shift/reset の直接実装法と Caml Light における実装

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 によって提案された限定継続の命令である [3]。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 を含むプログラムは，CPS 変換を施せば実行出来る。また，直接実装もいくつか示されている [4, 7]。しかし，CPS 変換を使う実装と直接実装のどちらが効率的か，という議論はまだ十分でない。

そこで，本研究ではある程度の記述力があり，多くの処理系の上で実行出来るシステムである Caml Light に shift/reset を導入した。実装にあたって，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　ZINC

Caml Light のプログラムは，字句解析，構文解析，パターンマッチコンパイル，コード生成，リンキングを経て，最終的な実行ファイルに変換される (図 1)。コード生成によって生成されるコード列は，ZINC 抽象機械 [6] の命令列に対応している。

ソースプログラム
⇓　**字句解析，構文解析**
抽象構文木
⇓　**パターンマッチコンパイル**
de Bruijn notation の拡張ラムダ計算
⇓　**コード生成**
コード列
⇓　**リンク**
実行ファイル

図 1: Caml Light の実行の流れ

ZINC 抽象機械は，引数を保存する引数スタック，戻り番地を表すフレームからなるリターンスタック，一時的な値を置く環境，実行結果が置かれるアキュムレータ，クロージャなどを置くヒープを利用して命令列を実行する。引数スタックに引数だけでなく関数適用の境界を表す印を積むことで，クロージャ生成を抑え，カリー化された関数も高速に実行している。

shift/reset を導入する上でも，ZINC の最適化を妨げない格好で実装することが望ましい。

3.2　実装方針

実装方針には MinCaml における実装 [7] の，

- reset のとき，スタックに reset の印を入れる
- shift のとき，直近の reset の印までのスタック領域をメモリに移動する
- reset の印を入れるときにはインバリアントを満たすようにする

を適用した。以下，shift や reset の実装を見る。

reset は，

1. 引数スタックに関数適用の境界を表す印 (MARK)，リターンスタックにリターンフレーム (戻り番地を表すフレーム) を格納
2. 引数スタックとリターンスタックに reset の印を保存
3. アキュムレータ上の reset の本体に相当するクロージャを展開して実行

という処理を行うように実装した。ZINC では引数スタックとリターンスタックの 2 本のスタックを使用するため，その両方に reset の印を入れる。引数スタックの reset の印の下には必ず MARK が積まれていて，リターンスタックの reset の印の下には必ずリターンフレームが積まれている，ということが Caml Light におけるインバリアントである。

その上で，shift は以下のように実装した。

1. 保存したい大きさのフレームをヒープに確保
2. 引数スタックとリターンスタックの，最も近い reset の印の 1 つ上までのフレームを 1 で確保したヒープに移動
3. 移動したスタックフレームの大きさ，プログラムカウンタ，環境などを 1 で確保したヒープに格納
4. 切り取った継続を第 1 引数として本体を実行するため，リターンスタックにクロージャ (1 で確保したヒープへのポインタ) を保存

5. アキュムレータ上の shift の本体に相当するクロージャを展開して実行

切り取った継続の呼び出し自体には特別な変換は施しておらず，一般のクロージャの関数適用と同じ流れで実行される。このクロージャは呼び出されると，以下の処理をするプログラムカウンタを読み込む。

1. 引数スタックとリターンスタックに reset の印を保存
2. アキュムレータに，リターンスタックの先頭 (キャッシュ部分) に保存されていた値をセット
3. 継続のクロージャから，引数スタックとリターンスタックにフレームをコピーし戻す
4. 保存されていた shift の継続にジャンプする

切り取った継続を実行するときには，第 1 引数が現在の結果と思って shift の継続の実行に行くので，2 でアキュムレータに第 1 引数をセットしている。

そして，shift や reset の本体の実行が終了したときには Return 命令と同じ処理を入れる。

1. 引数スタックとリターンスタックを，reset の印まで下げる
2. Return 命令と同じ処理を行う

Return 命令は，関数適用した結果が関数になったとき，すでにその引数が渡されている場合は呼び出し先に戻らず，直接結果の関数を展開する ZINC 特有の命令である。reset などの実行結果が関数になる場合も，この最適化を施したいので入れておく。

4 プログラム変換

上に示した実装法の正当性を保証するため，プログラム変換の手法を利用して ZINC を導出する。本章では，証明に向けた試みを示す。すでに戻り番地の復活，退避も含めた機械語の導出はなされている [2] ため，ZINC 固有の挙動を導入するように変換を施す。

今回は多引数の関数適用を許す，de Bruijn Notation のラムダ計算に shift/reset を加えた言語を対象とし，それに対応するインタプリタへ変換を施す。

4.1 スタック導入

非関数化，線形リスト化，フレーム分割，非線形化，再関数化によって，スタックを導入する。ZINC のスタックは引数スタックとリターンスタックの 2 本あるため，再関数化の後にさらにスタックを分割する。

4.2 Return 導入

ZINC 特有の命令である Return 命令に対応する挙動 (引数が十分に与えられているときには，結果の関数に直接ジャンプする) を導入する。

この後は，Grab に対応する挙動の導入，戻り番地の導入などを経て，コンビネータで表現，非関数化，線形リスト化，コード平坦化の変換により，ZINC に対応するコンパイラと仮想機械を導出する見通しである。

5 関連研究

Gasbichler らは control と shift/reset の直接実装方法を示し，実際に Scheme 48 上で実装した [4]。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，および実行効率が向上することを示した。

Rompf らは選択的な CPS 変換を利用して，Scala に shift/reset を実装した [8]。大規模で実際的なプログラミング言語における shift/reset の実装を実現しているが，直接実装との比較は今後の課題である。

Kiselyov はマルチプロンプトの限定継続を直接実装するための一般的なアプローチを示し，実際に OCaml と Scheme で実装した [5]。既存の実装に手を加えずに出来る点で，Kiselyov の手法は優れている。

6 まとめと今後の課題

本稿では，shift/reset を直接実装する方法とその正当性の証明に向けた試みを示した。いくつか興味深いプログラムも動いている。今後の課題は証明を完成させること，システム全体を再コンパイルした上でより多くのプログラムを実行してみること，CPS 変換や選択的な CPS 変換との効率を比較することである。

参考文献

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

[2] K. Asai and A. Kitani. Functional derivation of virtual machine for delimited continuations. In *PPDP 2010*, (July 2010). To appear.

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

[4] M. Gasbichler, and M. Sperber. “Final shift for call/cc: direct implementation of shift and reset”, *ICFP 2002*, pp. 271–281 (October 2002).

[5] Oleg Kiselyov. Delimited control in OCaml, abstractly and concretely. system description. In *FLOPS 2010*, pp. 304–320 (April 2010).

[6] X. Leroy. “The Zinc experiment: An economical implementation of the ML language”, Technical report, INRIA, February 1990.

[7] M. Masuko, and K. Asai. “Direct implementation of shift and reset in the MinCaml compiler”, *ML 2009*, pp. 49–60 (October 2009).

[8] T. Rompf, I. Maier, and M. Odersky. “Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-Transform” *ICFP2009*, pp. 317–328 (September 2009).