Extension of Type-Based Approach to Generation of Stream-Processing Programs by Automatic Insertion of Buffering Primitives

Kohei Suenaga*, Naoki Kobayashi**, and Akinori Yonezawa***

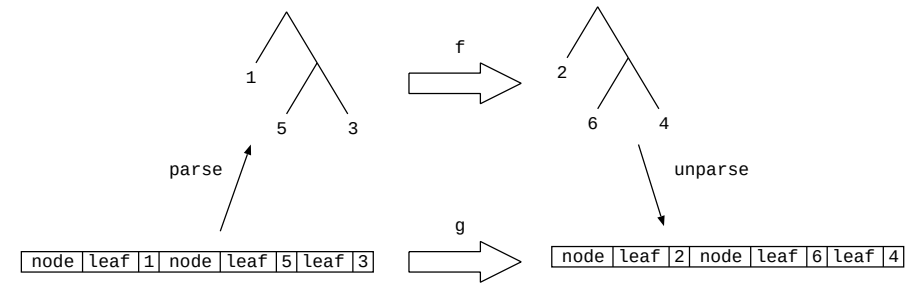
*University of Tokyo, kohei@yl.is.s.u-tokyo.ac.jp
**Tohoku University, koba@kb.ecei.tohoku.ac.jp
***University of Tokyo, yonezawa@yl.is.s.u-tokyo.ac.jp

Abstract. In our previous paper, we have proposed a framework for automatically translating tree-processing programs into stream-processing programs. However, in writing programs that require buffering of input data, a user has to explicitly use *buffering primitives* which copy data from input stream to memory or copy constructed trees from memory to an output stream. Such explicit insertion of buffering primitives is often cumbersome and worsens the readability of the program. We overcome the above-mentioned problems by developing an algorithm which, given any simply-typed tree-processing program, automatically inserts buffering primitives. The resulting program is guaranteed to be well-typed under our previous ordered-linear type system, so that the program can be further transformed into an equivalent stream-processing program using our previous framework.

1 Introduction

There are two ways for processing tree-structured data such as XML [1]: one is to manipulate data using a tree representation (e.g., DOM API [16], XDuce [4, 5], CDuce [15] in the case of XML processing), and the other is to use a stream representation (e.g., SAX, in the case of XML processing). Since large tree-structured data are typically stored in files using the stream representation, the former approach requires that the data be first loaded into memory and converted into the tree representation. On the other hand, the former approach has an advantage that it is easier to read and write programs.

To take the best of both approaches, in our previous paper [7], we have proposed a framework in which a user can write a tree processing program, which is then automatically transformed into an equivalent stream processing program. For example, consider the programs in Figure 1. A user writes the tree-processing program, which takes a binary tree t as an input, and returns the tree whose leaf values are incremented by 1. A system then automatically transforms the program into the stream-processing program, which is more efficient for data stored in the stream representation since there is no need to construct trees on memory. We have implemented a generator of XML stream processing programs



Tree-Processing Program:
fix $(f, t,$ **case** t **of leaf** $x \Rightarrow$ **leaf** $(x+1)$ | **node** $x_1\ x_2 \Rightarrow$ **node** $(f\ x_1)\ (f\ x_2))$
Stream-Processing Program:
fix$(g, t,$ **case read**() **of**
leaf $\Rightarrow$ **let** $x =$ **read**() **in write leaf**; **write**; $(x+1)$
| **node** $\Rightarrow$ **write node**; $g\ ()$; $g\ ()$)

Fig. 1. Tree-processing and stream-processing

based on the framework, and confirmed that the approach works well for certain programs [6].

Our previous framework [7], however, imposes a severe restriction on tree-processing programs. The framework can deal with only programs that access each node of an input tree only once, in the depth-first, left-to-right order. For example, consider the program *swap_deep*′ in Figure 2. It swaps the children of nodes whose depth is more than n. Stream-processing would be effective since the program traverses the input tree mostly in the depth-first, left-to-right order, but our previous framework simply rejects it. In principle, a user can write any tree-processing by explicitly inserting primitives for copying data from an input stream to memory or copying constructed trees from memory to an output stream (both of which are called *buffering primitives* below). For example, one can rewrite the program *swap_deep*′ to the program *swap_deep* by inserting a buffering primitive *s2m*, which copies data from the input stream to memory. Our previous framework can then be applied to obtain a stream-processing program, which constructs only deep sub-trees on memory. Such explicit insertion of buffering primitives is, however, often cumbersome and worsens the readability of the program. Moreover, whether a program conforms to the access order restriction is checked by using a static type system with ordered linear types (inspired by ordered linear logic [12]), so a programmer has to understand the type system to insert buffering primitives in appropriate places.

We overcome the above-mentioned problems by developing an algorithm which, given any simply-typed tree-processing program (without the access order restriction), automatically inserts buffering primitives. The resulting program is guaranteed to be well-typed under our previous ordered-linear type sys-

Ill-typed tree-processing program:

$$
\begin{array}{l}
\mathit{swap\_deep}' \stackrel{\text{def}}{\equiv} \\
\textbf{let } \mathit{swap} = \\
\quad \textbf{fix } (f, t, \textbf{case } t \textbf{ of } \textbf{leaf } x \Rightarrow \textbf{leaf } x \mid \textbf{node } x_1\ x_2 \Rightarrow \textbf{node } (f\ x_2)\ (f\ x_1)) \textbf{ in} \\
\quad \lambda n.\textbf{fix } (\mathit{swap\_deep}, t, \\
\qquad \textbf{if } n = 0 \textbf{ then } \mathit{swap}\ (t) \\
\qquad \textbf{else case } t \textbf{ of} \\
\qquad\quad \textbf{leaf } x \Rightarrow \textbf{leaf } x \\
\qquad \mid \textbf{node } x_1\ x_2 \Rightarrow \textbf{node } (\mathit{swap\_deep}\ (n-1)\ x_1)\ (\mathit{swap\_deep}\ (n-1)\ x_2))
\end{array}
$$

Well-typed tree-processing program:

$$
\begin{array}{l}
\mathit{swap\_deep} \stackrel{\text{def}}{\equiv} \\
\textbf{let } \mathit{swap} = \\
\quad \textbf{fix } (f, t, \\
\quad\quad \textbf{mcase } t \textbf{ of } \textbf{mleaf } x \Rightarrow \textbf{leaf } x \mid \textbf{mnode } x_1\ x_2 \Rightarrow \textbf{node } (f\ x_2)\ (f\ x_1)) \textbf{ in} \\
\quad \lambda n.\textbf{fix } (\mathit{swap\_deep}, t, \\
\qquad \textbf{if } n = 0 \textbf{ then } \mathit{swap}\ (\mathit{s2m}\ t) \\
\qquad \textbf{else case } t \textbf{ of} \\
\qquad\quad \textbf{leaf } x \Rightarrow \textbf{leaf } x \\
\qquad \mid \textbf{node } x_1\ x_2 \Rightarrow \textbf{node } (\mathit{swap\_deep}\ (n-1)\ x_1)\ (\mathit{swap\_deep}\ (n-1)\ x_2))
\end{array}
$$

Resulting stream-processing program:

$$
\begin{array}{l}
\mathit{swap\_deep\_strm} \stackrel{\text{def}}{\equiv} \\
\textbf{let } \mathit{swap} = \\
\quad \textbf{fix } (f, t, \textbf{mcase } t \textbf{ of } \textbf{mleaf } x \Rightarrow \textbf{write leaf}; \textbf{ write } x \\
\qquad\qquad\qquad \mid \textbf{mnode } x_1\ x_2 \Rightarrow \textbf{write node}; \ f\ (); \ f\ ()) \textbf{ in} \\
\quad \lambda n.\textbf{fix } (\mathit{swap\_deep}, t, \\
\qquad \textbf{if } n = 0 \textbf{ then } \mathit{swap}\ (\mathit{s2m}\ t) \\
\qquad \textbf{else case read}() \textbf{ of} \\
\qquad\quad \textbf{leaf} \Rightarrow \textbf{let } x = \textbf{read}() \textbf{ in write leaf}; \textbf{ write } x \\
\qquad \mid \textbf{node} \Rightarrow \textbf{write node}; \ \mathit{swap\_deep}\ (n-1)\ (); \ \mathit{swap\_deep}\ (n-1)\ ())
\end{array}
$$

Fig. 2. A program which swaps children of nodes whose depth is more than n

tem [7], so that the program can be further transformed into an equivalent stream-processing program using our previous framework [7].

For example, the program *swap_deep*′ in Figure 2, which is ill-typed in the type system in [7], is translated into the program *swap_deep* in Figure 2 using the algorithm presented in this paper. As *swap_deep* is well-typed in the type system of [7], it can be translated into a stream-processing program *swap_deep_strm* with the framework in [7].

The rest of the paper is organized as follows. In Section 2, we briefly review our previous framework [7]. Section 3 presents non-deterministic rules for inserting buffering primitives and proves the soundness of the rules. Then, we present a deterministic algorithm for inserting buffering primitives. We discuss related work in Section 6, and conclude in Section 7.

Terms, values and evaluation contexts:

$$
\begin{array}{lll}
M \text{ (terms)} & ::= & i \mid \mathbf{fix}\ (f, x, M) \mid x \mid M_1\ M_2 \mid M_1 + M_2 \\
& & \mid \mathbf{leaf}\ M \mid \mathbf{node}\ M_1\ M_2 \mid \mathbf{mleaf}\ M \mid \mathbf{mnode}\ M_1\ M_2 \\
& & \mid \mathit{s2m} \mid \mathit{m2s} \mid \mathbf{letbuf}\ x = M_1\ \mathbf{in}\ M_2 \\
& & \mid \mathbf{case}\ y\ \mathbf{of\ leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2 \\
& & \mid \mathbf{mcase}\ y\ \mathbf{of\ mleaf}\ x \Rightarrow M_1 \mid \mathbf{mnode}\ x_1\ x_2 \Rightarrow M_2 \\
\tau \text{ (types)} & ::= & \mathbf{Int} \mid \mathbf{Tree}^d \mid \tau_1 \to \tau_2 \\
d \text{ (uses)} & ::= & 1 \mid \omega \mid +
\end{array}
$$

Fig. 3. The syntax of the tree-processing language and types.

2 Language and Type System for Tree-Processing

This section gives an overview of our previous framework for generation of stream-processing programs [7]. The source language is a call-by-value λ-calculus extended with binary trees. The framework can easily be extended to deal with XML [6].

2.1 Language

Figure 3 gives the syntax of the tree-processing language. The operational semantics is summarized in the full version [14].

The meta-variables x and i range over the sets of variables and integers respectively. The first line of M gives standard constructs for the λ-calculus. $\mathbf{fix}\ (f, x, M)$ is a function that takes an argument to x and evaluates M. The whole function is referred to by f in M. We write $\lambda x.M$ for $\mathbf{fix}\ (f, x, M)$ when f is not free in M. We write $\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2$ for $(\lambda x.M_2)\ M_1$. Especially, if M_2 contains no free occurrence of x, we write $M_1; M_2$ for it.

The next line gives two kinds of tree constructors. **leaf** and **node** are constructors for *non-buffered trees*, which are intended to be represented in the stream format, and can be accessed only in a restricted manner. **mleaf** and **mnode** are constructors for *buffered trees*, which are stored in memory and can be accessed in an arbitrary manner.

The third line gives primitives for changing tree representations: The primitive *s2m* converts non-buffered trees to buffered trees, and *m2s* converts buffered trees to non-buffered trees. For a technical reason, we also have a construct $\mathbf{letbuf}\ x = M_1\ \mathbf{in}\ M_2$, which is operationally the same as $(\lambda x.M_2)M_1$.

The last two lines of the definition of terms gives destructors for the two versions of trees.

2.2 Type System

As mentioned in Section 1, we use an ordered linear type system to ensure that buffered trees are accessed in the appropriate order (i.e., the left-to-right, depth-first order).

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

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

$$\frac{\Gamma \mid \Delta \vdash M : \mathbf{Int}}{\Gamma \mid \Delta \vdash \mathbf{mleaf}\ M : \mathbf{Tree}^{\omega}} \quad \text{(T-MLeaf)}$$

$$\frac{\Gamma \mid \Delta_1 \vdash M_1 : \mathbf{Tree}^{\omega} \qquad \Gamma \mid \Delta_2 \vdash M_2 : \mathbf{Tree}^{\omega}}{\Gamma \mid \Delta_1, \Delta_2 \vdash \mathbf{mnode}\ M_1\ M_2 : \mathbf{Tree}^{\omega}} \quad \text{(T-MNode)}$$

$$\frac{\begin{array}{c}\Gamma, x : \mathbf{Int} \mid \Delta \vdash M_1 : \tau \\ \Gamma \mid x_1 : \mathbf{Tree^1}, x_2 : \mathbf{Tree^1}, \Delta \vdash M_2 : \tau\end{array}}{\Gamma \mid y : \mathbf{Tree^1}, \Delta \vdash \mathbf{case}\ y\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2 : \tau} \quad \text{(T-Case)}$$

$$\frac{\begin{array}{c}\Gamma, y : \mathbf{Tree}^{\omega}, x : \mathbf{Int} \mid \Delta \vdash M_1 : \tau \\ \Gamma, y : \mathbf{Tree}^{\omega}, x_1 : \mathbf{Tree}^{\omega}, x_2 : \mathbf{Tree}^{\omega} \mid \Delta \vdash M_2 : \tau\end{array}}{\Gamma \mid \Delta \vdash \mathbf{mcase}\ y\ \mathbf{of}\ \mathbf{mleaf}\ x \Rightarrow M_1 \mid \mathbf{mnode}\ x_1\ x_2 \Rightarrow M_2 : \tau} \quad \text{(T-MCase)}$$

Fig. 4. A part of typing rules of $\Gamma \mid \Delta \vdash M : \tau$

$$\begin{array}{l}\mathcal{A}(\mathbf{leaf}\ M) = \mathbf{write}(\mathbf{leaf}); \mathbf{write}(\mathcal{A}(M)) \\ \mathcal{A}(\mathbf{node}\ M_1\ M_2) = \mathbf{write}(\mathbf{node}); \mathcal{A}(M_1); \mathcal{A}(M_2) \\ \mathcal{A}(\mathbf{case}\ y\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2) = \\ \qquad \mathbf{case}\ \mathbf{read}()\ \mathbf{of}\ \mathbf{leaf} \Rightarrow \mathbf{let}\ x = \mathbf{read}()\ \mathbf{in}\ \mathcal{A}(M_1) \\ \qquad\qquad\qquad\qquad \mid \mathbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}(M_2)\end{array}$$

Fig. 5. Translation algorithm

The syntax of types is given in Figure 3. As usual, **Int** is the type of integers and $\tau_1 \to \tau_2$ is the type of functions from τ_1 to τ_2. We have three kinds of tree types. $\mathbf{Tree}^{\omega}$ is the type of buffered trees. $\mathbf{Tree^1}$ and $\mathbf{Tree}^{+}$ are the types of input trees and output trees respectively.

A type judgment of our type system is $\Gamma \mid \Delta \vdash M : \tau$. Here, Γ is a usual type environment, which is a mapping from a finite set of variables to types. We, however, impose a restriction that the codomain of Γ does not contain $\mathbf{Tree^1}$ or $\mathbf{Tree}^{+}$. Δ is an *ordered linear type environment*, which is a sequence of bindings $x_1 : \mathbf{Tree^1}, \ldots, x_n : \mathbf{Tree^1}$ where $x_1 \cdots x_n$ are different from each other. That environment specifies not only that $x_1, \ldots, x_n$ are bound to input trees, but also that each of $x_1, \ldots, x_n$ must be accessed exactly once in this order and that each of the subtrees bound to $x_1, \ldots, x_n$ must be accessed in the left-to-right, depth-first order.

Figure 4 gives key typing rules. For the full rules, see the full version [14].

2.3 Translation Algorithm

If a program is well-typed in the type system presented above, the program can be translated into an equivalent stream-processing program using a straightforward algorithm. Figure 5 shows the highlight of the algorithm $\mathcal{A}$, which converts tree constructors into stream output operations, and tree destructors into stream input operations. For other term constructors, $\mathcal{A}$ simply works as a homomorphism; For example, $\mathcal{A}(M_1 + M_2) = \mathcal{A}(M_1) + \mathcal{A}(M_2)$. The program *swap_deep_strm* in Figure 2 is obtained from *swap_deep* by using $\mathcal{A}$.

The definition of stream-processing language is in our previous paper [7]. We have proved that the algorithm preserves the semantics of programs.

3 Non-deterministic Specification for Automatic Insertion of Buffering Primitives

Now we discuss a method for automatically inserting *s2m* and *m2s*. Let us write $\Gamma \vdash_{\lambda\to} M : \tau$ for the type judgment for the usual *simply-typed λ-calculus* (see the full version [14]). The goal is to transform any program M such that $\emptyset \vdash_{\lambda\to} M : \mathbf{Tree} \to \mathbf{Tree}$ into an equivalent program M' such that $\emptyset \mid \emptyset \vdash M' : \mathbf{Tree}^{\mathbf{1}} \to \mathbf{Tree}^{+}$, by inserting *s2m* and *m2s* into M.

We first define correct transformations in a declarative and non-deterministic manner. We introduce a new judgment $\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \tau$. The judgment means that (1) M and M' are equivalent if we ignore the representation of trees, and (2) $\Gamma \mid \Delta \vdash M' : \tau$ holds.

Definition 1. *$\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \tau$ is the least relation that satisfies the rules in Figure 6.*

For example, the rule Tr-StreamToMem says that to transform M under the assumption that x is an input tree, we can first insert the conversion $s2m(x)$, and then transform M under the assumption that x is a buffered tree.

Note that the rules are non-deterministic in the sense that there may be more than one valid transformations for each source program M. We present an algorithm that choose one from possible translations in the next section.

The following theorem guarantees the soundness of the judgment:

Theorem 1 (Soundness of $\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \tau$). *If $\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \tau$ holds, then $\Gamma \mid \Delta \vdash M' : \tau$ and $M \equiv erase(M')$.*

Here, $erase(M')$ is the term obtained by removing *s2m* and *m2s*, and replacing constructors and destructors for non-buffered trees with those for buffered trees. The first property of the lemma means that the result of the translation is well-typed (so that our previous framework can be applied to generate a stream-processing program). The second property states that the semantics of the program is preserved by the transformation.

The following lemma guarantees that there is at least one valid transformation for any simply-typed program.

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

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

$$\Gamma, x : \tau \mid \emptyset \vdash x \rightsquigarrow x : \tau \qquad \text{(Tr-Var1)}$$

$$\Gamma \mid x : \mathbf{Tree^1} \vdash x \rightsquigarrow x : \mathbf{Tree^1} \qquad \text{(Tr-Var2)}$$

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

$$\frac{f : \mathbf{Tree^1} \to \tau, \Gamma \mid x : \mathbf{Tree^1} \vdash M \rightsquigarrow M' : \tau}{\Gamma \mid \emptyset \vdash \mathbf{fix}\ (f, x, M) \rightsquigarrow \mathbf{fix}\ (f, x, M') : \mathbf{Tree^1} \to \tau} \qquad \text{(Tr-Fix2)}$$

$$\frac{\Gamma \mid \Delta_1 \vdash M_1 \rightsquigarrow M_1' : \tau' \to \tau \qquad \Gamma \mid \Delta_2 \vdash M_2 \rightsquigarrow M_2' : \tau'}{\Gamma \mid \Delta_1, \Delta_2 \vdash M_1\ M_2 \rightsquigarrow M_1'\ M_2' : \tau} \qquad \text{(Tr-App)}$$

$$\frac{\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \mathbf{Int}}{\Gamma \mid \Delta \vdash \mathbf{leaf}\ M \rightsquigarrow \mathbf{leaf}\ M' : \mathbf{Tree}^+} \qquad \text{(Tr-Leaf1)}$$

$$\frac{\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \mathbf{Int}}{\Gamma \mid \Delta \vdash \mathbf{leaf}\ M \rightsquigarrow \mathbf{mleaf}\ M' : \mathbf{Tree}^\omega} \qquad \text{(Tr-Leaf2)}$$

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

$$\frac{\Gamma \mid \Delta_1 \vdash M_1 \rightsquigarrow M_1' : \mathbf{Tree}^\omega \qquad \Gamma \mid \Delta_2 \vdash M_2 \rightsquigarrow M_2' : \mathbf{Tree}^\omega}{\Gamma \mid \Delta_1, \Delta_2 \vdash \mathbf{node}\ M_1\ M_2 \rightsquigarrow \mathbf{mnode}\ M_1'\ M_2' : \mathbf{Tree}^\omega} \qquad \text{(Tr-Node2)}$$

$$\frac{\Gamma, x : \mathbf{Int} \mid \Delta \vdash M_1 \rightsquigarrow M_1' : \tau \qquad \Gamma \mid x_1 : \mathbf{Tree^1}, x_2 : \mathbf{Tree^1}, \Delta \vdash M_2 \rightsquigarrow M_2' : \tau}{\begin{array}{c}\Gamma \mid y : \mathbf{Tree^1}, \Delta \vdash \mathbf{case}\ y\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2 \\ \rightsquigarrow \mathbf{case}\ y\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1' \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2' : \tau\end{array}} \qquad \text{(Tr-Case1)}$$

$$\frac{\begin{array}{c}\Gamma, y : \mathbf{Tree}^\omega, x : \mathbf{Int} \mid \Delta \vdash M_1 \rightsquigarrow M_1' : \tau \\ \Gamma, y : \mathbf{Tree}^\omega, x_1 : \mathbf{Tree}^\omega, x_2 : \mathbf{Tree}^\omega \mid \Delta \vdash M_2 \rightsquigarrow M_2' : \tau\end{array}}{\begin{array}{c}\Gamma, y : \mathbf{Tree}^\omega \mid \Delta \vdash \mathbf{case}\ y\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2 \\ \rightsquigarrow \mathbf{mcase}\ y\ \mathbf{of}\ \mathbf{mleaf}\ x \Rightarrow M_1' \mid \mathbf{mnode}\ x_1\ x_2 \Rightarrow M_2' : \tau\end{array}} \qquad \text{(Tr-Case2)}$$

$$\frac{\Gamma, x : \mathbf{Tree}^\omega \mid \Delta \vdash M \rightsquigarrow M' : \tau}{\Gamma \mid x : \mathbf{Tree^1}, \Delta \vdash M \rightsquigarrow \mathbf{letbuf}\ x = s2m(x)\ \mathbf{in}\ M' : \tau} \qquad \text{(Tr-StreamToMem)}$$

$$\frac{\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \mathbf{Tree}^\omega}{\Gamma \mid \Delta \vdash M \rightsquigarrow m2s(M') : \mathbf{Tree}^+} \qquad \text{(Tr-MemToStream)}$$

Fig. 6. Rules for the judgment $\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \tau$

Lemma 1. *If* $\Gamma' \vdash_{\lambda\to} M : \tau$ *then there exist* Γ, Δ, M' *and* τ' *such that* $\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \tau'$ *and* $\Gamma' = \mathit{eraseuse}(\Gamma \cup \Delta)$ *and* $\tau = \mathit{eraseuse}(\tau')$.

Here, $\mathit{eraseuse}(\cdot)$ removes uses $(+, 1, \omega)$ from types.

We can easily check that the relation $\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \tau$ contains a transformation that is optimal (in the sense that the resulting program copies as few trees as possible to memory) among those preserving typing and the structure of the source program. To formally state that property, let us write $M \rightsquigarrow M'$ if M' is obtained from M by inserting **letbuf** $x = \mathit{s2m}(x)$ **in** and *m2s* and/or replacing some occurrences of **leaf**, **node**, and **case** with **mleaf**, **mnode**, and **mcase**. The following theorem states that *any* transformation that performs only such replacement and preserves types can be obtained by the transformation rules in Section 3, so that an optimal transformation can also be obtained.

Theorem 2 (Completeness of $\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \tau$**).** *If* $\Gamma \vdash_{\lambda\to} M : \mathbf{Tree} \to \mathbf{Tree}$ *and* $\Gamma \mid \emptyset \vdash M' : \mathbf{Tree^1} \to \mathbf{Tree}^+$ *and* $M \rightsquigarrow M'$*, then* $\Gamma \mid \emptyset \vdash M \rightsquigarrow M' : \mathbf{Tree^1} \to \mathbf{Tree}^+$.

Note that if we allow more aggressive transformation, we may obtain a more efficient program. For example, consider the program

$$\begin{array}{l}\mathbf{fix}\ (f, t, \mathbf{case}\ t\ \mathbf{of}\ \ \mathbf{leaf}\ x \Rightarrow \mathbf{leaf}\ x \\ \qquad\qquad\qquad\qquad \mid \mathbf{node}\ x_1\ x_2 \Rightarrow \mathbf{node}\ (\mathbf{node}\ (f\ x_1)\ (f\ x_1))\ (f\ x_2)).\end{array}$$

If we allow code duplication, we would have the following program:

$$\begin{array}{l}\mathbf{let}\ g = \mathbf{fix}(f, t, \mathbf{mcase}\ t\ \mathbf{of}\ \mathbf{mleaf}\ x \Rightarrow \mathbf{leaf}\ x \\ \qquad\qquad\qquad\qquad \mid \mathbf{mnode}\ x_1\ x_2 \Rightarrow \mathbf{node}\ (\mathbf{node}\ (f\ x_1)\ (f\ x_1))\ (f\ x_2)\ \mathbf{in} \\ \quad \mathbf{fix}(f, t, \mathbf{case}\ t\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow \mathbf{leaf}\ x \\ \qquad\qquad \mid \mathbf{node}\ x_1\ x_2 \Rightarrow \\ \qquad\qquad\quad \mathbf{node}\ (\mathbf{letbuf}\ x_1 = \mathit{s2m}(x_1)\ \mathbf{in}\ \mathbf{node}\ (g\ x_1)\ (g\ x_1))\ (f\ x_2)))\end{array}$$

The program above does not buffer x_2, while any programs derived by $\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \tau$ buffers x because f must have type $\mathbf{Tree}^\omega \to \mathbf{Tree}^+$. It is one of our future work to deal with such transformation.

4 Automatic Insertion Algorithm

The transformation rules presented in the previous section are non-deterministic in the sense that there may be more than one possible M' and τ that satisfy $\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \tau$. We next present an algorithm for choosing one among those possibilities.

The algorithm consists of two sub-algorithms $\mathcal{I}$ and $\mathcal{P}$. Given a program of type $\mathbf{Tree} \to \mathbf{Tree}$, $\mathcal{I}$ inserts *s2m* and generates an intermediate program of type $\mathbf{Tree^1} \to \mathbf{Tree}^\omega$. $\mathcal{P}$ takes the intermediate program as an input, inserts *m2s*, and generates a program of type $\mathbf{Tree^1} \to \mathbf{Tree}^+$.

We focus on the algorithm $\mathcal{I}$ below, since $\mathcal{P}$ is fairly straightforward. $\mathcal{P}$ is briefly discussed at the end of this section.

4.1 Algorithm $\mathcal{I}$

We first give an overview of the algorithm $\mathcal{I}$. We shall introduce a new form of transformation judgment $\Theta \vdash M \rightsquigarrow M' : \tau$. Θ, called a *semi-ordered type environment*, is a combination of a type environment Γ and Δ. The rules for $\Theta \vdash M \rightsquigarrow M' : \tau$ is more deterministic than $\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \tau$: In fact, there is only one transformation rule for each syntactic form of M. Using the new transformation rules, we can construct an algorithm $\mathcal{I}_1$, which, given Θ, M, and τ which may contain use variables to denote unknown uses, outputs M' and C, where C is a set of constraints on the use variables such that $\theta\Theta \vdash M \rightsquigarrow \theta M' : \theta\tau$ holds if and only if the substitution θ satisfies C. Using $\mathcal{I}_1$, the algorithm $\mathcal{I}$ works as follows.

$$\begin{array}{l}\mathcal{I}(M) = \text{let } (M', C) = \mathcal{I}_1(\emptyset, M, \mathbf{Tree}^{\mathbf{1}} \to \mathbf{Tree}^{\omega}) \text{ in} \\ \qquad\quad \text{let } \theta = \mathit{solve}(C) \text{ in} \\ \qquad\quad \theta M'\end{array}$$

Now let us look at the construction of $\mathcal{I}_1$ more closely. We construct $\mathcal{I}_1$ in three steps. First, we introduce a judgment $\Theta \vdash_I M \rightsquigarrow M' : \tau$ by combining Γ and Δ of $\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \tau$. Then, we obtain $\Theta \vdash M \rightsquigarrow M' : \tau$ by deriving syntax-directed rules from $\Theta \vdash_I M \rightsquigarrow M' : \tau$. Finally, we derive $\mathcal{I}_1$ from $\Theta \vdash M \rightsquigarrow M' : \tau$.

We first define semi-ordered type environments. The semi-ordered type environment is necessary since at the time of running $\mathcal{I}_1$, we cannot tell which variable should be put into an ordered linear type environment Γ and which should be put into an ordinary type environment Δ.

Definition 2. *The use of a type τ, written $|\tau|$, is defined by:*

$$|\mathbf{Int}| = \omega \qquad |\tau_1 \to \tau_2| = \omega \qquad |\mathbf{Tree}^d| = d$$

Below, we use the total order $\geq$ on uses, defined by $\omega \geq 1$.

Definition 3 (Semi-ordered type environment). *A* semi-ordered type environment*, represented by Θ, is a sequence $x_1 : \tau_1, \ldots, x_n : \tau_n$ where each x_i is distinct from each other and $|\tau_i| \geq |\tau_j|$ whenever $i \leq j$. We write $x \gtrdot_\Theta y$ if x occurs before y in Θ. We write $|\Theta| \geq d$ for $\forall x \in \mathit{dom}(\Theta)$. $|\Theta(x)| \geq d$. $|\Theta| \leq d$ is defined in the same way.*

In the definition of $\Theta \vdash M \rightsquigarrow M' : \tau$, we use two predicates, $\Theta_1 \succeq \Theta_2$ and $\mathit{merge}(\Theta, \Theta_1, \Theta_2)$. $\Theta_1 \succeq \Theta_2$ means that Θ_1 is obtained by replacing some of $\mathbf{Tree}^{\mathbf{1}}$ in Θ_2 with $\mathbf{Tree}^{\omega}$.

Definition 4. *We write $\tau_1 \succeq \tau_2$ when either $\tau_1 = \tau_2$ or $\tau_1 = \mathbf{Tree}^{\omega}$ and $\tau_2 = \mathbf{Tree}^{\mathbf{1}}$. The relation is pointwise extended to that on semi-ordered environment; $x_1 : \tau_1, \ldots, x_n : \tau_n \succeq x_1 : \tau'_1, \ldots, x_n : \tau'_n$ iff $\tau_i \succeq \tau'_i$ for every $i \in \{1, \ldots, n\}$*

Intuitively, $\mathit{merge}(\Theta, \Theta_1, \Theta_2)$ defined below means that if variables can be accessed according to Θ, then they can be first accessed according to Θ_1 and

$$coerce^{\Theta \Rightarrow \Theta}(M) = M$$
$$coerce^{(\Theta_1, x:\mathbf{Tree^1}, \Theta_2) \Rightarrow (\Theta_1, x:\mathbf{Tree}^\omega, \Theta_2')}(M) = (\mathbf{letbuf}\ x = s2m(x)\ \mathbf{in}\ coerce^{\Theta_2 \Rightarrow \Theta_2'}(M))$$

Fig. 7. Definition of $coerce^{\Theta \Rightarrow \Theta'}()$.

$$\frac{\tau = \Theta(x) \qquad \forall y \in dom(\Theta) \backslash \{x\}.|\Theta(y)| \geq \omega}{\Theta \vdash_I x \rightsquigarrow x : \tau} \quad (\text{Tr-Var'})$$

$$\frac{f : \tau_1 \to \tau_2, \Theta, x : \tau_1 \vdash_I M \rightsquigarrow M' : \tau_2 \qquad \forall y \in dom(\Theta).|\Theta(y)| \geq \omega}{\Theta \vdash_I \mathbf{fix}\ (f, x, M) \rightsquigarrow \mathbf{fix}\ (f, x, M') : \tau_1 \to \tau_2} \quad (\text{Tr-Fix'})$$

$$\frac{\Theta_1 \vdash_I M_1 \rightsquigarrow M_1' : \mathbf{Tree}^\omega \qquad \Theta_2 \vdash_I M_2 \rightsquigarrow M_2' : \mathbf{Tree}^\omega \qquad merge(\Theta, \Theta_1, \Theta_2)}{\Theta \vdash_I \mathbf{node}\ M_1\ M_2 \rightsquigarrow \mathbf{mnode}\ M_1'\ M_2' : \mathbf{Tree}^\omega} \quad (\text{Tr-Node'})$$

$$\frac{\Theta' \vdash_I M \rightsquigarrow M' : \tau \qquad \Theta' \succeq \Theta}{\Theta \vdash_I M \rightsquigarrow coerce^{\Theta \Rightarrow \Theta'}(M') : \tau} \quad (\text{Tr-StreamToMem'})$$

Fig. 8. A part of rules for $\Theta \vdash_I M \rightsquigarrow M' : \tau$

then according to Θ_2. For example, if $\Theta = x : \mathbf{Tree}^\omega, y : \mathbf{Tree^1}, z : \mathbf{Tree^1}$, then $merge(\Theta, y : \mathbf{Tree^1}, (x : \mathbf{Tree}^\omega, z : \mathbf{Tree^1}))$ holds, but $merge(\Theta, z : \mathbf{Tree^1}, (x : \mathbf{Tree}^\omega, y : \mathbf{Tree^1}))$ does not, since the latter violates the condition that y should be read first.

Definition 5 (Merge of semi-ordered type environments). *Θ is a* merge *of Θ_1 and Θ_2, represented by $merge(\Theta, \Theta_1, \Theta_2)$, if and only if the following properties are satisfied:*

(1) $dom(\Theta_1) \cup dom(\Theta_2) \subseteq dom(\Theta)$ *and* $\Theta_1(x) = \Theta(x)$ *for all* $x \in dom(\Theta_1)$ *and* $\Theta_2(y) = \Theta(y)$ *for all* $y \in dom(\Theta_2)$
(2) $x \succcurlyeq_{\Theta_1} y \implies x \succcurlyeq_\Theta y$ *and* $x \succcurlyeq_{\Theta_2} y \implies x \succcurlyeq_\Theta y$
(3) $x \in dom(\Theta) \backslash (dom(\Theta_1) \cup dom(\Theta_2)) \implies |\Theta(x)| \geq \omega$
(4) $x \in dom(\Theta_1) \cap dom(\Theta_2) \implies |\Theta(x)| \geq \omega$
(5) If $y \in dom(\Theta_1)$, $x \in dom(\Theta_2)$, *and* $x \succcurlyeq_\Theta y$, *then* $|\Theta(x)| \geq \omega$.

By the well-formedness condition of semi-ordered type environments, Θ_1 and Θ_2 can be decomposed into Γ_1, Δ_1 and Γ_2, Δ_2, where Γ_i is a sequence of bindings on types of use ω and Δ_i is a linear type environment. Thus, the conditions of $merge(\Theta, \Theta_1, \Theta_2)$ above essentially mean that Θ is of the form $\Gamma, \Delta_1, \Delta_2$ where Γ is obtained by merging Γ_1 and Γ_2 and adding extra bindings on types of use ω.

Figure 8 shows a part of rules for $\Theta \vdash_I M \rightsquigarrow M' : \tau$. The definition of $coerce^{\Theta \Rightarrow \Theta'}(M)$, which is used in the rule Tr-StreamToMem', is given in Figure 7. It inserts *s2m* for each x such that $\Theta(x) = \mathbf{Tree^1}$ and $\Theta'(x) = \mathbf{Tree}^\omega$.

$$\frac{\forall y \in dom(\Theta')\backslash\{x\}.\ |\Theta'(y)| \geq \omega \qquad \Theta' \succeq \Theta \qquad \tau = \Theta'(x)}{\Theta \vdash x \rightsquigarrow coerce^{\Theta \Rightarrow \Theta'}(x) : \tau} \text{ (Tr-SD-Var)}$$

$$\frac{\forall x \in dom(\Theta').\ |\Theta'(x)| \geq \omega \qquad \Theta' \succeq \Theta}{\Theta \vdash i \rightsquigarrow coerce^{\Theta \Rightarrow \Theta'}(i) : \mathbf{Int}} \text{ (Tr-SD-Int)}$$

$$\frac{\begin{array}{c}\Theta_1' \vdash M_1 \rightsquigarrow M_1' : \mathbf{Int} \qquad \Theta_2' \vdash M_2 \rightsquigarrow M_2' : \mathbf{Int} \\ \Theta_1' \succeq \Theta_1 \qquad \Theta_2' \succeq \Theta_2 \qquad merge(\Theta, \Theta_1, \Theta_2)\end{array}}{\Theta \vdash M_1 + M_2 \rightsquigarrow coerce^{\Theta_1 \Rightarrow \Theta_1'}(M_1') + coerce^{\Theta_2 \Rightarrow \Theta_2'}(M_2') : \mathbf{Int}} \text{ (Tr-SD-Plus)}$$

$$\frac{\begin{array}{c}f : \tau_1 \to \tau_2, \Theta', x : \tau_1 \vdash M \rightsquigarrow M' : \tau_2 \\ f : \tau_1 \to \tau_2, \Theta', x : \tau_1 \succeq f : \tau_1 \to \tau_2, \Theta, x : \tau_1' \qquad \forall y \in dom(\Theta).\ |\Theta(y)| \geq \omega\end{array}}{\Theta \vdash \mathbf{fix}\ (f, x, M) \rightsquigarrow \mathbf{fix}\ (f, x, coerce^{(f:\tau_1\to\tau_2, \Theta, x:\tau_1) \Rightarrow (f:\tau_1\to\tau_2, \Theta', x:\tau_1')}(M')) : \tau_1 \to \tau_2} \text{ (Tr-SD-Fix)}$$

$$\frac{\begin{array}{c}\Theta_1' \vdash M_1 \rightsquigarrow M_1' : \tau_1 \to \tau_2 \qquad \Theta_2' \vdash M_2 \rightsquigarrow M_2' : \tau_1 \\ \Theta_1' \succeq \Theta_1 \qquad \Theta_2' \succeq \Theta_2 \qquad merge(\Theta, \Theta_1, \Theta_2)\end{array}}{\Theta \vdash M_1\ M_2 \rightsquigarrow coerce^{\Theta_1 \Rightarrow \Theta_1'}(M_1')\ coerce^{\Theta_2 \Rightarrow \Theta_2'}(M_2') : \tau_2} \text{ (Tr-SD-App)}$$

$$\frac{\Theta' \vdash M \rightsquigarrow M' : \mathbf{Int} \qquad \Theta' \succeq \Theta}{\Theta \vdash \mathbf{leaf}\ M \rightsquigarrow \mathbf{mleaf}\ coerce^{\Theta \Rightarrow \Theta'}(M') : \mathbf{Tree}^{\omega}} \text{ (Tr-SD-Leaf)}$$

$$\frac{\begin{array}{c}\Theta_1' \vdash M_1 \rightsquigarrow M_1' : \mathbf{Tree}^{\omega} \qquad \Theta_2' \vdash M_2 \rightsquigarrow M_2' : \mathbf{Tree}^{\omega} \\ \Theta_1' \succeq \Theta_1 \qquad \Theta_2' \succeq \Theta_2 \qquad merge(\Theta, \Theta_1, \Theta_2)\end{array}}{\Theta \vdash \mathbf{node}\ M_1\ M_2 \rightsquigarrow \mathbf{mnode}\ coerce^{\Theta_1 \Rightarrow \Theta_1'}(M_1')\ coerce^{\Theta_2 \Rightarrow \Theta_2'}(M_2') : \mathbf{Tree}^{\omega}} \text{ (Tr-SD-Node)}$$

$$\frac{\begin{array}{c}\Theta_1' \vdash y \rightsquigarrow y' : \mathbf{Tree}^{d} \qquad \Theta_2' \vdash M_1 \rightsquigarrow M_1' : \tau \qquad \Theta_3' \vdash M_2 \rightsquigarrow M_2' : \tau \\ \Theta_1' \succeq \Theta_1 \qquad \Theta_2' \succeq x : \mathbf{Int}, \Theta_{2L}, \Theta_{2R} \qquad \Theta_3' \succeq \Theta_{2L}, x_1 : \mathbf{Tree}^{d}, x_2 : \mathbf{Tree}^{d}, \Theta_{2R} \\ merge(\Theta, \Theta_1, (\Theta_{2L}, \Theta_{2R})) \qquad M_1'' = coerce^{(\Theta_{2L}, \Theta_{2R}) \Rightarrow (\Theta_2' \backslash \{x:\mathbf{Int}\})}(M_1') \\ M_2'' = coerce^{(\Theta_{2L}, x_1:\mathbf{Tree}^{d}, x_2:\mathbf{Tree}^{d}, \Theta_{2R}) \Rightarrow \Theta_3'}(M_2')\end{array}}{\Theta \vdash \begin{array}{l}\mathbf{case}\ y\ \mathbf{of} \\ \quad \mathbf{leaf}\ x \Rightarrow M_1 \\ \quad \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2\end{array} \rightsquigarrow \begin{array}{l}\mathbf{case}\ y\ \mathbf{of} \\ \quad \mathbf{leaf}\ x \Rightarrow M_1'' \\ \quad \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2''\end{array} : \tau} \text{ (Tr-SD-Case)}$$

Fig. 9. Typing rules for the judgment $\Theta \vdash M \rightsquigarrow M' : \tau$.

Note that $coerce^{\Theta \Rightarrow \Theta'}(\cdot)$ is an operation on terms, so that it is reduced in the program transformation phase (when Θ and Θ' have been completely determined), not when the program is executed.

Next, we introduce a judgment $\Theta \vdash M \rightsquigarrow M' : \tau$.

Definition 6. *The relation $\Theta \vdash M \rightsquigarrow M' : \tau$ is the least relation closed under the rules in Figure 9.*

The rules in Figure 9 are syntax-directed version of the rules in Figure 8. For example, Tr-SD-Node corresponds to applications of the rule Tr-StreamToMem', followed by an application of Tr-Node'.

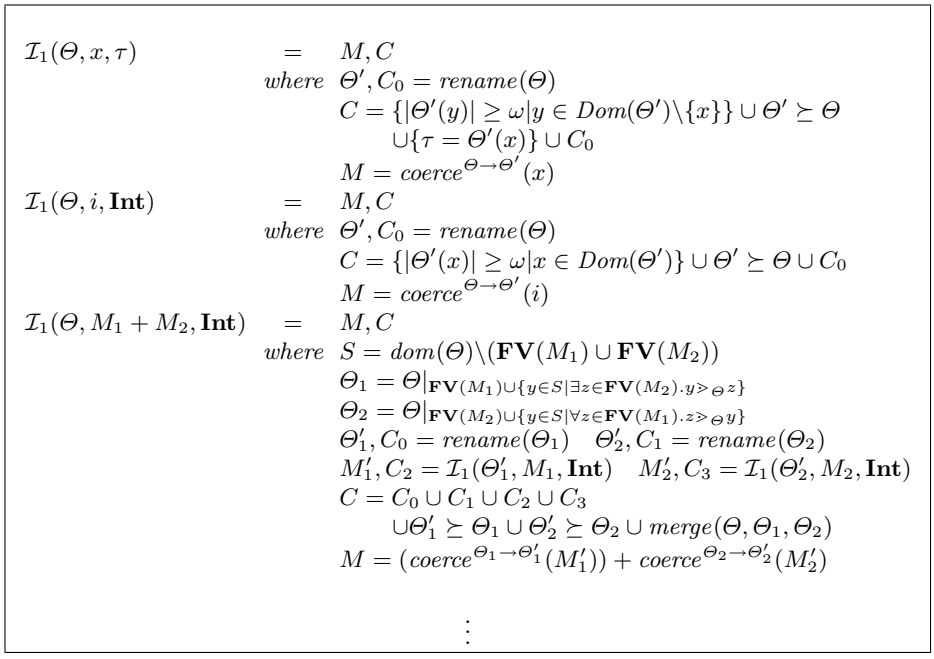


Fig. 10. Automatic insertion algorithm. *typeof*(M) returns the type of M inferred by the type reconstruction algorithm for $\Gamma \vdash_{\lambda\to} M : \tau$ **case**^{d} means **case** if $d = 1$ and **mcase** if $d = \omega$

The following theorems describe soundness and completeness of $\Theta \vdash M \rightsquigarrow M' : \tau$ with respect to $\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \tau$:

Theorem 3 (Soundness of $\Theta \vdash M \rightsquigarrow M' : \tau$). *If $\Theta \vdash M \rightsquigarrow M' : \tau$ holds, then there exist Γ and Δ that satisfy $\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \tau$ and $\Theta = \Gamma, \Delta$.*

Theorem 4 (Completeness of $\Theta \vdash M \rightsquigarrow M' : \tau$). *Suppose that $\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \tau$ is derived without using* Tr-Leaf1*,* Tr-Node1 *and* Tr-MemToStream*. Then, there exists Θ such that*

- $\Theta = \Gamma, \Delta$
- $\Theta \vdash M \rightsquigarrow M' : \tau$

Based on the rules in Figure 9, we construct $\mathcal{I}_1$ in Figure 10 that takes Θ, M and τ as input and returns the result of translation M' and constraints C. C consists of inequalities between uses and equalities between types. It is obtained by reading the rules in Figure 9 in a bottom-up manner. In Figure 10, *rename*(Θ) returns a pair of the type environment obtained by replacing the uses variables occurs in Θ with fresh use variables, and a set of constraints for the renamed type environment being well-formed (i.e.,$|\Theta(x_i)| \geq |\Theta(x_j)|$ for any $x_i \gtrdot_\Theta x_j$.) By abuse of notations, we write $\Theta_1 \succeq \Theta_2$ and *merge*$(\Theta, \Theta_1, \Theta_2)$ for the constraints

on uses required for $\Theta_1 \succeq \Theta_2$ and $merge(\Theta, \Theta_1, \Theta_2)$ to hold respectively. The function $typeof(N)$ returns the simple type of N.[1]

The following theorem states soundness of $\mathcal{I}_1$.

Theorem 5. *Suppose $\mathcal{I}_1(\Theta, M, \tau) = M', C$. Then, θ is a solution of C if and only if $\theta\Theta \vdash M \rightsquigarrow \theta M' : \theta\tau$ holds.*

Unfortunately, the converse of Theorem 5 does not hold. For example, consider the program $M = ((f\ x) + 1) + (f\ z)$. M can be transformed to both $M'_1 = ((f\ x) + \mathbf{letbuf}\ y = s2m(y)\ \mathbf{in}\ 1) + (f\ z)$ and $M'_2 = ((f\ x) + 1) + (\mathbf{letbuf}\ y = s2m(y)\ \mathbf{in}\ 1 + (f\ z))$ under the semi-ordered environment $\Theta = f : \mathbf{Tree^1} \to \mathbf{Int}, x : \mathbf{Tree^1}, y : \mathbf{Tree^1}, z : \mathbf{Tree^1}$ by the transformation rules in Figure 9. Only the latter derivation can, however, be derived by algorithm $\mathcal{I}_1$. This is because $\mathcal{I}_1(M_1 + M_2)$ divides the semi-ordered environment Θ into Θ_1 and Θ_2 in a fixed way (see Figure 10). This is not a problem from the viewpoint of the optimality of the transformation result: for any term M' obtained from M by using the rules in Figure 9, algorithm $\mathcal{I}_1$ generates a term M'' that is as efficient as M'. In the above example, M'_1 is as efficient as M'_2 ($s2m(y)$ in both terms can be replaced by *skip_tree*. See Section 5.), so that producing only M'_1 is sufficient.

Let $(M'', C) = \mathcal{I}_1(M)$. The constraints C can be reduced to a set of constraints on uses of the form $\{u_1 \geq d_1, \ldots, u_n \geq d_n\}$, where $u_1, \ldots, u_n$ are distinct use variables and $d_1, \ldots, d_n$ are expressions contructed from use variables, constants, and the operation $max(d, d')$ that takes an upper-bound of two uses d and d'. Since $d_1, \ldots, d_n$ are monotonic on $u_1, \ldots, u_n$, we can apply the standard algorithm [13] to obtain the least solution of C. The output of algorithm $\mathcal{I}$ is the term obtained by substituting the least solution for the use variables in M'' and reducing *coerce*.

4.2 Algorithm $\mathcal{P}$

We design $\mathcal{P}$ in a way similar to $\mathcal{I}$. We first introduce a judgment $\Gamma \vdash M \rightsquigarrow M' : \tau$ in a syntax-directed manner. Figure 11 shows a part of the rules for $\Gamma \vdash M \rightsquigarrow M' : \tau$. In the figure, $\tau_1 \succeq_{\mathcal{P}} \tau_2$ is the least reflexive transitive binary relation that satisfies $\mathbf{Tree}^+ \succeq_{\mathcal{P}} \mathbf{Tree}^\omega$. Γ is not ordered since $\mathcal{I}$ already guarantees that variables of type $\mathbf{Tree^1}$ are accessed in the correct order.

Based on $\Gamma \vdash M \rightsquigarrow M' : \tau$, we construct a sub-algorithm $\mathcal{P}_1$ that takes Γ, M and τ, and returns M' and C where C is a set of constraints on the use variables such that $\theta\Gamma \vdash M \rightsquigarrow \theta M' : \theta\tau$ if and only if the substitution θ satisfies C.

By combining $\mathcal{I}$ and $\mathcal{P}$, we have an algorithm that transform any program M such that $\emptyset \vdash M : \mathbf{Tree} \to \mathbf{Tree}$ into M' such that $\emptyset \mid \emptyset \vdash M' : \mathbf{Tree^1} \to \mathbf{Tree}^+$.

For the definition of $\mathcal{P}$, see the full version [14].

[1] Here, we assume that the type reconstruction algorithm for $\emptyset \vdash_{\lambda\to} M : \mathbf{Tree} \to \mathbf{Tree}$ is applied, and that the type of each subterm has been already determined. The variables whose types are not uniquely determined are not accessed, so we can safely assume that $typeof()$ returns $\mathbf{Int}$ for those variables.

$$coerce\_out^{\mathbf{Tree}^{\omega} \Rightarrow \mathbf{Tree}^{+}}(M) = m2s(M)$$
$$coerce\_out^{\tau \Rightarrow \tau}(M) = M$$

$$\frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash M \rightsquigarrow M' : \tau_2' \qquad \tau_2 \succeq_{\mathcal{P}} \tau_2'}{\Gamma \vdash \mathbf{fix}\ (f, x, M) \rightsquigarrow \mathbf{fix}\ (f, x, coerce\_out^{\tau_2' \Rightarrow \tau_2}(M')) : \tau_1 \to \tau_2} \text{(TO-Fix)}$$

$$\frac{\Gamma \vdash M \rightsquigarrow M' : \mathbf{Int}}{\Gamma \vdash \mathbf{mleaf}\ M \rightsquigarrow \mathbf{leaf}^{d}\ M' : \mathbf{Tree}^{d}} \text{(TO-Leaf)}$$

$$\frac{\begin{array}{rr}\Gamma \vdash M_1 \rightsquigarrow M_1' : \mathbf{Tree}^{d_1} & \Gamma \vdash M_2 \rightsquigarrow M_2' : \mathbf{Tree}^{d_2} \\ \mathbf{Tree}^{d} \succeq_{\mathcal{P}} \mathbf{Tree}^{d_1} & \mathbf{Tree}^{d} \succeq_{\mathcal{P}} \mathbf{Tree}^{d_2} \\ M_1'' = coerce\_out^{\mathbf{Tree}^{d_1} \Rightarrow \mathbf{Tree}^{d}}(M_1) & M_2'' = coerce\_out^{\mathbf{Tree}^{d_2} \Rightarrow \mathbf{Tree}^{d}}(M_2)\end{array}}{\Gamma \vdash \mathbf{mnode}\ M_1\ M_2 \rightsquigarrow \mathbf{node}^{d}\ M_1''\ M_2'' : \mathbf{Tree}^{d}} \text{(TO-Node)}$$

$$\frac{\begin{array}{c}\Gamma \vdash M \rightsquigarrow M' : \mathbf{Tree}^{d} \qquad \Gamma, x : \mathbf{Int} \vdash M_1 \rightsquigarrow M_1' : \tau_1 \\ \Gamma, x_1 : \mathbf{Tree}^{d}, x_2 : \mathbf{Tree}^{d} \vdash M_2 \rightsquigarrow M_2' : \tau_2 \\ \tau \succeq_{\mathcal{P}} \tau_1 \qquad \tau \succeq_{\mathcal{P}} \tau_2\end{array}}{\begin{array}{lcll} & \mathbf{case}^{d}\ M\ \mathbf{of} & & \mathbf{case}^{d}\ M'\ \mathbf{of} & \\ \Gamma \vdash & \quad \mathbf{leaf}^{d}\ x \Rightarrow M_1 & \rightsquigarrow & \quad \mathbf{leaf}^{d}\ x \Rightarrow coerce\_out^{\tau_1 \Rightarrow \tau}(M_1) & : \tau \\ & \mid \mathbf{node}^{d}\ x_1\ x_2 \Rightarrow M_2 & & \mid \mathbf{node}^{d}\ x_1\ x_2 \Rightarrow coerce\_out^{\tau_2 \Rightarrow \tau}(M_2) & \end{array}} \text{(TO-Case)}$$

Fig. 11. A part of declarative definition of the algorithm that inserts *m2s*

5 Post-processing to eliminate redundant buffering

Our algorithm presented so far inserts *s2m* and *m2s*, which copy trees from the input stream to memory, and from the memory to the output stream. Therefore, for example, the identity function $\lambda x.x$ of type $=\mathbf{Tree} \to \mathbf{Tree}$ is transformed into $\lambda x.\mathbf{letbuf}\ x = s2m(x)\ \mathbf{in}\ m2s(x)$, which contains redundant buffering. We apply the following transformation rules in the post-processing phase to eliminate such redundant buffering, before applying our previous framework [7].

$$\mathbf{letbuf}\ x = s2m(x)\ \mathbf{in}\ m2s(x) \Longrightarrow copy\_tree(x)$$
$$\mathbf{letbuf}\ x = s2m(x)\ \mathbf{in}\ M \Longrightarrow skip\_tree(x); M \quad \text{if } x \notin \mathbf{FV}(M)$$

Here, *copy_tree*(x) copies a tree from the input stream to the output stream without buffering the tree, and *skip_tree*(x) simply ignores a tree in the input stream. For example, the program $\lambda x.\mathbf{letbuf}\ x = s2m(x)\ \mathbf{in}\ m2s(x)$ is replaced by $\lambda x.copy\_tree(x)$.

6 Related Work

Nakano and Nishimura [8–11] proposed a method for translating tree-processing programs to stream-processing programs using attribute grammars or attribute

tree transducers [2]. In their method, programmers write XML processing as an attribute grammar or an attributed tree transducer. Then, those are composed with parsing and unparsing ones by using composition methods such as descriptional composition [3] and translated to a grammar that directly deals with streams. An advantage of our method is that we can deal with source programs that involve side-effects (e.g. programs that print the value of every leaf) while that seems difficult in their method based on attribute grammars (since the order is important for side effects). We also believe that our correctness proof is simpler than theirs. Comparison between the efficiency[2] of programs generated by our method and that of those generated by their method is left for future work.

From the viewpoint of ordered linear logic [12], $\mathcal{I}$ can be viewed as an algorithm that obtains a proof in a restricted fragment of ordered linear logic from a proof in intuitionistic logic.

7 Conclusion

We have proposed a method for automatically inserting buffering primitives into tree-processing programs; by combining it with our previous framework, any simply-typed tree-processing program can automatically be transformed into an equivalent stream-processing program. We have already implemented a prototype system to automatically insert buffering primitives. We plan to extend it to implement a generator for XML stream-processing programs.

References

1. Tim Bray, Jean Paoli, C.M.Sperberg-McQueen, and Eve Maler. Extensible markup language (XML) 1.0 (second edition). Technical report, World Wide Web Consortium, October 2000. `http://www.w3.org/TR/REC-xml`.
2. Zoltán Fülöp. On attributed tree transducers. In *Acta Cybernetica*, volume 5, pages 261–280, 1981.
3. Harald Ganzinger and Robert Giegerich. Attribute coupled grammars. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, 1984.
4. Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology (TOIT)*, 3(2):117–148, 2003.
5. Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 11–22, September 2000.
6. Koichi Kodama. Derivation of XML stream processor based on ordered linear type. Master's thesis, Tokyo Institute of Technology, March 2005.
7. Koichi Kodama, Kohei Suenaga, and Naoki Kobayashi. Translation of tree-processing programs into stream-processing programs based on ordered linear type. In Wei Ngan Chin, editor, *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, Proceedings*, volume 3302 of *Lecture Notes in Computer Science*, pages 41–56, November 2004.

[2] More precisely, which part of an input tree can be processed in a stream-processing manner and which part is copied to memory

8. Keisuke Nakano. Composing stack-attributed tree transducers. Technical Report METR–2004–01, Department of Mathematical Informatics, University of Tokyo, Japan, 2004.
9. Keisuke Nakano. An implementation scheme for XML transformation languages through derivation of stream processors. In *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, Proceedings*, volume 3302 of *Lecture Notes in Computer Science*, pages 74–90, November 2004.
10. Keisuke Nakano and Susumu Nishimura. Deriving event-based document transformers from tree-based specifications. In Mark van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001.
11. Susumu Nishimura and Keisuke Nakano. XML stream transformer generation through program composition and dependency analysis. *Science of Computer Programming*, 54:257–290, August 2004.
12. Jeff Polakow. *Ordered linear logic and applications*. PhD thesis, Carnegie Mellon University, June 2001. Available as Technical Report CMU-CS-01-152.
13. Jakob Rehof and Torben Mogensen. Tractable constraints in finite semilattices. *Science of Computer Programming*, 35(2):191–221, 1999.
14. Kohei Suenaga, Naoki Kobayashi, and Akinori Yonezawa. Extension of type-based approach to generation of stream-processing programs by automatic insertion of buffering primitives. Full paper. Available from `http://www.yl.is.s.u-tokyo.ac.jp/~kohei/doc/paper/lopstr05-full.pdf`.
15. V.Benzaken, G.Castagna, and A.Frisch. CDuce: An XML-centric general-purpose language. In *Proceedings of the ACM International Conference on Functional Programming*, 2003.
16. W3C. *Document Object Model (DOM) Level 1 Specification*, October 1998.