MinCaml コンパイラにおける shift/reset の実装

増子萌　(指導教官:浅井健一)

1　はじめに

継続とは，計算のある時点における「残りの計算」を表す概念である。プログラミング言語に継続を扱える機能を導入すると，ユーザによるプログラムの実行順序の制御が可能になる。継続が有用に使われる例には，例外処理の制御，深い分岐からの脱出，非決定性プログラミングなどがある。

2　shift/reset とは

shift/reset とは Danvy と Filinski らによって提案された限定継続のための命令である [2] 。ここで，shift は現在の継続を切り取る命令，reset は shift で切り取られる継続の範囲を限定する命令である。shift/reset を使うと，プログラムを CPS (継続渡し形式，continuation-passing style) で書くことなく，継続が扱えるようになる。

3　shift/reset の直接実装

shift/reset を含むプログラムを実行するには，CPS 形式のインタプリタを使うか，call/cc を利用して挙動を模倣すればよい。しかし，どちらの方法も実行速度が遅い。shift/reset 自体の効率についての議論は，直接実装した上で行う必要がある。本章では shift/reset を直接実装する方法を示す。

本研究では，OCaml のサブセット MinCaml のコンパイラ，MinCaml コンパイラ [4] を shift/reset とリストで拡張した言語をコンパイルして，PowerPC の機械語のコードを出力するコンパイラを作成した。実装にあたり，型推論には Asai らの shift/reset を含む多相の型システム [1] を用いた。

3.1　実装の概要

継続の概念に基づくと，プログラム全体は継続のフレームが連なったスタックと捉えられる。すると，reset はスタックに reset の印を入れる命令，shift は reset の印までのフレームを切り取って関数にし，引数の関数に渡す命令と考えられる (図 1) 。

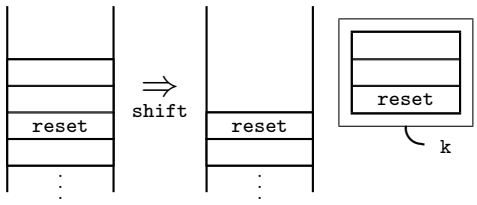

図 1: shift を呼び出したときのスタックの変化

この考えに基づく実装を，以下に示す。reset の印にはレジスタをひとつポインタ (reset pointer，以下 rp) として用いた。

3.2　reset の実装

reset は外部関数として呼び出し，

① 戻り番地と rp をスタックに保存

② reset の引数の関数 r を呼び出す

③ 戻り番地と rp を復活

④ 戻り番地へジャンプ

の順で処理を行うよう実装した。

reset は引数の式を空の継続 (恒等関数) で実行する。reset を呼び出した時点での戻り番地の保存によって，reset r の実行後に実行すべき計算の番地を保存する。さらに rp も保存することで，この戻り番地と reset までのスタックの状態が r 内で実行される shift で切り取られないようにしている。

3.3　shift の実装

shift も reset と同様，外部関数として呼び出し，

① 現在の rp の 1 つ上までのフレームをヒープに移動

② shift を呼び出した時点での戻り番地を使って関数 k のクロージャを作る

③ k を第 1 引数として shift の引数の関数 s を呼び出す

④ 戻り番地と rp を復活

⑤ 戻り番地へジャンプ

の順で処理を行うよう実装した。引数の関数 s の実行後は最も近い reset の残りの計算に移るので，戻り番地と rp の復活をする。また rp をスタックに残すのは，s も空の継続で実行するからである。

k を呼び出すときに実行するのは shift で切り取った継続の計算なので，k の呼び出しには shift でヒープに移動したフレームと，shift を呼び出した時点での戻り番地が必要になる。これらの情報から k のクロージャを作り，実際に k を呼び出すときには，

① k が呼び出された時点での戻り番地と rp をスタックに保存

② k に対応するフレームをスタックにコピー

③ shift を呼び出した時点での戻り番地へジャンプ

の順で処理を行うよう実装した。

3.4 機械語による実装

以上の考察を行うと shift/reset の直接実装が可能となる。実際に必要となるのは shift，reset，および k を実現する関数のみで，これらは PowerPC の機械語のコードで shift が 30 行程度，reset が 20 行程度，k が 20 行程度である。

4 lazy な実装

上に示した実装では，shift のたびにフレームの移動が必要になる。しかし，reset r という式を実行するとき，スタックには r の関数フレームが作られる。shift (fun k -> 1 + k 3) のように shift の中でのみ k が用いられる場合は，対応するフレームをヒープには確保せず，スタックに確保しても問題ない。一方，shift (fun k -> k) のように shift を抜けた後にも k が用いられる場合は，スタックのフレームが消された後にも対応出来るよう，ヒープにもフレームを確保する必要がある。このような場合を，k が **エスケープする** という。また，フレームをスタックに残しておくと，shift (fun k -> k 3) のように末尾位置での k の呼び出しがある場合，残しておいたフレームが利用出来る。shift の引数の関数 s で k がどのように用いられているかによって適切な処理を施すと，フレームのコピーを lazy に行う実装も実現出来る。

5 比較

本章では，3 章の実装と 4 章の実装の比較を行う。

- prefix　リストの prefix のリストを返す関数。

	時間 (秒)	メモリ (byte)
CPS	2.489	11158096
shift/reset	2.503	11378360
lazy	2.558	11290360

shift/reset と CPS 形式の実行時間に大きな差はない。CPS 変換を施すことなく同程度の速度が得られている。

- queen　N-queen 問題の解を返す関数。

	時間 (秒)	メモリ (byte)
shift/reset	1.357	80152072
lazy	0.713	53872712

k がエスケープする場合。

	時間 (秒)	メモリ (byte)
shift/reset	1.423	86721912
lazy	1.072	204979032

コピーするフレームが大きいので，lazy な実装による実行速度の向上が見られる。k がエスケープする場合は，lazy な実装の方がヒープを多く使う。

- times　リストの要素の積を返す関数。

	時間 (秒)	メモリ (byte)
通常の再帰	0.217	16024
CPS	0.293	80056032
shift/reset	0.427	80176024
lazy	0.173	176024
再帰 (OCaml)	0.501	–
例外 (OCaml)	0.251	–

shift/reset は shift (fun k -> 0) と，k を使わないにも関わらず対応するフレームをヒープに移動するので遅い。lazy な実装ではスタックのフレームを破棄するだけなので，他に比べて速くなっている。

6 関連研究

Gasbichler らは control と shift/reset の直接実装の方法を示し，実際に Scheme48 上で実装した [3] 。それにより，call/cc を利用した間接的な実装に起因するいくつかのオーバーヘッドが軽減出来ること，および実行効率の向上がもたらされることを示した。

Ugawa らはスタックベース処理系における一級継続の実装法として，遅延スタックコピー法を提案した [5] 。lazy な実装のアイディアはこの遅延スタックコピー法による。

7 まとめと今後の課題

本稿では，shift/reset を直接実装する方法，さらにフレームのコピーを lazy に行う実装方法を示した。実際に実装を行ったところ，どちらの実装でも shift/reset を含むいくつかのプログラムが実行出来ている。今後の課題としては，より多くのプログラムを使ってのベンチマーク，直接実装法の正当性の証明などが挙げられる。

参考文献

[1] K. Asai, and Y. Kameyama. “Polymorphic Delimited Continuations”, *APLAS 2007*, pp. 239–254 (November 2007).

[2] O. Danvy, and A. Filinski, “Abstracting Control”, *LFP 1990*, pp. 151–160 (June 1990).

[3] M. Gasbichler, and M. Sperber, “Final Shift for Call/cc: Direct Implementation of Shift and Reset”, *ICFP 2002*, pp. 271–281 (October 2002).

[4] E. Sumii, “MinCaml: A Simple and Efficient Compiler for a Minimal Functional Language”. *FDPE 2005*, pp. 27–38 (September 2005).

[5] T. Ugawa, N. Minagawa, T. Komiya, M. Yasugi, and T. Yuasa. “Lazy Stack Copying and Stack Copy Sharing for the Efficient Implementation of Continuations”. *APLAS 2003*, pp. 410–426 (October 2003).