Translation of Tree-processing Programs into Stream-processing Programs based on Ordered Linear Type

Koichi Kodama*, Kohei Suenaga**, and Naoki Kobayashi***

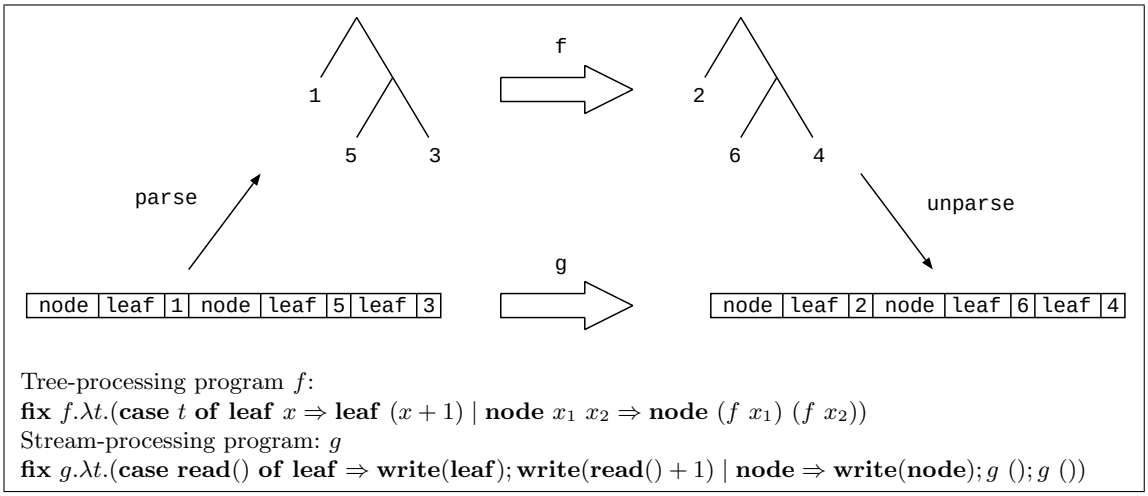
*Tokyo Institute of Technology, kodama@kb.cs.titech.ac.jp
**University of Tokyo, kohei@yl.is.s.u-tokyo.ac.jp
***Tokyo Institute of Technology, kobayasi@kb.cs.titech.ac.jp

Abstract. There are two ways to write a program for manipulating tree-structured data such as XML documents and S-expressions: One is to write a tree-processing program focusing on the logical structure of the data and the other is to write a stream-processing program focusing on the physical structure. While tree-processing programs are easier to write than stream-processing programs, tree-processing programs are less efficient in memory usage since they use trees as intermediate data. Our aim is to establish a method for automatically translating a tree-processing program to a stream-processing one in order to take the best of both worlds. We define a programming language for processing binary trees and a type system based on ordered linear type, and show that every well-typed program can be translated to an equivalent stream-processing program.

1 Introduction

There are two ways to write a program for manipulating tree-structured data such as XML documents [6] and S-expressions: One is to write a tree-processing program focusing on the logical structure of the data and the other is to write a stream-processing program focusing on the physical structure. For example, as for XML processing, DOM (Document Object Mode) API and programming language XDuce [12] are used for tree-processing, while SAX (Simple API for XML) is for stream-processing.

Figure 1 illustrates what tree-processing and stream-processing programs look like for the case of binary trees. The tree-processing program f takes a binary tree t as an input, and performs case analysis on t. If t is a leaf, it increments the value of the leaf. If t is a branch, f recursively processes the left and right subtrees. If actual tree data are represented as a sequence of tokens (as is often the case for XML documents), f must be combined with the function *parse* for parsing the input sequence, and the function *unparse* for unparsing the result tree into the output sequence, as shown in the figure. The stream-processing program g directly reads/writes data from/to streams. It reads an element from the input stream using the **read** primitive and performs case-analysis on the element. If the input is the **leaf** tag, g outputs **leaf** to the



Tree-processing program f:
fix $f.\lambda t.$(**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: g
fix $g.\lambda t.$(**case read**() **of leaf** $\Rightarrow$ **write**(**leaf**); **write**(**read**() $+ 1$) | **node** $\Rightarrow$ **write**(**node**); $g\ ()$; $g\ ()$)

Fig. 1. Tree-processing and stream-processing

output stream with the **write** primitive, reads another element, adds 1 to it, and outputs it. If the input is the **node** tag, g outputs **node** to the output stream and recursively calls the function g twice with the argument ().

Both of the approaches explained above have advantages and disadvantages. Tree-processing programs are written based on the logical structure of data, so that it is easier to write, read, and manipulate (e.g. apply program transformation like deforestation [23]) than stream-processing programs. On the other hand, stream-processing programs have their own advantage that intermediate tree structures are not needed, so that they often run faster than the corresponding tree-processing programs.

The goal of the present paper is to achieve the best of both approaches, by allowing a programmer to write a tree-processing program and automatically translating the program to an equivalent stream-processing program. To clarify the essence, we use a λ-calculus with primitives on binary trees, and show how the translation works.

The key observation is that we can obtain from a tree-processing program the corresponding stream-processing program simply by replacing case analyses on an input tree with case analyses on input tokens, and replacing tree constructions with stream outputs, *as long as the tree-processing program traverses and constructs trees from left to right in the depth-first manner*. In fact, the stream-processing program in Figure 1, which satisfies the above criterion, is obtained from the tree-processing program in that way. In order to check that a program satisfies the criterion, we use the idea of ordered linear types [19, 20]. Ordered linear types, which are an extension of linear types [3, 22], describe not only how often but also *in which order* data are used. Our type system designed based on the ordered linear types guarantees that a well-typed program traverses and constructs trees from left to right and in the depth-first order. Thus, every well-

Tree-processing program:
fix $f.\lambda t.$(**case** t **of leaf** $x \Rightarrow$ **leaf** $x \mid$ **node** $x_1\ x_2 \Rightarrow$ **node** $(f\ x_2)\ (f\ x_1)$)

Fig. 2. A program that swaps children of every node

typed program can be translated to an equivalent stream-processing program. The tree-processing program f in Figure 1 is well-typed in our type system, so that it can automatically be translated to the stream-processing program g. On the other hand, the program in Figure 2 is not well-typed in our type system since it accesses the right sub-tree of an input before accessing the left sub-tree. In fact, we would obtain a wrong stream-processing program if we simply apply the above-mentioned translation to the program in Figure 2.

The rest of the paper is organized as follows: To clarify the essence, we first focus on a minimal calculus in Section 2–4. In Section 2, we define the source language and the target language of the translation. We define a type system of the source language in Section 3. Section 4 presents a translation algorithm, shows its correctness and discuss the improvement gained by the translation. The minimal caclulus is not so expressive; especially, one can only write a program that does not store input/output trees on memory at all. (Strictly speaking, one can still store some information about trees by encoding it into lambda-terms.) Section 5 describes several extensions to recover the expressive power. With the extensions, one can write a program that selectively buffers input/output trees on memory, while the type system guarantees that the buffering is correctly performed. After discussing related work in Section 6, we conclude in Section 7.

For the restriction of space, proofs are omitted in this paper. They are found in the full version [14].

2 Language

We define the source and target languages in this section. The source language is a call-by-value functional language with primitives for manipulating binary trees. The target language is a call-by-value, impure functional language that uses imperative streams for input and output.

2.1 Source Language

The syntax and operational semantics of the source language is summarized in Figure 3.

The meta-variables x and i range over the sets of variables and integers respectively. The meta-variable W ranges over the set of values, which consists of integers i, lambda-abstractions $\lambda x.M$, and binary-trees V. A binary tree V is either a leaf labeled with an integer or a tree with two children. (**case** M **of leaf** $x \Rightarrow M_1 \mid$ **node** $x_1\ x_2 \Rightarrow M_2$) performs case analysis on a tree. If M is a leaf, x is bound to its label and M_1 is evaluated. Otherwise, x_1

Terms, values and evaluation contexts:

$$
\begin{array}{lll}
M \text{ (terms)} & ::= & i \mid \lambda x.M \mid x \mid M_1\ M_2 \mid M_1 + M_2 \mid \mathbf{fix}\ f.M \\
& & \mid \mathbf{leaf}\ M \mid \mathbf{node}\ M_1\ M_2 \\
& & \mid (\mathbf{case}\ M\ \mathbf{of\ leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2) \\
V \text{ (tree values)} & ::= & \mathbf{leaf}\ i \mid \mathbf{node}\ V_1\ V_2 \\
W \text{ (values)} & ::= & i \mid \lambda x.M \mid V \\
E_s \text{ (evaluation contexts)} & ::= & [\,] \mid E_s\ M \mid (\lambda x.M)\ E_s \mid E_s + M \mid i + E_s \\
& & \mid \mathbf{leaf}\ E_s \mid \mathbf{node}\ E_s\ M \mid \mathbf{node}\ V\ E_s \\
& & \mid (\mathbf{case}\ E_s\ \mathbf{of\ leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2)
\end{array}
$$

Reduction rules:

$$E_s[i_1 + i_2] \longrightarrow E_s[\mathit{plus}(i_1, i_2)] \quad \text{(Es-Plus)}$$

$$E_s[(\lambda x.M)W] \longrightarrow E_s[[W/x]M] \quad \text{(Es-App)}$$

$$E_s[\mathbf{fix}\ f.M] \longrightarrow E_s[[\mathbf{fix}\ f.M/f]M] \quad \text{(Es-Fix)}$$

$$E_s[\mathbf{case\ leaf}\ i\ \mathbf{of\ leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2] \longrightarrow E_s[[i/x]M_1] \quad \text{(Es-Case1)}$$

$$E_s[\mathbf{case\ node}\ V_1\ V_2\ \mathbf{of\ leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2] \longrightarrow E_s[[V_1/x_1, V_2/x_2]M_2] \quad \text{(Es-Case2)}$$

Fig. 3. The syntax, evaluation context and reduction rules of the source language. $\mathit{plus}(i_1, i_2)$ is the sum of i_1 and i_2.

and x_2 are bound to the left and right children respectively and M_2 is evaluated. **fix** $f.M$ is a recursive function that satisfies $f = M$. Bound and free variables are defined as usual. We assume that α-conversion is implicitly applied so that bound variables are always different from each other and free variables.

We write **let** $x = M_1$ **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.

2.2 Target Language

The syntax and operational semantics of the source language is summarized in Figure 4. A stream, represented by the meta variable S, is a sequence consisting of **leaf**, **node** and integers. We write $\emptyset$ for the empty sequence and write $S_1; S_2$ for the concatenation of the sequences S_1 and S_2.

read is a primitive for reading a token (**leaf**, **node**, or an integer) from the input stream. **write** is a primitive for writing a value to the output stream. The term (**case** e **of leaf** $\Rightarrow e_1 \mid$ **node** $\Rightarrow e_2$) performs a case analysis on the value of e. If e evaluates to **leaf**, e_1 is evaluated and if e evaluates to **node**, e_2 is evaluated. **fix** $f.e$ is a recursive function that satisfies $f = e$. Bound and free variables are defined as usual.

Terms, values and evaluation contexts:

$$
\begin{array}{rl}
e \text{ (terms)} ::= & v \mid x \mid e_1\ e_2 \mid e_1 + e_2 \mid \mathbf{fix}\ f.e \\
& \mid \mathbf{read}\ e \mid \mathbf{write}\ e \mid (\mathbf{case}\ e\ \mathbf{of\ leaf} \Rightarrow e_1 \mid \mathbf{node} \Rightarrow e_2) \\
v \text{ (values)} ::= & i \mid \mathbf{leaf} \mid \mathbf{node} \mid \lambda x.e \mid () \\
E_t \text{ (evaluation contexts)} ::= & [\,] \mid E_t\ e \mid (\lambda x.e)\ E_t \mid E_t + e \mid i + E_t \\
& \mid \mathbf{read}\ E_t \mid \mathbf{write}\ E_t \\
& \mid (\mathbf{case}\ E_t\ \mathbf{of\ leaf} \Rightarrow e_1 \mid \mathbf{node} \Rightarrow e_2)
\end{array}
$$

Reduction rules:

$$(E_t[v_1 + v_2], S_i, S_o) \longrightarrow (E_t[\mathit{plus}(v_1, v_2)], S_i, S_o) \quad \text{(Et-Plus)}$$

$$(E_t[(\lambda x.M)v], S_i, S_o) \longrightarrow (E_t[[v/x]M], S_i, S_o) \quad \text{(Et-App)}$$

$$(E_t[\mathbf{fix}\ f.e], S_i, S_o) \longrightarrow (E_t[[\mathbf{fix}\ f.e/f]e], S_i, S_o) \quad \text{(Et-Fix)}$$

$$(E_t[\mathbf{read}()], v; S_i, S_o) \longrightarrow (E_t[v], S_i, S_o) \quad \text{(Et-Read)}$$

$(E_t[\mathbf{write}\ v], S_i, S_o) \longrightarrow (E_t[()], S_i, S_o; v)$ (when v is an integer, **leaf** or **node**) (Et-Write)

$$(E_t[\mathbf{case\ leaf\ of\ leaf} \Rightarrow e_1 \mid \mathbf{node} \Rightarrow e_2], S_i, S_o) \longrightarrow (E_t[e_1], S_i, S_o) \quad \text{(Et-Case1)}$$

$$(E_t[\mathbf{case\ node\ of\ leaf} \Rightarrow e_1 \mid \mathbf{node} \Rightarrow e_2], S_i, S_o) \longrightarrow (E_t[e_2], S_i, S_o) \quad \text{(Et-Case2)}$$

Fig. 4. The reduction rules of the target language.

We write **let** $x = e_1$ **in** e_2 for $(\lambda x.e_2)\ e_1$. Especially, if e_2 does not contain x as a free variable, we write $e_1; e_2$ for it.

Figure 5 shows programs that take a tree as an input and calculate the sum of leaf elements. The source program takes a tree t as an argument of the function, and performs a case analysis on t. If t is a leaf, the program binds x to the element and returns it. If t is a branch node, the program recursively applies f to the left and right children and returns the sum of the results. The target program reads a tree (as a sequence of tokens) from the input stream, performs a case analysis on tokens, and returns the sum of leaf elements. Here, we assume that the input stream represents a valid tree. If the input stream is in a wrong format (e.g., when the stream is **node**; 1; 2), the execution gets stuck.

3 Type System

In this section, we present a type system of the source language, which guarantees that a well-typed program reads every node of an input tree exactly once from left to right in the depth-first order. Thanks to this guarantee, any well-typed

A source program:
fix *sumtree*.λt.(**case** t **of leaf** $x \Rightarrow x$ | **node** $x_1\ x_2 \Rightarrow$ (*sumtree* x_2) + (*sumtree* x_1))
A target program:
fix *sumtree*.λt.(**case read**() **of leaf** $\Rightarrow$ **read**() | **node** $\Rightarrow$ *sumtree* () + *sumtree* ())

Fig. 5. Programs that calculate the sum of leaf elements of an binary tree.

program can be translated to an equivalent, stream-processing program without changing the structure of the program, as shown in the next section. To enforce the depth-first access order on input trees, we use ordered linear types [19, 20].

3.1 Type and Type Environment

Definition 1 (Type). The set of *types*, ranged over by τ, is defined by:

$$\begin{aligned} \tau \text{ (type)} &::= \mathbf{Int} \mid \mathbf{Tree}^d \mid \tau_1 \to \tau_2 \\ d \text{ (mode)} &::= - \mid + \end{aligned}$$

Int is the type of integers. For a technical reason, we distinguish between input trees and output trees by types. We write $\mathbf{Tree}^-$ for the type of input trees, and write $\mathbf{Tree}^+$ for the type of output trees. $\tau_1 \to \tau_2$ is the type of functions from τ_1 to τ_2.

We introduce two kinds of type environments for our type system: ordered linear type environments and (non-ordered) type environments.

Definition 2 (Ordered Linear Type Environment). An *ordered linear type environment* is a sequence of the form $x_1{:}\mathbf{Tree}^-, \ldots, x_n{:}\mathbf{Tree}^-$, where $x_1, \ldots, x_n$ are different from each other. We write Δ_1, Δ_2 for the concatenation of Δ_1 and Δ_2.

An ordered linear type environment $x_1 : \mathbf{Tree}^-, \ldots, x_n : \mathbf{Tree}^-$ 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 subtrees bound to $x_1, \ldots, x_n$ must be accessed in the left-to-right, depth-first order.

Definition 3 (Non-Ordered 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* $\{\tau_1, \ldots, \tau_n\}$ *does not contain* $\mathbf{Tree}^d$.

We use the meta-variable Γ for non-ordered type environments. We often write $\Gamma, x : \tau$ for $\Gamma \cup \{x : \tau\}$, and write $x_1 : \tau_1, \ldots, x_n : \tau_n$ for $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$.

Note that a non-ordered type environment must not contain variables of tree types. $\mathbf{Tree}^-$ is excluded since input trees must be accessed in the specific order. $\mathbf{Tree}^+$ is excluded in order to forbid output trees from being bound to variables. For example, we will exclude a program like **let** $x_1 = t_1$ **in let** $x_2 = t_2$ **in node** $x_2\ x_1$. This restriction is convenient for ensuring that trees are constructed in the specific (from left to right, and in the depth-first manner) order.

$$\frac{}{\Gamma \mid x : \mathbf{Tree}^- \vdash x : \mathbf{Tree}^-} \quad \text{(T-Var1)}$$

$$\frac{}{\Gamma, x : \tau \mid \emptyset \vdash x : \tau} \quad \text{(T-Var2)}$$

$$\frac{}{\Gamma \mid \emptyset \vdash i : \mathbf{Int}} \quad \text{(T-Int)}$$

$$\frac{\Gamma \mid x : \mathbf{Tree}^- \vdash M : \tau}{\Gamma \mid \emptyset \vdash \lambda x.M : \mathbf{Tree}^- \to \tau} \quad \text{(T-Abs1)}$$

$$\frac{\Gamma, x : \tau_1 \mid \emptyset \vdash M : \tau_2}{\Gamma \mid \emptyset \vdash \lambda x.M : \tau_1 \to \tau_2} \quad \text{(T-Abs2)}$$

$$\frac{\Gamma \mid \Delta_1 \vdash M_1 : \tau_2 \to \tau_1 \qquad \Gamma \mid \Delta_2 \vdash M_2 : \tau_2}{\Gamma \mid \Delta_1, \Delta_2 \vdash M_1 M_2 : \tau_1} \quad \text{(T-App)}$$

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

$$\frac{\Gamma, f : \tau_1 \to \tau_2 \mid \emptyset \vdash M : \tau_1 \to \tau_2}{\Gamma \mid \emptyset \vdash \mathbf{fix}\ f.M : \tau_1 \to \tau_2} \quad \text{(T-Fix)}$$

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

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

$$\frac{\Gamma \mid \Delta_1 \vdash M : \mathbf{Tree}^- \qquad \Gamma, x : \mathbf{Int} \mid \Delta_2 \vdash M_1 : \tau \qquad \Gamma \mid x_1 : \mathbf{Tree}^-, x_2 : \mathbf{Tree}^-, \Delta_2 \vdash M_2 : \tau}{\Gamma \mid \Delta_1, \Delta_2 \vdash \mathbf{case}\ M\ \mathbf{of\ leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2 : \tau} \quad \text{(T-Case)}$$

Fig. 6. Rules of typing judgment

3.2 Type Judgment

A type judgement is of the form $\Gamma \mid \Delta \vdash M : \tau$, where Γ is a non-ordered type environment and Δ is an ordered linear type environment. The judgment means "If M evaluates to a value under an environment described by Γ and Δ, the value has type τ and the variables in Δ are accessed in the order specified by Δ." For example, if $\Gamma = \{f : \mathbf{Tree}^- \to \mathbf{Int}\}$ and $\Delta = x_1 : \mathbf{Tree}^-, x_2 : \mathbf{Tree}^-$,

$$\Gamma \mid \Delta \vdash \mathbf{node}\ (f\ x_1)\ (f\ x_2) : \mathbf{Int}$$

holds, while

$$\Gamma \mid \Delta \vdash \mathbf{node}\ (f\ x_2)\ (f\ x_1) : \mathbf{Int}$$

does not. The latter program violates the restriction specified by Δ that x_1 and x_2 must be accessed in this order.

$\Gamma \mid \Delta \vdash M : \tau$ is the least relation that is closed under the rules in Figure 6. T-Var1, T-Var2 and T-Int are the rules for variables and integer constants.

As in ordinary linear type systems, these rules prohibit variables that do not occur in a term from occurring in the ordered linear type environment. (In other words, weakening is not allowed on an ordered linear type environment.) That restriction is necessary to guarantee that each variable in an ordered linear type environment is accessed exactly once.

T-Abs1 and T-Abs2 are rules for lambda abstraction. Note that the ordered type environments of the conclusions of these rules must be empty. This restriction prevents input trees from being stored in function closures. That makes it easy to enforce the access order on input trees. For example, without this restriction, the function

$$\lambda t.\mathbf{let}\ g = \lambda f.(f\ t)\ \mathbf{in}\ (g\ \mathit{sumtree}) + (g\ \mathit{sumtree})$$

would be well-typed where *sumtree* is the function given in Figure 5. However, when a tree is passed to this function, its nodes are accessed twice because the function g is called twice. The program above is actually rejected by our type system since the closure $\lambda f.(f\ t)$ is not well-typed due to the restriction of T-Abs2.[1]

T-App is the rule for function application. The ordered linear type environments of M_1 and M_2, Δ_1 and Δ_2 respectively, are concatenated in this order because when $M_1\ M_2$ is evaluated, (1) M_1 is first evaluated, (2) M_2 is then evaluated, and (3) M_1 is finally applied to M_2. In the first step, the variables in Δ_1 are accessed in the order specified by Δ_1. In the second and third steps, the variables in Δ_2 are accessed in the order specified by Δ_2, On the other hand, because there is no restriction on usage of the variables in a non-ordered type environment, the same type environment (Γ) is used for both subterms.

T-Leaf and T-Node are rules for tree construction. We concatenate the ordered type environments of M_1 and M_2, Δ_1 and Δ_2, in this order as we did in T-App.

T-Case is the rule for case expressions. If M matches $\mathbf{node}\ x_1\ x_2$, subtrees x_1 and x_2 have to be accessed in this order after that. This restriction is expressed by $x_1 : \mathbf{Tree}^-, x_2 : \mathbf{Tree}^-, \Delta_2$, the ordered linear type environment of M_2.

T-Fix is the rule for recursion. Note that the ordered type environment must be empty as in T-Abs2.

The program in Figure 1 is typed as shown in Figure 7. On the other hand, the program in Figure 2 is ill-typed: $\Gamma \mid x_1{:}\mathbf{Tree}^-, x_2{:}\mathbf{Tree}^- \vdash \mathbf{node}\ (f\ x_2)\ (f\ x_1){:}\ \mathbf{Tree}^+$ must hold for the program to be typed, but it cannot be derived by using T-Node.

3.3 Examples of Well-typed Programs

Figure 8 shows more examples of well-typed source programs. The frist and second programs (or the catamorphism [16]) apply the same operation on every

[1] We can relax the restriction by controlling usage of not only trees but also functions, as in the resource usage analysis [13]. The resulting type system would, however, become very complex.

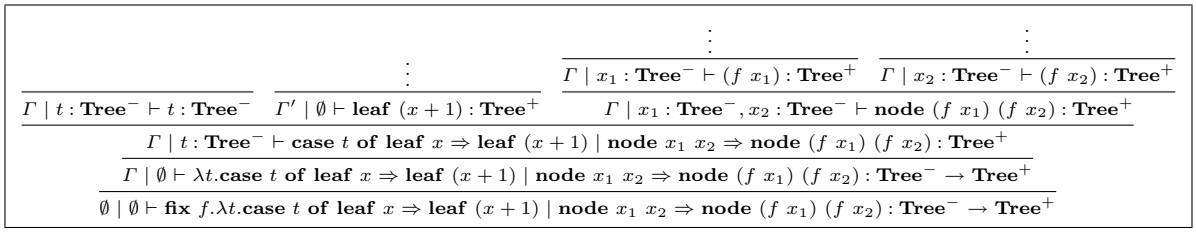


Fig. 7. An example of typing derivation. $\Gamma = \{f{:}\mathbf{Tree}^- \to \mathbf{Tree}^+\}$, $\Gamma' = \{f{:}\mathbf{Tree}^- \to \mathbf{Tree}^+, x : \mathbf{Int}\}$

```
fix tree_map.λf.λt.(case t of leaf x ⇒ leaf (f x) | node x1 x2 ⇒ node (tree_map f x1) (tree_map f x2))
fix tree_fold.λf.λg.λt.(case t of leaf n ⇒ (f n) | node t1 t2 ⇒ (g (tree_fold f g t1) (tree_fold f g t2)))
fix inc_alt.λt.(case t of leaf x ⇒ leaf x | node x1 x2 ⇒ node
      (case x1 of leaf y ⇒ leaf (y + 1)
            | node y1 y2 ⇒ node (inc_alt y1) (inc_alt y2))
      (case x2 of leaf z ⇒ leaf (z + 1)
            | node z1 z2 ⇒ node (inc_alt z1) (inc_alt z2))

let copy_tree =
    fix copy_tree.λt.(case t of leaf x ⇒ leaf x | node x1 x2 ⇒ node (copy_tree x1) (copy_tree x2)) in
let skip_tree = fix skip_tree.λt.(case t of leaf x ⇒ 0 | node x1 x2 ⇒ (skip_tree x1); (skip_tree x2)) in
      λt.(case t of leaf x ⇒ leaf x | node x1 x2 ⇒ (skip_tree x1); (copy_tree x2))
```

Fig. 8. Examples of well-typed programs.

node of the input tree. (The return value of the function *tree_fold* cannot, however, be a tree.) One can also write functions that process nodes in a non-uniform manner, like the third program in Figure 8 (which increments the value of each leaf whose depth is odd).

The fourth program takes a tree as an input and returns the right subtree. Due to the restriction imposed by the type system, the program uses subfunctions *copy_tree* and *skip_tree* for explicitly copying and skipping trees. (See Section 7 for a method for automatically inserting those functions.)

3.4 Type Checking Algorithm

We show an algorithm that takes as an input a triple (Γ, Δ, M) consisting of a non-ordered type environment Γ, an ordered linear type environment Δ and a type-annotated term M, and outputs τ such that $\Gamma \mid \Delta \vdash M : \tau$ and reports failure if such τ does not exists. Here, by a type-annotated term, we mean a term whose bound variables are annotated with types.

The algorithm is basically obtained by reading typing rules in a bottom-up manner, but a little complication arises on the rules T-App, T-Plus, etc., for which we do not know how to split the ordered linear type environment for sub-terms. We avoid this problem by splitting the ordered type environment

$$\textbf{fix}\ f.\lambda t.\textbf{case}\ t\ \textbf{of leaf}\ x \Rightarrow \textbf{leaf}\ x \mid \textbf{node}\ x_1\ x_2 \Rightarrow (\lambda t'.\textbf{node}\ (f\ t')\ (f\ x_2))\ x_1$$

Fig. 9. Program that is not typed due to the restriction on closures though the access order is correct.

lazily [11]. To make the lazy splitting of the ordered type environment explicit, we use a relation $\Gamma \mid \Delta \vdash M : \tau \nearrow \Delta'$. The relation means that $\Gamma \mid \Delta_1 \vdash M : \tau$ holds for Δ_1 such that $\Delta = \Delta_1, \Delta'$. As the specification of the type-checking algorithm, it can be read "Given a triple (Γ, Δ, M), as an input, the algorithm outputs the type τ of M and the unused ordered type environment Δ'."

For example, if $\Gamma = f : \mathbf{Tree}^- \to \mathbf{Tree}^+$ and $\Delta = x_1 : \mathbf{Tree}^-, x_2 : \mathbf{Tree}^-$,

$$\Gamma \mid \Delta \vdash (f\ x_1) : \mathbf{Tree}^+ \nearrow x_2 : \mathbf{Tree}^-$$

holds.

By using the relation above, the procedure for type-checking function applications is formalized as:

$$\frac{\Gamma \mid \Delta_1 \vdash M_1 : \tau_1 \to \tau_2 \nearrow \Delta_2 \qquad \Gamma \mid \Delta_2 \vdash M_2 : \tau_1 \nearrow \Delta_3}{\Gamma \mid \Delta_1 \vdash M_1\ M_2 : \tau_2 \nearrow \Delta_3}$$

The above rule specifies that given $(\Gamma, \Delta_1, M_1 M_2)$ as an input, we should (i) first type-check M_1 and obtain a type τ_3 and the unused environment Δ_2, (ii) type-check M_2 using Γ and Δ_2 and obtain τ_1 and the unused environment Δ_3, and then (iii) unifies τ_3 with $\tau_1 \to \tau_2$ and outputs τ_2 and Δ_3 as a result.

To check whether $\Gamma \mid \Delta \vdash M : \tau$ holds, it is sufficient to check whether $\Gamma \mid \Delta \vdash M : \tau \nearrow \emptyset$ holds. The whole rules for the relation $\Gamma \mid \Delta \vdash M : \tau \nearrow \Delta'$ is given in Appendix A.

4 Translation Algorithm

In this section, we define a translation algorithm for well-typed source programs and prove its correctness.

4.1 Definition of Translation

Translation algorithm $\mathcal{A}$ is shown in Figure 10. $\mathcal{A}$ maps a source program to a target program, preserving the structure of the source program and replacing operations on trees with operations on streams.

4.2 Correctness of Translation Algorithm

The correctness of the translation algorithm $\mathcal{A}$ is stated as follows.

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

Fig. 10. Translation Algorithm

Definition 4. A function $[\![\cdot]\!]$ from the set of trees to the set of streams is defined by:

$$
\begin{array}{c}
[\![\, \mathbf{leaf}\ i \,]\!] = \mathbf{leaf}; i \\
[\![\, \mathbf{node}\ V_1\ V_2 \,]\!] = \mathbf{node}; [\![V_1]\!]; [\![V_2]\!]
\end{array}
$$

Theorem 1 (Correctness of Translation).
If $\emptyset \mid \emptyset \vdash M : \mathbf{Tree}^- \to \tau$ *and* τ *is* **Int** *or* $\mathbf{Tree}^+$*, the following properties hold for any tree value* V*:*

(i) $M\ V \longrightarrow^* i$ *if and only if* $(\mathcal{A}(M)(), [\![V]\!], \emptyset) \longrightarrow^* (i, \emptyset, \emptyset)$
(ii) $M\ V \longrightarrow^* V'$ *if and only if* $(\mathcal{A}(M)(), [\![V]\!], \emptyset) \longrightarrow^* ((), \emptyset, [\![V']\!])$

The above theorem means that a source program and the corresponding target program evaluates to the same value. The clause (i) is for the case where the result is an integer, and (ii) is for the case where the result is a tree.

We give an outline of the proof of Theorem 1 below. The basic idea of the proof is to show a correspondence between reduction steps of a source program and those of the target program. However, the reduction semantics given in Section 2 is not convenient for showing the correspondence. We define another reduction semantics of the source language and prove: (1) the new semantics is equivalent to the one in Section 2 (Corollary 1 below), and (2) evaluation of the source program based on the new semantics agrees with evaluation of the corresponding target program (Theorem 3 below). First, we define a new operational semantics of the source language. The semantics takes the access order of input trees into account.

Definition 5 (Ordered Environment). An ordered environment is a sequence of the form $x_1 \mapsto V_1, \ldots, x_n \mapsto V_n$, where $x_1, \ldots, x_n$ are distinct from each other.

We use a meta-variable δ to represent an ordered environment. Given an ordered environment $x_1 \mapsto V_1, \ldots, x_n \mapsto V_n$, a program must access variables $x_1, \ldots, x_n$ in this order.

Definition 6 (New Reduction Semantics). *The reduction relation* $(M, \delta) \longrightarrow (M', \delta')$ *is the least relation that satisfies the rules in Figure 11.*

$$U ::= x \mid i \mid \lambda x.M$$

$$(E_s[i_1 + i_2], \delta) \longrightarrow (E_s[plus(i_1, i_2)], \delta) \qquad \text{(Es2-Plus)}$$

$$(E_s[(\lambda x.M)U], \delta) \longrightarrow (E_s[[U/x]M], \delta) \qquad \text{(Es2-App)}$$

$$(E_s[\mathbf{case}\ y\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2], (y \mapsto \mathbf{leaf}\ i, \delta)) \longrightarrow (E_s[[i/x]M_1], \delta) \qquad \text{(Es2-Case1)}$$

$$(E_s[\mathbf{case}\ y\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2], (y \mapsto \mathbf{node}\ V_1\ V_2, \delta)) \longrightarrow (E_s[M_2], (x_1 \mapsto V_1, x_2 \mapsto V_2, \delta)) \qquad \text{(Es2-Case2)}$$

$$(E_s[\mathbf{fix}\ f.M], \delta) \longrightarrow (E_s[[\mathbf{fix}\ f.M/f]M], \delta) \qquad \text{(Es2-Fix)}$$

Fig. 11. The new reduction semantics of the source language.

The meta-variable U in Figure 11 ranges over the set of trees and non-tree values. The differences between the new reduction semantics above and the original one in Section 2 are: (1) input trees are substituted in the original semantics while they are held in ordered environments in the new semantics (compare Es-Case2 with Es2-Case2), and (2) input trees must be accessed in the order specified by δ in the new semantics (note that variable y that is being referred to must be at the head of the ordered environment in Es2-Case1 and Es2-Case2). Thus, evaluation based on the new semantics can differ from the one in Section 2 only when the latter one succeeds while the former one gets stuck due to the restriction on access to input trees. As the following theorem (Theorem 2) states, that cannot happen if the program is well-typed, so that both semantics are equivalent for well-typed programs (Corollary 1).

Theorem 2. *Suppose* $\Gamma \mid \langle\langle \delta \rangle\rangle \vdash M : \tau$. *Then the following conditions hold.*

- M *is a value or a variable, or* $(M, \delta) \longrightarrow (M', \delta')$ *holds for some* M' *and* δ'.
- *If* $(M, \delta) \longrightarrow (M', \delta')$ *holds, then* $\Gamma \mid \langle\langle \delta' \rangle\rangle \vdash M' : \tau$.

Corollary 1. *If* $\emptyset \mid \emptyset \vdash M : \mathbf{Tree}^- \to \tau$ *and if* $\tau \in \{\mathbf{Int}, \mathbf{Tree}^+\}$, $MV \longrightarrow^* W$ *if and only if* $(Mx, x \mapsto V) \longrightarrow^* (W, \emptyset)$ *for any tree value* V.

The following theorem states that the evaluation of a source program under the new rules agrees with the evaluation of the target program.

Theorem 3. *If* $\emptyset \mid \emptyset \vdash M : \mathbf{Tree}^- \to \tau$ *and* $\tau \in \{\mathbf{Int}, \mathbf{Tree}^+\}$ *holds, the following statements hold for any tree value* V.

(i) $(Mx, x \mapsto V) \longrightarrow^* (i, \emptyset)$ *holds if and only if* $(\mathcal{A}(M)(), [\![V]\!], \emptyset) \longrightarrow^* (i, \emptyset, \emptyset))$

(ii) If $(Mx, x \mapsto V) \longrightarrow^* (V', \emptyset)$ *holds if and only if* $(\mathcal{A}(M)(), [\![V]\!], \emptyset) \longrightarrow^* ((), \emptyset, [\![V']\!]))$

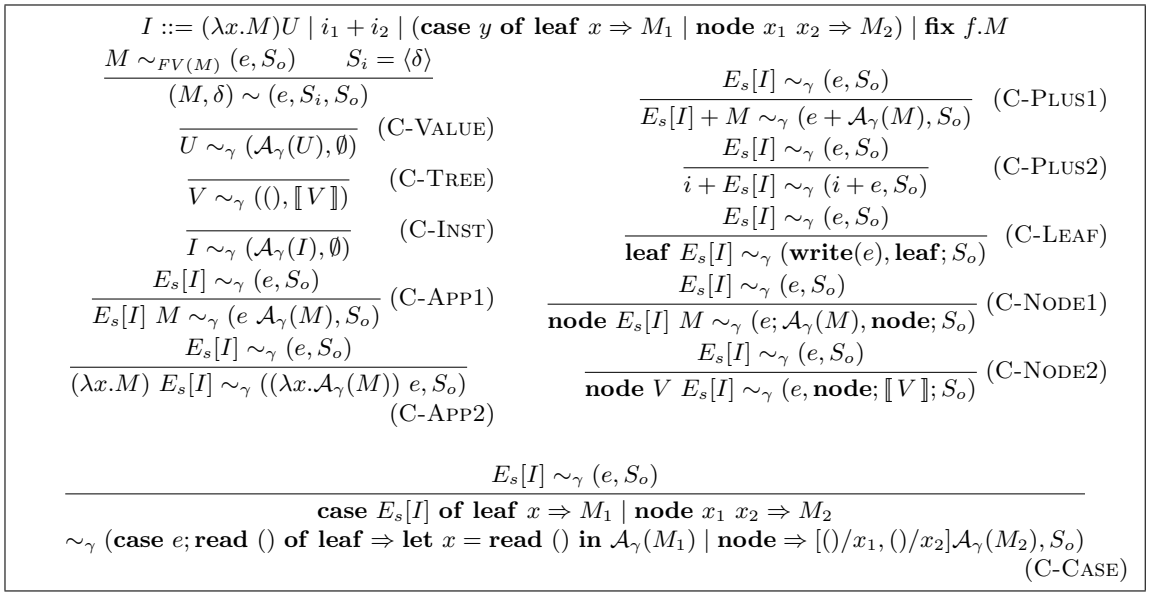


Fig. 12. Correspondence between run-time states of source and target programs.

We hereafter give an outline of the proof of Theorem 3. Figure 13 illustrates the idea of the proof (for the case where the result is a tree). The relation $\sim$ (defined later in Definition 8) in the diagram expresses the correspondence between an evaluation state of a source program (M, δ) and a state of a target program (e, S_i, S_o). We shall show that the target program $\mathcal{A}(M)$ can always be reduced to a state corresponding to the inital state of the source program M (Lemma 1 below) and that reductions and the correspondence relation commute (Lemma 2). Those imply that the whole diagram in Figure 13 commutes, i.e., the second statement of Theorem 3 holds.

To define the correspondence $(M, \delta) \sim (e, S_i, S_o)$ between states, we use the following function $\langle\cdot\rangle$, which maps an ordered environment to the corresponding stream.

Definition 7. A function $\langle\cdot\rangle$ from the set of ordered environments to the set of streams is defined by:

$$\langle\emptyset\rangle = \emptyset$$
$$\langle x \mapsto V, \delta\rangle = [\![V]\!]; \langle\delta\rangle$$

Definition 8 (Correspondence between States). The relations $(M, \delta) \sim (e, S_i, S_o)$ and $M \sim_\gamma (e, S_o)$ are the least relations closed under the rules in Figure 12.

In the figure, the meta-variable γ denotes a set of variables. $\mathbf{FV}(M)$ is a set of free variables in M. $\mathcal{A}_\gamma(M)$ is the term obtained from $\mathcal{A}(M)$ by replacing

every occurrence of variables in γ with $()$. The meta-variable I represents the term that is being reduced. Note that any term M can be written as $E_s[I]$ if it is reducible.

In the relation $(M, \delta) \sim (e, S_i, S_o)$, e represents the rest of computation, S_i is the input stream, and S_o is the already output streams. For example, $(\mathbf{node}(\mathbf{leaf}\ 1)(\mathbf{leaf}\ (2+3)), \emptyset)$ corresponds to $(2+3, \emptyset, \mathbf{node}; \mathbf{leaf}; 1; \mathbf{leaf})$.

We explain some of the rules in Figure 12 below.

- C-Tree: A source program V represents a state where the tree V has been constructed. Thus, it corresponds to $((), [\![V]\!])$, where there is nothing to be computed and V has been written to the output stream.
- C-Node1: A source program $\mathbf{node}\ E_s[I]\ M$ represents a state where the left subtree is being computed. Thus, the rest computation of the target program is $(e; \mathcal{A}_\gamma(M))$ where e is the rest computation in $E_s[I]$, and $\mathcal{A}_\gamma(M)$ represents the computation for constructing the right subtree. The corresponding output stream is $\mathbf{node}; S_o$ because $\mathbf{node}$ represents the root of the tree being constructed, and S_o represents the part of the left subtree that has been already constructed.

Lemmas 1 and 2 below imply that the whole diagram in Figure 12 commutes, which completes the proof of Theorem 3.

Definition 9. A function $\langle\langle\cdot\rangle\rangle$ from the set of ordered environments to the set of ordered linear type environments is defined by:

$$\langle\langle\emptyset\rangle\rangle = \emptyset$$
$$\langle\langle x \mapsto V, \delta\rangle\rangle = x : \mathbf{Tree}^-, \langle\langle\delta\rangle\rangle$$

Lemma 1. *Suppose $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash M : \tau$. Then, there exist e and S_i and S_o that satisfy*

- $(M, \delta) \sim (e, S_i, S_o)$
- $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \longrightarrow^* (e, \langle\delta\rangle, S_o)$

Lemma 2. *If $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash M : \tau$ and $(M, \delta) \sim (e, S_i, S_o)$, the following conditions hold:*

- *If $(M, \delta) \longrightarrow (M', \delta')$, then there exist e' and S_i' and S_o' that satisfy $(e, \langle\delta\rangle, S) \longrightarrow^+ (e', \langle\delta'\rangle, S')$ and $(M', \delta') \sim (e', S_i', S_o')$.*
- *If $(e, \langle\delta\rangle, S)$ is reducible, there exist M' and δ' that satisfy $(M, \delta) \longrightarrow (M', \delta')$.*

4.3 Efficiency of Translated Programs

Let M be a source program of type $\mathbf{Tree}^- \to \mathbf{Tree}^+$. We argue below that the target program $\mathcal{A}(M)$ runs more efficiently than the source program $\mathit{unparse} \circ M \circ \mathit{parse}$, where *parse* is a function that parses the input stream and returns a binary tree, and *unparse* is a function that takes a binary tree as an input and writes it to the output stream. Note that the fact that the target program is a

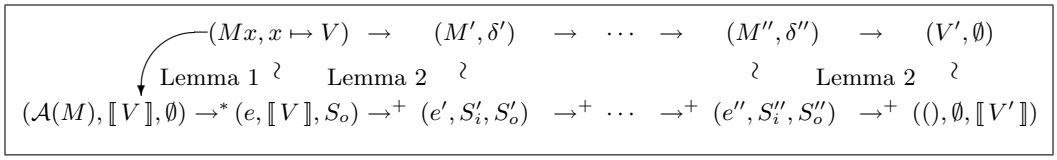


Fig. 13. Evaluation of a source and the target program.

stream-processing program does not necessarily imply that it is more efficient than the source program: In fact, if the translation $\mathcal{A}$ were defined by $\mathcal{A}(M) = \mathit{unparse} \circ M \circ \mathit{parse}$, obviously there would be no improvement.

The target program being more efficient follows from the fact that the translation function $\mathcal{A}$ preserves the structure of the source program, with only replacing tree constructions with stream outputs, and case analyses on trees with stream inputs and case analyses on input tokens. More precisely, by inspecting the proof of soundness of the translation (which is available in the full version of the paper), we can observe:[2]

- When a closure is allocated in the execution of M (so that the heap space is consumed), the corresponding closure is allocated in the corresponding reduction step of $\mathcal{A}(M)$, and vice versa.
- When a function is called in the execution of M (so that the stack space is consumed), the corresponding function is called in the corresponding reduction step of $\mathcal{A}(M)$, and vice versa.
- When a case analysis on an input tree is performed in the execution of M, a token is read from the input stream and a case analysis on the token is performed in the corresponding reduction step of $\mathcal{A}(M)$.
- When a tree is constructed in the execution of M, the corresponding sequence of tokens is written on the output stream in the corresponding reduction steps of $A(M)$.

By the observation above, we can conclude:

- The memory space allocated by $\mathcal{A}(M)$ is less than the one allocated by $\mathit{unparse} \circ M \circ \mathit{parse}$, by the amount of the space for storing the intermediate trees output by *parse* and M (except for an implementation-dependent constant factor).
- The number of computation steps for running $\mathcal{A}(M)$ is the same as the one for running $\mathit{unparse} \circ M \circ \mathit{parse}$ (up to an implementation-dependent constant factor).

Thus, our translation is effective especially when the space for evaluating M is much smaller than the space for storing input and output trees.

We performed a preliminary experiment to support the argument above. Figure 14 shows the execution time and the total size of allocated heap memory

[2] To completely formalize these observations, we need to define another operational semantics that makes the heap and the stack explicit.

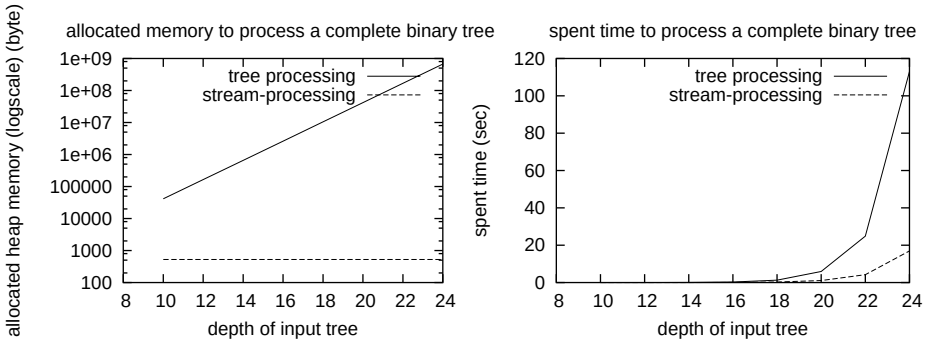


Fig. 14. Result of experiments. Inputs are binary trees whose height varies from 10 to 24. The experiment was performed on Sun Enterprise E4500/E5500, 400 MHz CPU, 13GB memory.

for the two versions of the program (both written manually in Objective Caml). As expected, the stream-processing program was more efficient. Especially, the heap memory size is constant for the stream-processing program while it is exponential in the depth of input trees for the tree-processing program.

5 Extensions

So far, we have focused on a minimal calculus to clarify the essence of our framework. This section shows how to extend the framework to be used in practice.

5.1 Constructs for storing trees on memory

By adding primitives for constructing and destructing trees on memory, we can allow programmers to selectively buffer input/output trees. Let us extend the syntax of the source and target languages as follows:

$$\begin{aligned} M &::= \cdots \mid \mathbf{mleaf}\ M \mid \mathbf{mnode}\ M_1\ M_2 \\ &\quad \mid\ (\mathbf{mcase}\ M\ \mathbf{of}\ \mathbf{mleaf}\ x \Rightarrow M_1 \mid \mathbf{mnode}\ x_1\ x_2 \Rightarrow M_2) \\ e &::= \cdots \mid \mathbf{mleaf}\ e \mid \mathbf{mnode}\ e_1\ e_2 \\ &\quad \mid\ (\mathbf{mcase}\ e\ \mathbf{of}\ \mathbf{mleaf}\ x \Rightarrow e_1 \mid \mathbf{mnode}\ x_1\ x_2 \Rightarrow e_2) \end{aligned}$$

Here, $\mathbf{mleaf}\ M$ and $\mathbf{mnode}\ M_1\ M_2$ are constructors of trees on memory and $\mathbf{mcase}\cdots$ is a destructor.

We also add type **MTree**, the type of trees stored on memory. The type system imposes no restriction on access order between variables of type **MTree** like type **Int** (so **MTree** is put in the ordinary type environment, not the ordered linear type environment). The translation algorithm $\mathcal{A}$ simply translates a source program, preserving the structure:

$$\begin{aligned} \mathcal{A}(\mathbf{mleaf}\ M) &= \mathbf{mleaf}\ \mathcal{A}(M) \\ \mathcal{A}(\mathbf{mnode}\ M_1\ M_2) &= \mathbf{mnode}\ \mathcal{A}(M_1)\ \mathcal{A}(M_2) \\ &\cdots \end{aligned}$$

```
fix strm_to_mem.λt.case t of leaf x ⇒ mleaf x
                          | node x1 x2 ⇒ mnode (strm_to_mem x1) (strm_to_mem x2)

fix mem_to_strm.λt.mcase t of mleaf x ⇒ leaf x
                          | mnode x1 x2 ⇒ node (mem_to_strm x1) (mem_to_strm x2)
```

Fig. 15. Definition of *strm_to_mem* and *mem_to_strm*

```
let mswap = fix f.λt.mcase t of mleaf x ⇒ leaf x | mnode x1 x2 ⇒ node (f x2) (f x1) in
    fix swap_deep.λn.λt.
        if n = 0 then
            mswap (strm_to_mem t)
        else
            case t of
                leaf x ⇒ leaf x
            | node x1 x2 ⇒ node (swap_deep (n − 1) x1) (swap_deep (n − 1) x2)
```

Fig. 16. A program which swaps children of nodes whose depth is more than n

These extensions are summarized in Appendix B.

With these primitives, a function *strm_to_mem*, which copies a tree from the input stream to memory, and *mem_to_strm*, which writes a tree on memory to the output stream, can be defined as shown in Figure 15.

Using the functions above, one can write a program that selectively buffers only a part of the input tree, while the type system guarantees that the selective buffering is correctly performed. For example, the program in Figure 16, which swaps children of nodes whose depth is more than n, only buffers the nodes whose depth is more than n.

The proof of Theorem 1 can be easily adapted for the extended language.

5.2 Side effects and multiple input trees

Since our translation algorithm preserves the structure of source programs, the translation works in the presence of side effects other than stream inputs/outputs.

Our framework can also be easily extended to deal with multiple input trees, by introducing pair constructors and refining the type judgment form to $\Gamma \mid \{s_1 : \Delta_1, \ldots, s_n : \Delta_n\} \vdash M : \tau$ where $s_1, \ldots, s_n$ are the names of input streams and each of $\Delta_1, \ldots, \Delta$ is an ordered linear type environment.

In Appendix C, we present examples of programs that take multiple input trees and use side effects.

5.3 Extention for dealing with XML

We discuss below how to extend our method to deal with XML documents.

The difference between binary trees and XML documents is that the latter ones (i) are rose trees and (ii) contain end tags that mark the end of sequences

in the stream format. The first point can be captured as the difference between the following types (we use ML-style type declarations):

```
datatype tree = leaf of int | node of tree*tree;
datatype xmltree = leaf of pcdata | node of label*attribute*treelist
and treelist = nil | cons of xmltree*treelist;
```

While the type `tree` represents binary trees, `xmltree` represents rose trees. Based on the difference between these types, we can replace the **case**-construct of the source language with the following two **case**-constructs.

$$\begin{array}{l}\textbf{caseElem}\ t\ \textbf{of}\ \textbf{leaf}(x) \Rightarrow M_1 \mid \textbf{node}(l, attr, tl) \Rightarrow M_2 \\ \textbf{caseSeq}\ tl\ \textbf{of}\ \textbf{nil} \Rightarrow M_1 \mid \textbf{cons}(x, xl) \Rightarrow M_2\end{array}$$

Typing rules can also be naturally extended. For example, the typing rule for the latter construct is:

$$\frac{\Gamma \mid \Delta_1 \vdash tl : \textbf{treelist} \qquad \Gamma \mid \Delta_2 \vdash M_1 : \tau \qquad \Gamma \mid x : \textbf{xmltree}, xl : \textbf{treelist}, \Delta_2 \vdash M_2 : \tau}{\Gamma \mid \Delta_1, \Delta_2 \vdash \textbf{caseSeq}\ tl\ \textbf{of}\ \textbf{nil} \Rightarrow M_1 \mid \textbf{cons}(x, xl) \Rightarrow M_2 : \tau}$$

The restriction on the access order is expressed by $x : \textbf{xmltree}, xl : \textbf{treelist}, \Delta_2$, as in T-Node.

The translation algorithm (1) maps the pattern **nil** in the source language to the pattern for closing tags. (2) prepares a stack and confirms well-formedness of input documents. Typing rules and the definition of the translation algorithm for these constructs are found in Appendix D.

6 Related Work

Nakano and Nishimura [17, 18] proposed a method for translating tree-processing programs to stream-processing programs using attribute grammars. In their method, programmers write XML processing with an attribute grammar. Then, the grammar is composed with parsing and unparsing grammars by using the descriptional composition [8] and translated to a grammar that directly deals with streams. *Quasi-SSUR condition* in [18] and *single use requirement* in [17], which force attributes of non-terminal symbols to be used at most once, seems to correspond to our linearity restriction on variables of tree types, but there seems to be no restriction that corresponds to our order restriction. As a result, their method can deal with programs (written as attribute grammars) that violate the order restriction of our type system (although in that case, generated stream-processing programs store a part of trees in memory, so that the translation may not improve the efficiency). On the other hand, an advantage of our method is that programs are easier to read and write since one can write programs as ordinary functional programs except for the restriction imposed by the type system, rather than as attribute grammars. Another advantage of our method is that we can deal with source programs that involve side-effects (e.g. programs that

```
N → node N1 N2
        N1.inh = f1 N.inh
        N2.inh = f2 N.inh N1.syn N1.inh
        N.syn = f3 N.inh N1.syn N1.inh N2.syn N2.inh
N → leaf i
        N.syn = f4 N.inh i

fix f.λinh.λt.case t of
        leaf x ⇒ f4 inh x
        node x1 x2 ⇒
                let N1.inh = f1 inh in
                let N1.syn = f N1.inh x1 in
                let N2.inh = f2 N.inh N1.syn N1.inh in
                let N2.syn = f N2.inh x2 in
                        f3 N.inh N1.syn N1.inh N2.syn N2.inh
```

Fig. 17. L-attributed grammar over binary trees and corresponding program.

```
N → node N1 N2
        N1.depth = N.depth + 1
        N2.depth = N.depth + 1
        N.result = node N1.result N2.result
N → leaf i
        if N.depth mod 2 = 0 then
                N.result = leaf (i + 1)
        else
                N.result = leaf i
```

Fig. 18. L-attributed grammar that corresponds to *inc_alt* in Figure 8.

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).

The class of well-typed programs in our language seems to be closely related to the class of L-attributed grammars [1]. In fact, any L-attributed grammar over the binary tree can be expressed as a program as shown in Figure 17. If output trees are not used in attributes, the program is well-typed. Conversely, any program that is well-typed in our language seems to be definable as an L-attribute grammar. The corresponding attribute grammar may, however, be awkward, since one has to encode control information into attributes. For example, the attribute grammar corresponding to *inc_alt* is shown in Figure 18.

There are other studies on translation of tree-processing programs into stream-processing programs. Some of them [2, 9, 10] deal with XPath expressions [4, 21] and others [15] deal with XQuery [5, 7]. Those translations are more aggressive than ours in the sense that the structure of source programs is changed so that input trees can be processed in one path. On the other hand, their target lan-

guages (XPath and XQuery languages) are restricted in the sense that they do not contain functions and side-effects.

There are many studies on program transformation [16, 23] for eliminating intermediate data structures of functional programs, known as deforestation or fusion. Although the goal of our translation is also to remove intermediate data structures from $\mathit{unparse} \circ f \circ \mathit{parse}$, the previous methods are not directly applicable since those methods do not guarantee that transformed programs access inputs in a stream-processing manner. In fact, *swap* in Figure 2, which violates the access order, can be expressed as a treeless program [23] or a catamorphism [16], but the result of deforestation is not an expected stream-processing program.

Actually, there are many similarities between the restriction of treeless program [23] and that of our type system. In treeless programs, (1) variables have to occur only once, and (2) only variables can be passed to functions. (1) corresponds to the linearity restriction of our type system. (2) is the restriction for prohibiting trees generated in programs to be passed to functions, which corresponds to the restriction that functions cannot take values of type $\mathbf{Tree}^+$ in our type system. The main difference is that:

- Our type system additionally imposes a restriction on the access order. This is required to guarantee that translated programs read input streams sequentially.
- We restrict programs with a type system, while the restriction on treeless programs is syntactic. Our type-based approach enables us to deal with higher-order functions. The type-based approach is also useful for automatic inference of selective buffering of trees, as discussed in Section 7.

The type system we used in this paper is based on the ordered linear logic proposed by Polakow [20]. He proposed a logic programming language Olli and logical framework OLF based on the logic. There are many similarities between our typing rules and his derivation rules for the ordered linear logic. For example, our type judgment $\Gamma \mid \Delta \vdash M : \tau$ corresponds to the judgment $\Gamma; \cdot; \Delta \vdash A$ of ordered linear logic. The rule T-Abs1 corresponds to a combination of the rules for an ordered linear implication and the modality (!). There is not, however, a complete correspondence, probably mainly because of the call-by-value semantics of our language. Petersen et al. [19] used ordered linear types to guarantee correctness of memory allocation and data layout. While they used an ordered linear type environment to express a spatial order, we used it to express a temporal order.

7 Conclusion

We have proposed a type system based on ordered linear types to enable translation of tree-processing programs into stream-processing programs, and proved the correctness of the translation.

As we stated in Section 3 and Section 5, one can write tree-processing programs that selectively skip and/or buffer trees by using *skip_tree*, *copy_tree*, *strm_to_mem* and *mem_to_strm*. However, inserting those functions by hand is sometimes tedious. We are currently studying a type-directed, source-to-source translation for automatically inserting these functions. For example, the rules for *skip_tree*, *copy_tree* and *strm_to_mem* are:

$$\frac{\Gamma \mid \Delta \vdash M \Longrightarrow M' : \tau}{\Gamma \mid x : \mathbf{Tree}^{-}, \Delta \vdash M \Longrightarrow \mathit{skip\_tree}(x); M' : \tau} \quad \text{(Tr-Skip)}$$

$$\frac{\Gamma \mid \Delta \vdash M \Longrightarrow M' : \mathbf{Tree}^{-}}{\Gamma \mid \Delta \vdash M \Longrightarrow \mathit{copy\_tree}(M') : \mathbf{Tree}^{+}} \quad \text{(Tr-Copy)}$$

$$\frac{\Gamma, x : \mathbf{MTree} \mid \Delta \vdash M \Longrightarrow M' : \tau}{\Gamma \mid x : \mathbf{Tree}^{-}, \Delta \vdash M \Longrightarrow \mathbf{let}\ x\ =\ \mathit{strm\_to\_mem}(x)\ \mathbf{in}\ M' : \tau} \quad \text{(Tr-StrToMem)}$$

The relation $\Gamma \mid \Delta \vdash M \Longrightarrow M' : \tau$ means that the source program M can be translated to the equivalent source program M' which uses *skip_tree*, *strm_to_mem*, etc. such that $\Gamma \mid \Delta \vdash M' : \tau$.

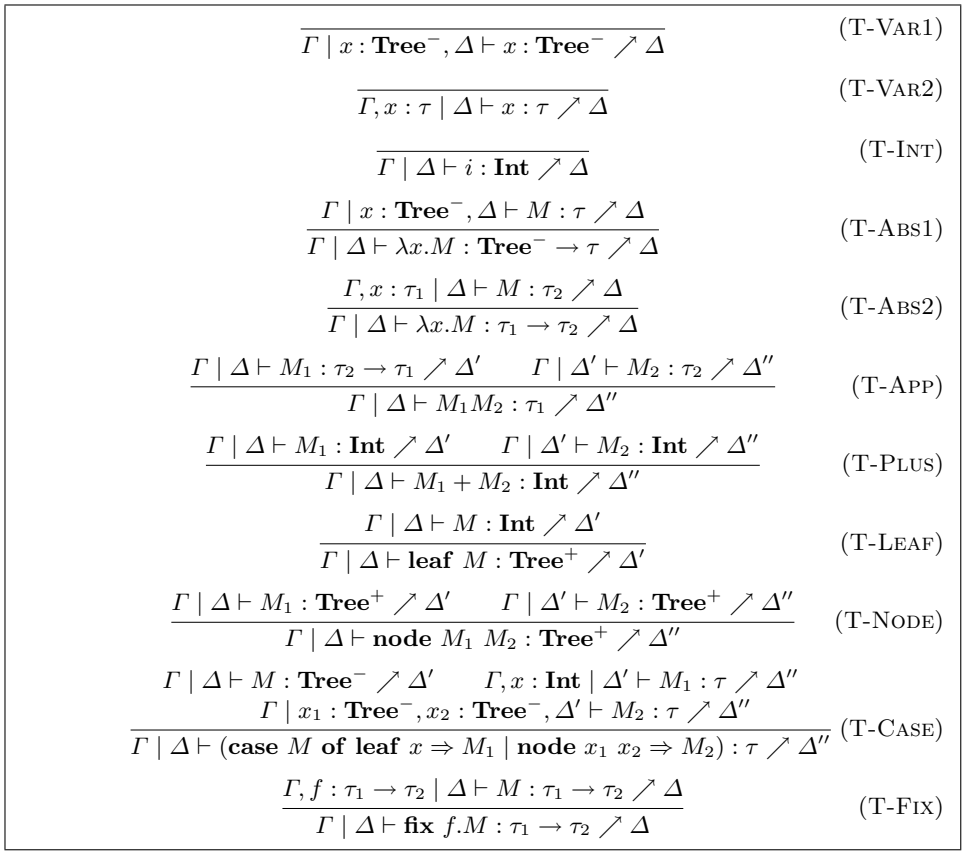
In addition to application to XML processing, our translation framework may also be useful for optimization of distributed programs that process and communicate complex data structures. Serialization/unserialization of data correspond to unparsing/parsing in Figure 1, so that our translation framework can be used for eliminating intermediate data structures and processing serialized data directly.

Acknowledgement We thank members of "Programming Language Principles" group at University of Tokyo and Tokyo Institute of Technology.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers*. Addison-Wesley Pub Co, 1986.
2. Iliana Avila-Campillo, Todd J. Green, Ashish Gupta, Makoto Onizuka, Demian Raven, and Dan Suciu. XMLTK: An XML toolkit for scalable XML stream processing. In *Proceedings of PLAN-X*, October 2002.
3. Henry G. Baker. Lively linear lisp – look ma, no garbage! *ACM SIGPLAN Notices*, 27(8):89–98, 1992.
4. Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Simeon. *XML Path Language (XPath) 2.0*. World Wide Web Consortium, November 2003. `http://www.w3.org/TR/xpath20/`.
5. Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. *XQuery 1.0: An XML Query Language*. World Wide Web Consortium, November 2003. `http://www.w3.org/TR/xquery/`.

6. 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`.
7. Don Chamberlin, Peter Frankhauser, Massimo Marchiori, and Jonathan Robie. *XML Query (XQuery) Requirements*. World Wide Web Consortium, November 2003. `http://www.w3.org/TR/xquery-requirements/`.
8. Harald Ganzinger and Robert Giegerich. Attribute coupled grammars. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, 1984.
9. T. Green, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. Technical report, University of Washington, 2001.
10. Ashish Kumar Gupta and Dan Suciu. Stream processing of XPath queries with predicates. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, jun 2003.
11. Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.
12. Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology (TOIT)*, 3(2):117–148, 2003.
13. Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 331–342, 2002.
14. Koichi Kodama, Kohei Suenaga, and Naoki Kobayashi. Translation of tree-processing programs into stream-processing programs based on ordered linear type. Available from `http://www.yl.is.s.u-tokyo.ac.jp/~kohei/doc/paper/translation.pdf`.
15. Bertram Ludäscher, Pratik Mukhopadhyay, and Yannis Papakonstantinou. A transducer-based XML query processor. In *Proceedings of the 28th International Conference on Very Large Data Bases*, 2002.
16. Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 124 – 144, 1991.
17. Keisuke Nakano. Composing stack-attributed tree transducers. Technical Report METR–2004–01, Department of Mathematical Informatics, University of Tokyo, Japan, 2004.
18. 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.
19. Leaf Petersen, Robert Harper, Karl Crary, and Frank Pfenning. A type theory for memory allocation and data layout. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2003.
20. Jeff Polakow. *Ordered linear logic and applications*. PhD thesis, Carnegie Mellon University, June 2001. Available as Technical Report CMU-CS-01-152.
21. Mark Scardina and Mary Fernandez. *XPath Requirements Version 2.0*. World Wide Web Consortium, August 2003. `http://www.w3.org/TR/xpath20req/`.
22. David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Proceedings of Functional Programming Languages and Computer Architecture*, pages 1–11, San Diego, California, 1995.
23. P. Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP '88. European Symposium on Programming, Nancy, France, 1988 (Lecture Notes in Computer Science, vol. 300)*, pages 344–358. Berlin: Springer-Verlag, 1988.



$$\frac{}{\Gamma \mid x : \mathbf{Tree}^{-}, \Delta \vdash x : \mathbf{Tree}^{-} \nearrow \Delta} \quad \text{(T-Var1)}$$

$$\frac{}{\Gamma, x : \tau \mid \Delta \vdash x : \tau \nearrow \Delta} \quad \text{(T-Var2)}$$

$$\frac{}{\Gamma \mid \Delta \vdash i : \mathbf{Int} \nearrow \Delta} \quad \text{(T-Int)}$$

$$\frac{\Gamma \mid x : \mathbf{Tree}^{-}, \Delta \vdash M : \tau \nearrow \Delta}{\Gamma \mid \Delta \vdash \lambda x.M : \mathbf{Tree}^{-} \to \tau \nearrow \Delta} \quad \text{(T-Abs1)}$$

$$\frac{\Gamma, x : \tau_1 \mid \Delta \vdash M : \tau_2 \nearrow \Delta}{\Gamma \mid \Delta \vdash \lambda x.M : \tau_1 \to \tau_2 \nearrow \Delta} \quad \text{(T-Abs2)}$$

$$\frac{\Gamma \mid \Delta \vdash M_1 : \tau_2 \to \tau_1 \nearrow \Delta' \qquad \Gamma \mid \Delta' \vdash M_2 : \tau_2 \nearrow \Delta''}{\Gamma \mid \Delta \vdash M_1 M_2 : \tau_1 \nearrow \Delta''} \quad \text{(T-App)}$$

$$\frac{\Gamma \mid \Delta \vdash M_1 : \mathbf{Int} \nearrow \Delta' \qquad \Gamma \mid \Delta' \vdash M_2 : \mathbf{Int} \nearrow \Delta''}{\Gamma \mid \Delta \vdash M_1 + M_2 : \mathbf{Int} \nearrow \Delta''} \quad \text{(T-Plus)}$$

$$\frac{\Gamma \mid \Delta \vdash M : \mathbf{Int} \nearrow \Delta'}{\Gamma \mid \Delta \vdash \mathbf{leaf}\ M : \mathbf{Tree}^{+} \nearrow \Delta'} \quad \text{(T-Leaf)}$$

$$\frac{\Gamma \mid \Delta \vdash M_1 : \mathbf{Tree}^{+} \nearrow \Delta' \qquad \Gamma \mid \Delta' \vdash M_2 : \mathbf{Tree}^{+} \nearrow \Delta''}{\Gamma \mid \Delta \vdash \mathbf{node}\ M_1\ M_2 : \mathbf{Tree}^{+} \nearrow \Delta''} \quad \text{(T-Node)}$$

$$\frac{\begin{array}{c}\Gamma \mid \Delta \vdash M : \mathbf{Tree}^{-} \nearrow \Delta' \qquad \Gamma, x : \mathbf{Int} \mid \Delta' \vdash M_1 : \tau \nearrow \Delta'' \\ \Gamma \mid x_1 : \mathbf{Tree}^{-}, x_2 : \mathbf{Tree}^{-}, \Delta' \vdash M_2 : \tau \nearrow \Delta''\end{array}}{\Gamma \mid \Delta \vdash (\mathbf{case}\ M\ \mathbf{of\ leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2) : \tau \nearrow \Delta''} \quad \text{(T-Case)}$$

$$\frac{\Gamma, f : \tau_1 \to \tau_2 \mid \Delta \vdash M : \tau_1 \to \tau_2 \nearrow \Delta}{\Gamma \mid \Delta \vdash \mathbf{fix}\ f.M : \tau_1 \to \tau_2 \nearrow \Delta} \quad \text{(T-Fix)}$$

Fig. 19. Rules for Type Checking

A Type Checking Algorithm

The relation $\Gamma \mid \Delta \vdash M : \tau \nearrow \Delta'$ mentioned in Section 3.4 is defined as the least relation closed under the rules in Figure 19.

B Extended Languages, Type System and Translation Algorithm

Figure 20 shows the extended languages, type system and translation algorithm for selective buffering of input trees.

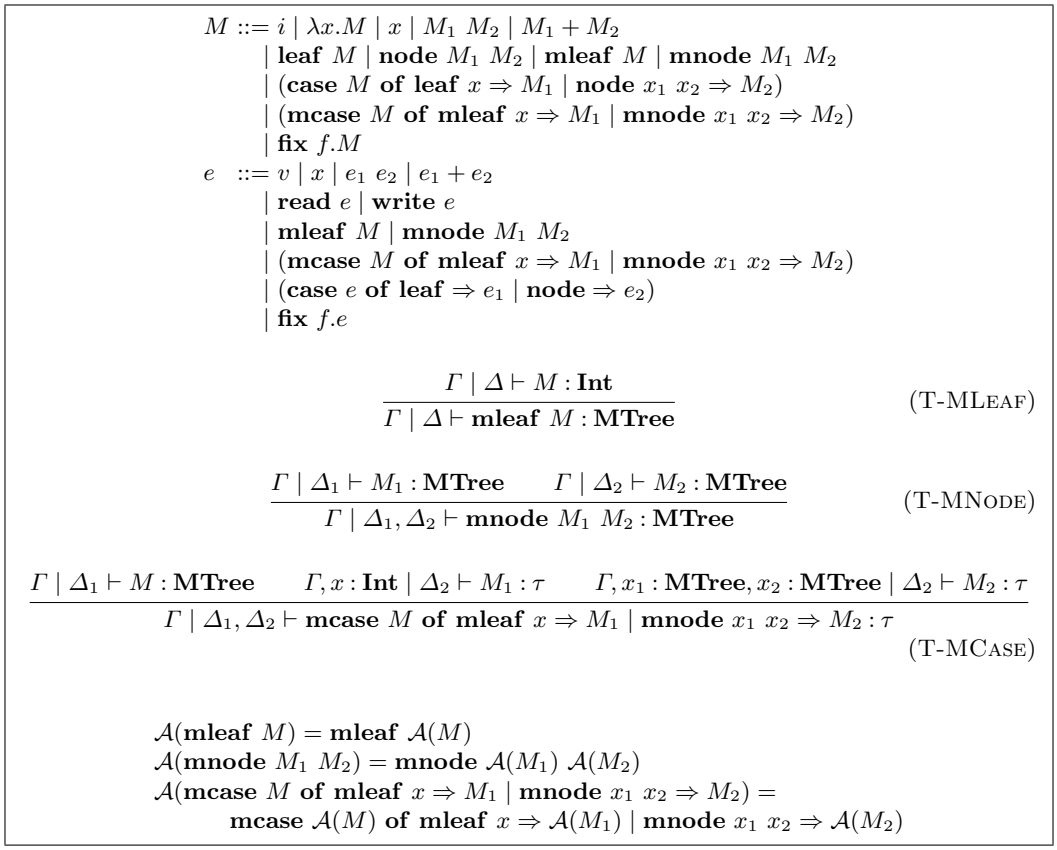


Fig. 20. Extended languages, type system and translation algorithm

C Examples of programs that use side effects and take multiple input trees

Figure 21 shows a program that prints out integer elements to standard error output in right-to-left, depth-first manner.

Figure 22 shows a program that takes two input trees and returns whether they are identical. Note that the structure-preserving translation works.

D Extensions for XML processing

Figure 23 shows typing rules and the definition of $\mathcal{A}$ for XML processing constructs we mentioned in Section 5. **readtoken**() reads an element from the input stream where the element is either **start_tag**$(l, attr)$, **end_tag**(l), **pcdata**(x) or **end_of_file**. The primitive **unread**() cancels the previous read operation. Values of type **attribute** are stored on memory like values of type **pcdata**, because attributes can occur in an arbitrary order (e.g., `<a b=''foo'' c=''baa''>` and `<a`

Source program

```
let f' = fix f'.λt.mcase t of mleaf x ⇒ print_err x | mnode x1 x2 ⇒ (f' x2);(f' x1) in
    fix f.λt.case t of
        leaf x ⇒ print_err x
    | node x1 x2 ⇒ let mt = strm_to_mem x1 in (f x2);(f' mt)
```

Target program

```
let f' = fix f'.λt.mcase t of mleaf x ⇒ print_err x | mnode x1 x2 ⇒ (f' x2);(f' x1) in
    fix f.λt.case read () of
        leaf ⇒ let x = read() in print_err x
        | node ⇒ let mt = strm_to_mem () in (f ());(f' mt)
```

Fig. 21. A program that prints out integer elements to standard error output in right-to-left, depth-first manner.

Source program

```
fix eq.λ(t1, t2).
    case t1 of
        leaf x ⇒ (case t2 of leaf y ⇒ x = y | node x1 x2 ⇒ false)
    | node x1 x2 ⇒
        (case t2 of leaf y ⇒ false | node y1 y2 ⇒ eq (x1, y1) && eq (x2, y2))
```

Target program

```
fix eq.λ(t1, t2).
    case read(t1) of
      leaf ⇒ let x = read(t1)in
                (case read(t2) of leaf ⇒ let y = read(t2) in x = y | node ⇒ false)
    | node ⇒ (case read(t2) of leaf ⇒ let y = read(t2) in false
                             |  node ⇒ eq (t1, t2) && eq (t1, t2))
```

Fig. 22. A program that takes two input trees and returns whether they are identical

`c=''baa'' b=''foo''>` have the same meaning). Thus, type **attribute** is put in the ordinary type environment , not in the ordered linear type environment, in the rule T-CaseElem.

In target programs, a stack is used to check well-formedness of input documents. When a start tag is read, the tag is pushed on the stack. When an end tag is read, it is compared with the tag stored on the top of the stack.

Note that the pattern **nil** in the source language is translated to the pattern for closing tags.

E The Proof of Theorem 2

In this section, we prove Theorem 2. We prepare the following lemma to prove the theorem.

$$\frac{\Gamma \mid \Delta_1 \vdash M : \mathbf{xmltree} \qquad \Gamma, x : \mathbf{pcdata} \mid \Delta_2 \vdash M_2 : \tau \qquad \Gamma, l : \mathbf{label}, attr : \mathbf{attribute} \mid tl : \mathbf{treelist}, \Delta_2 \vdash M_1 : \tau}{\Gamma \mid \Delta_1, \Delta_2 \vdash \mathbf{caseElem}\ M\ \mathbf{of}\ \mathbf{node}(l, attr, tl)\ \Rightarrow\ M_1\ \mid\ \mathbf{leaf}(x)\ \Rightarrow\ M_2 : \tau}\ \text{(T-CaseElem)}$$

$$\frac{\Gamma \mid \Delta_1 \vdash tl : \mathbf{treelist} \qquad \Gamma \mid \Delta_2 \vdash M_1 : \tau \qquad \Gamma \mid x : \mathbf{xmltree}, xl : \mathbf{treelist}, \Delta_2 \vdash M_2 : \tau}{\Gamma \mid \Delta_1, \Delta_2 \vdash \mathbf{caseSeq}\ tl\ \mathbf{of}\ \mathbf{nil} \Rightarrow M_1 \mid \mathbf{cons}(x, xl) \Rightarrow M_2 : \tau}\ \text{(T-CaseSeq)}$$

$$\begin{array}{l}
\mathcal{A}(\mathbf{caseElem}\ M\ \mathbf{of}\ \mathbf{node}(l, attr, tl)\ \Rightarrow\ M_1\ \mid\ \mathbf{leaf}(x)\ \Rightarrow\ M_2) = \\
\quad \mathbf{case}\ \mathbf{readtoken}()\ \mathbf{of} \\
\qquad \mathbf{start_tag}(l, attr)\ \Rightarrow\ \mathbf{push}(l); [()/tl]\mathcal{A}(M_1) \\
\qquad \mid\ \mathbf{pcdata}(x)\ \Rightarrow\ \mathcal{A}(M_2) \\
\qquad \mid\ _\ \Rightarrow\ \mathbf{raise}\ \mathit{IllFormedException}\ (*\ \text{end_tag or end of file}\ *) \\
\\
\mathcal{A}(\mathbf{caseSeq}\ M\ \mathbf{of}\ \mathbf{nil}\ \Rightarrow\ M_1\ \mid\ \mathbf{cons}(x, xs)\ \Rightarrow\ M_2) = \\
\quad \mathbf{case}\ \mathbf{readtoken}()\ \mathbf{of} \\
\qquad \mathbf{end_tag}(l)\ \Rightarrow\ \mathbf{let}\ l' = \mathbf{pop}()\ \mathbf{in} \\
\qquad\qquad\qquad \mathbf{if}\ l = l'\ \mathbf{then}\ \mathcal{A}(M_1)\ \mathbf{else}\ \mathbf{raise}\ \mathit{IllFormedException} \\
\qquad \mid\ _\ \Rightarrow\ (\mathbf{unread}(); [()/x, ()/xs]\mathcal{A}(M_2))
\end{array}$$

Fig. 23. Typing rules and definition of translation for XML processing constructs

Lemma 3 (type substitution). *If* $\Gamma, x : \tau' \mid \Delta \vdash M : \tau$ *and* $\Gamma \mid \emptyset \vdash N : \tau'$ *hold,* $\Gamma \mid \Delta \vdash [N/x]M : \tau$.

Proof. We use induction on the derivation tree of $\Gamma, x : \tau' \mid \Delta \vdash M : \tau$.

- Case T-Var1:
  $M = z$
  $\Delta = z : \mathbf{Tree}^-$
  Since $[N/x]z = z$, we have $\Gamma \mid \Delta \vdash [N/x]M : \tau$.
- Case T-Var2:
  $M = z$
  $z : \tau' \in \{\Gamma, x : \tau'\}$
  $\Delta = \emptyset$
  If $z = x$, then $[N/x]M = N$ and $\tau = \tau'$. Thus, $\Gamma \mid \Delta \vdash [N/x]M : \tau$ holds from the assumption $\Gamma \mid \emptyset \vdash N : \tau'$. If $z \neq x$, then $[N/x]M = z$. Thus, $\Gamma \mid \Delta \vdash [N/x]M : \tau$ holds.
- Case T-Int:
  $M = i$
  $\tau = \mathbf{Int}$,
  $\Delta = \emptyset$
  Since $[N/x]M = M$, we have $\Gamma \mid \Delta \vdash [N/x]M : \tau$.
- Case T-Abs1:
  $M = \lambda z.M_2$
  $\tau = \mathbf{Tree}^- \to \tau_2$

$\Gamma, x : \tau' \mid z : \mathbf{Tree}^- \vdash M_2 : \tau_2$
$\Delta = \emptyset$

$\Gamma \mid z : \mathbf{Tree}^- \vdash [N/x]M_2 : \tau_2$ follows from the induction hypothesis. Thus, $\Gamma \mid \emptyset \vdash \lambda z.[N/x]M_2 : \mathbf{Tree}^- \to \tau_2$ follows from T-Abs1. Because $[N/x]M = \lambda z.[N/x]M_2$ (Note that we assume that every bound variable be different as we mentioned in Section 2.1), we have $\Gamma \mid \Delta \vdash [N/x]M : \tau$ as required.

– Case T-Abs2:

$M = \lambda z.M_1$
$\tau = \tau_1 \to \tau_2$
$\Gamma, x : \tau', z : \tau_1 \mid \emptyset \vdash M_1 : \tau_2$
$\Delta = \emptyset$

$\Gamma, z : \tau_1 \mid \emptyset \vdash [N/x]M_1 : \tau_2$ follows from the induction hypothesis. Thus, $\Gamma \mid \emptyset \vdash \lambda z.[N/x]M_2 : \tau_1 \to \tau_2$ follows from T-Abs2. Because $[N/x]M = \lambda z.[N/x]M_2$, we have $\Gamma \mid \Delta \vdash [N/x]M : \tau$ as required.

– Case T-App:

$M = M_1 M_2$
$\Gamma, x : \tau' \mid \Delta_1 \vdash M_1 : \tau_2 \to \tau$
$\Gamma, x : \tau' \mid \Delta_2 \vdash M_2 : \tau_2$
$\Delta = \Delta_1, \Delta_2$

$\Gamma \mid \Delta_1 \vdash [N/x]M_1 : \tau_2 \to \tau$ and $\Gamma \mid \Delta_2 \vdash [N/x]M_2 : \tau_2$ follow from the induction hypothesis. Because $([N/x]M_1\ [N/x]M_2) = [N/x](M_1\ M_2)$, we have $\Gamma \mid \Delta \vdash [N/x]M : \tau$ from T-App as required.

– Case T-Plus:

$M = M_1 + M_2$
$\tau = \mathbf{Int}$
$\Gamma, x : \tau' \mid \Delta_1 \vdash M_1 : \mathbf{Int}$
$\Gamma, x : \tau' \mid \Delta_2 \vdash M_2 : \mathbf{Int}$
$\Delta = \Delta_1, \Delta_2$

$\Gamma \mid \Delta_1 \vdash [N/x]M_1 : \mathbf{Int}$ and $\Gamma \mid \Delta_2 \vdash [N/x]M_2 : \mathbf{Int}$ follow from the induction hypothesis. Because $[N/x]M_1 + [N/x]M_2 = [N/x](M_1 + M_2)$, we have $\Gamma \mid \Delta \vdash [N/x]M : \tau$ from T-Plus.

– Case T-Leaf:

$M = \mathbf{leaf}\ M_1$
$\tau = \mathbf{Tree}^+$
$\Gamma, x : \tau' \mid \Delta \vdash M_1 : \mathbf{Int}$

$\Gamma \mid \Delta \vdash [N/x]M_1 : \mathbf{Int}$ follows from the induction hypothesis. Because $\mathbf{leaf}\ [N/x]M_1 = [N/x](\mathbf{leaf}\ M_1)$, we have $\Gamma \mid \Delta \vdash [N/x]M : \tau$ from T-Leaf.

– Case T-Node:

$M = \mathbf{node}\ M_1\ M_2$
$\tau = \mathbf{Tree}^+$
$\Gamma, x : \tau' \mid \Delta_1 \vdash M_1 : \mathbf{Tree}^+$
$\Gamma, x : \tau' \mid \Delta_2 \vdash M_2 : \mathbf{Tree}^+$
$\Delta = \Delta_1, \Delta_2$

$\Gamma \mid \Delta_1 \vdash [N/x]M_1 : \mathbf{Tree}^+$ and $\Gamma \mid \Delta_2 \vdash [N/x]M_2 : \mathbf{Tree}^+$ follow from the induction hypothesis. Because $\mathbf{node}\ [N/x]M_1\ [N/x]M_2 = [N/x](\mathbf{node}\ M_1\ M_2)$, we have $\Gamma \mid \Delta \vdash [N/x]M : \tau$ from T-Node.

– Case T-Case:
$$M = \mathbf{case}\ M_0\ \mathbf{of\ leaf}\ z \Rightarrow M_1 \mid \mathbf{node}\ z_1\ z_2 \Rightarrow M_2$$
$$\Gamma, x : \tau' \mid \Delta_1 \vdash M_0 : \mathbf{Tree}^-$$
$$\Gamma, x : \tau', z : \mathbf{Int} \mid \Delta_1 \vdash M_1 : \tau$$
$$\Gamma, x : \tau' \mid z_1 : \mathbf{Tree}^-, z_2 : \mathbf{Tree}^-, \Delta_2 \vdash M_2 : \tau$$
$$\Delta = \Delta_1, \Delta_2$$
$\Gamma \mid \Delta_1 \vdash [N/x]M_0 : \mathbf{Tree}^-$ and $\Gamma, z : \mathbf{Int} \mid \Delta_2 \vdash [N/x]M_1 : \tau$ and $\Gamma \mid z_1 : \mathbf{Tree}^-, z_2 : \mathbf{Tree}^-, \Delta_2 \vdash M_2 : \tau$ follow from the induction hypothesis. Because $\mathbf{case}\ [N/x]M_0\ \mathbf{of\ leaf}\ z \Rightarrow [N/x]M_1 \mid \mathbf{node}\ z_1\ z_2 \Rightarrow [N/x]M_2 = [N/x]M$, we have $\Gamma \mid \Delta \vdash [N/x]M : \tau$ from T-Case.
– Case T-Fix:
$$M = \mathbf{fix}\ f.M_1,$$
$$\tau = \tau_1 \to \tau_2$$
$$\Gamma, x : \tau', f : \tau_1 \to \tau_2 \mid \Delta \vdash M_1 : \tau_1 \to \tau_2$$
$\Gamma, f : \tau_1 \to \tau_2 \mid \Delta \vdash [N/x]M_1 : \tau_1 \to \tau_2$ follows from the induction hypothesis. Because $\mathbf{fix}\ f.[N/x]M_1 = [N/x](\mathbf{fix}\ f.M_1)$, we have $\Gamma \mid \Delta \vdash [N/x]M : \tau$ from T-Fix.

□

Proof of Theorem 2. The first condition can be easily proved by induction on the derivation tree of $\Gamma \mid \langle\langle\delta\rangle\rangle \vdash M : \tau$. Here we only show the proof of the second condition.

From the assumption $(M, \delta) \longrightarrow (M', \delta')$, there exist E_s and I that satisfy $M = E_s[I]$. We use structural induction on E_s.

– Case $E_s = [\,]$.

- Case $I = i_1 + i_2$.
  $\langle\langle\delta\rangle\rangle = \emptyset$ and $\tau = \mathbf{Int}$ follow from the assumption $\Gamma \mid \langle\langle\delta\rangle\rangle \vdash M : \tau$ and T-Plus and T-Int. $M' = \mathit{plus}(i_1, i_2)$ and $\delta' = \delta$ (and thus, $\delta' = \emptyset$) follow from Es2-Plus. Thus, $\Gamma \mid \langle\langle\delta'\rangle\rangle \vdash \mathit{plus}(i_1, i_2) : \mathbf{Int}$ holds from T-Int as required.
- Case $I = (\lambda x.N)\ U$.
  First, suppose $U = i$ or $U = \lambda y.N'$ for some i or y and N'. Then, $\delta = \emptyset$ and $\Gamma \mid \emptyset \vdash U : \tau'$ and $\Gamma, x : \tau' \mid \emptyset \vdash N : \tau$ follow from the assumption $\Gamma \mid \langle\langle\delta\rangle\rangle \vdash M : \tau$ and T-App and T-Abs2. $M' = [U/x]N$ and $\delta' = \delta$ (and thus, $\delta' = \emptyset$) follow from Es2-App. Thus, $\Gamma \mid \langle\langle\delta'\rangle\rangle \vdash M' : \tau$ follows from Lemma 3 as required.
  Next, suppose $U = y$ for some y. Then, $M' = [y/x]N$ and $\delta' = \delta$ follow from Es2-App. $\langle\langle\delta\rangle\rangle = y : \mathbf{Tree}^-$ and $\Gamma \mid x : \mathbf{Tree}^- \vdash N : \tau$ follow from the assumption $\Gamma \mid \langle\langle\delta\rangle\rangle \vdash M : \tau$ and T-App and T-Abs1. Thus, as easily seen, $\Gamma \mid y : \mathbf{Tree}^- \vdash [y/x]N : \tau$. Thus, $\Gamma \mid \langle\langle\delta'\rangle\rangle \vdash M' : \tau$ follows as required.
- Case $I = (\mathbf{case}\ y\ \mathbf{of\ leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2)$ with $\delta = (y \mapsto \mathbf{leaf}\ i, \delta'')$.
  $\Gamma \mid \langle\langle\delta\rangle\rangle \vdash M : \tau$ must have been derived by using T-Case. Thus,

we have $\Gamma, x : \mathbf{Int} \mid \langle\langle \delta'' \rangle\rangle \vdash M_1 : \tau$. Because $\Gamma \mid \emptyset \vdash i : \mathbf{Int}$, we have $\Gamma \mid \langle\langle \delta'' \rangle\rangle \vdash [i/x]M_1 : \tau$ from Lemma 3. Because $M' = [i/x]M_1$ and $\delta' = \delta''$ follow from Es2-Case1, we have $\Gamma \mid \langle\langle \delta' \rangle\rangle \vdash M' : \tau$ as required.

- Case $I = (\mathbf{case}\ y\ \mathbf{of\ leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2)$ with $\delta = (y \mapsto (\mathbf{node}\ V_1\ V_2), \delta'')$.
  $\Gamma \mid \langle\langle \delta \rangle\rangle \vdash M : \tau$ must have been derived by using T-Case. Thus, we have $\Gamma \mid x_1 : \mathbf{Tree}^-, x_2 : \mathbf{Tree}^-, \langle\langle \delta'' \rangle\rangle \vdash M_2 : \tau$. Because $M' = M_2$ and $\delta' = x_1 \mapsto V_1, x_2 \mapsto V_2, \delta''$ follow from Es2-Case2, we have $\Gamma \mid \langle\langle \delta' \rangle\rangle \vdash M' : \tau$ as required.
- Case $I = \mathbf{fix}\ f.N$.
  $\Gamma \mid \langle\langle \delta \rangle\rangle \vdash M : \tau$ must have been derived by using T-Fix. Thus, we have $\delta = \emptyset$ and $\Gamma, f : \tau \mid \emptyset \vdash N : \tau$. $M' = [\mathbf{fix}\ f.N/f]N$ and $\delta' = \delta$ follow from Es2-Fix. Because $\Gamma \mid \emptyset \vdash \mathbf{fix}\ f.N : \tau$, we have $\Gamma \mid \emptyset \vdash [\mathbf{fix}\ f.N/f]N : \tau$ from Lemma 3. Thus, we have $\Gamma \mid \langle\langle \delta' \rangle\rangle \vdash M' : \tau$ as required.

– Case $E_s = E'_s\ N$.
$\Gamma \mid \langle\langle \delta \rangle\rangle \vdash M : \tau$ must have been derived by using T-App. Thus, we have $\Gamma \mid \langle\langle \delta_1 \rangle\rangle \vdash E'_s[I] : \tau' \to \tau$ and $\Gamma \mid \langle\langle \delta_2 \rangle\rangle \vdash N : \tau'$ and $\delta = \delta_1, \delta_2$ for some δ_1, δ_2 and τ'. From the induction hypothesis, there exist δ'_1 and M'' that satisfy $\Gamma \mid \langle\langle \delta'_1 \rangle\rangle \vdash M'' : \tau' \to \tau$ and $(E'_s[I], \delta_1) \longrightarrow (M'', \delta'_1)$. Because $M' = M''\ N$ and $\delta' = \delta'_1, \delta_2$, we have $\Gamma \mid \langle\langle \delta' \rangle\rangle \vdash M' : \tau$ from T-App as required.

– Case $E_s = (\lambda x.N)E'_s$.
$\Gamma \mid \langle\langle \delta \rangle\rangle \vdash M : \tau$ must have been derived by using T-App. Thus, we have $\Gamma \mid \emptyset \vdash \lambda x.N : \tau' \to \tau$ and $\Gamma \mid \langle\langle \delta \rangle\rangle \vdash E'_s[I] : \tau'$ for some τ'. From the induction hypothesis, there exists M'' that satisfies $\Gamma \mid \langle\langle \delta' \rangle\rangle \vdash M'' : \tau'$ and $(E'_s[I], \delta) \longrightarrow (M'', \delta')$. Because $M' = (\lambda x.N)\ M''$, we have $\Gamma \mid \langle\langle \delta' \rangle\rangle \vdash M' : \tau$ from T-App as required.

– Case $E_s = E'_s + N$.
$\Gamma \mid \langle\langle \delta \rangle\rangle \vdash M : \tau$ must have been derived by using T-Plus. Thus, we have $\tau = \mathbf{Int}$ and $\Gamma \mid \langle\langle \delta_1 \rangle\rangle \vdash E'_s[I] : \mathbf{Int}$ and $\Gamma \mid \langle\langle \delta_2 \rangle\rangle \vdash N : \mathbf{Int}$ and $\delta = \delta_1, \delta_2$ for some δ_1, δ_2. From the induction hypothesis, there exist δ'_1 and M'' that satisfy $\Gamma \mid \langle\langle \delta'_1 \rangle\rangle \vdash M'' : \mathbf{Int}$ and $(E'_s[I], \delta_1) \longrightarrow (M'', \delta'_1)$. Because $M' = M'' + N$ and $\delta' = \delta'_1, \delta_2$, we have $\Gamma \mid \langle\langle \delta' \rangle\rangle \vdash M' : \tau$ from T-App as required.

– Case $E_s = i + E'_s$.
$\Gamma \mid \langle\langle \delta \rangle\rangle \vdash M : \tau$ must have been derived by using T-Plus. Thus, we have $\tau = \mathbf{Int}$ and $\Gamma \mid \langle\langle \delta \rangle\rangle \vdash E'_s[I] : \mathbf{Int}$. From the induction hypothesis, there exists M'' that satisfies $\Gamma \mid \langle\langle \delta' \rangle\rangle \vdash M'' : \mathbf{Int}$ and $(E'_s[I], \delta) \longrightarrow (M'', \delta')$. Because $M' = i + M''$, we have $\Gamma \mid \langle\langle \delta' \rangle\rangle \vdash M' : \tau$ from T-Plus as required.

– Case $E_s = \mathbf{leaf}\ E'_s$.
$\Gamma \mid \langle\langle \delta \rangle\rangle \vdash M : \tau$ must have been derived by using T-Leaf. Thus, we have $\tau = \mathbf{Tree}^+$ and $\Gamma \mid \langle\langle \delta \rangle\rangle \vdash E'_s[I] : \mathbf{Int}$. From the induction hypothesis, there exists M'' that satisfies $\Gamma \mid \langle\langle \delta' \rangle\rangle \vdash M'' : \mathbf{Int}$ and $(E'_s[I], \delta) \longrightarrow (M'', \delta')$. Because $M' = \mathbf{leaf}\ M''$, we have $\Gamma \mid \langle\langle \delta' \rangle\rangle \vdash M' : \tau$ from T-Leaf as required.

– Case $E_s = \mathbf{node}\ E'_s\ N$.
$\Gamma \mid \langle\langle \delta \rangle\rangle \vdash M : \tau$ must have been derived by using T-Node. Thus, we have $\tau = \mathbf{Tree}^+$ and $\Gamma \mid \langle\langle \delta_1 \rangle\rangle \vdash E'_s[I] : \mathbf{Tree}^+$ and $\Gamma \mid \langle\langle \delta_2 \rangle\rangle \vdash N : \mathbf{Tree}^+$ and $\delta = \delta_1, \delta_2$ for some δ_1, δ_2. From the induction hypothesis, there exist δ'_1

and M'' that satisfy $\Gamma \mid \langle\langle \delta_1' \rangle\rangle \vdash M'' : \mathbf{Tree}^+$ and $(E_s'[I], \delta_1) \longrightarrow (M'', \delta_1')$. Because $M' = \mathbf{node}\ M''\ N$ and $\delta' = \delta_1', \delta_2$, we have $\Gamma \mid \langle\langle \delta' \rangle\rangle \vdash M' : \tau$ from T-Node as required.

- Case $E_s = \mathbf{node}\ V\ E_s'$.
  $\Gamma \mid \langle\langle \delta \rangle\rangle \vdash M : \tau$ must have been derived by using T-Node. Thus, we have $\tau = \mathbf{Tree}^+$ and $\Gamma \mid \langle\langle \delta \rangle\rangle \vdash E_s'[I] : \mathbf{Tree}^+$. From the induction hypothesis, there exists M'' that satisfies $\Gamma \mid \langle\langle \delta' \rangle\rangle \vdash M'' : \mathbf{Tree}^+$ and $(E_s'[I], \delta) \longrightarrow (M'', \delta')$. Because $M' = \mathbf{node}\ V\ M''$, we have $\Gamma \mid \langle\langle \delta' \rangle\rangle \vdash M' : \tau$ from T-Node as required.
- Case $E_s = (\mathbf{case}\ E_s'\ \mathbf{of}\ \mathbf{leaf} x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2)$.
  $\Gamma \mid \langle\langle \delta \rangle\rangle \vdash M : \tau$ must have been derived by using T-Case. Thus, we have $\Gamma \mid \langle\langle \delta_1 \rangle\rangle \vdash E_s'[I] : \mathbf{Tree}^-$ and $\Gamma, x : \mathbf{Int} \mid \langle\langle \delta_2 \rangle\rangle \vdash M_1 : \tau$ and $\Gamma \mid x_1 : \mathbf{Tree}^-, x_2 : \mathbf{Tree}^-, \langle\langle \delta_2 \rangle\rangle \vdash M_2 : \tau$ and $\delta = \delta_1, \delta_2$ for some δ_1 and δ_2. From the induction hypothesis, there exist δ_1' and M'' that satisfy $\Gamma \mid \langle\langle \delta_1' \rangle\rangle \vdash M'' : \mathbf{Tree}^-$ and $(E_s'[I], \delta_1) \longrightarrow (M'', \delta_1')$. Because $M' = (\mathbf{case}\ M''\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2)$ and $\delta' = \delta_1', \delta_2$, we have $\Gamma \mid \langle\langle \delta' \rangle\rangle \vdash M' : \tau$ from T-Case as required.

□

F The Proof of Lemma 1

For the proof of Lemma 1, we prepare the following lemma.

Lemma 4. *If $x \notin \gamma$, then $\mathcal{A}_\gamma([M_1/x]M_2) = [\mathcal{A}_\gamma(M_1)/x]\mathcal{A}_\gamma(M_2)$.*

Proof. This follows from straightforward induction on the structure of M_2.

□

Proof of Lemma 1. To prove $(M, \delta) \sim (e, S_i, S_o)$, it is sufficient to prove $M \sim_{\mathbf{FV}(M)} (e, S_o)$. We hereafter write γ for $\mathbf{FV}(M)$ and S for S_o.

First, suppose that M is not reducible. Then, $M = U$ or $M = V$.

- Case $M = U$.
  Let $e = \mathcal{A}_\gamma(M)$ and $S = \emptyset$. Then $M \sim_\gamma (e, S)$ follows from C-Value. $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \longrightarrow^* (e, \langle\delta\rangle, S)$ is obvious.
- Case $M = V$.
  Let $e = ()$ and $S = [\![\, V \,]\!]$. Then $M \sim_\gamma (e, S)$ follows from C-Tree. $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \longrightarrow^* (e, \langle\delta\rangle, S)$ follows from the structural induction on V below:
  - Case $V = \mathbf{leaf}\ i$.
    In this case, $\mathcal{A}_\gamma(M) = (\mathbf{write}(\mathbf{leaf}); \mathbf{write}(i))$. Thus, $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \longrightarrow^* (e, \langle\delta\rangle, S)$ holds.
  - Case $V = \mathbf{node}\ V_1\ V_2$.
    In this case, $\mathcal{A}_\gamma(M) = (\mathbf{write}(\mathbf{node}); \mathcal{A}_\gamma(V_1); \mathcal{A}_\gamma(V_2))$ and $S = \mathbf{node}; [\![\, V_1 \,]\!]; [\![\, V_2 \,]\!]$. Because $\emptyset \mid \emptyset \vdash V_1 : \mathbf{Tree}^+$ and $\emptyset \mid \emptyset \vdash V_2 : \mathbf{Tree}^+$, we have $(\mathcal{A}_\gamma(V_1), \emptyset, \emptyset) \longrightarrow^* ((), \emptyset, [\![\, V_1 \,]\!])$ and $(\mathcal{A}_\gamma(V_2), \emptyset, \emptyset) \longrightarrow^* ((), \emptyset, [\![\, V_2 \,]\!])$ from the induction hypothesis. Thus, $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \longrightarrow^* (e, \langle\delta\rangle, S)$. (Note that $\delta = \emptyset$ because $\emptyset \mid \emptyset \vdash V : \mathbf{Tree}^+$.)

Next, suppose that M is reducible. Then, there exist E_s and I such as $M = E_s[I]$. We use structural induction on E_s.

- Case $E_s = [\,]$.
  In this case, $M = I$. Let $e = \mathcal{A}_\gamma(M)$ and $S = \emptyset$. Then, $M \sim_\gamma (e, S)$ follows from C-Inst. $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \longrightarrow^* (e, \langle\delta\rangle, S)$ is obvious.
- Case $E_s = E'_s\ M'$.
  In this case, $M = E'_s[I]\ M'$. $\emptyset \mid \langle\langle\delta_1\rangle\rangle \vdash E'_s[I] : \tau' \to \tau$ and $\emptyset \mid \langle\langle\delta_2\rangle\rangle \vdash M' : \tau'$ and $\delta = \delta_1, \delta_2$ follow for some δ_1 and δ_2 from the assumption $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash M : \tau$. Thus, $E'_s[I] \sim_\gamma (e', S')$ and $(\mathcal{A}_\gamma(E'_s[I]), \langle\delta_1\rangle, \emptyset) \longrightarrow^* (e', \langle\delta_1\rangle, S')$ follow for some e' and S' from the induction hypothesis. Let e be $e'\ \mathcal{A}_\gamma(M')$ and S be S'. Then, $M \sim_\gamma (e, S)$ follows from C-App1. Because $\mathcal{A}_\gamma(M) = \mathcal{A}_\gamma(E'_s[I])\ \mathcal{A}_\gamma(M')$, we have $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \longrightarrow^* (e, \langle\delta\rangle, S)$.
- Case $E_s = (\lambda x.M')\ E'_s$.
  In this case, $M = (\lambda x.M')\ E'_s[I]$. $\emptyset \mid \emptyset \vdash (\lambda x.M') : \tau' \to \tau$ and $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash E'_s[I] : \tau'$ follow for some τ' from the assumption $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash M : \tau$. Thus, $E'_s[I] \sim_\gamma (e', S')$ and $(\mathcal{A}_\gamma(E'_s[I]), \langle\delta\rangle, \emptyset) \longrightarrow^* (e', \langle\delta\rangle, S')$ follow for some e' and S' from the induction hypothesis. Let e be $(\lambda x.\mathcal{A}_\gamma(M'))\ e'$ and S be S'. Then, $M \sim_\gamma (e, S)$ follows from C-App2. Because $\mathcal{A}_\gamma(M) = (\lambda x.\mathcal{A}_\gamma(M'))\ \mathcal{A}_\gamma(E'_s[I])$, we have $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \longrightarrow^* (e, \langle\delta\rangle, S)$.
- Case $E_s = E'_s + M'$.
  In this case, $M = E'_s[I] + M'$. $\emptyset \mid \langle\langle\delta_1\rangle\rangle \vdash E'_s[I] : \mathbf{Int}$ and $\emptyset \mid \langle\langle\delta_2\rangle\rangle \vdash M' : \mathbf{Int}$ and $\delta = \delta_1, \delta_2$ follow for some δ_1 and δ_2 from the assumption $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash M : \tau$. Thus, $E'_s[I] \sim_\gamma (e', S')$ and $(\mathcal{A}_\gamma(E'_s[I]), \langle\delta_1\rangle, \emptyset) \longrightarrow^* (e', \langle\delta_1\rangle, S')$ follows for some e' and S' from the induction hypothesis. Let e be $e' + \mathcal{A}_\gamma(M')$ and S be S'. Then, $M \sim_\gamma (e, S)$ follows from C-Plus1. Because $\mathcal{A}_\gamma(M) = \mathcal{A}_\gamma(E'_s[I]) + \mathcal{A}_\gamma(M')$, we have $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \longrightarrow^* (e, \langle\delta\rangle, S)$.
- Case $E_s = i + E'_s$.
  In this case, $M = i + E'_s[I]$. $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash E'_s[I] : \mathbf{Int}$ follow from the assumption $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash M : \tau$. Thus, $E'_s[I] \sim_\gamma (e', S')$ and $(\mathcal{A}_\gamma(E'_s[I]), \langle\delta\rangle, \emptyset) \longrightarrow^* (e', \langle\delta\rangle, S')$ follow for some e' and S' from the induction hypothesis. Let e be $i + e'$ and S be S'. Then, $M \sim_\gamma (e, S)$ follows from C-Plus2. Because $\mathcal{A}_\gamma(M) = i + \mathcal{A}_\gamma(E'_s[I])$, we have $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \longrightarrow^* (e, \langle\delta\rangle, S)$.
- Case $E_s = \mathbf{leaf}\ E'_s$.
  In this case, $M = \mathbf{leaf}\ E'_s[I]$. $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash E'_s[I] : \mathbf{Tree}^+$ follows from the assumption $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash M : \tau$. Thus, $E'_s[I] \sim_\gamma (e', S')$ and $(\mathcal{A}_\gamma(E'_s[I]), \langle\delta\rangle, \emptyset) \longrightarrow^* (e', \langle\delta\rangle, S')$ follow for some e' and S' from the induction hypothesis. Let e be $\mathbf{write}(e')$ and S be $\mathbf{leaf}; S'$. Then, $M \sim_\gamma (e, S)$ follows from C-Leaf. $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \longrightarrow^* (e, \langle\delta\rangle, S)$ follows from $\mathcal{A}_\gamma(M) = \mathbf{write}(\mathbf{leaf}); \mathbf{write}(\mathcal{A}_\gamma(E'_s[I]))$.
- Case $E_s = \mathbf{node}\ E'_s\ M'$.
  In this case, $M = \mathbf{node}\ E'_s[I]\ M'$. $\emptyset \mid \langle\langle\delta_1\rangle\rangle \vdash E'_s[I] : \mathbf{Tree}^+$ and $\emptyset \mid \langle\langle\delta_2\rangle\rangle \vdash M' : \mathbf{Tree}^+$ and $\delta = \delta_1, \delta_2$ follow for some δ_1 and δ_2 from the assumption $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash M : \tau$. Thus, $E'_s[I] \sim_\gamma (e', S')$ and $(\mathcal{A}_\gamma(E'_s[I]), \langle\delta_1\rangle, \emptyset) \longrightarrow^* (e', \langle\delta_1\rangle, S')$ follows for some e' and S' from the induction hypothesis. Let e be $e'; \mathcal{A}_\gamma(M')$ and S be $\mathbf{node}; S'$. Then, $M \sim_\gamma (e, S)$ follows from C-Node1. Because $\mathcal{A}_\gamma(M) = \mathbf{write}(\mathbf{node}); \mathcal{A}_\gamma(E'_s[I]); \mathcal{A}_\gamma(M')$, we have $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \longrightarrow^* (e, \langle\delta\rangle, S)$.

– Case $E_s = \mathbf{node}\ V\ E'_s$.
In this case, $M = \mathbf{node}\ V\ E'_s[I]$. $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash E'_s[I]{:}\mathbf{Tree}^+$ follows from the assumption $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash M{:}\tau$. Thus, $E'_s[I] \sim_\gamma (e', S')$ and $(\mathcal{A}_\gamma(E'_s[I]), \langle\delta\rangle, \emptyset) \longrightarrow^* (e', \langle\delta\rangle, S')$ follow for some e' and S' from the induction hypothesis. Let e be e' and S be $\mathbf{node}; [\![V]\!]; S'$. Then, $M \sim_\gamma (e, S)$ follows from C-Node2. Because $\mathcal{A}_\gamma(M) = \mathbf{write}(\mathbf{node})$;
$\mathcal{A}_\gamma(V); \mathcal{A}_\gamma(E'_s[I])$, we have $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \longrightarrow^* (e, \langle\delta\rangle, S)$.
– Case $E_s = (\mathbf{case}\ E'_s\ \mathbf{of\ leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2)$.
In this case, $(M = (\mathbf{case}\ E'_s[I]\ \mathbf{of\ leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2))$. $\emptyset \mid \langle\langle\delta_1\rangle\rangle \vdash E'_s[I] : \mathbf{Tree}^-$ and $x : \mathbf{Int} \mid \langle\langle\delta_2\rangle\rangle \vdash M_1 : \tau$ and $\emptyset \mid x_1 : \mathbf{Tree}^-, x_2 : \mathbf{Tree}^-, \langle\langle\delta_2\rangle\rangle \vdash M_1 : \tau$ and $\delta = \delta_1, \delta_2$ follow for some δ_1 and δ_2 from the assumption $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash M : \tau$. Thus, $E'_s[I] \sim_\gamma (e', S')$ and $(\mathcal{A}_\gamma(E'_s[I]), \langle\delta_1\rangle, \emptyset) \longrightarrow^* (e', \langle\delta_1\rangle, S')$ follows for some e' and S' from the induction hypothesis. Let e be $\mathbf{case}\ e'; \mathbf{read}()\ \mathbf{of\ leaf} \Rightarrow \mathbf{let}\ x = \mathbf{read}()\ \mathbf{in}\ \mathcal{A}_\gamma(M_1) \mid \mathbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2)$ and S be S'. Then, $M \sim_\gamma (e, S)$ follows from C-Case. Because $\mathcal{A}_\gamma(M) = \mathbf{case}\ \mathcal{A}_\gamma(E'_s[I]); \mathbf{read}()\ \mathbf{of\ leaf} \Rightarrow \mathbf{let}\ x = \mathbf{read}()\ \mathbf{in}\ \mathcal{A}_\gamma(M_1) \mid \mathbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2)$. $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \longrightarrow^* (e, \langle\delta\rangle, S)$ holds.

□

G The Proof of Lemma 2

We prove Lemma 2 in this section.

Proof. The second property follows immediately from the definition of $M \sim_\gamma (e, S)$. (If M is irreducible, then $M \sim_\gamma (e, S)$ must follow either from C-Value or C-Tree, which implies that e is irreducible too.)

We prove the first property below. To prove $(M', \delta') \sim (e', S'_i, S'_o)$, it is sufficient to prove $M' \sim_{\mathbf{FV}(M)} (e', S'_o)$. We hereafter write γ for $\mathbf{FV}(M)$ and S' for S'_o.

Suppose $(M, \delta) \longrightarrow (M', \delta')$. Then, $M = E_s[I]$ for some E_s and I. We use structural induction on E_s.

– Case $E_s = [\,]$.

- Case $I = i_1 + i_2$.
In this case, $(M, \delta) \longrightarrow (M', \delta')$ must have been derived by using Es2-Plus. Thus, $M' = \mathit{plus}(i_1, i_2)(= i)$ and $\delta' = \delta$. $M \sim_\gamma (e, S)$ implies $e = \mathcal{A}_\gamma(I) = i_1 + i_2$ and $S = \emptyset$. Let $e' = i$ and $S' = \emptyset$. Then $(e, \langle\delta\rangle, S) \longrightarrow^+ (e', \langle\delta'\rangle, S')$ and $M' \sim_\gamma (e', S')$ hold as required.
- Case $I = (\lambda x.M_1)U$.
$(M, \delta) \longrightarrow (M', \delta')$ must have been derived by using Es2-App. So, it must be the case that $M' = [U/x]M_1$ and $\delta' = \delta$. $M \sim_\gamma (e, S)$ implies $e = \mathcal{A}_\gamma(I) = (\lambda x.\mathcal{A}_\gamma(M_1))\mathcal{A}_\gamma(U)$ and $S = \emptyset$. By Lemma 4, we have:

$$\begin{aligned}(e, \langle\delta\rangle, \emptyset) &\longrightarrow ([\mathcal{A}_\gamma(U)/x]\mathcal{A}_\gamma(M_1), \langle\delta\rangle, \emptyset) \\ &= (\mathcal{A}_\gamma(M'), \langle\delta\rangle, \emptyset).\end{aligned}$$

By Lemma 1, there exist e'' and S'' that satisfy $(\mathcal{A}_\gamma(M'), \langle\delta\rangle, \emptyset) \longrightarrow^* (e'', \langle\delta\rangle, S'')$ and $M' \sim_\gamma (e'', S'')$. Let e' be e'' and S' be S''. Then, $(e, \langle\delta\rangle, S) \longrightarrow^+ (e', \langle\delta'\rangle, S')$ and $M' \sim_\gamma (e', S')$ hold as required.

- Case $I = \mathbf{case}\ y\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2$ with $\delta = (y \mapsto \mathbf{leaf}\ i, \delta_1)$.
  $(M, \delta) \longrightarrow (M', \delta')$ must have been derived by using Es2-Case1. So, it must be the case that $M' = [i/x]M_1$ and $\delta' = \delta_1$. $M \sim_\gamma (e, S)$ implies $e = \mathcal{A}_\gamma(I) = \mathbf{case}\ (); \mathbf{read}()\ \mathbf{of}\ \mathbf{leaf} \Rightarrow \mathbf{let}\ x = \mathbf{read}()\ \mathbf{in}\ \mathcal{A}_\gamma(M_1) \mid \mathbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2)$ and $S = \emptyset$. $(e, \mathbf{leaf}; i; \langle\delta_1\rangle, \emptyset) \longrightarrow^+ (\mathcal{A}_\gamma(M'), \langle\delta_1\rangle, \emptyset)$ follows from Lemma 4. By Lemma 1, there exist e'' and S'' that satisfy $(\mathcal{A}_\gamma(M'), \langle\delta_1\rangle, \emptyset) \longrightarrow (e'', \langle\delta_1\rangle, S'')$ and $M' \sim_\gamma (e'', S'')$. Let e' be e'' and S' be S''. Then, we have $(e, \langle\delta\rangle, S) \longrightarrow^+ (e', \langle\delta'\rangle, S')$ and $M' \sim_\gamma (e', S')$ as required.
- Case $I = \mathbf{case}\ y\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2$ with $\delta = (y \mapsto \mathbf{node}\ V_1\ V_2, \delta_1)$.
  $(M, \delta) \longrightarrow (M', \delta')$ must have been derived by using Es2-Case2. So, it must be the case that $M' = M_2$ and $\delta' = x_1 \mapsto V_1, x_2 \mapsto V_2, \delta_1$. $M \sim_\gamma (e, S)$ implies $e = \mathcal{A}_\gamma(I) = \mathbf{case}\ (); \mathbf{read}()\ \mathbf{of}\ \mathbf{leaf} \Rightarrow \mathbf{let}\ x = \mathbf{read}()\ \mathbf{in}\ \mathcal{A}_\gamma(M_1) \mid \mathbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2)$ and $S = \emptyset$. As easily seen, $[()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2) = \mathcal{A}_{\gamma\cup\{x_1,x_2\}}(M_2)$.Thus, $(e, \mathbf{node}; [\![V_1]\!]; [\![V_2]\!]; \langle\delta_1\rangle, \emptyset) \longrightarrow^+ (\mathcal{A}_{\gamma\cup\{x_1,x_2\}}(M_2), [\![V_1]\!]; [\![V_2]\!]; \langle\delta_1\rangle, \emptyset)$.
  By Lemma1, there exist e'' and S'' that satisfy
  $(\mathcal{A}_{\gamma\cup\{x_1,x_2\}}(M_2), [\![V_1]\!]; [\![V_2]\!]; \langle\delta_1\rangle, \emptyset) \longrightarrow^*$
  $(e'', [\![V_1]\!]; [\![V_2]\!]; \langle\delta_1\rangle, S'')$ and $M_2 \sim_\gamma (e'', S'')$. Let $e' = e''$ and $S' = S''$. Then, we have $(e, \langle\delta\rangle, S) \longrightarrow^+ (e', \langle\delta'\rangle, S')$ and $M' \sim_\gamma (e', S')$ as required.
- Case $I = \mathbf{fix}\ f.M_1$.
  $(M, \delta) \longrightarrow (M', \delta')$ must have been derived by using Es2-Fix. So, it must be the case that $M' = [\mathbf{fix}\ f.M_1/f]M_1$ and $\delta' = \delta$. $M \sim_\gamma (e, S)$ implies $e = \mathcal{A}_\gamma(I) = \mathbf{fix}\ f.\mathcal{A}_\gamma(M_1)$ and $S = \emptyset$. By Lemma4, we have:

$$\begin{aligned}(e, \langle\delta\rangle, \emptyset) &\longrightarrow ([\mathbf{fix}\ f.\mathcal{A}_\gamma(M_1)/f]\mathcal{A}_\gamma(M_1), \langle\delta\rangle, \emptyset)\\ &= (\mathcal{A}_\gamma(M'), \langle\delta\rangle, \emptyset).\end{aligned}$$

  By Lemma1, there exsist e'' and S'' that satisfy $(\mathcal{A}_\gamma(M'), \langle\delta\rangle, \emptyset) \longrightarrow^* (e'', \langle\delta\rangle, S'')$ and $M' \sim_\gamma (e'', S'')$. Let e' be e'' and S' be S'' Then, we have $(e, \langle\delta\rangle, S) \longrightarrow^+ (e', \langle\delta'\rangle, S')$ and $M' \sim_\gamma (e', S')$ as required.

– Case $E_s = E_1 M_2$.
  There exists M_1' that satisfies $M' = M_1'\ M_2$ and $(E_1[I], \delta) \longrightarrow (M_1', \delta')$. $M \sim_\gamma (e, S)$ must have been derived from C-App1. Thus, there exists e_1 that satisfies $e = e_1\ \mathcal{A}_\gamma(M_2)$ and $E_1[I] \sim_\gamma (e_1, S)$. Because M is well-typed, $E_1[I]$ is also well-typed. Thus, from the induction hypothesis, we have $(e_1\ \mathcal{A}_\gamma(M_2), \langle\delta\rangle, S) \longrightarrow^+ (e_1'\ \mathcal{A}_\gamma(M_2), \langle\delta'\rangle, S_1)$ and $M_1' \sim_\gamma (e_1', S_1)$ for some e_1' and S_1.
  First, suppose that M_1' is reducible. Let e' be $e_1'\ \mathcal{A}_\gamma(M_2)$ and S' be S_1. Then, $M' \sim_\gamma (e', S')$ follows from C-App1 as required.
  Next, suppose that M_1' is not reducible. Because M_1' is a function-typed term, $M_1' \sim_\gamma (e_1', S_1)$ must have been derived from C-Value. Thus, $e_1' = \mathcal{A}_\gamma(M_1')$

and thus, $e'_1\ \mathcal{A}_\gamma(M_2) = \mathcal{A}_\gamma(M')$. From Lemma 1, there exist e'' and S'' that satisfy $(e'_1\ \mathcal{A}_\gamma(M_2), \langle\delta'\rangle, S_1) \longrightarrow^* (e'', \langle\delta'\rangle, S'')$ and $M' \sim_\gamma (e'', S'')$. Let e' be e'' and S' be S''. Then, $(e, \langle\delta\rangle, S) \longrightarrow^+ (e', \langle\delta'\rangle, S')$ and $M' \sim_\gamma (e', S')$ follow as required.

- Case $E_s = (\lambda x.M_2)\ E_1$.
  There exists M'_1 such that $M' = (\lambda x.M_2)\ M'_1$ and $(E_1[I], \delta) \longrightarrow (M'_1, \delta')$. $M \sim_\gamma (e, S)$ must have been derived from C-App2. Thus, there exists e_1 that satisfies $e = (\lambda x.\mathcal{A}_\gamma(M_2))\ e_1$ and $E_1[I] \sim_\gamma (e_1, S)$. Because M is well-typed, $E_1[I]$ is also well-typed. Thus, from the induction hypothesis, we have $((\lambda x.\mathcal{A}_\gamma(M_2))\ e_1, \langle\delta\rangle, S) \longrightarrow^+ ((\lambda x.\mathcal{A}_\gamma(M_2))\ e'_1, \langle\delta'\rangle, S_1)$ and $M'_1 \sim_\gamma (e'_1, S_1)$ for some e'_1 and S_1.
  First, suppose that M'_1 is reducible. Let e' be $(\lambda x.\mathcal{A}_\gamma(M_2))\ e'_1$ and S' be S_1. Then, $M' \sim_\gamma (e', S')$ follows from C-App2 as required.
  Next, suppose that M'_1 is a value. Because a tree-typed value cannot be passed to a function, $M'_1 \sim_\gamma (e'_1, S_1)$ must have been derived from C-Value, not from C-Tree. Thus, $e'_1 = \mathcal{A}_\gamma(M'_1)$ and thus, $(\lambda x.\mathcal{A}_\gamma(M_2))\ e'_1 = \mathcal{A}_\gamma(M')$ From Lemma 1, there exist e'' and S'' that satisfy $((\lambda x.\mathcal{A}_\gamma(M_2))\ e'_1, \langle\delta'\rangle, S_1) \longrightarrow^* (e'', \langle\delta'\rangle, S'')$ and $M' \sim_\gamma (e'', S'')$. Let e' be e'' and S' be S''. Then, $(e, \langle\delta\rangle, S) \longrightarrow^+ (e', \langle\delta'\rangle, S')$ and $M' \sim_\gamma (e', S')$ follow as required.
- Case $E_s = E_1 + M_2$.
  There exists M'_1 that satisfies $M' = M'_1 + M_2$ and $(E_1[I], \delta) \longrightarrow (M'_1, \delta')$. $M \sim_\gamma (e, S)$ must have been derived from C-Plus1. Thus, there exists e_1 that satisfies $e = e_1 + \mathcal{A}_\gamma(M_2)$ and $E_1[I] \sim_\gamma (e_1, S)$. Because M is well-typed, $E_1[I]$ is also well-typed. Thus, from the induction hypothesis, we have $(e_1 + \mathcal{A}_\gamma(M_2), \langle\delta\rangle, S) \longrightarrow^+ (e'_1 + \mathcal{A}_\gamma(M_2), \langle\delta'\rangle, S_1)$ and $M'_1 \sim_\gamma (e'_1, S_1)$ for some e'_1 and S_1.
  First, suppose that M'_1 is reducible. Let e' be $e'_1 + \mathcal{A}_\gamma(M_2)$ and S' be S_1. Then, $M' \sim_\gamma (e', S')$ follows from C-Plus1 as required.
  Next, suppose that M'_1 is not reducible. Because M'_1 is an integer, $M'_1 \sim_\gamma (e'_1, S_1)$ must have been derived from C-Value. Thus, $e'_1 = \mathcal{A}_\gamma(M'_1)$ and thus, $e'_1 + \mathcal{A}_\gamma(M_2) = \mathcal{A}_\gamma(M')$. From Lemma 1, there exist e'' and S'' that satisfy $(e'_1 + \mathcal{A}_\gamma(M_2), \langle\delta'\rangle, S_1) \longrightarrow^* (e'', \langle\delta'\rangle, S'')$ and $M' \sim_\gamma (e'', S'')$. Let e' be e'' and S' be S''. Then, $(e, \langle\delta\rangle, S) \longrightarrow^+ (e', \langle\delta'\rangle, S')$ and $M' \sim_\gamma (e', S')$ follow as required.
- Case $E = i_2 + E_1$.
  There exists M'_1 that satisfies $M' = i_2 + M'_1$ and $(E_1[I], \delta) \longrightarrow (M'_1, \delta')$. $M \sim_\gamma (e, S)$ must have been derived from C-Plus2. Thus, there exists e_1 that satisfies $e = i_2 + e_1$ and $E_1[I] \sim_\gamma (e_1, S)$. Because M is well-typed, $E_1[I]$ is also well-typed. Thus, from the induction hypothesis, we have $(i_2 + e_1, \langle\delta\rangle, S) \longrightarrow^+ (i_2 + e'_1, \langle\delta'\rangle, S_1)$ and $M'_1 \sim_\gamma (e'_1, S_1)$ for some e'_1 and S_1.
  First, suppose that M'_1 is reducible. Let e' be $i_2 + e'_1$ and S' be S_1. Then, $M' \sim_\gamma (e', S')$ follows from C-Plus2 as required.
  Next, suppose that M'_1 is a value. $M'_1 \sim_\gamma (e'_1, S_1)$ must have been derived from C-Value. Thus, $e'_1 = \mathcal{A}_\gamma(M'_1)$ and thus, $i_2 + e'_1 = \mathcal{A}_\gamma(M')$ From Lemma 1, there exist e'' and S'' that satisfy $(i_2 + e'_1, \langle\delta'\rangle, S_1) \longrightarrow^*$

$(e'', \langle\delta'\rangle, S'')$ and $M' \sim_\gamma (e'', S'')$. Let e' be e'' and S' be S''. Then, $(e, \langle\delta\rangle, S) \longrightarrow^+ (e', \langle\delta'\rangle, S')$ and $M' \sim_\gamma (e', S')$ follow as required.

- Case $E_s = \mathbf{leaf}\ E_1$.
  There exists M_1' that satisfies $M' = \mathbf{leaf}\ M_1'$ and $(E_1[I], \delta) \longrightarrow (M_1', \delta')$. $M \sim_\gamma (e, S)$ must have been derived from C-Leaf. Thus, there exist e_1 and S_1 that satisfies $e = \mathbf{write}(e_1)$ and $E_1[I] \sim_\gamma (e_1, S_1)$ and $S = \mathbf{leaf}; S_1$. Because M is well-typed, $E_1[I]$ is also well-typed. Thus, from the induction hypothesis, we have $(e_1, \langle\delta\rangle, S_1) \longrightarrow^+ (e_1', \langle\delta'\rangle, S_1')$ and $M_1' \sim_\gamma (e_1', S_1')$ for some e_1' and S_1'.
  First, suppose that M_1' is reducible. Let e' be $\mathbf{write}(e_1')$ and S' be $\mathbf{leaf}; S_1$. Then, $M' \sim_\gamma (e', S')$ follows from C-Leaf as required.
  Next, suppose that M_1' is a value. Because M is well-typed, M_1' is an integer (let the integer be i_1') and $M_1' \sim_\gamma (e_1', S_1')$ must have been derived from C-Value. Thus, $e_1' = i_1'$ and $S_1' = \emptyset$. Let e' be $()$ and S' be $\mathbf{leaf}; i_1'$. Then, $M' \sim_\gamma (e', S')$ follows from C-Tree and $(\mathbf{write}(e_1'), \langle\delta'\rangle, \mathbf{leaf}; S_1') \longrightarrow (e', \langle\delta'\rangle, S')$ holds.
- Case $E_s = \mathbf{node}\ E_1\ M_2$.
  There exists M_1' that satisfies $M' = \mathbf{node}\ M_1'\ M_2$ and $(E_1[I], \delta) \longrightarrow (M_1', \delta')$. $M \sim_\gamma (e, S)$ must have been derived from C-Node1. Thus, there exist e_1 and S_1 that satisfies $e = e_1; \mathcal{A}_\gamma(M_2)$ and $E_1[I] \sim_\gamma (e_1, S_1)$ and $S = \mathbf{node}; S_1$. Because we assume that M is well-typed, $E_1[I]$ is also well-typed. Thus, from the induction hypothesis, we have $(e_1; \mathcal{A}_\gamma(M_2), \langle\delta\rangle, \mathbf{node}; S_1) \longrightarrow^+ (e_1'; \mathcal{A}_\gamma(M_2), \langle\delta'\rangle, \mathbf{node}; S_1')$ and $M_1' \sim_\gamma (e_1', S_1')$ for some e_1' and S_1'.
  First, suppose that M_1' is reducible. Let e' be $e_1'; \mathcal{A}_\gamma(M_2)$ and S' be $\mathbf{node}; S_1'$. Then, $M' \sim_\gamma (e', S')$ follows from C-Node1 as required.
  Next, suppose that M_1' is a value (let the value be V_1'). $M_1' \sim_\gamma (e_1', S_1)$ must have been derived from C-Tree. Thus, $e_1' = ()$ and $S_1 = [\![V_1']\!]$. Thus, $(e, \langle\delta\rangle, S) \longrightarrow^+ (\mathcal{A}_\gamma(M_2), \langle\delta'\rangle, \mathbf{node}; [\![V_1']\!])$. From Lemma 1, there exist e_2 and S_2 that satisfy $M_2 \sim_\gamma (e_2, S_2)$ and $(\mathcal{A}_\gamma(M_2), \langle\delta'\rangle, \mathbf{node}; [\![V_1']\!]) \longrightarrow^* (e_2, \langle\delta'\rangle, \mathbf{node}; [\![V_1']\!]; S_2)$. Let e' be e_2 and S' be $\mathbf{node}; [\![V_1']\!]; S_2$. Then, $M' \sim_\gamma (e', S')$ follows from C-Node2 and $(e, \langle\delta\rangle, S) \longrightarrow^+ (e', \langle\delta'\rangle, S')$ holds.
- Case $E_s = \mathbf{node}\ V_2\ E_1$.
  There exists M_1' that satisfies $M' = \mathbf{node}\ V_2\ M_1'$ and $(E_1[I], \delta) \longrightarrow (M_1', \delta')$. $M \sim_\gamma (e, S)$ must have been derived from C-Node2. Thus, there exist e_1 and S_1 that satisfies $e = e_1$ and $E_1[I] \sim_\gamma (e_1, S_1)$ and $S = \mathbf{node}; [\![V_2]\!]; S_1$. Because M is well-typed, $E_1[I]$ is also well-typed. Thus, from the induction hypothesis, we have $(e_1, \langle\delta\rangle, \mathbf{node}; [\![V_2]\!]; S_1) \longrightarrow^+ (e_1', \langle\delta'\rangle, \mathbf{node}; [\![V_2]\!]; S_1')$ and $M_1' \sim_\gamma (e_1', S_1')$ for some e_1' and S_1'.
  First, suppose that M_1' is reducible. Let e' be e_1' and S' be $\mathbf{node}; [\![V_2]\!]; S_1'$. Then, $M' \sim_\gamma (e', S')$ follows from C-Node1 as required.
  Next, suppose that M_1' is a value (let the value be V_1'). $M_1' \sim_\gamma (e_1', S_1)$ must have been derived from C-Tree. Thus, $e_1' = ()$ and $S_1 = [\![V_1']\!]$, and thus, $(e, \langle\delta\rangle, S) \longrightarrow^+ ((), \langle\delta'\rangle, \mathbf{node}; [\![V_2]\!]; [\![V_1']\!])$. Let e' be $()$ and S' be $\mathbf{node}; [\![V_2]\!]; [\![V_1']\!]$. Then, $M' \sim_\gamma (e', S')$ follows from C-Tree as required.

– Case $E_s = (\mathbf{case}\ E_1\ \mathbf{of\ leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2)$.
There exists M_1' that satisfies $M' = (\mathbf{case}\ M_1'\ \mathbf{of\ leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2)$ and $(E_1[I], \delta) \longrightarrow (M_1', \delta')$. $M \sim_\gamma (e, S)$ must have been derived from C-Case. Thus, there exists e_1 that satisfies $e = (\mathbf{case}\ e_1; \mathbf{read}()\ \mathbf{of\ leaf} \Rightarrow \mathbf{let}\ x = \mathbf{read}()\ \mathbf{in}\ M_1 \mid \mathbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2))$ and $E_1[I] \sim_\gamma (e_1, S)$. Because M is well-typed, $E_1[I]$ is also well-typed. Thus, from the induction hypothesis, we have $((\mathbf{case}\ e_1; \mathbf{read}()\ \mathbf{of\ leaf} \Rightarrow \mathbf{let}\ x = \mathbf{read}()\ \mathbf{in}\ M_1 \mid \mathbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2)), \langle\delta\rangle, S) \longrightarrow^+ ((\mathbf{case}\ e_1'; \mathbf{read}()\ \mathbf{of\ leaf} \Rightarrow \mathbf{let}\ x = \mathbf{read}()\ \mathbf{in}\ M_1 \mid \mathbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2)), \langle\delta'\rangle, S'')$ and $M_1' \sim_\gamma (e_1', S'')$ for some e_1' and S''.
First, suppose that M_1' is reducible. Let e' be $(\mathbf{case}\ e_1'; \mathbf{read}()\ \mathbf{of\ leaf} \Rightarrow \mathbf{let}\ x = \mathbf{read}()\ \mathbf{in}\ M_1 \mid \mathbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2))$ and S' be S''. Then, $M' \sim_\gamma (e', S')$ follows from C-Case.
Next, suppose that M_1' is not reducible. Because M_1' is a tree-typed term, M_1' is a variable (let it be y_1'). Because $M_1' \sim_\gamma (e_1', S'')$ must have been derived from C-Value, $e_1' = \mathcal{A}_\gamma(y_1')$. Thus, $(e, \langle\delta\rangle, S) \longrightarrow^+ (\mathcal{A}_\gamma(M'), \langle\delta'\rangle, S'')$. From Lemma 1, there exist e'' and S_1 that satisfy $M' \sim_\gamma (e'', S_1)$ and $(\mathcal{A}_\gamma(M'), \langle\delta'\rangle, S'') \longrightarrow^* (e'', \langle\delta'\rangle, S_1)$. Let e' be e'' and S' be S_1. Then, $(e, \langle\delta\rangle, S) \longrightarrow^+ (e', \langle\delta'\rangle, S')$ and $M' \sim_\gamma (e', S')$ hold as required.

□

H Proof of Theorem 3

This section proves Theorem 3.

Proof. First of all, note that $\emptyset \mid x : \mathbf{Tree}^- \vdash M\ x : \tau$ follows an assumption $\emptyset \mid \emptyset \vdash M : \mathbf{Tree}^- \to \tau$ and $\emptyset \mid x : \mathbf{Tree}^- \vdash x : \mathbf{Tree}^-$.

We prove only (ii) hereafter. (i) can be proved in the same way.

Assume $((M\ x), x \mapsto V) \to^* (V', \emptyset)$. Because $\emptyset \mid x : \mathbf{Tree}^- \vdash (M\ x) : \tau$ holds, there exist e, S_i and S_o such that $((M\ x), x \mapsto V) \sim (e, S_i, S_o)$ and $(\mathcal{A}(M)(), S_i, \emptyset) \to^* (e, S_i, S_o)$ from Lemma 1 [3]. From the definition of $\sim$, $S_i = [\![V]\!]$. Because of Theorem 2 and Lemma 2, there exists a sequence of reduction $(e, [\![V]\!], S_o) \to^* (e', \emptyset, S_o')$ that satisfies $(V', \emptyset) \sim (e', \emptyset, S_o')$. From the definition of $\sim$, $e' = ()$ and $S_o' = [\![V']\!]$. Thus, $(\mathcal{A}(M)(), [\![V]\!], \emptyset) \to^* ((), \emptyset, [\![V']\!])$ holds.

Next, assume $(\mathcal{A}(M)(), [\![V]\!], \emptyset) \to^* ((), \emptyset, [\![V']\!])$. As we stated above, there exist e, S_i and S_o such that $((M\ x), x \mapsto V) \sim (e, [\![V]\!], S_o)$ and $(\mathcal{A}(M)(), [\![V]\!], \emptyset) \to^* (e, [\![V]\!], S_o)$. Because applicable reduction rule can be uniquely determined at each step of reduction, $(e, [\![V]\!], S_o) \to^* ((), \emptyset, [\![V']\!])$ holds.

In the following, we prove "if $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash M' : \mathbf{Tree}^+$ and $(e, \langle\delta\rangle, S_o') \longrightarrow^* ((), \emptyset, [\![V']\!]))$ and $(M', \delta) \sim (e, S_i', S_o')$ hold, $(M', \delta) \longrightarrow^* (V', \emptyset)$ holds". We use mathematical induction on the number of reduction step of $(e, \langle\delta\rangle, S_o') \longrightarrow^* ((), \emptyset, [\![V']\!]))$. With this fact, by letting M' be $M\ x$ and δ be $x \mapsto V$, $((M\ x), x \mapsto$

[3] Because $\emptyset \mid \emptyset \vdash M : \mathbf{Tree}^- \to \tau$ holds, $\mathbf{FV}(M) = \emptyset$. Thus, $\mathcal{A}_{\mathbf{FV}(M) \cup \{x\}}(M\ x) = \mathcal{A}(M)()$.

$V) \longrightarrow^* (V', \emptyset)$ holds because $\emptyset \mid x : \mathbf{Tree}^- \vdash (M\ x) : \mathbf{Tree}^+$ follows $((M\ x), x \mapsto V) \rightarrow^* (V, \emptyset)$ and Theorem 2.

- In the case of $n = 0$, $M' = V'$ and $\delta = \emptyset$ hold because $e = (), \langle\delta\rangle = \emptyset, S'_o = [\![V']\!]$ and $(M', \emptyset) \sim (e, \emptyset, S'_o)$ hold. Thus, $(M', \delta) \longrightarrow^* (V', \emptyset)$ holds.
- In the case of $n \geq 1$, there exist e', S_i, S''_o that satisfies

$$(e, \langle\delta\rangle, S'_o) \longrightarrow (e', S_i, S''_o) \longrightarrow^* ((), \emptyset, [\![V']\!]))$$

From Lemma 2, there exist M'', δ', S'''_o that satisfies
- $(M', \delta) \longrightarrow (M'', \delta')$
- $(M'', \delta') \sim (e'', \langle\delta'\rangle, S'''_o)$
- $(e, \langle\delta\rangle, S'_o) \longrightarrow^+ (e'', \langle\delta'\rangle, S'''_o)$

Since the reduction is deterministic, $(e'', \langle\delta'\rangle, S'''_o) \longrightarrow^* ((), \emptyset, [\![V']\!]))$ holds. From the induction hypothesis, $(M'', \delta') \longrightarrow^* (V', \emptyset)$. Thus, $(M', \delta) \longrightarrow^* (V', \emptyset)$ holds.

□