JAIST Repository
https://dspace.jaist.ac.jp/

Title	逐次型戦略に基づくTRSコンパイラの構築
Author(s)	関，康夫
Citation	
Issue Date	1997-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/1072
Rights	
Description	Supervisor:酒井 正彦，情報科学研究科，修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

修 士 論 文

逐次型戦略に基づく TRS コンパイラの構築

指導教官　酒井正彦 助教授

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

関康夫

1997 年 2 月 14 日

Copyright © 1997 by Yasuo Seki

目 次

1 はじめに **1**

2 TRS **3**

2.1 基本用語 3

2.2 項書換え系 4

3 Forward-Branching オートマトンを用いた効率的な書換え **6**

3.1 直交項書換え系 6

3.1.1 直交項書換え系 6

3.1.2 必須リデックス 7

3.1.3 強逐次性 8

3.2 Forward-Branching 9

3.3 HNF アルゴリズムと正当性 11

3.4 NF を求めるアルゴリズム 16

4 TRS コンパイラの実装 **17**

4.1 変換プロセス 17

4.2 効率的実現の詳細 19

4.2.1 セル構造 19

4.2.2 項の共有 20

4.2.3 参照カウンタ 20

5 評価 **22**

5.1 評価 22

6 まとめ **25**

謝辞 **26**

参考文献 **26**

A 評価プログラム **29**

A.1 本コンパイラ/Cdimple コンパイラ用評価プログラム 29

A.2 Elan コンパイラ用評価プログラム 31

A.3 SML-NJ コンパイラ用評価プログラム 34

A.4 Gofer コンパイラ用評価プログラム 38

第 1 章

はじめに

項書換え系は等式を左辺から右辺への書換え規則と見なした計算モデルである．項書換え系は関数型言語や代数仕様のように等式に基づくプログラム，定理自動証明，プログラムの検証・変換などの基礎を与えるモデルとして多くの研究がなされてきた．

項が与えられると書換え規則の適用により簡約され，これ以上簡約できない項が得られる．この簡約出来ない項は正規形と呼ばれ計算の解と見なされる．与えられた項が正規形を持つのに簡約順序によっては正規形が得られないことがある．従って，正規形を持つ項に対し正規形の得られる簡略化戦略 (正規化戦略) は重要である．

関数型言語を実行するシステムとして gofer や sml，elan など多くのシステムが開発されている．gofer は小さな haskell ライク言語のための関数型言語システムである [Jon94]．gofer は遅延評価 (lazy evaluation) や高階関数 (higher-order functions)，多相 (polymorphism) 型などの特徴を持つ．sml は ml 言語のシステムであるが，gofer のような特徴の他に非関数的な機能である例外処理を備えている．このようなシステムを使って実用的なアプリケーションが構築されている．関数型言語の計算モデルである項書換え系の効率のよい実現は，このような実用的なアプリケーションを構築するうえでも重要である．しかしながら，システムによっては項が正規形を持つのに，正規形が求まらずに停止しないことがある．この点から正規形を持つ項に対しては必ず正規形を求めるシステムの存在が重要である．

シンプルな機能を持ち TRS を実行する手続き型のプログラムを生成するシステムとして酒井らの Cdimple [Sak87] がある．これは TRS から C プログラムへのコンパイラであり，生成されたプログラムは最内戦略で項を書換えて正規形を求める．また，Cdimple はユーザ定義の C プログラムを取り込むことができる．しかしながら最内戦略で評価を行っているため正規形が存在しても求められないことがある．

左線形かつ重なりのない直交書換え系においては，正規形を持つ項はインデックスを書換えることにより，必ずその正規形を求めることができることが知られている．Huet らは与えられた項からインデックスを探索するオートマトンを示している [HL91]．しかし，一度書換えると次のリデックスを発見するために，

項のルートから再びチェックしなければならず効率が悪い．インデックスを効率的に発見する手法として Strandh らの FB インデックス探索オートマトンがある [Str89, Dura94]．このオートマトンでは書換え後，次のリデックスを探索するためには書換えた項から見て行けばよい．

本研究ではこの FB インデックス探索オートマトンを手続き型言語のプログラムに埋め込むことにより，高速に実行するための TRS コンパイラの構築を目的として Cdimple の改良を行う．

本研究で構築した TRS コンパイラは FB インデックスオートマトンを埋め込んだ C 言語プログラムを生成する．本コンパイラで生成されたプログラムは Gofer で生成されたプログラムより若干高速であることを示す．また，gofer compiler や sml compiler, elan compiler などで生成したプログラムでは停止しないが，本コンパイラで生成したプログラムでは停止する例を示す．

以下では，第 2章で項書換え系に関する諸定義を行う．第 3章では，Huet ら [HL91] により提案された逐次性について解説し，Strandh[Str89] や Durand[Dura94] により提案された forward-branching(FB) について説明する．更に，FB のクラスでは FB インデックスオートマトンを用いると項が正規形を持つとき必ず正規形が求まることを示す．第 4章では，TRS コンパイラの実装方法について述べる．第 5章では，競合する TRS コンパイラ (Cdimple，Elan compiler，Sml-NJ compiler，Gofer compiler) とのコンパイルされたプログラムの実行速度について比較した結果について述べる．第 6章では，本研究のまとめと今後の課題について述べる．

第 2 章

TRS

本章では項書換え系 (TRS) に関する基本的な用語や概念を文献 [HL91, KM91] に従って定義する.

2.1 基本用語

関数記号の集合を $\mathcal{F} = \{F, G, H, \cdots\}$, 変数の集合を $\mathcal{V} = \{x, y, z, \cdots\}$ (ただし $\mathcal{F} \cap \mathcal{V} \neq \phi$) とする. このとき**項** $(t, s, \cdots)$ は写像 $arity : \mathcal{F} \to \mathcal{N}$ を用いて次のように再帰的に定義される.

- $x \in \mathcal{V}$ は項である.
- $F \in \mathcal{F}$, $arity(F) = n$ $(n \geq 0)$ に対し $t_1, \cdots, t_n$ が項ならば $F(t_1, \cdots, t_n)$ も項である.

$\mathcal{V}$ と $\mathcal{F}$ により生成される項全体よりなる集合を $T(\mathcal{F}, \mathcal{V})$ で記し, 二つの項 t_1, t_2 が同一ならば $t_1 \equiv t_2$ と記す. また, 項 $t \equiv F(t_1, \cdots, t_n)$ に出現している変数の集合を $Var(t)$ で表し, 最外の関数記号 F を $head(F(t_1, \cdots, t_n)) = F$ で記し**根記号**と呼ぶ.

正整数の列の集合 N_+^*を用いて項に出現する関数記号の**出現位置**を以下のように定める.

定義 2.1.1 $\Lambda \in \mathcal{O}(t),\ u \in \mathcal{O}(t_i) \Rightarrow iu \in \mathcal{O}(F(t_1, t_2, \cdots, t_n))\ for\ 1 \leq i \leq n.$

- 根記号の出現位置を ε (空列を意味する) とする.
- $C[F(t_1, \cdots, t_n)]$ における F の出現位置が u のとき各 t_i の根記号の出現位置を $u \cdot i$ とする.

この出現位置に関して $u \preceq v \overset{\text{def}}{\iff} \exists w,\ v = u \cdot w$ で半順序 $\preceq$ を定義する. また, 項 t の**部分項**で出現位置 u を根とする部分項 t' を t/u , あるいは, u を省略して $t' \trianglelefteq t$ と記す.

この時, 出現位置 $v \preceq u$ に対し t' は v の内側に出現すると言う. 特に $u \neq \varepsilon$ のとき t' を t の**真部分項**と呼ぶ.

定義 2.1.2 項 t，s とする．$\exists p \in \mathcal{O}(t)$ であるとき 項 t の p の位置を s に置き換えることを $t[s]_p$ と表す．

代入θは変数から項への写像である．これは$\theta(F(t_1,\ldots,t_n)) \equiv F(\theta(t_1),\cdots,\theta(t_n))$ のように項から項への写像に拡張される．以後は $\theta(t)$ を $t\theta$ で略記する．

2.2 項書換え系

この節では文献 [HL91, KM91] に従って項書換え系 (TRS) を定義する．また，本論文を通して使用される項書換え系に関する用語を定義する．項書換え系を計算モデルと考えると，項は計算の対象を表し，項の書換えは計算に対応する．

定義 2.2.1 書換え規則とは，項 l, rの順序対 (l, r) で $l \notin \mathcal{V}$かつ $Var(l) \supseteq Var(r)$ の変数条件を満たすものである．書換え規則の集合 Rによる 2 項関係 $\underset{\mathcal{R},p}{\longrightarrow}$ を以下のように定義する．

$$t \underset{\mathcal{R},p}{\longrightarrow} s \overset{\text{def}}{\iff} \exists l \to r \in R, \exists \theta, t_p \equiv l\theta \text{かつ } s \equiv t[r\theta]_p$$

明らかな場合には $t \underset{\mathcal{R},p}{\longrightarrow} s$ を $t \underset{\mathcal{R}}{\longrightarrow} s$ あるいは $t \to s$ で略記する．ここで $t \underset{\mathcal{R}}{\longrightarrow} s$ を t から s への**リダクション**と呼ぶ．また，t の部分項 $l\theta$の出現を**リデックス**と呼び，リデックス $l\theta$における lに属する関数記号の出現位置の集合 (すなわち $l\theta$の上部 l) を**リデックス・パターン**と呼ぶ．リデックスを持たない項を**正規形**と呼ぶ．関係$\underset{R}{\to}$の反射推移閉包を$\underset{R}{\overset{*}{\to}}$, 推移閉包を$\underset{R}{\overset{+}{\to}}$, 対称閉包を$\underset{R}{\leftrightarrow}$で表す．また，$t_1 \overset{*}{\to} s \overset{*}{\leftarrow} t_2$となる項 s が存在する時 $t_1 \downarrow t_2$で記す．

定義 2.2.2 項書換え系 (TRS) は，関数記号の集合 $\mathcal{F}$, 変数の集合 $\mathcal{V}$, 書換え規則の集合 $R = \{l_1 \to r_1, \cdots, l_n \to r_n\}$ によって定義される$< \mathcal{F}, \mathcal{V}, R >$である．以後書き換え規則の集合 Rで項書換え系を表す．

例 2.2.3 次の例は $0, S(0), S(S(0)), \cdots$を $0, 1, 2, \cdots$とみなした場合，自然数上の加算に対応する項書換え系の例である．

$$R = \begin{cases} x + 0 & \to & x \\ x + S(y) & \to & S(x + y) \end{cases}$$

この項書換え系において次のようなリダクション列が存在する．

$$\underline{S(S(0)) + S(0)} \to S(\underline{S(S(0)) + 0}) \to S(S(S(0)))$$

このリダクション列は項 $S(S(0)) + S(0)$ を評価した結果が正規形 $S(S(S(0)))$ となることを表し，$2 + 1$ を計算した答が 3 であると言う事を意味する．ここで，下線はリデックスに対応する．具体的には，項 $S(S(S(0)) + 0)$ はリデックス $S(S(0)) + 0$ とリデックスパターン $\{1, 12\}$ を持つ．

項書換え系を計算モデルとして考えた場合，停止性はいかなる書き換えによる計算も必ず停止する事を，また，合流性はその書き換えによって得られる答えが高々一つである事を保証する性質である．したがって，項書換え系の定理自動証明や関数言語への応用を考えた場合，停止性と合流性は特に重要な性質となっている．以下では停止性と合流性の定義を簡単に行なう．

定義 2.2.4 項書換え系 R において無限リダクション列 $t_0 \to t_1 \to t_2 \to \cdots$ が存在しない時 R は**停止性**を持つという．

定義 2.2.5 任意の項 t_1, t_2に対し *"*$t_1 \overset{*}{\leftarrow} t \overset{*}{\rightarrow} t_2$ならば $t_1 \downarrow t_2$*"* が成立するとき，項 t は Rにおいて合流性を持つという．任意の項 t が Rにおいて合流性を持つ時 Rは**合流性**を持つ，または**チャーチ・ロッサー (Church-Rosser) 性**を持つという．

第 3 章

Forward-Branching オートマトンを用いた効率的な書換え

ここでは Strandh, Durand ら [Str89, Dura94] の定義に従って，Forward-Branching system についての説明と，その準備として直交項書換え系における関連する事柄について Huet ら [HL91]，Klop ら [KM91] の定義に従って説明する．

3.1 直交項書換え系

3.1.1 直交項書換え系

この節では，直交項書換え系 (orthogonal TRS) と呼ばれる項書換え系を定義する．そのために，左線形と重なりについて説明する．

項は，同じ変数を 2 つ以上含まないときに，線形 (linear) であるという．TRS $\mathcal{R}$ は，すべての書換え規則の左辺が線形であるとき，左線形 (left linear) であるという．

$l \to r$ と $l' \to r'$ を $\mathcal{R}$ の書換え規則とする．l と l' は同じ変数を含まないように変数の名前変えが行われているとする．出現 u における l の部分項 $t/u \notin \mathcal{V}$ に対して，$\theta \equiv l'\theta$ を満たす θ が存在するとき $l \to r$ と $l' \to r'$ は重なると言う．ただし，$l \to r$ と $l' \to r'$ が同じ書換え規則であるときは，$u \neq \varepsilon$ とする．任意の 2 つの書換え規則が重ならないとき，$\mathcal{R}$ は重なりがないという．

例 3.1.1 $\mathcal{R}$ を次のようにする．

$$\mathcal{R} = \begin{cases} F(G(A,x),y) \to H(x) \\ G(x,B) \to G(x,x) \end{cases}$$

$\mathcal{R}$ は左線形である．変数の名前変えをして $l \equiv F(G(A,x),y), l' \equiv G(z,B)$ とすると $1 \in \mathcal{O}(l)$ において重なりがあるため，$\mathcal{R}$ は重なりを持つ．

定義 3.1.2　TRS $\mathcal{R}$ が左線形で重なりがないならば，$\mathcal{R}$ を直交項書換え系 (orthogonal TRS) という．

3.1.2　必須リデックス

一般に，正規形でない項には複数のリデックスが存在するため，書換えるリデックスにより異なる簡約が得られる．

例 3.1.3 (Huet & Lévy [HL91])　$\mathcal{R}$ を次のようにする．

$$\mathcal{R} = \begin{cases} F(x,A) \to B \\ C \to C \\ D \to A \end{cases}$$

項 $F(C,D)$ は以下の簡約列が示すように正規形 B をもつ．

$$F(C,\underline{D}) \to \underline{F(C,A)} \to B$$

しかし，$F(C,D)$ からの無限の簡約列も存在する．

$$F(\underline{C},D) \to F(\underline{C},D) \to \cdots$$

この例が示すように，合流性をみたす TRS でも正規形を求めることができない場合がある．したがって，正規形をもつ項に対しては，必ずその正規形が得られる簡約戦略が必要である．そのような戦略を正規化戦略という．簡約戦略と正規化戦略は，抽象簡約系において以下のように定義される．

定義 3.1.4　$\mathcal{A} = \langle A, \to \rangle$ とする．二項関係 $\to_s$ は $\to_s \subseteq \to^+$ でかつ$\to_s$ に関する任意の正規形は $\to$ に関する正規形でもあるとする．このとき，$\to_s$ を $\mathcal{A}$(または $\to$) に対する簡約戦略 (reduction strategy) という．$\to_s \subseteq \to$ であるときは 1 ステップ戦略といい，そうでないときは多ステップ戦略という．

定義 3.1.5　$\mathcal{A}$(または $\to$) に対する簡約戦略 $\to_s$ が正規化戦略 (normalizing strategy) であるとは，$\to$ に関する正規形をもつ任意の $a \in A$ に対して，$a \equiv a_0 \to_s a_1 \to_s a_2 \to_s \cdots$ なる無限の簡約列が存在しないことである．

直交項書換え系に対する正規化戦略については，Huet ら [HL91] によって重要な結果が示されている．以下では，その結果について説明する．

定義 3.1.6　$A : t \to s$ を $l \to r \in \mathcal{R}$ に対し $t/u \equiv l\theta$, $s \equiv t[u \leftarrow r\theta]$ である 1 ステップ簡約とする．出現 $v \in \mathcal{O}(t)$ の子孫 (descendant) の集合 $v\backslash A$ を次のように定義する．

$$v\backslash A = \begin{cases} \{v\} & v < u \text{ または } v \perp u \text{ のとき} \\ \{u.w_1.v_1 | r/w_1 \equiv l/w\} & v = u.w.v_1, w \in \mathcal{O}_{\mathcal{V}}(l) f \text{のとき} \\ \phi & \text{上記以外のとき} \end{cases}$$

出現の集合 V に対し，$V\backslash A = \bigcup_{v \in V} v\backslash A$ とし，子孫は任意の長さの簡約に対しても定義される．直交項書換え系の場合，リデックス出現の子孫は必ずリデックス出現である．

定義 3.1.7　u を t のリデックス出現とする．t から正規形への任意の簡約において，u のある子孫が書換えられているならば，u は必須 (needed) であるといい， t/u を必須リデックスという．

定理 3.1.8 (Huet & Lévy [HL91])　$\mathcal{R}$ を直交項書換え系とする．

(1) t が正規形でないならば，t は必須リデックスをもつ．

(2) t が正規形をもつならば，必須リデックスを無限に書換える t からの簡約列は存在しない．　□

3.1.3　強逐次性

前節において述べたように，項が正規形をもつならば，必須リデックスを繰り返し書換えることによって正規形が得られる．したがって，必須リデックスを書換える簡約は正規化戦略である．一般に，リデックスが必須リデックスかどうかは正規形への全ての簡約を調べなければならないが，これは一般には決定不能であるため Huet [HL91] らによって必須リデックスを決定できる TRS を強逐次系が提案された．これ以後においては，特に述べない限り，項書換え系 $\mathcal{R}$ は直交であるとして話を進める．

Ω を $\mathcal{F}$ に出現しない定数とする．$\mathcal{T}(\mathcal{F} \cup \{\Omega\}, \mathcal{V})$ を $\mathcal{T}_\Omega$ で表し，その要素を Ω-項と呼ぶ．Ω は未簡約項を表すための定数である．Ω-項 t の Ω-出現の集合 $\mathcal{O}_\Omega(t)$ を $\mathcal{O}_\Omega(t) = \{u \in \mathcal{O}(t) | t/u \equiv \Omega\}$ ，t の Ω 以外の出現の集合を$\overline{\mathcal{O}}_\Omega(t) = \mathcal{O}(t) - \mathcal{O}_\Omega(t)$ と定義する．リデックスを含まない Ω-項を Ω-正規形，リデックスも Ω も含まない Ω-項を正規形という．Ω-正規形の集合を NF_Ω，正規形の集合を NF で表す．

$\mathcal{T}_\Omega$ 上の接頭順序 $\leq$ を次のように定義する．

(i) 任意の $t \in \mathcal{T}_\Omega$ に対し，$\Omega \leq t$

(ii) $s_i \leq t_i$ $(1 \leq i \leq n)$ のとき，$F(s_1, \cdots, s_n) \leq F(t_1, \cdots, t_n)$

(iii) 任意の $x \in \mathcal{V}$ に対し，$x \leq x$.

Ω-項 r が存在して $r \geq t$ かつ $r \geq s$ であるとき，$t \uparrow s$ と書く．このとき t と s の上限を $t \sqcup s$ と表す．また，t と s の下限を $t \sqcap s$ と表す．

t_Ω は t の全ての変数を Ω で置き換えて得られた Ω-項，t_x は t の全ての Ω を x で置き換えて得られた項を表す．

定義 3.1.9　簡約関係 $\to_?$ を次のように定義する．$u \in \mathcal{O}(t)$ がリデックス出現で，Ω-項 p が存在して $s \equiv t[u \leftarrow p]$ ならば，$t \to_? s$．

定義 3.1.10　$\mathcal{T}_\Omega$ 上の述語 $nf_?$ を次のように定義する．

$$nf_?(t) = true \iff s \in \text{NF が存在して } t \to_?^* s.$$

定義 3.1.11 (インデックス)　Ω-項 t の Ω-出現 u がインデックス (index) であるとは，任意の Ω-項 s に対し $s \geq t$ かつ $nf_?(s) = true$ ならば $s/u \not\equiv \Omega$ が成立することである．項 t のインデックスの集合を $I(t)$ で表す．

以下では，$u \in I(t)$ なる $u \in \mathcal{O}_\Omega(t)$ が存在するとき，t はインデックスをもつという．

定義 3.1.12 (強逐次系)　任意の Ω を含む Ω-正規形が $nf_?$ に関するインデックスをもつならば，$\mathcal{R}$ を強逐次系 (strong sequential system) という．

インデックスの計算可能性は以下のように示されている．

定義 3.1.13　簡約関係 $\to_\Omega$ を次のように定義する．ある $u \in \mathcal{O}(t)$, 書換え規則の左辺 l に対し，$t/u \uparrow l_\Omega$, $t/u \not\equiv \Omega$ かつ $s \equiv t[u \leftarrow \Omega]$ ならば，$t \to_\Omega s$．

簡約 $\to_\Omega$ は，合流性，停止性をみたす [KM91, Toy93]．$\to_\Omega$ に関する t の正規形は一意に定まるので，以下ではこれを $\omega(t)$ と表す．次の補題は，Huet ら [HL91] や Klop ら [KM91] によって示されている．

補題 3.1.14　$t \in \mathcal{T}_\Omega, u \in \mathcal{O}_\Omega(t), \bullet$ を t と書換え規則に現れない定数とする．以下の条件は同値である．

i). $u \in I(t)$;

ii). $\omega(t) \not\equiv \omega(t[u \leftarrow \bullet])$;

iii). $u \in \mathcal{O}(\omega(t[u \leftarrow \bullet]))$. □

直交項書換え系に対する強逐次性の決定可能性は Huet ら [HL91] や Klop ら [KM91]，Comon[Com95] によって示されている．

3.2 Forward-Branching

ここでは Forward-Branching(FB) に関する用語を定義する．

定義 3.2.1 はじめに $Red_\Omega, Red^{\prec}_\Omega, Red'_\Omega, F(\vec{\Omega})$ について定義しておく.

$Red_\Omega = \{t_\Omega \mid t \in Red\}$

$Red^{\prec}_\Omega = \{t \mid \Omega \prec t \prec t', t' \in Red_\Omega\}$

$Red'_\Omega = \{t \mid \exists s \in Red_\Omega, \exists u \in \mathcal{O}(s), s|_u = t \ and \ head(t) \notin \mathcal{C}\}$

$F(\vec{\Omega}) \equiv F(\Omega, \cdots, \Omega)$

定義 3.2.2 (firm occurrence) *Let* $u \in \mathcal{O}_\Omega(t)$,

$$u \ is \ firm \ occurrence \ of \ t \overset{\text{def}}{\iff} \forall v \in \overline{\mathcal{O}}_\Omega(t), [t/v \in NF_\Omega \vee v \prec u] \wedge u \in I(t).$$

定義 3.2.3 (transitive firm occurrence) *Let* $u \in \mathcal{O}_\Omega(t)$, u *is transitive firm occurrence of* t

$\overset{\text{def}}{\iff}$ u *is a firm occurrence of* $t \wedge \forall s \in Red'_\Omega[t \prec s \Rightarrow s/u \not\equiv \Omega]$

定義 3.2.4 (状態)

$$Q \overset{\text{def}}{=} \{\langle t, p\rangle | t \in Red^{\prec}_\Omega, p \ is \ a \ transitive \ firm \ occurrence \ of \ t\} \cup Red_\Omega$$

特に最終状態の集合 Q_f と初期状態 q_0 は以下のように定義する.

$$Q_f \overset{\text{def}}{=} Red_\Omega$$

$$q_0 \overset{\text{def}}{=} \langle \Omega, \Lambda \rangle$$

定義 3.2.5 (インデックスポイント) インデックスポイントは $\exists t \in Red^{\prec}_\Omega, \exists p, t|_p = \Omega$ であるような項 t と出現 p の対 *(t,p)* である.

オートマトンを考えたとき，このインデックスポイントはオートマトンの状態に相当する.

定義 3.2.6 (transfer function [Dura94]) オートマトン A において，q から次の状態へ遷移するときに適用される関数記号を f とする．関数記号を適用して遷移先の状態を求める関数 *transfer function* $\eta(q, f)$ は次のように定義される． $\eta : Q * F \to 2^Q$の定義

$$\eta(\langle t, p\rangle, f) = \{\langle t[f(\Omega, \cdots, \Omega)]_p, p\rangle \in Q\} \cup \{t[f(\Omega, \cdots, \Omega)]_p \in Red_\Omega\}$$

非決定的だがひとつだけ選べば充分. $\eta(q, f)$ を用いて次の状態へ遷移することを *transfer transition* と呼ぶ.

状態 s において transfer transition ができないとき，つまり，インデックスポイントでのチェックが失敗したときにそのポイント，もしくは祖先のポイントからチェックする必要がある．このときに遷移可能な状態を求める関数として$\phi(s)$ を定義する.

定義 3.2.7 [failure function]

$$\phi : Q \to Q * Q$$

$$\phi(\langle t, p\rangle) = (\langle t/p', q\rangle, \langle s, p'\rangle)$$

$where$

p' *is maximal in* $\mathcal{O}(t)$

$s.t.\ p = p'q \wedge \langle t/p', q\rangle \in Q$

$s \leq t\ \wedge\ \langle s, p'\rangle \in Q$

$\phi(s)$ を用いて次の状態へ遷移することを *failure transition* と呼ぶ.

本論文では特に混乱が生じない限り transfer transition のことを transfer，failure transition のことを failure もしくは 単に fail と呼ぶことにする.

定義 3.2.8 (オートマトン) オートマトン A は 次のように構成される.

$$A = \langle Q, Q_f, \mathcal{O}, q_0, \eta, \phi\rangle$$

定義 3.2.9 インデックスオートマトンはインデックスポイントとそれらに作用する *tranfer transition, failure transition* から構成されている.

定義 3.2.10 (FB index tree [Dura94]) インデックスオートマトンのどの状態 q に対して初期状態から *transfer transition* のみで到達可能であるならば，インデックスオートマトンを *FB* インデックスオートマトンと呼ぶ.

定義 3.2.11 (Forward-branching [Dura94])

$$\mathcal{R}\ is\ forward-branching \overset{\text{def}}{\iff} \forall t \in Red^{\prec}_{\Omega}[\exists u \in \mathcal{O}(t)[u\ is\ a\ transitive\ firm\ occurrence\ of\ t]]$$

今後は特に混乱が生じない限り FB インデックスオートマトンが存在するクラスを FB と呼ぶ.

例 3.2.12 *forward-branching* の例を示す.

$R = \{F(G(x, A), A) \to x, F(G(x, A), B) \to G(x, x), G(B, B) \to B\}$ とする．よって，

$Red_\Omega = \{F(G(\Omega, A), A), F(G(\Omega, A), B), G(B, B)\}$

$Red'_\Omega = Red_\Omega \cup \{G(\Omega, A)\}$

を得る．Red_R における *FB* インデックスオートマトン A は図 *3.1* のようになる.

FB のクラスが強逐次性 (Strongly Sequential) のクラスのサブセットであることは Durand[Dura94] により示されている.

3.3 HNF アルゴリズムと正当性

外山による Transitive system(TS) は書換えた場所から次のインデックスを探すことのできるクラスである [Toy93]．これは Strandh による forward-branching class(FB) と等しいことが Durand により示された [Dura94]．よって TS のクラスにおける性質が FB についても同様に言えることになる.

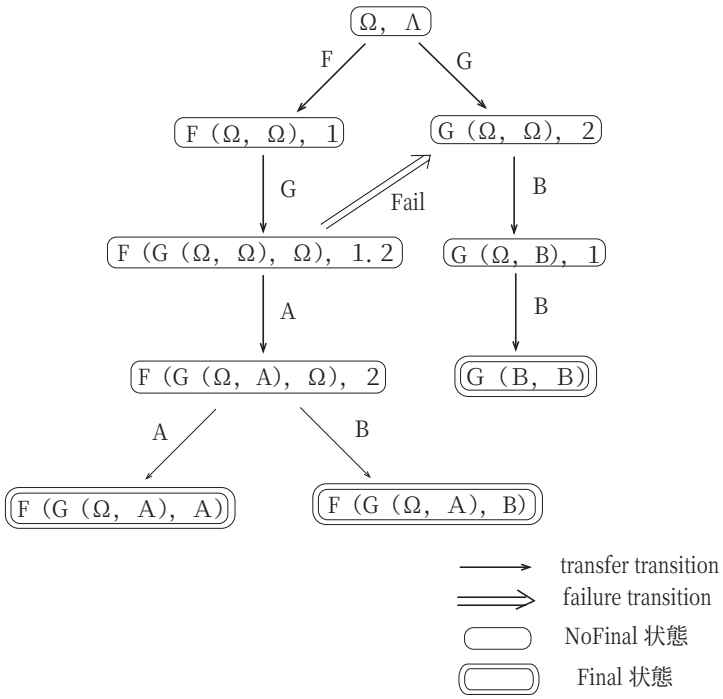

図 3.1: FB インデックスオートマトン A

定義 3.3.1 (Transitive index[Toy93])

- $q \in I(s)$ とする．任意の $p \in I(t)$ に対して $pq \in I(t[s]_p)$ ならば q は s の推移的インデックスと呼ばれる．このとき $q \in I_T(s)$ と書く．
- $t \xrightarrow[\mathcal{R},p]{} s$ とする．このとき，$p \in I_T(t)$ ならば $t \xrightarrow[\mathcal{R},p]{} s$ を推移的簡約と呼び $t \xrightarrow[T]{} s$ と表す．

定義 3.3.2

- もし，$p \in I_T(t[\Omega]_p)$ かつ $p \neq \Lambda$ ならば $t \rhd_T t/p$ と表す．
- $\rightarrow_T \cup \rhd_T$ は $\Rightarrow$ と表す．

補題 3.3.3 ([Sak97]) *transitive TRS R* において，もし，項 t が正規形を持つならば次のような無限の簡約列は存在しない．

$t \equiv t_0 \Rightarrow t_1 \Rightarrow t_2 \Rightarrow \cdots$ □

定義 3.3.4 [marked term]
TRS R について考える．

- 項 t の最外の記号を $head(t)$ と表記する．
- 関数記号の集合を $\mathcal{F}$ とする．マーク付き関数記号の集合 $\mathcal{F}^*$ は $\{f^* | f \in \mathcal{F}\}$ を表す．ただし，各 f^* は f と *arity* が同じであるような新しい関数記号である．マーク付き項 *(marked term)* の集合は $\mathcal{T}(\mathcal{F} \cup \mathcal{F}^*, \mathcal{V})$ と表すが，簡単に $\mathcal{T}^*$ と表す．

- 項 t をマーク付き項とする．t から全ての $mark$ を取り除いた項を $e(t)$ と表す．$\delta(t)$ は $\forall s \trianglelefteq t$, $head(s) \in \mathcal{F}$ であるような全ての部分項 s を Ω で置き換えたものである．また $t \equiv f(t_1, \cdots, t_n)\ (0 \leq n)$ であるとき $\bar{\delta}(t)$ は $f(\delta(t_1), \cdots, \delta(t_n))$ を表す．

定義 3.3.5 (strong HNF[Toy93]) もし，$\omega(t) \not\equiv \Omega$ であるとき項 t は 強頭正規形 *(strong head-normal form)* であるという．

補題 3.3.6 ([Toy93]) もし，項 t が *strong HNF* であるならば，t は *HNF* である． □

定義 3.3.7 [well marked]

$t \in \mathcal{T}^*$ が *well marked* であるとはつぎの条件を満たしたときである．

$\forall s \trianglelefteq t[head(s) \in \mathcal{F}^* \Rightarrow e(\delta(s)) \in NF_\Omega]$.

補題 3.3.8 ([Sak97]) $t \in \mathcal{T}^*$ が *well marked* であるとする．このとき，$head(t) \notin \mathcal{F}$ ならば $e(t)$ は *strong HNF* である． □

はじめに，2 つの手続き lookup と HNF を示す．FB オートマトンの状態 q，与えられた項 t とする．q_0 は FB オートマトンの初期状態を表す．また状態 q に関する情報として以下のような表記をする．

- 状態 $q = \langle t, p \rangle$ におけるインデックス
  $position(\langle t, p \rangle) = p$
- 状態 $q = \langle t, p \rangle$ から transfer transition で適用可能なシンボルの集合
  $nextsymbols(q) = \{f | \eta(q, f) \neq \emptyset\}$
- 状態 qから transfer transition でシンボル f を適用して遷移可能な状態
  $\eta(q, f)$
- 状態 qから failure transition で遷移可能な状態
  $\phi(q)$
- 状態 $q = \langle t, p \rangle$ が fainal state のとき，マッチするルール
  $left(\langle t, p \rangle) = t$

$Let\ t, s \in \mathcal{T}_\Omega, p \in \mathcal{O}(s);$
$proc.\ of\ lookup(t, \langle s, p \rangle)$
  $if\ s \in Q_f\ then\ return\ \langle suc, s \in Red_\Omega \rangle$
  $else\ /*\ no_final\ */$
    $if\ head(t|_p) \in nextsymbols(\langle s, p \rangle)\ then\ lookup(t, \eta(head(t|_p)))/*\ transfer\ */$
    $else\ /*\ fail\ */$
      $if\ \langle s, p \rangle = \langle \Omega, \Lambda \rangle\ then\ return\ \langle fail, (\langle \Omega, \Lambda \rangle, \langle \Omega, \Lambda \rangle)\rangle/*\ HNF\ */$
      $else\ /*\ failure\ */$
        $return\ \langle fail, \phi(\langle t, p \rangle)\rangle$
$endproc$

図 3.2: アルゴリズム *lookup*

補題 3.3.9 手続き *lookup* について以下のことが言える．

i). $if\ lookup(t) = \langle fail, (q, \langle s', p\rangle)\rangle\ , p \neq \Lambda then p \in I_T(t[\Omega]_p)$

ii). $if\ lookup(t) = \langle fail, (q, q_0)\rangle\ then$
$t\ is\ in\ strong\ HNF,$

iii). $if\ lookup(t) = \langle suc, s\rangle\ then s \leq t$

証明 *i)* 定義 *3.2.7*より明らかである．

ii) 補題 *3.3.8*より明らか．

iii) lookup の手続きより明らか． □

fail したとき，項 t を fail 先の状態 q からチェックすることと，初期状態 q_0 からチェックすることは同じであるから次のことが言える．

命題 3.3.10 $lookup(t, \langle\Omega, \Lambda\rangle) = \langle fail, (q, \langle s, p\rangle)\rangle$ であるとき，$lookup(t|_p, q) = lookup(t|_p, \langle\Omega, \Lambda\rangle)$ である．

証明 $q = q_0$ のときは問題ないことは明らかである．$q \neq q_0$ のときを考える．このとき q_0 から q に *transfer* のみで至るパス Φ が存在することになる．

$lookup(t, q_0)$ において *transfer* により遷移してきた順列を $(q_0, q_1, , \cdots, q', \cdots, q_n)$ とする．ある q_i において $\phi(q_i) = q_0$ であり，$\phi(q_{i+1}) = q_j, \phi(q_{i+2}) = q_{j+1}, \cdots$, というような同じような系列 $(q_0, q_j, q_{j+1}, \cdots)$ が存在する．$q_i = q'$ であるとすると $\phi(q_{i+k}) = q$ となるような k が存在し，$\Phi = (q_0, q_j, q_{j+1}, \cdots, q_{j+l}, q)$ となる．さらに，$\Phi = (q_0, q_j, q_{j+1}, \cdots, q) = (\phi(q_i), \phi(q_{i+1}), \cdots, \phi(q_{i+k}))$ となる．つまり，$lookup(t, q_0)$ が $< fail, (q, \langle s, p\rangle) >$ を返すまでに，すでに $(q_i, \cdots, q_{i+k-1})$ の過程で $t|_p$ の $(q_0, \cdots, q_{j+l})$ の過程もチェックしたことになる．

よって，$lookup(t|_p, q) = lookup(t|_p, q_0)$ である． □

命題 3.3.10と同じようなことを文献 [Dura94] では [Failure points Lemma] として示されている．

fail 先から戻って来た項 t' を元の項 t のポジション p に代入したとき，項 $t[t']_p$ について q' から再チェックすることと初期状態 q_0 から再チェックすることは同じであるから次のことが言える．

命題 3.3.11 $lookup(t, \langle\Omega, \Lambda\rangle) = \langle fail, (q, \langle s, p\rangle)\rangle$ であるとき，$\forall t', lookup(t[t']_p, \langle s, p\rangle) = lookup(t[t']_p, \langle\Omega, \Lambda\rangle)$ である．

証明 *q' = (M,p)* である．$lookup(t, q_0)$ において $\exists q_n, \phi(q_n) = q$ であるような遷移の系列 $(q_0, \cdots, q', \cdots, q_{n-1}, q_n)$ が存在することになる．つまり，$(q_0, \cdots, q', \cdots, q_n)$ までは *transfer* で遷移してきたことになる．項 t のポジション p にあたる部分項 $t|_p$ が変更されたのであるから，$(q_0, \cdots, q')$ までの *transfer* による遷移は同じである．よって，ポジション p をチェックする状態 *q'* から再びチェックすればよい．従って $lookup(t[t']_p, q') = lookup(t[t']_p, q_0)$ である． □

$proc.\ of\ HNF(t)$
$HNF(t) = HNF_rec(t, \langle \Omega, \Lambda \rangle)$
$where$
$HNF_rec(t, q) =$
(1) $if\ lookup(e(t), q) = \langle suc, l \rangle$
$then$
$let\ t \xrightarrow[l \to r \in \mathcal{R}]{} t'$
$return\ HNF_rec(t', \langle \Omega, \Lambda \rangle)$
(2) $if\ lookup(e(t), q) = \langle fail, (q_{fail}, \langle s, p \rangle) \rangle$
$then$
$(2-1)\ head(t|_p) \in \mathcal{F}^*\ or\ p = \Lambda\ goto\ (4)$
$(2-2)\ head(t|_p) \in \mathcal{F}\ and\ p \neq \Lambda\ return\ HNF_rec(t[HNF_rec(t|_p, q_{fail})]p, \langle s, p \rangle)$
(4) $if\ head(t) \in F\ then\ headmarked(t)$
$else\ return(t)$
$endproc$

図 3.3: アルゴリズム HNF

補題 3.3.12 $s \in Q$, $t \in T^*$ は *well marked* とする．もし，*e(t)* の正規形が存在するならば以下のことが言える．

i). $t \xrightarrow{*}_T HNF_rec(t, q)$,

ii). $head(HNF_rec(t, q)) \notin \mathcal{F}$, *and*

iii). $HNF_rec(t, q)$ *is well marked.*

証明 補題 *3.3.3*より $\Vdash$ は停止性を持つので，ネータ帰納法により証明する．

$\Vdash$ の基本経路として {*(1),(2-2),(2-1)-(3)*} の *3* つの経路が存在する．

- *(2-1)-(3)* の場合

  i),ii) は明らかである．t は *well marked* であるから $\forall s \trianglelefteq t[head(s) \in F^* \Rightarrow e(\delta(s)) \in NF_\Omega]$. $head(t|_p) \in F^*$ のとき $t = HNF(t)$ である．$p = \Lambda$ のとき 補題 *3.3.9ii)* より t は *strong HNF* を持つ．よって，$e(\delta(t)) \in NF_\Omega$ である．

  定義 *3.3.7*より *HNF(t)* は *well marked* である．

- *(1)* の場合

  リダクション $t \to t'$ は項のトップポジションでの書換えである．つまり，$t \to_T t'$．帰納法の仮定により $t \to_T t' \xrightarrow{*}_T HNF(t')$．また，$head(HNF(t')) \notin \mathcal{F}$ であるから，$HNF(t') \equiv HNF(t)$．よって，$t \to_T t' \xrightarrow{*}_T HNF(t') \equiv HNF(t)$. つまり，*i)-iii)* は正しい．

- *(2-2)* の場合

  はじめに $t1 = HNF_rec(t|_p, q)$ が *well marked* であることを示す．*(2-2)* の過程を経た *lookup* の手続きにおいて $lookup(e(t|_p), q) = lookup(e(t|_p), q_0)$ であることは命題 *3.3.10*から明らかである．よって $HNF_rec(t|_p, q) = HNF_rec(t|_p, q_0)$ である．ここで帰納法の仮定が使えるから *t1* は *i)-iii)* を満たす．

つぎに，$t2 = HNF_rec(t[t1]_p, q')$ が *well marked* であることを示す．ここで，$lookup(e(t[t1]_p), q') = lookup(e(t[t1]_p), q_0)$ であることは命題 *3.3.11*から明らかである．よって $HNF_rec(t[t1]_p, q') = HNF_rec(t[t1]_p, q_0)$ である．帰納法の仮定より，*t2* は *i)-iii)* を満たす．

以上のことから t に関して *i)-iii)* がいえる．

補題 3.3.13 *Let* $t \in T^*$ は *well marked* である．もし，*e(t)* の正規形が存在するならば以下のことが言える．

i). $t \xrightarrow{*}_T$ *HNF(t),*

ii). $head(HNF(t)) \notin F$, *and*

iii). HNF(t) is well marked.

証明 *HNF* の手続きより $HNF(t) = HNF_rec(t, q_0)$ である．よって，補題 *3.3.12*より明らか． □

3.4 NF を求めるアルゴリズム

図 3.4に示す NFの手続きにより項 t の正規形 $NF(t)$ を求めることが出来る．

$proc\ of\ NF(t)$
　$t' = NF(t)$
　$if\ (f(t_1, \cdots, t_m) = t')\ then\ return\ f(NF(t_1), \cdots, NF(t_m))$
　$elsereturn\ t$
$endproc$

図 3.4: アルゴリズム NF

第 4 章

TRS コンパイラの実装

TRS の各関数を C 言語の関数に翻訳，各 C 関数に lookup や HNF のアルゴリズムを埋め込んでいる．

4.1 変換プロセス

書換え規則の左辺の最外の位置に出現する関数記号 fについて 1 つの C 言語関数を生成する．項書換え系 $\mathcal{R}$ の規則の内で，左辺の最も外側の関数記号が fである規則の集合を $\mathcal{R}(f)$ とする．すなわち $\mathcal{R}(f) = \{l | l \to r \in \mathcal{R}, head(l) = f\}$ である．コンパイラは n 引数の関数記号 fに対し，次の形の定義文を生成する．

c_f$(s, a_1, \cdots, a_n)$
TERM $a_1, \cdots, a_n$;
State s;
{
　　＜各状態への分岐＞　　＜関数の本体＞
}

“c_” は関数記号 fの C 言語関数を表す．第 1 引数の s は FB オートマトンのどの状態で評価を行うのかの通し番号である．$a_1, \cdots, a_n$は関数記号 fの引数である．

＜各状態への分岐＞
{
　　goto s_i;
}

＜関数の本体＞は次の＜実行部＞と＜条件部＞で構成される．

＜実行部＞
{
　s_i:
　　rewrite;
}

＜条件部＞
{
　　s_i:
　　switch(head(a_i))
　　case s_1: { ＜実行部＞ | ＜条件部＞ }
　　case s_2: { ＜実行部＞ | ＜条件部＞ }
　　⋮
　　case s_m: { ＜実行部＞ | ＜条件部＞ }
　　other: { ＜実行部＞ }
}

＜実行部＞では *rewirte* において規則を適用した際の書換え処理を行う．＜条件部＞では項と規則の左辺とのマッチングを行う．a_iは関数 f の i 番目の引数である．$head(t)$ は項 t の最外の関数記号を表す．$s_1, s_2, \cdots, s_m$ は m 個の関数記号を意味し，$head(a_i) = s_1$のときは case s_1 : の後の文が実行される．各 case 文の後の { ＜実行部＞ | ＜条件部＞ } では，更にマッチング処理が必要なときは＜条件部＞となり，規則の適用が可能なときは＜実行部＞となる．

関数記号 f を最外に持つ規則が 1 つしかなく，fの引数が 0，または引数全てが変数であるときは＜実行部＞のみで構成される．それ以外の場合は，＜条件部＞と＜実行部＞の組合せで構成される．

図 4.1はコンパイラのアルゴリズムである．

```
gen_function(SS)
{
    State_1 = {s|s ∈ SS, level(s) = 1};
    for eachs ∈ State_1
        begin
            gen_function_head(f,s);
            gen_body(f,s);
            gen_function_tail(f,s);
        end
}
gen_body(f,s)
{
    if ( no_final(s) ) then
        gen_switch_case(s);
        for eachf ∈ nextsymbol(s)
            begin
                print_switch_case(f);
                for eachs ∈ Next(s, f)
                    begin
                        gen_body(f,s);
                    end
            end
    else
        gen_rewrite(s);
    endif
}
```

図 4.1: コンパイラのアルゴリズム

4.2 効率的実現の詳細

ここでは効率的実現のために実際に行っている実装方法ついていくつか説明する.

4.2.1 セル構造

項は図 4.2に示したような構造のセルを用いて木構造のように構成されている.

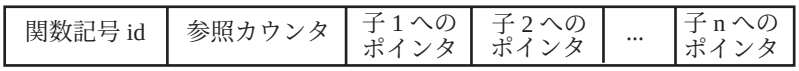

図 4.2: セルの構造

セルは項を木と見なしたとき，1 つのノードを表す．そのため，セルは関数記号の情報や子供セルの情報，後で説明する参照カウンタの情報を持つ．セルを用いることにより，セルへのポインタを使って項を表すことができる．書換えにおいても図 4.3に示すようにポインタの付け替えで済む.

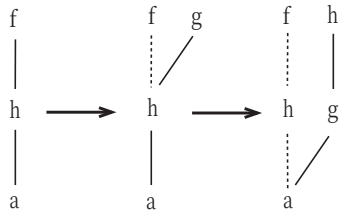

図 4.3: セル構造での書換え

4.2.2 項の共有

書換えを行う際，同じ項を 2 度以上コピーするにはコストがかかる．そこで規則の右辺に同じ変数が 2 度以上出現するときは図 4.4に示すように同じ cell へのポインタを共有するようにしている．

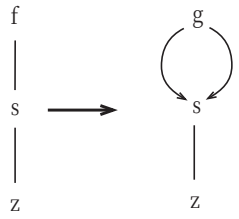

図 4.4: セルの共有

このため使用されるメモリ領域の節約にも役立っている．

このセルの共有が効果を発揮するのは，共有しているセルを頭にした項自身が書換えられたときである．一度計算した場所を再び計算する必要がなくなる．この効果により，計算速度の高速化が図られる．しかし，セル内の情報の更新が新たなオーバヘッドとなる．

4.2.3 参照カウンタ

項 t 内で参照されているセル c が項 s でも参照されていて，つまり共有されていたとする．通常，項 t を書換える際，セル c が不要なゴミになったとするとセル c は回収される．しかし，セル c は項 s でも参照されているためセル c が回収されると不都合が生じる (図 4.5参照)．そこで，共有されているセルには参照している親の数に関する情報が必要である．参照カウンタ (reference counter) はそのセルを参照している親の数を表す (図 4.6参照)．

この参照カウンタを導入することにより，項 t を書換えてもセル c は参照カウンタを -1 すれば良く，項 s はセル c を参照したままに出来る．参照カウンタはセルの情報を見るだけでよく，不要なセルは即時に回収される．

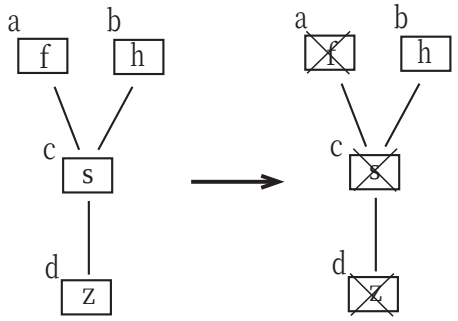

図 4.5: 共有セルの誤った回収

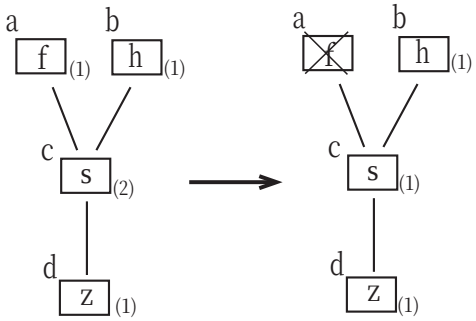

図 4.6: 参照カウンタ

第 5 章

評価

5.1 評価

本研究で作成したコンパイラが生成するプログラムと Cdimple[Sak87]，Elan compiler[KKV95, Vit96]，Standard ML of New-Jersey compiler[AM91]，Gofer compiler[Jon94] が生成するプログラムの実行速度の比較を行った．本コンパイラ，Cdimple，Elan compiler，Gofer compiler は C 言語プログラムを最初に生成し，C コンパイラを用いて実行プログラムを生成する．デフォルトの C コンパイラとして本コンパイラや Cdimple は cc を用い，Elan や Gofer は gcc を用いている．評価条件を出来るだけ同じにするために C コンパイラは UNIX 標準の cc を用いた．オプション引数には "$-O$" を付加している．これは cc と gcc では同じ C プログラムをコンパイルしても明らかな速度差が出るからである．SML-NJ については実行プログラムへの変換に変更を加えることが困難であったためデフォルトのままで評価している．

評価に用いた問題は **fact 8**，**naive reverse 1000**, **fib 20**，**quick sort 30/1000**，**test1** である．**fact 8** は 8!を求める問題である．**fib 20** は $s(s(\cdots s(z)\cdots))$ で表現された数のフィボナッチ数を求める問題である．**naive reverse 1000** は 1000 個の要素を並べかえる問題である．**quick sort 30/1000** は 1 種類の乱数系列の 30 個の要素，1000 個の要素に対するクイックソートの問題である．**fact** や **fib** 上の数は $s(s(\cdots s(z)\cdots))$ の形で表現されている．また，**naive reverse** や **quick sort** 上の要素は $Cons(123,(Cons(456,(\cdots))))$ の形で表現されている．評価に用いた問題のプログラムは付録にある．表 5.1は SS5 (SunOS 4.1.3) 上での実行結果である．

表 5.1: 実行速度の比較

	Our Compiler	Cdimple	Elan	SML-NJ	Gofer
fact 8	586ms	456ms	74ms	290ms	684ms
fib 20	509ms	478ms	113ms	120ms	722ms
naive reverse 1000	7712ms	9760ms	566ms	1230ms	[a] 9132ms
quick sort 20	[d] 0ms	14086ms	2912ms	1850ms	2 ms
quick sort 30	16ms	[b] —	2886250ms	1852450ms	6 ms
quick sort 1000	543ms	[b] —	[e]△	[e] △	[a] 518ms
fbexamp	[d] 0ms	[c] ∞	[c] ∞	[c] ∞	[c] ∞

[a] C コンパイラとして cc ではなく gcc を用いている.
[b] メモリに関するエラーが発生する.
[c] 停止しない
[d] 1ms 未満
[e] 停止するが簡約回数の増加により未だ解が求まっていない.

quick sort の問題はプログラムの中で if 関数を使用する．通常の関数型言語システムでは if は組み込み関数としてシステムに実装されていることが多い．項書換えシステムとして考えた場合，この if 関数だけが特別な振るまいをするのはおかしいと考えプログラム中で新たに if 関数を定義している．この新たに定義している if 関数のため，最左最内で実行するシステムでは要素の数が増えるに従って書換え回数が大きくなり解が求まるまでに時間がかかるようになる．山本ら [Yam87] は評価に用いた問題と同じプログラムに対し，最左最外，最左最内では解を求めるためにある乱数系列では以下のような書換え回数が必要であることを示している．

$$\text{最左最外}\begin{cases} n=2k & (7n*n+16n-20)/4 \\ n=2k+1 & (7n*n+16n-19)/4 \end{cases}$$

$$\text{最左最内}\begin{cases} n=2k & 4*2^n+(7n*n+16n-36)/4 \\ n=2k+1 & 4*2^n+(7n*n+16n-35)/4 \end{cases}$$

この書換え回数の差異により Elan や SML-NJ で生成したプログラムは **quick sort** の問題に関しては他のコンパイラのものより遅い結果となっている.

fact，**fib**，**naive reverse** の問題では本コンパイラで生成されたプログラムの実行速度は他のコンパイラで作成されたプログラムの速度に比べ遅い．この理由には次の点が考えられる.

- 書換え戦略が最外戦略
- セルへの参照の増減のたびにカウンタの更新が必要

本コンパイラはリデックスを探索するために項の外側からインデックスを調べる．このため，戦略として

は最外戦略を基本としている．しかし，他のコンパイラは最内戦略で書換えていく．このため，最内戦略に基づくアルゴリズムと比較すると最外戦略に基づくアルゴリズムの実行速度は遅くなると考えられる．

参照カウンタを導入したことにより，ゴミになったかどうかの判定がセルの情報だけで決定出来ることになった．また，ゴミなったセルは即時に回収することができる．しかし，これらのためにセルへの参照が増減するたびにカウンタを更新しなければならない．参照カウンタの増減によるオーバヘッドは大きいと考える．

例 5.1.1 [**fbexamp**]

$$\mathcal{R} = \left\{ \begin{array}{rcl} f(a,b) & \rightarrow & a \\ f(x,c) & \rightarrow & b \\ g(c) & \rightarrow & h(c) \\ h(c) & \rightarrow & g(c) \end{array} \right.$$

この *TRS R* において *f(g(c),c)* の正規形を求めようとすると，次のような簡約列が考えられる．

1. $f(\underline{g(c)}, c) \rightarrow f(\underline{h(c)}, c) \rightarrow f(\underline{g(c)}, c) \rightarrow \cdots$

2. $\underline{f(g(c), c)} \rightarrow b$

1. は最左最内戦略で書換えるときに得られる簡約列である．*Elan* や *Sml-NJ* などのコンパイラで生成されたプログラムで実行するとこのようになり，無限に書換えられ停止しない．*2.* は本コンパイラにより得られるプログラムの簡約列である．項 *f(g(c),c)* のインデックスは出現位置 *2(*項を左から見て *2* つ目の c の位置*)* であるから *2* 番目の書換え規則が適用され，正規形 b が求まる．

このように他のコンパイラでは正規形が存在しても停止せずに求まらないが，本コンパイラでは正規形が求まる例が存在する．

第 6 章

まとめ

本コンパイラを構築したことにより，インデックスを効率的に探索し，正規化戦略を実現したプログラムが作成可能となった．実行速度による評価結果からもわかるように本コンパイラにより生成されたプログラムは Gofer のものより若干高速である．FB なクラスにおいては正規化戦略となるため，答えがあれば必ず求められるという特徴を持つ．

しかし，最内戦略を書換え戦略の基本にしている他のコンパイラに比べ遅い．この原因には関数呼び出しによるオーバヘッドの蓄積，やゴミ集めの際に発生するオーバヘッドが考えられる．今後は関数呼び出しの際のオーバヘッドの減少やゴミ集めの方法の変更を検討していく．

謝辞

本研究にあたって，多大な御指導を承った酒井正彦助教授，外山芳人教授に謹んで感謝致します．また，熱心に御討論頂いた研究室の皆様に感謝します．

参考文献

[AM91] Andrew W. Appel, David B. MacQueen, *Standard ML of New Jersey*, CS-TR-329-91, Dept. of Computer Science, Princeton University, June 1991.

[Com95] Hubert Comon, *Sequentiality, Second Order Monadic Logic and Tree Automata*, Proc. 10th IEEE Symp. Logic in Computer Science, pp.508-517, 1995.

[Dura94] I. Durand, *Bounded, Strongly Sequential and Forward-Branching Term Rewriting Systems*, Journal of Symbolic Computation, 18, pp.319-352, 1994.

[HL91] G. Huet, J.-J. Lévy, *Computations in Orthogonal Rewriting Systems, I,II*, In J.-L.Lassez, G.Plotkin, "Computational Logic", pp.397-443, MIT press, 1991.

[Jon94] Mark P. Jones, *The implementation of the Gofer functional programming system*, Research Report YALEU/DCS/RR-1030, Yale University, 1994.

[KKV95] C. Kirchner, H. Kirchner, M. Vittek, *Designing Constraint Logic Programming Languages using Computational Systems*, In P. Van Hentenryck and V. Saraswat, editors, Principles and Practice of Constraint Programming, MIT press, pp.131-158, 1995.

[KM91] J. W. Klop, A. Middeldorp, *Sequentiality in Orthogonal Term Rewriting Systems*, Journal of Symbolic Computation, 12, pp.161-195, 1991.

[Sak97] Masahiko Sakai, *Left-incompatible Term Rewriting Systems and Functional Strategy*, unpublished, 1997.

[SS90] D. J. Sherman, R. I. Strandh, *An Abstract Machine for Efficient Implementation of Term Rewriting*, Technical Report 90-012, University of Chicago, 1990.

[Str89] R. I. Strandh, *Classes of equational programs that compile into efficient machine code*, LNCS, 355, pp.449-461, 1989.

[Toy93] Y. Toyama, S. Smetsers, M. van Eekelen, R. Plasmeijer, *The functional strategy and transitive term rewriting systems*, In M.R.Sleep, M. van Eekelen, "Term Graph Rewriting", pp.61-75, Wiley Professional Computing, 1993.

[Vit96] Marian Vittek, *A Compiler for Nondeterministic Term Rewriting Systems*, RTA96, LNCS 1103, pp.154-168, 1996.

[Sak87] 酒井，坂部，稲垣，抽象データ型の代数的仕様の直接実現系 *Cdimple*, コンピュータソフトウェア, Vol.4, No.4, pp.16-27, 1987.

[Sak95] 酒井正彦，関数型戦略 - 左非整合な項書換え系の拡張 -, 信学技報，Vol.95, No.144, pp.55-62，1995.

[Yam87] 山本，直井，板部，稲垣，項書換えシステムにおける必須な書換え戦略の効率信学会，COMP87-37, pp.21-30, 1987.

付録 A

評価プログラム

A.1 本コンパイラ/Cdimple コンパイラ用評価プログラム

本コンパイラは Cdimple の改良を試みたものであるから共に読み込むプログラムの形式は同じである.

プログラム 1 — fact8.cd —

```
%%
Z    :          -> int;
S    : int      -> int;
Sum  : int,int  -> int;
Mult : int,int  -> int;
Fact : int      -> int;
%%
Sum(Z(),y)==y;
Sum(S(x),y)==S(Sum(x,y));
Mult(Z(),x)==Z();
Mult(S(x),y)==Sum(y,Mult(x,y));
Fact(Z())==S(Z());
Fact(S(x))==Mult(S(x),Fact(x));
%%
```

プログラム 2 — fib.cd —

```
%%
Z    :          -> int;
S    : int      -> int;
P    : int,int  -> int;
Fib  : int      -> int;
%%
Fib(Z())         == Z();
Fib(S(Z()))      == S(Z());
Fib(S(S(x)))     == P(Fib(x),Fib(S(x)));
P(Z(),x)         == x;
P(S(x),y)        == S(P(x,y));
%%
```

プログラム 3 — qsort_ifqs.cd —

このプログラムでは組み込み関数の *IF* と *GE_INT* を使用している．これは *GE_INT* の返す値が *'0'* もしくは *'1'* であり，*'true'*，*'false'* を返さないためである．ユーザ定義の関数において *'true'*，*'false'* として認識できるように追加した．

```
GE_INT: int,int -> bool;
%%
T: -> bool;
F: -> bool;
NIL: -> list;
Cons: int,list -> list;
Append: list,list -> list;
Pair: list,list -> pairlist;
Qs: list -> list;
Qs1: int,pairlist -> list;
PartI: int,list -> pairlist;
PartI1: int,list,list,list -> pairlist;
Ifqs: bool,pairlist,pairlist -> pairlist;
Geqs: int,int -> bool;
%%
Qs(NIL())                          == NIL() ;
Qs(Cons(p, NIL()))                 == Cons(p, NIL()) ;
Qs(Cons(p, Cons(x, l)))            == Qs1(p, PartI(p, Cons(x, l))) ;
Qs1(p, Pair(lpart, gpart))         == Append(Qs(lpart), Cons(p, Qs(gpart))) ;
PartI(p, l)                        == PartI1(p, l, NIL(), NIL()) ;
PartI1(p, Cons(x, l), lpart, gpart) ==
                                   Ifqs(Geqs(p, x),
                                         PartI1(p, l, Cons(x, lpart), gpart),
                                         PartI1(p, l, lpart, Cons(x, gpart))) ;
PartI1(p, NIL(), lpart, gpart)     == Pair(lpart, gpart) ;
Append(NIL(), l)                   == l ;
Append(Cons(x, l1), l2)            == Cons(x, Append(l1, l2)) ;
Ifqs(T(),x,y)                      == x;
Ifqs(F(),x,y)                      == y;
Geqs(x,y)                          == IF(GE_INT(x,y),T(),F());
%%
```

プログラム 4 — rev.cd —

```
%%
NIL   :           -> list;
Cons  : int,list  -> list;
Rev   : list      -> list;
App   : list,list -> list;
%%
Rev(NIL())      == NIL();
Rev(Cons(x,xs)) == App(Rev(xs),Cons(x,NIL()));
App(NIL(),xs)   == xs;
App(Cons(a,as),xs) == Cons(a,App(as,xs));
%%
```

プログラム 5 — 各プログラムに追加した C 関数 — 時間計測の方法等を含む．

```
#include <sys/time.h>
#include <sys/times.h>
#include <sys/resource.h>

struct rusage ru_start, ru_end, ru_diff;

#define cui_time_s() getrusage(RUSAGE_SELF,&ru_start)
```

```
#define cui_time_e() getrusage(RUSAGE_SELF,&ru_end)

void print_time_diff()
{
    ru_diff.ru_utime.tv_sec =
      ru_end.ru_utime.tv_sec -
        ru_start.ru_utime.tv_sec;
    ru_diff.ru_utime.tv_usec =
      ru_end.ru_utime.tv_usec -
        ru_start.ru_utime.tv_usec;
    ru_diff.ru_stime.tv_sec =
      ru_end.ru_stime.tv_sec -
        ru_start.ru_stime.tv_sec;
    ru_diff.ru_stime.tv_usec =
      ru_end.ru_stime.tv_usec -
        ru_start.ru_stime.tv_usec;

    printf(" ( user time = %.0lf msec  system time = %.0lf msec ) \n",
           (ru_diff.ru_utime.tv_sec*1e6+ru_diff.ru_utime.tv_usec)/1e3,
           (ru_diff.ru_stime.tv_sec*1e6+ru_diff.ru_stime.tv_usec)/1e3);
}

extern int errorcnt;

void main(argc,argv)
int argc;
char *argv[];
{
        _TERM tm1,tm2, interm, outterm, getterm(),eval();
        tm1=tm2=(_TERM)_NULL;
        puts("Now go !");
        interm = getterm(symdom,basedom);
        if(errorcnt)exit(1);
        putterm(interm,symdom);
        putchar('\n');
        cui_time_s();
        outterm=eval(interm,symdom);
        cui_time_e();
        putterm(outterm,symdom);
        putchar('\n');
        print_time_diff();
}
```

A.2 Elan コンパイラ用評価プログラム

Elan では時間計測のために Cdimple のところで示したプログラム 5 と同様の関数を使用した．また，Elan で実行するためには lgi ファイルも必要であるが簡単なものなので今回は特に示さないことにする．

プログラム 6 — fact8.eln —

```
module fact8
sort nat;
op
  global
   z            : nat;
   s(@@)         : (nat) nat;
   p(@@,@@)       : (nat nat) nat;
   m(@@,@@)       : (nat nat) nat;
```

```
   fact(@@)      : ( nat ) nat;

   fact8         : nat;
endop

rules for nat
declare x,y : nat;
bodies

  [] p(z,x)     => x                    end
  [] p(s(x),y)  =>s(p(x,y))             end

  [] m(z,x)     => z                    end
  [] m(s(x),y)  => p(y,m(x,y))          end

  [] fact(z)    => s(z)                 end
  [] fact(s(x)) => m(s(x),fact(x))      end

  [] fact8 => fact(s(s(s(s(s(s(s(s(z))))))))) end

end of rules
end of module
```

プログラム 7 — fib20.eln —

```
module fib20
sort nat;
op
  global
   z            : nat;
   s(@)         : (nat) nat;
   pl(@,@)      : (nat nat) nat;
   fib(@)       : ( nat ) nat;

   fib20        : nat;
endop

rules for nat
declare x,y : nat;
bodies

  [] pl(z,x)    => x                    end
  [] pl(s(x),y) =>s(pl(x,y))            end

  [] fib(z)     => z                    end
  [] fib(s(z))  => s(z)                 end
  [] fib(s(s(x)))       => pl(fib(x),fib(s(x))) end

  [] fib20      =>
        fib(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(z)))))))))))))))))) end
end of rules
end of module
```

プログラム 8 — qsort20.eln —

```
module qsort20
import bool int;
sort list pairlist;
op
  global
```

```
        NIL : list;
        Cons(@,@) : (int list) list;
        Append(@,@) : (list list) list;
        Qs(@) : (list) list;
        Qs1(@,@) : (int pairlist) list;

        IF(@,@,@) : (bool pairlist pairlist) pairlist;
        Pair(@,@) : (list list) pairlist;
        PartI(@,@) : (int list) pairlist;
        PartI1(@,@,@,@) : (int list list list) pairlist;

        qs20 : list;
endop

rules for pairlist
declare p,x : int;
        ls1,ls2 : pairlist;
        l,lpart,gpart : list;
bodies
  [] IF(true,ls1,ls2)   => ls1 end
  [] IF(false,ls1,ls2)  => ls2 end
  [] PartI(p, l)        =>  PartI1(p, l, NIL, NIL) end
  [] PartI1(p, Cons(x, l), lpart, gpart) =>
        IF(p >= x,
                PartI1(p, l, Cons(x, lpart), gpart),
                PartI1(p, l, lpart, Cons(x, gpart))) end
  [] PartI1(p, NIL, lpart, gpart) =>  Pair(lpart, gpart) end
end of rules

rules for list
declare p,x : int;
        l,lpart,gpart,l1,l2 : list;
bodies
  [] Qs(NIL)                    => NIL end
  [] Qs(Cons(p, NIL))           =>  Cons(p, NIL) end
  [] Qs(Cons(p, Cons(x, l)))    =>  Qs1(p, PartI(p, Cons(x, l))) end
  [] Qs1(p, Pair(lpart, gpart)) =>  Append(Qs(lpart), Cons(p, Qs(gpart))) end
  [] Append(NIL, l)             =>  l end
  [] Append(Cons(x, l1), l2)    =>  Cons(x, Append(l1, l2)) end

  [] qs20 => Qs(
        Cons(435,Cons(413,Cons(572,Cons(827,Cons(228,
        Cons(585,Cons(505,Cons(568,Cons(128,Cons(756,
        Cons(900,Cons(477,Cons(657,Cons(317,Cons(792,
        Cons(673,Cons(977,Cons(390,Cons(942,Cons(659,
        NIL))))))))))))))))))))) end

end of rules
end of module
```

プログラム 9 — rev1000.eln —

```
module rev1000
import int;
sort list;
op
  global
   Nil          : list;
   Cons(@,@)    : (int list) list;
   Append(@,@)  : (list list) list;
   Rev(@)       : (list) list;
```

```
   rev1000      : list;
endop

rules for list
declare x : int;
        xs,ys : list;
bodies

  [] Append(Nil,xs)             => xs                    end
  [] Append(Cons(x,xs),ys)      => Cons(x,Append(xs,ys)) end

  [] Rev(Nil)           => Nil                           end
  [] Rev(Cons(x,xs))    => Append(Rev(xs),Cons(x,Nil))   end

  [] rev1000 => Rev( /* ここに 1000 個の要素が入る */
        Cons(435,Cons(413,Cons(572,Cons(827,Cons(228,
        Cons(585,Cons(505,Cons(568,Cons(128,Cons(756,
        Cons(900,Cons(477,Cons(657,Cons(317,Cons(792,
        Cons(673,Cons(977,Cons(390,Cons(942,Cons(659,
        Cons(679,Cons(196,Cons(120,Cons(667,Cons(803,
        /* 長いので省略 */
        Cons(642,Cons(35,Cons(872,Cons(388,Cons(27,
        Cons(547,Cons(183,Cons(298,Cons(153,Cons(760,
        Nil
        ))))))))))
        /* 長いので省略 */
        ))))))))))))))))))))))))
        ) end

end of rules
end of module
```

A.3 SML-NJ コンパイラ用評価プログラム

Sml-NJ では実行プログラムを生成するまでの過程が複雑であるため，時間計測のプログラムを付加する適当な場所が見つからなかった．そこで，Sml-NJ では時間を計測するために，時間計測用の関数を Sml のプログラム内に記述している．

プログラム 10 — fact8.sml —

```
          datatype Nat = Z | S of Nat

fun utime ()
    = let
          val (System.Timer.TIME now) = #usr(System.Unsafe.CInterface.gettime ())
      in
          (#sec now)*1000 + (#usec now) div 1000
      end

fun time f args
    = let
          val starttime = utime ()
          val result = f args
      in
          print "\n";
          print (utime() - starttime);
          print "msec.\n";
```

```
            result
        end

    fun sum(Z,y) = y
      | sum(S(x),y) = S(sum(x,y));

    fun mult(Z,x) = Z
      | mult(S(x),y) = sum(y,mult(x,y));

    fun fact(Z) = S(Z)
      | fact(S(x)) = mult(S(x),fact(x));

    fun fib(Z) = Z
      | fib(S(Z)) = S(Z)
      | fib(S(S(x))) = sum(fib(x),fib(S(x)));

    fun fact8_test() = (
             time fact(S(S(S(S(S(S(S(Z)))))))));
             print "\n"
            );

    fun mainprog(x:string list, y:string list)= fact8_test();
    exportFn("fact8",mainprog);
```

プログラム 11 — fib20.sml —

```
            datatype Nat = Z | S of Nat

    fun utime ()
        = let
              val (System.Timer.TIME now) = #usr(System.Unsafe.CInterface.gettime ())
          in
              (#sec now)*1000 + (#usec now) div 1000
          end

    fun time f args
        = let
              val starttime = utime ()
              val result = f args
          in
              print "\n";
              print (utime() - starttime);
              print "msec.\n";
              result
          end

    fun sum(Z,y) = y
      | sum(S(x),y) = S(sum(x,y));

    fun mult(Z,x) = Z
      | mult(S(x),y) = sum(y,mult(x,y));

    fun fact(Z) = S(Z)
      | fact(S(x)) = mult(S(x),fact(x));

    fun fib(Z) = Z
      | fib(S(Z)) = S(Z)
      | fib(S(S(x))) = sum(fib(x),fib(S(x)));

    fun fib20_test() = (
            time fib(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(Z)))))))))))))))))));
            print "\n"
```

```
        );
fun mainprog(x:string list, y:string list)= fib20_test();
exportFn("fib20",mainprog);
```

プログラム 12 — qsort20.sml —

```
signature SIGPROB =
    sig
        type List
        val append : List * List -> List
        val qs : List -> List
        val qs1 : int -> List * List -> List
        val parti : int -> List -> List * List
        val parti1 : int -> List -> List -> List -> List * List
    end;

structure PROB =
    struct
        datatype List = Nil | Cons of int * List

fun utime ()
    = let
          val (System.Timer.TIME now) = #usr(System.Unsafe.CInterface.gettime ())
      in
          (#sec now)*1000 + (#usec now) div 1000
      end

fun time f args
    = let
          val starttime = utime ()
          val result = f args
      in
          print "\n";
          print (utime() - starttime);
          print "msec.\n";
          result
      end

fun append(Nil,xs) = xs
  | append(Cons(b,bs),xs) = Cons(b,append(bs,xs));

fun ifqs true y1 y2 = y1
  | ifqs false y1 y2 = y2

fun parti1 p (Cons(x,l)) lp gp = ifqs (p >= x)
                                 (parti1 p l (Cons(x,lp)) gp)
                                 (parti1 p l lp (Cons(x,gp)))
  | parti1 p Nil lp gp = (lp,gp)

fun parti p l = parti1 p l Nil Nil;

fun qs (Nil) = Nil
  | qs (Cons(x,Nil)) = Cons(x,Nil)
  | qs (Cons(x,xs)) =
        let
                fun qs1 p (lp,gp) = append((qs lp),Cons(p,(qs gp)))
        in
                qs1 x (parti x xs)
        end
end;

open PROB;
```

```
(* to use Makestring.intToStr *)
use "makestring-sig.sml";
use "makestring.sml";

val list20 =
        Cons(435,Cons(413,Cons(572,Cons(827,Cons(228,
        Cons(585,Cons(505,Cons(568,Cons(128,Cons(756,
        Cons(900,Cons(477,Cons(657,Cons(317,Cons(792,
        Cons(673,Cons(977,Cons(390,Cons(942,Cons(659,
        Nil))))))))))))))))))))

fun qsort20() = time qs list20

fun printZ(Nil) = print "qsort20 end\n"
  | printZ(Cons(x,xs)) = printZ(xs);

fun mainprog(x:string list, y:string list) = printZ(qsort20());
exportFn("qsort20",mainprog);
```

プログラム 13 — rev1000.sml —

```
signature SIGPROB =
    sig
        type 'a List
        val Nil : 'a List
        val Cons : 'a * 'a List -> 'a List
        val append : 'a List * 'a List -> 'a List
        val rev0 : 'a List -> 'a List
        val printZ : 'a List * string -> string
    end;
structure PROB =
    struct
        datatype 'a List = Nil | Cons of 'a * 'a List

fun utime ()
    = let
          val (System.Timer.TIME now) = #usr(System.Unsafe.CInterface.gettime ())
      in
          (#sec now)*1000 + (#usec now) div 1000
      end

fun time f args
    = let
          val starttime = utime ()
          val result = f args
      in
          print "\n";
          print (utime() - starttime);
          print "msec.\n";
          result
      end

fun append(Nil,xs) = xs
  | append(Cons(b,bs),xs) = Cons(b,append(bs,xs));

fun rev0(Nil)= Nil
  | rev0(Cons(x,xs))= append(rev0(xs),Cons(x,Nil));

end;

open PROB;
(* to use Makestring.intToStr *)
```

```
use "makestring-sig.sml";
use "makestring.sml";

fun rev1000() = time rev0(
        Cons(435,Cons(413,Cons(572,Cons(827,Cons(228,
        Cons(585,Cons(505,Cons(568,Cons(128,Cons(756,
        Cons(900,Cons(477,Cons(657,Cons(317,Cons(792,
        Cons(673,Cons(977,Cons(390,Cons(942,Cons(659,
        (* 長いので省略 *)
        Nil
        (* 長いので省略 *)
        ))))))))))
        ))))))))))
        );

fun printZ(Nil) = print "rev1000 end\n"
  | printZ(Cons(x,xs)) = printZ(xs);

fun mainprog(x:string list, y:string list) = printZ(rev1000());
exportFn("rev1000",mainprog);
```

A.4 Gofer コンパイラ用評価プログラム

Gofer ではここに示した評価プログラム以外に, 時間計測のために Gofer のソースファイルの中の runtime.c を変更している．また，各プログラムでは正規形を求めるために本質的ではない余分な関数が定義されいる．しかし，総合的な実行時間に比べれば余分な関数の実行時間は十分無視できるぐらい小さいので，評価ではこれらの余分な関数の時間も含めている．

プログラム 14 — fact8.gs —

```
main ~(Success:_) = [AppendChan "stdout" printZFact8]

printZ Z z = z
printZ (S xs) z = printZ xs z

data Nat = Z | S Nat
p Z x = x
p (S x) y = S (p x y)
m Z x = Z
m (S x) y = p y (m x y)
fact Z = S Z
fact (S x) = m (S x) (fact x)
----
fact8 = fact (S (S (S (S (S (S (S (S Z))))))))

printZFact8 = (printZ fact8 "fact8")
```

プログラム 15 — fib20.gs —

```
main ~(Success:_) = [AppendChan "stdout" printZFib20]

printZ Z z = z
printZ (S xs) z = printZ xs z

data Nat = Z | S Nat
```

```
fib Z = Z
fib (S Z) = S Z
fib (S (S x)) = p (fib x) (fib (S x))
p Z x = x
p (S x) y = S (p x y)
fib20 = fib (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S
        Z))))))))))))))))))))

printZFib20 = (printZ fib20 "fib20")
```

プログラム 16 — qsort20.gs —

```
main ~(Success:_) = [AppendChan "stdout" printQsort20]

printNil Nil z = z
printNil (Cons x xs) z = printNil xs z

printQsort20 = printNil qs20 "qsort20 end"

data List = Nil | Cons Int List
qs Nil = Nil
qs (Cons x Nil) = (Cons x Nil)
qs (Cons x xs) = qs1 x (parti x xs)
qs1 p (lp,gp) = app (qs lp) (Cons p (qs gp))
parti p l = parti1 p l Nil Nil
parti1 p (Cons x l) lp gp =
        ifqs (p>=x) (parti1 p l (Cons x lp) gp) (parti1 p l lp (Cons x gp))
parti1 p Nil lp gp = (lp,gp)
ifqs True x y = x
ifqs False x y = y
app Nil l = l
app (Cons x l1) l2 = (Cons x (app l1 l2))

qs20 = qs
        (Cons 435 (Cons 413 (Cons 572 (Cons 827 (Cons 228
        (Cons 585 (Cons 505 (Cons 568 (Cons 128 (Cons 756
        (Cons 900 (Cons 477 (Cons 657 (Cons 317 (Cons 792
        (Cons 673 (Cons 977 (Cons 390 (Cons 942 (Cons 659
        Nil))))))))))))))))))))
```

プログラム 17 — rev1000.gs —

```
data List = Nil | Cons Int List
rev Nil = Nil
rev (Cons x xs) = app (rev xs) (Cons x Nil)
app Nil xs = xs
app (Cons x xs) ys = Cons x (app xs ys)

mkcons [] = Nil
mkcons (x:xs) = (Cons x (mkcons xs))
cons1000 = mkcons list1000b
rev1000 = rev cons1000

printZ Nil z = z
printZ (Cons x xs) z = printZ xs z
printZRev1000 = printZ rev1000 "rev1000 end"
main ~(Success:_) = [AppendChan "stdout" printZRev1000]

list1000b =
        [188,127,519,809,106,820,163,958,795,675
        ,261,232,279,596,630,666,726,469,403,136
```

```
 /* 中略 */
,204,301,673,557,606,547,294,227,911,209
,761,206,216,428,837,878,325,13,502,249
]
```