

Scheme コンパイラにおけるレジスタ割り付けについて

前田敦司 中西正和
慶応義塾大学理工学部数理科学科

ソースコード変換は現在Lisp コンパイラの構成法として広く用いられている。とくに、レキシカルなスコープを持つLisp 方言においては中間言語としてCPS (continuation-passing style)が重要な役割を果たしている。CPSは機械語に近い手続き的な性質を持ちながらそのままソースプログラムとみなすこともできるため、コンパイラの中間言語として非常に適している。しかしながら、CPSを用いたコンパイラはレジスタ割り付けに関して特に有利な点はないものと考えられてきた。それはソースコード変換の枠内で記憶域割り付けを取り扱う有効なモデルが存在しないことに起因する。

ここでは記憶域割り付けを扱うに足る柔軟性を備えた新しい中間表現を提示する。MCPSと名付けたこの記法を用いることによりプログラムのデータフローをCPSを用いた場合よりもはるかに詳細に記述することができる。また、MCPSはコンパイラの中間言語としてCPSが持つ利点をそのまま備えている。

Register Allocation in Scheme Compiler

MAEDA Atusi and NAKANISI Masakazu
Department of Mathematics
Faculty of Science and Technology
Keio University
3-14-1, Hiyosi, Kouhoku, Yokohama 223, Japan

Source code transformation technique is widely used as development approach for modern Lisp compilers. Particularly, for lexically-scoped dialect of Lisp an intermediate language called CPS (continuation-passing style), which reflects the imperative nature of machine language but still is valid subset of source language, serves as an extremely powerful tool in constructing optimizing compilers. However, such compilers' ability is relatively restricted in the area of register allocation. It is due to the absence of effective model which express storage management issues.

Here we present a new intermediate representation flexible enough to handle this problem. In this new notation, named MCPS, data flow of programs can be expressed far more concisely than in CPS. And advantages of CPS as intermediate compiler language is kept unchanged.

1 Scheme コンパイラとCPS

近年のLispコンパイラの構成としてソースコード変換を重要な要素として用いるものが増えている [Brooks82] [Rees and Adams] [Kranz]。その基本的な考え方は、

1. 極めて少ない数の基本的な特殊形式を用意する。
2. その他の特殊形式はマクロとして実現する。ユーザが特殊形式を定義する際もマクロを用いる。

というものである。このような構成法には次のような利点がある。

1. 基本的な特殊形式の数を制限することにより処理系の核を小さく保つことができる。
2. その他の特殊形式(マクロ)は基本的な形に展開して処理するため、構文を拡張した際も処理系に手を加える必要がない。

しかし、このような構成の処理系が現実的なものとなるためにはマクロとして定義した構文についても効率の良いコードを生成できるかどうか、すなわち最適化の能力が大きな課題となる。

Steele は [Steele78] において Lisp プログラムのプリミティブである関数呼出しとラムダ式に新たな解釈を与えることによって内部形式として Lisp のソースコードを用いるコンパイラが高い最適化能力を持ち得ることを示した。ここではこのコンパイラ RABBIT において中間コードとして用いられた CPS について解説する。

なお、今回の発表においては説明のため [Steele78] で用いられたものとほぼ同じ Scheme 言語のサブセット [Sussman and Steele] [Steele and Sussman] を用いる。

1.1 関数呼出しの手続き的解釈

通常、Lisp において関数呼出し(function application)の際インタプリタの関数 apply は以下の動作を行なう。

<0. 関数に渡す実引数の値を求める。>

1. 後で実行を継続するために必要な情報をスタックなどに退避させる。
2. 関数の入口番地に飛ぶ。その際スタックやレジスタを用いて引数を渡す。

<3. 渡された引数をもとに関数の本体を実行し、呼出しもとへ復帰する。その際スタックやレジスタを介して値を返す。>

4. 1で退避させた情報を復帰する。

ただし、上で<>でくくった部分は apply 以外のルーチン (Lisp1.5 の万能関数では eval, evalis) で実行される部分である。

しかしながら、関数呼出しの解釈を次のように変えることによって関数呼出しの際に情報をスタックに退避させることは必ずしも必要でなくなる。

<0. 関数に渡す実引数すべてについて:

- 必要な情報を退避させる。
- 実引数を評価する。
- 退避させておいた情報を復帰する。

>

1. 関数の入口番地に飛ぶ。その際スタックやレジスタを用いて引数を渡す。

<2. 関数の本体を実行する。>

すなわち関数呼出しを 引数を渡すGOTO とみなすのである。この解釈に従えば、従来は関数型言語になじまないとされていた GOTO やループを用いた手続き的なプログラムをも Lisp の枠内で自然に表現することができる。

1.2 Continuation-Passing Style

RABBIT では、コンパイラの間中コードとして2種類の Scheme のサブセットを用いている。

その1つは、すべてのマクロを展開して式を定数、変数、関数呼出し、基本的な特殊形式だけを用いて表わした他はソースコードとまったく同じ構造の Scheme プログラムであり、もう1つは continuation-passing style (CPS) という形式である。

CPS とは、先の Scheme の関数呼出しの解釈 (tail-recursive なセマンティクス) に加えて、関数の値の返しかたについて新たな解釈を加え、Scheme のプログラムを変形することによって得られるプログラムのことである。

今すべての関数が陽な引数に加えてもう1つの引数 (continuation) をとるものとする。continuation は1引数の関数で、すべての関数は値を返すかわりに計算の結果を実引数として continuation を呼出すと考える。

(例外として、continuation は continuation 引数をとらない。) この continuation を陽に表わし、さらに関数呼出しの実引数として continuation かまたは変数、定数しか書かないよう変形したプログラムを CPS のプログラムと呼ぶ。約束として continuation 引数は他の引数の前に書くものとする、例えば、

```
(define (f x) x)
```

という関数はCPSに変換すると

```
(define (f' cont x)
  (cont x))
```

となり、

```
(define (g x y)
  (+ (* x y) 1))
```

は

```
(define (g' cont x y)
  (* (lambda (temp)
      (+ cont temp 1))
     x y))
```

のようになる。 $*$ のcontinuation引数として渡されているラムダ式の本体では、受け取った引数temp（乗算の結果）と1を引数として+を呼出している。 $+$ に対するcontinuationとしてg'自身のcontinuationを渡しているのは+の結果がそのままg'の結果として「返る」こと、すなわち+の呼出しがtail-recursiveなものであることを表わしている。CPSのプログラムは以下のような性質を持っている。

(1) すべての中間結果が陽に変数として現われる。

上の例を見れば明らかなように、式を評価する際の中間結果はすべてcontinuationの陽に名前をついた変数として現われる。従って、コンパイラはユーザが定義した変数と中間結果を保持する一時的な変数を区別することなく記憶域割り付けなどの処理を統一的に行なうことができる。

(2) 関数の呼出し順序が一意である。

Schemeはapplicative order (call-by-value)の言語であるが、多くのLisp処理系(left-to-right)と異なって引数の評価順序は不定である。しかしながら、CPSに変換した後のプログラムでは関数呼出しの実引数として直接関数呼出しが現われることはないので(continuationのラムダ式の中に現われることはある)applicative orderの約束から関数の呼出し順序は一意に定まる。

ここで注意すべきなのは、SchemeプログラムからCPSへの変換が一意でないことである。いくつかの可能な変換のうち一つを選ぶことで評価順序を一意に定めることになる。

(3) 評価にスタックを必要としない。

あるプログラムをCPSに変換しても、依然としてSchemeのプログラムとして実行することが可能で、結果はまったく等しいものとなる。（ただし、 $+$ や $*$ などのプリミティブな関数のCPS版を用意する必要がある。）CPSプログラムを評価するにはスタックはまったく消費しない。なぜならば、すべての実引数はcontinuation、変数、定数のいずれかであり、すべての関数呼出しが単なる「代入+GOTO」にすぎないからである。

再帰的な関数をCPSに変換した場合、再帰に必要な情報はスタックでなくcontinuation内にbound variableとして閉じこめられている。（関数closureの実現法としてスタックを用いる可能性はあるが、それははまた別の問題である。ここでは関数呼出しと、ラムダ式によるbinding機構はScheme言語のprimitiveとしてとらえている。）

すなわちcontinuationはreturn addressあるいはstack frameを抽象化し、ソースコードのレベルで陽に書き表したものとみなすことができる。

以上みてきたように、CPSプログラムはもとのプログラムの意味を反映しながら機械語に近い性質を持ち、コンパイラの間言語として理想的な性質を備えている。実際、CPSプログラムを、コンパイラの間言語としてよくみられる4つ組や基本ブロックの一般化とみることもできる。その場合1つの関数呼出しが1つの4つ組またはブロックに対応し、continuationは4つ組の結果へのポインタや次のブロックへのポインタに相当する。

2 記憶域割り付け

2.1 レジスタ割り付け

汎用計算機上のコンパイラではレジスタの使い方がオブジェクトコードの質に大きな影響を与える。式の中の中間結果をうまくレジスタに割り当てることによりコードの効率は向上する。

Lispにおいてはレジスタ割り付けは関数のinline展開と切り離して扱うことはできない。極端な例としてinline展開を全く行わず、すべての関数を実際に呼び出すような処理系ではたとえ変数をレジスタに割り付けたとしても関数呼び出しのたびにそれらをすべて退避／復帰する必要があるため効率が向上しない。

2.2 TNBIND

TNBINDとはWulfらがBliss-11のコンパイラ[Wulf]で用いた記憶域割り付けの技法である。その特徴はユーザが宣言した変数と中間結果を保持するための記憶域を区別せず、一括して扱うことにある。

式のノードすべてに値を保持する一時変数(temporary nameまたはTN)を割り当てる。その後、TNはユーザ変数とまったく区別なく扱われる。すべての変数(TN)とユーザ変数について生存期間と重要度を解析し、重要度に応じてレジスタを割り付ける。

割り付けの際さらにpreferencingという技法を用いる。代入文において、代入される変数と右辺の式のTNは同じ“preference list”に入れられる。“preference list”とは、同じ記憶域に割り付けることが望ましい変数の集まりである。もし可能ならば同じpreference listの中の変数に同一の記憶域に割り付けることによって転送命令を削除し、必要なレジスタの数を減らすことができる。

これらの技法をCPSで表現されたプログラムに適用することを考える。CPSプログラムにおいてはすべての一時変数がユーザ変数と同じ形で表現されているので、極めて自然にTNBINDの技法を適用することができる。この意味でもCPSはコンパイラの間言語として理想的な性質を持っているといえる。

2.3 記憶域割り付けに関する問題点

変数をレジスタに割り付けることによって一般にはオブジェクトコードの効率が向上することはすでに述べた。

しかしながら、つねにレジスタに変数を割り付けるのが最適とは言えない、レジスタは関数呼び出しの際には退避させる必要があるため、退避/復帰の頻度がアクセスされる回数よりも多い場合にはレジスタ割り付けによってかえって効率が低下する。そのような場合には変数をスタック内のメモリに割り付けた方がよいことになる。

例えば次のような例を考えてみる。

```
(if <pred>
  <関数呼び出しを含まないループ> ; (1)
  <関数呼び出しを含むループ> ; (2)
)
```

式(1)と式(2)の両方でアクセスされている変数はレジスタに割り付けたとしてもスタックに割り付けたとしても最適な結果は得られない。理想的には式(1)ではそのような変数はレジスタに、式(2)ではスタックに割り付けるべきである。ところがRABBITを含めて

従来のコンパイラではこのような理想的な割り付けを行うことはできなかった。その大きな理由の1つとして、RABBITで用いているようなソースコード変換としての最適化ではこのような問題をうまく形式化できなかったことが挙げられる。

次の章では上で述べたような問題を取り扱うことのできる新しいモデルを提唱する。

3 新しい記憶域割り付けの技法

前章で述べた問題を解決し、ソースコード変換の技法を拡張してより一般的な記憶域割り付けを扱うための新たなモデルをここで提出する。この章で扱うCPSのレベルにおいては1つのcontinuationの内側でかつ次に呼ばれるcontinuationの外側の部分

```
(lambda (val)
  (fn next-cont arg1 ... argn))
```

をコンパイラの扱う単位として「原子ブロック」と呼ぶことにする。

3.1 MCPS

SteeleのCPSにおいてはcontinuationは通常の関数クロージャとして表現されているため、コード生成の前に退避させる変数を改めて解析する必要がある。すなわち、クロージャの中に閉じこめられている変数は後の計算に必要なもので、退避させておく必要がある。例えば、CPS

```
(1- (lambda (val1)
      (fact (lambda (val2)
                (* cont val2 n)))))) --- (1)
```

において、contおよびnはクロージャに閉じ込められており、1-, factの呼出しに際して退避させておく必要がある。

ここで、CPSを変更して、すべての変数をcontinuationに陽に引数として与えるようにする。このためにCOMMON LISP[Steele84]の表記法を借りて次のように&auxパラメータとして表わす。

```
(lambda (v &aux (var1 val1) (var2 val2) ...)
  body)
```

上において、仮引数リストの&aux以降に現われる要素は補助変数の宣言であり、補助変数var1, ...,

varnがラムダ式の本体bodyにおいてそれぞれ初期値 val1, ..., valnに束縛されることを示す。

このように変更を加えたCPSでは上の(1)式は

```
(1- (lambda (val1 &aux (cont1 cont) (n1 n))
      (fact (lambda (val2 &aux (cont2 cont1)
                              (n2 n1))
              (* cont2 val2 n2)))) --- (1)'
```

となる。

この新しい形式のCPSを特に区別してMCPS (Modified Continuation-Passing Style)と呼ぶことにする。

3.2 MCPSの性質

MCPSは、CPSと比較した時、以下のような性質を持つ。

- (1) すべての局所変数が最も内側の continuation で束縛されている。
- (2) ほとんどの変数に対する代入は関数呼び出しに変換できる。

性質(2)について説明すると、変数v1...vnが束縛されている環境でviに対する代入を行うMCPSの式

```
(lambda (new-value &aux (vi' vi)
                        ... (vi' vi)
                        ... (vn' vn))
  (cont (setq vi' new-value)))
```

は

```
(lambda (new-value &aux (vi' vi)
                        ... (vi' vi)
                        ... (vn' vn))
  (cont' new-value))
```

ただし

```
cont = (lambda (val &aux (vi'' vi')
                    ... (vi'' vi')
                    ... (vn'' vn'))
        ...)
```

```
cont' = (lambda (val &aux (vi'' new-value)
                    ... (vi'' new-value)
                    ... (vn'' vn'))
        ...)
```

と変換できる。すなわち代入は「変数の値に対する変更」ではなくて「現在生きている変数の集合の要素を1つ置き換えること」とみなす。この場合変数の値は変更されることはない。(1.1で述べた関数の手続きの解釈に対比してこれを代入の関数的解釈と呼ぶことにする。) 性質(2)において「ほとんどの」という表現を使っている理由については後述する。

また、MCPSに対してcontinuationの本体で一度も参照されない変数を引数リストから削除する最適化を施すと、結果として得られるMCPSではcontinuationは必要な変数のみを1つ前の原子ブロックから受け取ることになる。

仮引数リストの&aux以降で宣言された変数はcontinuationの本体の計算に必要な変数である。ある原子ブロック

```
(lambda (v &aux (var1 val1) ... (varn valn))
  (fn cont arg1 ...))
```

において、この原子ブロックの本体が関数呼出しを伴うならばcontinuation contに渡される変数はスタックに退避させなければならない。また、v, var1, ..., varnのうちcontに渡されないものは通常のコンパイラでいえば、この原子ブロックでその生存期間が終わる変数に相当する(後述)。

このような表現を用いると、不必要に多くの変数が導入されるように見えるかもしれない。しかし変数の宣言は必ずしも新たな記憶域を必要としないことに注意されたい。&aux以降の変数宣言で初期値として別の変数が与えられるときには2.2で述べたpreferencingの考え方をを用いることができる。多くの場合は&aux以降の変数宣言は単なる呼び名の変更にすぎず、実際には何のコードも出力されない。

さて、MCPSでは変数に対して従来のコンパイラとはいささか異なる扱い方を必要とする。例えば、一つ一つの変数に対して生存期間の概念を考えることはMCPSではあまり意味が無い。なぜならば、すべての局所変数の生存期間は1つの原子ブロックに限られるからである。かわりに1で述べた“preference list”が従来のコンパイラでの変数に対応する。

MCPSにおける変数の扱いは従来のコンパイラにおけるそれよりもはるかに完全で一般的である。なぜならば、従来のコンパイラでは必ず1つの記憶域に割り付けられた変数が、MCPSを用いた表現では単に「1つの記憶域に割り付けることが望ましい」変数の集まりにすぎず、必要に応じて記憶場所を変更することができるからである。

また、従来のコンパイラでは2つの変数の生存期間が重なるかどうかを完全に解析することは困難であっ

た。条件文による枝分かれした制御の流れのもとでは、単純な「期間」の概念では2つの変数が重なるように見えても実際には決して同時には使われないような場合があり得るからである。[Wulf]

MCPSにおいては個々の変数はそれぞれただ1つの原子ブロックにおいてのみ現れ、また従来の変数のかわりに preference list すなわち変数の集まりを用いるので、従来のコンパイラにおける生存期間の重なりは2つの preference list に対応する原子ブロックの集まりが重なりを持つかどうかを調べることににより正確に検出できる。さらに、もし重なった場合は重なった部分だけを別扱いにする (preference list を2つに分ける) ことも可能である。

また局所変数に対する代入の消去によって従来のコンパイラで行われてきた局所変数に対する副作用の解析が(上に述べた意味での)変数の生存期間の解析に包含されてしまうことも見のがせない。

ただし、副作用解析がまったく不要になるわけではない。大域変数やデータ構造に対する副作用は依然として解析を必要とする。また、実は局所変数に対する代入を関数的に解釈することができないような場合も存在する。例えば、次のような関数を考える。

```
(define (inc state)
  (lambda ()
    (setq state (1+ state)))))
```

この関数は引数として数値を与えると引数なしの関数を返す。返された関数は、以後呼び出されるごとに最初の数値を1ずつインクリメントし、その増えた値を返す。この例は関数を、内部状態を持ついわば一種のオブジェクト[Goldberg]として用いている例である。

この関数を MCPS に変換すると次のようになる。(代入の変換は行わない。)

```
(define (inc cont state)
  (cont (lambda (c1 &aux (s1 state)) ;(1)
          (1+ (lambda (new-state) ;(2)
                 (c1 (setq s1 new-state)))))))
```

内側の(2)と印をつけたラムダ式では s1 の名前のつけかえが行われていないことに注目されたい。これは、(1)と印をつけた外側のラムダ式を評価した結果の関数が continuation (1回しか呼び出されない)でなく、データとして inc から返されているから(何回呼び出されるかわからないから)にほかならない。このような場合、変数 s1 はレジスタでもスタックでもなく、必要な間だけ値を保持できるようヒープに割り付けねばならない。

4 他の業績との比較

本研究の最適化モデルを構築する上で Wulf らの [Wulf]、Steele の [Steele78] から大きな影響を受けた。3.2 で述べたように本研究の主題である MCPS による最適化モデルはこれらの研究をさらに推し進め一般化したものといえることができる。

MCPS モデルの主な成果である記憶域の再割り付けに類似した研究として FORTRAN H のコンパイラ [Lowry and Medlock] がある。このコンパイラはループごとにレジスタの割り付けを評価しなおし、必要ならば再割り付けを行う。しかし、判断のきっかけがループのみに限られており本論文のモデルに比して一般性に欠けるうえ実現方法も比較的 ad hoc なものである。

5 研究の現状と今後の課題

3 で示した記憶域割り付けの技法の有効性を検証するため実験的なコンパイラを作成した。コンパイラは COMMON LISP により記述しており、Scheme プログラムを COMMON LISP のサブセットにコンパイルする。オブジェクトコードはさらに COMMON LISP のコンパイラによってコンパイルされ、最終的に機械語の出力を得る。結果として得られた機械語プログラムは、やはり COMMON LISP によって記述した Scheme インタプリタから呼び出すことができる。

SUN3/50 上の KCL をバックエンドの COMMON LISP システムとして用いた場合、コンパイルによる速度の向上は約 20 から 30 倍になっている。

現在のコンパイラでは変数の生存期間の長さや使用頻度を用いた複雑なコスト解析は行なっておらず、比較的単純なアルゴリズムで記憶域の再割り付け(必要に応じたスタック↔レジスタ間の移動)の実現を主眼としている。

ソースコードとして実行できる形態でありながら機械語に近い性質をあわせ持つ CPS の特性は MCPS においてもそのまま保たれているので、ソースコード変換による種々の最適化技法 [Steele78] [Brooks82] [Rees and Adams] [Kranz] を適用することが可能である。今後の課題としては、まずこれらの技法を応用したより本格的なコンパイラの作成が挙げられる。

また、MCPS によって可能になった新たな最適化技法(記憶域の再割り付け、代入の消去)の評価と従来の最適化技法との相互作用についても、今後さらに研究を進めてゆく。

References

- [Brooks82] Brooks, Rodney A., et al. *An Optimizing Compiler for Lexically Scoped Lisp.*, Proceedings of the 1982 ACM Compiler Construction Conference, ACM (Boston, Massachusetts, June 1982), pp.261-275.
- [Brooks86] Brooks, Rodney A., et al., *Design of an Optimizing, Dynamically Retargetable Compiler.*, Proceedings of the 1986 ACM Symposium on Lisp and Functional Programming, ACM (Cambridge, Massachusetts, August 1986), pp.67-85
- [Goldberg] Goldberg, Adele, and Robson, David, *Smalltalk-80: The Language and Its Implementation.*, Addison-Wesley (1983)
- [Kranz] Kranz, David, et al., *ORBIT: An Optimizing Compiler for Scheme*, Proceedings of the 1986 ACM Symposium on LISP and Functional Programming, ACM (Cambridge, Massachusetts, August 1986), pp.219-233
- [Lowry and Medlock] Lowry, E., and Medlock, C. W., *Object Code Optimization.*, CACM Vol. 12, No. 1, ACM (January 1969), pp.13-22.
- [前田] 前田敦司, *Lisp コンパイラにおける記憶域割り付けに関する研究*, 修士論文, 慶応義塾大学理工学研究科 (1988年3月)
- [Rees and Adams] Rees, Jonathan A., and Adams, Norman I. IV, *T: a Dialect of Lisp or, LAMBDA: the Ultimate Programming Tool.*, Proceedings of the 1982 ACM Symposium on LISP and Functional Programming, ACM (Pittsburgh, Pennsylvania, August 1982), pp.114-122
- [Steele and Sussman] Steele, Guy L. Jr., and Sussman, Gerald J., *The Revised Report on SCHEME: A Dialect of LISP.*, AI Memo 452, MIT AI Lab. (Cambridge, Massachusetts, January 1978)
- [Steele78] Steele, Guy L. Jr., *RABBIT: A Compiler for SCHEME (A Study in Compiler Optimization).*, AI-TR-474, MIT AI Lab. (Cambridge, Massachusetts, May 1978)
- [Steele84] Steele, Guy L. Jr., et al., *Common Lisp: the Language.*, Digital Press (1984)
- [Sussman and Steele] Sussman, Gerald J., and Steele, Guy L. Jr., *SCHEME: An Interpreter for Extended Lambda Calculus.*, AI Memo 349, MIT AI Lab. (Cambridge, Massachusetts, December 1975)
- [Wulf] Wulf, William A., et al., *The Design of an Optimizing Compiler.*, American Elsevier (New York, 1975)