mid \Delta_1 \vdash \mathbf{fix}(f, x, M_1) : \tau_1 \to \tau$ has to be T-Fix1, T-Fix2 or T-Fix3. We perform case-analysis on the rule. Note that $\Psi_1 = \emptyset$ and $\Delta_1 = \emptyset$ in every case, so that $\Psi = \Psi_2$ and $\Delta = \Delta_2$.
  - If the last rule is T-Fix1, we have $\Gamma, f : \tau_1 \to \tau, x : \tau_1 \mid \emptyset \mid \emptyset \vdash M_1 : \tau$. From the fact that τ_1 is in the unrestricted type environment, exactly one of (1), (2) or (3) in Lemma 1 holds. In every case $\Delta = \emptyset$, so that we have $\Gamma \mid \emptyset \mid \emptyset \vdash [f \mapsto \mathbf{fix}(f, x, M_1), x \mapsto v]M_1 : \tau$ from Lemma 2. From Lemma 4, we have $\Gamma \mid \Psi \mid \emptyset \vdash M' : \tau$. The statement follows from $B' = B$, $H' = H$ and $S' = S$.
  - If the last rule is T-Fix2, we have $\Gamma, f : \mathbf{tree}^\sharp \to \tau \mid x : \mathbf{tree}^\sharp \mid \emptyset \vdash M_1 : \tau$ and $\Gamma \mid \Psi \mid \Delta \vdash v : \mathbf{tree}^\sharp$. From Lemma 1, we have $v = y \in dom(\Psi)$ and $\Delta = \emptyset$ for some y. Thus, from Lemma 2 and Lemma 3, we have $\Gamma \mid y : \mathbf{tree}^\sharp \mid \emptyset \vdash M' : \tau$. From Lemma 4, we have $\Gamma \mid \Psi \mid \emptyset \vdash M' : \tau$, which finishes up this case.
  - The case T-Fix3 can be proved in a similar way to the case T-Fix2.

□

Proof of Theorem 1. Induction on the definition of $\longrightarrow^*$.

- Suppose that $(M, \emptyset, \emptyset, x \mapsto V) = (M', B', H', S')$. First, note that $\emptyset \mid \emptyset \mid x : \mathbf{tree}^1 \vdash (\emptyset, \emptyset, (x \mapsto V))$ holds. From $\emptyset \mid \emptyset \mid x : \mathbf{tree}^1 \vdash M : \mathbf{tree}^+$ and Theorem 2, M is either a value or $(M, \emptyset, \emptyset, (x \mapsto V))$ is reducible. Suppose M is a value. Then, from Lemma 1, M is a tree value. If $(M, \emptyset, \emptyset, (x \mapsto V))$ is reducible, the statement of the theorem obviously holds.
- Suppose $(M, \emptyset, \emptyset, x \mapsto V) \longrightarrow (M_1, B_1, H_1, S_1) \longrightarrow^* (M', B', H', S')$. Then, from Theorem 3, there exist Γ_1, Ψ_1 and Δ_1 such that $\Gamma_1 \mid \Psi_1 \mid \Delta_1 \vdash M_1 : \mathbf{tree}^+$ and $\Gamma_1 \mid \Psi_1 \mid \Delta_1 \vdash (B_1, H_1, S_1)$. Thus, from I.H., M' is a tree value or (M', B', H', S') is reducible.

□

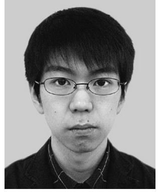
(Received March 1, 2010)
(Accepted November 5, 2010)

(Original version of this article can be found in the Journal of Information Processing Vol.19, pp.74–87.)

Editor's Recommendation

The authors proposed a type-based method for automatic transformation of tree-processing programs into efficient stream-processing ones. The proposed methods produces a program that allows a finer control over memory allocation of the tree structure, and therefore gives an improved memory efficiency in runtime. The improvement is achieved by a refinement of ordered linear types.

(Shinichi Honiden FIT2009 Program Chair)



Ryosuke Sato received his B.S. (in 2008) and M.S. (in 2010) degrees from Tohoku University. He is a Ph.D. student in Graduate School of Information Sciences, Tohoku University. He is interested in program verification.

Kohei Suenaga received his B.Sc. (in 2003), M.Sc. (in 2005) and Ph.D. (2008) degrees in Information Science and Technology from the University of Tokyo. After working for Tohoku University as a post-doctoral researcher and for IBM Research − Tokyo as a researcher, he is now a post-doctoral researcher in Faculdade de Ciências da Universidade de Lisboa. He is interested in program verification based on formal methods. He is a member of IPSJ, ACM and JSSST.

Naoki Kobayashi was born in 1968, and received his B.S., M.S., and D.S. degrees from the University of Tokyo in 1991, 1993 and 1996, respectively. He is a professor in Graduate School of Information Sciences, Tohoku University. His current major research interests are in principles of programming languages. In particular, he is interested in type systems and program verification.