# 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

児玉 紘一
東京工業大学
大学院情報理工学研究科
kodama@kb.cs.titech.ac.jp

小林 直樹
東京工業大学
大学院情報理工学研究科
kobayasi@kb.cs.titech.ac.jp

末永 幸平
東京大学
大学院情報理工学系研究科
kohei@yl.is.s.u-tokyo.ac.jp

## 概 要

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文書の処理の際に効率が悪い．我々の研究の最終的な目標は，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．本論文ではその第一歩として、二分木を扱う関数型プログラミング言語と順序付き線形型を用いた型システムを定義し、型付けされる任意のプログラムはそれと等価なストリーム処理プログラムへ変換できることを示す．

## 1 序論

XML 文書 [5] や Lisp の S 式のようなデータ表現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．例えば XML 文書処理では，前者の手法として DOM(DocumentObjectModel) API やプログラミング言語 XDuce[10]，後者の手法として，SAX (Simple API for XML) などがある．前者の手法は，データの論理的構造に基づいているためにプログラム記述が容易である反面，文章全体の構文解析結果をメモリ内に木構造として展開してから処理するため，処理時間や使用メモリの面で問題がある．一方，後者の手法では，文書をストリームとして順次処理するため処理時間や使用メモリの面では有効だが，物理的構造に基づいて記述をする必要があるためプログラマへの負担が大きい．XML 文書処理の必要性が増している今日，このような処理の記述性と効率の両立の難しさは，重要な問題となってきている．

我々は，上記の問題点を解決するため，プログラマには XML などの文書の処理を木構造処理のプログラムとして記述させ，それをストリーム処理プログラムへと自動変換する手法の確立を目指している．本論文ではその第一歩として，二分木を基本データ型に持つ関数型言語のプログラムをストリーム処理言語のプログラムへと変換する方法を示す．

図 1 は木構造処理とストリーム処理の流れである．これは二分木の葉の各要素に 1 を加えた木を結果として返す処理の様子である．木構造処理では二分木を表す文書を受け取るとそれをメモリ上に木として構築し，この木に対して処理を行い，結果として新しい木を得る．このため処理を行う前に木がメモリ上に展開されなければならず，文書が巨大である場合は問題となる．他方，ストリーム処理では文書を先頭から読み込み，順次出力する．このため木がメモリ上に展開されるのを待つことなく処理を行うことができるので，特に巨大な文書を扱う場合には，メモリ効率の面でも計算時間の面でも木構造処理を行うプログラムよりも優れている．

図 2 の上半分は図 1 で示した処理を実現するプログラムの例である．木構造処理のプログラムは変数 $t$ に入力木を受け取り，**case** によってパターンマッチを行う．$t$ とマッチしたものが葉の場合は要素を変数 $x$ に束縛し，それに 1 を足したものを葉の要素として返す．木の場合は再帰関数 $f$ に左の木と右の木をそれぞれ適用し，それらを木の左の要素，右の要素として返す．

図 2 の下半分は以上の処理をストリーム処理を行うターゲット言語に変換したものである．ストリーム処理のプログラムでは変数 $t$ に () を受け取ると処理が始まる．入力は命令 **read** を用いて入力ストリームから一つ

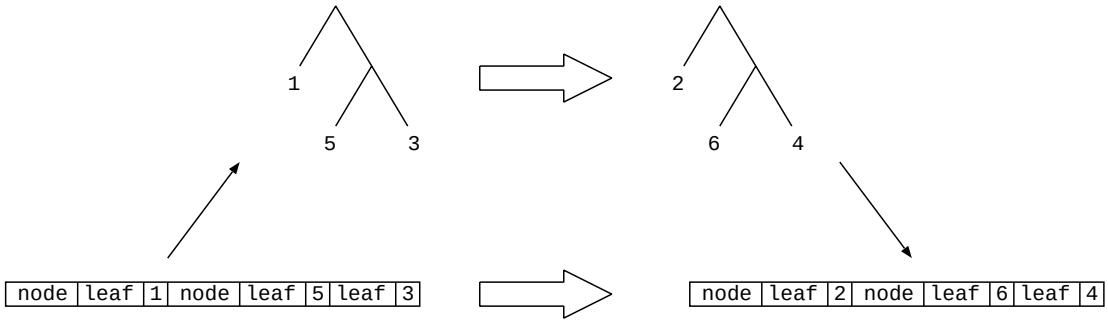


図 1: 木構造処理とストリーム処理

木構造処理：

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

ストリーム処理：

$\mathbf{fix}\ f.\lambda t.(\mathbf{case}\ \mathbf{read}()\ \mathbf{of}\ \mathbf{leaf} \Rightarrow \mathbf{write}(\mathbf{leaf}); \mathbf{write}(\mathbf{read}() + 1) \mid \mathbf{node} \Rightarrow \mathbf{write}(\mathbf{node}); f\ (); f\ ())$

図 2: 二分木の各葉に 1 を足した木を作るプログラム

読み込み，これに対してパターンマッチを行う．マッチしたものが **leaf** タグの場合は，**leaf** タグともう一段階読み込んだ結果に 1 を足したものを命令 **write** によって出力ストリームに順に書き出す．**node** タグの場合は **node** タグを命令 **write** によって出力ストリームに書き出してから，再帰関数 $f$ () を二回呼ぶ．

自動変換にあたっての問題点は，木構造を扱う全てのプログラムがストリーム処理プログラムに変換できるわけではないことである．これはストリーム処理が実行に従って順次文書を読み進めるため，文書を遡っての実行や文書の先読みを必要とする処理が不可能であることに起因するものである．例えば図 3 のようなプログラムはストリーム処理には変換できない．これは **node** $(f\ x_2)\ (f\ x_1)$ の部分で左の木よりも先に右の木を読み込む必要があるためである．またストリーム処理では結果を評価が終わったものから順次出力していくため，木構造処理のプログラムがメモリ中に出力木を保持して一斉に書き出すという操作をするものである場合は変換ができない．

そこで，ストリーム処理プログラムに変換できないプログラムを排除するため，本研究では順序付き線形型 [14, 15] に基づく型システムを構築する．順序付き線形型は，線形型 [2, 17] の拡張であり，その型の値には，丁度一回だけ用いられるだけでなく，特定の順序で用いられなければならないという制約が付加されている．入力木の各部分木にそのような順序付き線形型を与えることにより，入力木の各ノードが左から深さ優先の順に丁度一回だけアクセスされることを保証することができる．これにより，我々の型システムで型付けできるプログラムは，それと等価なストリーム処理プログラムへと変換できることを保証できる．

本論文の以下の構成は次の通りである．2 節で木構造処理プログラムを記述するためのソース言語および変換後のストリーム処理プログラムを記述するためのターゲット言語の定義を行う．3 節においてソース言語の型システムを定義する．4 節ではストリーム処理プログラムへの変換のアルゴリズムとその正しさについて論じる．5 節では効率，表現力，XML 文書処理への適用に対する議論を行う．6 節で関連研究に触れ，7 節で結論を述べる．

## 2 言語仕様

本節ではソース言語とターゲット言語の定義を行う．前者はプログラマが実際に記述する，整数を葉の要素に持つ二分木の木構造処理を扱う型付き言語であり，後者は二分木をストリームとして処理を行う言語である．これらはパターンマッチを行って処理を分岐するが，ソース言語ではこのパターンマッチを二分木の構造に対して行うのに対し，ターゲット言語ではストリームの先頭を読み込んだものに対して行う．また，ソース言語ではプログラムは一つの木を入力として受けとり結果を返す関数として記述されるのに対し，ターゲット言語では入力

木構造処理：
**fix** $f.\lambda t.(\mathbf{case}\ t\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow \mathbf{leaf}\ x \mid \mathbf{node}\ x_1\ x_2 \Rightarrow \mathbf{node}\ (f\ x_2)\ (f\ x_1))$

図 3: 左右の木を入れ替えるプログラム

ストリーム，出力ストリームが一つずつ存在し，入力は入力ストリームを命令 **read** によって読み込むことで得られ，結果が木であった場合においては返り値は () となり，出力ストリームに命令 **write** によってタグや要素を書き出していくことで対応する木が出力される．

## 2.1 ソース言語

ソース言語は木構造を扱う型付きの関数型言語である．型規則についての詳細は 3 節にて述べる．

**定義 2.1 (ソース言語の構文)** ソース言語の項 $M$ は以下のように定義される．

$$
\begin{array}{rcl}
M & ::= & W \mid x \mid M_1\ M_2 \mid M_1 + M_2 \\
& & \mid \mathbf{leaf}\ M \mid \mathbf{node}\ M_1\ M_2 \\
& & \mid (\mathbf{case}\ M\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2) \\
& & \mid \mathbf{fix}\ f.M \\
V & ::= & \mathbf{leaf}\ i \mid \mathbf{node}\ V_1\ V_2 \\
W & ::= & i \mid \lambda x.M \mid V
\end{array}
$$

$W$ は値を表すメタ変数である．また $x$ は変数，$i$ は整数を示すメタ変数である．値は整数 $i$，ラムダ抽象 $\lambda x.M$，および二分木の値を表すメタ変数 $V$ であり，二分木は要素として整数を 1 つ持つ葉 **leaf** $i$，または葉か部分木を 2 つ持つ木 **node** $V_1\ V_2$ である．($\mathbf{case}\ M\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2$) は，まず $M$ で木構造に対してパターンマッチを行い，葉であれば要素を $x$ に束縛して $M_1$ を，部分木の場合は左右の部分木をそれぞれ $x_1$，$x_2$ に束縛して $M_2$ を評価することを意味する．**fix** $f.M$ は $f = M$ を満たす再帰関数である．

また，$M_1; M_2$ を $(\lambda x.M_2)\ M_1$ において $M_2$ に $x$ が含まれない場合における略記とする．これは $M_1$，$M_2$ の逐次実行を意味するものである．

**定義 2.2 (ソース言語の評価文脈)** ソース言語の評価文脈 *(Evaluation Contexts)*$E_s$ は以下のように定義される．

$$
\begin{array}{rcl}
E_s & ::= & [\,] \mid E_s\ M \mid (\lambda x.M)\ E_s \mid E_s + M \mid i + E_s \\
& & \mid \mathbf{leaf}\ E_s \mid \mathbf{node}\ E_s\ M \mid \mathbf{node}\ V\ E_s \\
& & \mid (\mathbf{case}\ E_s\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow M_1 \mid \mathbf{node}\ x_1\ x_2 \Rightarrow M_2)
\end{array}
$$

この評価文脈を用いて，簡約規則を定義する．

**定義 2.3 (ソース言語の簡約規則)** ソース言語の簡約関係 $M \to M'$ は，図 *4* の規則を満たす最小の関係である．

## 2.2 ターゲット言語

ターゲット言語はストリーム処理を扱う関数型言語である．ソース言語が木構造を扱っていたのに対し，ターゲット言語では二分木をタグと要素の列として扱う．

まずストリームの定義を行う．

**定義 2.4 (ストリーム)** ストリームは以下のように定義される．

$$
S \quad ::= \quad \mathbf{leaf}; S \mid \mathbf{node}; S \mid i; S \mid \emptyset
$$

$\emptyset$ は空列である．ストリームの中には **leaf** タグ，**node** タグ，および整数値が連続して入っている．先頭は左である．例えば木構造で **node** (**leaf** 1) (**node**(**leaf** 2)(**leaf** 3)) と表現される木をストリームで表現すると，$\mathbf{node}; \mathbf{leaf}; 1; \mathbf{node}; \mathbf{leaf}; 2; \mathbf{leaf}; 3$ のようになる．ターゲットプログラムでは入力を入力ストリームの読み込みによって行い，プログラムの実行結果が木の場合はそれを出力ストリームに書き出す．また，ターゲットプログラムでは入力ストリーム，出力ストリームはそれぞれ一つずつ用意されているものとする．

**定義 2.5 (ターゲット言語の構文)** ターゲット言語の項 $e$ は以下のように定義される．

$$
\begin{array}{rcl}
e & ::= & v \mid x \mid e_1\ e_2 \mid e_1 + e_2 \\
& & \mid \mathbf{read}\ e \mid \mathbf{write}\ e \\
& & \mid \mathbf{case}\ e\ \mathbf{of}\ \mathbf{leaf} \Rightarrow e_1 \mid \mathbf{node} \Rightarrow e_2 \\
& & \mid \mathbf{fix}\ f.e \\
v & ::= & i \mid \mathbf{leaf} \mid \mathbf{node} \mid \lambda x.e \mid ()
\end{array}
$$

$v$ は値を表すメタ変数である．また $x$ は変数，$i$ は整数を表すメタ変数である．$+$ は整数の和を表す演算子である．**read** は () を適用すると入力ストリームから 1 つ読み込み，それを結果として返す命令である．**write** は値を 1 つとり，それを出力ストリームに書き出し，結果として () を返す命令である．$\mathbf{case}\ e\ \mathbf{of}\ \mathbf{leaf} \Rightarrow e_1 \mid \mathbf{node} \Rightarrow e_2$

$$E_s[i_1 + i_2] \longrightarrow E_s[plus(i_1, i_2)] \quad \text{(Es-Plus)}$$
$$E_s[(\lambda x.M)W] \longrightarrow E_s[[W/x]M] \quad \text{(Es-App)}$$
$$E_s[\textbf{case leaf } i \textbf{ of leaf } x \Rightarrow M_1 \mid \textbf{node } x_1\ x_2 \Rightarrow M_2] \longrightarrow E_s[[i/x]M_1] \quad \text{(Es-Case1)}$$
$$E_s[\textbf{case node } V_1\ V_2 \textbf{ of leaf } x \Rightarrow M_1 \mid \textbf{node } x_1\ x_2 \Rightarrow M_2] \longrightarrow E_s[[V_1/x_1, V_2/x_2]M_2] \quad \text{(Es-Case2)}$$
$$E_s[\textbf{fix } f.M] \longrightarrow E_s[[\textbf{fix } f.M/f]M] \quad \text{(Es-Fix)}$$

図 4: ソース言語の簡約規則．ただし $plus(i_1, i_2)$ は $i_1$ と $i_2$ の整数としての和．

は $e$ に対してパターンマッチを行い，**leaf** タグであれば $e_1$ を，**node** タグであれば $e_2$ を評価することを意味する．**fix** $f.e$ は $f = e$ を満たす再帰関数である．

また，**let** $x = e_1$ **in** $e_2$ は $(\lambda x.e_2)\ e_1$ の略記であり，変数 $x$ に $e_1$ の結果を束縛した上で $e_2$ を行うことを意味する．特に $(\lambda x.e_2)\ e_1$ において $e_2$ に $x$ が含まれない場合は $e_1; e_2$ を略記として用い，これは $e_1$，$e_2$ の逐次実行を意味する．

**定義 2.6 (ターゲット言語の評価文脈)** ターゲット言語の評価文脈 *(Evaluation Contexts)*$E_t$ は以下のように定義される．

$$\begin{array}{rcl} E_t & ::= & [\,] \mid E_t\ e \mid (\lambda x.e)\ E_t \mid E_t + e \mid i + E_t \\ & & \mid \textbf{case } E_t \textbf{ of leaf} \Rightarrow e_1 \mid \textbf{node} \Rightarrow e_2 \end{array}$$

この評価文脈と入出力ストリームを用いて，簡約規則を定義する．

**定義 2.7 (ターゲット言語の簡約規則)** ターゲット言語の簡約関係 $(e, S_i, S_o) \rightarrow (e', S_i', S_o')$ は図 *5* の規則を満たす最小の関係である．

ただし $S_i$ と $S_i'$ は入力ストリームの状態を，$S_o$ と $S_o'$ は出力ストリームの状態を表す．

図 6 は二分木の葉の要素の総和を求めるプログラムである．ソース言語によるプログラムは変数 $t$ に入力木を受け取り，**case** 文によってパターンマッチを行う．$t$ とマッチしたものが葉の場合は要素を変数 $x$ に束縛し，それを結果として返す．木の場合は再帰関数 $f$ に左の木と右の木をそれぞれ適用して，この足し算を行う．ターゲット言語によるプログラムでは変数 $t$ に () を受け取ると処理が始まる．入力は命令 **read** を用いて入力ストリームから一つ読み込み，これに対してパターンマッチを行う．マッチしたものが **leaf** タグの場合はもう一段階読み込んだ結果を結果として返し．**node** タグの場合は再帰関数 $f$ に () を適用したもの二つを足し算する．

図 2 のプログラムでは対応するソース言語側での結果が木であるため，ターゲット言語側では () が返り，出力ストリームに結果が書き込まれたが，図 6 のソース言語側での結果は整数になるため，ターゲット言語側でも同様に整数が返る．

# 3 型システム

本節では，ソース言語の型システムを与える．この型システムで型付けできるプログラムは，(1) 入力木の各ノードを深さ優先の順で一度だけ読み，(2) 返り値が木である場合には深さ優先の順でノードを構成することが保証される．これにより，我々の型システムで型付けできるプログラムは，それと等価なストリーム処理プログラムに変換することができる．プログラムが入力木を深さ優先の順で読むことを保証するため，順序付き線形型を用いて変数のアクセス順序を制限する．

## 3.1 型と型環境

**定義 3.1 (型)** ソース言語の型の集合は以下の構文によって定義される．

$$\begin{array}{rcl} \tau\ (\text{型}) & ::= & \textbf{Int} \mid \textbf{Tree}^d \mid \tau_1 \rightarrow \tau_2 \\ d\ (\text{モード}) & ::= & - \mid + \end{array}$$

**Int** は整数の型，$\textbf{Tree}^d$ は木の型である．木の型には入力木型 $\textbf{Tree}^-$ と出力木型 $\textbf{Tree}^+$ が存在する．入力木型は読み込みだけに使われる木の型で，出力木型はプログラム中で生成される出力木に対する型である．また，$\tau_1 \rightarrow \tau_2$ は $\tau_1$ 型から $\tau_2$ 型への関数型である..

我々の型システムでは，順序付きとそうでない型環境の 2 種類の型環境を導入する．前者は，変数の間にアクセス順序の制限の入った型環境であり，入力木が左から順に一回づつアクセスされることを保証するために用いる．後者は，変数のアクセス順序の制限はなく，プログラム実行中に木以外の中間結果を保持するために用いられる変数の型を管理するために用いる．

$$(E_t[v_1 + v_2], S_i, S_o) \longrightarrow (E_t[plus(v_1, v_2)], S_i, S_o) \quad \text{(Et-Plus)}$$
$$(E_t[(\lambda x.M)v], S_i, S_o) \longrightarrow (E_t[[v/x]M], S_i, S_o) \quad \text{(Et-App)}$$
$$(E_t[\mathbf{read}()], v; S_i, S_o) \longrightarrow (E_t[v], S_i, S_o) \quad \text{(Et-Read)}$$
$$(E_t[\mathbf{write}(v)], S_i, S_o) \longrightarrow (E_t[()], S_i, S_o; v) \quad \text{(Et-Write)}$$
$$(E_t[\mathbf{case\ leaf\ of\ leaf} \Rightarrow e_1 \mid \mathbf{node} \Rightarrow e_2], S_i, S_o) \longrightarrow (E_t[e_1], S_i, S_o) \quad \text{(Et-Case1)}$$
$$(E_t[\mathbf{case\ node\ of\ leaf} \Rightarrow e_1 \mid \mathbf{node} \Rightarrow e_2], S_i, S_o) \longrightarrow (E_t[e_2], S_i, S_o) \quad \text{(Et-Case2)}$$
$$(E_s[\mathbf{fix}\ f.e], S_i, S_o) \longrightarrow (E_t[[\mathbf{fix}\ f.e/f]e], S_i, S_o) \quad \text{(Et-Fix)}$$

図 5: ターゲット言語の簡約規則. ただし $plus(v_1, v_2)$ は $v_1$ と $v_2$ の整数としての和.

木構造処理 (ソースプログラム)：
$\mathbf{fix}\ f.\lambda t.(\mathbf{case}\ t\ \mathbf{of\ leaf}\ x \Rightarrow x \mid \mathbf{node}\ x_1\ x_2 \Rightarrow (f\ x_2) + (f\ x_1))$
ストリーム処理 (ターゲットプログラム)：
$\mathbf{fix}\ f.\lambda t.(\mathbf{case}\ \mathbf{read}()\ \mathbf{of\ leaf} \Rightarrow \mathbf{read}() \mid \mathbf{node} \Rightarrow f\ () + f\ ())$

図 6: 二分木の葉の要素の総和を求めるプログラム

**定義 3.2 (順序付き線形型環境)** 順序付き線形型環境は，$x_1 : \mathbf{Tree}^-, \ldots, x_n : \mathbf{Tree}^-$ の形の列である．ただし，$x_1, \ldots, x_n$ が互いに異なるものとする．順序付き型環境をメタ変数 $\Delta$ を用いて表す．$\Delta_1, \Delta_2$ は，順序付き線形型環境 $\Delta_1$ と $\Delta_2$ の連接を表す．

順序付き線形型環境 $x_1 : \mathbf{Tree}^-, \ldots, x_n : \mathbf{Tree}^-$ は，入力の部分木 $x_1, \ldots, x_n$ をこの順で丁度一回づつアクセスしなければならないことを表す．

**定義 3.3 (順序無し型環境)** （順序無し）型環境は，変数の型への束縛の集合 $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ である．ただし，$x_1, \ldots, x_n$ が互いに異なり，$\{\tau_1, \ldots, \tau_n\}$ は $\mathbf{Tree}^d$ を含まないものとする．型環境をメタ変数 $\Gamma$ を用いて表す．$\Gamma, x : \tau$ は，$\Gamma$ に $x : \tau$ を追加して得られる型環境を表す．また，$\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ を $x_1 : \tau_1, \ldots, x_n : \tau_n$ と略記する．

順序無し型環境には入力木型 $\mathbf{Tree}^-$ だけではなく出力木型 $\mathbf{Tree}^+$ も含まれない．これはストリーム処理のプログラムでは出力は評価が済むと順次書き出されなくてはならないため，メモリの中に出力木を溜め込み，後でまとめて出力するようなプログラムを排除するためである．

## 3.2 型判定規則

上に述べた型環境，順序付き線形型環境を用い，$\Gamma \mid \Delta \vdash M : \tau$ の形の型判定関係を定義する．これは，$M$ 中の自由変数が $\Gamma$，$\Delta$ に記述された型の値に束縛された環境の下で $M$ を評価すると，評価結果が型 $\tau$ の値になり，かつ評価の際には，$\Delta$ 中の変数に束縛されている部分木はこの順でアクセスされることを表す．例えば，$\Gamma = \{f : \mathbf{Tree}^- \to \mathbf{Int}\}$ かつ $\Delta = x_1 : \mathbf{Tree}^-, x_2 : \mathbf{Tree}^-$ のとき，

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

は成り立つが，

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

は成り立たない．後者は，$\Delta$ によって課せられている，入力木 $x_1, x_2$ をこの順でアクセスしなければならないという制約に違反している．

**定義 3.4 (型判定関係)** 型判定関係 $\Gamma \mid \Delta \vdash M : \tau$ は，図 7 の規則を満たす最小の関係である．

T-Var1，T-Var2 および T-Int は，変数および整数定数に関する型判定規則である．これらの規則では，式の中に現れる変数以外の変数が順序付き線形型環境に出現することを禁じている．これは，順序付き線形型環境に現れる変数がちょうど一回づつアクセスされることを保証するためである．

T-Abs1 および T-Abs2 は，$\lambda$ 抽象に関する規則である．T-Abs1 や T-Abs2 では，束縛される変数以外の変数が順序付き線形型環境に現れることを禁じている．これは，$\lambda$ 式によって生成されるクロージャの中に入力木

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

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

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

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

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

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

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

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

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

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

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

図 7: 型判定規則

が含まれることを防ぎ，入力木のアクセス順序を制御しやすくするための制限である[1]．

T-App は関数適用に関する規則である．$M_1 M_2$ を評価する際には，まず $M_1$ を評価した後に $M_2$ が評価されるため，$M_1$，$M_2$ の順序付き型環境 $\Delta_1$，$\Delta_2$ をこの順で結合している．一方，順序無し型環境はアクセス順序や回数に関する制限がないため，同一の型環境 $\Gamma$ が各部分式で用いられている．

T-Leaf および T-Node は，木を構成するための構文に関する規則である．T-App と同様，$M_1$，$M_2$ の順序付き型環境 $\Delta_1$，$\Delta_2$ をこの順で結合していることに注意されたい．

T-Case は，入力木の形による分岐命令に関する規則である．入力木 $M$ が $\mathbf{node}\ x_1\ x_2$ にマッチする場合，以後は部分木 $x_1, x_2$ がこの順でアクセスされなければならない．これを示すものは $M_2$ の順序付き型環境 $x_1 : \mathbf{Tree}^-, x_2 : \mathbf{Tree}^-, \Delta_2$ である．

T-Fix は，再帰関数のための型付け規則である．T-Abs2 と同様，順序付き線形型環境は空であるという制限を課していることに注意されたい．

以上の型判定規則に沿って図 2 のプログラムを型付けしたときの導出木を図 8 に示す．また，図 3 のプログラムは型付けできない．このプログラムが型付けされるためには $\Gamma \mid x_1 : \mathbf{Tree}^-, x_2 : \mathbf{Tree}^- \vdash \mathbf{node}\ (f\ x_2)\ (f\ x_1) : \mathbf{Tree}^+$ が必要だが，これは成り立たない．

### 3.3 型システムの健全性

上で定義した型システムは，通常の型システムと同様，型保存 (preservation) および進行 (progress) の性質を満たす．これらの性質により，式の評価が途中で停止せず，正しい型の値が得られることが保証される．

**定理 3.1 (型保存 (preservation))** $\Gamma \mid \Delta \vdash M : \tau$ かつ $M \longrightarrow M'$ ならば，ある $\Delta'$ が存在して $\Gamma \mid \Delta' \vdash M' : \tau$．

**定理 3.2 (進行 (progress))** $\emptyset \mid \emptyset \vdash M : \tau$ ならば，$M$ は値，もしくは $M \longrightarrow M'$ なる $M'$ が存在する．

### 3.4 型検査アルゴリズム

順序無し型環境 $\Gamma$，順序付き線形型環境 $\Delta$，束縛変数に対する型宣言が付いたソースプログラム $M$ を入力として，$\Gamma \mid \Delta \vdash M : \tau$ を満たす $\tau$ が存在するかどうかを判定するアルゴリズムを構成することができる．

基本的には型判定規則に従って下から上に導出木を作っていくのだが，T-App などでは $\Delta$ の分割方法が決まらないので，$\Gamma \mid \Delta \vdash M : \tau$ の代わりに関係 $\Gamma \mid \Delta \vdash M : \tau \Rightarrow \Delta'$ を導入する． これは「$M$ 中の自由変数が $\Gamma$，$\Delta$ に記述された型の値に束縛された環境の下で $M$ を評価すると，評価結果が型 $\tau$ の値になり，$\Delta'$ に含まれる変数が使われずに残る」ことを表す．例えば $\Gamma = f : \mathbf{Tree}^- \to \mathbf{Tree}^+$，$\Delta = x_1 : \mathbf{Tree}^-, x_2 : \mathbf{Tree}^-$ とすると

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

が成り立つ．この関係を用いると，規則T-App に対応する型検査規則は

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

と書ける．

この型検査規則は次のように型検査アルゴリズムとして見ることができる．$\Gamma, \Delta_1, (M_1\ M_2)$ が入力として与えられると $\Gamma, \Delta_1, M_1$ を入力として再帰的に型検査を行い出力として $\tau_1 \to \tau_2$ と $\Delta_2$ を得る．次に $\Gamma, \Delta_2, M_2$ を入力として再び型検査を行い出力として $\tau_3$ と $\Delta_3$ を得る．最後に $\tau_1 = \tau_3$ であることを確かめて $\tau_2$ と $\Delta_3$ を返す．

$\Gamma \mid \Delta \vdash M : \tau$ が成り立つかどうかを判定するには $\Gamma \mid \Delta \vdash M : \tau \Rightarrow \emptyset$ が成り立つかどうかを判定すればよい．従ってソースプログラム $M$ が型付けできるかどうかを判定するには $\emptyset, \emptyset, M$ をアルゴリズムに与え，出力された $\tau, \Delta'$ について $\Delta' = \emptyset$ が成り立つかどうかを見ればよい．

## 4 変換アルゴリズム

3 節で与えられた型システムによって型付けできるプログラムは，それと等価なターゲット言語のプログラムに変換できる．本節ではこの変換アルゴリズムを定義し，その正しさを証明する．

### 4.1 変換の定義

**定義 4.1 (項の変換関数)** ソース言語の項を入力とし，ターゲット言語の項を出力する関数 $\mathcal{A}$ を，図 9 によって定義する．

[1] この制限を緩めるためには，リソース使用法解析 [11] のように，関数の使用順序に関する情報も型で表せばよいが，型システムが複雑になる．

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

図 8: 導出の例. ただし $\Gamma = \{f : \mathbf{Tree}^- \to \mathbf{Tree}^+\}$, $\Gamma' = \{f : \mathbf{Tree}^- \to \mathbf{Tree}^+, x : \mathbf{Int}\}$

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

図 9: 変換アルゴリズム

以下で定義する関数 $[\![\cdot]\!]$ は，入力木を入力ストリームに変換するための関数である．

**定義 4.2 (木表現からストリームへの関数)** 木表現をストリームへ写像する関数 $[\![\cdot]\!]$ を以下に定義する．

$$\begin{aligned}
[\![\mathbf{leaf}\ i]\!] &= \mathbf{leaf}; i \\
[\![\mathbf{node}\ V_1\ V_2]\!] &= \mathbf{node}; [\![V_1]\!]; [\![V_2]\!]
\end{aligned}$$

## 4.2 変換アルゴリズムの正当性

この節では図 9 に示した変換アルゴリズム $\mathcal{A}$ が正しいことを証明する．変換アルゴリズムの正当性は以下の定理によって保証される．

**定理 4.1 (変換の正しさ)** $\emptyset \mid \emptyset \vdash M : \mathbf{Tree}^- \to \tau$ かつ $\tau \in \{\mathbf{Int}, \mathbf{Tree}^+\}$ ならば，任意の値 $V$ について以下が成り立つ．

*(i)* $MV \longrightarrow^* i$ ならば，$(\mathcal{A}(M)(), [\![V]\!], \emptyset) \longrightarrow^* (i, \emptyset, \emptyset)$

*(ii)* $MV \longrightarrow^* V'$ ならば，$(\mathcal{A}(M)(), [\![V]\!], \emptyset) \longrightarrow^* ((), \emptyset, [\![V']\!])$

*(iii)* $(\mathcal{A}(M)(), [\![V]\!], \emptyset) \longrightarrow^* (i, \emptyset, \emptyset)$ ならば，$MV \longrightarrow^* i$

*(iv)* $(\mathcal{A}(M)(), [\![V]\!], \emptyset) \longrightarrow^* ((), \emptyset, [\![V']\!])$ ならば $MV \longrightarrow^* V'$

(i) はソースプログラムの評価結果が整数であれば，ターゲットプログラムの評価結果も整数になることを，(ii) はソースプログラムの評価結果が木であれば，ターゲットプログラムはそれに対応する木を出力ストリームに出力することを表す．(iii) および (iv) は，逆にターゲットプログラムの評価が成功すれば，ソースプログラムも対応する結果を返すことを表す．

以下に，定理 4.1 の証明の概略を示す．証明の基本アイデアは，ソースプログラムの簡約の各ステップについて，対応するターゲットプログラムの簡約ステップが存在することを示すことである．ただし，2 節のソース言語の簡約規則はこの対応を示すのに不便であるため，

ソース言語のもう一つの簡約規則を定義し，(1) 2 節のソース言語の簡約規則と新しい簡約規則とが等価であること，(2) 新しい簡約規則に基づくソースプログラムの簡約ステップとターゲットプログラムの簡約ステップとが対応すること，を示す．

まず 2 節で定義したソース言語に対して順序を考慮した操作的意味を新たに定義し直す．

**定義 4.3 (順序付き環境)** **順序付き環境**は，$x_1 \mapsto V_1, \ldots, x_n \mapsto V_n$ の形の列である．

順序付き環境をメタ変数 $\delta$ を用いて表す．順序付き環境 $x_1 \mapsto V_1, \ldots, x_n \mapsto V_n$ は通常の環境（変数の有限集合から値への写像）と異なり，変数の値を $x_1, \ldots, x_n$ の順でしか読むことができない．

**定義 4.4 (更新されたソース言語の簡約規則)** 順序を考慮したソース言語の簡約関係 $(M, \delta) \longrightarrow (M', \delta')$ は図 *10* の規則を満たす最小の関係である．

図 10 中で導入されているメタ変数 $U$ は，木以外の値および入力木と対応する変数である．2 節で定義した簡約関係 $M \longrightarrow M'$ との違いは，入力木が代入されずに環境に保持されること（Es-Case2 と Es2-Case2 を比較せよ），および入力木にアクセスする際に，順序の制約がついていること（Es2-Case1 と Es2-Case2 では，参照される変数 $y$ が順序付き型環境の先頭になければならないことに注意せよ）である．従って，両者の簡約規則で違いが生じる可能性があるのは，2 節で定義した簡約関係 $M \longrightarrow M'$ では簡約ができるが，新しい簡約規則では順序の制限のために簡約が止まってしまう場合のみである．次の定理は，項が型付けできるのであれば，新しい簡約規則でも簡約が止まることはないことを保証する．

**定理 4.2** $\emptyset \mid x : \mathbf{Tree}^- \vdash M : \tau$ かつ $(M, x \mapsto V) \longrightarrow^* (M', \delta')$ ならば，$M'$ は値または変数であるか，ある $M'', \delta''$ が存在して $(M', \delta') \longrightarrow (M'', \delta'')$ が成り立つ．

上の定理より，型付けできるプログラムについては，2 つの簡約関係が等価であることがわかる．

**系 4.3** $\emptyset \mid \emptyset \vdash M : \mathbf{Tree}^- \to \tau$ かつ $\tau \in \{\mathbf{Int}, \mathbf{Tree}^+\}$ ならば，任意の値 $V$ に対して以下が成り立つ．

*(i)* $MV \longrightarrow^* W$ ならば，$(Mx, x \mapsto V) \longrightarrow^* (W, \emptyset)$

*(ii)* $(Mx, x \mapsto V) \longrightarrow^* (W, \emptyset)$ ならば $MV \longrightarrow^* W$

次の定理は，新しい簡約規則の下でのソースプログラムの評価とターゲットプログラムの評価とが一致することを示す．定理 4.1 は，系 4.3 と定理 4.4 の系である．

**定理 4.4** $\emptyset \mid \emptyset \vdash M : \mathbf{Tree}^- \to \tau$ かつ $\tau \in \{\mathbf{Int}, \mathbf{Tree}^+\}$ ならば，任意の $V$ に対して以下が成り立つ．

*(i)* $(Mx, x \mapsto V) \longrightarrow^* (i, \emptyset)$ ならば
$(\mathcal{A}(M)(), [\![V]\!], \emptyset) \longrightarrow^* (i, \emptyset, \emptyset))$

*(ii)* $(Mx, x \mapsto V) \longrightarrow^* (V', \emptyset)$ ならば
$(\mathcal{A}(M)(), [\![V]\!], \emptyset) \longrightarrow^* ((), \emptyset, [\![V']\!]))$

*(iii)* $(\mathcal{A}(M)(), [\![V]\!], \emptyset) \longrightarrow^* (i, \emptyset, \emptyset))$ ならば
$(Mx, x \mapsto V) \longrightarrow^* (i, \emptyset)$

*(iv)* $(\mathcal{A}(M)(), [\![V]\!], \emptyset) \longrightarrow^* ((), \emptyset, [\![V']\!]))$ ならば
$(Mx, x \mapsto V) \longrightarrow^* (V', \emptyset)$

以下では定理 4.4 の証明の概略を示す．証明は以下のように進む．まずソース言語の評価ステップとターゲット言語の評価ステップとの間の対応関係を定義する．次に簡約によってこの関係が保存されることを示す．最後に型付けられる項 $M$ と $\mathcal{A}(M)$ との間にこの対応をとることができることを示す．

はじめに評価ステップ間の対応関係を定義する．

**定義 4.5 (対応関係)** 対応関係 $M \sim (e, S_o)$ および $M \sim_\gamma (e, S_o)$ は図 11 の規則を満たす最小の関係である．

ただし $FV(M)$ は $M$ 中の自由変数の集合を意味する．また，関数 $\mathcal{A}_\gamma$ は定義 4.1 による関数 $\mathcal{A}$ の変数 $x$ に関する定義を

$$\mathcal{A}_\gamma(x) = \begin{cases} () & (x \in \gamma) \\ x & (x \notin \gamma) \end{cases}$$

に変更したものとする．図 11 中で導入されているメタ変数 $I$ は，簡約すべき項である．任意の項 $M$ は，次の簡約が存在するならば，必ず $E_s[I]$ の形で表せることに注意されたい．

$M \sim (e, S_o)$ は，ソースプログラムの実行状態 $M$ に対応するターゲットプログラムの実行状態が $(e, S_o)$ であることを表す．ただし，$e$ は残りの計算を表し，$S_o$ は出力済みのストリームを表す．例えば，ソースプログラムの **node** (**leaf** 1) (**leaf** $(2+3)$) という状態は，ターゲットプログラムの $(2+3, \mathbf{node}; \mathbf{leaf}; 1; \mathbf{leaf})$ という状態に対応づけられる．以下に対応関係の主要な規則を説明する．

規則C-Tree: $V$ は木を構成し終えた状態を表しているので，「残りの計算はなく，$V$ をストリームに出力し終えた状態」$((), [\![V]\!])$ が対応づけられる．

規則C-Node1: **node** $E_s[I]$ $M$ は，木の左側を構成中の状態を表している．したがって，ターゲットプログラム側の残りの計算は，$E_s[I]$ の部分の残りの計算 $e$ お

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

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

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

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

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

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

図 10: 更新されたソース言語の簡約規則．ただし $plus(i_1, i_2)$ は $i_1$ と $i_2$ の整数としての和.

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

$$\frac{M \sim_{FV(M)} (e, S_o)}{M \sim (e, S_o)}$$

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

$$\frac{}{V \sim_\gamma ((), [\![ V ]\!])} \text{(C-Tree)}$$

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

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

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

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

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

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

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

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

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

図 11: $M$ と $(e, S)$ の対応関係 $\sim$

よび $M$ に対応する計算 $\mathcal{A}_\gamma(M)$ を合わせた $e;\mathcal{A}_\gamma(M)$ である．また，ソースプログラムの出力木のうち構成済みの部分は，ルートがノードであることを表すタグおよび $E_s[I]$ 中の構成済みの部分であるので，対応する出力ストリームは $\mathbf{node};S_o$ となる．

次のステップに入る前に，補題を述べるのに使用する関数を三つ導入する．

**定義 4.6 (順序付き環境からストリームへの関数)** 順序付き環境をストリームへ写像する関数 $\langle\cdot\rangle$ を以下に定義する．

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

**定義 4.7 (順序付き環境から順序付き型環境への関数)** 順序付き環境を順序付き線形型環境へ写像する関数 $\langle\langle\cdot\rangle\rangle$ を以下に定義する．

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

補題 4.5 は更新された簡約規則においても型保存の性質が成り立ち，さらに簡約後の項は順序付き環境に対応する順序つき型環境のもとで型付けできることを述べている．

**補題 4.5 (更新された簡約における型保存)** $\Gamma\mid\langle\langle\delta\rangle\rangle\vdash M{:}\tau$ かつ $(M,\delta)\longrightarrow(M',\delta')$ ならば $\Gamma\mid\langle\langle\delta'\rangle\rangle\vdash M'{:}\tau$.

**証明** $(M,\delta)\longrightarrow(M',\delta')$ なので $M=E_s[I]$ と書ける．この評価文脈 $E_s$ の構造に関する帰納法による．

□

補題 4.6 は対応関係 $\sim$ が成り立っているならば，簡約を行っても対応関係が保存されることを述べている．$\longrightarrow^+$ は $\longrightarrow$ の推移的閉包を表す．

**補題 4.6 (簡約による対応の保存)** $\emptyset\mid\langle\langle\delta\rangle\rangle\vdash M:\tau$ かつ $\tau\in\{\mathbf{Int},\mathbf{Tree}^+\}$ と仮定する．$M\sim(e,S)$ ならば以下が成り立つ．

- $(M,\delta)\longrightarrow(M',\delta')$ ならば，$(e,\langle\delta\rangle,S)\longrightarrow^+(e',\langle\delta'\rangle,S')$ かつ $M'\sim(e',S')$ を満たす $e',S'$ が存在する．
- $(e,\langle\delta\rangle,S)$ が *1* ステップ簡約可能ならば $(M,\delta)\longrightarrow(M',\delta')$ なる $M',\delta'$ が存在する．

**証明** *1* 番目の条件は $(M,\delta)\longrightarrow(M',\delta')$ なので $M=E_s[I]$ と書ける．この評価文脈 $E_s$ の構造に関する帰納法による．*2* 番目の条件は対応関係 $M\sim_\gamma(e,S)$ の定義より明らか．

□

補題 4.6 により，型付けができておりターゲット言語の項との対応関係が成り立てば，簡約によって対応関係が保存されることが保証された．あとは $M$ と $\mathcal{A}(M)$ の間でこの対応関係を確認すればよい．実は $M$ と $\mathcal{A}(M)$ の間にはこの対応関係は成り立たないのだが，$\mathcal{A}(M)$ を数ステップ簡約したところで $M$ との対応がとれることが次の補題からわかる．

**補題 4.7 (対応の存在)** $\emptyset\mid\langle\langle\delta\rangle\rangle\vdash M:\tau$ かつ $\tau\in\{\mathbf{Int},\mathbf{Tree}^+\}$ ならば，次の二つの条件を満たす $e,S_o$ が存在する．

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

**証明** $M$ の構造に関する帰納法による．

□

補題 4.6 と補題 4.7 を用いれば，定理 4.4 は次のように証明できる．

**証明 (定理 4.4 の証明)** $\emptyset\mid\emptyset\vdash M:\mathbf{Tree}^-\to\tau$ と $\emptyset\mid x:\mathbf{Tree}^-\vdash x:\mathbf{Tree}^-$ より $\emptyset\mid x:\mathbf{Tree}^-\vdash M\ x:\tau$ が成り立つ．$\tau\in\{\mathbf{Int},\mathbf{Tree}^+\}$ であるから補題 *4.7* より $M\ x\sim(e,S_o)$ かつ $(\mathcal{A}_{FV(M)\cup\{x\}}(M\ x),[\![V]\!],\emptyset)\to^*(e,[\![V]\!],S_o)$ を満たす $e,S_o$ が存在する．$FV(M)=\emptyset$ かつ $x\notin FV(M)$ なので $\mathcal{A}_{FV(M)\cup\{x\}}(M\ x)=\mathcal{A}(M)()$ が成り立つ．

以下 $(ii)$ と $(iv)$ について証明する．$(i)$ と $(iii)$ は同様に証明できる．

*(ii)* 仮定 $((M\ x),x\mapsto V)\to^*(V',\emptyset)$ と補題 *4.5*，*4.6* より簡約列 $(e,[\![V]\!],S_o)\to^*(e',\emptyset,S'_o)$ が存在して $V'\sim(e',S'_o)$ が成り立つ．$\sim$ の定義より $e'=()$ かつ $S'_o=[\![V']\!]$．したがって $(\mathcal{A}(M)(),[\![V]\!],\emptyset)\to^*((),\emptyset,[\![V']\!])$ が成り立つ．

*(iv)* $(\mathcal{A}(M)(),[\![V]\!],\emptyset)\to^*((),\emptyset,[\![V']\!])$ を仮定する．簡約の各ステップで簡約方法は一意に定まるので，$(e,[\![V]\!],S_o)\to^*((),\emptyset,[\![V']\!])$ が成り立つ．

次に「$\emptyset\mid\langle\langle\delta\rangle\rangle\vdash M'{:}\mathbf{Tree}^+$ かつ $(e,\langle\delta\rangle,S'_o)\longrightarrow^*((),\emptyset,[\![V']\!]))$ および $M'\sim(e,S'_o)$ が成り立つならば，$(M',\delta)\longrightarrow^*(V',\emptyset)$ が成り立つ」ことを $(e,\langle\delta\rangle,S'_o)\longrightarrow^*((),\emptyset,[\![V']\!]))$ の簡約ステップ数 $n$ に関する数学的帰納法により示す．これにより，$M'=M\ x$, $\delta=x\mapsto V$ とおけば，$((M\ x),x\mapsto V)\to^*(V,\emptyset)$ と補題 *4.5* より $\emptyset\mid x:\mathbf{Tree}^-\vdash(M\ x){:}\mathbf{Tree}^+$ なので $((M\ x),x\mapsto V)\longrightarrow^*(V',\emptyset)$ が導かれる．

– $n = 0$ のとき，$e = (), \langle\delta\rangle = \emptyset, S'_o = [\![ V' ]\!]$ なので，$M' \sim (e, S'_o)$ より，$M' = V', \delta = \emptyset$. よって $(M', \delta) \longrightarrow^* (V', \emptyset)$ が成り立つ.

– $n \geq 1$ のとき，

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

を満たす $e', S_i, S''_o$ が存在する．補題 *4.6* より，以下の条件を満たす $M'', \delta', S'''_o$ が存在する.

* $(M', \delta) \longrightarrow (M'', \delta')$
* $M'' \sim (e'', S'''_o)$
* $(e, \langle\delta\rangle, S'_o) \longrightarrow^+ (e'', \langle\delta'\rangle, S'''_o)$

簡約の決定性より，$(e'', \langle\delta'\rangle, S'''_o) \longrightarrow^* ((), \emptyset, [\![ V' ]\!]))$ が成り立つ．従って，帰納法の仮定より $(M'', \delta') \longrightarrow^* (V', \emptyset)$．よって $(M', \delta) \longrightarrow^* (V', \emptyset)$ が成り立つ.

□

# 5 議論

本節は木構造操作からストリーム操作のプログラムへと変換することによって得られる効率と，言語の表現力について議論する.

## 5.1 効率

ソース言語による型付け可能なプログラムは 4 節で定義された変換規則に従い，ターゲット言語によるストリーム操作のプログラムへと変換される．以下の議論から，ターゲットプログラムはソースプログラムよりも入力木を展開しない分だけメモリの消費量が少なく，かつ計算ステップ数も増えないことがわかる.

まずメモリの消費量について比較する．補題 4.6 と補題 4.7 により，ソースプログラムの各実行ステップにターゲットプログラムの実行の複数ステップが対応することがわかる．ターゲットプログラムの実行中でメモリの割り当てが起こるのは，(i) クロージャが生成されてヒープにメモリが割り当てられる場合，(ii) 関数呼び出しが起こってスタックが延びる場合，である．補題 4.6 の証明の詳細をみると，このときの対応するソースプログラムの実行ステップでも，同様にクロージャの割り当て，関数呼び出しが起こることがわかる．したがって，ターゲット言語のみで起こるような余計なメモリ割り当ては存在しない．このため変換で得られたプログラムは木構造操作のプログラムよりも入力木用のメモリだけ使用メモリが少ないことがわかる（ただし，より厳密な議論を行うためには，ヒープを明示化した操作的意味論を用いる必要がある）.

次に計算ステップであるが，補題 4.6 と補題 4.7 によれば，ソースプログラムの 1 実行ステップはターゲットプログラムの複数の実行ステップに対応するが，証明の詳細を見ると，その増加分は出力ストリームに値を書き出すための実行ステップのみである．ソースプログラムの場合は，実行後にメモリに構成された出力木を最後にストリームに出力する操作が必要であるので，その操作を含めれば，結局計算ステップ数はソースプログラムとターゲットプログラムで本質的な違いはない（ただし簡約の定義の仕方に依存して定数倍の誤差はありうる）.

## 5.2 表現力

ソース言語で記述されたプログラムは 3 節で定義された型システムによって型付けされなければならないため，任意の関数プログラムが記述できるわけではない．以下では，ソース言語の表現力について議論を行う.

ソース言語で記述できるプログラムには図 12 で示したツリーマップ，図 13 で示したツリーフォールドのようなものが挙げられる．これらは，木の各ノードについて一様な処理を施すものであるが，それ以外にも図 14 のような，再帰の度に木構造を二段階以上アンフォールドし，ノードによって異なる操作をするようなプログラムも記述可能である.

またソース言語によるプログラムは順序付き線形型によって，入力木の各部分木を深さ優先の順で丁度一回ずつ使わなければならない，という制限を受けるが，入力の一部を捨てるようなプログラムも図 15 のように記述できる.

図 15 中の $M$ は入力として受けとった木の右側の部分木を返すプログラムである．我々の定義したソース言語では $M$ を単純に $M = \lambda t.(\mathbf{case}\ t\ \mathbf{of}\ \mathbf{leaf}\ x \Rightarrow \mathbf{leaf}\ x \mid \mathbf{node}\ x_1\ x_2 \Rightarrow x_2)$ とすると，型付けが出来ない．まず上のプログラムでは入力木の左部分木 $x_1$ が使用されていないため，順序付き線形型によって各部分木は丁度一回ずつ使用されなければいけない，という制限に反する．また，我々の型システムでは入力木型と出力木型を区別しているため入力木の右部分木 $x_2$ を出力として用いることができない．このため図 15 では *skip_tree* と *copy_tree* を用いて記述している．*skip_tree* は型 $\mathbf{Tree}^- \to \mathbf{Int}$ を持つ関数であり，これを利用して左部分木を段階的に全て読み込み，値を捨てる．また *copy_tree* は型 $\mathbf{Tree}^- \to \mathbf{Tree}^+$ を持つ関数であり，これを用いて右部分木を段階的に入出力する．なお，*skip_tree* などを変換時に自動的に挿入するように型システムを拡張することも可能である.

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

図 12: ツリーマップ

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

図 13: ツリーフォールド

$$\begin{aligned}\textbf{fix}\ \mathit{double_unfold}.\lambda t.(&\textbf{case}\ t\ \textbf{of}\ \textbf{leaf}\ x \Rightarrow \textbf{leaf}\ x \mid \textbf{node}\ x_1\ x_2 \Rightarrow \textbf{node}\\ &(\textbf{case}\ x_1\ \textbf{of}\ \textbf{leaf}\ y \Rightarrow \textbf{leaf}\ (y+1)\\ &\qquad \mid \textbf{node}\ y_1\ y_2 \Rightarrow \textbf{node}\ (\mathit{double_unfold}\ y_1)\ (\mathit{double_unfold}\ y_2))\\ &(\textbf{case}\ x_2\ \textbf{of}\ \textbf{leaf}\ z \Rightarrow \textbf{leaf}\ (z+1)\\ &\qquad \mid \textbf{node}\ z_1\ z_2 \Rightarrow \textbf{node}\ (\mathit{double_unfold}\ z_1)\ (\mathit{double_unfold}\ z_2)))\end{aligned}$$

図 14: 偶数段目の木の要素に 1 を加えた木を返す関数

$M = \lambda t.(\textbf{case}\ t\ \textbf{of}\ \textbf{leaf}\ x \Rightarrow \textbf{leaf}\ x \mid \textbf{node}\ x_1\ x_2 \Rightarrow (\mathit{skip_tree}\ x_1); (\mathit{copy_tree}\ x_2))$
ただし *copy_tree* および *skip_tree* はそれぞれ
$\textbf{fix}\ \mathit{copy_tree}.\lambda t.(\textbf{case}\ t\ \textbf{of}\ \textbf{leaf}\ x \Rightarrow \textbf{leaf}\ x \mid \textbf{node}\ x_1\ x_2 \Rightarrow \textbf{node}\ (\mathit{copy_tree}\ x_1)\ (\mathit{copy_tree}\ x_2))$
$\textbf{fix}\ \mathit{skip_tree}.\lambda t.(\textbf{case}\ t\ \textbf{of}\ \textbf{leaf}\ x \Rightarrow 0 \mid \textbf{node}\ x_1\ x_2 \Rightarrow (\mathit{skip_tree}\ x_1); (\mathit{skip_tree}\ x_2))$
で定義される関数とする.

図 15: 入力木の右側の木を返すプログラム

記述できないプログラムは，ストリームの順序で木を扱っていないものである．これには例えば図 3 のように木の左右を入れ替えるものが考えられる．また，アクセス順序は正しいがクロージャに入力木を入れるようなプログラムも記述不可能である．

### 5.3 XML 文書処理への適用

本研究では整数を葉の要素に持つ二分木を対象としたが，本研究のアプローチは XML 文書の処理にも応用できると考えられる．

本研究で対象としている二分木と，XML 文書との違いは，後者は (i) 論理的構造がラベル付きの $n$ 分木（しかも $n$ が定まらない場合がある）であり，(ii) 物理的構造（文書ストリーム）に各部分木の終端を表す閉じタグが入っている，という点である．一点目の違いは，型の上では次のような違いとして表される（下では ML の型宣言の構文を用いている）．

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

型 `tree` が二分木のデータ型を表すのに対し，`xmltree` がラベルつきの $n$ 分木を表す．このような型の違いに従えば，XML 処理のためにはソース言語の **case** 文は次の 2 種類に置き換えればよいことがわかる．

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

同様に型付け規則も自然に拡張することができ，後者の **case** 文の型付け規則は

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

とすればよい．T-Node と同様，$x : \textbf{xmltree}, xl : \textbf{treelist}, \Delta_2$ の部分がアクセス順序に関する制限を与えている．

二点目の違いである閉じタグについては，上の **caseSeq** 文に対応するターゲット言語の **case** 文において空列（**Nil**）であるかどうかを判定するために用いることになる（**caseSeq** $xl$ **of nil** $\Rightarrow \cdots$ が **case readtoken**() **of close_tag** $\Rightarrow \cdots$ となる）．

## 6 関連研究

木構造処理を行うプログラムからストリーム処理器を自動導出する研究としては中野 [13, 18] らのものがある． [13] で提案されている手法は，属性文法を用いて XML 文書に対する変換を記述し，ストリームと木構造の間の parsing, unparsing を行う文法と変換を表す文法とを記述的合成法 [7] を用いて合成することで，ストリームを直接扱う XML 処理器を自動的に得るというものである．この手法の我々の手法に対する利点としては，出力する木を一部メモリ上に貯めておけるという点が挙げられる．我々の手法では順序付き線形型の制限によりこのようなことができない．逆に我々の手法の利点としては, 型による制限を除けば普通の関数型プログラムとして変換を記述できるので分かりやすいということが挙げられる．また, 属性文法を使った場合は副作用を伴う処理が書けないが，我々の手法では副作用を伴った式を記述することも可能である．

[18] では [13] で提案された手法のフロントエンドとなるプログラミング言語 XTiSP が提案されている．この言語で変換を記述すれば，属性文法への翻訳が自動的になされるのでプログラマは直接属性文法を書く必要がない．XTiSP と我々の手法とでは再帰的な処理を記述する能力に差がある．我々の手法では入力木に対する再帰的な処理が記述できるが，XTiSP では再帰的な処理は制限された形でしか記述できない[2]．

木構造に対する操作をストリームに対する操作に自動変換する研究として，他に XPath 式 [3, 16] を対象としたもの [1, 8, 9] や XQuery [4, 6] を対象としたもの [12] が挙げられる．前者は木を受け取って条件にマッチする部分木の集合を返す関数として，後者は木を受け取って何らかの値を返す関数として我々のソース言語でも実装することができる．しかし，我々の手法では $\textbf{Tree}^+$ 型の制限があるために，これらの関数が返した値を別の計算に利用するためには一旦返り値をストリームにして別の計算の入力として扱えるようにする必要があるのに対し，これらの手法では中間にストリームを挟む必要がない．逆に我々の研究では再帰関数もサポートしているが, これらの研究で関数の使用までサポートしているものは我々の知る限りない．

戸沢ら [19] はストリーム処理を行う言語 XP++ を設計し，プログラムに対して DTD に基づいた型検査を行うことで，生成する XML 文書が DTD に合っていることをコンパイル時に保証している．この研究は我々の研究とは違い，ストリームを処理するプログラムを容

[2] XTiSP では *tree_map* のように木構造の各ノードを一様に処理する形の再帰しか記述できない．このため, 我々の手法で記述できる, 木の深さを引数として受け取ってその深さに応じて処理を変えるような変換は XTiSP では記述できない．

易に記述できる言語を開発することを目的としている．この言語ではストリームの操作を直接ユーザが記述しなければならないので，我々の提案する手法に比べてプログラミングが難しくなると考えられる．

本論文で利用した順序付き線形型は Polakow が [15] で提案した順序付き線形論理をベースとしている．この論文では，線形論理の拡張として順序付き線形論理を導入し，証明探索のアルゴリズムを与えている．さらに，順序付き線形論理の応用として順序付き線形論理プログラミング言語 Olli と順序付き線形論理フレームワーク OLF を提案している．順序付き線形論理を応用した研究としては，他に Petersen ら [14] のメモリ割り当てとデータ配置の正しさを保証する型システムが挙げられる．この型システムでは順序付き型環境が順序がメモリ上にあるオブジェクトの並び順を表現するために用いられているのに対し，我々の研究では時間的な順序関係を表現するのに用いられている．

## 7 結論

本研究では二分木に対する木構造処理プログラムを等価なストリーム処理プログラムへと変換するための順序付き線形型を用いた型システムと変換アルゴリズムの定義を行った．プログラムが変換可能かどうかは型システムによって静的に判定でき，また変換システムによって得られたプログラムは論理的に変換前のものと等価であることが保証される．この研究により木構造処理の利点とストリーム処理の利点の両方を得ることができるようになった．今後の課題として，実際の XML 文書などへの適用があげられる．XML では木の構造が複雑になり，タグの名前や属性など考慮すべき点が多い．また，本研究で扱えるプログラムには木構造を左から深さ優先順に丁度一回ずつ走査しなければならないという強い制限が課せられているが，木構造の一部をメモリに保持することを許すなど，より広範囲のプログラムが扱えるように型システムを拡張することも今後の課題である．

### 謝辞

本研究の内容に関して一緒にご議論いただいた東京大学米澤研究室および東京工業大学小林研究室の「プログラミング言語の基礎」グループのメンバーの方々，有益なコメントをくださった査読者の方々に感謝いたします．

## 参考文献

[1] 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 PLANX*, oct 2002.

[2] Henry G. Baker. Lively linear lisp – look ma, no garbage! *ACM SIGPLAN Notices*, 27(8):89–98, 1992.

[3] 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/.

[4] 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/.

[5] 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, oct 2000. http://www.w3.org/TR/REC-xml.

[6] 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/.

[7] Harald Ganzinger and Robert Giegerich. Attribute coupled grammars. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, 1984.

[8] T. Green, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. Technical report, University of Washington, 2001.

[9] Ashish Kumar Gupta and Dan Suciu. Stream processing of XPath queries with predicates. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, jun 2003.

[10] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology (TOIT)*, 3(2):117–148, 2003.

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

[12] 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.

[13] Keisuke Nakano and Susumu Nishimura. Deriving event-based document transformers from tree-based specifications. In Mark van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001.

[14] 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.

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

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

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

[18] 中野 圭介. XML ストリーム処理器の自動導出が可能な XML 変換言語の設計. In **日本ソフトウェア科学会第** *20* **回大会** *(2003* **年度***)* **論文集**, 2003.

[19] 戸沢 晶彦, 萩谷 昌己. XML ストリーム処理に対する型検査およびノード補間の技法. In *PPL2003* **プログラム・オンライン論文集**, 2003.