最新比較評価向け M-Closure の再実装

廣田 陽紀　八杉 昌宏　平石 拓

高水準プログラミング言語の実装におけるマシン独立な低水準言語として C 言語を使いたいことがある．ごみ集めなどの高水準サービスの実現には実行スタック中の存命の変数の値にアクセスする必要があり，そのような合法的実行スタックアクセスのための言語機構として計算状態操作機構が八杉らにより提案されている．計算状態操作機構として GNU C コンパイラ（GCC）がサポートする入れ子関数（から生成されるクロージャ）が利用可能なことから，クロージャ生成コストを削減した M-closure が提案され，GCC 3.4.6 および GCC 4.6.3 の改造により実装されてきた．本研究では GCC 13 系列など最新の GCC の改造により M-closure を再実装し，L-closure などその他の計算状態操作機構あるいは GCC 改造によらない変換ベースの実装との比較研究を可能とする．

1　はじめに

様々な計算機（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，高水準プログラミング言語の実装においては，ある程度，移植性が良くマシン独立な低水準言語として C 言語を使いたいことがある．高水準言語から C 言語への翻訳系，あるいは C 言語による高水準言語インタプリ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，実行スタックを使う．

高水準言語に備わっている高水準実行時サービスの中には，ごみ集め，負荷分散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がある．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，実行スタック深くに眠る呼び出し元の存命の変数の値にアクセスするとしても，実行スタックに直接（非合法的に）アクセスするべきではない．

合法的実行スタックアクセスのための言語機構として計算状態操作機構 [3][10][4][7][6][5] が提案されている．計算状態操作機構を備えた拡張 C 言語（など）をマシン独立な低水準言語として用いるのである．

本研究では，計算状態操作機構のうち，“M-closure” に着目する．従来は “closure” [10][11][7] と称していたが，用語が一般的過ぎた点を解消したいことと，別途提案されている “L-closure” [10][3][11] との区別をより明確にしたいことから，最近の研究 [6][5] と同様に “M-closure” という用語を用いることとする．

計算状態操作機構として GNU C コンパイラ（GCC）がサポートする入れ子関数（から生成されるクロージャ） [1] が利用可能である．（間接的に）呼び出された入れ子関数は親関数の変数に合法的にアクセス可能となる．

M-closure は，GCC のクロージャ生成に伴うコストを削減しており，GCC 3.4.6 の改造 [10] および GCC 4.6.3 の改造 [11] により実装されてきた．また，本格的な変換を S 式の変形として実現した形の M-closure [7] も実装されている．M-closure は，後

M-Closures for the Latest Comparative Evaluation, Re-implemented

Haruki Hirota, 九州工業大学大学院情報工学府, Graduate School of Computer Science and Systems Engineering, Kyushu Institute of Technology.

Masahiro Yasugi, 九州工業大学大学院情報工学研究院, Dept. of Computer Science and Networks, Kyushu Institute of Technology.

Tasuku Hiraishi, 京都橘大学工学部情報工学科, Dept. of Information and Computer Science, Faculty of Engineering, Kyoto Tachibana University.

述する L-closure [10][3][11] のように積極的に生成・維持コストを削減しない分，呼び出しコストが低く，また変換に基づく L-closure の実装 [3][11] とは異なり遅延判定コスト [7] を要しないという特徴がある．

本研究の貢献は，GCC 13 系列など最新の GCC の改造により M-closure を再実装し，L-closure [10][3][11] などその他の計算状態操作機構 [1][6][5] あるいは GCC 改造によらない変換ベースの実装 [7] との比較研究を可能とすることである．

本稿の構成は以下の通りである．2 章では，計算状態操作機構の仕様・分類を述べ，その利用例について述べる．3 章では GCC 3.4.6，GCC 4.6.3 の改造による M-closure の既存実装について述べ，4 章で最新バージョンにおける M-closure の再実装について述べる．5 章では今後の展望について述べる．

2 計算状態操作機構

2.1 仕様・分類

2.1.1 入れ子関数の形態

2.1.1.1 GCC 拡張の入れ子関数

GCC では，C への拡張として入れ子関数（nested function） [1] 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異なり，以下の g のように，

```
void f(int a){ void g(){ a++; } h(g); }
```

親関数 f の中に入れ子に書かれていて，親関数の a のような lexical スコープの変数の値に合法的にアクセスできる．また，(h やその先から）間接的に呼び出せ，関数ポインタとしても通常のトップレベルの関数との相互運用性が確保されている．

GCC の実装で用いられる「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命令列であり，スタック上に動的に生成される．ただし，データ／命令キャッシュのフラッシュや実行スタックを命令実行可能とするためなどの高い生成コストを要する．

2.1.1.2 L-Closure

L-closure は軽量 lexical closure であり，

```
void f(int a){
  void lightweight g(){ a++; } h(g); }
```

のように GCC の入れ子関数とは別の「型」として，通常のトップレベルの関数との相互運用性はないとする．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．

2.1.1.3 M-Closure

M-closure は L-closure における `lightweight` の代わりに `closure` と書くこととし，各種コストについて L-closure より中庸的な機構とする．

2.1.2 持続型例外処理機構

別形態の計算状態操作機構として提案されている持続型例外処理機構 [5] は，非局所脱出することなく例外ハンドラを呼び出せるとしたもので，ポインタではなく動的スコープに基づくためより安全である．

2.1.3 ラムダ式

C++言語のラムダ式は入れ子関数に似た形態の計算状態操作機構として利用できる．ラムダ式の値（クロージャ）を渡すには，`auto` を用いるか `std::function` クラステンプレートを用いることが多い．

2.2 計算状態操作機構の利用例

2.2.1 並列言語 Tascell や HOPE

並列言語 Tascell [2] は，ワークスティールに基づく負荷分散を提供する．Tascell ワーカは，タスク生成を要求されない限り，自身の実行スタックを用いて逐次計算を行う．要求されたら，計算状態操作機構を利用することで一時的バックトラックを行い，最古のタスク生成可能状態を復元することで，期待値としてより大きなタスクを生成する．

HOPE [8] は階層的計算省略に基づく耐障害並列実行モデルであり，既存の並列実行モデルが「複数ワーカで仕事を分担」に基づくのとは全く逆に，耐障害性のための完全冗長実行からの階層的計算省略に基づく．どの HOPE ワーカも分割統治における全範囲（同一プログラム内の全計算）を異なる順序で担当しつつ，実行時に他ワーカから結果を得た部分の計算は省略する．高速化のため，HOPE 言語の実装においては，複数の実行モードを設けて結果の送信や確認を行うかを判断しており，遅延された実行モードの動的切替（に伴う実行スタック中の結果などへのアクセス）に計算状態操作機構を利用している．

2.2.2 JALKD/XC と JAKLD/SC

L-closure（や M-closure）で実装した Scheme インタプリタ JAKLD/XC [9] では，計算状態操作機構をコピー型ごみ集めや一級継続のキャプチャに用いている．SC 言語（S 式ベース C 言語）に入れ子関数を備えさせた拡張 SC 言語 SC-NF で再実装した JAKLD/SC [7] では，標準 C 言語への変換に基づく SC-NF の L-closure モデルや M-closure モデルによる実装も利用可能となっている．

2.2.3 Tascell++

Tascell [2] にインスパイアされた C++プログラミングによる新しいワークスティールフレームワークとして Tascell++ [6] が提案されている．計算状態操作機構としてのラムダ式の値の受け取りの効率化のため，`std::function` クラステンプレートに基づく型の代わりに，型消去技法よるカスタマイズされた仮想関数と派生クラステンプレートが用いられている．

3 M-Closure の既存実装

3.1 GCC 3.4.6

2007 年頃に GCC 3.4.6 改造により L-Closure を実装する際，M-closure についても実装していた [10]．GCC 3.4.6 では，Yacc 上位互換の Bison により生成された構文解析器が使われている．C プログラムを構文解析した結果得られる構文木（の主なノード）は `tree` という型で表現されている．tree 表現上の最適化はインライン展開や定数畳み込み程度であり，tree 表現から RTL（regiter transfer language）表現に展開（expand）された後に RTL 表現上で様々な最適化が行われる．RTL 表現上でレジスタ割り当てが行われた後，アセンブリ言語のコードが生成される．

3.1.1 構文解析

GCC 3.4.6 の構文解析を M-closure に対応させるため，Bison の入力ファイルに生成規則を追加するような形で構文の簡単な拡張を行うとともに，関数型の tree 表現を拡張し，M-closure 型であることも表現できるように改造していた．ただし，C 言語の構文における関数型は関数呼び出しの形を逆に読む形をしているため，構文解析中の形の上だけの関数呼び出しの tree 表現にも M-closure であることが付加される．

3.1.2 型検査

tree 表現上での型検査の際に M-closure 型を通常の関数型と区別するように改造していた．

3.1.3 RTL

GCC 3.4.6 の tree 表現から RTL 表現への展開を M-closure に対応させるため，通常の関数呼び出しとは別の M-closure 呼び出しを行えるようにするとともに，GCC のトランポリンの代わりに M-closure を設置できるように改造していた．

前者については，一見，関数呼び出しのように見える場合でも，tree 表現される呼び出し対象の型が M-closure 型である場合は，呼び出し準備として M-closure へのポインタ（従来の関数ポインタに相当）に関して，M-closure の 2 ワード目から静的リンク（GCC においては `static_chain`）を取り出して特定のレジスタにセットするような RTL 命令と，1 ワード目から呼び出し対象を関数ポインタとして取り出す RTL 命令を生成する．関数ポインタで呼び出せる関数は，特定のレジスタで静的リンクを受け取ることで，外側の関数の局所変数などにアクセス可能となる．tree 表現における入れ子関数から外側の関数の変数へのアクセスは RTL 表現では静的リンクを介したアクセスに修正される．

後者については，関数ポインタと静的リンクが埋め込まれたトランポリンの代わりに，その 2 ワードを単に設置する（RTL 命令を生成する）だけである．

GCC 3.4.6 改造の時点では，M-closure よりも L-closure [10] に注目しており，再利用性や実行性能を高めるために，RTL 表現の事実上の拡張（実際には拡張なしで特殊なパターンの利用）を含め，はるかに複雑な改造が行われていた．

3.1.4 コード生成

L-closure についてはアセンブリコード生成部分も拡張・改造する必要があったのに対し，M-closure については RTL 表現への展開までで完結しており，コード生成部分の改造は必要がなかった．

3.2 GCC 4.6.3

2012 年頃の GCC 4.6.3 改造では M-closure についてのみ実装していた [11]．GCC 4 系列では

c-parser.c として書かれた構文解析器が使われるようになった．C プログラムを構文解析した結果得られる構文木（の主なノード）は tree という型以外に，構造体も適材適所で併用されるようになった．GCC 4 系列から標準的な tree 表現は GENERIC と呼ばれるようになり，RTL 表現への展開前に GIMPLE と呼ばれる中間表現が用いられるようになった．GENERIC から GIMPLE への変換後，入れ子関数がアクセスする変数を構造体のフィールドに置き換えるような変換などが施される．さらに，high GIMPLE から low GIMPLE への低水準化が行われた後，low GIMPLE 表現上で様々な最適化が行われる．low GIMPLE 表現から RTL 表現への展開（expand）は，tree 表現を介しての展開も併用して行われる．RTL 表現上でも様々な最適化が行われ，RTL 表現上でレジスタ割り当てが行われた後，アセンブリコードが生成される．

3.2.1 構文解析と GENERIC

M-closure の構文は GCC 3.4.6 の改造のときと同じ設計とし，GCC 4.6.3 では手書きの構文解析器 c-parser.c を改造していた．M-closure 用に追加されるべき生成規則に対しても生成規則右辺の共通プレフィックス部分の構文解析はそのままであり，丸カッコなら関数型，角カッコなら配列型とするような場合分けのところに，M-closure のキーワード（closure あるいはピリオドも可）の場合を加える程度であった．また，構文解析の上だけの構造体に M-closure であることが付加されるように改造していた．

GENERIC と呼ばれるようになった tree 表現については，関数型の tree 表現を拡張し，M-closure 型であることも表現できるように改造していたのは GCC 3.4.6 の場合と同様である．tree 表現上での型検査についても GCC 3.4.6 の場合と同様である．

3.2.2 GIMPLE

GCC 4.6.3 では GCC 3.4.6 にはなかった GIMPLE という GENERIC よりもシンプルな命令の中間表現が用いられるようになったが，型を表す tree 表現は GIMPLE でも GENERIC の表現が用いられているため，M-closure 型のための GIMPLE 独自の拡張は必要なかった．GENERIC から GIMPLE への変換（gimplify）についても M-closure 用の独自の修正の必要はなかった．

GENERIC から GIMPLE への変換後，入れ子関数がアクセスする変数を構造体のフィールドに置き換えるような変換などが施されるが，この部分は修正の必要がなかった．ただし，入れ子関数自体のアクセスについては，構造体のトランポリン型フィールドに置き換えるような変換などが施される．これらの置き換えは，GCC 3.4.6 の RTL とは異なり，GCC 4.6.3 では GIMPLE においてビルトイン関数も利用して行われる．

構造体のトランポリン型フィールドに関する M-closure 対応としては，トランポリン型フィールドの代わりに M-closure 型フィールドとなるように，参照時に__builtin_adjust_trampoline 関数の代わりに調整なしとなるように，初期化時に__builtin_init_trampoline 関数の代わりに，__builtin_init_closure 関数を用いるように改造していた．

high GIMPLE から low GIMPLE への低水準化や low GIMPLE 表現上で様々な最適化に対するこれらの改造の影響はなかった．

3.2.3 RTL とビルトイン関数

GCC 4.6.3 での low GIMPLE 表現から RTL 表現への展開（expand）は，tree 表現を介しての展開も併用して行われており，GCC 3.4.6 における tree 表現から RTL への展開（expand）と共通する部分が多くみられた．例えば，一見，関数呼び出しのように見える場合でも，tree 表現される呼び出し対象の型が M-closure 型である場合は，呼び出し準備として M-closure へのポインタに関して，M-closure の 2 ワード目から静的リンクを取り出して特定のレジスタにセットするような RTL 命令と，1 ワード目から呼び出し対象を関数ポインタとして取り出す RTL 命令を生成する．ただし GCC 4.6.3 では，特定のレジスタの番号が関数定義以外に関数型からも得られるようにするための軽微な改造を要した．

GCC 4.6.3 での RTL 表現への展開時に，__builtin_init_closure 関数なども展開されるように改造した．関数ポインタと静的リンクが埋め込まれたトランポリン（命令列）の代わりに，その 2 ワー

ドを単に設置する（RTL 命令を生成する）だけなのは，GCC 3.4.6 と同様である．

GCC 4.6.3 でも M-closure については RTL 生成までで完結しており，コード生成部分は（レジスタ番号を得る部分を除き）改造する必要がなかった．

4 M-Closure の再実装

4.1 GCC 13.3

2024 年現在において，GCC 13.3 改造による M-closure の再実装を行った．改造パッチの行数を小節名に付す．

4.1.1 GCC 4.6.3 との違い

GCC 13.3 を C コンパイラとしてみたとき，2012 年の GCC 4.6.3 との相違点はそれほど多くない．以下に主要な相違点を整理する．

- 1 点目として，GCC 4.6.3 ではコンパイラが C 言語で記述されていたのに対し，GCC 13.3 では C++言語で記述されている点がある．
- 2 点目として，トランポリンに代わり得るものとして descriptor と呼ばれる仕組みが特に Ada コンパイラ向けに準備されている点がある．descriptor は静的リンクと入れ子関数本体のペアを用いる点で M-closure と同様であるが，関数アドレスの未使用ビットを見て通常の関数ポインタであるか descriptor であるかを判別することが考えられている．
- 3 点目として，tree 表現上の操作で動的なデータ型検査が追加可能となっている点がある．
- 4 点目として，「関数定義または関数型」を扱えるようになった部分が増えている点がある．

4.1.2 構文解析と GENERIC（546 行）

M-closure の構文は GCC 3.4.6，GCC 4.6.3 の改造のときと同じ設計としつつ，`closure` というキーワードが GCC の外部ライブラリのソースプログラムで使われたりするようにもなっていたため，`mclosure` というキーワードに変更した．

GCC 13.3 の構文解析器は `c-parser.cc` という C++プログラムに変わっているものの，GENERIC の拡張や型検査を含め，GCC 4.6.3 を構成する C プログラムの改造とほぼ同様の改造となった．M-closure 型であるか確認するところでは，相違点 3 点目の検査の対象となるようにし，本実装の開発効率を高めた．

4.1.3 GIMPLE（140 行）

GCC 13.3 において，トランポリンと descriptor（相違点 2 点目）のどちらを利用するかは，ソース言語を単位として決定すべきとされており，特に，Ada コンパイラ向けの descriptor の利用が想定されている．つまり，C コンパイラとして用いるときは descriptor の利用が想定されていない．

GCC 13.3 における構造体のトランポリン型フィールドに関する M-closure 対応としては，トランポリン型フィールドの代わりに M-closure 型フィールドとなるように，参照時に`__builtin_adjust_trampoline` 関数の代わりに`__builtin_adjust_closure` 関数を用いるように，初期化時に`__builtin_init_trampoline` 関数の代わりに，`__builtin_init_closure` 関数を用いるようにより統一的に改造した．これは，descriptor に関する場合が GCC 13.3 では加わっており，それと統一的となるように M-closure に関する場合を加えたためである．

4.1.4 RTL とビルトイン関数（149 行）

GCC 13.3 での low GIMPLE 表現から RTL 表現への展開（expand）は，GCC 4.6.3 と共通する部分が多くみられた．GCC 13.2 では「関数定義または関数型」を扱えるようになった部分（相違点 4 点目）が増えており，その扱いのための軽微な改造が不要となった．

GCC 13.3 での RTL 表現への展開時に，`__builtin_adjust_closure` 関数なども展開されるように改造した．GCC 13.3 でも M-closure については RTL 生成までで完結しており，コード生成部分は改造する必要がなかった．

4.2 移植性

GCC 13.3 改造における M-closure 実装は，RTL 生成までで完結しており，改造前の GCC 13.3 が入れ子関数を含めて動作可能な環境では基本，動作可能な移植性を持つといえる．

2024 年現在，改造前の GCC 13.3 がそのままで

は動作可能でない環境としては，Apple Silicon の macOS（Darwin）環境がある．macOS では Xcode で Clang コンパイラが利用可能であるが，GCC の入れ子関数拡張はサポートされてない．Apple Silicon 向けの GCC 13.3 が Homebrew からパッケージで提供されているものの，GCC 13.3 ソースに対して Iain Sandoe 氏によるパッチが用いられている．その目的の 1 つは，Apple Silicon の Darwin 環境では実行スタックを実行可能にすることが禁じられており，トランポリンの本体を実行スタックではなくヒープに設置することである．

Iain Sandoe 氏のパッチ適用後の GCC 13.3 ソースにさらに本研究の M-closure パッチを当てたところ，ヒープ設置のトランポリンによる条件分岐やインデントの違いで，2 箇所のみ手動でパッチを当てることになったが，ビルドに成功している．また少し話は変わるが，調査の過程で GCC 14.2 ソースに対してもパッチを当ててみたところ，手動でバッチを当てるのは数箇所程度で済みそうである．

4.3 動作確認

GCC 13.3 改造による M-closure 実装は，Intel Xeon Gold の Ubuntu 20.04 LTS 環境，AMD EPYC の CentOS 7.4 環境，Apple M2 Ultra の macOS Sonoma 14.6.1 (Darwin 23.6.0) において，M-closure の間接呼び出しを含むプログラムで動作確認した．

5 今後の展望：比較研究へ

今後の展望の 1 つに，M-closure 間の比較研究がある．改造済 GCC-4.6.3 と改造済 GCC-13.3 の比較だけでなく，SC-NF プログラムから M-closure モデルによって本格的に変換された C プログラムを GCC 13.3 でコンパイルした場合と，本研究で再実装した M-closure（改造済 GCC 13.3 でコンパイルした場合）の比較という「対等に最新の」比較もある．

2 つ目として，L-closure などその他の計算状態操作機構との比較においても，本研究で再実装した M-closure を安定した基準的なものとして用いていく．

これらの比較評価は 2.2 節で利用例として示した Tascell [2]，HOPE [8]，JAKLD/SC [7] などの高水準言語の実装に用いるべき機構の評価に適用できる．

謝辞 本研究の一部は JSPS 科研費 JP21K19774 ならびに JP22K11984 の助成を受けたものです．

参 考 文 献

[1] Breuel, T. M.: Lexical Closure for C++, *In Proceedings of the USENIX C++ Conference*, 1988, pp. 293–304.

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

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

[4] Tazuke, M., Yasugi, M., Hiraishi, T., and Umatani, S.: Reducing Invocation Costs of L-Closures, *IPSJ Trans. Programming*, Vol. 6, No. 2(2013), pp. 13–32. (in Japanese).

[5] Yasugi, M., Emoto, K., and Hiraishi, T.: Designing Restartable Exception Handling Mechanisms for Implementing Efficient and Safe High-level Languages, *Journal of Information Processing*, Vol. 32(2024), pp. 436–450.

[6] Yasugi, M., Hiraishi, T., and Takeuchi, C.: Portable Implementations of Work Stealing, *Proceedings of International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia 2024)*, January 2024, pp. 12–22.

[7] Yasugi, M., Ikeuchi, R., Hiraishi, T., and Komiya, T.: Evaluating Portable Mechanisms for Legitimate Execution Stack Access with a Scheme Interpreter in an Extended SC Language, *Journal of Information Processing*, Vol. 27(2019), pp. 177–189.

[8] Yasugi, M., Muraoka, D., Hiraishi, T., Umatani, S., and Emoto, K.: HOPE: A Parallel Execution Model Based on Hierarchical Omission, *Proceedings of the 48th International Conference on Parallel Processing (ICPP 2019)*, August 2019, pp. 77:1–77:11.

[9] 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(2010), pp. 1–17.

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

[11] 田附正充, 八杉昌宏, 平石拓, 馬谷誠二: L-Closure の呼び出しコストの削減, 情報処理学会論文誌　プログラミング, Vol. 6, No. 2(2013), pp. 13–32.