情報処理学会論文誌　プログラミング　Vol.6　No.2　13–32 (Aug. 2013)

L-Closureの呼び出しコストの削減

田附 正充[1,†1]　八杉 昌宏[2,a)]　平石 拓[3]　馬谷 誠二[1]

受付日 2012年11月12日, 採録日 2013年4月24日

概要：本論文では L-closure という言語機構に関して 2 つの実装を提案する．1 つは複数回同じ L-closure が呼び出された場合などの呼び出しコストの削減，もう 1 つは GNU C コンパイラ（GCC）4 系列への元々呼び出しコストの低い closure の実装である．L-closure は入れ子関数定義を評価すると生成される軽量レキシカルクロージャで，これを持つ拡張 C 言語を，高水準言語コンパイラの中間言語として採用することで，ごみ集めなどの高水準サービスを効率良く実装できる．現在 L-closure の実装として，GCC 3 系列の拡張によるコンパイラ実装と標準 C 言語への翻訳による実装があり，これらは L-closure の初期化処理を遅延したり，低い維持コスト（アクセスされる変数のレジスタ割当て）を実現しているが，呼び出しコストは高い．翻訳による実装では，C のスタックとは別のスタック（明示的スタック）を用意し，L-closure 呼び出し時に C のスタックの内容を明示的スタックへ一時的に移すことで，入れ子関数を持つ関数の局所変数へのアクセスを実現していた．本研究では，L-closure からのリターンの後，C スタック全体を再構築するのではなく，フレームごとに再構築することで，再度 L-closure が呼び出されたときのスタック間の値の移動を減らし，呼び出しコストを削減する．L-closure の研究の一環として，生成/維持コストはかかるものの呼び出しコストが低い closure も提案してきた．本研究ではまた，高度な最適化のために内部構造が刷新された GCC 4 系列において，これを実装したので報告する．

キーワード：入れ子関数，L-closures，実行スタック，コンパイラ，トランスレータ

Reducing Invocation Costs of L-Closures

Masami Tazuke[1,†1]　Masahiro Yasugi[2,a)]　Tasuku Hiraishi[3]　Seiji Umatani[1]

Received: November 12, 2012, Accepted: April 24, 2013

Abstract: This paper proposes two new implementations for a language mechanism called “L-closures.” First, we reduce the invocation costs of L-closures especially when an L-closure is called multiple times. Second, we implement an associated language mechanism called “closures” with inherently low invocation costs in GNU C Compiler (GCC) version 4. L-closures are lightweight lexical closures created by evaluating nested function definitions. By using intermediate languages extended with L-closures in high-level compilers, we can implement high-level services such as garbage collection efficiently. As existing implementations of L-closures, we have a compiler-based implementation as an enhanced GCC version 3, and a transformation-based implementation as a translator into standard C. In these existing implementations, the initialization of an L-closure is delayed until the L-closure is actually invoked, and low maintenance costs of L-closures (register allocation for accessed variables) are realized; however, we accept high invocation costs. In the existing translator, we enable L-closures to access local variables of the enclosing function, by preparing an explicit stack in C other than the C (execution) stack and by temporarily moving contents of the C stack into the explicit stack upon every invocation of an L-closure. In the first new implementation of this study, we reduce invocation costs by restoring the C stack frame by frame rather than restoring the entire C stack after returning from the L-closure, so that we can eliminate value movements between the stacks upon the subsequent invocations of the L-closure. In our study of L-closures, we have proposed “closures” to enable low invocation costs by accepting moderate initialization/maintenance costs. In this study, we also report the second new implementation, in which we implement closures in GCC version 4 whose internal structure has been reformed for advanced optimizations.

Keywords: nested functions, L-closures, execution stacks, compilers, translators

1. はじめに

様々な種類の計算機（マシン）について優れたコード生成器を実装するのは手間がかかる作業である．このため，高水準言語向けコンパイラの作成においては，ある程度移植性が良くマシン独立な中間言語としてC言語を使うことも多い．つまり，高水準言語からC言語への翻訳系（トランスレータ）のみを開発するのである．

Cプログラムをコンパイルして得られる機械語プログラムの多くは実行効率を考慮し，実行スタックを使う．関数呼び出しの際にはスタックフレームが割り当てられ，関数のパラメータや局所変数のほか，戻り番地，1つ前のフレームポインタ，calleeセーブレジスタなどのスペースのために使われる．高性能・高信頼プログラミング言語に備わってる高水準実行時サービスの中には，ごみ集め，自己デバッグ，スタックトレース，チェックポインティング，マイグレーション，継続，マルチスレッド，負荷分散などのように，その効率良いサポートのためには実行スタックの内容を見たり変更したりする必要があるものも多い．しかし，C言語では，関数呼び出し中に呼び出されたほうの関数は，呼び出したほうの関数の局所変数に効率良くアクセスすることはできない．そのような局所変数のいくつかはcalleeセーブレジスタに値をとるようにできるかもしれないのだが，**ポインタ**に基づくアクセスはそのようなコンパイラの最適化技法の邪魔となる．さらに，スタックフレームのレイアウトはマシン依存であり，実行中のプログラム自身による偽造ポインタによる直接的スタック操作は本質的には不正である．実際，スタックフレームのデータはアプリケーションレベルのデータではなく実行のためのメタデータであるし，そのような不正なアクセスはセキュリティ上の問題にもつながる．

たとえばごみ集め（GC）を実装するためには，コレクタはすべての**ルート**を見つけられなくてはならない．各ルートは，ごみ集めされるヒープ中のオブジェクトへの参照を保持する．Cでは，呼び出し元におけるポインタ変数がオブジェクトへの参照を保持しているかもしれないが，それは，実行スタック中に眠ってしまっているかもしれない．たとえ直接的スタック操作を許すとしても，コレクタがスタック中のルート（参照）を他の要素から区別するのは難しい．**スタック上のマップ**というものが使えるかもしれないが，それは本来のCのデータでもないし，その準備にはコンパイラによる特別なサポートを必要とする．よって，通常，保守的コレクタ [2] がいくつかの制約下において使われることになる．一方，コピーGCを正しく実装するためには，コレクタは正確にすべてのルートをスキャンする必要がある．オブジェクトは空間の間を移動させられるため，すべてのルートポインタはオブジェクトの新しい場所を参照すべきだからである．正確なコピーGCは「構造体とポインタ」に基づく翻訳手法 [8], [9] によっても実現できるが，局所変数を構造体のフィールドへと翻訳すると，多くのコンパイラ最適化技法は使えなくなる．

このような問題を解決するために，強力で移植性の高い新しい中間言語が研究されている．たとえば，C-- [18], [20] は，（Cより低水準にあたる）移植性の高いアセンブリ言語であり，「スタック歩き」するための実行時システムを提供することで，実行スタック中に眠る変数へのアクセスを可能としている．これによりC-- は，ごみ集めを含む高水準実行時サービスのいくつかを実現するための中間言語として適している．

我々は，このような中間言語の設計に役立つ言語機構，“L-closure” の研究，提案 [13], [29], [31], [33] を行っている．L-closureは入れ子関数定義を評価すると生成される軽量lexical closureであり，lexical closureは生成時環境におけるlexicalスコープの変数にアクセスできるため，その間接的な呼び出しにより合法的なスタックアクセスの手段を提供する．我々が提案しているL-closureを取り入れたC言語 [29], [31], [33] では，C-- と比較し，よりすっきりと高水準サービスがサポートされるだけでなく，Cコンパイラ拡張で実装する場合，既存のコンパイラの大部分のモジュールやリンカなどの関係ツールを再利用することで，低コストでの処理系実装が可能となる．

L-closureは，その生成コストや維持コストを積極的に削減するという方針を採用しており，呼び出しコストは犠牲にしてよいとしている．

現在L-closureの実装方式として，Cコンパイラを拡張する方式と，標準的なC言語へと翻訳するトランスレータ方式の研究を行っている．それぞれすでに発表に至っている実装例があり，前者はGNU Cコンパイラ（GCC）3.4.6を拡張した実装 [33]，後者はS式ベースC言語（SC言語）処理系を利用した「L-closureを持つ拡張SC言語LW-SC」の変換ベースの実装 [13] である．SC言語処理系 [10], [12], [35], [36] とは，我々が提案している変換ベースによる言語拡張を支援するシステムであり，利用者は「変形規則セット」を追加することで，拡張SC言語を標準的なC言語へと変換可能なSC-0言語へと変換することができる．

本論文では，L-closureに関して実装を新たに2つ提案する．1つは複数回同じL-closureが呼び出された場合な

[1] 京都大学大学院情報学研究科
Graduate School of Informatics, Kyoto University, Sakyo, Kyoto 606–8501, Japan
[2] 九州工業大学大学院情報工学研究院
Department of Artificial Intelligence, Kyushu Institute of Technology, Iizuka, Fukuoka 820–8502, Japan
[3] 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, Sakyo, Kyoto 606–8501, Japan
[†1] 現在，楽天株式会社情報技術部
Presently with IT Department, Rakuten, Inc.
[a)] yasugi@ai.kyutech.ac.jp

どの呼び出しコストの削減を実現する実装，もう 1 つは GCC 4 系列への元々呼び出しコストの低い closure の実装である．前者は上述の LW-SC の変換ベースの実装と同じく SC 言語処理系を利用し実装を行った．以前の LW-SC の変換ベースの実装では，C のスタックとは別のスタック（明示的スタック）を用意し，L-closure の呼び出し時に C のスタックの内容を明示的スタックへと一時的に移すことで，入れ子関数を持つ関数の局所変数へのアクセスを実現していた．この方式の問題点は，L-closure の呼び出しが起こるたびに C のスタックの内容を明示的スタックへと移す操作や，L-closure からリターンするときに明示的スタックに移していた内容を C のスタックへと積み直す操作が発生し，L-closure の呼び出しが頻繁に起こるプログラムではオーバヘッドが生じてしまい，プログラム全体の実行時間が大きくなってしまう点である．そこで，本研究で提案する実装では，一度 L-closure の呼び出しが起こり C のスタックの内容が明示的スタックに移され，L-closure の実行が終了しリターンするときに，明示的スタックに移していた内容をすべて C のスタックへと積み直すのではなく，L-closure を呼び出した関数のフレームのみを C のスタックへと戻し実行を再開することで，再度 L-closure の呼び出しがあった場合のスタック間の移動操作を減らし，L-closure の生成コストや維持コストを増やすことなく，呼び出しコストを削減する．

後者は，L-closure の研究の一環として提案している “closure” に関する実装である．closure は，生成・維持コストはかかるものの呼び出しコストを低くした言語機構である．たとえば，L-closure は実際に呼び出しが起こるまで初期化が遅延されるのに対し，closure では遅延を行わない．Closure は，すでに GCC 3 系列に実装・発表済み [29], [31], [33] であるが，本研究では高度な最適化のために内部構造が刷新された GCC 4 系列においてこれを実装したので報告する．

本研究の貢献は，L-closure の実用性を高めることができたという点にある．L-closure は生成・維持コストのために，呼び出しコストを犠牲にしてよいとしているが，実用において，L-closure の呼び出しが少ないプログラムであっても大きなオーバヘッドが生じてしまうのであれば，使用することがはばかられる．既存の L-closure の実装モデルと比べ，呼び出しコストを抑えることで，実用性が増したといえる．また，GCC 4 系列への closure の実装を行うことでより新しいコンパイラで利用することができ，より進化した最適化技術のもとで利用することができる．

以下，2 章ではサンプルプログラムを使い L-closure について説明する．また，既存の L-closure および closure の実装についても説明する．3 章では提案する L-closure の変換ベース実装の実装モデルについて述べる．4 章では SC 言語処理系に基づく変換ベースの実装について述べる．5 章では closure の GNU C コンパイラ 4 系列へのコンパイラベースの実装について述べる．6 章では L-closure の複数の実装について性能評価，考察を行う．7 章では関連研究について述べる．8 章では 3 章で述べた実装モデルの応用について議論する．9 章ではまとめおよび今後の課題を述べる．

2. L-Closure：安全な計算状態操作機構

2.1 概要

ある高水準言語のプログラムにおいて，再帰的に 2 分探索木のノードをトラバースし，対応する探索データを持つ連想リストを作成するものとしよう．そのような高水準言語プログラムは，**図 1** に示すような C プログラムへと翻訳されるとする．ここで，`getmem` は新しいオブジェクトをヒープ中に割り当てるものとし，コピー GC のコレクタが，`x`，`rest`，`a` や `kv` といったルートとなる変数をすべてスキャンできなくてはならないとする．もちろん，`bin2list` が再帰的に呼び出されているような状況も考える．

計算状態操作機構 L-closure [13], [29], [31], [33] を用いると，コピー GC をともなうプログラムを**図 2** のように簡潔にすっきりとした表現で書くことができる．メモリアロケータ `getmem` は，入れ子関数定義を評価して生成される L-closure `scan1` を引数にとり，これを使うコピー型コレクタを起動するかもしれない．つまり，コピー型コレクタは `scan1` を間接呼び出しすることでルート（`x`，`rest`，`a`，`kv`）をスキャンしてオブジェクトの移動を行うとともに，さらに，入れ子状に L-closure `scan0` の間接呼び出しを行う[*1]．`scan0` の実体は呼び出し元における `scan1` の別のインスタンスかもしれない．スタックの底に達するまで L-closure の呼び出しを繰り返すことで実行スタック全体についてすべてのルートがスキャンできる．

図 2 において，`bin2list` の変数（`x`，`rest`，`a`，`kv`）は（callee セーブ）レジスタが割り当てられる可能性を持ってほしいが，よくある Pascal スタイル（静的リンクを用いて外側の関数の変数へアクセス）の実装を L-closure の実装として用いると `bin2list` におけるこれらの変数へのアクセスにレジスタ操作よりも遅いメモリ操作が必要になってしまう．というのは，`scan1` もまた，通常，静的リンクを通してスタックメモリ中のこれらの変数の値へとアクセスするためである．先に述べた「構造体とポインタ」に基づく翻訳手法 [8], [9] でのスタック歩きでも同じ問題が起こる．

そこで，L-closure は実装方針として，このような L-closure の**維持**コストを削減すること，すなわち，これらの変数へのレジスタ割当てを可能とすることを我々の目標とする．同時に，実装方針として，L-closure の**生成**コストも削減す

[*1] あるいは，`scan1` が `scan0` をリターンすることで，末尾呼び出しを除去することも考えられる．

```
Alist *bin2list(Bintree *x, Alist *rest){
  Alist *a = 0; KVpair *kv = 0;
  if(x->right) rest = bin2list(x->right, rest);
  kv = getmem(&KVpair_d);                /* allocation */
  kv->key = x->key; kv->val = x->val;
  a = getmem(&Alist_d);                  /* allocation */
  a->kv = kv; a->cdr = rest;
  rest = a;
  if(x->left) rest = bin2list(x->left, rest);
  return rest;
}
```

図 1　サンプルプログラム：木–リスト変換
Fig. 1　A motivating example: tree-to-list conversion.

```
typedef void *(*move_f)(void *);

                /* scan0 is an L-closure pointer. */
Alist *bin2list(void (*scan0) lightweight (move_f),
                Bintree *x, Alist *rest){
  Alist *a = 0; KVpair *kv = 0;
  void scan1 lightweight (move_f mv){ /* create L-closure */
    x = mv(x); rest = mv(rest);       /* scan roots */
    a = mv(a); kv = mv(kv);           /* scan roots */
    scan0(mv);                        /* scan older roots */
  } /* pass pointer to L-closure "scan1" on the following calls. */
  if(x->right) rest = bin2list(scan1, x->right, rest);
  kv = getmem(scan1, &KVpair_d);    /* allocation */
  kv->key = x->key; kv->val = x->val;
  a = getmem(scan1, &Alist_d);      /* allocation */
  a->kv = kv; a->cdr = rest;
  rest = a;
  if(x->left) rest = bin2list(scan1, x->left, rest);
  return rest;
}
```

図 2　L-closure（入れ子関数）での GC ルートスキャン
Fig. 2　Scanning GC roots with L-closures (nested functions).

る．一方で，L-closure の**呼び出し**コストは高くなっても構わないものとする．ごみ集めにおけるルートスキャンのためなど多くの高水準サービスにおいて，L-closure は頻繁に生成されつつ，たまにしか呼び出されないため，総合的なオーバヘッドをかなり削減することができる．

2.2　C コンパイラ拡張による closure の実装の実装モデル

C コンパイラ拡張による実装を行う場合に推奨される closure の実装モデル [29], [33] について述べる．

我々は，L-closure の研究の一環として closure という言語機構の提案・研究を行っている．L-closure が実装方針として，呼び出しコストを犠牲にしても生成・維持コストを積極的に削減するとしているのに対し，closure は生成・維持コストはかかるものの呼び出しコストを低くするという方針を取っている．

本実装モデルでは，局所変数定義に実行点が至ると（論理的には）その変数の場所が作られるのと同様に，入れ子関数定義に実行点が至ると，入れ子関数本体とその環境のペアである lexical closure を（論理的に）生成される．入れ子関数は生成時の lexical スコープの変数にアクセスでき，lexical closure へのポインタは，lexical closure を間接呼び出しするために利用できる．lexical closure は変数の場所と同様にスタック上に作られるため，ごみ集めのある言語と異なり，lexical closure へのポインタを入れ子関数定義のあるブロックの実行完了後は使うことができない．

また，入れ子関数を通常のトップレベルの関数とは意味上別個に扱う．これにより，入れ子関数に関しては異なる呼び出し列を用いて性能改善を結び付けることができる．

closure は，入れ子関数本体とその環境（静的リンク）のペアとして実装できる．closure ポインタについてはこのペアを指すようにすればよい．closure ポインタを使って間接呼び出しをするときには，コンパイル時にすでに通常の関数ポインタとは区別されているため，コンパイルされた呼び出し列としては，静的リンク（ペアの後者）をセットしてから入れ子関数本体（ペアの前者）を呼び出すようにする．

2.3　C コンパイラ拡張による L-closure の実装の実装モデル

L-closure についても，前節の closure の実装と同じように実装することができる．ただし，L-closure の実装方針に従い生成コストを最小化するため，L-closure の初期化（ペアの初期化）は実際に呼び出されるまで遅延される．つま

表 1　用語
Table 1　Terminology.

変換前のプログラム	変換後のプログラム
関数	変換後関数
入れ子関数	変換後入れ子関数
リターン	return
呼び出し	call

り，生成コストは事実上 0 とすることになる．

また L-closure の維持コストを最小化するために，もし，関数 f が L-closure 型の入れ子関数 g を所有していて g は f の局所変数（もしくはパラメータ）x にアクセスするのであれば，x は場所を 2 つ用いることにする．具体的には，共有のための場所と，レジスタ割当て候補として私用の場所を準備する．そして，その間の一貫性維持を遅延させる．具体的には，g を実際に呼び出すまで私用の場所から共有のための場所へ保存を遅延させる．また，g を呼び出した場合に限り f にリターンするときに共有のための場所から私用の場所へ反映する．

このように closure と L-closure の言語機構は違う特徴を持っているので，実際のプログラムに対しては，状況に応じて適切な方を用いればよい．

2.4　従来の変換ベース実装の実装モデル

従来の標準的な C 言語へと翻訳する変換に基づく L-closure の実装モデル [13] について述べる．以下，変換前のプログラムの話であるか，変換後の話であるかを明確にするため，**表 1** のように用語を用いる．

実装モデルの概要は以下のようにまとめられる．

- すべての入れ子関数の定義を，変換後トップレベルへ移動させる．
- 入れ子関数がその入れ子関数を持つ関数（入れ子関数定義を持っていた関数）の局所変数へとアクセスできるようにするため，C のスタックとは別のスタック（**明示的スタック**）をメモリ上に用意する．入れ子関数が呼び出された時点で C のスタックの内容（局所変数の値など）は明示的スタックへと移され，変換後入れ子関数にはその入れ子関数を持つ関数の明示的スタック上のフレームへのポインタを渡す．
- 変換後入れ子関数の実行終了後，明示的スタックに移していた内容（入れ子関数呼び出し中に入れ子関数を呼び出した場合は差分のみ）をすべて C のスタックへと積み直す．

図 3 のプログラムを実装モデルに従い変換した場合を例に，従来の実装モデルについて説明する．このプログラムでは，関数 `foo` の中で定義されている入れ子関数 `g1` が，`foo` から呼び出された関数 `h` の中で間接的に呼び出されている．**図 4** に，C のスタックと明示的スタックの状態の

```
int h(int i, int (*g)(int)){
  return g(i);
}

int foo(int a){
  int x = 0;
  int g1(int b){
    x++;
    return a+b;
  }
  return h(10, g1)+x;
}

int main(){
  return foo(1);
}
```

図 3　入れ子関数を持つ C プログラムの例
Fig. 3　An example of a C program with a nested function.

遷移の一部を示す．変換されたプログラムは，入れ子関数 `g1` の間接的な呼び出しが起きると次の処理を行う．

(1) 入れ子関数 `g1` の引数を明示的スタックへプッシュする．また，「`g1` 本体へのポインタとその外側の関数の明示的スタック上のフレームへのポインタ」のペアを指すポインタをプッシュする．（図 4 中の “push arguments and a pointer”）
(2) 実行中の（変換後）関数 `h` や `foo` からリターンしながら，すべての局所変数および引数を明示的スタックへ保存する．（同 “move variables in C stack to explicit stack”）
(3) 変換後 `main` 関数まで局所変数および引数の保存が終了したら，明示的スタックのトップにあるペアへのポインタを利用して，変換後入れ子関数 `g1` を call する．さらに，ペアの片方から外側の関数のフレームへのポインタを取得し，明示的スタックのトップから引数を取得する．その後，`g1` の処理を開始する．（同 “call g1”）
(4) `g1` の処理が終了後，返り値を明示的スタックへプッシュし，return する．（同 “returning from g1”）
(5) 変換後関数 `foo`，`h` を順に呼び出しながら，明示的スタックにある局所変数や引数の値を，C のスタックへと移す．（同 “move variables in explicit stack to C stack”）

3.　変換ベース実装の実装モデル

本章では，標準的な C 言語へと翻訳する変換に基づく L-closure の実装において，生成・維持コストを抑えつつ，呼び出しコストもできるだけ抑える実装を提案する．

2.4 節で述べた従来の実装モデルは，入れ子関数の呼び出しがあるたびに C のスタックの内容はすべて明示的スタックへと移され，入れ子関数の実行終了時に明示的スタックに移されていた内容はすべて C のスタックへと移されていた．これにより，何回も入れ子関数の呼び出しが起きるプログラムでは，スタック間の内容の移動が多くなり大きな

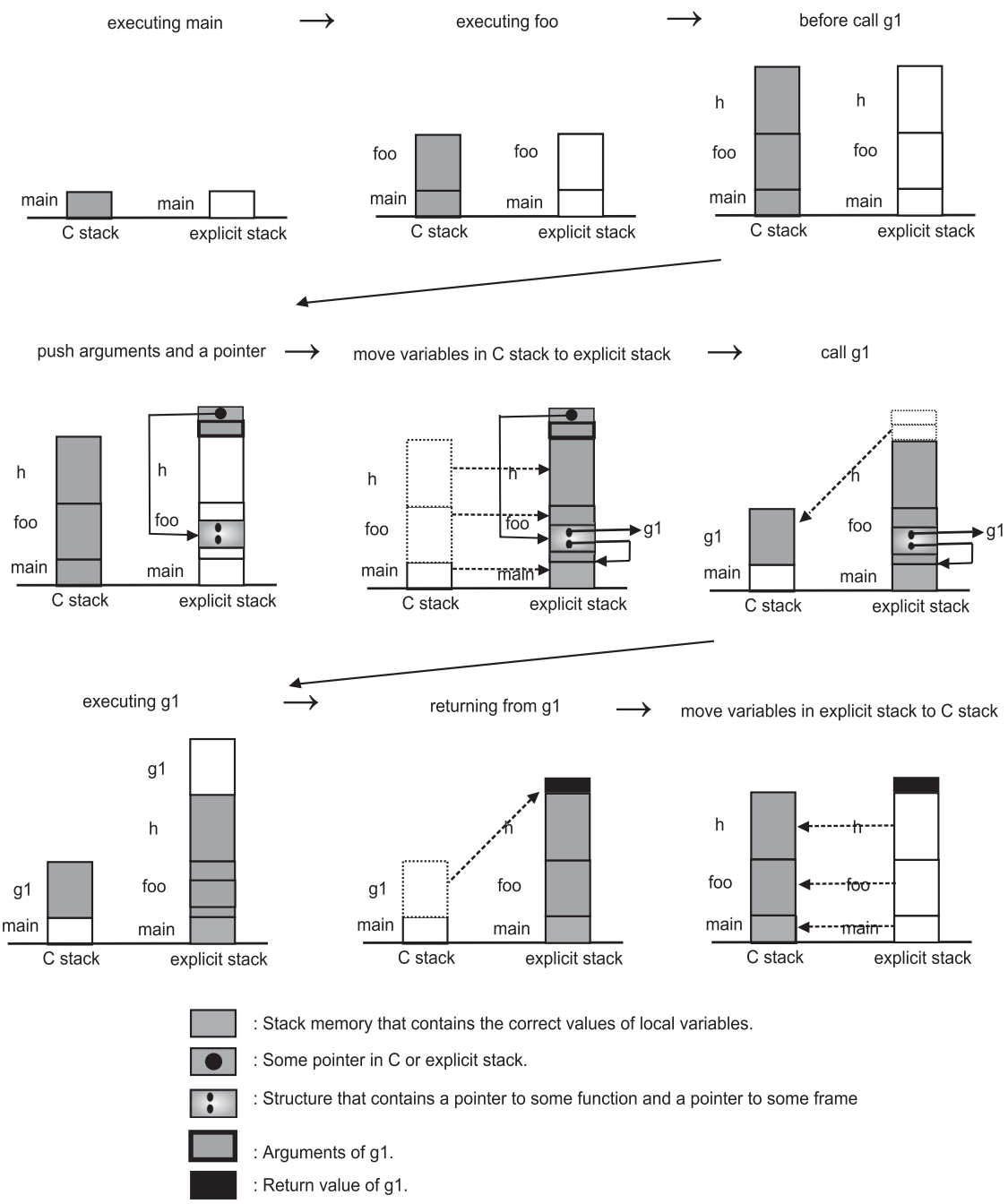

図 4　従来の変換ベースの L-closure の実装の詳細

Fig. 4　Details of old transformation-based implementation of L-closures.

オーバヘッドとなってしまう．

本研究ではこの点に注目し改善を行った．入れ子関数の実行終了時に明示的スタックに移していた内容をすべて C のスタックへと移すのではなく，入れ子関数を呼び出した関数のフレームのみを C のスタックへと移す．これにより，再度入れ子関数の呼び出しが起きたときのスタック間の内容の移動を減らし，呼び出しコストを抑える．

3.1　実装モデル概要

実装の概要のうち，2.4 節からの差分は以下の点である．

- 入れ子関数の実行終了時に，明示的スタックに移していた内容のすべてを C のスタックへと積み直すことなしに，入れ子関数呼び出しを行った関数のフレームのみを C のスタックへと戻し実行を再開する．

新しい実装モデルでは，（表 1 の用語で変換前のトップレベルの）関数についてそれぞれ対応する**引数結果補助関数**を用意する．変換後関数は，高速に call できるよう，その引数や結果の種類は変換前を反映して様々とした．このため，再開時に変換後関数を同じ方法で直接 call することはできない．代わりに，call の方法が 1 通りに正規化された引数結果補助関数を call し，引数結果補助関数が明示的スタック上の（様々な型や個数の）引数を渡しながら変換

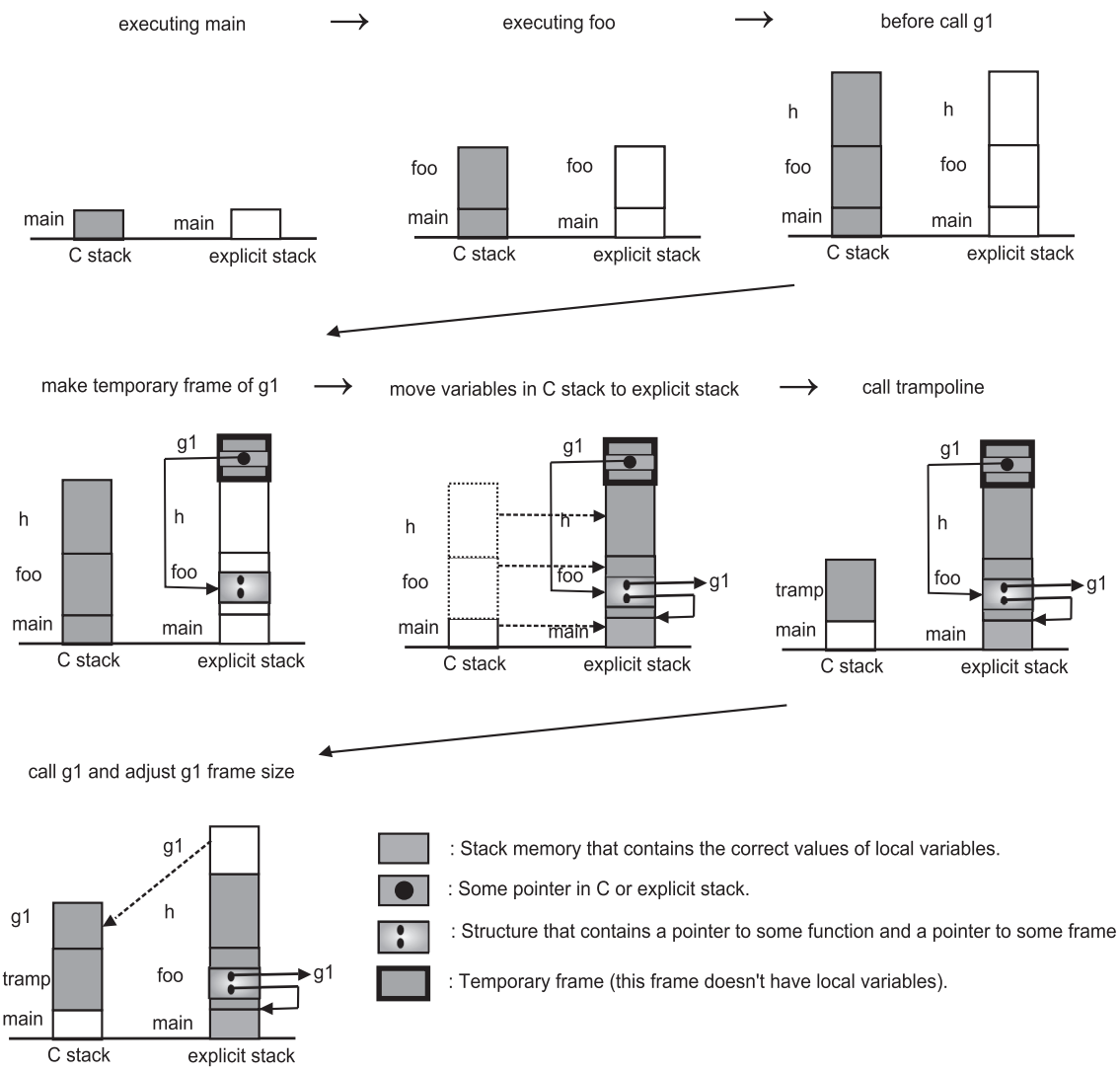

図 5　入れ子関数 g1 の呼び出し時のスタックの状態

Fig. 5　Details of stack condition when nested function g1 is called.

後関数を call し，その（様々な型の）結果を明示的スタックに保存することとした．また，引数結果補助関数を call するため，プログラム全体で 1 つ**共通トランポリン関数**を用意する．

2.4 節と同様に，図 3 のプログラムを実装モデルに従い変換した場合を例に，実装モデルを説明する．また，変換後プログラムの全体は，付録 A.1 に示す．

図 5 は，実行開始から入れ子関数 g1 の呼び出しが完了するまでの 2 つのスタックの状態の遷移を表している．また，**図 6** は，入れ子関数 g1 の実行が終了し，g1 から h へリターンし，その後 h から foo にリターンするときの 2 つのスタックの状態の遷移を表している．

まずは，入れ子関数 g1 の呼び出しが完了するまでの処理を説明する．入れ子関数 g1 の間接的な呼び出しが起きると，次の処理を行う．

(1) 明示的スタックに g1 の仮フレームを作り，仮フレームの中に「g1 本体へのポインタとその外側の関数の明示的スタック上のフレームへのポインタ」のペアを指すポインタを保存する．また，g1 の引数を保存する．（図 5 中の “make temporary frame of g1”）

(2) 実行中の変換後関数 h や foo から return しながら，すべての局所変数および引数を明示的スタックへ保存する．（同 “move variables in C stack to explicit stack”）また，図には示していないが，各関数の引数結果補助関数，および各関数の呼び出し先の関数のフレームへのポインタと，呼び出し元の関数のフレームへのポインタを保存する．

(3) 変換後 main 関数まで局所変数および引数の保存が終了したら，共通トランポリン関数を call する．（同 “call trampoline”）

(4) 共通トランポリン関数は，変換後入れ子関数 g1 の仮フレームからポインタペアを取り出し，g1 を call する．g1 は，ポインタペアから外側の関数のフレームへのポインタおよび引数を取得する．その後，仮フレー

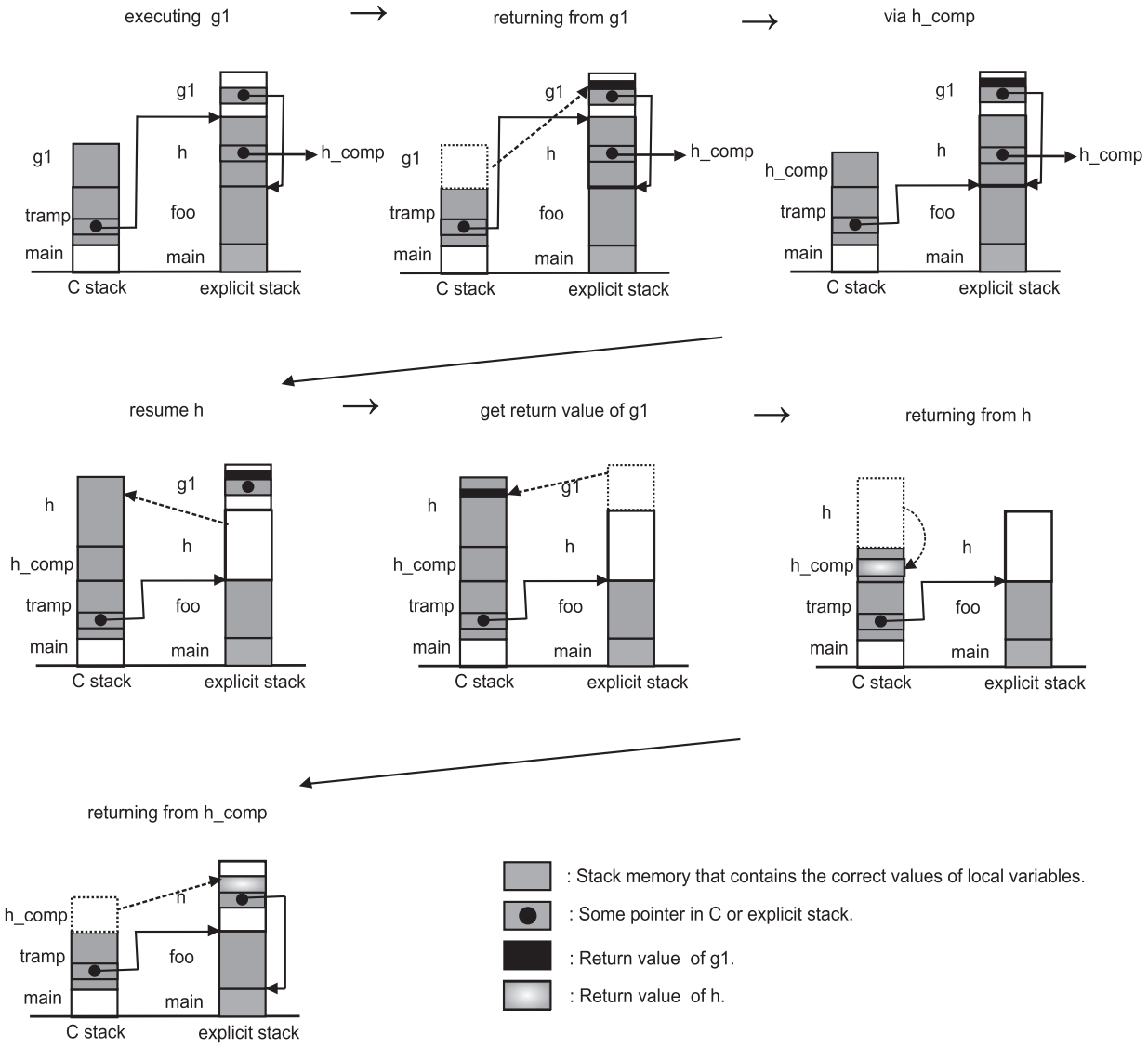

図 6　入れ子関数 g1 からリターンするときのスタックの状態
Fig. 6　Details of stack condition in returning from nested function g1.

ムのサイズを正式なフレームのサイズへと変更する．（同 “call g1 and adjust g1 frame size”）

従来の実装モデルと比較をしていくと，まず (1) では，従来は引数などを明示的スタックへプッシュしていたのに対し，本実装モデルでは仮フレームを作成し，そこに保存という操作を行っている．これは，(2) において，呼び出し元の関数のフレームへのポインタの保存を行うための領域など，引数以外の領域もあらかじめ確保しているためである．また，正式なフレームではなく，仮フレームというのは，正式なフレームであれば g1 の局所変数の領域も確保しておかなければならないが，この時点では呼び出す関数がどのような局所変数を持っているか分からないため，その領域は確保しないためである．(2) の処理は，従来の処理に加えて引数結果補助関数，および呼び出し元，呼び出し先のフレームへのポインタを保存している．これは，入れ子関数 g1 からリターンする際の後述する処理で使用する．(3) では変換後入れ子関数 g1 を call するために，共通トランポリン関数を call している．

次に，入れ子関数 g1 の実行が終了し，g1 から h へリターン，その後 h の実行が終了し，h から foo へリターンするときの処理を説明する．g1 の実行が終了すると，次の処理を行う．

(5) 変換後入れ子関数 g1 は返り値を明示的スタック上の g1 のフレームに保存し，return する．（図 6 中の “returning from g1”）
(6) 共通トランポリン関数は，この時点で g1 のフレームへのポインタを所持している．そのポインタをたどり，引数結果補助関数 h を call する．（同 “via h_comp”，h_comp は引数結果補助関数 h を指す）
(7) 引数結果補助関数 h は，変換後関数 h を call する．明示的スタックの h のフレームの内容，引数については引数結果補助関数 h が，局所変数については変換後関

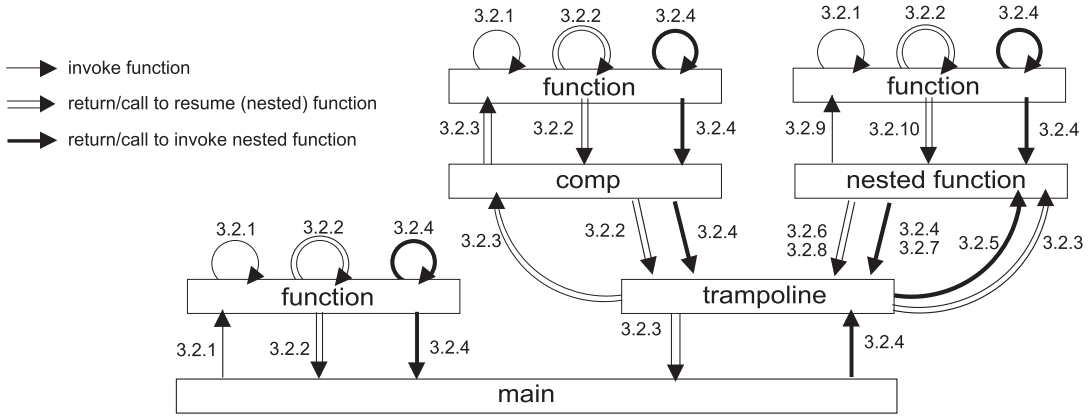

図 7　変換後プログラムの流れ
Fig. 7　Flow of translated program.

数 h が C のスタックへと移す．（同 “resume h”）
(8) 明示的スタック上の g1 のフレームに保存されている返り値を取得する．（同 “get return value of g1”）
(9) 変換後関数 h の実行が終了後，引数結果補助関数 h へ返り値を return する．（同 “returning from h”）
(10)引数結果補助関数 h は，受け取った返り値を明示的スタック上の h のフレームに保存し，return する．（同 “returning from h_comp”）
(11)以下，g1 を h に，h を foo に読み替えて，(6) へと戻る．

(6)，(7) のとおり，共通トランポリン関数は変換後関数（ここでは h）を直接 call する代わりに，対応する引数結果補助関数を call することで，h や foo といった関数の引数や結果（の個数や型）の違いを吸収している．ここで，引数結果補助関数自体は次節で述べるように，明示的スタック上のフレームへのポインタを引数とし，return 後の指示（「リターン指示」か「入れ子関数呼び出し指示」）を結果とする．

3.2　実装モデル詳細

本節では入れ子関数から入れ子関数を呼び出す場合や，入れ子関数呼び出しが一度終了した後，再度入れ子関数呼び出しが起きた場合などもカバーした，より詳細な実装モデルの説明を行う．以下，表 1 の用語を引き続き用いる．

図 7 に，変換後プログラムが取りうる流れを示す．図中の comp は引数結果補助関数である．各矢印が変換後関数の call や return を表しており，ラベルはその処理の変換の詳細が書かれている節を示している．

以下，処理ごとに場合分けを行い説明する．

3.2.1　関数 f から関数 g を呼び出すとき

変換後関数 f からは，明示的スタックのスタックポインタ（esp）と通常の引数を渡して変換後関数 g を call する．変換後関数 g 側では，esp の下位ビットを見ることで，開始か再開かを判定する．この場合は変換後関数からの call は開始と判定され，明示的スタックに管理と返り値と引数と局所変数に必要な領域を確保し，関数 g の処理を開始する．

3.2.2　関数 g から関数 f にリターンするとき

変換後関数 g では，明示的スタック上の g のフレームの，呼び出し先の関数のフレームを保存するためのポインタ（child frame フィールド）に 0 を保存する．

関数 g が開始時の場合，変換後関数 g から変換後関数 f に通常の return を行う．変換後関数 f では返り値が「**特別値**」かどうかを判定する．特別値の場合，g のフレームの child frame フィールドの値が 0 かを判定する．これは，たまたま「特別値」と同じ値をリターンした可能性があるため，本当に特別値であるかを判断するためである．「本当の特別値」は後述の入れ子関数の呼び出しの際に用いられる．ここではたまたま「特別値」と同じ値をリターンしたと判定され，関数 f は関数 g の呼び出し後の処理を再開する．変換後関数 g の返り値が特別値でない場合，関数 f は単に関数 g の呼び出し後の処理を再開する．

関数 g が再開時の場合，変換後関数 g は引数結果補助関数 g（変換前の g という名前をここでも用いる）から call されており，引数結果補助関数 g に通常の return を行う．引数結果補助関数 g では，返り値が「特別値」かどうかを判定する．特別値の場合，同様に，g のフレームの child frame フィールドの値が 0 かを判定する．ここではたまたま「特別値」と同じ値をリターンしたと判定され，返り値を明示的スタックの g のフレームに保存し，共通トランポリン関数に「**リターン指示**（を示す値）」を return する．以下，3.2.3 項の処理を行う．

3.2.3　共通トランポリン関数が「リターン指示（を示す値）」を受け取ったとき

共通トランポリン関数が「リターン」に対応する場合について，関数へのリターンだけではなく，入れ子関数へのリターンの場合も合わせて述べる．

共通トランポリン関数が，「リターン指示（を示す値）」を受け取るのは，引数結果補助関数，または変換後入れ子関

数からの戻り値としてである．引数結果補助関数の場合，対応する関数を g，変換後入れ子関数の場合，対応する入れ子関数を g とする．

共通トランポリン関数では，返り値が「リターン指示」であると判定後，g のフレームから呼び出し元の関数のフレームを保存するポインタ（parent frame フィールド）を取り出す．ここでは，g の呼び出し元を f とする．

f が main 関数，すなわち parent frame フィールドの値が 0 であった場合，共通トランポリン関数から変換後 main 関数へ return する．変換後 main 関数では，main のフレームから引数，局所変数，一時変数の回復，g のフレームから返り値の取り出しを行う．そして，main の処理を再開する．

f が main 関数以外の関数であった場合，f のフレームから引数結果補助関数 f を取り出し，f のフレームへのポインタを引数として，引数結果補助関数 f を call する．引数結果補助関数 f では，f のフレームから引数を取り出して，変換後関数 f を call する．このとき，esp となる引数には，f のフレームのアドレスにビットを立てたものとする．変換後関数 f では，esp の下位ビットを見ることで，開始か再開かを判定する．引数結果補助関数からの call は，すべて再開である．再開と判定されたあと，esp のビットクリア，esp の値の修正による明示的スタック上の f のフレームの再認識，f のフレームから再開位置番号の取り出し，再開位置へジャンプを行う．また，f のフレームから局所変数，一時変数の回復，g のフレームから返り値の取り出しを行う．そして，関数 f の処理を再開する．

f が入れ子関数であった場合，f のフレームから変換後入れ子関数 f を取り出し，call する．変換後入れ子関数 f では，再開位置番号で，開始か再開かを判定する．ここでは，再開と判定され，再開位置番号に対応する再開位置へジャンプする．また，f のフレームから引数，局所変数，一時変数の回復，g のフレームから返り値の取り出しを行う．そして，入れ子関数 f の処理を再開する．

3.2.4　関数 g から入れ子関数 h を呼び出すとき

変換後関数 g では，明示的スタックに h の仮フレームを確保し，引数を仮フレームに保存する．h の関数ポインタとなる「(変換後入れ子関数 h と静的リンクという）ペアのアドレス」を保存する．ただし，この時点ではペアが未初期化である可能性がある．h の仮フレームの child frame フィールドに「特別値としての h の仮フレーム自身」を保存する．

(1) g が main の場合，

main の引数や局所変数，変換時に生成した一時変数などを，明示的スタック上の main のフレームに保存する．main が入れ子関数を持っていた場合，変換後入れ子関数の本体関数ポインタと，静的リンクのペアを保存（初期化）する．main のフレームの child frame フィールドに h の仮フレームのアドレスを，h の仮フレームの parent frame フィールドに 0 を保存し，共通トランポリン関数の「入れ子関数呼び出し指示」call を行う．続きの処理は，3.2.5 項に示すとおりとなる．

(2) g が main 以外の場合，

g の引数や局所変数，変換時に生成した一時変数などを，明示的スタック上の g のフレームに保存する．また，再開時に現在の実行ポイントへとジャンプできるよう，再開位置番号を保存する．さらに，引数結果補助関数 g のアドレスを保存する．関数 g が入れ子関数を持っていた場合，変換後入れ子関数の本体関数ポインタと，静的リンクのペアを保存（初期化）する．g のフレームの child frame フィールドに h の仮フレームのアドレスを，h のフレームの parent frame フィールドに g のフレームのアドレスを保存し，「特別値」を return する．g からの return 後の処理には call 元に応じて以下の場合がある．

・変換後関数 g が，変換後 main 関数から call されていた場合，変換後 main 関数では，返り値が「特別値」であるかどうかを判定する．特別値と判定後，g のフレームの child frame フィールドの値が 0 であるか判定する．ここでは 0 ではないので，入れ子関数呼び出しであると判定する．上記の (1) 以降の処理を，g を main に，h を g に，仮フレームをフレームに読み替えて行う．

・変換後関数 g が，変換後関数 f（main 以外）から call されていた場合，変換後関数 f では，返り値が「特別値」であるかどうかを判定する．特別値と判定後，g のフレームの child frame フィールドの値が 0 であるか判定する．ここでは 0 ではないので，入れ子関数呼び出しであると判定する．上記の (2) 以降の処理を，g を f に，h を g に，仮フレームをフレームに読み替えて行う．

・変換後関数 g が，引数結果補助関数 g から call されていた場合，引数結果補助関数 g では，返り値が「特別値」であるか判定する．特別値と判定された後，さらに g のフレームの child frame フィールドの値が 0 であるかを判定する．ここでは，0 ではないので「入れ子関数呼び出し」と判定する．共通トランポリン関数に，「入れ子関数呼び出し指示（を示す値)」を return する．続きの処理は 3.2.5 項に示すとおりとなる．

・変換後関数 g が変換後入れ子関数 i から call されていた場合．変換後入れ子関数 i では，返り値が「特別値」であるかどうかを判定する．特別値と判定後，g のフレームの child frame フィールドの値が 0 であるか判定する．i の引数や局所変数，変換時に生成した一時変数などを，明示的スタック上の i のフレームに保存する．また，再開時に現在の実行ポイントへとジャ

ンプできるよう，再開位置番号を保存する．関数 i が入れ子関数を持っていた場合，変換後入れ子関数の本体関数ポインタと，静的リンクのペアを保存（初期化）する．i のフレームの child frame フィールドに g の仮フレームのアドレスを，g のフレームの parent frame フィールドに i のフレームのアドレスを保存し，共通トランポリン関数に「入れ子関数呼び出し指示（を示す値）」を return する．続きの処理は，3.2.5 項に示すとおりとなる．

3.2.5　共通トランポリン関数が「入れ子関数呼び出し指示（を示す値）」を受けとったとき

共通トランポリン関数が，「入れ子関数呼び出し指示（を示す値）」を受け取るのは，変換後 `main` 関数からの call として，もしくは引数結果補助関数，変換後入れ子関数のいずれかからの return としてである．変換後 `main` 関数からの call であった場合，（変換前） `main` 関数が呼び出し中の（あるいは呼び出そうとする）関数（あるいは入れ子関数）を g，引数結果補助関数からの return であった場合，対応する関数を g，変換後入れ子関数からの return であった場合，対応する入れ子関数を g とする．共通トランポリン関数では，この「入れ子関数呼び出し指示」を受け取ると，g のフレーム（あるいは仮フレーム）の child frame フィールドを，「特別値としての仮フレームへのポインタ自身」に行き着くまでたどる．「特別値としての仮フレームへのポインタ自身」を child frame フィールドに持つフレームが，呼び出すべき入れ子関数の仮フレームであるから，仮フレームを取り出すことができる．この入れ子関数を h とすると，h の仮フレームからペアのアドレスを取り出し，変換後入れ子関数 h の本体関数ポインタと静的リンクを得る．静的リンクを h の仮フレームに保存し，再開位置番号を開始を表す番号として，変換後入れ子関数 h を call する．変換後入れ子関数 h では，仮フレームを伸ばして正式なフレームにする．そして，入れ子関数 h の処理を開始する．

3.2.6　入れ子関数 h から関数 g にリターンするとき

変換後入れ子関数 h では，返り値を明示的スタックの h のフレームに保存する．入れ子関数は共通トランポリン関数から call されているので，共通トランポリン関数に「リターン指示」を return．続きの処理は，3.2.3 項に示すとおりとなる．

3.2.7　入れ子関数 h から入れ子関数 i を呼び出すとき

変換後入れ子関数 h では，明示的スタックに i の仮フレームを確保し，引数を仮フレームに保存する．i の関数ポインタとなる「ペアのアドレス」を保存する．ただし，この時点ではペアが未初期化である可能性がある．i の仮フレームの child frame フィールドを「特別値としての i の仮フレーム自身」とする．h の引数や局所変数，変換時に生成した一時変数などを，明示的スタック上の h のフレームに保存する．また，再開時に現在の実行ポイントへとジャンプできるよう，再開位置番号を保存する．さらに，変換後入れ子関数 h のアドレスを保存する．入れ子関数 h が入れ子関数を持っていた場合，変換後入れ子関数の本体関数ポインタと，静的リンクのペアを保存（初期化）する．h のフレームの child frame フィールドに i の仮フレームのアドレスを，i のフレームの parent frame フィールドに h のフレームのアドレスを保存し，「入れ子関数呼び出し指示（を示す値）」を return する．続きの処理は，3.2.5 項に示すとおりとなる．

3.2.8　入れ子関数 i から入れ子関数 h にリターンするとき

変換後入れ子関数 i では，返り値を明示的スタックの i のフレームに保存する．入れ子関数は共通トランポリン関数から call されているので，共通トランポリン関数に「リターン指示」を return．続きの処理は，3.2.3 項に示すとおりとなる．

3.2.9　入れ子関数 i から関数 g を呼び出すとき

3.2.1 項の，関数 f を入れ子関数 i に置き換えて，同様の処理を行う．

3.2.10　関数 g から入れ子関数 i にリターンするとき

3.2.2 項の，関数 f を入れ子関数 i に置き換えて，同様の処理を行う．

4.　SC 言語処理系に基づく実装

本研究では，提案する実装モデルを SC 言語処理系 [10], [12], [35], [36] を用いて実装した．SC 言語処理系は，変換ベースによる言語拡張を支援するシステムとして研究・開発された．SC 言語は S 式ベースの構文を持つ 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 言語処理系では，SC-0 言語から C 言語への変換器を提供しているため，利用者は拡張 SC 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．今回の場合，「L-closure を持つ拡張 SC 言語 LW-SC」の変換ベース実装のために，SC-0 言語に入れ子関数の機能を追加した言語を，本研究で提案する実装モデルに従い SC-0 言語へと変換する変換器を作成する．

新しい変換器は，変形規則を書くことで実装できる．本研究で提案する実装モデルを実現するための変形規則は，従来の実装から多く流用することができた．具体的には，従来の実装の変形規則は 4 つのフェーズに分かれており，実装モデルを実現するためのプログラムの変換は，そのうちの 1 つである `nestfunc` 規則セットで行われている．本研究では `nestfunc` 規則セットの変更のみで，提案する実装モデルを実装できた．従来の実装の `nestfunc` 規則セットからの追加・変更は 220 行程度であり，`nestfunc` 規則セットの総行数は 830 行程度となった．

5. GCC に基づく closure の実装

本章では，L-closure の研究の一環として提案している “closure” の，GNU C コンパイラ（GCC）4.6.3 に基づくコンパイラベースの実装を行ったので報告する．closure の概要は，2.2 節で示したとおりである．我々はすでに GCC-3.4.6 への clsoure の実装を発表 [33] しており，GCC 3 と本研究で使用した GCC 4 の内部構造の違いとその影響について説明する．

5.1 実装方針

GCC では，C への拡張として独自の入れ子関数が使える．GCC の入れ子関数は通常のトップレベルの関数との相互運用性を確保するために，「トランポリン」という技法を用いている [3]．3 章で提案した実装モデルで使われている共通トランポリン関数とは，言葉は似ているが別のものである．トランポリンとは静的リンクとして必要な環境をセットしてから入れ子関数本体のコードへジャンプする数命令の短いコードのことであり，スタック上に動的に生成される．トランポリンのコードアドレスを関数ポインタとすることによって通常のトップレベルの関数との相互運用性が確保される．ただし，(1) スタック上にコードを動的に生成することや，(2) アーキテクチャによってはプロセッサの持つ命令キャッシュを明示的にフラッシュする必要があること，(3) OS がスタックに実行可能なコードを置くことを制限している場合は，その制限の解除[*2]という高い生成コストが発生する．

GCC への closure の実装は，GCC 3 系列，4 系列ともにトランポリンにならう形で比較的容易に実装できる．つまり，GCC が入れ子関数を実現するのに用いるトランポリンの代わりに，静的リンクとコードアドレスというポインタペアを用いる．ただし，GCC 3 から GCC 4 へのバージョンアップにより GCC の内部構造が刷新されており，トランポリンの実装自体が変わっているため，closure の実装方針は変わらないが，実装をそのまま移植できるというものではなかった．

5.2 内部構造の変化と実装への影響

フロントエンドにおいて，GCC 3 系列では構文解析器が Bison で自動生成されていたのに対し，GCC 4 系列では C 言語で直接書かれており，closure の実装においてもそれに対応した変更が必要であった．

また，中間表現にも変更があり，GCC 3 系列では生成しようとしているコードの中間表現として，RTL（Register Transfer Language）と呼ばれる中間言語のみが採用されていた．RTL は C よりはアセンブリ言語に近い中間言語である．RTL は，様々な最適化などにより変形されたあと，アセンブリコードへと変換される．GCC 4 系列では，RTL に加えて新たに GENERIC および GIMPLE と呼ばれる 2 つの中間表現が追加された．GCC 4 系列では，抽象構文木はまず言語に非依存な木構造を持つ表現である GENERIC へと変換される．次に GENERIC は GIMPLE へと変換される．GIMPLE は，GENERIC と同じく木構造を持つ表現であるが，式のオペランドは 3 つまでに制限されるなど，GENERIC により制限を加えた表現となっている．そして，GIMPLE で最適化がかけられたあと，RTL へと変換される．

5.1 節で述べたとおり，closure の実装はトランポリンにならう形で実装できる．GCC 3 系列では RTL の段階でトランポリンの初期化を行うコードを生成するのに対し，GCC 4 系列では GIMPLE の段階で，トランポリンの初期化用の組み込み関数を呼び出すコードを生成し，RTL の段階で展開する．本研究の closure の実装についても，これにならう形で実装を行った．

また，L-closure の実装に関して 2.3 節で述べたとおり，L-closure の実装では実装手法として変数の場所を 2 つ設けるという方法を取っている．GCC 3 系列では RTL レベルでこの変換を実装していたが，GCC 4 系列では GIMPLE の段階で変数から構造体のフィールドに変換するようになり，GCC 3 系列での L-closure の実装手法はそのまま適用できなくなった．

6. 性能測定

入れ子関数を使わない通常の C プログラムに対しては，5 章で拡張したコンパイラも拡張前と同じコードを生成する．よって，lexical closure の生成・維持コストを測定するために，まずは高水準サービスのための入れ子関数をともなうプログラムと，それを省いた標準の C プログラムとの比較を行った．また，L-closure の複数の実装について性能評価を行うため，2.3 節で述べた GCC-3.4.6 による L-closure の実装についても評価した．

fib(37) (check-pointing)　チェックポインティングのためのスタック状態キャプチャ機能つきで，37 番目の Fibonacci 数を再帰的に求める．

また，分散環境にも対応した遅延分割型負荷分散（load balancing）フレームワーク Tascell [11], [14], [16], [27], [30] を利用して，次のプログラムの性能測定を行った．Pentomino については，タスク分割の際のその場更新の部分解に対するバックトラック時の更新一時的取り消しにも，入れ子関数を用いる．

fib(37) (load-balancing)　再帰による Fibonacci 数の計算であり，fib (3) すらタスク分割の候補とする細粒度計算を行う．

Comp(20000)　単純にサイズ N の配列の 2 要素間の比

[*2] 制限が解除できない場合も考えられる．

較を行う（$N = 20000$）．データサイズ $O(N)$，計算量 $O(N^2)$ である．

Pentomino　Pentomino パズルのすべての解をバックトラック探索で求める．

LU(1000)　再帰を用いた N 次正方行列の LU 分解（$N = 1000$）．データサイズ $O(N^2)$，計算量 $O(N^3)$．

図 1 に示したように，これらの高水準サービスのサポートのために，各関数は入れ子関数定義を所有する．つまり，呼び出しごとに（すくなくとも論理的には）lexical closure が生成される．ここで，この測定では入れ子関数は呼び出されないようにした．つまり，チェックポインティング，ワークスティールは起動されない．

さらに，本研究の目的である，L-closure の呼び出しコストの削減がどの程度達成できたかを測定するため，次のプログラムの性能測定を行った．このプログラムは，故意に入れ子関数の呼び出しが頻繁に起こるように設計してある．

Quick sort　クイックソートの比較器として，通常の関数ポインタまたは closure ポインタを渡してクイックソート本体から間接的に呼び出す．

また，実際に高水準サービスが利用された場合に，本研究での実装がどの程度の性能改善が見込めるかを測定するため Tascell を利用し，次のプログラムの性能測定を行った．このプログラムは Quick sort とは違い，故意に頻繁に入れ子関数の呼び出しを起こすのではなく，多数のワーカによる負荷分散を実現するときの，ワークスティール時の入れ子関数呼び出しがそれなりに多く含まれるプログラムとなっている．

spanning tree　与えられたハイパーキューブの全域木を求める．頂点数を N，枝数を M とすると，グラフサイズは 2^N（$N = 20$），計算量は $O(N + M)$．

測定環境・測定条件は以下のとおりである．

- CPU（使用したモード）
  - UltraSPARC T2 Plus 1.4 GHz (SPARC v9，32 bit)
  - AMD Opteron 6238 2.6 GHz (x86-64)
  - Intel Xeon E5-2670 2.6 GHz (x86-64)
- コンパイラ
  - closure を組み込んだ GCC-4.6.3，ただし L-closure，closure に関してのみ GCC-3.4.6 についても測定，`-O2 -fno-optimize-sibling-calls`．x86-64 で，GCC-3.4.6 の L-closure を使用する場合 `-fno-omit-frame-pointer`．
  - Intel C++ Compiler Version 12.1.3（ICC）（入れ子関数を持たないもののみを測定）．

表 2，**表 3**，**表 4** に性能測定の結果をまとめる．ここで，“no closures” は高水準サービスをともなわない単なる C プログラムである．つまり，関数呼び出しごとに入れ子関数定義をともなったり，入れ子関数のポインタを追加の引数として渡すことはしていない．“trampolines” は GCC の従来の入れ子関数を使用した場合である．“new LW-SC” は本研究で実装した変換に基づく L-closure の実装であり，“old LW-SC” は従来の実装である．“L-closures” は GCC 3 に実装されたコンパイラベースの L-closure である．trampoline では，関数呼び出し頻度が高いいくつかの関数では，実行時間が倍以上になっているものも見られる．一方 L-closure や LW-SC については，“trampolines” に比べて，“no closures” により近いものが多く見られる．L-closure を備える “L-closures”，“old LW-SC”，“new LW-SC” の 3 つに注目すると，生成・維持コストについてはどれもほぼ同じ値を示している一方，呼び出しコストが大きくかかる表 3 のプログラムにおいては，本研究で実装した “new LW-SC” が大幅に良い結果となっている．つまり，提案手法が目的とする，生成・維持コストをそのままに，呼び出しコストを小さくすることが達成できた．また，表 4 のプログラムにおいても，複数ワーカを使った負荷分散を行った場合，old-LWSC より new-LWSC の方が速いという結果となっており，たとえば UltraSPARC に注目すると，old-LWSC は 1 ワーカを用いたときと比較して，48 ワーカでは 26.2 倍の速度向上であるのに対し，new-LWSC では 27.4 倍の速度向上であった．Intel Xeron では，1 ワーカと 16 ワーカを比べたときの速度向上では下回っているものの，両ワーカともに絶対的な実行時間の短さで上回っているという結果になった．さらに，表には載せていないが，Tascell 版 fib(37) を UltraSPARC 上で 64 ワーカで並列実行したところ，new LW-SC で 1 ワーカを用いたときと比較して，old LW-SC は 9.41 倍の速度向上であったのに対し，呼び出しコストの低い new LW-SC では 10.6 倍の速度向上があり，かつ実行時間も短いという結果も得られた．

また，今回実装した closure (gcc4) も，予想どおり，表 3 のプログラムにおいてきわめて低い呼び出しコストを示した．

表 2 で生成・維持コストに関して “L-closures” と “closure (gcc4)” を見ると，SPARC においては多くのプログラムで “L-closures” の方が性能が良いが，x86-64 では性能が悪かった．これは，L-closures はレジスタの利用を促進しており，callee-save レジスタを利用するようなレジスタ割当てを促進するが，callee-save レジスタの保存・回復には 1 回とはいえ，コストが必要である．x86-64 では，フレームポインタの利用のための保存・回復が必要なときのみとなったが，L-closure ではフレームポインタの利用が必須である．このようなことから，特に fib(37) や，Comp(20000) のような関数本体での変数アクセスが極端に少ないプログラムでは，callee-save レジスタの利用コストが不利になる．Pentomino のような関数本体での変数アクセスが極端に少なくないプログラムでは，L-closures と closure (gcc4) はほぼ互角であった．また，LW-SC も同様の理由で closure (gcc4) より性能が悪くなっている．LW-SC は，L-closures

表 2　性能測定（生成・維持コスト）

Table 2　Performance measurements (for creation and maintenance cost).

S: UltraSPARC
O: AMD Opteron
X: Intel Xeron

Elapsed time in seconds
(relative time to plain C)

		no closures	trampolines	closures (gcc3)	L-closures	old LW-SC	new LW-SC	closure (gcc4)
fib(37) checkpointing	S (GCC)	1.40 (1.00)	6.54 (4.67)	4.05 (2.89)	1.92 (1.37)	2.08 (1.49)	1.96 (1.40)	2.23 (1.59)
	O (GCC)	0.152 (1.00)	0.298 (1.96)	0.214 (1.41)	0.273 (1.8)	0.312 (2.05)	0.281 (1.85)	0.214 (1.41)
	X (GCC)	0.130 (1.00)	0.226 (1.74)	0.180 (1.38)	0.183 (1.41)	0.222 (1.71)	0.247 (1.9)	0.166 (1.28)
	X (ICC)	0.145 (1.00)	–	–	–	0.253 (1.74)	0.222 (1.53)	–
fib(37) load balancing	S (GCC)	1.38 (1.00)	6.39 (4.63)	3.13 (2.27)	2.17 (1.57)	2.10 (1.52)	2.21 (1.60)	2.44 (1.77)
	O (GCC)	0.148 (1.00)	0.306 (2.07)	0.229 (1.55)	0.282 (1.91)	0.324 (2.19)	0.322 (2.18)	0.261 (1.76)
	X (GCC)	0.103 (1.00)	0.255 (2.48)	0.192 (1.86)	0.226 (2.19)	0.248 (2.41)	0.234 (2.27)	0.217 (2.11)
	X (ICC)	0.127 (1.00)	–	–	–	0.285 (2.24)	0.283 (2.23)	–
Comp(20000) load balancing	S (GCC)	9.15 (1.00)	33.7 (3.68)	20.1 (2.20)	12.2 (1.33)	11.1 (1.21)	11.0 (1.20)	15.0 (1.64)
	O (GCC)	1.45 (1.00)	1.90 (1.31)	1.83 (1.26)	1.94 (1.34)	2.01 (1.39)	1.97 (1.36)	1.64 (1.13)
	X (GCC)	0.938 (1.00)	1.55 (1.65)	1.35 (1.44)	1.45 (1.55)	1.33 (1.42)	1.31 (1.40)	1.29 (1.38)
	X (ICC)	1.39 (1.00)	–	–	–	2.09 (1.50)	2.10 (1.51)	–
Pentomino load balancing	S (GCC)	3.78 (1.00)	6.42 (1.70)	7.7 (2.04)	3.97 (1.05)	6.17 (1.63)	5.23 (1.38)	5.70 (1.51)
	O (GCC)	1.09 (1.00)	1.23 (1.13)	1.51 (1.39)	1.21 (1.11)	1.40 (1.28)	1.50 (1.38)	1.20 (1.10)
	X (GCC)	0.895 (1.00)	1.02 (1.14)	1.22 (1.36)	1.02 (1.14)	1.12 (1.25)	1.14 (1.27)	1.01 (1.13)
	X (ICC)	0.91 (1.00)	–	–	–	1.19 (1.31)	1.10 (1.21)	–
LU(1000) load balancing	S (GCC)	10.0 (1.00)	9.92 (0.992)	10.5 (1.05)	10.3 (1.03)	10.1 (1.01)	10.1 (1.01)	10.0 (1.00)
	O (GCC)	0.466 (1.00)	0.456 (0.979)	0.528 (1.13)	0.643 (1.38)	0.466 (1.00)	0.466 (1.00)	0.457 (0.981)
	X (GCC)	0.288 (1.00)	0.388 (1.35)	0.778 (2.70)	0.778 (2.70)	0.401 (1.39)	0.289 (1.00)	0.289 (1.00)
	X (ICC)	0.163 (1.00)	–	–	–	0.166 (1.02)	0.161 (0.988)	–

表 3　性能測定（呼び出しコスト）(1/2)

Table 3　Performance measurements (for invocation costs) (1/2).

S: UltraSPARC
O: AMD Opteron
X: Intel Xeron

Elapsed time in seconds
(relative time to plain C)

		no closures	trampolines	closures (gcc3)	L-closures	old LW-SC	new LW-SC	closures (gcc4)
Quick sort	S (GCC)	0.431 (1.00)	0.427 (0.991)	0.595 (1.38)	24.9 (57.8)	8.64 (20.0)	2.27 (5.27)	0.413 (0.958)
	O (GCC)	0.0988 (1.00)	0.102 (1.03)	0.101 (1.02)	1.57 (15.9)	1.85 (18.7)	0.316 (3.20)	0.101 (1.02)
	X (GCC)	0.0455 (1.00)	0.0553 (1.22)	0.0461 (1.01)	1.25 (27.5)	1.46 (32.1)	0.254 (5.58)	0.0507 (1.11)
	X (ICC)	0.0445 (1.00)	–	–	–	1.75 (39.3)	0.285 (6.40)	–

表 4　性能測定（呼び出しコスト）(2/2)

Table 4　Performance measurements (for invocation costs) (2/2).

S: UltraSPARC
O: AMD Opteron
X: Intel Xeron

Elapsed time in seconds

		ワーカ数	old LW-SC	new LW-SC
spanning tree	S (GCC)	1	2.28	2.33
		48	0.0870	0.0849
	O (GCC)	1	0.460	0.438
		48	0.0706	0.0706
	X (GCC)	1	0.288	0.273
		16	0.0324	0.0321

よりさらに性能が悪くなっているものも見られるが，これは，たとえば明示的スタックの管理コストであったり，変換後の return において，入れ子関数呼び出しか区別するコストが加わっているためである．一方，SPARC において L-closures の方が性能が良い理由は，SPARC アーキテクチャではレジスタウィンドウが利用でき，callee-save レジスタの保存・回復のコストが必要となるかは遅延され，平均すると少なくなるからである．SPARC アーキテクチャは，スパコンの「京」で用いられており，そのような環境で L-closure や LW-SC を使えば，Tascell のような並列言語処理系の高性能実装が可能となると期待できる．

ICC に注目すると，“L-closures” を発表した段階では，LW-SC により生成された C プログラムを GCC 3 系列でコンパイルするより，ICC でコンパイルする方が良い結果が出るものが多かった．しかし今回は，一概に ICC の方が良いという結果ではなかった．これは，GCC 4 系列へのバージョンアップにともない内部構造が刷新され，より多くの最適化が行われていることが関係していると考えられる．しかし，ICC は LU などのプログラムでは依然 GCC より良い結果となった．

7. 関連研究

7.1 高水準サービスと実装手法

L-closure [13], [29], [31], [33] を用いることで, 1 章, 2.1 節で注目したような，ごみ集めのような高水準サービス以外にも，様々な高水準サービスを高い移植性と再利用性を持つ形で実装できる．実際に，L-closure を用いて，

(1) ごみ集め/一級継続/真の末尾再帰に関する実装手法 [32]
(2) マルチスレッドに関する実装手法 [10], [24]
(3) 動的負荷分散に関する実装手法 [11], [14], [16], [27], [30]

が提案されてきた．

L-closure を用いない場合には，通常，それぞれの高水準サービスごとに実装手法を開発する必要があり，

(1) ごみ集めに特化した実装手法 [2], [9]
(2) デバッガに特化した実装手法 [8]
(3) マイグレーション/一級継続/チェックポインティングに特化した実装手法 [4], [17], [21], [22]
(4) 真の末尾再帰に特化した実装手法 [1]
(5) マルチスレッドに特化した実装手法 [19], [26], [28]
(6) 動的負荷分散に特化した実装手法 [5], [6], [7], [15], [23], [25], [34]

などが提案されている．これらの実装手法は多くの共通点（手の込んだ翻訳手法やコンパイラ拡張など）を持つが，L-closure のような形では実装基盤の再利用はなされていなかった．なお，C−− [18], [20] は例外処理，ルートスキャンなど，複数の特定のサービスに対するインタフェースは提供するが，L-closure のレベルの汎用性はない．

7.2 インクリメンタルなフレーム管理

本研究で提案した新しい LW-SC 言語の変換手法（L-closure の翻訳に基づく実装手法）は，以前の方式 [13] と比較して，インクリメンタルなフレーム管理を特徴とする．

インクリメンタルなフレーム管理というアイディアは，前節で述べたような各種高水準サービスの実装の中でも用いられてきた．特に，マイグレーション/一級継続/チェックポインティングにおいては，スタック全体を一括で扱う手法 [17], [21], [22] とは異なり，フレーム単位でインクリメンタルに扱う，インクリメンタルスタック/ヒープ法 [4] が，一級継続の効率の良い手法として知られている．この手法では，ヒープへ退避した継続を呼び出す時点では，フレームを 1 つだけコピーしてスタックへ戻す．その先のフレームは必要に応じて戻す．一方，継続のキャプチャの際はスタック上のフレームすべてをヒープへ退避した後，スタックを空にする．以上の動作によりフレームの共有が可能となり，フレームのコピー量を削減できる．一方，提案手法は L-closure を実現するためのものであり，共有は行わない．また，インクリメンタルスタック/ヒープ法は Scheme の変数などを持つようなリロケータブルなフレームを想定しており，C 言語のスタックを実行スタックとして利用できるわけではない．そこで，提案手法で提供される L-closure を用いれば，移植性の高いフレームアクセスが可能となり，文献 [32] のように一級継続が実装できる．（なお，文献 [32] は，真の末尾再帰に関する実装手法として提案されているため，一級継続の実装としては改善の余地がある．）

マルチスレッド言語 OPA とその負荷分散の実装 [28], [34] でも，インクリメンタルなフレーム管理を行っている．OPA 言語の実装では，Cilk 言語の実装 [6] と同様に，1 つのメソッドに対して速いバージョンの関数と遅いバージョンの関数を生成し，サスペンドされたスレッド（あるいはさらにスティールされたスレッド）の再開について遅いバージョンの関数を用いていた．本研究で提案する方式は，より汎用的な L-closure の実装手法としてインクリメンタルなフレーム管理を用いるとともに，コンパクトな引数結果補助関数のみを追加で生成することで，コードサイズを抑えている．

8. 議論

C 言語では，関数呼び出しが起こるたびに呼び出された関数のフレームがスタックへと積まれ，そのままリターンすることなしに関数呼び出しが起こり続けると，いずれスタック領域が溢れてしまう．これは，関数が末尾呼び出しや再帰呼び出しであっても同様であり，スタックの使用領域量については注意を払わなければならない．本来，関数の末尾呼び出しが起きると，呼び出し元の関数の引数や局

所変数の値といった情報は不要であるが，C の関数フレームはリターンアドレスを含んでいるため，単純に呼び出し元関数のフレームを削除するだけでは，呼び出し先の関数がリターンできない状態になってしまう．しかし，本研究で提案した実装モデルに変更を加えることで，C のスタックが溢れそうになれば明示的スタックへと内容を移し，かつ末尾呼び出しを行っている関数のフレームは明示的スタック上にも存在しないという状態にできる．

ここで，関数 f が関数 g を呼び出し，関数 g が関数 h を末尾呼び出しする場合を例に考える．関数 g が関数 h を末尾呼び出しするとき，その時点で g のフレームにある引数や局所変数の値といった情報はすでに不要である．今回提案した実装では，変換後関数 h の引数として明示的スタックのスタックポインタを渡しているが，これを明示的スタック上の g のフレームへのポインタを渡すように変更する．これにより，明示的スタック上の h のフレームは，g のフレームがあった場所に作られる，つまり g のフレームは潰される．このような状態で，入れ子関数呼び出し時と同じ，C のスタックの内容を明示的スタックへ移す操作を行うと，h のフレームの parent frame フィールドには f のフレームへのポインタが入り，h から f へと直接リターン（実際には引数結果補助関数および共通トランポリン関数を経由する）することができる．

LW-SC 言語にこのような機能を取り込むには，C 言語，および SC 言語には末尾呼び出しを示すキーワードや構文が存在しないため，新たに構文として定める必要がある．また，LW-SC では C のスタックの内容を明示的スタックへと移す操作は，入れ子関数呼び出し時に自動で起こる場合に限られているが，上記の目的を考えると，C のスタックが溢れそうな場合にスタック間の内容の移動を行いたいので，そのような場合に実行されるような機能を導入する必要がある．

9. おわりに

本研究では，合法的実行スタックアクセスを可能とする安全な計算状態操作機構 “L-closure” の変換ベース実装の，新しい実装モデルを提案し，SC 言語を利用して実際に L-closure を持つ SC 言語，LW-SC 言語を実装した．LW-SC は，最終的に C 言語へと変換されるため，新しいターゲットマシンへの移植やサポートが容易であるという特徴がある．

また，性能評価の結果，現在 L-closure のコンパイラベースの実装としてある，GCC 3 を拡張した実装よりも，呼び出しコストが低くなっていることが見てとれた．単に移植性が高いというだけでなく，使用用途によっては，コンパイラベースのものより実行時間を短くできると考えられる．

本研究では，上記の実装に加えて，L-closure の研究の一環として提案している “closure” のコンパイラベースの実装を，GCC-4.6.3 に対して行った．GCC 4 系列は，GCC 3 系列から内部構造が刷新されており，最適化技術が進化しており，GCC 3 系列から GCC 4 系列へと移植を行うことで，同じ closure でも実行時間を短くできると期待される．

これらの成果により，本研究は L-closure の実用性を高めることに貢献したといえる．今後の課題として，LW-SC に関しては，8 章で述べた関数が末尾呼び出しの場合の消費スペースの削減の実装を行いたい．また，GCC 4 系列に関しては L-closure の実装も行い，GCC 4 系列の最適化技術の恩恵を受けられるようにしたい．

謝辞　本研究の一部は，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）の助成を得て行った．L-closure の GCC 4 系列への実装におけるフロントエンドの開発を行っていた釜江典裕氏に感謝致します．

参考文献

[1] Baker, H.G.: CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A., *ACM SIGPLAN Notices*, Vol.30, pp.17–20 (1995).
[2] Boehm, H.-J. and Weiser, M.: Garbage Collection in an Uncooperative Environment, *Software Practice & Experience*, Vol.18, No.9, pp.807–820 (1988).
[3] Breuel, T.M.: Lexical Closures for C++, *Usenix Proceedings, C++ Conference*, pp.293–304 (1988).
[4] Clinger, W.D., Hartheimer, A.H. and Ost, E.M.: Implementation Strategies for First-Class Continuations, *Higher-Order and Symbolic Computation*, Vol.12, pp.7–45 (1999).
[5] Feeley, M.: A Message Passing Implementation of Lazy Task Creation, *Proc. International Workshop on Parallel Symbolic Computing: Languages, Systems, and Applications*, LNCS, No.748, pp.94–107, Springer-Verlag (1993).
[6] Frigo, M., Leiserson, C.E. and Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language, *ACM SIGPLAN Notices* (*PLDI'98*), Vol.33, No.5, pp.212–223 (1998).
[7] Goldstein, S.C., Schauser, K.E. and Culler, D.E.: Lazy Threads: Implementing a Fast Parallel Call, *Journal of Parallel and Distributed Computing*, Vol.3, No.1, pp.5–20 (1996).
[8] Hanson, D.R. and Raghavachari, M.: A Machine-Independent Debugger, *Software — Practice & Experience*, Vol.26, No.11, pp.1277–1299 (1996).
[9] Henderson, F.: Accurate Garbage Collection in an Uncooperative Environment, *Proc. 3rd International Symposium on Memory Management*, pp.150–156 (2002).
[10] 平石　拓，李　暁ろ，八杉昌宏，馬谷誠二，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.SIG 1 (PRO 24), pp.40–56 (2005).
[11] Hiraishi, T., Yasugi, M., Umatani, S. and Yuasa, T.: Backtracking-based Load Balancing, *Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (*PPoPP 2009*), pp.55–64 (2009).
[12] Hiraishi, T., Yasugi, M. and Yuasa, T.: Implementing S-Expression Based Extended Languages in Lisp, *Proc. International Lisp Conference*, pp.179–188 (2005).
[13] Hiraishi, T., Yasugi, M. and Yuasa, T.: A

Transformation-Based Implementation of Lightweight Nested Functions, *IPSJ Digital Courier*, Vol.2, pp.262–279 (2006). [*IPSJ Trans. Programming*, Vol.47, No.SIG 6 (PRO 29), pp.50–67 (2006)].

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

[15] Mohr, E., Kranz, D.A. and Halstead, Jr., R.H.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, *IEEE Trans. Parallel and Distributed Systems*, Vol.2, No.3, pp.264–280 (1991).

[16] 大林竜太，平石　拓，八杉昌宏，馬谷誠二，湯淺太一：遅延分割型負荷分散フレームワークの試験実装，情報処理学会第 55 回プログラミング研究会（SWoPP2005）(2005).

[17] Pettyjohn, G., Clements, J., Marshall, J., Krishnamurthi, S. and Felleisen, M.: Continuations from Generalized Stack Inspection, *Proc. 10th ACM SIGPLAN International Conference on Functional Programming* (*ICFP 2005*), pp.216–227 (2005).

[18] Peyton Jones, S., Ramsey, N. and Reig, F.: C--: A Portable Assembly Language that Supports Garbage Collection, *International Conference on Principles and Practice of Declarative Programming*, pp.1–28 (1999).

[19] Plevyak, J., Karamcheti, V., Zhang, X. and Chien, A.A.: A Hybrid Execution Model for Fine-Grained Languages on Distributed Memory Multicomputers, *Supercomputing'95* (1995).

[20] Ramsey, N. and Jones, S.P.: A Single Intermediate Language That Supports Multiple Implementations of Exceptions, *Proc. ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pp.285–298 (2000).

[21] Sekiguchi, T., Sakamoto, T. and Yonezawa, A.: Portable Implementation of Continuation Operators in Imperative Languages by Exception Handling, *Advances in Exception Handling Techniques* (*the book grow out of a ECOOP 2000 workshop*), *LNCS 2022*, pp.217–233, Springer (2001).

[22] Strumpen, V.: Compiler Technology for Portable Checkpoints (1998), available from ⟨http://theory.lcs.mit.edu/~strumpen/porch.ps.gz⟩.

[23] Strumpen, V.: Indolent Closure Creation, Technical Report MIT-LCS-TM-580, MIT (1998).

[24] 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).

[25] Taura, K., Tabata, K. and Yonezawa, A.: StackThreads/MP: Integrating Futures into Calling Standards, *Proc. ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming* (*PPoPP*), pp.60–71 (1999).

[26] Taura, K. and Yonezawa, A.: Fine-grain Multithreading with Minimal Compiler Support — A Cost Effective Approach to Implementing Efficient Multithreading Languages, *Proc. Conference on Programming Language Design and Implementation*, pp.320–333 (1997).

[27] 八杉昌宏，小宮常康，湯淺太一：入れ子関数を利用する動的負荷分散と高水準記述，情報処理学会論文誌：コンピューティングシステム，Vol.45, No.SIG 11 (ACS 7), pp.368–377 (2004).

[28] 八杉昌宏，馬谷誠二，鎌田十三郎，田畑悠介，伊藤智一，小宮常康，湯淺太一：オブジェクト指向並列言語 OPA のためのコード生成手法，情報処理学会論文誌：プログラミング，Vol.42, No.SIG 11 (PRO 12), pp.1–13 (2001).

[29] Yasugi, M., Hiraishi, T. and Yuasa, T.: Lightweight Lexical Closures for Legitimate Execution Stack Access, *Proc. 15th International Conference on Compiler Construction* (*CC2006*), LNCS, No.3923, pp.170–184, Springer-Verlag (2006).

[30] Yasugi, M., Komiya, T. and Yuasa, T.: An Efficient Load-Balancing Framework Based on Lazy Partitioning of Sequential Programs, *Proc. Workshop on New Approaches to Software Construction*, pp.65–84 (2004).

[31] 八杉昌宏：L-Closure：安全な計算状態操作機構，情報処理，Vol.51, No.7, p.885 (2010).

[32] 八杉昌宏，小島啓史，小宮常康，平石　拓，馬谷誠二，湯淺太一：L-Closure を用いた真に末尾再帰的な Scheme インタプリタ，情報処理学会論文誌プログラミング，Vol.3, No.5, pp.1–17 (2010).

[33] 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), pp.63–83 (2008).

[34] 馬谷誠二，八杉昌宏，小宮常康，湯淺太一：オブジェクト指向並列言語 OPA のための遅延正規化手法，情報処理学会論文誌：プログラミング，Vol.45, No.SIG 5 (PRO 21), pp.12–25 (2004).

[35] 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, pp.225–238 (2006).

[36] 平石　拓，八杉昌宏，湯淺太一：SC 言語処理系における変形規則の再利用機構，コンピュータソフトウェア，Vol.28, No.1, pp.258–271 (2011).

付　　録

A.1　図 3 の変換例

```
typedef long size_t;
struct fframe;

/* 正規化された関数の型 */
typedef int (*normfn_t)( struct fframe * );

/* 関数とフレームへのポインタのペア */
typedef struct {
  normfn_t fn;
  struct fframe *fr;
} closure_t;

/* 入れ子関数, もしくは引数結果補助関数へのポインタ */
typedef union {
  normfn_t fn;
  closure_t *clos;
} fn_or_clos_t;
typedef double Align_t;

/* 明示的スタック上のフレームの定型部分 */
struct fframe{
  struct fframe *parent_fr;
  struct fframe *child_fr;
  struct fframe *xfp;
  fn_or_clos_t f_or_c;
  int call_id;
};

/* 共通トランポリン関数 */
void trampoline (char *esp);

```

```
/* 宣言 */
struct h_frame;
struct foo_frame;
struct g1_in_foo_frame;
struct main_frame;
int h_comp( struct h_frame * );
int foo_comp( struct foo_frame * );

/* 関数 h のフレーム */
struct h_frame{
  struct fframe *parent_fr;
  struct fframe *child_fr;
  struct fframe *xfp;
  fn_or_clos_t f_or_c;
  int call_id;
  int retval;
  int i;
  closure_t *g;
  int tmp;
};

int h (char *esp, int i, closure_t *g)
{
  int tmp;
  size_t esp_flag = (size_t )esp&3;
  struct h_frame *efp;
  char *argp;
  /* 下位ビットから開始か再開かを判断 */
  if(esp_flag )
    {
      /* 再開の場合の処理 */
      /* 下位ビットの修正 */
      esp = (char *)((size_t )esp^esp_flag);
      /* 明示的スタック上のフレームへのポインタ */
      efp = (struct h_frame *)esp;
      /* h のフレームを確保 */
      esp = (char *)((Align_t *)esp
            +(sizeof(struct h_frame )
              +sizeof(Align_t )+-1)
            /sizeof(Align_t ));
      /* 再開位置へ goto */
    LGOTO: switch( (*efp).call_id )
        {
        case 1:
          goto L_CALL;
        }
      goto L_CALL;
    }
  else;efp = (struct h_frame *)esp;
  /* h のフレームを確保 */
  esp = (char *)((Align_t *)esp
        +(sizeof(struct h_frame )
          +sizeof(Align_t )+-1)
        /sizeof(Align_t ));
  {
    {
      /* 入れ子関数呼び出しの処理 */
      /* 仮フレームを作成 */
      struct temp_frame{
        struct fframe *parent_fr;
        struct fframe *child_fr;
        struct fframe *xfp;
        fn_or_clos_t f_or_c;
        int call_id;
        int retval;
        int arg;
      };
      struct temp_frame *tfp;
      tfp = (struct temp_frame *)esp;
      /* 仮フレームに引数を保存 */
      tfp->arg = i;
      /*「特別値としての仮フレームへのポインタ自身」
         を保存 */
      tfp->child_fr = (struct fframe *)tfp;
      /* ペアのアドレスを保存 */
      (tfp->f_or_c).clos = g;
      /* h の引数, 局所変数を保存 */
      efp->tmp = tmp;
      efp->g = g;
      efp->i = i;
      /* 仮フレームの parent frame フィールドに
         h のフレームのアドレスを保存 */
      tfp->parent_fr = (struct fframe *)efp;
      /* h のフレームの child frame フィールドに
         仮フレームのアドレスを保存 */
      efp->child_fr = (struct fframe *)tfp;
      /* 再開位置番号を保存 */
      efp->call_id = 1;
      /* 引数結果補助関数 h のアドレスを保存 */
      (efp->f_or_c).fn = (normfn_t )h_comp;
      /* 特別値を return */
      return (int )0-1;
    L_CALL: ;
      /* 再開時にここに戻ってくる */
      /* 局所変数の回復 */
      tmp = efp->tmp;
      g = efp->g;
      i = efp->i;
      /* 返り値の取り出し */
      tfp = (struct temp_frame *)efp->child_fr;
      tmp = tfp->retval;
    }
    {
      efp->child_fr = 0;
      return tmp;
    }
  }
}

/* 引数結果補助関数 h */
int h_comp (struct h_frame *efp)
{
  int tmp1;
  /* 引数を取り出し h を呼び出す */
  tmp1 = h((char *)efp+1, efp->i, efp->g);
  /* 返り値が特別値であるかを判定 */
  if(tmp1 == (int )0-1&&efp->child_fr )
    return 1;
  else;efp->retval = tmp1;
  return 0;
}

struct g1_in_foo_frame{
  struct fframe *parent_fr;
  struct fframe *child_fr;
  struct foo_frame *xfp;
  fn_or_clos_t f_or_c;
  int call_id;
  int retval;
  int b;
};

struct foo_frame{
  struct fframe *parent_fr;
  struct fframe *child_fr;
  struct fframe *xfp;
  fn_or_clos_t f_or_c;
  int call_id;
  int retval;
  closure_t g1;
  int a;
  int tmp2;
```

```
  int x;
};

/* 変換前は入れ子関数 g1 */
int g1_in_foo (struct g1_in_foo_frame *efp)
{
  int b = efp->b;
  char *esp = (char *)efp;
 LGOTO: ;
  /* 変換前に入れ子関数 g1 を定義していた
     関数 foo のフレーム */
  struct foo_frame *xfp = efp->xfp;
  efp = (struct g1_in_foo_frame *)esp;
  esp = (char *)((Align_t *)esp
        +(sizeof(struct g1_in_foo_frame )
          +sizeof(Align_t )+-1)
        /sizeof(Align_t ));
  (xfp->x)++;
  {
    efp->retval = xfp->a+b;
    return 0;
  }
  return 0;
}

int foo (char *esp, int a)
{
  int tmp2;
  int x;
  size_t esp_flag = (size_t )esp&3;
  struct foo_frame *efp;
  if(esp_flag )
    {
      esp = (char *)((size_t )esp^esp_flag);
      efp = (struct foo_frame *)esp;
      esp = (char *)((Align_t *)esp
            +(sizeof(struct foo_frame )
              +sizeof(Align_t )+-1)
            /sizeof(Align_t ));
    LGOTO: switch( (*efp).call_id )
        {
        case 1:
          goto L_CALL2;
        }
      goto L_CALL2;
    }
  else;efp = (struct foo_frame *)esp;
  esp = (char *)((Align_t *)esp
        +(sizeof(struct foo_frame )
          +sizeof(Align_t )+-1)
        /sizeof(Align_t ));
  x = 0;
  {
    /* 返り値が特別値であるかを判定 */
    if(__builtin_expect(
         (tmp2 = h(esp, 10, &efp->g1))
                 == (int )0-1, 0)
       &&__builtin_expect(
         ((struct fframe *)esp)->child_fr, 1) )
      {
        /* 特別値だった場合の処理 */
        /* 引数および局所変数の保存 */
        efp->x = x;
        efp->tmp2 = tmp2;
        efp->a = a;
        /* 入れ子関数の初期化 */
        efp->g1.fn = (normfn_t )g1_in_foo;
        efp->g1.fr = (struct fframe *)efp;
        /* 引数結果補助関数 foo のアドレスを保存 */
        (efp->f_or_c).fn = (normfn_t )foo_comp;
        /* h のフレームの parent frame フィールドに
           foo のフレームのアドレスを保存*/
        ((struct fframe *)esp)->parent_fr
             = (struct fframe *)efp;
        /*foo のフレームの child frame フィールドに
          h のフレームのアドレスを保存*/
        efp->child_fr = (struct fframe *)esp;
        /* 再開位置番号の保存 */
        efp->call_id = 1;
        /* 特別値を return */
        return (int )0-1;
      L_CALL2: ;
        /* 再開時にここに戻ってくる */
        /* 局所変数の復帰 */
        x = efp->x;
        tmp2 = efp->tmp2;
        a = efp->a;

        /* 返り値の取り出し */
        struct child_frame{
          struct fframe *parent_fr;
          struct fframe *child_fr;
          struct fframe *xfp;
          fn_or_clos_t f_or_c;
          int call_id;
          int retval;
        };
        struct child_frame *cfp;
        cfp = (struct child_frame *)efp->child_fr;
        tmp2 = cfp->retval;
      }
    else;{
      efp->child_fr = 0;
      return tmp2+x;
    }
  }
}

int foo_comp (struct foo_frame *efp)
{
  int tmp1;
  tmp1 = foo((char *)efp+1, efp->a);
  if(tmp1 == (int )0-1&&efp->child_fr )
    return 1;
  else;efp->retval = tmp1;
  return 0;
}

struct main_frame{
  struct fframe *parent_fr;
  struct fframe *child_fr;
  struct fframe *xfp;
  fn_or_clos_t f_or_c;
  int call_id;
  int retval;
  int tmp3;
};

int main ()
{
  int tmp3;
  /* 明示的スタック */
  char estack[262144];
  char *esp = estack;
  struct main_frame *efp;
 LGOTO: ;
  efp = (struct main_frame *)esp;
  esp = (char *)((Align_t *)esp
        +(sizeof(struct main_frame )
          +sizeof(Align_t )+-1)
        /sizeof(Align_t ));
  {
```

```
316     if(__builtin_expect(
317          (tmp3 = foo(esp, 1)) == (int )0-1, 0)
318        &&__builtin_expect(
319            ((struct fframe *)esp)->child_fr, 1) )
320       {
321         efp->tmp3 = tmp3;
322         /* foo のフレームの
323            parent frame フィールドに 0 を保存 */
324         ((struct fframe *)esp)->parent_fr = 0;
325         efp->child_fr = (struct fframe *)esp;
326         /* 共通トランポリン関数の call */
327         trampoline(esp);
328         tmp3 = efp->tmp3;
329
330         struct child_frame{
331           struct fframe *parent_fr;
332           struct fframe *child_fr;
333           struct fframe *xfp;
334           fn_or_clos_t f_or_c;
335           int call_id;
336           int retval;
337         };
338         struct child_frame *cfp;
339         cfp = (struct child_frame *)esp;
340         tmp3 = cfp->retval;
341       }
342     else;{
343       efp->child_fr = 0;
344       return tmp3;
345     }
346   }
347 }
348
349 /* 共通トランポリン関数 */
350 void trampoline(char *esp){
351   struct fframe *fr = (struct fframe *)esp;
352   for(;;){
353     /* child frame フィールドが
354        「特別値としての仮フレームへのポインタ自身」
355        に行き着くまで辿る */
356     while(fr != fr->child_fr) fr = fr->child_fr;
357     /* ペアのアドレスを取り出す */
358     closure_t *clos = fr->f_or_c.clos;
359     fr->f_or_c.fn = clos->fn;
360     /* 静的リンクを保存 */
361     fr->xfp = clos->fr;
362     /* 入れ子関数の開始番号 */
363     fr->call_id = 0;
364     /* 引数結果補助関数 or 変換後入れ子関数の call
365      返り値から入れ子関数呼び出し指示であるかを判断*/
366     while((fr->f_or_c.fn)(fr) == 0)
367       if(!(fr = fr->parent_fr))
368         return;
369   }
370 }
```

田附 正充

1987 年生．2011 年京都大学工学部情報学科卒業．2013 年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修了．同年より楽天株式会社に勤務．プログラミング言語処理系に興味を持つ．

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．1993～1995 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講師．2003 年同大学助教授．2007 年同大学准教授．2012 年より九州工業大学大学院情報工学研究院教授．博士（理学）．1998～2001 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．平成 21 年度情報処理学会論文誌プログラミング優秀論文賞受賞．

平石 拓　（正会員）

1981 年生．2003 年京都大学工学部情報学科卒業．2005 年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修了．2008 年同専攻博士課程修了．2007～2008 年日本学術振興会特別研究員．同年より京都大学学術情報メディアセンター助教．博士（情報学）．並列プログラミング言語，高性能計算に興味を持つ．平成 21 年度情報処理学会論文誌プログラミング優秀論文賞受賞．日本ソフトウェア科学会会員．

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．同年同大学院情報学研究科産学官連携研究員．2005 年同研究科助手．2007 年より同研究科助教．博士（情報学）．プログラミング言語，コンパイラ，並列/分散処理に興味を持つ．日本ソフトウェア科学会，ACM 各会員．