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

平石 拓[†] 李 暁璐[††] 八杉 昌宏[†]
馬谷 誠二[†] 湯淺 太一[†]

現在，実用的システムの開発に C 言語は欠かせないが，細粒度マルチスレッド機能などを追加するために C 言語を拡張するのはそれほど容易ではない．言語拡張の実現方式としては，C コンパイラに手を加えるもの以外に，拡張 C プログラムを変換し，C コードを生成する方式もある．後者の方式では，最初にプログラムを抽象構文木（AST）に変換し，拡張に必要な解析や変形などを行った後，C コード生成を行うことが多い．AST のデータ構造には従来，構造体，オブジェクト指向言語のオブジェクト，直和型のデータなどが用いられているが，本研究では，AST を S 式で表現し，それをそのままプログラムとして用いることを提案する．このため，S 式ベースの構文を持つ C 言語，SC 言語を設計した．SC 言語では新しいコンストラクトの追加が容易であり，また，S 式は，変換時に便利な動的変数も備えた Common Lisp 言語を使って簡単に入出力や解析や変形ができるため，言語拡張の rapid prototyping が可能となる．ただし，Common Lisp 言語ではパターンマッチングを直接書くことができないので，本研究では，backquote マクロの記法をパターン部にも用いた変形規則の表記も提案する．このような SC 言語は他の高水準言語の中間言語として用いることも可能である．本論文では，実際に SC 言語に細粒度マルチスレッド機能を追加した例も示す．

Language Extension by Rule-based Transformation for S-expression-based C Languages

Tasuku Hiraishi,[†] Xiaolu Li,[††] Masahiro Yasugi,[†]
Seiji Umatani[†] and Taiichi Yuasa[†]

The C language is often indispensable for developing practical systems, but it is not so easy to extend the C language by adding a new feature such as fine-grained multithreading. We can implement language extension by modifying a C compiler, but sometimes we can do it by translating an extended C program into C code. In the latter method, we usually convert the source program to an Abstract Syntax Tree (AST), apply analysis or transformation necessary for the extension, and then generate C code. Structures, objects (in object-oriented languages), or variants are traditionally used as the data structure for an AST. In this research, we propose a new scheme where an AST is represented by an S-expression and such an S-expression is also used as (a part of) a program. For this purpose we have designed the SC language, the C language with S-expression-based syntax. This scheme allows rapid prototyping of language extension because (1) adding new constructs to the SC language is easy, (2) S-expressions can easily be read/printed, analyzed, and transformed in the Common Lisp language, which features dynamic variables useful for translation. Since pattern matching cannot be described directly in Common Lisp, we also propose denoting transformation rules with patterns using the backquote-macro notation. Such an SC language can also be used as an intermediate language for other high-level programming languages. This paper also shows a practical example where fine-grained multithreading features are added to the SC language.

1. はじめに

現在，C 言語は実用システムの開発のうえで欠かせないものとなっており，細粒度マルチスレッド機能などを持つ言語の設計が，C 言語の拡張によって行われることも多い．このような拡張の実現において，C コンパイラに直接手を加えるのではなく，拡張 C コードを C コードに変換する方式を用いることができれ

† 京都大学情報学研究科通信情報システム専攻
Department of Communications and Computer Engineering, Graduate School of Informatics, Kyoto University
†† 東京大学情報理工学系研究科電子情報学専攻
Information and Communication Engineering, Graduate School of Information Science and Technology, The University of Tokyo

ば，実装のコストや移植性などの面で有効である．このとき，拡張 C コードを直接変換すると効率が悪いので，コードをまず抽象構文木（AST）に変換してから拡張に必要な解析や変形などを行った後，それを C コードに変換するという方法をとるのが一般的である．AST のデータ構造には，構造体や，オブジェクト指向言語のオブジェクト，直和型のデータなどが用いられているが，本論文では AST を S 式で表現し，その S 式表現をそのままプログラムとして用いることを提案する．

S 式とは以下で定義される，Lisp 言語において一般に利用される文法単位である．

$\langle$S 式$\rangle$ ::= $\langle$原子式$\rangle$|(⟨S 式⟩ . ⟨S 式⟩)

(⟨S 式⟩$_1$⟨S 式⟩$_2$ $\cdots$ ⟨S 式⟩$_n$)

≡ (⟨S 式⟩$_1$. (⟨S 式⟩$_2$. ($\cdots$ (⟨S 式⟩$_n$. nil)) $\cdots$)

提案方式の実現のために我々は，この S 式ベースの構文を持つ C 言語（SC 言語）の設計およびこの言語を扱う処理系（SC 言語処理系）を実装した．拡張言語の実装はこの処理系を利用した C 言語への変換によって行うことができる．S 式は Common Lisp などの Lisp 処理系を用いて容易に変形や入出力を扱えるため，拡張言語の rapid prototyping が可能となる．また，こうしたコード変形規則の記述には S 式のパターンマッチングを用いるのが便利である．そこで本処理系では，backquote マクロの記法をパターン部に用いたパターンマッチングを直接記述できるようにしている．

本論文の構成は以下のとおりである．まず上記の SC 言語処理系の詳細を 2 章で述べる．次に 3 章で実際にこの処理系を用いた，細粒度マルチスレッド言語を C 言語コードへの変換によって実装する例について述べる．4 章では，言語に簡単なコンストラクトを追加する場合について，提案方式を，同じく source to source 変換によって言語拡張を行っている Cilk のコンパイラに手を加える場合と比較し，評価を行う．そして 5 章で関連研究を紹介し，6 章で今後の課題をあげ，最後に 7 章でまとめる．

2. SC 言語処理系

2.1 概　　要

本研究では，S 式のコード変形による言語拡張を実現するために以下の SC 言語処理系を実装した．これは，以下のモジュールから構成される，Common Lisp 処理系[14] 上で動作するシステムである．

- **SC プリプロセッサ**
  ソースコードのトップレベルに記述された `defmacro`，あるいは `include` に従って，マクロの展開や，ファイルの挿入を行う．
- **SC 変換器**
  変換元の S 式（プログラム）と変形規則を入力として受け取り，S 式を与えられた規則に従って変形する．
- **SC コンパイラ**
  SC-0 言語のコードを C 言語のコードに変換する．SC-0 言語は，本処理系の核となる S 式ベースの言語である．2.2 節で詳細を述べる．

本処理系によるコードの変換の流れを**図 1** に示す．S 式ベースの拡張言語で記述されたプログラムは，SC 変換器を通して SC-0 言語のコードに変形される（SC 変換器の各適用の前に SC プリプロセッサによる前処理が実行される）．最終的に出力された SC-0 言語のコードはさらに SC コンパイラ，C コンパイラを通して実行形式に変換される．

拡張言語の実装は，SC 変換器に与える変形規則をプラグインとして記述することによって行う．図 1 に示したように複数の規則を用意し，それらを順に適用

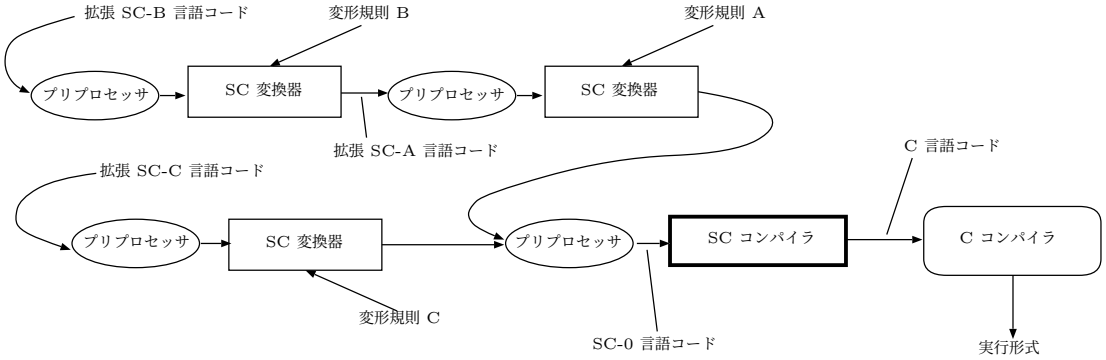


図 1 SC 言語処理系における変換の流れ
Fig. 1 Translation flow in SC-language system.

表 1 SC 言語処理系の各モジュールの行数
Table 1 The number of lines of each SC-language system module.

	行数
SC プリプロセッサ	70
SC 変換器	270
SC コンパイラ	1,600

していくこともできる．

なお，各モジュールの実装にかかった Lisp プログラムの行数を**表 1** に示す．以下，2.2 節で SC-0 言語について，2.3 節で SC プリプロセッサについて詳述し，2.4 節で言語拡張を行うための変形規則について説明する．

2.2 SC-0 言語

本研究では，SC 変換器の最終的な変形先の言語として，以下の特徴を持つ，SC-0 言語を設計した．

- S 式ベースの，Lisp 言語風の構文を持つ．
- ただし意味的には C 言語とほぼ同等である．C 言語で記述可能なコードは，ほぼ 1 対 1 の対応で SC-0 言語でも記述できる（ただし下に述べるとおり，C 言語の一部の構文は，SC-0 には含まれない）．
- 人手によるプログラミングに用いる言語として十分実用的である．

つまり，C 言語の構文を素直に S 式ベースに置き換えたものが SC-0 言語だと考えればよい．ただし C 言語の機能のうち，以下のものは SC-0 には含まれない．

- `->` 演算子
- `for` 構文と `while` 構文（したがって，SC-0 の繰返し構文は `do-while` のみ）

これらの構文，演算子は簡単な変形規則によって，SC-0 言語の表現に置き換え可能であり，またそのような規則を処理系の標準として用意し，その定義ファイルを取り込むことでこれらを利用できるようにしているので，プログラマがこの仕様を意識する必要はない．逆に，このように SC-0 言語の仕様を小さくすることで，変形規則を記述する際に想定しなければならない入力のパターンが少なくなるという利点がある．

SC-0 に上記の構文やその他の簡単な拡張構文を加えた言語を **SC-1 言語**と呼ぶ．SC-1 自体も，本論文の主題である拡張方式で実装している．実際のプログラミングはこの SC-1 言語，あるいはこの言語を変形規則によって拡張した言語で行うことになる．以下，SC-1 言語の構文の各要素について，C 言語と比較しながら説明する．

- 式

C 言語で利用できる演算子はすべて SC-1 言語でも利用可能である．ただし，一部の演算子は記号を変更している．以下に例を示す．

```
 3 + 4 + 5
→ (+ 3 4 5)
 f( x, y[3][4], s->a.b )
→ (f x (aref y 3 4) (fref s -> a b))
 x -= (int)(&b) / *c
→ (-= x (/ (cast int (ptr b)) (mref c)))
 (i<n)?a:b
→ (if-exp (< i n) a b)
```

- 文

上であげた `while`，`for` 構文のほか，`label`，`case`，`default`，`if`，`switch`，`do-while`，`goto`，`continue`，`break`，`return` が C 言語と同様に利用できる．それに加えて，C 言語の複文（{···}）に相当する構文として（`begin` ···），空文に対して `()` を規定している．式文の概念も C 言語と同じである．たとえば，C 言語の

```
for( i=0 ; i<=n ; i++ ){
 if( x < i ){ x++; y++; }
  else return i;
 while( x < y ) x*=2;
}
```

に相当する SC-1 コードは以下のようになる．

```
(for ( (= i 0) (<= i n) (inc i) )
  (if (< x i)
      (begin (inc x) (inc y))
      (return i))
  (while (< x y) (*= x 2)))
```

- 型・型定義

型の表現は C と SC-1 では大きく異なる．C 言語では，`int` 型の配列の宣言は，

```
int a[5];
```

のように型情報が基本型と配列指定子で分離した形になっているが，SC-1 においてはすべての型情報を 1 つの S 式（**型式**，type-expression）で表現する．たとえば上の例に対応する SC-1 のコードは，

```
(def a (array int 5))
```

と書ける．SC-0 ではこれにより，型の記述が読みやすくなっている．C 言語の

```
typedef void *(*(*gg_t)(void *(*)
                (int, int))) (int, int);
```

のような表現は解釈が困難だが，SC-1 で

```
(deftype gg-t
 (ptr (fn (ptr (fn (ptr void) int int))
```

```
        (ptr (fn (ptr void) int int)))))
```

と書くと理解しやすい．ここで，

`(fn (ptr void) int int)` は，

`void *f(int, int);`

のような関数 f の型である．

- 定義・宣言

SC-1 の変数の定義・宣言の構文は，

(*storage-class-specifier identifier type-expression initializer*$_{opt}$)

関数の定義・宣言の構文は多くの場合，

(*storage-class-specifier* (*identifier-list*) (fn *type-expression-list*) *block-item-list*)

である．*storage-class-specifier*（記憶クラス指定子）には，C 言語のそれに相当する `static`，`extern`，`auto` などを指定する．記憶クラスを省略する宣言を表すときは，`def`，`decl` を指定する☆．いくつか例を示す．

```
 int x;
→ (def x int)
 extern y; /* extern int y; */
→(extern y int)
 static long z=x+1;
→ (static z long (+ x 1))
 int f(char x){ return x*x; }
→ (def (f x) (fn int char)
      (return (* x x)))
```

- 構造体・共用体

以下の例に示すとおり，ほぼ C 言語と同じ記法である．

```
 struct sab {
   int a;
   long b;
 };
→ (def (struct sab)
      (def a int)
      (def b long))
```

図 2 に SC-1 言語のコードの例を，それに対応する C 言語のコードを**図 3** に示す．また，SC-1 言語の完全な構文を付録 A.1 に示す．

2.3 SC プリプロセッサ

SC プリプロセッサは，プログラム中に以下の形の S 式があると（`defmacro` はトップレベルのみ），それぞれに対応した処理を実行する．これらの処理は，2.1 節でも示したように，SC 変換器や SC コンパイラの処理に先立って実行される．

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

図 2 SC-1 のプログラム例
Fig. 2 An example SC-1 program.

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

図 3 図 2 に相当する C プログラム
Fig. 3 An equivalent C program for Fig. 2.

- (`include` ⟨ファイル名⟩)

この S 式の位置に，⟨ファイル名⟩で指定されたファイルの中身を挿入する．C 言語の `#include` ディレクティブに相当する．

- (`defmacro` ⟨マクロ名⟩ ⟨ラムダリスト⟩ ⟨S 式⟩$_1$ $\cdots$ ⟨S 式⟩$_n$)

この S 式を Common Lisp の `defmacro` 式として評価しマクロ定義を行う．そして，プログラム中のすべての (⟨マクロ名⟩$\cdots$) の形の S 式に対して，同じく Common Lisp の `macroexpand` 関数によるマクロ展開を実行し，元の S 式をその結果に置き換える．

2.4 変形規則の仕様

SC 変換器に与える変形規則は，以下のような形式の関数として記述する．

(⟨関数名⟩ ⟨パターン⟩ ⟨引数⟩$_1$ $\cdots$ ⟨引数⟩$_n$)
`->` ⟨式⟩

定義された関数は，通常の Lisp 関数として呼び出すことができる．呼び出しが起こると，まず ⟨パターン⟩ と第 1 引数のパターンマッチングを行う．マッチした場合，⟨式⟩ が Common Lisp 処理系によって評価され，その評価値が関数の返り値となる．⟨引数⟩$_2$ 以下を指定した場合，これらは通常の引数と同様に扱われる☆☆．

また，同名の関数を複数定義することができる．こ

☆ 関数や構造体で本体やメンバが空の場合，それが定義か宣言かを `def`，`decl` で区別する．たとえば，`(def (f) (fn void))` は関数定義 `void f(){}`で，`(decl (f) (fn void))` はプロトタイプ宣言 `void f();` である．

☆☆ パターン以外のデータの受渡しは Lisp の動的変数を利用するのが便利なので，実際に第 2 引数以下を利用することは少ない．

の場合，各関数定義に指定されているパターンを定義順に調べていき，最初にマッチした関数定義で指定されている ⟨式⟩ の評価値を返り値とする．

⟨パターン⟩ は以下の要素からなる S 式で表現する．

(1) ⟨シンボル⟩
指定したシンボルと同じシンボルにマッチ．

(2) ,⟨シンボル⟩
任意の 1 要素にマッチする．

(3) ,@⟨シンボル⟩
0 個以上の任意の要素列にマッチ．

(4) ,⟨シンボル⟩[⟨関数名⟩]
⟨関数名⟩ で指定された関数に要素を渡し，返り値が真であった要素にマッチする．

(5) ,@⟨シンボル⟩[⟨関数名⟩]
0 個以上の任意の要素列に対し，各要素を関数名で指定された関数に順に渡し，そのすべての返り値が真であった場合，その要素リストにマッチする．

[⟨関数名⟩] で用いる関数は，変形規則とは別に，Common Lisp の関数定義と同様に定義しておくことができる（処理系の組み込み関数も指定できる）．また，マッチした S 式全体に x という特別な変数がバインドされるほか，(2)〜(5) では実際にマッチした部分の S 式に ⟨シンボル⟩ の名前の変数がバインドされ，-> の右辺の式で使用することができる．

関数の定義例を以下に示す☆．

```
(EX (,a[numberp] ,b[numberp]))
  -> `(,a ,b ,(+ a b))
(EX (,a ,b))
  -> `(,a ,b ,a ,b)
(EX (,a ,b ,@rem))
  -> rem
(EX ,otherwise)
  -> '(error)
```

この関数 EX の適用は以下のようになる．

```
(EX '(3 8))    → (3 8 11)
(EX '(x 8))    → (x 8 x 8)
(EX 8)         → (error)
(EX '(3))      → (error)
(EX '(x y z))  → (z)
```

変形規則は，変形に必要な（一般には複数の）関数を以上のように定義したものである．このうちの関数の 1 つ（最初に呼び出す関数）を，変形したいコード（S 式）に適用することで変形結果が得られる．

☆ パターンは `(,a[numberp] ,b[numberp]) のように，リストの先頭にも backquote を書く記法のほうが，backquote マクロの評価とパターンマッチングの対称性を考えると適切である．しかし，そのようにすると規則の記述の際，ほぼすべてのパターンの先頭に backquote を書かなければならないという不都合が生じる．本研究では記述の効率を優先し，本文中のような記法を採用した．

3. SC 言語処理系による言語拡張例

本研究では，SC 言語処理系を用いた言語拡張の 1 つとして，SC-0 言語に細粒度マルチスレッド機能を追加した言語（MT-SC 言語）の実装を行った．拡張言語の仕様や C 言語レベルでのマルチスレッドの実装方式は，文献 15) で提案されているものを用いた．

以下で言語仕様と，実際にどのような変形が必要かについて説明するが，スケジューリングの方式やマルチスレッドの実装方式の正当性など詳細については，上記の文献を参照していただきたい．

3.1 拡張言語の仕様

MT-SC のマルチスレッド機能において，スレッドは以下のような仕様を持つ．

- スレッドは活性状態（active），あるいは中断状態（suspended）のいずれかの状態をとる．活性状態のスレッドが実際にプロセッサにスケジューリングされたときに，スレッドは進行する．
- スレッドは実行時に活性状態で生成され，実行を開始する．
- 生成時に実行すべきコードが与えられ，実行が終了したスレッドは消滅する．
- 活性状態のスレッドは自らを中断（suspend）し，中断状態になることができる．この際，自分のスレッドの継続（continuation）を保存する．
- 他のスレッドは，上記の継続を用いて，その中断状態のスレッドを再開（resume）し，活性状態にすることができる．

以上の仕様を満たすため，MT-SC には，SC-0 に加えて以下のプリミティブを追加する．

- (thread-create *declaration-list* *statement-list*)
  declaration-list で指定された変数を定義，宣言した環境で *statement-list* を実行するスレッドを生成する．
- (thread-suspend *identifier* *statement-list*)
  変数 *identifier* に現在の継続を保存した状態で，*statement-list* を実行し，スレッドを中断する．
- (thread-resume *expression*)
  中断状態のスレッドの継続を *expression* にとり，そのスレッドを再開させる．

MT-SC のプログラム例として，Fibonacci 関数を計算する例のための関数定義を**図 4** に示す．

```
(def (pfib n) (fn int int)
  (def x int) (def y int)
  (def nn int 0) (def c cont 0)
  (if (<= n 2)
      (return 1)
      (begin
        (thread-create ()
          (= x (pfib (- n 1)))
          (if (== (++ nn) 0)
              (thread-resume c)))
        (= y (pfib (- n 2)))
        (if (< (-- nn) 0)
            (thread-suspend c0
                            (= c c0)))
        (return (+ x y)))))
```

図 4 MT-SC のプログラム例
Fig. 4 An example MT-SC program.

3.2 マルチスレッドの実装方式

細粒度マルチスレッド機能を C 言語レベルで実現するために，文献 15) では GNU C コンパイラ (GCC)[13] が C 言語の拡張機能として提供している入れ子関数を利用している．入れ子関数とは関数の中に定義される関数のことである．入れ子関数の内部では関数内の局所変数だけでなく，その親関数の変数にもアクセスできる．この仕様を実現するために，GCC では入れ子関数のポインタに実行コードのアドレスだけではなく，入れ子関数が定義された時点における環境の情報も含ませている．このような入れ子関数のポインタを用いると，スタックの途中にある関数フレームのローカル変数を参照するクロージャをスタックトップで呼び出すことができる．そこで，各関数に自分と同等の計算を行う入れ子関数を持たせれば，入れ子関数のポインタをスレッドの継続として利用できる．このことを利用し，MT-SC のコードを以下の方針で C 言語のプログラムに変換することにより，MT-SC を実装する．

- スケジューリングおよびスレッド再開用の関数や，その他の補助関数，型の定義を新たに追加する．また，スレッド管理スタックをグローバルな配列として追加する．
- MT-SC の関数内に，自身と同等の計算を行う入れ子関数，関数の実行がどこまで終わったかを記憶するためのラベル変数を追加するなどの変形を施す．また，関数呼び出しの際にスレッドの継続の受渡しができるように，関数の定義，呼び出しの際の引数を追加する．
- `thread-create`，`thread-suspend`，および `thread-resume` はそれぞれの処理用に定義した C コードの断片に展開する．
- そのほか，関数呼び出しの前に上記のラベル変数が更新されるようにする，入れ子関数の中では `return` を適切な処理に置き換えるなどの変換が必要である．

3.3 変形規則

以上で述べた変形を，本提案方式に従い，MT-SC のプログラムから SC-0 のプログラムへの変形規則によって実装する．ただし，変形規則の記述を簡潔にするため，また変形規則の一部を再利用できるように，この変形は 1 つの変形規則ではなく，以下の 4 つの変形規則を順に適用していくことによって行う．

(1) プログラム中のすべての式（*expression*）に，その式の型情報を明示的に付与する変形規則（**type 規則**）．
(2) プログラム中の関数呼び出し式が部分式として現れないようにする変形規則（**temp 規則**）．
(3) 3.2 節の変形を実際に行う規則（**multithread 規則**）．
(4) (1) で付与した型情報を外し，SC-0 言語のコードに変形する規則（**untype 規則**）．

以下，これらの 4 つの規則について説明する．

3.3.1 type 規則

temp 規則や multithread 規則の変形を実行するには，プログラム中の式が何の型を持つかという情報が必要である．type 規則で定義する変形によって，そのような型情報をプログラム中に明示的に付与する．具体的には以下のような変形を行う．

- プログラム中に現れるすべての *expression* を `(the` *type-expression expression* `)` の形にする．
- 関数呼び出しの先頭に `call` を挿入する（これはこの先の変形規則の記述をしやすくするための便宜上の変形であり，必ずしも必要ではない）．

図 5 に変形規則の一部を示す．コードの先頭に定義されている `Tp0` を最初に呼び出すことで変形が行われる（これ以降に示す規則でも同様に，コードの先頭に定義されている関数が最初に適用する関数である）．`Tp1` で，定義された変数の情報を追加し，その情報を元に `Tpe` で式に型情報を追加する変形を行っている．変数の有効範囲は，図の例のように動的変数を用いて管理すると比較的簡単に扱うことができる．

図 6，**図 7** にそれぞれ type 規則による変形前，変形後のコードの例を示す．

3.3.2 temp 規則

プログラム中にたとえば，`(f (g x))` のような形の関数呼び出し式が存在すると，multithread 規則で `g` の関数実行完了直後の継続が扱いにくい．temp 規則は，このように部分式中に関数呼び出しが現れないよ

```
(Tp0 (,@declaration-list) )
-> (progn
    ...   中略   ...
    (let (*str-alist* *v-alist* *lastv-alist*)
     (mapcar #'Tp1 declaration-list)))
;;;;; declaration ;;;;;
(Tp1 (,scs[SC-SPEC] ,id[ID] ,texp ,@init))
-> (progn
    (push (cons id (remove-type-qualifier texp))
          *v-alist*)
     `(,scs ,id ,texp ,@(mapcar #'Tpi init)))
(Tp1 (,scs[SC-SPEC] (,@id-list[ID])
      (fn ,@texp-list) ,@body))
-> (let* ((texp-list2
           (mapcar #'rmv-tqualifier texp-list))
          (*v-alist* (cons (cons (first id-list)
                           `(ptr (fn ,@texp-list2)))
                              *v-alist*))
          (new-body nil))
    (let ((b-list
           (cmpd-list (cdr id-list)
                      (cdr texp-list2))))
     (setq new-body
      (let((*v-alist* (append b-list *v-alist*))
           (*str-alist* *str-alist*))
        (mapcar #'Tpb body))))
    `(,scs (,@id-list)
      (fn ,@texp-list),@new-body))
...   中略   ...
(Tp1 ,otherwise)
-> (error "sytax error")
;;;;;;; body ;;;;;;
(Tpb (do-while ,exp ,@body))
-> (switch ,(Tpe exp)
     ,@(let ((*v-alist* *v-alist*)
             (*str-alist* *str-alsit*))
         (mapcar #'Tpb body)))
...   中略   ...
(Tpb ,otherwise)
-> (let ((expression-stat (Tpe otherwise)))
     (if (eq '$not-expression expression-stat)
         (Tp1 otherwise)
       expression-stat))
;;;;; expression ;;;;;
(Tpe ,id[ID])
-> `(the ,(assoc-vartype id) ,id)
...   中略   ...
(Tpe (ptr ,exp))
-> (let ((exp-with-type (Tpe exp)))
      `(the (ptr ,(cadr exp-with-type))
        (ptr ,exp-with-type)))
(Tpe (mref ,exp))
-> (let* ((exp-with-type (Tpe exp))
          (exp-type (cadr exp-with-type)))
     `(the ,(deref-type exp-type)
       (mref ,exp-with-type)))
(Tpe (,fexp[EXPRESSION] ,@arg-list))
-> (let* ((fexp-with-type (Tpe fexp))
          (fexp-type (cadr fexp-with-type))
          (type-fn (cadr fexp-type)))
     `(the ,(cadr type-fn)
       (call (the ,type-fn
              ,(caddr fexp-with-type))
        ,@(mapcar #'Tpe arg-list))))
(Tpe ,otherwise)
-> '$not-expression
```

図 5 type 規則（一部）
Fig. 5 Part of type rules.

```
(def (g x) (fn int int)
  (return (* x x)))

(def (f x) (fn double double)
  (return (+ x x)))

(def (h x) (fn char double)
  (return (f (g x))))
```

図 6 type 規則の変形例（変形前）
Fig. 6 An example transformation by type-rule (before).

```
(def (g x) (fn int int)
  (return (the int
              (* (the int x) (the int x)))))

(def (f x) (fn double double)
  (return (the double
          (+ (the double x) (the double x)))))

(def (h x) (fn char double)
  (return (the double
           (call (the (fn double double) f)
            (the int (call (the (fn int int) g)
                           (the double x)))))))
```

図 7 type 規則の変形例（変形後）
Fig. 7 An example transformation by type-rule (after).

```
(def (g x) (fn int int)
  (return
     (the int
          (+ (the int
                  (= (the int x) (the int 3)))
                  (the int
                     (call (the (fn int int) g)
                     (the int x)))))))
```

図 8 temp 規則の変形例（変形前）
Fig. 8 An example transformation by temp-rule (before).

うに変形を行う．また，同様の理由で `return` の返り値の式，`if` の条件式中などにも関数呼び出しは現れないようにする．さらに，そのような変形を行うために必要になる一時変数（temporary variables）の宣言を関数本体の先頭に追加する．

たとえば**図 8** のコードはこの規則によって**図 9** のコードに変形される．

図 10 に，temp 規則の一部を示す．実際の式の変形は `Tmpe` で行っているが，この返り値は

- 新たに追加する必要がある変数宣言のリスト
- 式の直前に挿入する代入式のリスト
- 当該の式の位置に代わって置き換える式

の 3 つ組になっていることに注意する．この返り値の情報を `Tmp2` や `Tmp` の `begin` の処理中などで結合し，適切なコードを出力する．

3.3.3 multithread 規則

temp 規則の変形結果のコードを，3.2 節で述べた方針に従って，マルチスレッド機能を実現するプログラムに変形する．

```
(def (g x) (fn int int)
  (def tmp1 int)
  (def tmp2 int)
  (the int
       (= (the int tmp1)
       (the int
            (= (the int x) (the int 3)))))
  (the int
       (= (the int tmp2)
          (the int
               (call (the (fn int int) g)
               (the int x)))))
  (return
    (the int
        (+ (the int tmp1) (the int tmp2)))))
```

図 9 temp 規則の変形例（変形後）
Fig. 9 An example transformation by temp-rule (after).

変形規則の一部を**図 11** に示す．先に述べたとおり，生成する入れ子関数はその親関数と同等の計算を実行するが，両者の本体のコードは完全には同じではない．そこで，`M0` 以外の関数の返り値は，「入れ子関数の本体用のコード」と「親関数の本体用のコード」の組とし，これを `M1` の関数定義の処理中などのときに結合する．

3.3.4 untype 規則

multithread 規則によって得られた出力には，type 規則で付与した型情報，関数呼び出し式に付与した `call` が含まれている．このコードをそのまま SC コンパイラに入力することはできないので，untype 規則によってそれらを取り除き，正しい SC-0 言語のコードに変形する．この変形は，先頭が `the` あるいは `call` であるリストを再帰的に検索して該当するリストから取り除くだけでよい．**図 12** に変形規則を示す．

3.3.5 ま と め

最後に，図 4 のプログラムに type，temp，multithread，untype の各規則を順に適用して得られたコードを付録 A.2 に示す．

また，各変形規則の行数を**表 2** に示す☆．ここで type，temp，untype の規則は他の言語拡張の際に部品としても再利用が可能であり☆☆，またその機会は多い．

また S 式を用いているため，変形規則のテストは簡単に行うことができる．たとえば，temp 規則への入力は S 式で簡単に作れ，その出力のチェックも簡単にできる．

☆ multithread 規則の行数は補助関数などの定義を含む．
☆☆ ただし，6.1 節で述べる問題点がある．

```
(Tmp0 (,@decl-list))
->(progn
    ...  中略  ...
    (let ((*used-id* (get-all-id x))
          (*prev-continue* nil))
      (mapcar #'Tmp1 x)))
;;;; declaration ;;;;;
(Tmp1 (,scs[SC-SPEC]
       (,@id-list[ID]) (fn ,@texp-list) ,@body))
-> (let* ((tmpbody (Tmp2 body))
          (newdecl (first tmpbody))
          (newbody (second tmpbody)))
    `(,scs (,@id-list)
      (,fntag ,@texp-list) ,@newdecl ,@newbody))
...  中略  ...
;;;;; body ;;;;;
(Tmp2 (,@item-list))
-> (let* ((tmpitemlist (mapcar #'Tmp item-list))
          (decl-list (apply #'append
                   (mapcar #'first tmpitemlist)))
          (prev-stat (apply #'append (mapcar
       #'(lambda (x) `(,@(second x) ,(third x)))
                              tmpitemlist))))
     (list decl-list prev-stat))
(Tmp (do-while ,exp ,@body))
->(let* ((tmpexp (Tmpe exp))
         (*prev-continue* (second tmpexp))
         (tmpbody (Tmp2 body)))
   (list (append (first tmpexp) (first tmpbody))
         nil
    (do-while ,(third tmpexp)
         ,@(second tmpbody) ,@*prev-continue*)))
(Tmp (return ,@exp))
-> (if (null exp)
       `(nil nil (return))
        (let ((tmpexp (Tmpe (car exp))))
          `(,(first tmpexp) ,(second tmpexp)
            (return ,(third tmpexp)))))
...  中略  ...
(Tmp ,otherwise)
->(let ((tmpe-exp (Tmpe otherwise)))
    (if (eq '$not-expression tmpe-exp)
        (list (list (Tmp1 otherwise)) nil)
        tmpe-exp))
;;;;; expression ;;;;;
(Tmpe (the ,texp (call ,fexp ,@arg-list)))
-> (case texp
     ((void)
      ...  中略  ...  )
 (otherwise
 (let*
 ((tmpexps (comb-list (mapcar #'Tmpe arg-list)))
  (tempid (generate-id "tmp"))
  (tmp-decl1 `(def ,tempid ,texp))
  (tmp-decl
   (append (first tmpexps) `(,tmp-decl1)))
  (tmp-set1 `(the ,texp (= (the ,texp ,tempid)
   (the ,texp (call ,fexp ,@(third tmpexps))))))
  (tmp-set
   (append (second tmpexps) `(,tmp-set1))))
 (list tmp-decl tmp-set `(the ,texp ,tempid)))))
(Tmpe (the ,texp (+ ,exp1 ,exp2)))
-> (let ((op (caaddr x))
    (t-exp1 (Tmpe exp1)) (t-exp2 (Tmpe exp2)))
 (list `(,@(first t-exp1) ,@(first t-exp2))
       `(,@(second t-exp1) ,@(second t-exp2))
       `(the ,texp (,op ,(third t-exp1)
                        ,(third t-exp2)))))
...  後略  ...
```

図 10 temp 規則（一部）
Fig. 10 Part of temp rules.

```
(M0 (,@declaration-list))
->(progn
 (defstruct finfo decl-list id label nfunc-name)
 ...   中略   ...
 ;;; スケジューリングなどの補助関数の宣言を include
 (cons '(include "multithread.sh")
  (let ((*used-id* (get-all-id x))
        (*cfunc* nil))
    (apply #'append
           (mapcar #'second (mapcar #'M1 x))))))
;;;;; declaration ;;;;;
(M1 (,scs[SC-SPEC] (,func-id ,@arg-id[ID])
     (fn ,func-texp ,@arg-texp) ,@body))
-> (let* ((*cfunc* (make-finfo
      :nfunc-name (generate-id func-id) :ln 0))
     (nestfunc-def (make-nestfunc (first temp)))
     ...   中略   ...)
    (list nil
    `((,scs (,func-id c_p ,@arg-id)
      (fn ,func-texp cont ,@arg-texp)
      ,@prev ,nestfunc-def ,@new-body ,@post))))
...   中略   ...
;;;;; body ;;;;;
(M2 (,@item-list))
-> (let* ((itemlist (mapcar #'Mb item-list))
     (nest-body (append-first itemlist))
     (parent-body (append-second itemlist)))
    (list nest-body parent-body))
;;;;; statement ;;;;;
(Mb (thread-suspend (the ,texp ,id) ,@body))
->(let* ((nfunc-name
          (finfo-nfunc-name *cfunc*))
         (m2-body (M2 body))
         (lbl-l (generate-id "@L")))
   (push (cons lbl-l nil) (finfo-label *cfunc*))
   (push `(def ,id thst_ptr)
          (finfo-decl-list *cfunc*))
   (list
    `((begin  (= ,id cp)
       (= (fref ,id -> c) ,nfunc-name)
       (= (fref ,id -> stat) thr_suspended)
       ,@(first m2-body) (= ln ,ln) (return)))
    `((begin  (= ,id (inc thst_top))
       (= (fref ,id -> c) ,nfunc-name)
       (= (fref ,id -> stat) thr_new_suspended)
       ,@(second m2-body)
       (= ln ,ln) (scheduling)))))
...   中略   ...
;;;;; expression ;;;;;
(Me (the ,texp (call (the
         (fn ,fexp1 ,@fexp) ,f-id) ,@arg-list)))
->(let* ((nfunc-name
          (finfo-nfunc-name *cfunc*))
    (label-l (generate-id "@L"))
    (incln-exp `(= ln ,ln))
    (call-ret  `(the ,texp (call (the
     (fn ,fexp1 (fn (ptr void) thst_ptr reason)
      ,@fexp) ,f-id) ,nfunc-name ,@arg-list)))
    (label-info `(label ,label-l nil)))
   (if (eq `void texp)
      ...   中略   ...
    (progn
     (push (cons label-l ln)
           (finfo-label *cfunc*))
     (list
       `(( ,incln-exp ,call-ret ,label-info))
       `(( ,incln-exp ,call-ret )))))))
...   中略   ...
```

図 11 multithread 規則（一部）
Fig. 11 Part of multithread rules.

```
(UTp0 ,decl-list)
-> (UTp decl-list)
(UTp (the ,texp ,exp))
-> (Utp exp)
(UTp (call ,@exp-list))
-> (mapcar #'Utp exp-list)
(UTp (,@lst))
-> (mapcar #'UTp lst)
(UTp ,otherwise)
-> otherwise
```

図 12 untype 規則
Fig. 12 Untype rules.

表 2 各変形規則の行数
Table 2 The number of lines of each transformatin-rule.

	行数
type 規則	450
temp 規則	340
multithread 規則	430
untype 規則	10

```
(def (f x) (fn int int)
  (label L1
   (do-while (> x 0)
     (do-while (<= x 10)
       (inc x)
       (if (< x 5)
           (continue L1)
           (break L1))
       (= x (g x)))
     (= x (h x))))
  (return x))
```

図 13 ラベル付き break と continue（SC-0）
Fig. 13 A labeled break/continue (in SC-0).

4. 評　　価

本提案方式による言語拡張の実装コストを評価するために，簡単な言語拡張について，SC-0 言語に変形規則によって拡張する場合と，Cilk[3] に同様の言語拡張を実装する場合を比較する．Cilk は，C 言語に細粒度マルチスレッド機能を追加した言語で，source to source の変換によって実装が行われている．今回の比較では，この変換プログラム（Cilk2C）のコードに手を加えることで Cilk の言語拡張を行う．

4.1 拡張言語の仕様

評価のために行う言語拡張として，Java 言語で利用可能なラベル付き `break`，`continue` を，SC-0 や Cilk でも記述可能にする．

拡張言語のプログラム例を**図 13**（SC-0），**図 14**（Cilk）に示す．ここでたとえば，図 14 中の `break L1` によって外側の `do-while` ループを脱出，`continue L1` によってこのループの次の反復に（条件式を評価した後）進むようにする．

```
int f( int x )
{
L1:
  do{
    do{
      x++;
      if( x>5 ) continue L1;
       else  break L1;
      x = g(x);
    } while (x <= 10);
    x = h(x);
  } while (x > 0);
  return x;
}
```

図 14 ラベル付き break と continue（Cilk）
Fig. 14 A labeled break/continue (in Cilk).

```
int
f (int x)
{
L1:
  do{
    do{
      x++;
      if (x < 5)  goto L12;
       else goto L13;
      x = g (x);
    } while (x <= 10);
    x = h (x);
    L12: ;
  } while (x > 0);
L13: ;
  return x;
}
```

図 15 ラベル付き break，continue の C 言語による実装
Fig. 15 Implementation of labeled breaks and continues in C.

4.2 変換方式

上記のようなラベル付き break，continue を C 言語への変換によって実装するには，それぞれのジャンプ先に新たにラベルを置き，各命令をそのラベルへの goto に置き換えればよい．図 13，図 14 のコードの場合，**図 15** のようなコードに変換する．

4.3 実装と比較

上記の言語拡張を実装するための変形規則は**図 16** のように記述できる．

同じ拡張を Cilk でも行った．Cilk から C の変換は，Cilk のコードをいったん構造体で表現した AST に直してから，解析，変形を行い C コードを出力するという方法をとっているため，AST の変形を行うコードに手を加えた．またそれに加え，元の言語に対しラベル付き break，continue の記法を許すという構文拡張を行っているため，AST を表現する構造体の定義や構文解析器（yacc ファイル）にも手を加える必要があった．

両者の実装コストを，手を加えたファイル数と行数

```
(LBC0 (,@declaration-list))
-> (progn
     (defvar *used-id*)
     (defvar *label-list*)
     (let ((*used-id* (get-all-identifier x))
           (*label-list* nil))
       (LBC1 declaration-list)))

(LBC1 (label ,id[ID] (do-while ,exp ,@body)))
(LBC1 (label ,id[ID] (switch ,exp ,@body)))
-> (let ((*label-list*
     (cons
      `(,id
        ,(generate-id
          (par-identifier id *used-id*))
        ,(generate-id
          (par-identifier id *used-id*)))
      *label-list*)))
     `(begin (label ,id
              (,(car (third x)) ,exp
               ,@(LBC1 body)
               (label ,(second
                        (first *label-list*))
                      nil)))
       (label ,(third (first *label-list*))
              nil)))
(LBC1 (continue ,id[ID]))
(LBC1 (break ,id[ID]))
-> (let ((label-tuple
          (car (member (par-identifier id)
                       *label-list*
                       :key #'(lambda (x)
                                (par-identifier (first x)))
                       :test #'string=))))
     (unless label-tuple
       (error "label ~s is undefined." id))
     `(goto ,(funcall
              (case (car x)
                ((continue) #'second)
                ((break) #'third))
              label-tuple)))
(LBC1 (,@list))
-> (mapcar #'LBC1 list)
(LBC1 ,otherwise)
-> otherwise
```

図 16 ラベル付き break，continue を実現するための変形規則
Fig. 16 The transformation rule for labeled breaks and continues.

表 3 ラベル付き break，continue の実装コストの比較
Table 3 Comparison of implementation costs for labeled breaks and continues.

	ファイル数	行数
SC 処理系	1	45
Cilk	9	112

によって比較したものを**表 3** に示す．Cilk のほうが手を加えた行数が多いほか，それ以上に，構文拡張による影響が多くのモジュール（ファイル）に分散して現れるため，それらすべての箇所を確認し，手を加えなければならないことによる手間が大きい．それに対して SC 処理系を利用した拡張では，変形を 1 カ所にまとめて記述できるため，実装は簡単に行える．

5. 関連研究

5.1 既存の言語拡張方式

Cilk[3] や OpenMP[8] のように，既存の言語を拡張した有用な実装は数多く存在する．しかしこれらのほとんどは，それぞれの特定の拡張のみを想定しており，様々な言語拡張のための一般的な枠組みを提供しているわけではない．

Lisp や Scheme 言語はプログラム自身が AST としての性質を持つ言語で，同時にプログラミング言語としての記述のしやすさも兼ね備えている．KCL[20]，SCHEME->C[5] のような C 言語へのコンパイラの実装も存在するが，これらの言語は，C 言語レベルの低レベルな処理の記述には適さない．

5.2 リフレクション

本来のリフレクション（自己反映計算）とは，実行中のプログラムが，それ自身に関するメタレベルの情報を具現化されたデータとして参照・変更することで自分自身の機能を動的に拡張したりするものである．拡張性の面で非常に強力ではあるが，インタプリタから情報を得るといった実現では，遅いインタプリタ実行となってしまう．このため，たとえばオブジェクト指向言語では，メソッド呼び出しにおいてのみメタレベルの情報を得て，メタオブジェクトで処理するといった方式[2] がとられることが多い．しかしこれには，拡張が行える場所がメソッド呼び出しに限られるなどの制限があった．

一方，OpenC++[1] や OpenJava[16] などでのコンパイル時リフレクションでは，拡張をコンパイル時のプログラム変換で実現する．コンパイル時リフレクションの一種である Java EPP[10)~12)] もこのような変形を利用して，言語拡張を行う．変形規則はプラグインとして記述するほか，構文拡張もこのプラグインによって比較的簡単に記述できる．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，AST そのものでプログラミングするわけではないという点で，本提案方式とは異なる．

後述のアスペクト指向に関しても同様だが，このようなコンパイル時リフレクションの処理を，SC 処理系の変形によって実現することも可能である．提案方式はそれに加えて，本論文で示した MT-SC 言語から SC-0 言語への変形のような，より本格的なコード変形を行うための機能を提供している．

5.3 アスペクト指向言語

アスペクト指向[6] 言語（AOP）は，プログラム中に分散した 1 つの関心事を 1 カ所にまとめて書けるようにするための枠組みを与えている．これは結合点（join point）と呼ばれる，コード中に適切に定義されたポイントにアスペクトとして定義されたメソッド呼び出しを自動的に埋め込むことによって実現されている．SC 処理系でも，適当な S 式の解析，変形を行う変形規則を記述することによってそのような枠組みを実装することが可能である．

5.4 XML を利用した変形

コード変形，解析のために用いる内部表現としては，S 式以外に XML[17] が利用されることも今後増えていくと思われる[4]．XML 上の変形規則は XSLT（XSL Transformation）[18] で記述できる．テンプレートルールでは，マッチするノードのパターンを指定する．変換元のノードや子ノードなどを参照するにはパターン変数ではなく，XPath を用いる．

XML も S 式と同様，解析，変形に適したデータ構造だが，人手によるプログラミングを行うには複雑な構文を持っており，本研究の目的のために利用するのには向いていない．

5.5 S 式のパターンマッチング

SC 変換器に与える変形規則のように S 式のパターンマッチングを直接記述できるようにする方式は，文献 9) など数多く存在する．本研究のパターンマッチの機能のうち，`,`⟨シンボル⟩ や `,`⟨シンボル⟩`[`⟨関数名⟩`]` に相当するものは，これらの多くが備えているが，`,@`⟨シンボル⟩ および `,@`⟨シンボル⟩`[`⟨関数名⟩`]` に相当するものは一般的ではない．

またパターン変数の記法は，たとえば `,`⟨シンボル⟩ に対応するものとしては `?`⟨シンボル⟩ などと書く場合が多いが，本研究では backquote マクロの評価とパターンマッチングの操作の対称性に着目し，より直観的である backquote マクロの記法を採用している．

5.6 S 式ベースの C 言語

本研究以外の S 式ベースの C 言語の実装としては，sexpc[7] がある．だがこれは，現時点では言語の設計・実装のみで完結しており，本研究における言語拡張方式などのような応用は提案されていない．

6. 今後の課題

今後の課題としては以下の点があげられる．

6.1 変形規則の独立性

提案方式において複数の規則を適用する際，後から適用される規則によって言語拡張がされていると，それに応じて先に適用される規則の定義を書き換えなければならないことがある．たとえば，最初 SC-0 言語用に記述した type 規則をその拡張言語である

MT-SC 言語に適用しようとすると，言語拡張によって thread-create などの構文が追加されていることを考慮して，

```
(Tpb (thread-create ,@body))
-> (thread-create
      ,@(let ((*v-alist* *v-alist*)
              (*str-alist* *str-alist*))
          (mapcar #'Tpb body)))
```

などの定義を type 規則に明示的に追加する必要がある（図 5 では省略されている）．このことは，特に個々の言語拡張を独立に開発する場面では大きな問題となる．

複数の言語拡張の完全な独立設計を可能にするのは困難である．だが少なくとも，変形規則が，ある程度の構文拡張などに対しては柔軟に対応できるようにすることは実用上必要である．そのため，変形規則の適用時にその変形規則に対して，必要な規則の差分を追加できるようにする仕組みを SC 変換器に組み込む予定である．

ただしこの場合，そのような差分の定義は後に適用される変形規則の設計者が行うことになるので，その設計者が先に適用される変形規則の内部仕様をある程度知っておかなければならないという問題が残る．そこで，一般的な構文拡張などに対しては差分情報を適当なパラメータとして受け取ることで変形規則自身が言語の変化に対応できるようにする，あるいはそのようなパラメータから変形規則を自動生成する変換器を実装する☆，などの方式を検討している．

6.2 既存の C ライブラリとの共存

これは，SC-0 言語で記述したプログラムを単に SC コンパイラで C 言語に変換するだけの場合は特に問題にならない．だが，変形規則を適用する際には，C 言語で書かれた関数のプロトタイプ宣言や外部変数などの情報が必要となることがあり，現段階の実装ではこれらの情報を SC 変換器による変形時に取り込むことはできない．安直な解決方法としては，C のヘッダファイルを人手で SC-0 に書き直すことが考えられるが，手間がかかりすぎる．そこで，C 言語から SC-0 言語への（半）自動変換ツール☆☆を実装することが考えられる．このとき，C プリプロセッサを適用した後のコードを変換すると，#define で定義された定数名などの情報を取り込めなくなるので，プリプロセッサ文も含めて相当する SC-0 言語の記述に変換する必要がある．

6.3 Lisp 処理系への依存

本処理系における S 式の変形は，大部分を Lisp 処理系の機能に頼っている．このことは，自由度の高い変形を許す反面，変形規則の記述に動的変数の利用など十分な Lisp の知識を要求し，一般のプログラマに対する規則記述の敷居を高くしている．この点に関しては，SC 変換器の機能を強化し，規則記述を支援するマクロや関数など Lisp のライブラリを充実させることで，一般的な手続きの記述を単純にする（複雑な変形は Lisp の能力を駆使して記述させる）などの解決策が考えられる．

6.4 デバッグの問題

元の SC 言語のプログラムに変形・変換を施すと，最後の C コードと元の SC コードの対応関係が分かりにくくなり，デバッグツールなどによるデバッグが行いにくくなるという問題が生じる．改善策としては，SC コンパイラが，出力する C 言語に #line ディレクティブを埋め込むなどが考えられる．ただし，変形規則によってコードが大幅に変形される場合などは，それだけでは不十分であり，拡張の実装者がデバッグのことも考慮した変形規則を記述するなどの対処が必要になる．

6.5 XC-cube をバックエンドとして利用

我々は並列処理の実現に適した言語として，並行協調拡張 C 言語 XC-cube [19] を設計・開発している．これを C 言語の代わりに，SC 処理系のバックエンドとして利用することで，より高性能な高水準並列処理言語の実装を行うことを目指している．

6.6 パターンマッチを利用した SC コンパイラ

現在の SC コンパイラの実装は，通常の Lisp のプログラムで行っているが，SC 変換器の出力部分を少し改変すればパターンマッチを用いた記述による実装も可能である．このような実装にしておくことで，上の XC-cube の利用のように，バックエンドとなる言語の変更に対して容易に対応できるようになる．

7. おわりに

本論文では，C 言語を S 式表現の AST で表現し，それをそのままプログラミング言語として利用し，S 式の変形によって言語拡張を行う手法を提案した．またそのような拡張を行うための処理系の実装を示し，実際の拡張例と評価について述べた．この手法によっ

☆ そのような変換器の実装には，現在の SC 変換器がほぼそのまま利用できる．

☆☆ この変換を完全に自動化することはできない．たとえば，(tx)(x) は，(cast tx x)（型キャスト）か (tx x)（関数呼び出し）か自動で判断できない．しかし，このような曖昧な部分の選択を人間に問い合わせるような，半自動化ツールであれば実用的なものとして実装可能であろう．

て，言語拡張を容易に行うことが可能になった．

一方，本処理系を用いることで，従来の C 言語のプログラミングを S 式によって行うことが可能になる（あるいは強制される）という側面もある．これが受け入れられない状況では，通常の C 言語などと似たの構文の言語の構文解析器を S 式ベースの拡張言語を AST として実装すればよい．そのような場合でも，rapid prototyping のために提案手法を用いることは有用である．

参 考 文 献

1) Chiba, S.: A Metaobject Protocol for C++, *ACM SIGPLAN Notices*, Vol.30, No.10, pp.285–299 (1995).
2) Chiba, S. and Masuda, T.: Designing an Extensible Distributed Language with a Meta-Level Architecture, *Proc. European Conference on Object Oriented Programming* (*ECOOP*), LNCS 707, pp.482–501 (1993).
3) Frigo, M., Leiserson, C.E. and Randall, L.: The Implementation of the Cilk-5 Multithreaded Language, *ACM SIGPLAN Notices*, Vol.33, No.5, pp.212–223 (1998).
4) 川島勇人，権藤克彦：ACML に基づくプログラム情報抽出システムの設計，日本ソフトウェア科学会第 20 回大会論文集 (2003).
5) Joel, F.B.: SCHEME->C a Portable Scheme-to-C Compiler, *WRL Research Report* (1989).
6) Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J.: Aspect-Oriented Programming, *Proc. European Conference on Object-Oriented Programming*, Akşit, M. and Matsuoka, S. (Eds.), Vol.1241, pp.220–242, Springer-Verlag, Berlin, Heidelberg and New York (1997).
7) Mastenbrook, B.: sexpc — s-expression to C translation (2004).
http://www.common-lisp.net/project/sexpc/
8) OpenMP Architecture Review Board: OpenMP — Simple, Portable, Scalable SMP Programming.
http://www.openmp.org
9) Queinnec, C.: Compilaation of Non-linear, Second Order Patterns on S-expressions, *PLILP'90*, LNCS 456, pp.340–357 (1990).
10) Roudier, Y. and Ichisugi, Y.: Integrating data-parallel and reactive constructs into Java, *Proc. France-Japan Workshop on OBPDC'97* (1997).
11) Roudier, Y. and Ichisugi, Y.: Java Data-parallel Programming using an Extensible Java Preprocessor, *Swopp'97* (1997).
12) Roudier, Y. and Ichisugi, Y.: Mixin Composition Strategies for the Modular Implementation of Aspect Weaving, *Aspect-Oriented Programming Workshop at ICSE'98* (1998).
13) Stallman, R.M.: Using and Porting GNU Compiler Collection (1999).
14) Steele Jr., G.L.: *Common Lisp: The Language*, Second Edition, Digital Press (1990).
15) 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), pp.26–40 (2002).
16) Tatsubori, M., Chiba, S., Killijian, M.-O. and Itano, K.: OpenJava: A Class-based Macro System for Java, *Reflection and Software Engineering*, Cazzola, W., Stroud, R.J. and Tisato, F. (Eds.), LNCS 1826, pp.119–135, Springer-Verlag (2000).
17) W3C Architecture Domain: Extensible Markup Language (XML).
http://www.w3.org/XML/
18) W3C Recommendation: XSL Transformations (XSLT) Version1.0 (1999).
http://www.w3.org/TR/xslt
19) 八杉昌宏，高田　潤，田畑悠介，小宮常康，湯淺太一：共有メモリ向けプリミティブとその GCC を使った実現，情報処理学会論文誌：プログラミング，Vol.43, No.SIG 1 (PRO 13), pp.118–132 (2002).
20) Yuasa, T.: Design and Implementation of Kyoto Common Lisp, *Journal of Information Processing*, Vol.13, No.3, pp.284–293 (1990).

付　　録

A.1 SC-1 言語の構文

A.1.1 External Definitions

translation-unit:
  external-declaration
  translation-unit external-declaration

external-declaration:
  declaration

A.1.2 Declarations

declaration-list:
  declaration
  declaration-list declaration

declaration:
  inlined-declaration
  (*identifier type-expression initializer*$_{opt}$)
  (*function-identifier* (`fn` *function-type-list*)
   [`:attr` *function-attribute*]$_{opt}$
   register-declarator$_{opt}$
   block-item-list$_{opt}$)

(*struct-or-union-specifier*
struct-declaration-list$_{opt}$)
(*enum-specifier enumerator-list*)

inlined-declaration-list:
inlined-declaration
inlined-declaration-list declaration

inlined-declaration:
(*storage-class-specifier identifier*
type-expression initializer$_{opt}$)
(*storage-class-specifier function-identifier*
(`fn` *function-type-list*)
[`:attr` *function-attribute*]$_{opt}$
register-declarator$_{opt}$*block-item-list*$_{opt}$)
(*def-or-decl struct-or-union-specifier*
struct-declaration-list$_{opt}$)
(`def` *enum-specifier enumerator-list*)
(*compound-storage-class-specifier*
type-expression init-declarator-list)
(`deftype` *identifier type-expression*)
(`deftype` *identifier struct-or-union*
struct-declaration-list$_{opt}$)
(`deftype` *identifier* `enum` *enumerator-list*)

function-identifier:
identifier
(*identifier-list*)

def-or-decl:
`def`
`decl`

init-declarator-list:
init-declarator
init-declarator-list init-declarator

init-declarator:
identifier
(*identifier initializer*)

storage-class-specifier: one of
`def decl extern extern-def extern-decl`
`static static-def`
`auto auto-def register register-def`

compound-storage-class-specifier: one of
`defs extern-defs static-defs`
`auto-defs register-defs`

function-attribute:
`inline`

register-declarator:
(`register` *identifier-list*)

struct-declaration-list:
struct-declaration
struct-declaration-list struct-declaration

struct-declaration:
declaration [`:bit` *expression*]$_{opt}$

enumerator-list:
enumerator
enumerator-list enumerator

enumerator:
enumeration-constant
(*enumeration-constant expression*)

enmuration-constant:
identifier

identifier-list:
identifier
identifier-list identifier

initializer-list:
initializer
initializer-list initializer

initializer:
expression
compound-initializer

compound-initializer:
(`array` *initializer-list*)
(`struct` *initializer-list*)

A.1.3 Type-expressions

type-expression:
type-specifier
(*type-qualifier-list type-expression*)
(`array` *type-expression*
array-subscription-list$_{opt}$)
(`ptr` *type-expression*)
(`fn` *function-type-list*)

function-type-list:
type-expression-list `va-arg`$_{opt}$

type-expression-list
type-expression
type-expression-list type-expression

type-specifier: one of
`void`
`char signed-char unsigned-char`
`short signed-short unsigned-short`
`int signed-int unsigned-int`
`long signed-long unsigned-long`
`long-long signed-long-long`
`unsigned-long-long`
`float double long-double`
struct-or-union-specifier
enum-specifier
typedef-name

array-subscription-list:
expression-list

struct-or-union-specifier:
(*struct-or-union identifier*)

struct-or-union:
`struct`
`union`

enum-specifier:
(`enum` *identifier*)

type-qualifier-list:
type-qualifier
type-qualifier-list type-qualifier

type-qualifier:
`const`
`restrict`
`volatile`

typedef-name:
identifier

A.1.4 Statements

statement:
  compound-statement
  expression-statement
  selection-statement
  iteration-statement
  jump-statement
  labeled-statement
  `()`

compound-statement:
  `(begin` *block-item-list*$_{opt}$`)`
  `(let (`*declaration-list*$_{opt}$`)` *block-item-list*$_{opt}$`)`

block-item-list:
  block-item
  block-item-list block-item

block-item:
  inlined-declaration
  statement

labeled-statement:
  `(label` *identifier statement* `)`
  `(case` *expression* `)`
  `(default)`

expression-statement:
  expression

selection-statement:
  `(if` *expression statement statement*$_{opt}$`)`
  `(switch` *expression block-item-list*$_{opt}$`)`

iteration-statement:
  `(while` *expression block-item-list*$_{opt}$`)`
  `(do-while` *expression block-item-list*$_{opt}$`)`
  `(for (`*expression-list*$_{opt}$ *expression expression*`)`
   block-item-list$_{opt}$`)`
  `(for`
   `(`*inlined-declaration-list*$_{opt}$
      expression expression `)`
   block-item-list$_{opt}$`)`
  `(loop` *block-item-list*$_{opt}$`)`

jump-statement:
  `(goto` *identifier* `)`
  `(continue)`
  `(break)`
  `(return` *expression*$_{opt}$`)`

A.1.5 Expressions

expression:
  identifier
  constant
  string-literal
  `(`*expression-list*`)`
  `(aref` *expression-list* `)`
  `(fref` *expression field-identifier-list* `)`
  `(inc` *expression* `)`
  `(dec` *expression* `)`
  `(++` *experssion* `)`
  `(--` *expression* `)`
  `(`*unary-operator expression* `)`
  `(sizeof` *expression* `)`
  `(sizeof` *type-expression* `)`
  `(cast` *type-expression expression* `)`
  `(`*operator expression-list* `)`
  `(`*comparator expression expression* `)`
  `(if-exp` *expression expression expression* `)`
  `(`*assignment-operator expression expression* `)`
  `(exps` *expression-list* `)`

expression-list:
  expression
  expression-list expression

field-identifier-list:
  field-identifier
  field-identifier-list field-identifier

field-identifier:
  identifier
  `->` *identifier*

operator: one of
  `* / % + - << >>`
  `bit-xor bit-and bit-or and or`

comparator: one of
  `< > <= >= == !=`

assignment-operator: one of
  `= *= /= %= += -= <<= >>=`
  `bit-and= bit-xor= bit-or=`

unary-operator: one of
  `ptr mref bit-not not`

A.2 変形規則による図 4 のプログラムの SC-0 プログラムへの変形結果

```
(decl (struct _thstelm))
(deftype reason enum rsn_cont rsn-retval)
(deftype cont
  (ptr (fn (ptr void)
   (ptr (struct _thstelm)) reason)))
(def (enum t-stat)
  thr_new_suspended thr_new_runnable
  thr_suspended thr_runnable thr_scheduled)
(def (struct _thstelm)
  (def c cont)
  (def stat (enum t-stat)))
(deftype thstelm (struct _thstelm))
(deftype thst_ptr (ptr thstelm))
(def thst (array thstelm 65536))
(def thst_top thst_ptr thst)
(deftype schdexit (ptr (fn void)))
(def cur_schd_exit schdexit 0)
(def cur_schd_thst_top thst_ptr thst)

(def (scheduling) (fn void)
 (def @l0 __label__)
 (def prev_exit schdexit cur_schd_exit)
 (def prev_thst_top thst_ptr cur_schd_thst_top)
 (def mythst_top thst_ptr thst_top)
 (def (nonlocalexit) (fn void) (goto @l0))

 (label @l0 nil)
 (= cur_schd_exit nonlocalexit)
 (= cur_schd_thst_top thst_top)
 (= cur_schd_thst_top mythst_top)
  (do-while 1
   (begin
    (def cp thst_ptr)
     (begin
      (= cp prev_thst_top)
      (if (< cp mythst_top)
          (do-while (exps (inc cp)
                          (< cp mythst_top))
            (if (!= (fref (mref cp) stat)
                    thr_scheduled)
                (break)))))
```

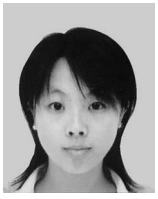
```
      (if (== cp mythst_top)
          (if prev_exit (prev_exit))))
    (begin
     (def cp thst_ptr)
     (def cc cont)
     (begin
      (= cp (- thst_top 1))
      (if (>= cp thst)
       (do-while (exps (dec cp) (>= cp thst))
        (if (or (== (fref (mref cp) stat)
                    thr_runnable)
                (== (fref (mref cp) stat)
                    thr_new_runnable))
            (break)))))
     (if (< cp thst) (exit 0))
     (do-while (== (fref (mref cp) stat)
                   thr_runnable)
      (= cc (fref (mref cp) c))
      (= (fref (mref cp) c) 0)
      (= (fref (mref cp) stat) thr_scheduled)
      (cc cp rsn_cont)))
    (if (and (> thst_top thst)
             (== (fref (mref (- thst_top 1))
                       stat)
                 thr_new_runnable))
        (begin
         (dec thst_top)
         (break))))
   (= cur_schd_exit prev_exit)
   (= cur_schd_thst_top prev_thst_top))

(def (thr_resume cp) (fn void thst_ptr)
 (if (== (fref (mref cp) stat) thr_suspended)
     (= (fref (mref cp) stat) thr_runnable)
     (if (== (fref (mref cp) stat)
             thr_new_suspended)
         (= (fref (mref cp) stat)
            thr_new_runnable))))

(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 #:|tmp2| int)
 (def c0 thst_ptr) (def t2 int) (def t1 int)

 (def (pfib_c cp rsn)
      (fn (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 t2))))
    (return))
  (return)
  (if (<= n 2)
      (begin
       (= (mref (cast (ptr int)
                      (c_p cp rsn_retval)))
          1))
      (begin
       (begin
        (begin
         (= ln 1)
         (= (fref thst_top -> c) pfib_c)
         (= (fref thst_top -> stat)
            thr_new_runnable)
         (inc thst_top))
        (begin
         (def #:|tmp| int)
         (def ln int 0)
         (def (nthr_c cp rsn)
              (fn (ptr void) thst_ptr reason)
          (switch rsn
           (case rsn_cont)
            (switch ln
             (case 1) (goto @l1))
            (return)
           (case rsn_retval)
            (switch ln
             (case 1)
             (return (cast (ptr void)
                           (ptr t1))))
            (return))
          (return)
          (= ln 1)
          (= t1 (pfib nthr_c (- n 1)))
          (label @l1 nil) (= #:|tmp| t1)
          (= x #:|tmp|)
          (inc nn)
          (if (== nn 0) (thr_resume c))
          (return))
         (= ln 1)
         (= #:|tmp| (pfib nthr_c (- n 1)))
         (= x #:|tmp|)
         (inc nn)
         (if (== nn 0) (thr_resume c)))
        (if (!= (fref (- thst_top 1) -> stat)
                thr_new_runnable)
            (scheduling)
            (dec thst_top))
        (label @l1 nil)
        (= ln 2)
        (= t2 (pfib pfib_c (- n 2)))
        (label @l2 nil)
        (= #:|tmp2| t2)
        (= y #:|tmp2|)
        (= nn (- nn 1))
        (if (< nn 0)
            (begin
             (= c0 cp)
             (= (fref c0 -> c) pfib_c)
             (= (fref c0 -> stat) thr_suspended)
             (= c c0)
             (= ln 3)
             (return)))
        (= (mref (cast (ptr int)
                       (c_p cp rsn_retval)))
           (+ x y)))))
       (label @l3 nil)
       (= (fref cp -> c) c_p)
       (= (fref cp -> stat) thr_runnable)
       (return))

      (if (<= n 2) (begin (return 1))
          (begin
           (begin
            (begin
             (= ln 1)
             (= (fref thst_top -> c) pfib_c)
             (= (fref thst_top -> stat)
                thr_new_runnable)
             (inc thst_top))
            (begin
             (def #:|tmp| int)
             (def ln int 0)
             (def (nthr_c cp rsn)
                  (fn (ptr void) thst_ptr reason)
              (switch rsn
```

```
    (case rsn_cont)
     (switch ln
      (case 1) (goto @l1))
     (return)
    (case rsn_retval)
     (switch ln (case 1)
             (return (cast (ptr void)
                           (ptr t1))))
     (return))
   (return)
   (= ln 1)
   (= t1 (pfib nthr_c (- n 1)))
   (label @l1 nil) (= #:|tmp| t1)
   (= x #:|tmp|) (inc nn)
   (if (== nn 0) (thr_resume c))
   (return))
  (= ln 1)
  (= #:|tmp| (pfib nthr_c (- n 1)))
  (= x #:|tmp|)
  (inc nn)
  (if (== nn 0) (thr_resume c)))
 (if (!= (fref (- thst_top 1) -> stat)
         thr_new_runnable)
     (scheduling)
     (dec thst_top))
 (= ln 2)
 (= #:|tmp2| (pfib pfib_c (- n 2)))
 (= y #:|tmp2|)
 (= nn (- nn 1))
 (if (< nn 0)
     (begin
      (= c0 (inc thst_top))
      (= (fref c0 -> c) pfib_c)
      (= (fref c0 -> stat)
         thr_new_suspended)
      (= c c0)
      (= ln 3)
      (scheduling)))
 (return (+ x y))))))
```

(平成 16 年 3 月 30 日受付)
(平成 16 年 8 月 25 日採録)

平石　拓

1981 年生．2003 年京都大学工学部情報学科卒業．同年より同大学大学院情報学研究科通信情報システム専攻修士課程に在学中．言語処理系，並列処理に興味を持つ．日本ソフトウェア科学会学生会員．

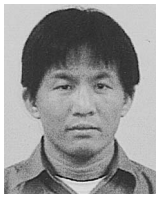


李　暁璐

1981 年生．2004 年京都大学工学部情報学科卒業．同年より東京大学大学院情報理工学系研究科電子情報学専攻修士課程に在学中．コンピュータビジョン，三次元画像処理に興味を持つ．

八杉　昌宏（正会員）

1967 年生．1989 年東京大学工学部電子工学科卒業．1991 年同大学大学院電気工学専攻修士課程修了．1994 年同大学院理学系研究科情報科学専攻博士課程修了．1993 年〜1995 年日本学術振興会特別研究員（東京大学，マンチェスター大学）．1995 年神戸大学工学部助手．1998 年京都大学大学院情報学研究科通信情報システム専攻講師．2003 年より同大学助教授．博士（理学）．1998 年〜2001 年科学技術振興事業団さきがけ研究 21 研究員．並列処理，言語処理系等に興味を持つ．日本ソフトウェア科学会，ACM 各会員．



馬谷　誠二（正会員）

1974 年生．1999 年京都大学工学部情報学科卒業．2001 年同大学大学院情報学研究科修士課程修了．2004 年同大学院情報学研究科博士後期課程修了．2004 年より産学官連携研究員として同大学院に在籍．博士（情報学）．並列/分散処理，プログラミング言語処理系に興味を持つ．日本ソフトウェア科学会，ACM 各会員．

湯淺　太一（フェロー）

1952 年神戸生．1977 年京都大学理学部卒業．1982 年同大学大学院理学研究科博士課程修了．同年京都大学数理解析研究所助手．1987 年豊橋技術科学大学講師．1988 年同大学助教授，1995 年同大学教授，1996 年京都大学大学院工学研究科情報工学専攻教授．1998 年同大学院情報学研究科通信情報システム専攻教授となり現在に至る．理学博士．記号処理，プログラミング言語処理系，並列処理に興味を持っている．著書『Common Lisp 入門』（共著），『C 言語によるプログラミング入門』，『コンパイラ』ほか．日本ソフトウェア科学会，電子情報通信学会，IEEE，ACM 各会員．