shift/reset による Caml Light の拡張に向けて

増子萌　　浅井健一
お茶の水女子大学
moe@pllab.is.ocha.ac.jp
asai@is.ocha.ac.jp

概 要

限定継続の命令 shift/reset を用いると，ユーザによるプログラムの実行順序の制御が可能となり，さまざまな応用が考えられはじめている。それに伴い，shift/reset の効率的な実装が求められるが，型付き言語における shift/reset の直接実装は，まだ MinCaml コンパイラにおけるものしかない。しかし，複雑で興味深い応用プログラムを記述するには MinCaml では不十分になってくる。そこで本研究では，強力なモジュールシステムなどはないが，ある程度の表現力が備わっており，多くの異なる処理系の上で実行出来る Caml Light に shift/reset を導入し，その直接実装を試みる。

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。プログラムで継続を扱えると，ユーザによるプログラムの実行順序の制御が可能となる。その応用例には，例外処理や大域的ジャンプ，非決定性プログラミング [5]，部分評価における let-insertion [19]，コルーチン，Web プログラミングなどがある。

継続はプログラム全体を継続渡し形式 (Continuation Passing Style，CPS) で書けば明示的に扱えるが，CPS 変換は大域的な処理である。一度継続渡し形式でない形 (直接形式) でプログラムを書いてから，プログラムの一部で継続を扱う形にしようとすると変更が多くなってしまう。そこで，直接形式のプログラムで継続を扱うために，call/cc [17] や control/prompt [7]，shift/reset [5]，set/cupto [10] などのオペレータが考えられて来た。この中でも shift/reset は継続を扱う範囲を指定出来ること，切り取る継続が静的に決まること，型システム [3, 4] が存在していることから扱いやすく，さまざまな応用が考えられはじめている [1, 2, 11]。

そこで，shift/reset の効率的な実装が求められるが，型付き言語における shift/reset の直接実装は，まだ MinCaml コンパイラ [18, 20] におけるもの [15] しかない。MinCaml でも色々なプログラムは記述出来るものの，複雑な応用プログラムを記述するには記述力が不十分になってくる。例えば MinCaml ではユーザが型定義をすることは出来ないが，shift/reset を使った部分評価 [1] をしようとすると，以下のような抽象構文木をプログラムに記述出来る必要がある。

```
type t = Var of string
       | Lam of string * t
       | App of t * t
       | Shift of string * t
       | Reset of t
       | Let of string * t * t
```

MinCaml を拡張すれば上のような構文木を扱えるようにするのも不可能ではないが，記述力に制限があることに変わりはない。また，拡張を重ねるのは労力が必要なだけでなく，バグを取り込む可能性も高くなり，あまり現実的でない。

そこで本研究では，ある程度の記述力が備わっていて，多くの処理系の上で実行出来る Caml Light に `shift/reset` を導入し，その直接実装を試みる。具体的には，MinCaml での実装における，

- `reset` のとき，印を入れる
- `shift` のとき，直近の印までの部分を移動する
- `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

という方針を適用する。より実用的なプログラミング言語での実装によって，今後の `shift/reset` の使用が推進されることが期待される。

本論文の構成は以下の通りである。2 章では `shift/reset`，3 章では Caml Light を簡単に説明する。4 章では実装の概要を述べ，5 章に実行例を挙げる。6 章では関連研究，7 章でまとめと今後の課題を述べる。

2 `shift/reset` とは

`shift/reset` は Danvy と Filinski によって提案された限定継続の命令である [5, 6]。直観的には，`shift` は現在の継続を切り取る命令，`reset` は `shift` が切り取る継続の範囲を限定する命令である。本研究では，`shift (fun <var> -> <exp>)` という式のとき `shift` の引数の関数に現在の継続を関数として渡して実行し，`reset (fun () -> <exp>)` という式のとき `reset` を空の継続で実行するものとする。空の継続で実行することは，その `reset` に囲まれた `shift` によって切り取られる継続の範囲を，引数の関数内に限定することを意味する。

例えば，`1 + reset (fun () -> 2 * shift (fun k -> 3 + k 4))` という式の場合，`shift` で切り取るのは『2 を掛ける』という継続 (`2 * □`) なので，

```
        1 + reset (fun () -> 2 * shift (fun k -> 3 + k 4))
    ⇒ 1 + (3 + 2 * 4) ⇒ 12
```

となる。また，`shift` で切り取った継続は通常の関数のように複数回使うことも出来て，
`1 + reset (fun () -> 2 * shift (fun k -> k 3 + k 4))` は `1 + (2 * 3 + 2 * 4) = 15` になる。

3 Caml Light

Caml Light は軽量で移植性の高い Caml 言語である [14]。現在では実装に変更が加えられることはなく，バージョンが固定されている。OCaml のような強力なモジュールシステムやオブジェクト指向はないが，そのぶん型推論などはそれほど複雑ではなく，`shift/reset` の導入を試みるのに良い土台といえる。

3.1 構成

Caml Light 全体の構成は以下のようになっている。

runtime バイトコードインタプリタとランタイムシステム。C で書かれている。

lib, compiler 標準ライブラリ，コンパイラ。Caml Light で書かれている。

linker, librar リンカ，ライブラリアン (複数のバイトコードオブジェクトファイルを 1 つのファイルにまとめるもの)。Caml Light で書かれている。

toplevel トップレベルのインタラクティブなシステム。Caml Light で書かれている。

lex, yacc lexer と parser のジェネレータ。lex は Caml Light，yacc は C で書かれている。

tools さまざまなユーティリティ。シェルスクリプト，Perl，C，Caml Light で書かれている。

OCaml と同様，コンパイラによる実行ファイル生成だけでなく，対話的にプログラムを入力して実行することも出来る。ユーザの入力したソースプログラムは以下のような変換を経て，最終的な実行ファイルにコンパイルされる。トップレベルの式も，実行ファイル生成を除いて同じ流れで実行される。

ソースプログラム
⇓ **字句解析，構文解析**
抽象構文木
⇓ **型推論，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

型推論には多相の型推論が用いられており，型情報は拡張ラムダ計算の部分まで保持される。また，最適化は一般的な比較に対する最適化と，末尾呼び出し最適化がなされている。コード生成によって生成されるコード列は，ZINC 抽象機械の命令列に対応している。

3.2 ZINC

ZINC 抽象機械 [13] は引数スタック，リターンスタック，環境，アキュムレータ，ヒープを用い，プログラムカウンタを使って命令列を実行する。それぞれ，以下のように用いられる。

引数スタック 関数適用の引数となる値と，その関数適用の引数部分はスタックのどこまでなのか，という境界を示す印が積まれる。

リターンスタック 戻り番地を表すリターンフレーム，および `let` などで一時的に導入された変数の値が主に格納される。後者はキャッシュと呼ばれる。リターンフレームにはプログラムカウンタ (戻り番地) に加え，環境とキャッシュサイズ (一時的に導入された変数の数) が含まれる。例外の実装に使用されるトラップフレームも積まれる。

環境 ヒープに置かれた値。クロージャを生成するとき，リターンスタックに格納されている値から作られる。

アキュムレータ 関数呼び出しの際，その関数のクロージャがセットされる。プリミティブの関数の場合は第 1 引数がセットされる。また，関数適用の実行結果はアキュムレータに保存される。

ヒープ クロージャや環境，組，レコード，浮動小数点数などが保存される。

引数スタックに印を積むことで，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。また，クロージャが頻繁には生成されないことから，関数の引数を含む一時的な値はリターンスタックに保存して，ヒープへの格納も抑えている。

3.3 ZINC での実行

この節では，ZINC 抽象機械がどのように関数適用などを実行しているのか説明する。ZINC 抽象機械は，カリー化された関数を高速に実行するため，2 つの最適化を用意している。1 つは複数の引数で (カリー化された) 関数を呼び出す際，引数 1 つごとにクロージャを作らなくて済むようにする最適化，もう 1 つは関数を返す際，その返された関数の引数がすでに渡されている場合には，返される関数を直接呼び出す最適化である。

```
(fun x -> fun y -> x + y) 7 8 ;;
```

というプログラムは，以下の ZINC のコード列にコンパイルされる。

```
    Pushmark        // 引数スタックに印を入れる
    Quote 8         // 8 をアキュムレータ (accu) にセット
    Push            // 引数スタックに accu の値を積む
    Quote 7         // 7 を accu にセット
    Push            // 引数スタックに accu の値を積む
    Closure 1       // Label 1 のクロージャを accu にセット
    Apply           // 関数適用
  function:         // (関数の定義)
    Label 1:
      Label 2:
        Grab        // 引数スタックから値を取ってくるか，クロージャを作る
      Label 3:
        Access 0    // 環境の 0 番目の値を accu にセット
        Push        // 引数スタックに accu の値を積む
        Access 1    // 環境の 1 番目の値を accu にセット
        Addint      // 足し算
        Return      // 呼び出し先に戻る
```

Pushmark 命令で引数スタックに印を積んだあとに 8，7 を引数スタックに積み，Label 1 の関数を呼び出す。末尾呼び出しでない場合の関数適用は Apply 命令で実行される。バイトコードインタプリタ上での Apply 命令の挙動は，

1. リターンスタックにプログラムカウンタ (= 戻り番地) を含むリターンフレームを格納
2. 引数スタックの先頭の値 (= 第 1 引数) をリターンスタックの現在のキャッシュエリアに移動
3. アキュムレータ上にある関数のクロージャから，プログラムカウンタと環境を読み込むことで，クロージャへジャンプ

と定義されている。

上の (fun x -> fun y -> x + y) 7 8 のようなプログラムを実行するときは，与えられた引数はすべてあらかじめ引数スタックに積まれる。その上で fun y -> x + y を「すでに y が与えられていたら，それをとってきて x + y を実行する。まだ y が与えられていなければ (仕方がないので) クロージャを作って返す」というコードにコンパイルする。引数スタックの先頭を見て，残りの引数が積まれているかどうかで処理を変えているのが以下の Grab 命令である。

- 引数スタックの先頭が印の場合
  1. 環境とプログラムカウンタからクロージャを生成し，アキュムレータにセット
  2. 引数スタックの印を破棄
  3. リターンフレームを復活することで呼び出し元へジャンプ (クロージャを生成して返す)

- 引数スタックの先頭が印でない場合
  1. 引数スタックの先頭の値をリターンスタックに格納
  2. キャッシュサイズを 1 増やす
  3. プログラムカウンタを 1 つ進める (引数を消費して次に進む)

印を使わないと，7 を適用したところでクロージャが生成され，それを展開して 8 を適用することになる。印を使うと，7 を適用したあとのスタックにまだ引数 (8) が積まれていることが分かるので，クロージャを作ることなく次の適用を実行出来る。

もう 1 つの最適化は，関数を返す場合の最適化である。例えば (`fun x -> let y = ... in fun z -> y`) 7 8 のようなプログラムを考えた場合，引数 7 を受け取ったら，返ってくるのは `fun z -> y` というクロージャである。これを素直にコンパイルすると，このクロージャを返し，その上で引数 8 で呼び出すことになるが，すでに引数スタックに残りの引数 (8) も積んであれば，`fun z -> y` を返す前に引数スタックを調べることで，直接 `fun z -> y` を呼び出すことができる。この処理を行っているのが，関数本体の末尾に入った `Return` 命令である。`Return` 命令は引数スタックの先頭に印があるかで，以下のように処理を分けている。

- 引数スタックの先頭が印の場合
  1. リターンスタックをキャッシュサイズ分，破棄
  2. 引数スタックの印を破棄
  3. リターンフレームを復活することで，呼び出し元へジャンプ (普通に return する)
- 引数スタックの先頭が印でない場合
  1. リターンスタックをキャッシュサイズ分，破棄
  2. 引数スタックの先頭の値 (= 第 1 引数) をリターンスタックに格納
  3. アキュムレータ上のクロージャに直接ジャンプ (直接，関数を呼び出す)

引数スタックの先頭が印の場合は，適用する値が残っていないので，リターンフレームを復活して呼び出し先に戻る。一方，引数スタックの先頭が印でない場合は，まだ適用する値が残っていることになる。このとき，(型チェックを通っていれば) アキュムレータにはクロージャがあるはずなので，それに引数スタックに積まれた値を適用する，という形になっている。これにより，戻り値である関数の引数がすでに与えられている場合に，リターンしてから再度呼び出すのではなく，直接呼び出すことが可能になっている。このようにして，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される。

`Grab` と `Return` のときに引数スタックの印を見るので，`reset`や `shift`を入れたときにもこれらが正しく保存，破棄されている必要がある。

4 実装

今回の `shift/reset`導入では，de Bruijn notation の拡張ラムダ計算の段階までは`shift`，`reset`用の構文をそれぞれ加え，単純にボディ部分(`shift (fun <var> -> <exp>)`，`reset (fun () -> <exp>)` の`<exp>`) を Caml Light の中間言語に合わせて変換した。型推論やコード生成の処理のため，`shift`と `reset`は特別な構文とみなして構文解析している。

コード生成の段階では，引数をクロージャとしてアキュムレータにセットし，プリミティブの `shift`，`reset`を実行する形に変換した。ここで，`shift`と `reset`のボディ部分の末尾には，

shift/reset の実行の終わりを示す命令を入れている。プリミティブの shift や reset の挙動は，生成されたコードを実行するバイトコードインタプリタで定義した。

なお，型推論には Asai と Kameyama の多相の型推論 [3] を使用した。Caml Light の関数の実行順序である call-by-value，right-to-left に合わせつつ，答の型も含めて推論している。Caml Light には value restriction (let に束縛される式が値でないときには，結果の型にある多相型は一度しか unify 出来ない，弱い多相型になる) が入っているので，それをそのまま使用し，今のところ let で束縛される式が pure なら多相にするという条件は入れていない。型については，5 章でもう少し詳しく説明する。

4.1 バイトコードインタプリタでの実装

MinCaml コンパイラの実装では，reset のときにスタックに印を入れ，shift のときには直近の印までの部分を切り取るという処理をしていた。また，reset の印を保存するときには直下に必ず戻り番地が保存されている，というインバリアントを守っていた。バイトコードインタプリタでの実装もこの考察にしたがった。以下，それぞれの実装を見る。

reset は，

1. リターンスタックにリターンフレーム (戻り番地) 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。reset の印の下には必ずリターンフレームが積まれているのが Caml Light におけるインバリアントである。

Caml Light には引数スタックとリターンスタックの 2 本のスタックがあるので，その両方に reset の印を入れる。引数スタックで使う reset の印は通常の値として積まれることはない特定の値，リターンスタックで使う reset の印は 1 つ前の reset の印を指すポインタ (reset ポインタ) として実装した。reset ポインタの値を保存するときには，現在の値をスタックに保存し，保存したスタックポインタの番地で更新する。

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. 2 つのスタックの移動したフレームの大きさ，プログラムカウンタ，環境，キャッシュサイズ，フレームをコピーする命令へのプログラムカウンタ，引数スタックの位置，ヒープのトラップポインタを，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に，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

ZINC の通常のクロージャが持つ情報はプログラムカウンタと環境だけだが，shift で作るクロージャにはスタックフレームやキャッシュサイズの情報も含める。これは，shift が呼び出されたときの状態を，切り取った継続の呼び出しの際に復元するためである。

例外の実装に用いられているトラップフレームには，スタックへのポインタ (古いトラップフレームへのポインタと，引数スタックへのポインタ) が含まれている。try のときには，リターンスタックにトラップフレームを保存し，現在のトラップフレームへのポインタ (トラップポイ

ンタ) を更新する。raise のときには，リターンスタックをトラップポインタまで下げ，引数スタックはそのトラップフレームが指すところまで下げられる。shift を切り取った結果，現在のトラップポインタの指しているトラップフレームも切り取られてしまう場合には，現在のトラップポインタを残っているフレームの最新のトラップフレームに変更する。また，切り取られたフレームがあとでコピーし戻された際，トラップフレームの中にあるポインタが正しくつながるようにする必要がある。

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，普通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にあった値をセット
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

切り取った継続の実行では，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。

現在の実装では，shift で切り取った継続の呼び出しのときには必ずリターンフレームが保存されるようにしているので，リターンフレームは保存せずに reset の印だけ保存している。これで，リターンスタックにおける reset の印の下には必ずリターンフレームが保存されている，というインバリアントを保っている。

そして，shift や reset の本体の実行が終了したときには，Returen 命令と同じ処理を入れる。これは，reset や切り取られた継続が関数を返す場合 (5.3 節参照)，すでに引数スタックにその引数が積まれているときには，返される関数を直接呼び出せるからである。

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

リターンスタックの reset ポインタ (reset の印) については，印の下までスタックを下げた上で，スタックに保存されていた値で更新する。

ここで，実際に reset や shift を含む式の実行がどのように行われているのか見てみる。

reset の最後には Return 命令を実行する

reset (fun () -> fun x -> x) 3 という式では，まず，引数スタックに MARK (Pushmark 命令で積まれる印) と 3 を積み，reset を実行する。その結果，reset のボディである fun x -> x が返り値としてアキュムレータにセットされた状態で reset 本体の実行が終了する。そして，引数スタックとリターンスタックは reset の印まで下げられ，引数スタックの先頭の値は 3 になる。この状態で Return 命令を実行するので，アキュムレータにある fun x -> x のクロージャに 3 が直接渡される。

このように，ZINC で関数適用の終わりに，まだ引数があるときにはクロージャを作らずに，直接ジャンプして関数適用を実行する，という動作が reset が入っても実現出来ている。

shift は引数スタックとリターンスタックの両方を切り取る

reset (fun () -> shift (fun k -> k (fun x -> x + 2)) 3) という式では，shift (fun k -> ...) 3 は関数適用なので，MARK と引数 3 を引数スタックに積み，shift を実行する。shift では引数スタックの MARK と 3 を切り取ってヒープに移動し，実行は shift の引数の関数に移

る。リターンスタックの reset の印の上には何も積まれていないので，リターンスタックからは何も切り取らない。

切り取った継続は引数スタックとリターンスタックの両方を復活する

k の実行では，リターンスタックの先頭にあった値 (fun x -> x + 2 のクロージャ) を shift を実行した結果と思ってアキュムレータにセットする。そして，引数スタックとリターンスタックに reset の印を積んだ上で，引数スタックに MARK と 3 をコピーし，shift の継続の実行に行く。

引数スタックには切り取られた継続に適用されている値，リターンスタックには継続を切り取った時点までの関数の呼び出し関係が保存されている。これらをスタックにコピーし戻すことで，切り取った継続の実行に必要な値を復活している。

4.2 GC

Caml Light にはもともと GC の機構が備わっている。GC が開始されると，引数スタックとリターンスタックに格納されている値からポインタが探索される。スタックにあるポインタはまだ生きているので，指しているフレームをヒープに格納し直す。また，ポインタは新しいフレームを指すように付け替える。さらに，格納し直すフレームのヘッダ情報 (フレームのサイズ，GC で使用される色，フレームの種類が分かる) から，フレーム内部のポインタも調べる必要があるかを判断して，必要な場合は再帰的に内部のポインタを再帰的にたぐって処理する。

shift で作るクロージャのフレームにもヘッダ情報があり，フレームサイズも正しく保存しているので，通常のクロージャと同様の処理で GC をすることが出来る。ただし，リターンスタックへ新たに reset ポインタを加えているので，それをたどる処理を加えている。現在のところ，この変更だけで shift/reset をサポートした状態での GC が正しく実現出来ている。

5 実行例

本章では，改造した Caml Light で shift/reset を含むプログラムを実行した例を示す。実際的なプログラミング言語でのプログラム例として，また，実装して出てきた問題点を述べるために，比較的多くの例を挙げている。

実行結果として表示される関数の型は，すべて答の前後の型も含めた 4 つ組にしている。そこで，はじめに shift/reset の型システムについて簡単に説明する。

5.1 shift/reset の型システム

shift/reset の型システムでは，judgement は式の型だけでなく答の前後の型も持った以下のような形で表される [3, 4]。

$$\Gamma; \alpha \vdash e : \tau; \beta$$

これは，型環境 Γ のもとで，e の型は τ 型であり，e の実行によって答の型は α 型から β 型に変化することを表す。答の型というのは，直感的には e を囲む reset が返す型のことである。

一方，任意の型 α について $\Gamma; \alpha \vdash e : \tau; \alpha$ が成り立つとき，つまり，e の実行が答の型に影響しないとき，式 e は pure であるといい，

$$\Gamma \vdash_p e : \tau$$

と表す。pure であることは，普通の CPS のプログラムの答の型は何でも良いことに対応する。定数や reset で囲まれた式は pure である。

関数の型は S / A -> T / B と答の前後の型を含めた 4 つ組で表される。/ の後ろに書かれた答の型 A，B を無視すると，通常の関数の型と同じ S -> T と読める。答の型を考慮すると，S 型から T 型への関数だが，適用されたときに答の型を A 型から B 型に変化させることを表す。任意の型 A について，ある関数の型が S / A -> T / A と表されるとき，その関数は pure であるといい，答えの型を省略して S -> T と表すことがある。今回の実装でも，関数が pure なときには答の型を省略して表示している。

5.2 times

例外処理の例として times 関数を示す。times は整数のリストを受け取り，その要素をすべて掛け合わせた値を返す。要素に 0 がある場合の結果は 0 になるので，0 が見つかった時点で「残りの計算を捨てて 0 を返す」というコードを shift/reset を用いて書くと，以下のようになる。

```
# let rec times0 = function
   | [] -> 1
   | 0 :: _ -> shift (fun k -> 0)
   | a :: rest -> a * times0 rest;;
times0 : int list / int -> int / int = <fun>
# let times lst = reset (fun () -> times0 lst);;
times : int list -> int = <fun>
# times [1; 2; 3];;
- : int = 6
# times [1; 2; 0; 3];;
- : int = 0
```

要素に 0 が見つかったときに shift を使って残りの計算 k を切り取り，それを使わずに 0 を返す。これにより，例外処理と同様の処理を実現している。ワイルドカード (_) を含む任意のパターンも書けるのが，MinCaml との相違点である。

先頭が 0 だった場合，shift が現在の継続を捨てて，取り囲む reset に 0 を返すことを反映して，times0 の答の前後の型は int になっている。したがって，reset (fun () -> print_int (times0 [1; 2; 3])) のように，取り囲む reset の型が unit の場合，型が付かない。一方，times は pure なので適用する時点の答の型は任意となり，reset (fun () -> print_int (times [1; 2; 3])) には型が付く。(この式の reset は，なくても良い。)

times0 と times の違いは，実行にも表れる。

```
# reset (fun () -> times0 [1; 2; 0; 3] + 4);;
- : int = 0
# reset (fun () -> times [1; 2; 0; 3] + 4);;
- : int = 4
```

times0 は □ + 4 という継続も含めて切り取るので結果が 0 になるのに対し，times は切り取る継続の範囲が関数内に収まるので，結果は 0 に 4 を足して，4 になる。関数が pure であるときには，reset なしで適用しても答の型は変化せず，継続の作用は関数適用の内部に限定される。

5.3 append, sprintf

関数の答の型が変化する例として，受け取った 2 つのリストを結合する append 関数を示す [3]。

```
# let rec append = function
   | [] -> shift (fun k -> k)
   | a :: rest -> a :: append rest;;
```

```
append : 'a list / 'b -> 'a list / ('a list -> 'b) = <fun>
# let app123 = reset (fun () -> append [1; 2; 3]);;
app123 : int list / '_a -> int list / '_a = <fun>
# let app123' lst = (reset (fun () -> append [1; 2; 3])) lst;;
app123' : int list -> int list = <fun>
# app123 [4; 5; 6];;
- : int list = [1; 2; 3; 4; 5; 6]
```

継続の答の型の型 (取り囲む reset に返される値の型) が 'b の状態で append が呼び出されたとしよう。最終的に shift (fun k -> k) が実行されると，ここで切り取られる継続は (append 自体は 'a list 型を返すので) 'a list -> 'b 型になる。この継続が取り囲む reset に返されるので，結果として継続の答の型は'b から 'a list -> 'b に変化している。したがって，この例は答の型が変わること (answer type modification) を許す型システムでなければ型が付かない。

value restriction が入っているため，let で束縛されている値 app123 の答の型は弱い多相型 '_a になっている。これは，let で束縛される値は pure である，という条件ならば通常の多相型 'a になる。ただし app123' のように int のリストを受け取る関数として定義しても 'a に出来る。(int list -> int list は int list / 'a -> int list / 'a の省略形である。)

もう 1 つ，関数の答の型が変化する例として，sprintf 関数を示す [3]。

```
# let int x = string_of_int x;;
int : int -> string = <fun>
# let str (x : string) = x;;
str : string -> string = <fun>
# let percent to_str = shift (fun k -> fun x -> k (to_str x));;
percent : ('a / 'b -> 'c / 'd) / 'e -> 'c / ('a / 'b -> 'e / 'd) = <fun>
# let sprintf p = reset (fun () -> p ());;
sprintf : (unit / 'a -> 'a / 'b) -> 'b = <fun>
# (sprintf               (* sprintf ("The value of %s is %d.", "x", 3) *)
   (fun () -> "The value of " ^ (percent str) ^ " is " ^ (percent int) ^ ".")) 3 "x";;
- : string = "The value of x is 3."
```

percent は，'a 型の値を文字列にする関数を受け取り，'a 型の値を受け取ったら文字列に変換してから percent が呼び出された時点の継続を施す関数を返す。この挙動を反映して，percent の答の型は'e 型から 'a / 'b -> 'e / 'd 型に変化している。

sprintf はフォーマットを表すサンクを受けとり，それを reset の中で実行する。実際に実行すると，フォーマットに書かれた percent の数だけ値を受け取って文字列を返す関数が返される。上の例では int 型の値 3 と string 型の値"x" を 1 つずつ受け取り，"The value of x is 3." という文字列が返っている。Caml Light の実行順序は right-to-left なので (percent int) が先に実行され，通常の sprintf 関数とは異なり int 型の値を先に受け取る形になっている。

この例の sprintf (fun () -> ...) で返ってくるのは，int 型の値を受け取り，string 型の値を受け取ったら文字列を返す関数である。したがって，バイトコードインタプリタで，shift の引数の式の実行を終えた時点での引数スタックの先頭が印かを考慮しないと，実行が失敗する例でもある。

5.4 prefix

shift の切り取る継続の答えの型が多相に使われている例として，prefx 関数[1]を示す [3]。

[1] prefix は Caml Light の予約語なので prefx としている。

```
# let rec visit = function
   | [] -> shift (fun k -> [])
   | a :: rest -> a :: shift (fun k -> k [] :: reset (fun () -> k (visit rest)));;
visit : 'a list / 'b -> 'a list / 'b list = <fun>
# let prefx lst = reset (fun () -> visit lst);;
prefx : 'a list -> 'a list list = <fun>
# prefx [1; 2; 3];;
- : int list list = [[1]; [1; 2]; [1; 2; 3]]
```

prefx は受け取ったリストの全てのプリフィックスをリストにして返す関数である。関数 visit で，リストに要素がある場合の shift に切り取られる継続 k の型は ∀ 't.'a list / 't -> 'a list / 't と，答の型が多相になる。実際に，k [] は最終結果としてプリフィックスのリストが返るので答の型が 'a list list，つまり，k の型は'a list / 'a list list -> 'a list / 'a list list として用いられているのに対し，k (visit rest) はその外側の reset にプリフィックスの 1 つが返るので，答の型は 'a list となり k の型は'a list / 'a list -> 'a list / 'a list になる。したがって，prefx は shift で切り取る継続 k の答の型が多相でなければ型が付かない。

5.5 traverse

ユーザが新しく型を定義する例として traverse を示す [10]。tree_t 型は整数を要素に持つ二分木，関数 walk は tree_t 型の木を受け取ったらそのセルの要素を 1 つ option 型で返す関数である。要素が見つかったときには関数 suspend を呼び出し，shift でそのときの継続を切り取って ref 型のセル resume に保存した上で要素を返す。resume に保存された継続を呼び出すと残りの計算が実行される。これは，実行を中断してそのときの状態を継続として保存し，あとで実行を再開しているので，プロセスのサスペンドをモデル化していると見なすことが出来る。

```
# type tree_t = Null
              | Cell of int
              | Pair of tree_t * tree_t;;
Type tree_t defined.
# let tree = Pair (Pair (Cell 1, Null), Pair (Cell 2, Cell 3));;
tree : tree_t = Pair (Pair (Cell 1, Null), Pair (Cell 2, Cell 3))
# let resume = ref (fun (x : int option) -> x);;          (* 残りの計算を保存するセル *)
resume : (int option / '_a -> int option / '_a) ref = ref <fun>
# let start f = reset (fun () -> f ());;                (* thunk を reset の中で実行 *)
start : (unit / 'a -> 'a / 'b) -> 'b = <fun>
# (* 呼び出されたときの継続を resume に保存し，値を返す *)
  let suspend v = shift (fun k -> resume := k; v);;
suspend : 'a / int option -> int option / 'a = <fun>
# let rec walk = function
   | Null -> None
   | Cell i -> suspend (Some i)
   | Pair (t1, t2) -> walk t1; walk t2;;
walk : tree_t / int option -> int option / int option = <fun>
# let get_first t = start (fun () -> walk t);;
get_first : tree_t -> int option = <fun>
# let get_next () = start (fun () -> !resume None);;
get_next : unit -> int option = <fun>
# get_first tree;;
- : int option = Some 1
```

```
# get_next ();;
- : int option = Some 2
# get_next ();;
- : int option = Some 3
# get_next ();;
- : int option = None
```

get_first は walk を呼び出して，受け取った木のセルの要素を 1 つ返す。その後 get_next が呼び出されると同じ木の残りの要素が返され，参照していないセルがなくなった時点で None が返る。

継続を保存する ref 型のセル resume の答の型は弱い多相型 '_a になっている。しかし，get_next の定義によって '_a は int option 型に変化するので，例えばさらに，

```
# let get_next' () =
     start (fun () -> (match !resume None with
                         | None -> print_string "none\n"
                         | Some i -> print_int i; print_newline ()));;
get_next' : unit -> unit = <fun>
```

という定義をしようとすると，答の型の unit が int option と unify 出来ず，型エラーになる。

5.6 部分評価

shift/reset を使ったオンライン部分評価の例を示す [1]。ここで，t 型は 1 章の抽象構文木，to_string は t 型の項を文字列に変換する関数，gensym は受け取った変数を fresh にして返す関数，init は gensym を初期化する関数である。

```
# let empty_env v = failwith ("unbound variable " ^ v);;                    (* 空の環境 *)
empty_env : string -> 'a = <fun>
# let get var env = env var;;
get : 'a -> ('a / 'b -> 'c / 'd) / 'b -> 'c / 'd = <fun>
# let add env name v var = if var = name then v else get var env;;
add : ('a / 'b -> 'c / 'b) -> 'a -> 'c -> 'a / 'b -> 'c / 'b = <fun>
# (* 動的な型と静的な型 *)
  type sval_t = Dyn of t | Sta of t * (sval_t / sval_t -> sval_t / sval_t);;
Type sval_t defined.
# let lift = function                                    (* プログラム（項）を取ってくる *)
  | Dyn d -> d
  | Sta (d, s) -> d;;
lift : sval_t -> t = <fun>
# let rec peval term env = match term with                          (* 部分評価器本体 *)
   | Var x -> get x env
   | Lam (x, t) ->
       let new_x = gensym x in let new_k = gensym "k" in
         Sta (Lam (new_x, Shift (new_k,
                lift (reset (fun () -> Dyn (Reset (App (Var new_k,
                  lift (peval t (add env x (Dyn (Var new_x))))))))))),
              fun arg -> peval t (add env x arg))
   | App (t1, t2) ->
       let f = peval t1 env in let a = peval t2 env in
         (match f with
            | Dyn d -> let new_t = gensym "t" in
                shift (fun cont -> Dyn (Let (new_t, App (d, lift a),
                                             lift (cont (Dyn (Var new_t))))))
            | Sta (d, s) -> s a)
```

```
   | Shift (k, t) ->
       shift (fun cont -> let new_v = gensym "v" in
                peval t (add env k
                          (Sta (Lam (new_v, Reset (lift (cont (Dyn (Var new_v))))),
                                cont))))
   | Reset t -> reset (fun () -> peval t env)
   | Let (x, t1, t2) -> peval (App (Lam (x, t2), t1)) env;;
peval :
 t / sval_t ->
 ((string / sval_t -> sval_t / sval_t) / sval_t -> sval_t / sval_t) /
 sval_t = <fun>
# let f term = init ();                                        (* gensym を初期化して, *)
    let result = lift (reset (fun () -> peval term empty_env)) in    (* 部分評価し, *)
    print_string (to_string result); print_newline ();;                (* 結果を出力 *)
f : t -> unit = <fun>
# let e = Lam ("x", Reset (App (Shift ("k", Var "k"), Var "x")));;
e : t = Lam ("x", Reset (App (Shift ("k", Var "k"), Var "x")))
# f e;;
(lam x1. (shift k2. (reset (k2 @ (lam v3. (reset (let t4 = (v3 @ x1) in t4)))))))
- : unit = ()
```

sval_t 型は，プログラムとして残される動的な値と，部分評価中に実行出来る静的な値を表す。動的な値 Dyn は部分評価の際には静的な値が分からないのでプログラムだけを持つが，静的な値 Sta は既知の値をプログラムと関数 (継続) の両方の形で保持している。関数 lift は sval_t 型の値からプログラムを得る関数である。peval が部分評価をする関数であり， shift は関数適用のときの let-insertion と， shift 自体の実行に使われている。

現在の実装では，let f (x : int -> int) = x のように，関数の答の型が省略されたときには pure な関数と見なして int / 'a -> int / 'a と型変数 'a の補完を行っているが，型定義のときは関数の答の型は省略出来ない形になっている。これは補完すると，

```
# type t = A of int / 'a -> int / 'a;;
Toplevel input:
> type t = A of int / 'a -> int / 'a;;
>                                  ^^
The type variable a is unbound.
```

と同じように，型定義の中に束縛されていない型変数が現れてしまうためである。機械的に型自体に多相の型変数を加え，type 'a t = A of int / 'a -> int / 'a などと定義すれば型が付く場合もあるが，構文木の中のどこかの場所で具体的に 'a が定まってしまった場合，全体として型が付かなくなることもある。実際に，上の sval_t 型は

```
type 'a sval_t =
  | Dyn of t
  | Sta of t * ('a sval_t / 'a -> 'a sval_t / 'a);;
```

という定義や,

```
type ('a, 'b) sval_t =
  | Dyn of t
  | Sta of t * (('a, 'b) sval_t / 'a -> ('a, 'b) sval_t / 'b);;
```

という定義では，peval を定義する段階で答えの型が unify 出来なくなり，破綻する。

6　関連研究

Gasbichler と Sperber は `control` と `shift/reset` の直接実装法を示し，実際に Scheme48 上で実装した [9]。それにより，`call/cc` を使った間接実装によるオーバーヘッドが軽減出来ること，実行効率が向上することを示した。Gasbichler らは incremental stack/heap 戦略を用いており，実装には Scheme48 の仮想機械である PreScheme を使用している。

Rompf らは Scala の型システムをプラグイン出来る機構を用いて answer type modification を許す `shift/reset` の実装を示した [16]。継続が切り取られる部分を型を使って特定し，その部分を選択的に CPS 変換することで，プログラム全体を CPS 変換するよりも効率的に `shift/reset` を実装している。大規模なプログラミング言語で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 [12]。例外とコントロールスタックのオーバーフローからの回復の機構を持つプログラミング言語ならば，既存の実装に手を加えずに直接実装が出来る。本研究では Caml Light の実装に変更を加えているので，その点で Kiselyov の手法は優れている。しかし answer type modification には対応していないので，5.3 節の `percent` 関数のような `shift` の使い方は，直接には出来ない。また，Caml Light はスタックのオーバーフローからの回復をサポートしていないので，この手法は適用出来ない。

7　まとめと今後の課題

5 章に示したように，関数の答の前後の型が変化するプログラムや `shift` で切り取る継続の答の型が多相に使われているプログラムを含めて，各種の例が動作している。MinCaml は機械語，Caml Light は ZINC 抽象機械と抽象度が異なるものの，インバリアントなどの基本的な方針はそのままで実装出来た。今のところ，型推論を通ったプログラムは正しく実行出来ているが，`call/cc` と 1 つのセルで `shift/reset` の挙動を模倣する，Filinski による answer type modification を許さない実装 [8] で実行出来るプログラムで，型推論を通っていないものがある。

今後の課題は上記の問題を解決して，システム全体を再度コンパイルすることである。現在は，もともと定義されているシグネチャを関数がすべて pure であると仮定して利用しているため，ライブラリに定義された関数 (特に `map` などの高階関数) を利用したプログラムの型推論が正しく出来ない場合がある。これはシグネチャを書き直して再コンパイルすれば解決出来る問題と考えられる。

実装を完成させた上では CPS 変換や選択的な CPS 変換などとの速度，効率の比較をしてみたい。実装自体に関しても，`reset` の印を入れる回数やフレームのコピー量を減らす工夫をしたいと考えている。最終的には ZINC 抽象機械での挙動の定義から，実装の正当性の証明を目指す。

謝辞 有益なコメントを下さった査読者の皆様に深く感謝致します。

参考文献

[1] Kenichi Asai. Logical relations for call-by-value delimited continuations. *Trends in Functional Programming*, 6:64–78, 2007.

[2] Kenichi Asai. On typing delimited continuations: Three new solutions to the printf problem. *Higher-Order and Symbolic Computation*, 2009. doi=10.1007/s10990-009-9049-5.

[3] Kenichi Asai and Yukiyoshi Kameyama. Polymorphic delimited continuations. In *5th Asian Symposium on Programming Languages and Systems, Lecture Notes in Computer Science 4807*, pages 239–254, November 2007.

[4] Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. Technical Report 89/12, DIKU, University of Copenhagen, July 1989.

[5] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 151–160, June 1990.

[6] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[7] Matthias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 180–190, January 1988.

[8] Andrzej Filinski. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 446–457, January 1994.

[9] Martin Gasbichler and Michael Sperber. Final shift for call/cc: direct implementation of shift and reset. In *Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*, pages 271–282, October 2002.

[10] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 12–23, 1995.

[11] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Shifting the stage: staging with delimited control. In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial Evaluation and Program Manipulation*, pages 111–120, January 2009.

[12] Oleg Kiselyov. Delimited control in OCaml, abstractly and concretely. system description. In *10th International Symposium on Functional and Logic Programming*, April 2010. To appear.

[13] Xavier Leroy. The zinc experiment: An economical implementation of the ML language. Technical report, INRIA, February 1990.

[14] Xavier Leroy. *The Caml Light system release 0.74*, December 1997.

[15] Moe Masuko and Kenichi Asai. Direct implementation of shift and reset in the MinCaml compiler. In *Proceedings of the 2009 ACM SIGPLAN workshop on ML*, pages 49–60, August 2009.

[16] Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 317–328, 2009.

[17] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten (Editors). Revised^6 report on the algorithmic language Scheme, `http://www.r6rs.org/`, 2007.

[18] Eijiro Sumii. MinCaml: a simple and efficient compiler for a minimal functional language. In *Proceedings of the 2005 workshop on Functional and Declarative Programming in Education*, pages 27–38, September 2005.

[19] Peter J. Thiemann. Cogen in six lines. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, pages 180–189, 1996.

[20] 住井 英二郎. MinCaml コンパイラ. **コンピュータソフトウェア**, 25(2):28–38, 2008.