記 号 処 理 68－4
（１９９３．３．１９）

CPS-conversionを用いたSchemeの処理系

三木裕史　中西正和
慶應義塾大学理工学部

Schemeのインタプリタを CPS-conversionを用いて実現する方法を提案しその有効性について報告する。CPSはソースプログラムの意味を反映しかつ機械語の特性を持ったコンパイラの中間言語としては理想的なプログラムである。またMCPSは従来のCPSをメモリ割り付けに関してもうまくモデル化できるように修正したものである。

この研究ではインタープリタレベルでもCPSが有効であると考え、コンパイラの中間言語としてのCPSの特性を踏まえて新たな側面を模索する。

The interpreter of Scheme with CPS-conversion

Hiroshi Miki and Masakazu Nakanishi
Faculty of Sience and Technology Keio University

We describe a way of the implementation of Scheme interpreter with CPS-conversion and the advantage. CPS is an ideal intermediate code for compiler since it reflects semantics of the source codeand has the characteristics like machine languages. MCPS is the modified CPS for memory allocation .

1 まえがき

Schemeは1975年にアクター理論を検証するためにSussmanとSteeleらによって開発されたLispの一方言である。Schemeはレキシカルスコープを採用しているためそうでないLispの方言と比較して、lambda計算とよく対応する。その中でも重要な特徴としてα変換とβ変換がSchemeにおいても可能であることが挙げられる。
α変換が可能なことから任意の関数の仮引数とその参照について、関数の意味を変えることなく変数名をつけかえることができる。またβ変換が可能なことから既知の関数が既知の実引数とともに呼び出されたとき、いくつかの条件を満たせば関数の本体の中の仮引数の参照を、対応する実引数で置き換えることが可能である。これらの性質がSteeleの作成したSchemeのコンパイラであるRABBIT[Steele]ではソースコードレベルの最適化に応用されている。
またSchemeのもう一つの重要な特性としてtail-recursiveなセマンティックスがある。これは関数呼び出しの解釈を次のように考えたものである。

1）関数に渡す実引数すべてについて
(i) 必要な情報を退避させる。
(ii) 実引数を評価する。
(iii) 退避させておいた情報を復帰させる。

2）関数の入り口番地に飛ぶ。その際スタックやレジスタを用いて引数を渡す。

3）関数の本体を実行する。

以上のように関数呼び出しを「引数を渡すgoto」とみなすことにより関数型言語になじまないとされてきたgotoやloopを用いた手続き的なプログラムをSchemeではLispの枠内で自然に表現することが可能となった。

次節でSteeleのRABBITコンパイラついて述べ、3節では前田のMCPSについて詳しく述べる。
これらの論文で実現されたコンパイラは中間言語としてのLispのプログラム、特にCPS(Continuation-Passiing Style)プログラムの有効性を論じている。

2 RABBITコンパイラ

SteeleはRABBITコンパイラにおいて内部形式としてのLispのプログラムが高い最適化能力を持つことを示した。RABBITは次のもので構成されている。

1）α変換
2）予備解析
3）最適化
4）CPS-conversion
5）環境とクロージャの解析
6）コード生成

1）はソースコードをコピーして新たな木構造を作りその際に変数名の交換を行ない、マクロがあればそれを展開する。

2）は1）で作られた木構造のそれぞれのノードに対して、そのノードまたはそのノード以下で参照される変数の集合を調べ、副作用を持つ式があるかないかを調べ、関数を実際に呼び出す必要があるかないか、インライン展開できるかどうかなどを調べる。それらの結果は木構造に付け加えられる。

3）最適化はβ変換を用いて以下の3つの方法で行なわれている。

i）複合式で関数部分がλ式でかつそのλ式が引数を持たない場合、以下の置き換えが可能である。

((lambda () body)) → body

ii）複合式の関数部分がλ式で、その本体で参照されない仮引数がある場合、その仮引数と対応する実引数は取り除くことができる。但し対応する実引数は副作用を持たないものとする。

((lambdaa (x1 x2 x3) body) a1 a2 a3)
→((lambda (x1 x2) body) a1 a2)
但しx3はbodyでは現われずかつa3は副作用を持たない

iii）複合式で関数部分がλ式である場合、その本体部分に現われる仮引数のいくつかは実引数で置き換えることが可能である。但し、仮引数が本体でただ一度だけ参照されていて、かつ対応する実引数が変数または定数である時

```
((lambda (v1 v2 ... vi ... vn) body)
      a1 a2 ... ai ... an)
→((lambda (v1 v2 ... vi-1 vi+1 ... vn) body')
    a1 a2 ... ai-1 ... ai+1 ... an)
```

ここでbody'はbodyの中のviの参照をaiに置き換えたものである。

4）CPS-conversionを行ない新しい木構造を作る。CPSはSchemeのtail-recursiveなセマンティックスに加え、関数の値の返し方についても新たな解釈を加えたSchemeのプログラムのことである。いますべての関数に対してもう一つの引数continuationを加える。ここでcontinuationは引数一つの関数であり、continuationの引数としてはcontinuationは取らないものとする。そしてすべての関数は値を返す代わりに、計算の結果を実引数としてcontinuationを呼び出すことにする。さらに関数の実引数としてはcontinuationかまたは変数、定数しか書かないようにSchemeのプログラムを変換する。こうして得られた別のSchemeのプログラムをContinuation-Passing Styleと呼ぶ。以下continuationは他の引数よりも前に書くことにする。

例えば以下のプログラムをCPSに変換すると

```
a = (define (f x y)   (+ (* x x) y))
b = (define (f cont x y)
       (* (lambda (gensym)
            (+ cont gensym y)) x  x)
```

となる。

* のcontinuationであるλ式の本体では受け取った引数gensymとyを引数として+を呼び出し、+に対するcontinuationとしてfのcontinuationを渡している。このことは+の結果がfの結果として返ることすなわち＋の呼び出しがtail-recursiveであることを意味している。このCPSのプログラムは以下の性質からコンパイラの中間言語として理想的であるといえる。

1）すべての計算の途中結果が陽に変数とし現われる。このことにより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ための一時的な変数を、区別することなく統一的に処理を行なうことができる。

2）関数の呼び出し順序が一意である。Schemeは引数の評価順序が不定であるが、CPSに変換した後では関数の実引数として関数呼び出しが直接現われることはないので関数の呼び出し順序は一意に定まる。しかしSchemeの通常プログラムからCPSへの変換は一意でないのでいくつか変換可能な候補のうちひとつを選ぶことで評価順序を一意に定めることになる。

3）評価のためににスタックを必要としない。なぜならすべての実引数はcontinuation、変数、定数のいずれかであり、すべての関数呼び出しが引数を渡すgotoだからである。再帰的な関数の場合は再帰に必要な情報はcontinuationの束縛変数として保持されておりスタックに保持されるのではない。

このようにCPSプログラムはもとのプログラムの意味を反映し、かつ機械語に近い性質を持っている。したがって最終的なオブジェクトコードへの変換は簡単に行なうことができる。実際のコンパイルの作業はソースプログラムからCPSプログラムへの変換であるともいえる。

5）では　4）で作られた木構造に対して
2）の操作を再び行ない、新しい情報を付け加える。

6）5）の木構造に対してオブジェクトコードを生成する。RABBITの出力するコードはprog,go,setq など極めて小さく、かつ手続き的な要素にだけに限定されたMacLispのサブセットであり、実際の機械語に変換するにはMacLispのコンパイラを用いている。

以上のようにRABBITではソースレベルプログラム変換を用いて種々最適化実現しているが記憶域割り付けに関しては考慮されていなかった。

汎用計算機上のコンパイラでは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って、オブジェクトコードの質は向上することが知られている。Lispにおいてはレジスタ割り当ては関数のインライン展開においてその効果発揮する。なぜならすべての関数を実際に呼び出すような処理系では変数をレジスタに割り当てても関数呼び出しの度にそれらを退避、復帰させる必要があるからである。RABBITで用いられている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することは出来なかったのである。

3 MCPS

[前田87I]]はRABBITで用いられたような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構築した。SteeleのCPSではcontinuationは通常の関数クロージャとして表現されていたためにコード生成の前に退避する変数を改めて解析する必要があった。クロージャの中で閉じ込められている変数は後の計算に必要で退避すべきものである。

例えば次のようなCPSプログラムを考える。

```
(1- (lambda (v1)
      (fact (lambda (v2)
          (* cont v2 n)))))
```

contおよびnはクロージャに閉じ込められており1-,factの呼び出し時に退避しておく必要がある。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。これをCommon-Lispの表記を借りて次のように表す。

```
(1- (lambda (v1 &aux (cont1 cont) (n1 n))
    (fact (lambda (v2 &aux (cont2 cont1) (n2 n1))
        (* cont2 v2 n2)))))
```

ここで仮引数リストの&aux以降に現われる要素は補助変数の宣言で補助変数cont2 ,n1がλ式の本体内部においてそれぞれcont, nに束縛されることを意味する。このように変更を加えたCPSがMCPSである。

MCPSはCPSと比較して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。このことよりCPSに変換後のクロージャの解析が不要になる。
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。

例えば以下のMCPSの式を考える。

```
(lambda (n &aux (var1 val1) (var2 val2) ...
                (vari vali) ... (varn valn))
  (cont (set! vari  n)))                    - (1)
```

但し

```
cont =
 (lambda (val &aux (var'1 var1)(var'2 var2) ...
                (var'i vari) ... (var'n varn))
          ...)
```

である。
すると(1)のプログラムは

```
(lambda (n &aux  (var1 val1) ...
                (vari vali) ... (varn valn))
      (cont' n))
```

但し

```
cont' =
 (lambda (val &aux (var'1 var1)(var'2 var2) ...
                (var'i n) ... (var'n varn))
          ...)
```

これは代入が変数の値の更新ではなく、参照される可能性のある変数の集合の要素を一つ置き換えることとみなされる。（代入の関数的解釈）またMCPSプログラムに対してRABBITで行なわれた最適化を行なうとcontinuationの本体で一度も参照されない変数が除去されるので結果として得るMCPSではcontinuationが受け取る変数は必要なものだけとなる。

変数の宣言により新たな記憶域が必要であるように思われるが、&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多くの場合、それらは変数の呼び名の付け替えにすぎず実際にはそれらに対して何のコードも出力されない。

MCPSにおいてはすべての局所変数の生存期間は一つの原始ブロック（一つのcontinuationの内側

でかつ次に呼ばれるcontinuationの外側の部分)に限られるため一つ一つ変数に対して生存期間の概念を考える必要はない。代わりにpreferencingという技法を用いる。preferencingは代入文において代入される変数と右辺の式の変数を同じ"prefer-ence list"に入れる。"preference list"は同じ記憶域に割り付けられることが望ましい変数の集まりである。同じpreference listの中の変数は同記憶域に割り付けられ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。MCPSにおいては従来のコンパイラでの変数に対応するものとしてpreference listを用いる。このことにより従来のコンパイラでは一つの記憶域に割り付けられていた変数は、MCPSでは同じ記憶域に割り付けられ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。

このMCPSを用いて前田が作ったコンパイラは変数の生存期間や使用頻度を用いた最適化を行なっていないにもかかわらずレジスタをうまく利用した良質なコードが生成されている。

4 MCPSバイトコードインタプリタ

MCPSでは、コンパイラでの最適化の負担を軽減させるものであるが結果としてその記憶割付け等はMCPS自身が担うことになる。MCPSバイトコードインタプリタを構築す際にコンパイラの利点を消滅させないような配慮をしなければならない。
[前田87I]では、実際上MCPSインタプリタはCommonLispにより記述されていることになり、CommonLispのコンパイラの効率に依存する。今回これをバイトコートインタプリタで実現することを考えた。このことにより移植性の向上や、MCPSに適したバイトコードを設計することによる効率の向上が期待される。

5 あとがき

今回の報告では変数寿命に応じたレジスタの割付け関しては詳しく述べなかったがMCPSを利用してこれらの最適化も今後の研究課題として考えられる。

参考文献

[Sussman and Steele]
Gerald Jay Sussman and Guy Lewis Steele Jr.
"SCHEME An Interpreter for extended Lambda calculus" AI Memo No 349 ,MIT AI Lab (Cambridge,Massachusetts,December . 1975)

[Steele]
Guy Lewis Steele Jr.
"RABIT: A Compiler for Scheme (A Study in Compiler Optimization)",AI-TR-474,MIT AI Lab. (Cambridge,Massachusetts,May 1978)

[前田87I]
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 (慶應義塾大学　1987)
[前田87II]
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 (情報処理学会　記号処理研究会　４８.5)
[Bondorf]
Anders Bondorf
"Improviing Binding Time without Explicit CPS-conversion" Proceedings of the 1992 ACM Symposium on Lisp and Functional Programming , p1-p10 June 1992

[Sarby and Felleisen]
Amr Saby,Matthias Felleisen
"Reasoning about Program in Continuation-Passing Style",Proceedings of the 1992 ACM Symposium on Lisp and Functional Programming , p288-p298, June 1992

[Danby and Lawall]
Olivier Danvy ,Julia L. Lawall
"Back to Direct Style II:First-Class Continuations" Proceedings of the 1992 ACM Symposium on Lisp and Functional Programming , p299-p310June, 1992

[revised4]
Wililam Clinger,Jonathan Rees

"Revised4 Report on the Alogorithmic Language Scheme",November 1991