TRANSLATION OF TREE-PROCESSING PROGRAMS INTO STREAM-PROCESSING PROGRAMS BASED ON ORDERED LINEAR TYPE
順序付き線形型に基づく木構造処理プログラムから ストリーム処理プログラムへの変換

by

Kohei Suenaga
末永幸平

A Master Thesis
修士論文

Submitted to
the Graduate School of the University of Tokyo
on 25 January 2005
in Partial Fulfillment of the Requirements
for the Degree of Master of
Information Science and Technology in Computer Science

Thesis Supervisor: Akinori Yonezawa　米澤 明憲
Professor of Information Science

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. The goal of this study 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. To achieve the goal, we first introduce a statically-typed language that accepts only tree-processing programs that traverse input trees from left to right in the depth-first order, and show an algorithm for translating well-typed tree-processing programs into stream-processing programs. We then remove the restriction on the access order by extending the language with primitives for selectively buffering part of trees on memory. With the extended language, programmers can write arbitrary tree-processing, but inserting the buffering primitives manually is sometimes tedious. We therefore also develop a type-based algorithm that inputs arbitrary tree-processing programs and automatically inserts the buffering primitives.

論文要旨

XML 文書や Lisp の S 式のようなデータに対する処理記述には, その論理的な構造に基づいて木構造処理として記述する手法と, 物理的な構造に基づいてストリーム処理として記述する手法がある. 前者は後者に比べて記述が容易である一方, データを木構造としてメモリ中に保持するため大規模な入力の処理の際に効率が悪い. この研究の目標は, XML 文書や Lisp の S 式のようなデータの処理を木構造処理を行うプログラムとして記述し, それをストリーム処理を行うプログラムに自動変換する方法を確立することである. これを実現するために, 入力木を左から右の深さ優先順序でアクセスする木構造処理プログラムのみが型付け可能な静的型付き言語を導入し, 型付け可能な木構造処理プログラムをストリーム処理プログラムに変換するアルゴリズムを示す. 次に, 入力木の一部をメモリ上にバッファリングするためのプリミティブを導入してこの言語を拡張し, アクセス順序に関する制約を取り除く. 拡張された言語においては, プログラマは任意の木構造処理プログラムを書くことができるが, バッファリングされた木を扱うためのプリミティブをプログラマ自身で挿入しなければならないので, 我々はこれらのプリミティブを自動的に挿入する型主導のアルゴリズムを開発し, プログラム記述をより容易に行えるようにする.

Acknowledgement

I thank Professor Akinori Yonezawa, the supervisor of the thesis, for his help.

I would also like to thank Naoki Kobayashi and Koichi Kodama, who participated in discussions and provided numerous comments throughout this study. In particular, the results described in Chapters 2-4 are the outcome of joint research with them.

I am grateful to the members of "Programming Language Principles" group at University of Tokyo and Tokyo Institute of Technology.

Contents

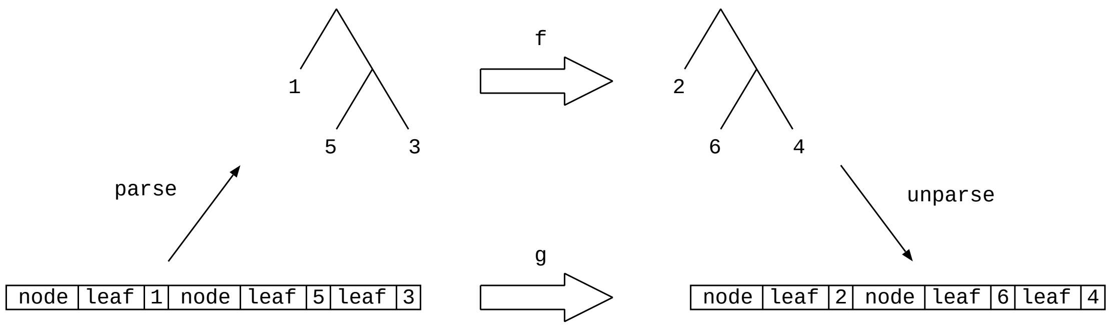
Chapter 1

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 1.1 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 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 [25]) than stream-processing programs. On the other hand, stream-processing programs have their own advantage that intermediate tree structures are not needed,


f
1
5
3
2
6
4
parse
unparse
g
node leaf 1 node leaf 5 leaf 3
node leaf 2 node leaf 6 leaf 4

Tree-processing program f:

$\mathbf{fix}\ f.\lambda t.(\mathbf{case}\ t\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow \mathbf{leaf}\ (x+1) \mid \mathbf{node}\ x_1\ x_2 \Rightarrow \mathbf{node}\ (f\ x_1)\ (f\ x_2))$

Stream-processing program: g

$\mathbf{fix}\ g.\lambda t.(\mathbf{case}\ \mathbf{read}()\ \mathbf{of}\ \mathbf{leaf}\ \Rightarrow\ \mathbf{leaf}\ (\mathbf{read}() + 1)$

$\qquad\qquad\qquad\mid\ \mathbf{node}\ \Rightarrow\ \mathbf{write}(\mathbf{node}); g\ (); g\ ())$

Figure 1.1: Tree-processing and stream-processing

so that they often run faster than the corresponding tree-processing programs if input/output trees are physically represented as streams, as in the case of XML.

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: (1) stream processing is most effective *when trees are traversed and constructed from left to right in the depth-first manner* and (2) in that case, we can obtain from the tree-processing program the corresponding stream-processing program simply by replacing case analysis on an input tree with case analysis on input tokens, and replacing tree constructions with stream outputs. In fact, the stream-processing program in Figure 1.1, 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 [20, 22]. Ordered linear types, which are an extension of linear types [3, 24], 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-typed program can be translated to an equivalent stream-processing program. The tree-processing program f in Figure 1.1 is well-typed in our type system, so that

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

Figure 1.2: An ill-typed program that swaps children of every node

Tree-processing program:
$\textbf{fix}\ f.\lambda t.(\textbf{mcase}\ t\ \textbf{of mleaf}\ x \Rightarrow \textbf{mleaf}\ x \mid \textbf{mnode}\ x_1\ x_2 \Rightarrow \textbf{mnode}\ (f\ x_2)\ (f\ x_1))$

Figure 1.3: A well-typed program that swaps children of every node

it can automatically be translated to the stream-processing program g.

On the other hand, the program in Figure 1.2 is not well-typed in our type system since it accesses the right sub-tree of an input before accessing the left sub-tree. To deal with that case, we extend our framework with *buffered trees*, which can be used arbitrary times and in arbitrary order, in the latter part of this thesis. The type system still guarantees that buffering is correctly performed. Figure 1.3 shows a program which is equivalent to one in Figure 1.2 and which uses buffered trees. **mleaf** and **mnode** are constructors of buffered trees and **mcase** expression is a pattern-matching for a buffered tree. The program is well-typed because buffered trees x_1 and x_2 can be used in arbitrary manner. Though efficiency is reduced, we can deal with more programs, especially programs that buffer only a part of input trees, with that extension. For the convenience of programmers, we also provide an algorithm that automatically determine which trees are to be buffered.

The rest of this thesis is organized as follows: To clarify the essence, we first focus on a minimal calculus in Chapter 2–4. In Chapter 2, we define the source language and the target language of the translation. We define a type system of the source language in Chapter 3. Chapter 4 presents a translation algorithm, shows its correctness and discuss the improvement gained by the translation. Chapter 5 describes extensions to allow buffering of trees. After discussing related work in Chapter 6, we conclude in Chapter 7.

This is a joint work with Koichi Kodama and Naoki Kobayashi. The work in Chapter 1 through Chapter 4 is done in cooperation. Chapter 5 is the original work of the author.

Chapter 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 2.1.

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. ($\textbf{case}\ M\ \textbf{of leaf}\ x \Rightarrow M_1 \mid \textbf{node}\ x_1\ x_2 \Rightarrow M_2$) 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 and x_2 are bound to the left and right children respectively and M_2 is evaluated. $\textbf{fix}\ f.M$ 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 $\textbf{let}\ x = M_1\ \textbf{in}\ M_2$ 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.

Terms, values and evaluation contexts:

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

Reduction rules:

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

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

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

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

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

Figure 2.1: 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.

2.2 Target Language

The syntax and operational semantics of the target language is summarized in Figure 2.2. 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** $\Rightarrow e_1$ | **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.

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 2.3 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 $\mathbf{node}; 1; 2$), the execution gets stuck.

Terms, values and evaluation contexts:

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

Reduction rules:

$$(E_t[v_1 + v_2], S_i, S_o) \longrightarrow (E_t[\mathit{plus}(v_1, v_2)], S_i, S_o) \qquad (\text{ET-PLUS})$$

$$(E_t[(\lambda x.M)v], S_i, S_o) \longrightarrow (E_t[[v/x]M], S_i, S_o) \qquad (\text{ET-APP})$$

$$(E_t[\textbf{fix}\ f.e], S_i, S_o) \longrightarrow (E_t[[\textbf{fix}\ f.e/f]e], S_i, S_o) \qquad (\text{ET-FIX})$$

$$(E_t[\textbf{read}()], v; S_i, S_o) \longrightarrow (E_t[v], S_i, S_o) \qquad (\text{ET-READ})$$

$$(E_t[\textbf{write}\ v], S_i, S_o) \longrightarrow (E_t[()], S_i, S_o; v) \qquad (\text{when } v \text{ is an integer, } \textbf{leaf} \text{ or } \textbf{node}) \qquad (\text{ET-WRITE})$$

$$(E_t[\textbf{case leaf of leaf} \Rightarrow e_1 \mid \textbf{node} \Rightarrow e_2], S_i, S_o) \longrightarrow (E_t[e_1], S_i, S_o) \ (\text{ET-CASE1})$$

$$(E_t[\textbf{case node of leaf} \Rightarrow e_1 \mid \textbf{node} \Rightarrow e_2], S_i, S_o) \longrightarrow (E_t[e_2], S_i, S_o) \ (\text{ET-CASE2})$$

Figure 2.2: The reduction rules of the target language.

A source program:

$\textbf{fix}\ \mathit{sumtree}.\lambda t.(\textbf{case}\ t\ \textbf{of leaf}\ x \Rightarrow x \mid \textbf{node}\ x_1\ x_2 \Rightarrow (\mathit{sumtree}\ x_1) + (\mathit{sumtree}\ x_2))$

A target program:

$\textbf{fix}\ \mathit{sumtree}.\lambda t.(\textbf{case read}()\ \textbf{of leaf} \Rightarrow \textbf{read}() \mid \textbf{node} \Rightarrow \mathit{sumtree}\ () + \mathit{sumtree}\ ())$

Figure 2.3: Programs that calculate the sum of leaf elements of an binary tree.

Chapter 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 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 [20, 22].

3.1 Type and Type Environment

Definition 3.1 (Type) The set of *types*, ranged over by τ, is defined by:

$$\begin{array}{rcl} \tau \text{ (type)} & ::= & \mathbf{Int} \mid \mathbf{Tree}^d \mid \tau_1 \to \tau_2 \\ d \text{ (use)} & ::= & 1 \mid +. \end{array}$$

Int is the type of integers. For a technical reason, we distinguish between input trees and output trees by types. We write $\mathbf{Tree^1}$ 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 3.2 (Ordered Linear Type Environment) An *ordered linear type environment* is a sequence of the form $x_1 : \mathbf{Tree^1}, \ldots, x_n : \mathbf{Tree^1}$, where $x_1, \ldots, x_n$ are different from each other. We write Δ_1, Δ_2 for the concatenation of Δ_1 and Δ_2.

An ordered linear type environment $x_1 : \mathbf{Tree^1}, \ldots, x_n : \mathbf{Tree^1}$ specifies not only that $x_1, \ldots, x_n$ are bound to trees, but also that each of $x_1, \ldots, x_n$ must be accessed exactly once in this order and that each of the subtrees bound to $x_1, \dots, x_n$ must be accessed in the left-to-right, depth-first order.

Definition 3.3 (Non-Ordered Type Environment) *A* (non-ordered) type environment *is a set of the form* $\{x_1 : \tau_1, \dots, x_n : \tau_n\}$ *where* $x_1, \dots, x_n$ *are different from each other and* $\{\tau_1, \dots, \tau_n\}$ *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 $x_1 : \tau_1, \dots, x_n : \tau_n$ for $\{x_1 : \tau_1, \dots, x_n : \tau_n\}$.

Note that a non-ordered type environment must not contain variables of tree types. $\mathbf{Tree}^{\mathbf{1}}$ 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** $x_1\ x_2$ when t_1 and t_2 have type $\mathbf{Tree}^{+}$. This restriction is convenient for ensuring that trees are constructed in the specific (from left to right, and in the depth-first manner) order.

3.2 Type Judgment

A type judgment 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 $\Gamma = \{f : \mathbf{Tree}^{\mathbf{1}} \to \mathbf{Tree}^{+}\}$ and $\Delta = x_1 : \mathbf{Tree}^{\mathbf{1}}, x_2 : \mathbf{Tree}^{\mathbf{1}}$,

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

holds, while

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

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 3.1. 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.

$$\Gamma \mid x : \mathbf{Tree^1} \vdash x : \mathbf{Tree^1} \tag{T-Var1}$$

$$\Gamma, x : \tau \mid \emptyset \vdash x : \tau \tag{T-Var2}$$

$$\Gamma \mid \emptyset \vdash i : \mathbf{Int} \tag{T-Int}$$

$$\frac{\Gamma \mid x : \mathbf{Tree^1} \vdash M : \tau}{\Gamma \mid \emptyset \vdash \lambda x.M : \mathbf{Tree^1} \to \tau} \tag{T-Abs1}$$

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

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

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

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

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

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

$$\frac{\begin{array}{c}\Gamma \mid \Delta_1 \vdash M : \mathbf{Tree^1} \qquad \Gamma, x : \mathbf{Int} \mid \Delta_2 \vdash M_1 : \tau \\ \Gamma \mid x_1 : \mathbf{Tree^1}, x_2 : \mathbf{Tree^1}, \Delta_2 \vdash M_2 : \tau\end{array}}{\Gamma \mid \Delta_1, \Delta_2 \vdash \mathbf{case}\ M\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2 : \tau} \tag{T-Case}$$

Figure 3.1: Rules of typing judgment

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \mid t : \mathbf{Tree^1} \vdash t : \mathbf{Tree^1}} \quad \dfrac{\vdots}{\Gamma' \mid \emptyset \vdash \mathbf{leaf}\ (x+1) : \mathbf{Tree}^+} \quad \dfrac{\dfrac{\vdots}{\Gamma \mid x_1 : \mathbf{Tree^1} \vdash (f\ x_1) : \mathbf{Tree}^+} \quad \dfrac{\vdots}{\Gamma \mid x_2 : \mathbf{Tree^1} \vdash (f\ x_2) : \mathbf{Tree}^+}}{\Gamma \mid x_1 : \mathbf{Tree^1}, x_2 : \mathbf{Tree^1} \vdash \mathbf{node}\ (f\ x_1)\ (f\ x_2) : \mathbf{Tree}^+}}{\Gamma \mid t : \mathbf{Tree^1} \vdash \mathbf{case}\ t\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow \mathbf{leaf}\ (x+1) \mid \mathbf{node}\ x_1\ x_2 \Rightarrow \mathbf{node}\ (f\ x_1)\ (f\ x_2) : \mathbf{Tree}^+}}{\Gamma \mid \emptyset \vdash \lambda t.\mathbf{case}\ t\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow \mathbf{leaf}\ (x+1) \mid \mathbf{node}\ x_1\ x_2 \Rightarrow \mathbf{node}\ (f\ x_1)\ (f\ x_2) : \mathbf{Tree^1} \to \mathbf{Tree}^+}}{\emptyset \mid \emptyset \vdash \mathbf{fix}\ f.\lambda t.\mathbf{case}\ t\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow \mathbf{leaf}\ (x+1) \mid \mathbf{node}\ x_1\ x_2 \Rightarrow \mathbf{node}\ (f\ x_1)\ (f\ x_2) : \mathbf{Tree^1} \to \mathbf{Tree}^+}$$

Figure 3.2: An example of typing derivation. $\Gamma = \{f : \mathbf{Tree^1} \to \mathbf{Tree}^+\}$, $\Gamma' = \{f : \mathbf{Tree^1} \to \mathbf{Tree}^+, x : \mathbf{Int}\}$

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 2.3. 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 $x_1 : \mathbf{Tree^1}, x_2 : \mathbf{Tree^1}, \Delta_2$, 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.

[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.

The program in Figure 1.1 is typed as shown in Figure 3.2. On the other hand, the program in Figure 1.2 is ill-typed: $\Gamma \mid x_1 : \mathbf{Tree^1}, x_2 : \mathbf{Tree^1} \vdash \mathbf{node}\ (f\ x_2)\ (f\ x_1) : \mathbf{Tree}^+$ 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 3.3 shows more examples of well-typed source programs. The first and second programs (or the catamorphism [16]) apply the same operation on every node of the input tree. (The return value of the function *tree_fold* cannot, however, be a tree because the value is passed to g.) One can also write functions that process nodes in a non-uniform manner, like the third program in Figure 3.3 (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 sub-functions *copy_tree* and *skip_tree* for explicitly copying and skipping trees.[2] (See Section 5.2 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 *lazily* [11]. To make the lazy splitting of the ordered type environment explicit, we use a relation $\Gamma \mid \Delta \vdash M{:}\tau \nearrow \Delta'$, which is defined as the least relation that satisfies the rules in Figure 3.5. The relation means that $\Gamma \mid \Delta_1 \vdash M : \tau$ holds for Δ_1 such that $\Delta = \Delta_1, \Delta'$. As the specification

[2]Due to the restriction that lambda abstractions cannot contain variables of type $\mathbf{Tree}^d$, we need to introduce sequential composition (;) as a primitive and extend typing rules with the following rule:

$$\frac{\Gamma \mid \Delta_1 \vdash M_1 : \tau' \qquad \Gamma \mid \Delta_2 \vdash M_2 : \tau \qquad \tau' \neq \mathbf{Tree}^d}{\Gamma \mid \Delta_1, \Delta_2 \vdash M_1; M_2 : \tau.} \tag{T-Seq}$$

```
fix $tree\_map.\lambda f.\lambda t.$(case $t$ of leaf $x \;\Rightarrow\;$ leaf $(f\ x)$
                                | node $x_1\ x_2 \;\Rightarrow$ node $(tree\_map\ f\ x_1)\ (tree\_map\ f\ x_2))$
fix $tree\_fold.\lambda f.\lambda g.\lambda t.$
  (case $t$ of leaf $n \;\Rightarrow\; (f\ n)$
          | node $t_1\ t_2 \;\Rightarrow\; (g\ (tree\_fold\ f\ g\ t_1)(tree\_fold\ f\ g\ t_2)))$
fix $inc\_alt.\lambda t.$(case $x$ of leaf $x \Rightarrow$ leaf $x$ | node $x_1\ x_2 \Rightarrow$ node
      (case $x_1$ of leaf $y \Rightarrow$ leaf $(y+1)$
            | node $y_1\ y_2 \Rightarrow$ node $(inc\_alt\ y_1)\ (inc\_alt\ y_2))$
      (case $x_2$ of leaf $z \Rightarrow$ leaf $(z+1)$
            | node $z_1\ z_2 \Rightarrow$ node $(inc\_alt\ z_1)\ (inc\_alt\ z_2))$

 let $copy\_tree =$
    fix $copy\_tree.\lambda t.$(case $t$ of leaf $x \;\Rightarrow\;$ leaf $x$
                       | node $x_1\ x_2 \;\Rightarrow\;$ node $(copy\_tree\ x_1)\ (copy\_tree\ x_2))$ in
 let $skip\_tree =$
    fix $skip\_tree.\lambda t.$(case $t$ of leaf $x \;\Rightarrow\; 0$
                       | node $x_1\ x_2 \;\Rightarrow\; (skip\_tree\ x_1); (copy\_tree\ x_2)$ in
    $\lambda t.$(case $t$ of leaf $x \Rightarrow$ leaf $x$ | node $x_1\ x_2 \Rightarrow (skip\_tree\ x_1); (copy\_tree\ x_2))$
```

Figure 3.3: Examples of well-typed programs.

```
fix $f.\lambda t.$case $t$ of leaf $x \Rightarrow$ leaf $x$ | node $x_1\ x_2 \Rightarrow (\lambda t'.$node $(f\ t')\ (f\ x_2))\ x_1$
```

Figure 3.4: Program that is not typed due to the restriction on closures though the access order is correct.

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 $\Gamma = f : \mathbf{Tree^1} \to \mathbf{Tree}^{+}$ and $\Delta = x_1 : \mathbf{Tree^1}, x_2 : \mathbf{Tree^1}$,

$$\Gamma \mid \Delta \vdash (f\ x_1) : \mathbf{Tree}^{+} \nearrow x_2 : \mathbf{Tree^1}$$

holds.

The relation above specifies the type-checking algorithm. For example, T-App in Figure 3.5 specifies, 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 Figure 3.5.

$$\frac{}{\Gamma \mid x : \mathbf{Tree^1}, \Delta \vdash x : \mathbf{Tree^1} \nearrow \Delta} \quad \text{(T-Var1)}$$

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

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

$$\frac{\Gamma \mid x : \mathbf{Tree^1}, \Delta \vdash M : \tau \nearrow \Delta}{\Gamma \mid \Delta \vdash \lambda x.M : \mathbf{Tree^1} \to \tau \nearrow \Delta} \quad \text{(T-Abs1)}$$

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

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

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

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

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

$$\frac{\begin{array}{c}\Gamma \mid \Delta \vdash M : \mathbf{Tree^1} \nearrow \Delta' \qquad \Gamma, x : \mathbf{Int} \mid \Delta' \vdash M_1 : \tau \nearrow \Delta'' \\ \Gamma \mid x_1 : \mathbf{Tree^1}, x_2 : \mathbf{Tree^1}, \Delta' \vdash M_2 : \tau \nearrow \Delta''\end{array}}{\Gamma \mid \Delta \vdash (\mathbf{case}\ M\ \mathbf{of\ leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2) : \tau \nearrow \Delta''} \quad \text{(T-Case)}$$

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

Figure 3.5: Rules for Type Checking

Chapter 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

The translation algorithm $\mathcal{A}$ is shown in Figure 4.1. $\mathcal{A}$ 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.

Definition 4.1 A function $[\![\,\cdot\,]\!]$ from the set of trees to the set of streams is defined by:

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

Theorem 4.1 (Correctness of Translation)
If $\emptyset \mid \emptyset \vdash M : \mathbf{Tree^1} \rightarrow \tau$ *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* $(\mathcal{A}(M)(), [\![\,V\,]\!], \emptyset) \longrightarrow^{*} ((), \emptyset, [\![\,V'\,]\!])$.

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

Figure 4.1: Translation Algorithm

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 4.1 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 4.1 below), and (2) evaluation of the source program based on the new semantics agrees with evaluation of the corresponding target program (Theorem 4.3 below). First, we define a new operational semantics of the source language. The semantics takes the access order of input trees into account.

Definition 4.2 (Ordered Environment) An ordered environment is a sequence of the form $x_1 \mapsto V_1, \dots, x_n \mapsto V_n$, where $x_1, \dots, x_n$ are distinct from each other.

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

Definition 4.3 (New Reduction Semantics) *The reduction relation* $(M, \delta) \longrightarrow (M', \delta')$ *is the least relation that satisfies the rules in Figure 4.2.*

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

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

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

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

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

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

Figure 4.2: The new reduction semantics of the source language.

The meta-variable U in Figure 4.2 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 4.2) states, that cannot happen if the program is well-typed, so that both semantics are equivalent for well-typed programs (Corollary 4.1).

Theorem 4.2 *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 4.1 *If* $\emptyset \mid \emptyset \vdash M : \mathbf{Tree^1} \to \tau$ *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 4.3 *If* $\emptyset \mid \emptyset \vdash M : \mathbf{Tree^1} \to \tau$ *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*
$(\mathcal{A}(M)(), [\![V]\!], \emptyset) \longrightarrow^* ((), \emptyset, [\![V']\!]))$

We hereafter give an outline of the proof of Theorem 4.3. Figure 4.4 illustrates the idea of the proof (for the case where the result is a tree). The relation $\sim$ (defined later in Definition 4.5) 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 initial state of the source program M (Lemma 4.1 below) and that reductions and the correspondence relation commute (Lemma 4.2). Those imply that the whole diagram in Figure 4.4 commutes, i.e., the second statement of Theorem 4.3 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 4.4 A function $\langle\cdot\rangle$ from the set of ordered environments to the set of streams is defined by:

$$\begin{array}{rcl} \langle\emptyset\rangle & = & \emptyset \\ \langle x \mapsto V, \delta\rangle & = & [\![V]\!]; \langle\delta\rangle \end{array}$$

Definition 4.5 (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 4.3.

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 4.3 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.

$$I \quad ::= \quad (\lambda x.M)U \mid i_1 + i_2 \mid (\mathbf{case}\ y\ \mathbf{of\ leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2) \mid \mathbf{fix}\ f.M$$

$$\frac{M \sim_{FV(M)} (e, S_o) \qquad S_i = \langle \delta \rangle}{(M, \delta) \sim (e, S_i, S_o)}$$

$$\frac{}{U \sim_\gamma (\mathcal{A}_\gamma(U), \emptyset)} \ \text{(C-Value)}$$

$$\frac{}{V \sim_\gamma ((), \llbracket V \rrbracket)} \ \text{(C-Tree)}$$

$$\frac{}{I \sim_\gamma (\mathcal{A}_\gamma(I), \emptyset)} \ \text{(C-Inst)}$$

$$\frac{E_s[I] \sim_\gamma (e, S_o)}{E_s[I]\ M \sim_\gamma (e\ \mathcal{A}_\gamma(M), S_o)} \ \text{(C-App1)}$$

$$\frac{E_s[I] \sim_\gamma (e, S_o)}{(\lambda x.M)\ E_s[I] \sim_\gamma ((\lambda x.\mathcal{A}_\gamma(M))\ e, S_o)} \ \text{(C-App2)}$$

$$\frac{E_s[I] \sim_\gamma (e, S_o)}{E_s[I] + M \sim_\gamma (e + \mathcal{A}_\gamma(M), S_o)} \ \text{(C-Plus1)}$$

$$\frac{E_s[I] \sim_\gamma (e, S_o)}{i + E_s[I] \sim_\gamma (i + e, S_o)} \ \text{(C-Plus2)}$$

$$\frac{E_s[I] \sim_\gamma (e, S_o)}{\mathbf{leaf}\ E_s[I] \sim_\gamma (\mathbf{write}(e), \mathbf{leaf}; S_o)} \ \text{(C-Leaf)}$$

$$\frac{E_s[I] \sim_\gamma (e, S_o)}{\mathbf{node}\ E_s[I]\ M \sim_\gamma (e; \mathcal{A}_\gamma(M), \mathbf{node}; S_o)} \ \text{(C-Node1)}$$

$$\frac{E_s[I] \sim_\gamma (e, S_o)}{\mathbf{node}\ V\ E_s[I] \sim_\gamma (e, \mathbf{node}; \llbracket V \rrbracket; S_o)} \ \text{(C-Node2)}$$

$$\frac{E_s[I] \sim_\gamma (e, S_o)}{\begin{array}{c}\mathbf{case}\ E_s[I]\ \mathbf{of\ leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2 \\ \sim_\gamma (\mathbf{case}\ e; \mathbf{read}\ ()\ \mathbf{of\ leaf} \Rightarrow \mathbf{let}\ x = \mathbf{read}\ ()\ \mathbf{in}\ \mathcal{A}_\gamma(M_1) \mid \mathbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2), S_o)\end{array}} \ \text{(C-Case)}$$

Figure 4.3: Correspondence between run-time states of source and target programs.

$$\begin{tikzcd}
 & (Mx, x \mapsto V) \arrow[r] \arrow[dl, bend right, "\text{Lemma 4.1}"] \arrow[d, phantom, "\wr"] & (M', \delta') \arrow[r, "\cdots"] \arrow[d, phantom, "\wr"] \arrow[l, phantom, "\text{Lemma 4.2}"] & (M'', \delta'') \arrow[r] \arrow[d, phantom, "\wr"] & (V', \emptyset) \arrow[d, phantom, "\wr"] \arrow[l, phantom, "\text{Lemma 4.2}"] \\
(\mathcal{A}(M), [\![V]\!], \emptyset) \arrow[r, "*"] & (e, [\![V]\!], S_o) \arrow[r, "+"] & (e', S'_i, S'_o) \arrow[r, "+ \cdots +"] & (e'', S''_i, S''_o) \arrow[r, "+"] & ((), \emptyset, [\![V']\!])
\end{tikzcd}$$

Figure 4.4: Evaluation of a source and the target program.

- C-NODE1: A source program **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 **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 4.1 and 4.2 below imply that the whole diagram in Figure 4.3 commutes, which completes the proof of Theorem 4.3.

Definition 4.6 A function $\langle\langle\cdot\rangle\rangle$ from the set of ordered environments to the set of ordered linear type environments is defined by:

$$\begin{array}{rcl} \langle\langle \emptyset \rangle\rangle & = & \emptyset \\ \langle\langle x \mapsto V, \delta \rangle\rangle & = & x : \mathbf{Tree^1}, \langle\langle \delta \rangle\rangle \end{array}$$

Lemma 4.1 *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 4.2 *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 $\mathbf{Tree^1} \rightarrow \mathbf{Tree^+}$. We argue below that the target program $\mathcal{A}(M)$ runs more efficiently than the source program *unparse* $\circ\ M \circ$ *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 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:[1]

- 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).

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

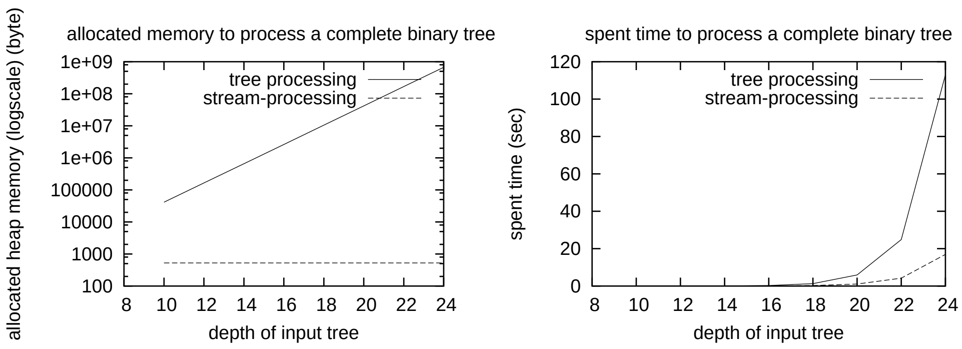


Figure 4.5: Result of experiments. Inputs are binary trees whose height varies from 10 to 24. The experiment was performed on Sun Enterprise E4500/E5500, 400 MHz CPU, 13GB memory.

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 4.5 shows the execution time and the total size of allocated heap 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.

Chapter 5

Introducing Selective Buffering of Input Trees

So far, we have focused on a minimal calculus which does not allow any buffering of trees. In this chapter, we introduce buffering constructs to our languages. With this extension, we can deal with more programs than the original calculus at the cost of memory efficiency.

5.1 Constructs for Storing Trees on Memory

Let us extend the syntax of the source and target languages as follows:

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

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

We also add type $\mathbf{Tree}^{\omega}$, the type of trees stored on memory. The type system imposes no restriction on access order between variables of type $\mathbf{Tree}^{\omega}$ like type **Int** (so $\mathbf{Tree}^{\omega}$ 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:

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

These extensions are summarized in Figure 5.1.

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 5.2.

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 5.3, 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 4.1 can be easily adapted for the extended language.

5.2 Automatic Insertion of Buffering Primitives

As we stated in 3.3 and 5.1, 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. To solve this problem, we provide a type-directed, source-to-source translation for automatically inserting these functions.

From this section, we write $\mathbf{fix}\ (f, x, M)$ for $\mathbf{fix}\ f.\lambda x.M$. With that new form, we treat $\lambda x.M$ as a syntax-sugar that represents $\mathbf{fix}\ (f, x, M)$ where $f \notin \mathbf{FV}(M)\backslash\{x\}$.

5.2.1 Type Judgment for the Automatic Insertion

We introduce a new judgment $\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \tau$ which means that (1) M can be translated to an equivalent program M' (except for the difference of buffered trees and ordinary trees) which uses *skip_tree*, *copy_tree*, *strm_to_mem* and *mem_to_strm* and (2) $\Gamma \mid \Delta M' : \tau$ holds. For example, both of

$$\emptyset \mid x_1 : \mathbf{Tree^1}, x_2 : \mathbf{Tree^1} \vdash x_2 \rightsquigarrow \mathit{skip\_tree}(x_1); \mathit{copy\_tree}(x_2) : \mathbf{Tree}^+$$

and

$$\emptyset \mid x_1 : \mathbf{Tree^1}, x_2 : \mathbf{Tree^1} \vdash x_2 \rightsquigarrow \begin{array}{l}\mathbf{let}\ x_1 = \mathit{strm\_to\_mem}(x_1)\ \mathbf{in} \\ \quad \mathit{copy\_tree}(x_2)\end{array} : \mathbf{Tree}^+$$

holds. As we can see in the above examples, the way of the translation is not unique. We will present an algorithm which selects one of possible translations in Section 5.2.2.

The relation $\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \tau$ is defined as the least relation satisfies Figure 5.4 and Figure 5.5. We explain some of those rules below:

$$
\begin{array}{rcl}
M & ::= & i \mid \lambda x.M \mid x \mid M_1\ M_2 \mid M_1 + M_2 \\
 & \mid & \textbf{leaf}\ M \mid \textbf{node}\ M_1\ M_2 \mid \textbf{mleaf}\ M \mid \textbf{mnode}\ M_1\ M_2 \\
 & \mid & (\textbf{case}\ M\ \textbf{of leaf}\ x \Rightarrow M_1 \mid \textbf{node}\ x_1\ x_2 \Rightarrow M_2) \\
 & \mid & (\textbf{mcase}\ M\ \textbf{of mleaf}\ x \Rightarrow M_1 \mid \textbf{mnode}\ x_1\ x_2 \Rightarrow M_2) \\
 & \mid & \textbf{fix}\ f.M \\
e & ::= & v \mid x \mid e_1\ e_2 \mid e_1 + e_2 \\
 & \mid & \textbf{read}\ e \mid \textbf{write}\ e \\
 & \mid & \textbf{mleaf}\ M \mid \textbf{mnode}\ M_1\ M_2 \\
 & \mid & (\textbf{mcase}\ M\ \textbf{of mleaf}\ x \Rightarrow M_1 \mid \textbf{mnode}\ x_1\ x_2 \Rightarrow M_2) \\
 & \mid & (\textbf{case}\ e\ \textbf{of leaf} \Rightarrow e_1 \mid \textbf{node} \Rightarrow e_2) \\
 & \mid & \textbf{fix}\ f.e
\end{array}
$$

$$
\frac{\Gamma \mid \Delta \vdash M : \textbf{Int}}{\Gamma \mid \Delta \vdash \textbf{mleaf}\ M : \textbf{Tree}^{\omega}} \quad (\text{T-ML}\textsc{eaf})
$$

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

$$
\frac{\Gamma \mid \Delta_1 \vdash M : \textbf{Tree}^{\omega} \qquad \Gamma, x : \textbf{Int} \mid \Delta_2 \vdash M_1 : \tau \qquad \Gamma, x_1 : \textbf{Tree}^{\omega}, x_2 : \textbf{Tree}^{\omega} \mid \Delta_2 \vdash M_2 : \tau}{\Gamma \mid \Delta_1, \Delta_2 \vdash \textbf{mcase}\ M\ \textbf{of mleaf}\ x \Rightarrow M_1 \mid \textbf{mnode}\ x_1\ x_2 \Rightarrow M_2 : \tau} \quad (\text{T-MC}\textsc{ase})
$$

$$
\begin{array}{l}
\mathcal{A}(\textbf{mleaf}\ M) = \textbf{mleaf}\ \mathcal{A}(M) \\
\mathcal{A}(\textbf{mnode}\ M_1\ M_2) = \textbf{mnode}\ \mathcal{A}(M_1)\ \mathcal{A}(M_2) \\
\mathcal{A}(\textbf{mcase}\ M\ \textbf{of mleaf}\ x \Rightarrow M_1 \mid \textbf{mnode}\ x_1\ x_2 \Rightarrow M_2) = \\
\qquad \textbf{mcase}\ \mathcal{A}(M)\ \textbf{of mleaf}\ x \Rightarrow \mathcal{A}(M_1) \mid \textbf{mnode}\ x_1\ x_2 \Rightarrow \mathcal{A}(M_2)
\end{array}
$$

Figure 5.1: Extended languages, type system and translation algorithm

```
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)
```

Figure 5.2: Definition of *strm_to_mem* and *mem_to_strm*

```
let mswap =
    fix f.λ t.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)
```

Figure 5.3: A program which swaps children of nodes whose depth is more than n

- TR-NODE1 and TR-NODE2: A program **node** M_1 M_2 can be translated in two ways: into an ordinary tree or into a buffered tree. We can deal with programs that apply a function to $\mathbf{Tree}^{+}$ value with these rules.
- TR-SKIP: We can deal with a program M which does not use the top of the ordered linear type environment $x : \mathbf{Tree^1}, \Delta$ by inserting skip operation on x.
- TR-COPY and TR-MEMTOSTREAM: We can coerce a value of type $\mathbf{Tree^1}$ and a value of type $\mathbf{Tree}^{\omega}$ to type $\mathbf{Tree}^{+}$ by applying *copy_tree*() and *strm_to_mem*() to the value.
- TR-STREAMTOMEM: We can buffer a tree bound to x with *strm_to_mem*() function before it is used if x is at the top of the ordered linear type environment.

The following theorem states that the judgment defines a correct translation:

Theorem 5.1 *If* $x_1 : \tau_1, \ldots, x_n : \tau_n \mid \langle\langle\delta\rangle\rangle \vdash M \rightsquigarrow M' : \tau$ *holds, then:*

- $x_1 : \tau_1, \ldots, x_n : \tau_n \mid \langle\langle\delta\rangle\rangle \vdash M' : \tau$

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{c}\Gamma \mid \Delta_1 \vdash M \leadsto M' : \mathbf{Tree^1} \\ x : \mathbf{Int}, \Gamma \mid \Delta_2 \vdash M_1 \leadsto M_1' : \tau \qquad x_1 : \mathbf{Tree}^\omega, x_2 : \mathbf{Tree}^\omega, \Gamma \mid \Delta_2 \vdash M_2 \leadsto M_2' : \tau\end{array}}{\begin{array}{c}\Gamma \mid \Delta_1, \Delta_2 \vdash \mathbf{case}\ M\ \mathbf{of\ leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2 \\ \leadsto \mathbf{case}\ M'\ \mathbf{of\ leaf}\ x \Rightarrow M_1' \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2' : \tau\end{array}} \qquad \text{(Tr-Case1)}$$

$$\frac{\begin{array}{c}\Gamma \mid \Delta_1 \vdash M \leadsto M' : \mathbf{Tree}^\omega \\ x : \mathbf{Int}, \Gamma \mid \Delta_2 \vdash M_1 \leadsto M_1' : \tau \qquad x_1 : \mathbf{Tree}^\omega, x_2 : \mathbf{Tree}^\omega, \Gamma \mid \Delta_2 \vdash M_2 \leadsto M_2' : \tau\end{array}}{\begin{array}{c}\Gamma \mid \Delta_1, \Delta_2 \vdash \mathbf{case}\ M\ \mathbf{of\ leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2 \\ \leadsto \mathbf{mcase}\ M'\ \mathbf{of\ mleaf}\ x \Rightarrow M_1' \mid \mathbf{mnode}\ x_1\ x_2 \Rightarrow M_2' : \tau\end{array}} \qquad \text{(Tr-Case2)}$$

Figure 5.4: Rules for source-to-source translation (1 of 2).

$$\frac{\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \tau}{\Gamma \mid x : \mathbf{Tree^1}, \Delta \vdash M \rightsquigarrow \mathit{skip\_tree}(x); M' : \tau} \quad (\textsc{Tr-Skip})$$

$$\frac{\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \mathbf{Tree^1}}{\Gamma \mid \Delta \vdash M \rightsquigarrow \mathit{copy\_tree}(M') : \mathbf{Tree}^{+}} \quad (\textsc{Tr-Copy})$$

$$\frac{x : \mathbf{Tree}^{\omega}, \Gamma \mid \Delta \vdash M \rightsquigarrow M' : \tau}{\Gamma \mid x : \mathbf{Tree^1}, \Delta \vdash M \rightsquigarrow \mathbf{let}\ x = \mathit{strm\_to\_mem}(x)\ \mathbf{in}\ M' : \tau} \quad (\textsc{Tr-StreamToMem})$$

$$\frac{\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \mathbf{Tree}^{\omega}}{\Gamma \mid \Delta \vdash M \rightsquigarrow \mathit{mem\_to\_strm}(M') : \mathbf{Tree}^{+}} \quad (\textsc{Tr-MemToStream})$$

Figure 5.5: Rules for source-to-source translation (2 of 2).

- *If* $([W_1/x_1, \ldots, W_n/x_n]M, \delta) \longrightarrow W$ *and* $([W_1/x_1, \ldots, W_n/x_n]M', \delta) \longrightarrow W'$, *then* $W \approx_s W'$ *for all values* $W_1, \ldots, W_n$ *that respect* $\tau_1, \ldots, \tau_n$.

$\approx_s$ *is defined as the least equivalence relation that satisfies the following rules:*

$$\mathbf{leaf}\ i \approx_s \mathbf{mleaf}\ i$$

$$\frac{V_1 \approx_s V_1' \qquad V_2 \approx_s V_2'}{\mathbf{node}\ V_1\ V_2 \approx_s \mathbf{mnode}\ V_1'\ V_2'}$$

5.2.2 Automatic Insertion Algorithm

As we saw in the examples in Section 5.2.1, given Γ, Δ and M, there are many possible M' and τ that satisfy $\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \tau$. We next present an algorithm $\mathcal{I}$ which chooses one among those possibilities.

Note that the algorithm $\mathcal{I}$ defined in the following is not optimal one because it use some approximations. Designing a better algorithm is left as future work.

The rules in Figure 5.4 and Figure 5.5 are not convenient for designing $\mathcal{I}$ because:

- they are not syntax-directed. For example, Tr-Skip can be applied to terms of any form.
- they use two kind of type environment. $\mathcal{I}$ cannot determine in which environment a variable of tree type is to be stored because use of a tree is not known until constraints are solved.

To solve those difficulty, we introduce a new type environment Θ and a new typing judgment $\Theta \vdash M \rightsquigarrow M' : \tau$. Before defining Θ and $\Theta \vdash M \rightsquigarrow M' : \tau$, we need to define use of types, a relation $\succeq$ and a predicate $\mathit{merge}(\Theta, \Theta_1, \Theta_2)$.

Definition 5.1 *The relation $d_1 \geq d_2$ is the least reflexive transitive relation that satisfies $\omega \geq 1$. $|\cdot|$ is a function defined as follows:*

$$\begin{array}{rcl} |\mathbf{Int}| & = & \omega \\ |\tau_1 \rightarrow \tau_2| & = & \omega \\ |\mathbf{Tree}^d| & = & d \end{array}$$

We write $|\Theta| \geq d$ for $\forall x \in \mathit{dom}(\Theta).\ |\Theta(x)| \geq d$. $|\Theta| \leq d$ is defined in the same way.

Definition 5.2 (Semi-ordered type environment) *A* semi-ordered type environment*, represented by Θ, is a sequence $x_1 : \tau_1, \ldots, x_n : \tau_n$ that satisfies $|\tau_i| \geq |\tau_j|$ if $i \leq j$. We write $x \gt_\Theta y$ if x occurs before y in Θ.*

A semi-ordered type environment Θ represents that variables bound to $\mathbf{Tree^1}$ in Θ have to be accessed in the order specifies by Θ and other variables can be freely used. We put the restriction $i \leq j \implies |\tau_i| \geq |\tau_j|$ in order to let constraint solving tractable.

Definition 5.3 *The relation $\succeq$ is the smallest reflexive transitive relation which satisfies $\Theta_1, x : \mathbf{Tree}^\omega, \Theta_2 \succeq \Theta_1, x : \mathbf{Tree^1}, \Theta_2$ and $|\Theta_1| \geq \omega$.*

$\Theta_1 \succeq \Theta_2$ intuitively means that Θ_1 is obtained by buffering some trees in Θ_2. Because only the first tree can be buffered, we need $|\Theta_1| \geq \omega$ in the definition above.

Definition 5.4 (Merge of semi-ordered type environments) *Θ is a* merge *of Θ_1 and Θ_2, represented by $\mathit{merge}(\Theta, \Theta_1, \Theta_2)$, if and only if the following properties are satisfied:*

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

$$
\begin{array}{rcl}
\mathit{coerce}^{\Theta \to \Theta}(M) & = & M \\
\mathit{coerce}^{(\Theta_1, x:\mathbf{Tree^1}, \Theta_2) \to (\Theta_1, x:\mathbf{Tree}^\omega, \Theta_2')}(M) & = & \begin{array}{l}\mathbf{let}\ x = \mathit{strm\_to\_mem}(x)\ \mathbf{in} \\ \quad \mathit{coerce}^{\Theta_2 \to \Theta_2'}(M)\end{array}
\end{array}
$$

Figure 5.6: Definition of the meta-level function $\mathit{coerce}^{\Theta \to \Theta'}()$.

$\mathit{merge}(\Theta, \Theta_1, \Theta_2)$ means that Θ is obtained by (1) concatenating Θ_1 and Θ_2 (condition 1, 2 and 3 in Definition 5.4), (2) adding some variables whose types have use ω (condition 4) and (3) reordering some variables whose types have use ω (condition 5). Note that the condition 4 in Definition 5.4 is not optimal because trees that can be skipped are buffered with the condition.

Definition 5.5 *A relation* $\Theta \vdash M \rightsquigarrow M' : \tau$ *is the least relation that satisfies the rules in Figure 5.7.*

$\mathit{coerce}^{\Theta \to \Theta'}()$, defined in Figure 5.6 is a meta-level function which inserts $\mathit{strm\_to\_mem}()$ in order to convert Θ to Θ'. Note that we use $\mathbf{Tree}^\omega$ in place of $\mathbf{Tree}^+$ for the simplicity of the presentation.

Those rules are obtained from the rules in Figure 5.4 and 5.5 simply by (1) replacing concatenation with merging and (2) applying the following rule for buffering to premises of each rule:

$$\frac{\Theta' \vdash M \rightsquigarrow M' : \tau \qquad \Theta' \succeq \Theta}{\Theta \vdash M \rightsquigarrow \mathit{coerce}^{\Theta \to \Theta'}(M') : \tau}$$

The following theorem guarantees the correctness of $\Theta \vdash M \rightsquigarrow M' : \tau$.

Theorem 5.2 *If* $\Theta \vdash M \rightsquigarrow M' : \tau$ *holds, then there exist* Γ *and* Δ *that satisfy* $\Gamma \mid \Delta \vdash M \rightsquigarrow M' : \tau$ *and* $\Theta = \Gamma, \Delta$.

Now, we define our algorithm $\mathcal{I}$. The algorithm $\mathcal{I}$ works as follows:

1. The type of each subterm is reconstructed ignoring the use d of tree type. We use the type reconstruction algorithm for $\lambda^\to$ [21]. An *use variable*, represented by meta-variable u, is assigned to each tree types.

2. $\mathcal{I}$ constructs a derivation tree according to the rules of Figure 5.7. $\mathcal{I}$ also generates constraints among use variables in this phase.

$$\frac{\forall y \in \mathit{dom}(\Theta')\backslash\{x\}.\ |\Theta'(y)| \geq \omega \qquad \Theta' \succeq \Theta \qquad \tau = \Theta'(x)}{\Theta \vdash x \rightsquigarrow \mathit{coerce}^{\Theta\to\Theta'}(x) : \tau} \quad \text{(T-SD-Var)}$$

$$\frac{\forall x \in \mathit{dom}(\Theta').\ |\Theta'(x)| \geq \omega \qquad \Theta' \succeq \Theta \qquad \tau = \Theta'(x)}{\Theta \vdash i \rightsquigarrow \mathit{coerce}^{\Theta\to\Theta'}(i) : \mathbf{Int}} \quad \text{(T-SD-Int)}$$

$$\frac{\begin{array}{c}\Theta_1 \vdash M_1 \rightsquigarrow M_1' : \mathbf{Int} \qquad \Theta_2 \vdash M_2 \rightsquigarrow M_2' : \mathbf{Int} \\ \Theta_1 \succeq \Theta_1' \qquad \Theta_2 \succeq \Theta_2' \qquad \mathit{merge}(\Theta, \Theta_1', \Theta_2')\end{array}}{\Theta \vdash M_1 + M_2 \rightsquigarrow \mathit{coerce}^{\Theta_1'\to\Theta_1}(M_1') + \mathit{coerce}^{\Theta_2'\to\Theta_2}(M_2') : \mathbf{Int}} \quad \text{(T-SD-Plus)}$$

$$\frac{\begin{array}{c}f : \tau_1 \to \tau_2, \Theta', x : \tau_1 \vdash M \rightsquigarrow M' : \tau_2 \\ f : \tau_1 \to \tau_2, \Theta', x : \tau_1 \succeq f : \tau_1 \to \tau_2, \Theta, x : \tau_1' \qquad \forall y \in \mathit{dom}(\Theta).\ |\Theta(y)| \geq \omega\end{array}}{\Theta \vdash \mathbf{fix}\ (f, x, M) \rightsquigarrow \mathbf{fix}\ (f, x, \mathit{coerce}^{(f:\tau_1\to\tau_2,\Theta',x:\tau_1)\to(f:\tau_1\to\tau_2,\Theta,x:\tau_1')}(M')) : \tau_1 \to \tau_2} \quad \text{(T-SD-Fix)}$$

$$\frac{\begin{array}{c}\Theta_1' \vdash M_1 \rightsquigarrow M_1' : \tau_1 \to \tau_2 \qquad \Theta_2' \vdash M_2 \rightsquigarrow M_2' : \tau_1 \\ \Theta_1' \succeq \Theta_1 \qquad \Theta_2' \succeq \Theta_2 \qquad \mathit{merge}(\Theta, \Theta_1, \Theta_2)\end{array}}{\Theta \vdash M_1\ M_2 \rightsquigarrow \mathit{coerce}^{\Theta_1\to\Theta_1'}(M_1')\ \mathit{coerce}^{\Theta_2\to\Theta_2'}(M_2') : \tau_2} \quad \text{(T-SD-App)}$$

$$\frac{\Theta' \vdash M \rightsquigarrow M' : \mathbf{Int} \qquad \Theta' \succeq \Theta}{\Theta \vdash \mathbf{leaf}\ M \rightsquigarrow \mathbf{mleaf}\ \mathit{coerce}^{\Theta\to\Theta'}(M') : \mathbf{Tree}^{\omega}} \quad \text{(T-SD-Leaf)}$$

$$\frac{\begin{array}{c}\Theta_1' \vdash M_1 \rightsquigarrow M_1' : \mathbf{Tree}^{\omega} \qquad \Theta_2' \vdash M_2 \rightsquigarrow M_2' : \mathbf{Tree}^{\omega} \\ \Theta_1' \succeq \Theta_1 \qquad \Theta_2' \succeq \Theta_2 \qquad \mathit{merge}(\Theta, \Theta_1, \Theta_2)\end{array}}{\Theta \vdash \mathbf{node}\ M_1\ M_2 \rightsquigarrow \mathbf{mnode}\ \mathit{coerce}^{\Theta_1\to\Theta_1'}(M_1')\ \mathit{coerce}^{\Theta_2\to\Theta_2'}(M_2') : \mathbf{Tree}^{\omega}} \quad \text{(T-SD-Node)}$$

$$\frac{\begin{array}{c}\Theta_1' \vdash M \rightsquigarrow M' : \mathbf{Tree}^{d} \qquad \Theta_2' \vdash M_1 \rightsquigarrow M_1' : \tau \qquad \Theta_3' \vdash M_2 \rightsquigarrow M_2' : \tau \\ \Theta_1' \succeq \Theta_1 \qquad \Theta_2' \succeq x : \mathbf{Int}, \Theta_{2L}, \Theta_{2R} \qquad \Theta_3' \succeq \Theta_{2L}, x_1 : \mathbf{Tree}^{d}, x_2 : \mathbf{Tree}^{d}, \Theta_{2R} \\ \mathit{merge}(\Theta, \Theta_1, (\Theta_{2L}, \Theta_{2R}))\end{array}}{\Theta \vdash \begin{array}{l}\mathbf{case}\ M\ \mathbf{of} \\ \quad \mathbf{leaf}\ x \Rightarrow M_1 \\ \ \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2\end{array} \rightsquigarrow \begin{array}{l}\mathbf{case}\ M'\ \mathbf{of} \\ \quad \mathbf{leaf}\ x \Rightarrow \mathit{coerce}^{(\Theta_{2L},\Theta_{2R})\to(\Theta_2'\backslash\{x:\mathbf{Int}\})}(M_1') \\ \ \mid \mathbf{node}\ x_1\ x_2 \Rightarrow \mathit{coerce}^{(\Theta_{2L},x_1:\mathbf{Tree}^{d},x_2:\mathbf{Tree}^{d},\Theta_{2R})\to\Theta_3'}(M_2')\end{array} : \tau} \quad \text{(T-SD-Case)}$$

Figure 5.7: Typing rules for the judgment $\Theta \vdash M \rightsquigarrow M' : \tau$.

$$
\begin{array}{lcl}
\mathit{rename}(\mathbf{Tree}^d) & = & \mathbf{Tree}^{d'} \qquad \mathit{where}\ (d' \text{ is fresh}) \\
\mathit{rename}(\tau) & = & \tau \qquad \mathit{where}\ \tau \neq \mathbf{Tree}^d \\
\\
\mathit{left\_than\_right}(\emptyset) & = & \emptyset \\
\mathit{left\_than\_right}(x : \tau) & = & \emptyset \\
\mathit{left\_than\_right}(x_1 : \tau_1, x_2 : \tau_2, \Theta) & = & \{|\tau_1| \geq |\tau_2|\} \cup \mathit{left\_than\_right}(x_2 : \tau_2, \Theta) \\
\\
\mathit{above\_than\_below}(\emptyset, \emptyset) & = & \emptyset \\
\mathit{above\_than\_below}((x : \tau_1, \Theta_1), (x : \tau_2, \Theta_2)) & = & \{|\tau_1| \geq |\tau_2|\} \cup \mathit{above\_than\_below}(\Theta_1, \Theta_2) \\
\\
\mathit{skip\_unused}(\Theta_1, S) & = & \{|\Theta_1(x)| \geq \omega | x \in \mathit{dom}(\Theta_1) \backslash S\} \\
\\
\mathit{mem\_permut}(\Theta, \emptyset, \Theta') & = & \emptyset \\
\mathit{mem\_permut}((\Theta_L, x : \tau', \Theta_R), (\Theta_1, x : \tau), \Theta_2) & = & \{|\Theta_L(y)| \geq \omega | y \in \mathit{dom}(\Theta_L) \cap \mathit{dom}(\Theta_2)\} \\
\\
\mathit{merge\_constr}(\Theta, \Theta_1, \Theta_2) & = & \mathit{skip\_unused}(\Theta, \mathit{dom}(\Theta_1) \cup \mathit{dom}(\Theta_2)) \cup \\
& & \mathit{mem\_permut}(\Theta, \Theta_1, \Theta_2) \cup \\
& & \{|\Theta(x)| \geq \omega | x \in \mathit{dom}(\Theta_1) \cap \mathit{dom}(\Theta_2)\}
\end{array}
$$

Figure 5.8: Definition of helper functions for automatic insertion algorithm $\mathcal{I}$.

3. $\mathcal{I}$ solves the constraints generated in the previous phase and determine in which point, which trees are to be buffered. Because constraints are inequalities between uses, we can use the constraint solving algorithm of [24]. According to the result, $\mathcal{I}$ appropriately inserts the buffering primitives.

Figure 5.9, 5.10, 5.11 and 5.12 are the definition of I. We give some remarks on the definition.

- $\mathit{above\_than\_below}(\Theta_1, \Theta_2)$ and $\mathit{merge\_constr}(\Theta, \Theta_1, \Theta_2)$ defined in Figure 5.8 generate constraints that corresponds to $\Theta_1 \succeq \Theta_2$ and $\mathit{merge}(\Theta, \Theta_1, \Theta_2)$ respectively.
- $\mathit{left\_than\_right}(\Theta)$ generates constraints that guarantees that $x \gtrdot_\Theta y \implies |\Theta(x)| \geq |\Theta(y)|$.
- The case of **case** expression only deal with case analysis for a variable. The defini-

$$
\begin{array}{lcl}
\mathcal{I}(\Theta, x, \tau) & = & M, C \\
 & \textit{where} & \Theta' = \textit{rename}(\Theta) \\
 & & C_0 = \textit{left\_than\_right}(\Theta') \\
 & & C_1 = \textit{above\_than\_below}(\Theta', \Theta) \\
 & & C_2 = \textit{skip\_unused}(\Theta, \{x\}) \\
 & & C = C_0 \cup C_1 \cup C_2 \cup \{\tau = \Theta'(x)\} \\
 & & M = \textit{coerce}^{\Theta \to \Theta'}(x) \\
\mathcal{I}(\Theta, i, \mathbf{Int}) & = & M, C \\
 & \textit{where} & \Theta' = \textit{rename}(\Theta) \\
 & & C_0 = \textit{left\_than\_right}(\Theta') \\
 & & C_1 = \textit{above\_than\_below}(\Theta', \Theta) \\
 & & C_2 = \textit{skip\_unused}(\Theta, \emptyset) \\
 & & C = C_0 \cup C_1 \cup C_2 \\
 & & M = \textit{coerce}^{\Theta \to \Theta'}(x) \\
\mathcal{I}(\Theta, M_1 + M_2, \mathbf{Int}) & = & M_1'' + M_2'', C \\
 & \textit{where} & \Theta_1 = \Theta|_{\mathbf{FV}(M_1)} \\
 & & \Theta_2 = \Theta|_{\mathbf{FV}(M_2)} \\
 & & \Theta_1' = \textit{rename}(\Theta_1) \\
 & & \Theta_2' = \textit{rename}(\Theta_2) \\
 & & C_0 = \textit{left\_than\_right}(\Theta_1') \\
 & & C_1 = \textit{left\_than\_right}(\Theta_2') \\
 & & C_2 = \textit{above\_than\_below}(\Theta_1', \Theta_1) \\
 & & C_3 = \textit{above\_than\_below}(\Theta_2', \Theta_2) \\
 & & M_1', C_4 = \mathcal{I}(\Theta_1', M_1, \mathbf{Int}) \\
 & & M_2', C_5 = \mathcal{I}(\Theta_2', M_2, \mathbf{Int}) \\
 & & C_6 = \textit{merge\_constr}(\Theta, \Theta_1, \Theta_2) \\
 & & M_1'' = \textit{coerce}^{\Theta_1 \to \Theta_1'}(M_1') \\
 & & M_2'' = \textit{coerce}^{\Theta_2 \to \Theta_2'}(M_2') \\
 & & C = C_0 \cup C_1 \cup C_2 \cup C_3 \cup C_4 \cup C_5 \cup C_6
\end{array}
$$

Figure 5.9: Automatic insertion algorithm (1 of 4).

$$
\begin{array}{lcl}
\mathcal{I}(\Theta, \mathbf{fix}\ (f, x, M), \tau \to \tau') & = & M', C \\
 & \textit{where} & \Theta' = f : \tau \to \tau', \Theta, x : \tau \\
 & & \Theta'' = \textit{rename}(\Theta') \\
 & & C_0 = \{|\tau| \geq \omega | (x : \tau) \in \Theta\} \\
 & & C_1 = \textit{left\_than\_right}(\Theta'') \\
 & & C_2 = \textit{above\_than\_below}(\Theta'', \Theta') \\
 & & C_3 = \mathcal{I}(\Theta'', M, \tau') \\
 & & C = C_0 \cup C_1 \cup C_2 \cup C_3 \\
 & & M' = \mathbf{fix}\ (f, x, \textit{coerce}^{\Theta' \to \Theta''}(M)) \\
\mathcal{I}(\Theta, M_1\ M_2, \tau) & = & M, C \\
 & \textit{where} & \Theta_1 = \Theta|_{\mathbf{FV}(M_1)} \\
 & & \Theta_2 = \Theta|_{\mathbf{FV}(M_2)} \\
 & & \Theta'_1 = \textit{rename}(\Theta_1) \\
 & & \Theta'_2 = \textit{rename}(\Theta_2) \\
 & & C_0 = \textit{left\_than\_right}(\Theta'_1) \\
 & & C_1 = \textit{left\_than\_right}(\Theta'_2) \\
 & & C_2 = \textit{above\_than\_below}(\Theta'_1, \Theta_1) \\
 & & C_3 = \textit{above\_than\_below}(\Theta'_2, \Theta_2) \\
 & & \tau' = \textit{typeof}(M_2) \\
 & & M'_1, C_4 = \mathcal{I}(\Theta'_1, M_1, \tau' \to \tau) \\
 & & M'_2, C_5 = \mathcal{I}(\Theta'_2, M_2, \tau') \\
 & & C_6 = \textit{merge\_constr}(\Theta, \Theta_1, \Theta_2) \\
 & & M''_1 = \textit{coerce}^{\Theta_1 \to \Theta'_1}(M'_1) \\
 & & M''_2 = \textit{coerce}^{\Theta_2 \to \Theta'_2}(M'_2) \\
 & & C = C_0 \cup C_1 \cup C_2 \cup C_3 \cup C_4 \cup C_5 \cup C_6 \\
\mathcal{I}(\Theta, \mathbf{leaf}\ M, \mathbf{Tree}^d) & = & M', C \\
 & \textit{where} & \Theta' = \textit{rename}(\Theta) \\
 & & C_0 = \textit{left\_than\_right}(\Theta') \\
 & & C_1 = \textit{above\_than\_below}(\Theta', \Theta) \\
 & & M'', C_2 = \mathcal{I}(\Theta', M, \mathbf{Int}) \\
 & & M' = \mathbf{mleaf}\ \textit{coerce}^{\Theta \to \Theta'}(M'') \\
 & & C = C_0 \cup C_1 \cup C_2 \cup \{d \geq \omega\}
\end{array}
$$

Figure 5.10: Automatic insertion algorithm (2 of 4). *typeof*(M) returns the type of M inferred by the type reconstruction algorithm for $\lambda^{\to}$

$$
\begin{array}{rcl}
\mathcal{I}(\Theta, \mathbf{node}\ M_1\ M_2, \mathbf{Tree}^d) & = & M, C \\
& \textit{where} & \Theta_1 = \Theta|_{\mathbf{FV}(M_1)} \\
& & \Theta_2 = \Theta|_{\mathbf{FV}(M_2)} \\
& & \Theta_1' = \mathit{rename}(\Theta_1) \\
& & \Theta_2' = \mathit{rename}(\Theta_2) \\
& & C_0 = \mathit{left\_than\_right}(\Theta_1') \\
& & C_1 = \mathit{left\_than\_right}(\Theta_2') \\
& & C_2 = \mathit{above\_than\_below}(\Theta_1', \Theta_1) \\
& & C_3 = \mathit{above\_than\_below}(\Theta_2', \Theta_2) \\
& & M_1', C_4 = \mathcal{I}(\Theta_1', M_1, \mathbf{Tree}^\omega) \\
& & M_2', C_5 = \mathcal{I}(\Theta_2', M_2, \mathbf{Tree}^\omega) \\
& & C_6 = \mathit{merge\_constr}(\Theta, \Theta_1, \Theta_2) \\
& & M_1'' = \mathit{coerce}^{\Theta_1 \to \Theta_1'}(M_1') \\
& & M_2'' = \mathit{coerce}^{\Theta_2 \to \Theta_2'}(M_2') \\
& & M = \mathbf{mnode}\ M_1''\ M_2'' \\
& & C = C_0 \cup C_1 \cup C_2 \cup C_3 \cup C_4 \cup C_5 \cup C_6 \cup \{d \geq \omega\}
\end{array}
$$

Figure 5.11: Automatic insertion algorithm (3 of 4).

tion does not lose generality by that restriction because $\begin{array}{l}\mathbf{case}\ M\ \mathbf{of} \\ \quad \mathbf{leaf}\ x \Rightarrow M_1 \\ \quad \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2\end{array}$ is equivalent to $\mathbf{let}\ y = M\ \mathbf{in}\ \begin{array}{l}\mathbf{case}\ y\ \mathbf{of} \\ \quad \mathbf{leaf}\ x \Rightarrow M_1 \\ \quad \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2\end{array}$ for a fresh variable y. By putting the restriction, we make it easy to determine where to insert x_1 and x_2.

The following theorem states the correctness of $\mathcal{I}$:

Theorem 5.3 *If $\mathcal{I}(\Omega, M, \tau) = M', C$, and if a substitution θ to use variables is obtained by solving constraints C, then $\theta\Omega \vdash M \rightsquigarrow \theta M' : \theta\tau$ holds.*

$$\mathcal{I}(\Theta, \begin{array}{l}\textbf{case } y \textbf{ of}\\ \quad \textbf{leaf } x \Rightarrow M_1\\ \mid \textbf{node } x_1\ x_2 \Rightarrow M_2\end{array}, \tau) \quad = \quad M, C$$

$$\begin{array}{rl}
\textit{where} & \Theta_0 = \Theta|_{\{y\}}\\
& S = (\mathbf{FV}(M_1)\backslash\{x\}) \cup (\mathbf{FV}(M_2)\backslash\{x_1, x_2\})\\
& \Theta_1 = (x : \mathbf{Int}, \Theta|_S)\\
& \Theta_2 = (\Theta_{2L}; y : \mathbf{Tree}^d; \Theta_{2R}) = \Theta|_{S\cup\{y\}}\\
& \Theta'_0 = \textit{rename}(\Theta_0)\\
& \Theta'_1 = \textit{rename}(\Theta_1)\\
& \Theta'_2 = \textit{rename}([\Theta_{2L}; x_1 : \mathbf{Tree}^{d'}; x_2 : \mathbf{Tree}^{d'}; \Theta_{2R}])\\
& \hfill (d' \text{ is fresh})\\
& C_0 = \textit{left\_than\_right}(\Theta'_0)\\
& C_1 = \textit{left\_than\_right}(\Theta'_1)\\
& C_2 = \textit{left\_than\_right}(\Theta'_2)\\
& C_3 = \textit{above\_than\_below}(\Theta'_0, \Theta_0)\\
& C_4 = \textit{above\_than\_below}(\Theta'_1, \Theta_1)\\
& C_5 = \textit{above\_than\_below}(\Theta'_2, \Theta_2)\\
& M', C_6 = \mathcal{I}(\Theta'_0, y, \mathbf{Tree}^{d'})\\
& M'_1, C_7 = \mathcal{I}(\Theta'_1, M_1, \tau)\\
& M'_2, C_8 = \mathcal{I}(\Theta'_2, M_2, \tau)\\
& C_9 = \textit{merge\_constr}(\Theta, \Theta_0, \Theta|_S)\\
& M'' = \textit{coerce}^{\Theta_0\to\Theta'_0}(M')\\
& M''_1 = \textit{coerce}^{\Theta_1\to\Theta'_1}(M'_1)\\
& M''_2 = \textit{coerce}^{\Theta_2\to\Theta'_2}(M'_2)\\
& M = \begin{array}{l}\textbf{case}^{d'}\ M'' \textbf{ of}\\ \quad \textbf{leaf } x \Rightarrow M''_1\\ \mid \textbf{node } x_1\ x_2 \Rightarrow M''_2\end{array}\\
& C = C_0 \cup \cdots \cup C_9
\end{array}$$

Figure 5.12: Automatic insertion algorithm (4 of 4).

Chapter 6

Related Work

We dealt with only binary trees in this paper. Koichi Kodama [14] extended our framework to general XML documents. He also performed some experiments and confirmed the effectiveness of our method.

Nakano and Nishimura [17–19] proposed a method for translating tree-processing programs to stream-processing programs using attribute grammars. In their method, programmers write XML processing with XTiSP [18], a XML processing language with iteration constructs that utilize XPath [4]. Then, the program is translated to an attributed grammar After that, 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 [19] 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. A program that violates the order restriction is translated into stream-processor that stores a part of tokens in the input stream. However, the relation between their method and our source-to-source translation method in Chapter 5.2 is not clear. An advantage of our method is that programmers can use higher-order functions in our language, while they cannot in XTiSP. Another advantage of our method is that we can deal with source programs that involve side-effects (e.g. programs that print the value of every leaf) while that seems difficult in their method based on attribute grammars (since programmers seems not to be able to specify evaluation order of their programs).

The class of well-typed programs that does not use buffered trees 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

```
$N \to \textbf{node}\ N_1\ N_2$
    $N_1.inh = f_1\ N.inh; N_2.inh = f_2\ N.inh\ N_1.syn\ N_1.inh$
    $N.syn = f_3\ N.inh\ N_1.syn\ N_1.inh\ N_2.syn\ N_2.inh$
$N \to \textbf{leaf}\ i$
    $N.syn = f_4\ N.inh\ i$

$\textbf{fix}\ f.\lambda inh.\lambda t.\textbf{case}\ t\ \textbf{of}$
    $\textbf{leaf}\ x \Rightarrow f_4\ inh\ x$
    $\textbf{node}\ x_1\ x_2 \Rightarrow$  $\textbf{let}\ N_1.inh = f_1\ inh\ \textbf{in}$
                        $\textbf{let}\ N_1.syn = f\ N_1.inh\ x_1\ \textbf{in}$
                        $\textbf{let}\ N_2.inh = f_2\ N.inh\ N_1.syn\ N_1.inh\ \textbf{in}$
                        $\textbf{let}\ N_2.syn = f\ N_2.inh\ x_2\ \textbf{in}$
                            $f_3\ N.inh\ N_1.syn\ N_1.inh\ N_2.syn\ N_2.inh$
```

Figure 6.1: L-attributed grammar over binary trees and corresponding program.

```
$N \to \textbf{node}\ N_1\ N_2$
    $N_1.depth = N.depth + 1$
    $N_2.depth = N.depth + 1$
    $N.result = \textbf{node}\ N_1.result\ N_2.result$
$N \to \textbf{leaf}\ i$
    $\textbf{if}\ N.depth\ \textbf{mod}\ 2\ =\ 0\ \textbf{then}\ N.result\ =\ \textbf{leaf}\ (i+1)$
    $\textbf{else}\ N.result\ =\ \textbf{leaf}\ i$
```

Figure 6.2: L-attributed grammar that corresponds to *inc_alt* in Figure 3.3.

in Figure 6.1. If output trees are not used in attributes, the program is well-typed. Conversely, any program that is well-typed and does not use buffered trees 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 6.2.

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, 23] 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 languages (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, 25] 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 1.2, which violates the access order, can be expressed as a treeless program [25] 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 [25] 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 differences are:

- 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 5.2.

The type system we used in this paper is based on the ordered linear logic proposed by Polakow [22]. He proposed a logic programming language Olli, logical framework OLF and ordered lambda calculus 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 (!). However, we cannot use ordered linear logic directly since it would make our type system unsound. Petersen et al. [20] 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.

Chapter 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.

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

We presented an algorithm $\mathcal{I}$ that automatically inserts buffering primitives. However, $\mathcal{I}$ has the following limitations:

- $\mathcal{I}$ stores output trees on memory.
- $\mathcal{I}$ cannot skip input trees. When $\mathcal{I}$ needs to skip an input tree, it stores the tree on memory.

We are currently studying how to solve those problems and improve the algorithm.

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 1.1, so that our translation framework can be used for eliminating intermediate data structures and processing serialized data directly.

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*, Vol. 27, No. 8, pp. 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*, June 2003.

[11] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, Vol. 110, No. 2, pp. 327–365, 1994.

[12] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology (TOIT)*, Vol. 3, No. 2, pp. 117–148, 2003.

[13] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 331–342, 2002.

[14] Koichi Kodama. 順序付き線形型を用いたXMLストリーム処理プログラムの生成. Master's thesis, Tokyo Institute of Technology, March 2005.

[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*, pp. 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. An implementation scheme for XML transformation languages through derivation of stream processors. In Wei Ngan Chin, editor, *In Proceedings of the 2nd Asian Symposium on Programming Languages and Systems (APLAS2004)*, Vol. 3302 of *Lecture Notes in Computer Science*, pp. 74–90. Springer-Verlag, November 2004.

[19] 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*, Vol. 44. Elsevier Science Publishers, 2001.

[20] 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.

[21] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[22] Jeff Polakow. *Ordered linear logic and applications.* PhD thesis, Carnegie Mellon University, June 2001. Available as Technical Report CMU-CS-01-152.

[23] Mark Scardina and Mary Fernandez. *XPath Requirements Version 2.0.* World Wide Web Consortium, August 2003. `http://www.w3.org/TR/xpath20req/`.

[24] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Proceedings of Functional Programming Languages and Computer Architecture*, pp. 1–11, San Diego, California, 1995.

[25] 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)*, pp. 344–358. Berlin: Springer-Verlag, 1988.

Appendix A

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

Figure A.1 shows a program that prints out integer elements to standard error output in right-to-left, depth-first manner.

Figure A.2 shows a program that takes two input trees and returns whether they are identical. Note that the structure-preserving translation works.

```
Source program
let f' = fix f'.λt.mcase t of mleaf x ⇒ print_err x | mnode x_1 x_2 ⇒ (f' x_2); (f' x_1) in
        fix f.λt.case t of
                leaf x  ⇒  print_err x
        | node x_1 x_2  ⇒  let mt = strm_to_mem x_1 in (f x_2); (f' mt)

Target program
let f' = fix f'.λt.mcase t of mleaf x ⇒ print_err x | mnode x_1 x_2 ⇒ (f' x_2); (f' x_1) 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)
```

Figure A.1: A program that prints out integer elements to standard error output in right-to-left, depth-first manner.

```
Source program
fix eq.λ(t_1, t_2).
        case t_1 of
                leaf x ⇒ (case t_2 of leaf y ⇒ x = y | node x_1 x_2 ⇒ false)
        | node x_1 x_2 ⇒
                (case t_2 of leaf y ⇒ false | node y_1 y_2 ⇒ eq (x_1, y_1) && eq (x_2, y_2))

Target program
fix eq.λ(t_1, t_2).
        case read(t_1) of
            leaf ⇒ let x = read(t_1)in
                        (case read(t_2) of leaf ⇒ let y = read(t_2) in x = y | node ⇒ false)
        | node ⇒ (case read(t_2) of leaf ⇒ let y = read(t_2) in false
                                    |  node ⇒ eq (t_1, t_2) && eq (t_1, t_2))
```

Figure A.2: A program that takes two input trees and returns whether they are identical

Appendix B

The Proof of Theorem 4.2

In this chapter, we prove Theorem 4.2. We prepare the following lemma to prove the theorem. The proof of that lemma is due to Koichi Kodama. The proof of Theorem 4.2 is done by the author.

Lemma B.1 (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*:*

  $M = z$

  $\Delta = z : \mathbf{Tree^1}$

  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*:*

  $M = \lambda z.M_2$

  $\tau = \mathbf{Tree^1} \rightarrow \tau_2$

  $\Gamma, x : \tau' \mid z : \mathbf{Tree^1} \vdash M_2 : \tau_2$

  $\Delta = \emptyset$

  $\Gamma \mid z : \mathbf{Tree^1} \vdash [N/x]M_2 : \tau_2$ *follows from the induction hypothesis. Thus,* $\Gamma \mid \emptyset \vdash \lambda z.[N/x]M_2 : \mathbf{Tree^1} \rightarrow \tau_2$ *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*:*

  $M = \lambda z.M_1$

  $\tau = \tau_1 \rightarrow \tau_2$

  $\Gamma, x : \tau', z : \tau_1 \mid \emptyset \vdash M_1 : \tau_2$

  $\Delta = \emptyset$

  $\Gamma, z : \tau_1 \mid \emptyset \vdash [N/x]M_1 : \tau_2$ *follows from the induction hypothesis. Thus,* $\Gamma \mid \emptyset \vdash \lambda z.[N/x]M_2 : \tau_1 \rightarrow \tau_2$ *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*:*

  $M = M_1 M_2$

  $\Gamma, x : \tau' \mid \Delta_1 \vdash M_1 : \tau_2 \rightarrow \tau$

  $\Gamma, x : \tau' \mid \Delta_2 \vdash M_2 : \tau_2$

  $\Delta = \Delta_1, \Delta_2$

  $\Gamma \mid \Delta_1 \vdash [N/x]M_1 : \tau_2 \rightarrow \tau$ *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*:*

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

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

- *Case* T-NODE*:*

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

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

- *Case* T-CASE*:*

  $M = \textbf{case } M_0 \textbf{ of leaf } z \Rightarrow M_1 \mid \textbf{node } z_1\ z_2 \Rightarrow M_2$
  $\Gamma, x : \tau' \mid \Delta_1 \vdash M_0 : \textbf{Tree}^{\mathbf{1}}$
  $\Gamma, x : \tau', z : \textbf{Int} \mid \Delta_1 \vdash M_1 : \tau$
  $\Gamma, x : \tau' \mid z_1 : \textbf{Tree}^{\mathbf{1}}, z_2 : \textbf{Tree}^{\mathbf{1}}, \Delta_2 \vdash M_2 : \tau$
  $\Delta = \Delta_1, \Delta_2$

  $\Gamma \mid \Delta_1 \vdash [N/x]M_0 : \textbf{Tree}^{\mathbf{1}}$ *and* $\Gamma, z : \textbf{Int} \mid \Delta_2 \vdash [N/x]M_1 : \tau$ *and* $\Gamma \mid z_1 : \textbf{Tree}^{\mathbf{1}}, z_2 : \textbf{Tree}^{\mathbf{1}}, \Delta_2 \vdash M_2 : \tau$ *follow from the induction hypothesis. Because* $\textbf{case } [N/x]M_0 \textbf{ of leaf } z \Rightarrow [N/x]M_1 \mid \textbf{node } z_1\ z_2 \Rightarrow [N/x]M_2 = [N/x]M$*, we have* $\Gamma \mid \Delta \vdash [N/x]M : \tau$ *from* T-CASE.

- *Case* T-FIX*:*

  $M = \textbf{fix } f.M_1,$
  $\tau = \tau_1 \to \tau_2$
  $\Gamma, x : \tau', f : \tau_1 \to \tau_2 \mid \Delta \vdash M_1 : \tau_1 \to \tau_2$

  $\Gamma, f : \tau_1 \to \tau_2 \mid \Delta \vdash [N/x]M_1 : \tau_1 \to \tau_2$ *follows from the induction hypothesis. Because* $\textbf{fix } f.[N/x]M_1 = [N/x](\textbf{fix } f.M_1)$*, we have* $\Gamma \mid \Delta \vdash [N/x]M : \tau$ *from* T-FIX.

□

Proof of Theorem 4.2. 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 B.1 as required.

    Next, suppose $U = y$ for some y. Then, $M' = [y/x]N$ and $\delta' = \delta$ follow from Es2-App. $\langle\langle \delta \rangle\rangle = y : \mathbf{Tree^1}$ and $\Gamma \mid x : \mathbf{Tree^1} \vdash N : \tau$ follow from the assumption $\Gamma \mid \langle\langle \delta \rangle\rangle \vdash M : \tau$ and T-App and T-Abs1. Thus, as easily seen, $\Gamma \mid y : \mathbf{Tree^1} \vdash [y/x]N : \tau$. Thus, $\Gamma \mid \langle\langle \delta' \rangle\rangle \vdash M' : \tau$ follows as required.
  - Case $I = (\mathbf{case}\ y\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2)$ 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 B.1. 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 $I = (\mathbf{case}\ y\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2)$ with $\delta = (y \mapsto (\mathbf{node}\ V_1\ V_2), \delta'')$.
    $\Gamma \mid \langle\langle \delta \rangle\rangle \vdash M : \tau$ must have been derived by using T-Case. Thus, we have $\Gamma \mid x_1 : \mathbf{Tree^1}, x_2 : \mathbf{Tree^1}, \langle\langle \delta'' \rangle\rangle \vdash M_2 : \tau$. 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 B.1. 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 $\Gamma \mid \langle\langle \delta_1 \rangle\rangle \vdash E_s'[I] : \mathbf{Tree^1}$ and $\Gamma, x : \mathbf{Int} \mid \langle\langle \delta_2 \rangle\rangle \vdash M_1 : \tau$ and $\Gamma \mid x_1 : \mathbf{Tree^1}, x_2 : \mathbf{Tree^1}, \langle\langle \delta_2 \rangle\rangle \vdash M_2 : \tau$ 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 $\Gamma \mid \langle\langle \delta_1' \rangle\rangle \vdash M'' : \mathbf{Tree^1}$ 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.

$\square$

Appendix C

The Proof of Lemma 4.1

For the proof of Lemma 4.1, we prepare the following lemma.

Lemma C.1 *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.*

$\square$

Proof of Lemma 4.1. 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 $E_s = \textbf{leaf}\ E'_s$.
  In this case, $M = \textbf{leaf}\ E'_s[I]$. $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash E'_s[I] : \textbf{Tree}^+$ 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 $\textbf{write}(e')$ and S be $\textbf{leaf}; S'$. 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 $\mathcal{A}_\gamma(M) = \textbf{write}(\textbf{leaf}); \textbf{write}(\mathcal{A}_\gamma(E'_s[I]))$.

- Case $E_s = \textbf{node}\ E'_s\ M'$.
  In this case, $M = \textbf{node}\ E'_s[I]\ M'$. $\emptyset \mid \langle\langle\delta_1\rangle\rangle \vdash E'_s[I] : \textbf{Tree}^+$ and $\emptyset \mid \langle\langle\delta_2\rangle\rangle \vdash M' : \textbf{Tree}^+$ 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 $\textbf{node}; S'$. Then, $M \sim_\gamma (e, S)$ follows from C-NODE1. Because $\mathcal{A}_\gamma(M) = \textbf{write}(\textbf{node}); \mathcal{A}_\gamma(E'_s[I]); \mathcal{A}_\gamma(M')$, we have $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \longrightarrow^* (e, \langle\delta\rangle, S)$.

- Case $E_s = \textbf{node}\ V\ E'_s$.
  In this case, $M = \textbf{node}\ V\ E'_s[I]$. $\emptyset \mid \langle\langle\delta\rangle\rangle \vdash E'_s[I] : \textbf{Tree}^+$ 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 $\textbf{node}; [\![V]\!]; S'$. Then, $M \sim_\gamma (e, S)$ follows from C-NODE2. Because $\mathcal{A}_\gamma(M) = \textbf{write}(\textbf{node});$
  $\mathcal{A}_\gamma(V); \mathcal{A}_\gamma(E'_s[I])$, we have $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \longrightarrow^* (e, \langle\delta\rangle, S)$.

- Case $E_s = (\textbf{case}\ E'_s\ \textbf{of leaf}\ x \Rightarrow M_1 \mid \textbf{node}\ x_1\ x_2 \Rightarrow M_2)$.
  In this case, $(M = (\textbf{case}\ E'_s[I]\ \textbf{of leaf}\ x \Rightarrow M_1 \mid \textbf{node}\ x_1\ x_2 \Rightarrow M_2))$. $\emptyset \mid \langle\langle\delta_1\rangle\rangle \vdash E'_s[I] : \textbf{Tree}^{\mathbf{1}}$ and $x : \textbf{Int} \mid \langle\langle\delta_2\rangle\rangle \vdash M_1 : \tau$ and $\emptyset \mid x_1 : \textbf{Tree}^{\mathbf{1}}, x_2 : \textbf{Tree}^{\mathbf{1}}, \langle\langle\delta_2\rangle\rangle \vdash M_1 : \tau$ 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 $\textbf{case}\ e'; \textbf{read}()\ \textbf{of leaf} \Rightarrow \textbf{let}\ x = \textbf{read}()\ \textbf{in}\ \mathcal{A}_\gamma(M_1) \mid \textbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2)$ and S be S'. Then, $M \sim_\gamma (e, S)$ follows from C-CASE. Because $\mathcal{A}_\gamma(M) = \textbf{case}\ \mathcal{A}_\gamma(E'_s[I]); \textbf{read}()\ \textbf{of leaf} \Rightarrow \textbf{let}\ x = \textbf{read}()\ \textbf{in}\ \mathcal{A}_\gamma(M_1) \mid \textbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2)$. $(\mathcal{A}_\gamma(M), \langle\delta\rangle, \emptyset) \longrightarrow^* (e, \langle\delta\rangle, S)$ holds.

□

Appendix D

The Proof of Lemma 4.2

We prove Lemma 4.2 in this chapter.

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, $M' = plus(i_1)i_2 (= i)$ 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 C.1, we have:*

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

By Lemma 4.1, 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 $I = \textbf{case } y \textbf{ of leaf } x \Rightarrow M_1 \mid \textbf{node } x_1\ x_2 \Rightarrow M_2$ with $\delta = (y \mapsto \textbf{leaf } i, \delta_1)$.*
$(M, \delta) \longrightarrow (M', \delta')$ 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 $e = \mathcal{A}_\gamma(I) = \textbf{case } (); \textbf{read}() \textbf{ of leaf} \Rightarrow \textbf{let } x = \textbf{read}() \textbf{ in } \mathcal{A}_\gamma(M_1) \mid \textbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2)$ and $S = \emptyset$. $(e, \textbf{leaf}; i; \langle\delta_1\rangle, \emptyset) \longrightarrow^+ (\mathcal{A}_\gamma(M'), \langle\delta_1\rangle, \emptyset)$ follows from Lemma C.1. By Lemma 4.1, 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 $I = \textbf{case } y \textbf{ of leaf } x \Rightarrow M_1 \mid \textbf{node } x_1\ x_2 \Rightarrow M_2$ with $\delta = (y \mapsto \textbf{node } V_1\ V_2, \delta_1)$.*
$(M, \delta) \longrightarrow (M', \delta')$ 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 $e = \mathcal{A}_\gamma(I) = \textbf{case } (); \textbf{read}() \textbf{ of leaf} \Rightarrow \textbf{let } x = \textbf{read}() \textbf{ in } \mathcal{A}_\gamma(M_1) \mid \textbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2)$ 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, $(e, \textbf{node}; [\![V_1]\!]; [\![V_2]\!]; \langle\delta_1\rangle, \emptyset) \longrightarrow^+ (\mathcal{A}_{\gamma\cup\{x_1,x_2\}}(M_2), [\![V_1]\!]; [\![V_2]\!]; \langle\delta_1\rangle, \emptyset)$. By Lemma4.1, 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 $I = \textbf{fix } f.M_1$.*
$(M, \delta) \longrightarrow (M', \delta')$ must have been derived by using Es2-Fix*. So, it must be the case that $M' = [\textbf{fix } f.M_1/f]M_1$ and $\delta' = \delta$. $M \sim_\gamma (e, S)$ implies $e = \mathcal{A}_\gamma(I) = \textbf{fix } f.\mathcal{A}_\gamma(M_1)$ and $S = \emptyset$. By LemmaC.1, we have:*

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

By Lemma4.1, 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 4.1, 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 4.1, 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 4.1, 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 4.1, 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 $E_s =$ **leaf** E_1.*
  *There exists M_1' that satisfies $M' =$ **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 $E_s =$ **node** E_1 M_2.*
  *There exists M_1' that satisfies $M' =$ **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 4.1, 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 $E_s = (\mathbf{case}\ E_1\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2)$.*
  There exists M_1' that satisfies $M' = (\mathbf{case}\ M_1'\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2)$ 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 $e = (\mathbf{case}\ e_1; \mathbf{read}()\ \mathbf{of}\ \mathbf{leaf} \Rightarrow \mathbf{let}\ x = \mathbf{read}()\ \mathbf{in}\ M_1 \mid \mathbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2))$ 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 $((\mathbf{case}\ e_1; \mathbf{read}()\ \mathbf{of}\ \mathbf{leaf} \Rightarrow \mathbf{let}\ x = \mathbf{read}()\ \mathbf{in}\ M_1 \mid \mathbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2)), \langle\delta\rangle, S) \longrightarrow^+$*

$((\textbf{case } e_1'; \textbf{read}() \textbf{ of leaf} \Rightarrow \textbf{let } x = \textbf{read}() \textbf{ in } M_1 \mid \textbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2)), \langle \delta' \rangle, S'')$ *and* $M_1' \sim_\gamma (e_1', S'')$ *for some* e_1' *and* S''*.*

First, suppose that M_1' *is reducible. Let* e' *be* $(\textbf{case } e_1'; \textbf{read}() \textbf{ of leaf} \Rightarrow \textbf{let } x = \textbf{read}() \textbf{ in } M_1 \mid \textbf{node} \Rightarrow [()/x_1, ()/x_2]\mathcal{A}_\gamma(M_2))$ *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 4.1, 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.*

$\square$

Appendix E

Proof of Theorem 4.3

This chapter proves Theorem 4.3. The proof of this chapter is due to Naoki Kobayashi.

Proof *First of all, note that $\emptyset \mid x : \mathbf{Tree^1} \vdash M\ x : \tau$ follows an assumption $\emptyset \mid \emptyset \vdash M : \mathbf{Tree^1} \rightarrow \tau$ and $\emptyset \mid x : \mathbf{Tree^1} \vdash x : \mathbf{Tree^1}$.*

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

Assume $((M\ x), x \mapsto V) \rightarrow^* (V', \emptyset)$. Because $\emptyset \mid x : \mathbf{Tree^1} \vdash (M\ x)\ : \tau$ holds, there exist e, S_i and S_o such that $((M\ x), x \mapsto V) \sim (e, S_i, S_o)$ and $(\mathcal{A}(M)(), S_i, \emptyset) \rightarrow^* (e, S_i, S_o)$ from Lemma 4.1[1]*. From the definition of $\sim$, $S_i = [\![V]\!]$. Because of Theorem 4.2 and Lemma 4.2, there exists a sequence of reduction $(e, [\![V]\!], S_o) \rightarrow^* (e', \emptyset, S'_o)$ that satisfies $(V', \emptyset) \sim (e', \emptyset, S'_o)$. From the definition of $\sim$, $e' = ()$ and $S'_o = [\![V']\!]$. Thus, $(\mathcal{A}(M)(), [\![V]\!], \emptyset) \rightarrow^* ((), \emptyset, [\![V']\!])$ holds.*

Next, assume $(\mathcal{A}(M)(), [\![V]\!], \emptyset) \rightarrow^* ((), \emptyset, [\![V']\!])$. 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 $(\mathcal{A}(M)(), [\![V]\!], \emptyset) \rightarrow^* (e, [\![V]\!], S_o)$. Because applicable reduction rule can be uniquely determined at each step of reduction, $(e, [\![V]\!], S_o) \rightarrow^* ((), \emptyset, [\![V']\!])$ 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 $x \mapsto V$, $((M\ x), x \mapsto V) \longrightarrow^* (V', \emptyset)$ holds because $\emptyset \mid x : \mathbf{Tree^1} \vdash (M\ x) : \mathbf{Tree^+}$ follows $((M\ x), x \mapsto V) \rightarrow^* (V, \emptyset)$ and Theorem 4.2.

- *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.*

[1] Because $\emptyset \mid \emptyset \vdash M : \mathbf{Tree^1} \rightarrow \tau$ holds, $\mathbf{FV}(M) = \emptyset$. Thus, $\mathcal{A}_{\mathbf{FV}(M) \cup \{x\}}(M\ x) = \mathcal{A}(M)()$.

- *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 4.2, 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.*

□