SC言語処理系における変形規則の再利用機構

平石 拓　八杉昌宏　湯淺太一

我々はSC言語処理系という，S式ベースの構文を持つC言語を利用することにより，C言語への変換による言語拡張の開発を支援するシステムを開発している．本論文では，SC言語処理系による言語開発の事例を通して明らかになった問題点を示し，それを解決するために行ったシステムの改良，特に，言語間の変換を定義する既存の変換フェーズ（変形規則セット）を拡張するための追加機能について述べる．Common LispのCLOSと動的変数を利用して，同一のコードから適用される変形関数を動的に決定する仕組みを採用したことにより，多くの言語拡張はベース言語の恒等変換や既存の変換との差分のみを記述して実装できるようになった．また，様々な拡張言語の実装で共通してよく利用される変換フェーズを，C言語への変換全体の一部として組み込むことも容易になった．実際，本処理系により，マルチスレッディング，ごみ集め，負荷分散などの機能を備えた複数の拡張言語を実装してきた．本論文では特にマルチスレッディング機能を例として提案機構の有効性の詳細を示す．

We are developing the SC language system, which facilitates language extensions by translation into C. SC languages are extended/plain C languages with an S-expression based syntax. This paper shows those problems in this system that we have found during our process of the development of extended languages using this system and our improvement of this system as our solutions to them. In particular, we focus on additional features to extend existing translation phases (transformation rule-sets) between languages. These features employ a mechanism that dynamically determines the transformation function to apply for a single code fragment. We implemented this mechanism by using CLOS and dynamic variables in Common Lisp. The proposed reuse mechanism enables us to implement extended languages only by writing the difference from existing translators including the identity translators into the base languages. It also helps us reuse a commonly-used rule-set as part of the entire translation. We implemented various features as extensions to C including multithreading, garbage collection, and load balancing. This paper discusses the multithreading case and shows the effectiveness of the proposed mechanism.

1　はじめに

C言語は実用システムの開発に欠かせないものとなっており，実用システムの開発を効率良く行うために拡張C言語が用いられることも多い．C言語拡張の実現方法としては，既存のCコンパイラに手を加える方法以外に，拡張Cプログラムを変換しCコードを生成する方法が考えられる．

我々は，このような変形ベースによる言語拡張を支援するシステムとして，SC言語処理系 [17] [16] [3] を提案している．SC言語とはS式ベースの（Lisp風の）構文を持つC言語および拡張C言語の総称であり，特に非拡張のC言語に相当するSC言語をSC-0言語と呼ぶ．SC-0言語からC言語への変換器を本処理系が提供しているため，利用者は拡張言語をSC-0言語への変換器として実装することができる．

S式で表現されたプログラムは構文解析なしに抽象構文木（Abstract Syntax Tree, AST）として直接扱うことができ，Lispを利用すればこのようなASTの解析・変形も容易に行える．さらに，Common Lispの動的変数を利用して変換時の文脈も簡単に扱えるため，上記のような変換器の実装は比較的容易に行

A Reuse Mechanism of Transformation Rules for the SC Language System

Tasuku Hiraishi, 京都大学学術情報メディアセンター, Academic Center for Computing and Media Studies, Kyoto University,

Masahiro Yasugi, Taiichi Yuasa, 京都大学情報学研究科, Graduate School of Informatics, Kyoto University,

える．

我々はこれまでに本システムを利用して様々な言語拡張の実装を行っているが，これらの開発の過程においてSC言語処理系自体の問題に直面することがあり，そのたびに処理系の改良を重ねてきている．例えば，S式間の変形関数の定義を簡便に書けるようにするためS式のパターンマッチの機能 [17][3] や，SCプログラムから既存のCヘッダファイル中の関数宣言やマクロ定義を利用できるようにするためC言語からSC-0言語への変換器 [16] を導入した．

本論文は，上記以外の重大な問題の1つとして，変形規則の実装のメンテナンス性・再利用性の問題を取り上げ，それに対して行った処理系の改良について述べる．この問題は，複数の変換フェーズを組み合わせて利用する際，各変換フェーズにおいて入力言語の全ての構文に対応するため，コードの複製が大量にできてしまうことの原因として存在する．我々はこれを解決するため，変換器を拡張するための機構をSC言語処理系に組み込んだ．この機構により，変換器の実装における差分プログラミングが可能になり，既存の変換器の実装をより効率的に再利用できるようになった．

本論文の貢献は以下の通りである．

- 言語拡張の変換ベースの実装において，各変換フェーズおよび変換フェーズの拡張（差分を適用した変換フェーズの実装）を，実装レベルでそれぞれクラスおよびクラス拡張に対応させるという枠組みが，実際の開発においてうまくはたらくことを示した．たとえば，既存言語向けの型情報追加変換フェーズを新しい言語用に拡張したり，既存の2つの言語拡張がある時に，片方の言語拡張の足りない部分を補完することで両方の拡張を同時に適用できるようにする際に役に立つ．
- 上記のような枠組みにおいて，提案する再利用性を高めた変換器記述言語はCLOSや動的変数を利用したシンプルな実装で提供できることを示した．

本論文ではまた，実際に本処理系を用いて開発したいくつかの高水準言語を紹介する．これらの言語は，LW-SC言語という入れ子関数の機能を追加した拡張SC言語 [3][4] を中間言語として利用することにより実現している．また特に，C言語にマルチスレッディングの機能を追加した言語MT-SCの開発の事例を取り上げ，提案する機構によって変換フェーズの実装・管理コストを削減できたことを示す．

2 SC言語処理系

本章では，SC言語処理系の機能について説明する．ただし，処理系の全体構成やSC言語そのものについては文献 [17][16][3] で既に示しているので，本論文を読み進めるために最低限必要な概略のみを2.1節で示した後，2.2節で変換フェーズ（変形規則セット）を拡張するための再利用機構を提案する．

本提案は，「変形規則セット」の概念を明示的に用いて変換フェーズを定義するものであり，その仕様はこれまでの文献で示したものと大きく異なる．そのため，2.2.1節〜2.2.2節でその仕様と利用例を詳しく述べる[†1]とともに，2.2.3節においてCLOSと動的変数を利用した実装を示す．

2.1 概要

SC言語処理系は，Common Lispで実装された，以下のモジュールから構成されるシステムである．

SCプリプロセッサ 他のSCファイルのインクルードやマクロ展開など，C言語のプリプロセッサに相当する処理を行う [16]．

SC変換器 SC言語のプログラムを，定義された変形規則に従って変換し別のSC言語のプログラムを生成する．

SCコンパイラ SC-0プログラムをCプログラムに変換する．

図1にSC言語処理系におけるコード変換の流れを示す．拡張SC言語のプログラムは（複数の）SC変換器によりSC-0言語のプログラムに変換され，その後，SCコンパイラによってC言語のプログラムに変換される．ただし，各変換フェーズの前にはSCプリプロセッサによる前処理が適用される．言語拡張の実

†1 2.2節のうち，2.2.1.3節で説明しているパターンの記法については，従来から大きな変更はないが，変形規則のコード例を理解するために必要である．

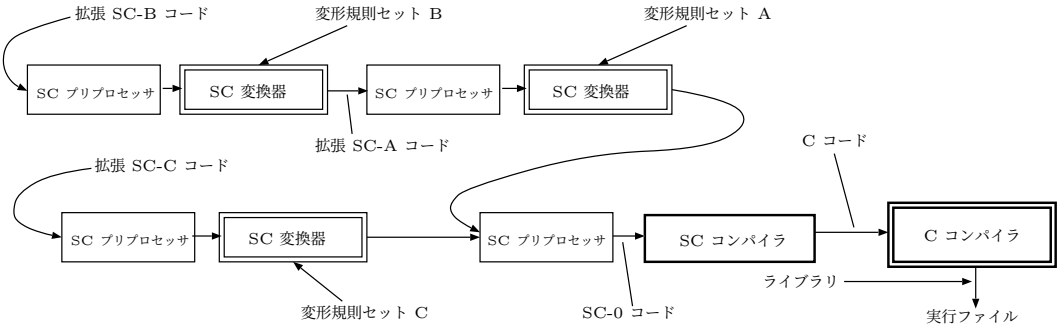

図 1　SC 言語処理系における変換の流れ

```
(def (sum a n) (fn int (ptr int) int)
  (def s int 0)
  (def i int 0)
  (do-while 1
    (if (>= i n) (break))
    (+= s (aref a (inc i))))
  (return s))
```

図 2　SC-0 プログラムの例

```
int sum (int* a, int n)  {
  int s=0;
  int i=0;
  do{
   if ( i >= n ) break;
   s += a[i++];
  } while(1);
  return s;
}
```

図 3　図 2 と等価な C プログラム．

装者は，変形規則を書くことで新しい SC 変換器を実装する．

SC 変換器による変換の最終ターゲットである SC-0 言語のプログラム例を図 2 に示す．このプログラムに SC コンパイラを適用すると，図 3 の C プログラムが生成される．なお，文献[17] に SC-0 言語の全構文を示している．

2.2　変形規則

2.2.1　仕様

図 1 における SC 変換器による各フェーズの変換は，**変形規則セット** により規定される．変形規則セットは，名前，パラメータ集合および親変形規則セットのリストで定義する．

また，変形規則セットに属する**変形関数**を定義できる．変形関数とは，変形規則セットにおいてある構文カテゴリに関する変形規則をとりまとめ，適用することにより変形を行う関数である．変形関数は，名前および変形規則の列から定義する． ここで，変形関数の名前には，慣例として `declaration` や `expression` のような構文カテゴリ名を用いることとする．また，変形規則は，S 式のパターンとそれに対応するアクションの組で定義する．

変形規則セットはオブジェクト指向言語のクラスと同様の階層構造を持つ．つまり，既存の変形規則セットを拡張して新たな変形規則セットを定義することができる．

定義された変形関数は，その名前を持つ Lisp 関数のように呼び出すことができる．しかし，通常の Lisp 関数とは異なり，実際に呼び出される関数の実体は単に名前のみから決定されるのではなく，名前と現在使用中の変形規則セットから決定される．呼び出しの詳細は 2.2.1.5 節で述べる．

2.2.1.1　変形規則セットの定義

図 1 に示した各々の変換フェーズは，変形規則セットとして定義される．規則セット定義の構文は以下の通りである．

`(define-ruleset` *rule-set-name*
　　(*parent-rule-set-name* ...)
　(*parameter-name default-value*) ...)

rule-set-name は定義しようとしている変形規則セッ

トの名前である．*parent-rule-set-name* には 0 個以上の定義済みの変形規則セットを指定できる．定義している変形規則セットは *parent-rule-set-name* で指定した変形規則セットの拡張となり，全ての変形関数およびパラメータをこれらの変形規則セットから継承する．

parameter-name で指定したパラメータや *default-value* の値はクラスのメンバおよびその初期値に相当し，変形規則の本体から（`ruleset-param` という名前の関数を用いて）参照することができる．ただし，いくつかの名前のパラメータは特別な意味を持つ．そのようなパラメータには例えば

- `entry`：変形規則セットが適用された際，最初に呼ばれる変形関数の名前（シンボル）を指定する．
- `default-input-handler`：1 引数をとる関数を値として持つ．この関数は，変形関数が適用された入力がどのパターンにもマッチしなかった場合，その入力を引数として呼び出される．

などがある．

2.2.1.2 変形関数の定義

変形規則セット *rule-set-name* に属する変形関数 *category* は，以下のように，パターンとアクションの組からなる変形規則を本体に並べて書いて定義する．

(`defrule` *category rule-set-name*
  ((`#?`$pattern_{11}$ ... `#?`$pattern_{1m_1}$)
   form-list$_1$)
  ...
  ((`#?`$pattern_{n1}$ ... `#?`$pattern_{nm_n}$)
   form-list$_n$)
  [(`otherwise` *form-list*$_{\mathrm{otherwise}}$)])†2

変形関数が適用されると，その引数の値（S 式）が $pattern_{11}$, ... $pattern_{1m_1}$, $pattern_{21}$, ... $pattern_{2m_1}$, ... $pattern_{n1}$, ... $pattern_{nm_n}$ の順でパターンにマッチするかがチェックされる．その結果 $pattern_{ik}$ に最初にマッチすると，*form-list*$_i$ が Lisp の式として評価され，その結果が変形関数の返り値となる．引数の値がどのパターンにもマッチしなかった場合，もし *form-list*$_{\mathrm{otherwise}}$ が書かれていればこの式が評価される．書かれていない場合は，変形規則セット *rule-set-name* のパラメータである `default-input-handler` 関数が，この値を引数として呼び出される．

`defrule` の代わりに `extendrule` をキーワードに用いて変形関数を定義することもできる．`extendrule` は `defrule` とほとんど同じ構文および意味を持つ．両者の違いは，引数の値がどのパターンにもマッチせず，`otherwise` 節も書かれていなかった場合の動作である．`extendrule` においてこの状態になると，親の変形規則セットで定義されている同名の変形関数が引き続き適用される．†3 *rule-set-name* が複数の親を持つ場合，適用順序は Common Lisp Object System（CLOS）の “class precedence list” [1] により決定される．

2.2.1.3 変形規則の記法

前述の通り，変形規則はパターンとアクションの組である．このうち，アクションは通常の Lisp フォームである．パターン（*pattern*）は Lisp のバッククォートマクロ風の記法で書ける．より正確には *pattern* は以下のいずれかの要素から構成される S 式である．

(1) *symbol*
シンボル *symbol* と `eq` の意味で等しいシンボルにマッチする．

(2) `,`*symbol*
任意の 1 要素にマッチする．

(3) `,@`*symbol*
0 個以上の要素列にマッチする．

(4) `,`*symbol* [*function*]
(`funcall #'`*function element*) を評価した結果が非 `nil` であるような要素 *element* にマッチする．

(5) `,@`*symbol* [*function*]
(`every #'`*function list*) を評価した結果が非 `nil` であるような要素列 *list* にマッチする．

†2 $m_i = 1$ の場合，*pattern* を囲む括弧は省略できる．

†3 ここで説明した `extendrule` の挙動を `defrule` のデフォルトの挙動とする選択も有り得たが，規則がさらに続くかどうかをプログラマに明確に意識させるため，本文のような仕様とした．

ここで，関数 *function* には変形関数あるいは通常の Common Lisp 関数（組み込み関数または変形規則とは別に定義したユーザ定義関数）を指定できる．また，`lambda` 特殊形式を *function* として直接書くこともできる．

`defrule`，`extendrule` の本体では，シンボル “`x`” が変形関数の引数として与えられた S 式全体に暗黙的に束縛されている．さらに，(1) 以外においては，シンボル *symbol* が S 式のマッチした部分に，対応するアクションの実行環境において自動的に束縛される．また，(4)，(5) における関数 *function* の返り値は，`get-retval` という名前のライブラリ関数を，対応するシンボル *symbol* を引数として呼び出すことで得ることができる．

2.2.1.4 変形規則セットの適用

定義された変形規則セット（変換フェーズ）は，`apply-ruleset` 関数を用いて以下のように適用する．

(`apply-ruleset` *input* :*rule-set-name*)

input には S 式または SC プログラムのソースファイルを指定する．

`apply-ruleset` 関数には，変形規則セットの定義時に指定した *parameter-name* に対応するキーワード引数を渡すこともできる．例えば，式

```
(apply-ruleset ~(* a b) :sc0-to-sc0
               :entry 'expression)
```

を評価すると，`sc0-to-sc0` 変形規則セットに属する `expression` 変形関数が `(* a b)` に適用される．

なお，`apply-ruleset` の処理中，「現在適用中の規則セット」を示す`*current-ruleset*`という名前のシンボルが，指定された変形規則セットに動的束縛される．

2.2.1.5 変形関数の適用

`defrule`，`extendrule` で定義された変形関数は，1 つの必須パラメータと複数のオプショナルパラメータを受け取る関数として呼び出すことができる．必須パラメータには変形対象となる S 式を与える．オプショナルパラメータには変形規則セット名（シンボル）を与えることができる．

変形規則セット名が省略された場合は，「現在の規則セット」に属する変形関数が呼ばれる．「現在の規則セット」に該当する変形関数が定義されていなかった場合は，祖先の変形規則セットを辿り，最初に見つかった変形関数が呼ばれる．祖先を辿っても変形関数が見つからなかった場合はエラーとなる．

変形規則セット名をオプショナルパラメータとして明示的に与えた場合，「現在の規則セット」ではなく，指定された規則セットに属する変形関数が呼ばれる（変形関数がその規則セットで定義されていなければ祖先を辿る）．この時，変形関数の実行中は「現在の規則セット」（動的変数`*current-ruleset*`）は与えられた規則セットに束縛される．

変形関数の本体で `ruleset-param` 関数を用いて変形規則セットのパラメータを参照できることは上で述べたが，その際に参照される変形規則セットは「現在の規則セット」である．

2.2.2 例

図 4 に変形規則セットの定義の例を示す．図中のコードの ‘~’ は，バッククォートとほぼ同じ意味を持つマクロ文字である．通常のバッククォートとの違いは，後続する S 式に含まれるシンボルが，「現在のパッケージ」（Common Lisp のシステムが提供する動的変数`*package*`の値）ではなく，SC コード用に区別されたパッケージにインターンされる点である．†4

図のコードでは，2 つの変形規則セットが定義されている．このうち `sc0-to-sc0` 規則セットは SC-0 プログラムから SC-0 プログラムへの恒等変換を定義しており，`sc1-to-sc0` 規則セットは SC-1 言語から SC-0 言語への変換を定義している．なお，SC-1 言語とは反復や変数束縛のためのいくつかのコンストラクトを SC-0 言語に追加した言語である．

ここで，Lisp 式

†4 通常の Common Lisp プログラミングにおいては，このような目的のために `keyword` パッケージが利用されることが多いが，全てのシンボルの前に ‘:’ を書く手間を省くためにこのようなマクロ文字を導入している．

```
(define-ruleset sc0-to-sc0 ()
  (entry 'sc-program)
  (default-input-handler #'no-match-error))
(defrule sc-program sc0-to-sc0
  (#?(,@decl-list)
   (mapcar #'declaration decl-list)) )
(defrule declaration sc0-to-sc0
  (#?(,scs[storage-class-specifier] ; 関数定義
      (,@id-list[identifier]) (fn ,@tlist)
      ,@body)
   ~(,scs (,@id-list) ,(third x)
       ,@(mapcar #'block-item body)))
  ...)
(defrule block-item sc0-to-sc0
  ((#?,bi[declaration]
    #?,bi[statement])
   (get-retval 'bi))
  )
(defrule statement sc0-to-sc0
  (#?(do-while ,exp ,@body)
   ~(do-while ,(expression exp)
      ,@(mapcar #'block-item body)))
  ...
  (otherwise (expression x)) ; 式文
  )
(defrule expression sc0-to-sc0 ...)
...

(define-ruleset sc1-to-sc0 (sc0-to-sc0))
(extendrule statement sc1-to-sc0
  ((#?(let (,@decl-list) ,@body) )
   ~(begin ,@(mapcar #'declaration decl-list)
           ,@(mapcar #'block-item body)) )
  ((#?(while ,exp ,@body) )
   (let ((cdt (expression exp)))
     ~(if ,cdt
          (do-while ,cdt
            ,@(mapcar #'block-item body))) ))
  ((#?(for (,@list ,exp2 ,exp3) ,@body) )
   (let ((e1-list (mapcar #'block-item list))
         (e2 (expression exp2))
         (e3 (expression exp3))
         (new-body (mapcar #'block-item body)))
     (list ~(begin
              ,@e1-list
              (if ,e2
                  (do-while (exps ,e3 ,e2)
                    ,@new-body))))
     ) )
  ((#?(loop ,@body) )
   ~(do-while 1 ,@(mapcar #'block-item body)))
  )
```

図 4 変形規則セットの定義の例

```
(statement ~(do-while x (while y (++ z)))
           :sc1-to-sc0)
```

の評価は以下のように進行する.

1. sc1-to-sc0 規則セットの statement 変形関数が与えられた S 式に適用されるが，どのパターンにもマッチしない.
2. sc0-to-sc0 規則セットの statement 変形関数が適用され，パターン#?(do-while ...) にマッチする.
3. sc1-to-sc0 規則セットには expression 変形関数が定義されていないため，sc0-to-sc0 規則セットの expression 変形関数が x に適用され，x が返り値となる.
4. sc1-to-sc0 規則セットには block-item 変形関数が定義されていないため，sc0-to-sc0 規則セットの block-item 変形関数が (while y (++ z)) に適用される．sc0-to-sc0 規則セットの block-item 中の変形規則のアクション部は statement 変形関数の呼び出しを含むが，ここで呼ばれるのは，sc0-to-sc0 規則セットの statement 変形関数ではなく，sc1-to-sc0 規則セットの statement 変形関数である．したがって，呼び出しの結果は (if y (do-while y (++ z))) となる.
5. (do-while x (if y (do-while y (++ z)))) が最終的に変形結果となる.

文脈によって変形関数が呼び出される際の「現在の規則セット」が異なると，呼び出される「現在の規則セットの変形関数」も変化する．例えば，(statement ~(while ...) :sc0-to-sc0) を評価すると，sc0-to-sc0 規則セットの statement 変形関数が (while ...) に適用されることになり，その結果 default-input-handler である no-match-error が呼び出される.

2.2.3 実装

これまで述べた変形規則セットの拡張機構は CLOS と動的変数を利用して実装している.

図 5 に define-ruleset，defrule，extendrule および apply-ruleset の実装コードの一部を示す．define-ruleset を defclass，defrule（extendrule）を defmethod にそれぞれ対応させることで，1 つの変形規則セットは 1 つの CLOS のクラスとして実装され，変形関数の実体は，属する変形規則セットに対応するクラスのオブジェクトを引数にとる CLOS の（マルチ）メソッドとして実装される．つまり，適用しようとしている変形規則セットの変形関数の実体が適切にディスパッチされるようにするには，変形規則セットに対応するクラスのオブジェクト

```
(defmacro define-ruleset
          (name parents &body parameters)
  `(defclass ,(ruleset-class-symbol name)
        ,(or (mapcar #'ruleset-class-symbol
                     parents)
             (list *base-ruleset-class-name*))
     ,(loop for (p v) in parameters
         collect `(,p :initform ,v
                      :initarg ,p))))

(defun apply-ruleset
       (input ruleset &rest initargs)
  (let ((*current-ruleset*
         (apply #'make-instance
                ruleset initargs)))
    (funcall (rule-function
              (slot-value *current-ruleset*
                          'entry))
             (if (or (stringp input)
                     (pathnamep input))
                 (sc-file:read-sc-file input)
                 input))))

(defun rulemethod-args (ruleset)
  `(x (,*ruleset-arg*
       ,(ruleset-class-symbol ruleset))))
(defmacro defrule
          (name ruleset &body pats-act-list)
  `(progn
     (defmethod ,(rule-method-symbol name)
                ,(rulemethod-args ruleset)
       (block ,name
         (case-match x
          ,@pats-act-list
          (otherwise
           ;; extendrule では (call-next-method)
           (call-otherwise-default
            x ',ruleset)))))
     (defun ,(rule-function-symbol name)
         (x &optional
            (ruleset *current-ruleset* r)
            &rest initargs)
       (if r
           (let ((*current-ruleset*
                  (apply #'make-instance
                         ruleset initargs)
             (,(rule-method-symbol name)
              x *current-ruleset*))
            (,(rule-method-symbol name)
             x *current-ruleset*)))))
```

図 5　変形規則セット定義関数の実装コード

をメソッドの引数とすればよい.

また，動的変数*current-ruleset*を適用中の変形規則セットを記憶するために利用しており，各変形関数名に対応した関数はこの動的変数を引数として実際のメソッドを呼び出すラッパーになっている．こうすることで，各変形関数呼び出し式において自明な引数*current-ruleset*を逐一書く必要がないようにしている.

```
(sc0-program (,@decl-list))
-> (mapcar #'sc0-declaration decl-list)) )
(sc0-declaration
 (,scs[sc0-storage-class-specifier]
  (,@id-list[sc0-identifier]) (fn ,@tlist) ,@body))
-> `(,scs (,@id-list) ,(third x)
     ,@(mapcar #'sc0-block-item body)))
...
(sc0-block-item ,bi[sc0-declaration])
(sc0-block-item ,bi[sc0-statement])
-> (get-retval 'bi)

(sc0-statement (do-while ,exp ,@body))
-> `(do-while ,(sc0-expression exp)
      ,@(mapcar #'sc0-block-item body)))
...
(sc0-statement ,otherwise)
-> (sc0-expression x)
...

(sc1-program (,@decl-list)) -> ...
(sc1-declaration
 (,scs[sc1-storage-class-specifier] ...)) -> ...
(sc1-block-item ,bi[sc1-declaration])
(sc1-block-item ,bi[sc1-statement])   -> ...

(sc1-statement (do-while ,exp ,@body)) -> ...
...
(sc1-statement (let (,@decl-list) ,@body))
-> `(begin ,@(mapcar #'sc1-declaration decl-list)
           ,@(mapcar #'sc-block-item body))
(sc1-statement (while ,exp ,@body))
-> (let ((cdt (sc1-expression exp)))
     `(if ,cdt
          (do-while ,cdt
            ,@(mapcar #'sc1-block-item body))) )
(sc1-statement (for (,@list ,exp2 ,exp3) ,@body))
-> (let ((e1-list (mapcar #'sc1-block-item list))
         (e2 (sc1-expression exp2))
         (e3 (sc1-expression exp3))
         (new-body (mapcar #'sc1-block-item body)))
     (list `(begin
              ,@e1-list
              (if ,e2
                  (do-while (exps ,e3 ,e2)
                    ,@new-body)))))
(sc1-statement (loop ,@body))
-> `(do-while 1 ,@(sc1-function-body body))
```

図 6　従来の仕様における変形規則の定義

3　議論

3.1　従来の変形規則記述言語の問題点

図 4 に相当する変形の実装は，SC 言語処理系の従来の変形規則記述言語では図 6 のように書ける．この記述言語の仕様では，全ての変形関数，変形規則は同一レベルにあり,「変形規則セット」の概念は明示的には存在していなかった[†5]．そのため，実装しようとしている拡張言語とベースとなる言語との差分がわずかであっても拡張言語の全ての構文に対する変形関数（がとりまとめる全ての変形規則）を定義しなければならないという問題があった.

†5　言葉自体は以前から用いている [17] [3] が，1 つの変換フェーズに関連付けられた規則のグループに対する便宜上の呼称である

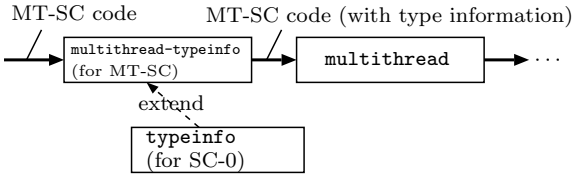

図 7 全体の変形の一部を提供する変形規則セット

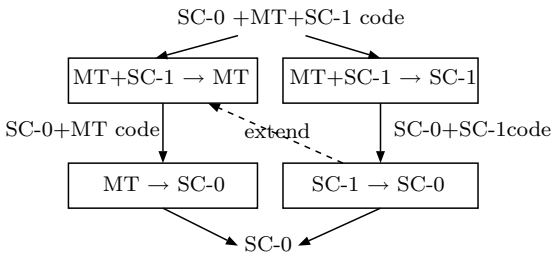

図 8 複数の変形規則セットの適用

たとえば，図 6 の sc1-statement の定義において sc0-statement を再利用しようとしても，sc0-statement には sc0-block-item の呼び出しが含まれており，これを sc1-block-item の呼び出しに書き換えなければ正しい変形規則とはならない．このため，sc0-statement を複製したコードの sc0-block-item の呼び出し部分を書き換えるという作業が発生し，これはコードの管理効率を著しく低下させる．

また，記述の直観性も，BNF 風の現在の記法と比べるとやや低いものであった．

3.2 変形規則セットの拡張機構の有効性

改良後の記述言語では，新たな変形規則セットを，既存の変形規則セットを拡張して定義することができるようになっている．図 4 においては，sc1-to-sc0 規則セットは SC-0 言語に対する恒等変形を実装している sc0-to-sc0 規則セットの拡張により定義されている．このように sc0-to-sc0 を共通のテンプレートとして用いることで，多くの SC-0 言語の拡張言語のための変形規則セットは，SC-0 言語との差分のみを書くことで実装できる．

ある程度本格的な機能を提供する拡張言語の実装においては，全体の変形を複数のフェーズに分けて実装することが多い．変形規則セットの拡張機構は，このような際に，多くの言語の実装において共通に利用されるような変形規則セットを利用する際にも有効である．実際，後述の MT-SC 言語（SC-1 言語にマルチスレッド機能を追加した言語）を実装する際，全ての式（*expression*）に型情報を追加するための typeinfo 変形規則セットを利用したが，SC-0 言語用に実装された typeinfo をそのまま MT-SC 言語のプログラムに適用することはできない．しかし，typeinfo 規則セットから MT-SC 言語対応の multithread-typeinfo 規則セットへの拡張は少ない手間で行える（図 7）．

変形規則セットの拡張機構はまた，複数の言語拡張を同時に適用する際にも有効である．2 つ言語拡張 MT（multithread）および SC-1 のための変形規則セットが実装済みであり，これらの機能を両方備えた拡張言語を実装したい場合を考える．そのような拡張は，図 8 に示すように，2 つの変形規則セットを直列に接続することで実現できる．しかし，例えば図の左の経路を選んだ場合，SC-1 用の規則セットは MT で追加された機能もサポートするように変更しなければならない．このような変更も，SC-1 用の規則セットのコードを複製したものに手を加えるという方法ではなく，MT の拡張に対応するためのコードを追加して SC-1 を拡張することにより行うことができる．ただしこの手法は，2 つの拡張の片方が SC-1 の繰り返し構文の追加程度の比較的単純なものである場合には有効だが，両方の拡張が高度なものであり，それらを合わせ持つ拡張言語のセマンティクスが明らかでない場合の困難を特に解決するわけではない．

図 9 に，MT-SC 言語から MT の拡張のみを備えた言語への変換を実現する変形規則セットのコードを示す．このような変形規則セットの定義は，sc1-to-sc0 規則セットを MT-SC 言語に対応させるための差分コードを書くのみで完了する．また，この差分コードは sc1-to-sc0 規則セットのコードとは独立に書くことができ，従来に比べてコードの管理がしやすくなっている．

なお，図 8 において二つの拡張が（同じ構文カテ

```
(define-ruleset multithread-sc1 (sc1-to-sc0))

(extendrule statement multithread-sc1
  (#?(thread-create ,dec-list ,@body)
   ~(thread-create
      ,(mapcar #'declaration dec-list)
      ,@(mapcar #'block-item body)) )
  (#?(thread-suspend ,id[identifier] ,@body)
   ~(thread-suspend ,id
      ,@(mapcar #'block-item body)) )
  (#?(thread-resume ,exp)
   ~(thread-resume ,(expression exp)))
  )
```

図 9　MT-SC 言語に対応するための sc1-to-sc0 規則セットの拡張

ゴリに関して拡張していても）完全に直交である場合は,

```
(define-ruleset A+B-to-sc0
                (A-to-sc0 B-to-sc0) ...)
```

のように多重継承を用いて多段変換によらずに変形を定義することもできる.

3.3　SC 言語処理系以外への応用

提案する再利用機構は CLOS と動的変数を利用したものである．このうち CLOS については，Lisp のマクロによって想定している利用形態に特化させることで，利用できる CLOS の機能に制限を加える代わりに，変形規則の記述の簡便性，可読性およびモジュール（＝変形規則セット）のメンテナンス性を高めたものと理解することができる．またここで，内部的に動的変数を利用して，変形規則セットに相当するクラスのオブジェクトを変形関数の呼び出しの際に暗黙的に引き渡すようにしたことで，変形規則の記述者がオブジェクトの存在を意識しなくてよいようにしている.

以上のように，この再利用機構は CLOS を特化したものであり，第一義的には一般性を意図したものではないが，関数プログラミングにおける相互再帰関数セットの拡張という一般化を考えることは可能である．すなわち,（オブジェクト指向プログラミングは行わない）関数プログラミングにおいて，ベースとなる相互再帰関数セットのうち一部の関数を，必要ならベース定義を再利用して再定義し，残りの関数はそのまま再利用するような拡張された相互再帰関数セットを定義・利用できるような再利用機構が考えられる.

4　関連研究

4.1　Java EPP

SC 言語処理系のように，言語の開発者が拡張部分をプラグインとして実装することにより変換ベースの言語拡張をサポートするシステムには，Java EPP [7][8][6] もある．ただし，EPP は Java コードを AST に変換するプリプロセッサの利用が前提であり，プログラミング言語が AST そのものではない点などが本提案方式とは異なる.

EPP においても，複数の拡張を同時に適用する際の問題は認識されており，例えば文献 [13] などで（一般のプログラミングにおけるモジュールによる機能拡張という，より一般的な問題として）議論されている．ここで提案されているプログラミング言語 MixJuice は，2 方向以上の拡張を同時に適用する際に生じる「実装欠損」の問題を解決するため，それを補うための「補完モジュール」を自動的にリンクする機能を備えている．ただし，補完モジュール自体は全ての適用しようとする拡張の実装を理解する者の手によって実装される必要がある.

この「補完モジュール」は，3.2 節の議論における mulththread-typeinfo や multithread-sc1 規則セットの拡張部分に相当するが，SC 言語処理系では，このような補完部分を独立に書くことはできるものの，それらを自動的にロードするような機構は備えていない．しかし，mulithtread と sc1 という 2 方向の拡張を適用する場合には，一般には multithread-sc1 という補完が必要であるという規則セットの命名規則も含めた慣例を守ることにより，ある程度安全に複数の拡張の適用が可能であると考えられる.

4.2　特定言語で閉じたプログラム変換モジュール

GCC [9] や COINS [12] では，ソース言語から中間言語，中間言語から実行形式への各変換器および中間言語の仕様を開発者に提供している．これにより，開発者は中間言語のプログラム変換を実装して最適化

等を実現できるようになっている．これらのフレームワークにおいては，同一言語内で閉じた，かつプログラムの意味を変えない変換しか基本的には扱わない．そのため，独立に開発された変換モジュールを組み合わせて利用する際の問題がそもそも生じず，そのための支援機構も特に存在しない．

SC 言語処理系は，SC-0 という中間言語から先のバックエンドの変換器を提供しているという点では GCC や COINS と共通しているが，拡張 SC 言語から SC-0 言語までの変換モジュールは本処理系の利用者による開発対象となっている．そのような適用前後で言語が異なる変換モジュールは，適用順序に気を付ける必要があるほか，1 つの変形関数の実装を異なる変換モジュールから単純に利用できないという問題が生じる．そのため，本論文で提案するようなモジュール管理機構が有用となる．

GCC や COINS が扱うような最適化等の開発を SC 言語処理系を用いて行うことも可能である．ただし，GCC や COINS においてはそれらが扱う中間言語用に提供されたデータフロー解析などの既存の支援機能を利用できるという利点がある．

5 拡張言語の実装

SC 言語処理系は，比較的単純なものから複雑なものまで様々な言語拡張の開発に利用することができる．前者の例としては，簡単な繰り返し構文の追加や，Java 風のラベル付き `break`/`continue` の追加 [17] などが考えられる．複雑な機能の例としては，例外ハンドラ [10] や，入れ子関数などが挙げられる．これらを実装するためには，一時変数のための宣言の追加や式の型情報を獲得できる仕組みが必要となる．

我々は以前の研究において，SC 言語に入れ子関数の機能を追加した LW-SC 言語 [3] [4] を提案している．入れ子関数とは，関数定義の本体で定義される関数のことである．入れ子関数の定義を評価すると，生成時環境における lexical スコープの変数にアクセス可能なクロージャが得られる．このクロージャを間接的に呼び出すことにより，合法的な実行スタックへのアクセスが可能となる．このことを利用し，LW-SC 言語を中間言語として用いることで，“stack walk” が必要な様々な高水準機能（チェックポインティングやコピー方式ごみ集めなど）が SC-0 言語，C 言語への変換により比較的容易かつ統一的な方法で実装できる．

また，LW-SC 言語は L-closure [11] [15] に基づいた入れ子関数の実装手法を採用することで，クロージャの維持・生成コストの大部分を削減しているため，上記の手法で実装した高水準言語は性能面でも効率良く動作する．

実際に我々が LW-SC 言語を中間言語として利用して実装した拡張言語には以下のものがある．

HSC（High-level SC）言語 動的に生成されたオブジェクトが（保守的でない）コピー方式ごみ集めによって回収されるようにした，メモリ安全な SC-1 言語 [5].

MT-SC (Multithread SC）言語 文献 [18] の手法を用いて，SC-1 言語に言語レベルのマルチスレッディングの機能を追加した SC-1 言語 [17].

Tascell 言語 文献 [2] で提案した論理スレッドフリーな動的負荷分散による並列計算フレームワーク用の記述言語であり，SC-1 言語に提案手法に基づく並列計算用のコンストラクトが追加されている．上記の文献では，C 言語風の構文を持つ言語として説明されているが，オリジナルの設計では SC-1 言語を拡張した S 式構文の言語であり，SC 言語処理系の変形規則セットにより実装されている．C 言語構文の Tascell は一般的な構文解析器に基づく構文のみの変換により実現される．

これらの言語のうち，LW-SC 言語と MT-SC 言語はぞれぞれ文献 [4]，[17] において，実際の変形規則コードレベルでの詳細を示している．本章では，例として MT-SC 言語を取り上げ，改良後の変形規則記述言語による実装と従来の実装を比較することで，提案する再利用機構が実際に有効であることを示す．

5.1 MT-SC 言語の仕様

5.1.1 仕様

まず，MT-SC 言語の仕様を説明しておく．この仕様は文献 [17] のものと同一である．

MT-SC 言語で追加されたプリミティブは以下の通

```
(def (pfib n) (fn int int)
  (def x int) (def y int)
  (def nn int 0) (def c cont 0)
  (if (<= n 1)
      (return 1)
      (begin
       (thread-create
         (= x (pfib (- n 1)))
         ;;  下の thread-suspend が実行されていれば
         ;;  そのスレッドを再開させる.
         (if (== (++ nn) 0)
             (thread-resume c)))
       (= y (pfib (- n 2)))
       ;; (pfib (- n 1)) の計算が終わっていなければ
       ;;  suspend して終了を待ち合わせる.
       (if (< (-- nn) 0)
           (thread-suspend c0 (= c c0)))
       (return (+ x y)))))
```

図 10　MT-SC 言語のプログラム例

りである.

- (`thread-create` *body*)：*body* を実行する新たなスレッドを生成する.
- (`thread-suspend` *identifier body*)：変数 *identifier* を現在の継続で束縛し，その継続を保存するために *body* を実行した後，自らのスレッドを “suspended” にする.
- (`thread-resume` *expression*)：“suspended” の状態のスレッドを再開する．*expression* は `thread-suspended` で保存しておいた継続である.

ここで，MT-SC 言語における「スレッド」は OS スレッドではなく，言語レベルのスレッドである．各スレッドは “active” か “suspended” のいずれかの状態を持つ．`thread-create` 文により生成されたスレッドは，最初は “active” の状態である．スレッドの実行中，`thread-suspend` 文を実行することにより自らを “suspended” の状態にすることができるが，この時にスレッドの継続を保存しておくことができる．他のスレッドが，この保存された継続を指定して `thread-resumue` 文を実行することにより，スレッドが再開される．与えられた計算が完了するとスレッドは消滅する．実行中のスレッドが “suspended” になるか消滅するとスケジューラが呼び出される．スケジューラは “active” なスレッドの中から次に実行するスレッドを選択してそこに制御を移す.

図 10 に MT-SC のプログラム例を示す.

```
(decl (struct _thstelm))
(deftype cont (ptr (lightweight (ptr void)
                    (ptr (struct _thstelm)) reason)))
(def (struct _thstelm)
  (def c cont)
  (def stat (enum t-stat)))
(deftype thst_ptr (ptr (struct _thstelm)))
(def thst (array (struct _thstelm) 4192)) ; スレッドスタック
(def thst_top thst_ptr thst) ; スレッドスタックのトップ

(def (pfib c_p n) (fn int cont int)
 (def ln int 0)
 (def x int) (def y int)
 (def nn int 0) (def c thst_ptr 0)  (def c0 thst_ptr)
 (def tmp2 int) (def tmp1 int)

 (def (pfib_c cp rsn)
      (lightweight (ptr void) thst_ptr reason)
   (switch rsn
     (case rsn_cont)
       (switch ln
         (case 1) (goto L1)
         (case 2) (goto L2)
         (case 3) (goto L3))
       (return)
     (case rsn_retval)
       (switch ln
         (case 2)
           (return (cast (ptr void) (ptr tmp2))))
       (return))
   (return)
   ... owner 関数の本体とほぼ同じ内容がここに埋め込まれる ...
   )

 (if (<= n 2)
     (return 1)
     (begin
       ;;  現在の継続をスレッドスタックに push する
       (begin
         (= ln 1)
         (= (fref thst_top -> c) pfib_c)
         (= (fref thst_top -> stat) thr_new_runnable)
         (inc thst_top))
       ;; thread-create の本体
       (begin
         (def ln int 0)
         (def (nthr_c cp rsn)
              (lightweight (ptr void) thst_ptr reason)
           ...)
         (= ln 1)
         (= x (pfib nthr_c (- n 1)))
         (inc nn)
         (if (== nn 0) (thr_resume c)))
       ;; thread-create 本体の処理が終了したので
       ;;  スレッドスタックから継続を pop する
       (if (!= (fref (- thst_top 1) -> stat)
               thr_new_runnable)
           ;;  スケジューラを呼び出し,
           ;;  active な別のスレッドに処理を移す
           (scheduling)
         (dec thst_top))
       ;; (label L1)  入れ子関数ではここに再開位置を示すラベル
       (= ln 2)
       (= y (pfib pfib_c (- n 2)))
       ;; (label L2)  入れ子関数ではここに再開位置を示すラベル
       (= nn (- nn 1))
       (if (< nn 0)
           (begin
             ;;  現在のスレッドを中断
             (= c0 (inc thst_top))
             (= (fref c0 -> c) pfib_c)
             (= (fref c0 -> stat) thr_new_suspended)
             (= c c0)
             (= ln 3)
             ;;  スケジューラを呼び出し,
             ;;  active な別のスレッドに処理を移す
             (scheduling)))
       ;; (label L3)  入れ子関数ではここに再開位置を示すラベル
       (return (+ x y)))))
```

図 11　LW-SC 言語を利用したマルチスレッドの実装

5.1.2 変換フェーズ

MT-SC 言語から SC-0 言語への変換のうち，LW-SC 言語から SC-0 言語への変換部分は実装済みのものをそのまま利用できる．そのため，MT-SC 言語のプログラムからマルチスレッディングの機能を実現する LW-SC 言語のプログラムへの変換を実装すれば，MT-SC 言語の実装は完成する． たとえば，図 10 のプログラムは図 11 のプログラムに変換される．MT-SC において，入れ子関数を利用してどのようにマルチスレッディングを実現しているかの詳細は文献 [18] で述べている．

変形規則の記述が無用に複雑にならないようにするため，また変形規則セットの再利用性を高めるため，MT-SC から LW-SC への変換は以下の 7 つのフェーズ（変形規則セット）に分割して実装した．†6

`multithread-sc1` **規則セット** MT-SC 言語（SC-0+SC-1+MT）から SC-0+MT に相当する言語への変換．

`multithread-rename` **規則セット** 次の hoisting のための準備として，関数内の全ての局所変数およびパラメータ名が衝突しないように名前を変換する．

`multithread-hoist` **規則セット** 関数内の全ての変数宣言を，関数本体の先頭に移動させる．

`multithread-typeinfo` **規則セット** プログラム中の全ての式に，型情報を明示的に付与する．

`multithread-temp` **規則セット** `multithread` 規則セットによる変形を単純にするため，関数呼び出し式が式文としてのみ現われるよう変形を行う．また，そのために必要な一時変数の変数宣言を追加する．

`multithread` **規則セット** 入れ子関数の追加など，マルチスレッド機能を実現するために必要な変形を行う．

`untype` **規則セット** `multithread-typeinfo` 規則セットで付与した型情報を削除する．

5.1.3 従来の SC 言語処理系における実装

MT-SC の言語機能の実装の主要部分である `multithread` 規則セットは一から書き起こす必要があった．また，`untype` 規則セットは，

(`the` *type-expression expression*)

の形の式を，

expression

に機械的に置き換えるだけの変形を行うものであり，入力プログラムの言語に依存しないので，実装済みのものを直接利用することができた．

一方，それ以外の 5 つの規則セットについては，3.1 節で述べた通り，SC-0 言語用に実装した規則セットのコードをコピーし，変形関数名を書き換え，追加されたコンストラクトのための変形規則を追加する，という実装を行う必要があった．この作業自体はそれほど大きな手間ではなかったが，コードのコピーが大量にできてしまい，もとの（SC-0 言語用の）規則セットのバグを修正したときに，MT-SC 言語用のコードも書き換えなければならないなど，メンテナンス性が大幅に低下した．

また，このようなコピーは HSC，Tascell と新しく言語を開発するたびに作られてしまい，問題をさらに大きくしていた．

5.1.4 再利用機構を利用した実装

改良後の処理系においては，`multithread-*` の形の名前の変形規則セット は，SC-0 言語向けの変形規則セットとして実装済みであった `sc1-to-sc0`, `typeinfo`, `temp` などの各規則セットを拡張して実装できた．各拡張に必要だったコーディングは `statement` 変形関数を `extendrule` で拡張する 10 行程度のみである．

`untype` 規則セットについては，従来の実装でも問題は発生していなかったので，今回の改良による変化は特に無かった．

`multithread` 規則セットを書き起こす必要があったのも従来と同様である．ただし，3.2 節で述べた `sc1-to-sc0` 規則セットの実装と同様，型情報付 SC-

†6 文献 [17] においては，`multithread-sc1`, `multithread-rename`, `multithread-hoist` に相当するフェーズは，あまり本質的な変換ではないことや，入力として hoisting が必要でないプログラムを仮定していたことから説明が省かれている．またターゲット言語が LW-SC ではなく SC-0 となっているが，これは L-closure ではなく，GCC が拡張機能として提供している入れ子関数を利用していたためである．

0 言語（SC-0 言語に `typeinfo` 規則セットによる変換を適用した言語）に対する恒等変換を実装する `sc0t-to-sc0t` 変形規則セットをあらかじめ定義しておけば，`multithread` 規則セットはその拡張規則セットとして，差分のみを書くことで実装できるようになる．`sc0t-to-sc0t` 規則セットは `temp` 規則セットの実装にも利用できるほか，HSC や Tascell など `typeinfo` 規則セットを利用する別の拡張言語の実装にも利用できるため，上記のような実装をしておくことで，総合的な実装コストを削減することができ，メンテナンス性も向上する．`sc0t-to-sc0t` 規則セット自身も `sc0-to-sc0` 規則セットの拡張として実装できる．

5.1.5 コード量の比較

表 1 に，MT-SC 言語を実装するために必要であった変形規則セットの，従来の変形規則記述言語と再利用機構導入後の記述言語における行数の比較を示す．後者の記述言語に関しては，それぞれの規則セットがどの規則セットの直接の拡張であるかも示している．

`sc0-to-sc0` や `sc0t-to-sc0t` を直接拡張して実装した規則セットについては行数に大きな差がない．これは，従来システムの変形規則セットでは，宣言（*declaration*）でも文（*statement*）でもない関数本体中のトップレベル要素は無条件に正しい式（*expression*）であるとみなすなど，本来行うべき構文チェック処理の一部を省略していたためである．

一方，`multithread-*` の名前の規則セットについては，コードのコピーがなくなったことによって行数が 90%以上削減できており，再利用機構の有効性が裏付けられている．

6 おわりに

我々は，SC 言語処理系を用いて様々な拡張 C 言語を実装し，その過程でいくつかの知見を得ることができた．その一つが，様々な拡張言語の実装に共通して用いることができるような変形（型情報の付与や一時変数のための変数宣言の追加）が存在し，それらは変形規則セットの拡張機構により効果的に再利用できることである．我々は現在でも本システムを，新しい言語のプロトタイピングや言語実装手法のテストのために実際に役立てており，今後も活用していく予定である．

本システムは，拡張 C 言語に限らず，Scheme や Java など他の言語を C 言語への変換によって実装するために用いることもできると考えている．特に，様々な高水準言語の実装に利用するための，ごみ集めを備えた型付中間言語 [14] の開発を本システムを用いて今後取り組んでいく予定である．

謝辞 本研究の一部は，「並列分散計算環境を安定有効活用する要求駆動型負荷分散」（21013027）（科学研究費特定領域研究「情報爆発時代に向けた新しい IT 基盤技術の研究」），科学研究費基盤研究（B）「安全な計算状態操作機構の実用化」（21300008），および科学研究費挑戦的萌芽研究「安全で高速な共通計算基盤のための低水準の型付中間言語」（20650004）の助成を得て行った．

参 考 文 献

[1] GUY L. Steele Jr.: *Common Lisp: The Language,* Second Edition, Digital Press, 1990.

[2] Hiraishi, T., Yasugi, M., Umatani, S., and Yuasa, T.: Backtracking-based Load Balancing, *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2009)*, February 2009, pp. 55–64.

[3] Hiraishi, T., Yasugi, M., and Yuasa, T.: Implementing S-Expression Based Extended Languages in Lisp, *Proceedings of the International Lisp Conference*, Stanford, CA, 2005, pp. 179–188.

[4] Hiraishi, T., Yasugi, M., and Yuasa, T.: A Transformation-Based Implementation of Lightweight Nested Functions, *IPSJ Digital Courier*, Vol. 2(2006), pp. 262–279. (IPSJ Transaction on Programming, Vol. 47, No. SIG 6(PRO 29), pp. 50-67.).

[5] Hiraishi, T., Yasugi, M., and Yuasa, T.: Experience with SC: Transformation-based Implementation of Various Language Extensions to C, *Proceedings of the International Lisp Conference*, Clare College, Cambridge, U.K., 2007, pp. 103–113.

[6] Roudier, Y. and Ichisugi, Y.: Mixin Composition Strategies for the Modular Implementation of Aspect Weaving, 1998.

[7] Roudier, Y. and Ichisugi, Y.: Integrating data-parallel and reactive constructs into Java, *France-Japan Workshop on Object-based Parallel and Distributed Computation*, 1997.

[8] Roudier, Y. and Ichisugi, Y.: Java Data-parallel Programming using an Extensible Java Preprocessor, *IPSJ SIG Notes*, Vol. 97, No. 78(1997), pp. 85–

表 1 MT-SC 言語から LW-SC 言語への変換に関係する変形規則セットの行数および改良後の記述言語における変形規則セットの拡張関係．太字は MT-SC 言語のために新たに作成した変形規則セット．

変形規則セット	従来の記述言語	改良後の記述言語	（拡張元の変形規則セット）
multithread-sc1	**108**	**9**	(sc1-to-sc0)
sc1-to-sc0	100	78	(sc0-to-sc0)
sc0-to-sc0	—	398	(—)
multithread-rename	**164**	**12**	(rename)
rename	152	152	(sc0-to-sc0)
multithread-hoist	**125**	**12**	(hoist)
hoist	114	105	(sc0-to-sc0)
multithread-typeinfo	**332**	**11**	(typeinfo)
typeinfo	321	345	(sc0-to-sc0)
multithread-temp	**265**	**12**	(temp)
temp	255	194	(sc0t-to-sc0t)
sc0t-to-sc0t	—	10	(sc0-to-sc0)
multithread	**263**	**245**	(sc0t-to-sc0t)
untype	22	31	(—)

90.

[9] Stallman, R. M.: *Using the GNU Compiler Collection*, Free Software Foundation, Inc., for gcc-3.2 edition, April 2002.

[10] Umatani, S., Shobayashi, H., Yasugi, M., and Yuasa, T.: Efficient and Portable Implementation of Java-style Exception Handling in C, *IPSJ Digital Courier*, Vol. 2(2006), pp. 238–247. (IPSJ Transaction on Programming, Vol. 47, No. SIG 6(PRO 29), pp. 50–67).

[11] Yasugi, M., Hiraishi, T., and Yuasa, T.: Lightweight Lexical Closures for Legitimate Execution Stack Access, *Proceedings of 15th International Conference on Compiler Construction (CC2006)*, Lecture Notes in Computer Science, No. 3923, Springer-Verlag, 2006, pp. 170–184.

[12] 中田育男, 渡邊坦, 佐々政孝, 森公一郎, 阿部正佳: COINS コンパイラ・インフラストラクチャの開発, コンピュータソフトウェア, Vol. 25, No. 1(2008), pp. 2–18.

[13] 一杉裕志, 田中哲: 差分ベースモジュール：クラス独立なモジュール機構, 技術報告, 2001. (AIST01-J00002-1).

[14] 八杉昌宏: 正確なごみ集めを前提とした低水準の型付中間言語の設計, 第 *9* 回プログラミングおよびプログラミング言語ワークショップ（*PPL'07*）, 2007, pp. 111–122.

[15] 八杉昌宏, 平石拓, 篠原丈成, 湯淺太一: L-Closure：高性能・高信頼プログラミング言語の実装向け言語機構, 情報処理学会論文誌：プログラミング, Vol. 49, No. SIG 1 (PRO 35)(2008), pp. 63–83.

[16] 平石拓, 八杉昌宏, 湯淺太一: 既存 C ヘッダファイルの構文の異なる言語での有効利用, コンピュータソフトウェア, Vol. 23, No. 2(2006), pp. 225–238.

[17] 平石拓, 李暁ろ, 八杉昌宏, 湯淺太一: S 式ベース C 言語における変形規則による言語拡張機構, 情報処理学会論文誌：プログラミング, Vol. 46, No. SIG1(PRO 24)(2005), pp. 40–56.

[18] 田畑悠介, 八杉昌宏, 小宮常康, 湯淺太一: 入れ子関数を利用したマルチスレッドの実現, 情報処理学会論文誌：プログラミング, Vol. 43, No. SIG 3 (PRO 14)(2002), pp. 26–40.