MinCaml コンパイラにおける shift/reset の実装

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

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う機能を導入すると，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。

継続をプログラムで扱うには，プログラムを CPS 形式で書くか，CPS 変換を施せばよい。しかし，CPS 変換を施すとプログラム全体が書き換わるので，局所的に継続を扱いたい場合には不向きである。

一方，call/cc や control/prompt，shift/reset などの継続を扱える命令を導入すると，プログラムを大きく書き換えることなく，継続が扱えるようになる。call/cc については効率的な実装方法が研究されているが，shift/reset の実装についてはまだあまり研究されていない。そこで本研究では，MinCaml コンパイラに限定継続の命令 shift/reset を導入し，その直接実装を試みる。またその改良として，フレームを lazy にコピーする実装方法を示す。

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う機能を導入すると，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。継続計算が有用に使われる例には，例外処理の制御，コルーチン，探索問題や非決定性プログラミング，深い分岐からの脱出などがある。

継続をプログラムで扱うには，プログラムを CPS 形式で書くか，CPS 変換を施せばよい。しかし CPS 変換は大域的な変換なので，プログラム全体の構造が大きく変わってしまい，少ない変更で継続を扱うことが出来ない。そのため例外のように局所的に継続を扱いたい場合には不向きである。

一方，call/cc [11] や control/prompt [5] ，shift/reset [3] などの継続を扱える命令を用いると，プログラムを大きく変更することなく継続が扱える。call/cc は Scheme や Ruby [10] に導入されており，効率的な実装方法の研究も数多くなされているが，shift/reset の実装については Gasbichler らの Scheme 48 における実装 [7] ぐらいしか例がない。そのため，(限定) 継続を使ったプログラムの開発を妨げるばかりでなく，その効率についての議論も難しくなっている。限定継続の様々な興味深い応用が考えられはじめている [1, 2, 9] 現在，これは残念な状況である。

このような状況を打開するため，本研究では Sumii による MinCaml コンパイラ [13] をベースに，限定継続の命令 shift/reset を導入し，その直接実装を行う。shift/reset は PowerPC の機械語により，スタックのフレームをヒープに移動することで実装される。本論文では実装の詳細を述べるとともに，shift/reset の効率について，いくつかのベンチマークプログラムで議論する。さらに，フレームのコピー回数を減らすため，lazy にフレームをコピーする手法についても紹介する。本研究により，shift/reset の直接実装法が確立されるとともに，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 章では継続と shift/reset について概説する。3 章では実際の実装，4 章では実装の改良として lazy にフレームをコピーする実装について述べ，5 章では 2 つの実装を評価する。6 章では関連研究について，7 章でまとめと今後の課題を述べる。

$$
\begin{aligned}
[\![x]\!] &= \lambda\kappa.\kappa\ x \\
[\![\lambda x.M]\!] &= \lambda\kappa.\kappa\ (\lambda x.[\![M]\!]) \\
[\![M\ N]\!] &= \lambda\kappa.[\![M]\!]\ (\lambda m.[\![N]\!]\ (\lambda n.m\ n\ \kappa)) \\
\\
[\![\texttt{shift (fun } k \texttt{ -> } M\texttt{)}]\!] &= \lambda\kappa.([\![M]\!]\ (\lambda m.m))[k \mapsto \lambda a.\lambda\kappa'.\kappa'(\kappa\ a)] \\
[\![\texttt{reset } M]\!] &= \lambda\kappa.\kappa\ ([\![M]\!]\ (\lambda m.m))
\end{aligned}
$$

図 1: 継続のセマンティクスと shift/reset

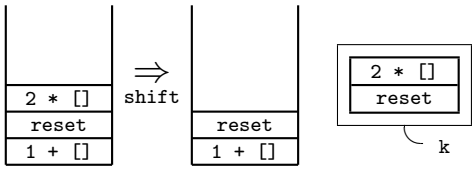

図 2: shift を呼び出したときの継続のスタックの変化

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ための命令である [3, 4] 。ここで，shift は現在の継続を切り取って関数にする命令，reset は shift が切り取る継続の範囲を限定する命令である。例えば，`1 + reset (2 * shift (fun k -> 3 + k 4))` という式を考えると，この式の shift で切り取られるのは『2 を掛ける』という継続なので，

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

となる。

継続のセマンティクスによる shift/reset の定義 [3] を図 1 に示す。$[k \mapsto \lambda a.\lambda\kappa'.\kappa'(\kappa\ a)]$ は，k を $\lambda a.\lambda\kappa'.\kappa'(\kappa\ a)$ で置き換えることを表す。`shift (fun` k `->` M`)` は現在の継続 κ を切り取って k に束縛した上で，式 M を空の継続で実行する。`reset` M は引数の式 M を空の継続 (恒等関数) で実行し，返って来た値を `reset` M 自体の継続 κ に渡す。

継続の概念でプログラムを解釈すると，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 は引数の式 M を空の継続で実行するのでスタックに reset の印を入れる命令，また，shift は reset の印までのフレームを切り取って関数にし，それを変数 k に束縛した上で式 M を実行する命令と考えられる。例えば，継続のスタックにおける `1 + reset (2 * shift (fun k -> ...))` という式の実行で shift を呼び出したときを考えると図 2 のようになる。reset と同様，shift の式 M も空の継続で実行するので，フレームを切り取るときには reset の印をスタックに残す。さらに shift/reset の場合，継続は静的に決定されるので，k のフレームにも reset の印が入る。

3 shift/reset の実装

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 [6] 。しかし，インタプリタはコンパイラに比べて実行速度が遅い。また，call/cc はスタックを全てヒープにコピーするので実行効率が悪い。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。そこで本章では，Gasbichler らが Scheme48 上に shift/reset を実装した方法 [7] に基づき，shift/reset を直接実装する方法を示す。

3.1 準備

今回の実装では，MinCaml コンパイラ [13] をもとに，MinCaml をリストと `shift`/`reset` で拡張した言語を受け取り，PowerPC の機械語のコードを出力するコンパイラを作成した。

MinCaml コンパイラは，OCaml のサブセット MinCaml のコンパイラである。変数定義，関数定義，関数適用などが出来て，整数，浮動小数点数，真偽値，変数，組，配列などが扱える。オリジナルの MinCaml コンパイラでは単相の型システムを用いているが，`shift`/`reset` が入ると答えの型に関する多相性が必須となるので，今回は Asai らの多相の型システム [2] を使用した。

3.2 実装方針

プログラムの実行中，一時的に保存したい値や戻り番地 (PowerPC の場合，戻り番地はリンクレジスタに格納される) はスタックに保存するので，`shift` を呼び出すときに，`reset` から `shift` までにスタックへ保存された変数が連なるフレームをヒープに移動すれば，`shift` が切り取る継続の計算に必要な値が保存出来る。また，`shift` を呼び出した時点での戻り番地にジャンプすれば，実際に継続の計算が出来る。そこで今回は，次のような実装をした。

- `reset` のときは，スタックに `reset` の印を入れる。
- `shift (fun` k `->` M`)` のときは，最も近い `reset` の印の 1 つ上までの部分をヒープに移動する。
- k を呼び出すときは，スタックに `reset` の印を入れた上で，対応するフレームをヒープからスタックにコピーする。

`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。印を入れるときにはそのときの `rp` の値をスタックに保存してからスタックポインタの値を `rp` に代入，印を削除するときにはそのときの `rp` の値を破棄してスタックから `rp` の値を復活する。`reset` や `shift`，k の呼び出しの処理以外で `rp` の値を変更することはない。PowerPC には汎用レジスタが 32 個あるのでレジスタをひとつ `rp` で占有したが，x86 のように汎用レジスタが少ない場合はメモリ上に確保すれば実装可能と思われる。なお，MinCaml コンパイラではスタックに組などのデータが置かれることはないため，ポインタの付け替えをすることなくスタックをコピーすることが可能である。以下，それぞれの実装を詳しく見ていく。各節中の ①，$\cdots$ の処理は厳格に機械語の実装に対応している。

3.3 `reset` の実装

`reset` M は，コンパイラ内部で `let rec` r `() =` M `in reset` r [1]という，関数を引数に受け取る形に変換する。`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 呼び出し

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 (図 3，`RA` は戻り番地 return address の略)。

`reset` は引数の式を空の継続で実行するので，`reset` r のときに行いたいのは r の計算とその後の計算を切り離すことである。`reset` を呼び出す時点での戻り番地の保存によって，`reset` r を実行した後に実行すべき計算の番地を保存するとともに，`rp` もスタックに保存することで，`reset` までのスタックの状態が r 内で実行される `shift` 命令によって壊されないようにしている。

[1] `let` r `() =` M `in reset` r でないのは，MinCaml の構文に再帰しない関数定義がないため。

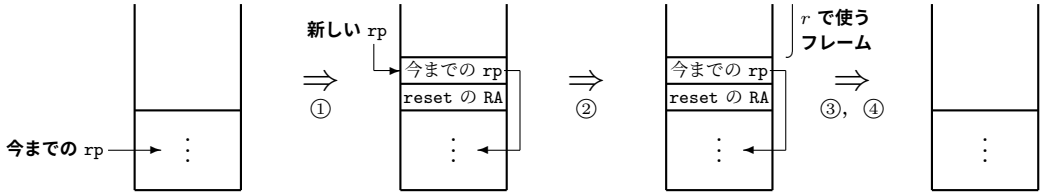

図 3: reset のときのスタックの変化

3.4 shift の実装

shift は引数の式に reset までの継続を切り取って関数にしたものを渡すので，shift（fun k -> M）のときに行いたいのは reset までの継続を関数にして k に束縛した上で，M を (空の継続で) 実行することである。reset までの継続の計算をするには，計算に必要な変数の値と，実行すべき命令列がわかればよい。まず計算に必要な変数の値は，reset が呼び出されてから shift が呼び出されるまでのスタックのフレームに保存されている。また実行すべき命令列は，shift を呼び出した時点での戻り番地で示される。したがって，このふたつの情報を k の表現としてヒープに確保する。具体的には shift は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 (図 4)。s の実行終了後は最も近い 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 を空の継続で実行するからである。

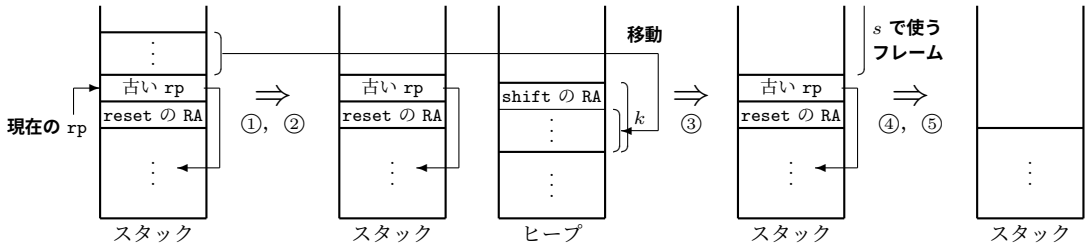

図 4: shift のときのスタックとヒープの変化

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，および shift を呼び出した時点での戻り番地が必要となる。これらの情報から k のクロージャを作り，実際に k が呼び出されたときには，

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ヒープからスタックにコピー

③ 保存されていた，shift を呼び出した時点での戻り番地 (図 5 の「shift の RA」) へジャンプ

の順で処理を行うよう実装した (図 5)。

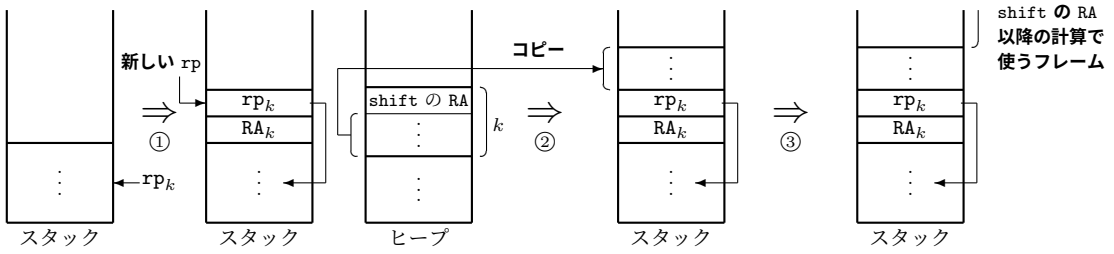

図 5: k を呼び出すときのスタックの変化

3.5 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。多少複雑であるが，一度これだけの考察を行うと，その後実際に必要となるのは shift，reset，そして k を実現する関数のみとなる。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 行程度となる。この実装は shift/reset を使わない場合，rp を占有する以外は今までと全く同じようにコンパイルされる。

4 lazy な実装

3 章で述べた実装では，shift のたびにフレームの移動が必要になる。しかし，実際にはフレームの移動が不要な場合がある。例えば，shift (fun k -> k 3) のように末尾位置でしか k を呼び出していない場合を考えると，shift の時点でフレームをヒープに移動し，その直後 k 3 を実行するときに shift で移動したのと同じフレームをスタックにコピーしてくることになる。この場合，shift の時点ではフレームを移動せずに残しておき，スタックに残したフレームをそのまま用いれば正しい結果が得られたはずである。そこで本章では，必要になるまでフレームをコピーしない **lazy な実装** を考える。

4.1 実装方針

reset r という式を考えると，引数の関数 r 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。r に shift s という呼び出しがある場合，reset を呼び出してから shift を呼び出すまでのスタックのフレームが s の引数 k に対応するフレームである。3 章の実装ではそのフレームを shift の時点でヒープにコピーし，スタックのフレームは破棄したが，lazy な実装では shift の時点ではコピーせず，この部分が k に対応するフレームであることを覚えておき，スタックに残す。そして実際に必要になったとき (k を呼び出すとき) にスタックトップへフレームをコピーする。スタックに残したフレームは，不要になった時点で破棄する。例えば 1 + reset (2 * shift (fun k -> 3 + k 4)) という式の実行を考えると，スタックは以下の図 6 のようになる。3 章の実装よる実行でのコピー回数 2 回よりも 1 回少なく，1 回で済む。

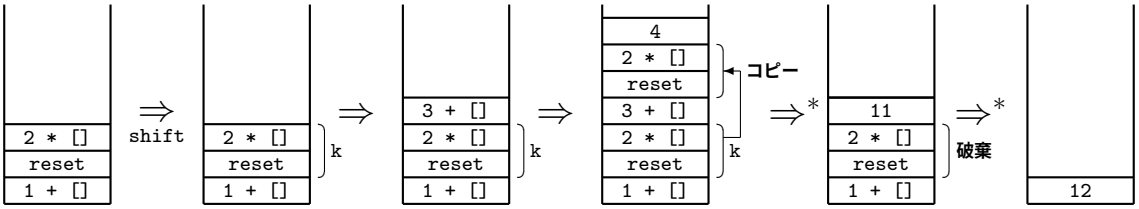

図 6: 1 + reset (2 * shift (fun k -> 3 + k 4)) の実行

さらに末尾位置での k の呼び出しがある場合は，スタックにフレームを残しておくことでフレームのコピーが全く不要になることもある。例えば `1 + reset (2 * shift (fun k -> k 3))` という式を考えると，スタックの状態は以下の図 7 のようになる。

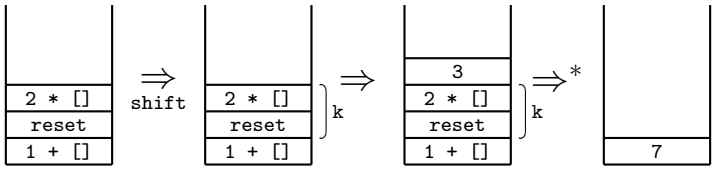

図 7: `1 + reset (2 * shift (fun k -> k 3))` の実行

末尾位置で `k` が呼び出される場合は，呼び出しの時点でスタックトップに `k` のフレームが残っている (左から 3 番目の図) ので，このフレームをそのまま利用するとコピーを完全に省ける。プログラムの字面上は `k 3` という関数呼び出しがあるにも関わらず，実装上は何もせずにスタックに残っているフレームに 3 を渡すのみで，`k 3` が実行出来ることになる。同じ式を 3 章の実装によって実行した場合のフレームのコピー回数が 2 回になるのに対し，この場合のコピー回数は 0 回で済む。`shift (fun k -> k (1 + k 3))` のように末尾位置での呼び出しと末尾位置でない呼び出しの両方がある場合は，末尾位置でない呼び出し `k 3` のときのみフレームをコピーすればよい。末尾位置での `k` の呼び出しは，やはりスタックに残っているフレームに `1 + k 3` の結果を渡すだけで実現出来る。

`shift (fun k -> k 3)` という式は，3 と書いても同じことなので，そもそもこのような式を含むプログラムをユーザが書くことはあまり多くない。しかし `shift`/`reset` を含むプログラムに対して部分評価 (強力な最適化) を行うと，その結果のプログラムにはこのような `shift` 式が多く出現することが知られている [1] 。したがって，末尾位置での k の呼び出しが効率的に実装出来るのは重要である。

以上が lazy な実装の基本的な考えだが，いつでもこのようにスタックにフレームを残しておいてよいわけではない。例えば `shift (fun k -> k)` という式を考えると，この式は現在の継続を切り取ってきて，それをそのまま関数として返してしまうので，`shift` を抜けた後にも k が使われ得る。それにも関わらずフレームをスタックにだけ確保したまま実行を進めると，実際に k を呼び出すときには k に対応するフレームが書き換わっている可能性がある。この場合，`shift` を抜けるまでに k のフレームをヒープにコピーする必要がある。

つまり，`shift` の引数の関数で k がどのように用いられているかによって，k のフレームの扱いを考えなければならない。以下，例を挙げながら場合分けを示す。

- `shift (fun k -> k 3)`

  k は `shift` の中でのみ用いられ，かつ末尾位置で呼び出されているものがある。この場合，フレームはスタックに確保したままで問題ない。末尾位置での k の呼び出しの場合は，スタックトップにフレームが残っているので，それをそのまま再利用し，コピーは不要になる。一方，末尾位置ではない k の呼び出し (例えば `shift (fun k -> k (k 3))` の内側の `k`) もあるときには，スタックからスタックトップへフレームをコピーする。

- `shift (fun k -> 1 + k 3)`

  k は `shift` の中でのみ用いられ，かつ末尾位置での呼び出しがない。上と同様，フレームはスタックに確保したままで問題なく，k を呼び出すときにはスタックからスタックトップへフレームをコピーする。ただし末尾位置での k の呼び出しはないので，`shift` を抜けるとき，スタックに残っているフレームを破棄する。

- `shift (fun k -> a.(0) <- k; k 3)`

  k がグローバルな配列 `a` に代入されるため，k は `shift` を抜けた後にも使われる可能性があり，かつ末尾位置で呼び出されているものがある。`shift` を抜けた後に用いられる場合を考えるとフレームはヒープにコピーする必要がある。さらに末尾位置での k の呼び出しに備え，スタックにもフレームを残しておく。末尾位置ではない k の呼び出しもあるときには，ヒープからスタックトップへフレームをコピーする。

- `shift (fun k -> a.(0) <- k; 1 + k 3)`

  k は `shift` を抜けた後にも使われる可能性があり，かつ末尾位置での呼び出しがない。フレームはヒープに移動する必要がある。末尾位置での k の呼び出しはないので，スタックにはフレームを残さない。k を呼び出すときにはヒープからスタックトップへフレームをコピーする。

上の 3 つめと 4 つめのように，`shift` を抜けた後にも k が使われる可能性がある場合を k が **エスケープする** という。lazy な実装の方針は `reset` の印に加えて `shift` の印も用い，以下のようになる。この中で `shift` の印は，直感的には各 `shift` で切り取る継続の境界を表す (詳しくは 4.3 節で述べる)。

- `reset` のときは，スタックに `reset` の印と `shift` の印を入れる。
- `shift (fun` k `->` M`)` のときは，M における k の使われ方で 4 通りに処理を分ける。
- k を呼び出すときは，スタックに `reset` の印と `shift` の印を入れた上で対応するフレームをヒープまたはスタックからスタックトップにコピーする。

`reset` の印と同様，`shift` の印にはレジスタをひとつポインタ (shift pointer，以下 `shp`) として用いた。以下，それぞれの実装を詳しく見ていく。

4.2 `reset` の実装

`reset` については，3 章の `reset` の処理に `shp` の保存，復活も加えるだけで，

① 戻り番地と `rp`，`shp` をスタックに保存
② `reset` の引数の関数 r 呼び出し
③ 戻り番地と `rp`，`shp` 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。

4.3 `shift` の実装

`shift` のときは，k がエスケープするか，しないか，必ず末尾位置での呼び出しがあるか，ない場合もあるかを調べ，各場合に対応する処理を施す。エスケープするか，しないかの判定の方針は，

- 関数の戻り値はエスケープする
- 関数がエスケープするならば，その関数を定義する時点での自由変数もエスケープする
- 組，リストがエスケープするならば，その要素もエスケープする
- 配列に入る値はエスケープする

とし，型を利用してエスケープ解析をした。このエスケープ解析は，住井 [12] の解析法を shift/reset も扱えるように拡張したものである。それ以外の点については全く標準的なので，他の方法，例えばHannan [8] が提示しているエスケープ解析などを利用しても出来ると思われる。また，必ず末尾位置に k の呼び出しがあるか，ある場合はどれが末尾位置での呼び出しの k かのチェックは構文的に行った。if 文のように分岐する場合は，すべての分岐において末尾位置での呼び出しがあるときのみ，必ず末尾位置での呼び出しがあるとした。

shift は reset と同じように，shift で切り取る継続の範囲を限定する働きもある。3 章の実装では shift のたびに rp までのフレームをヒープに移動していたので，移動した時点で reset のときと同じ状態になり，正しく限定の働きもしていた。しかし lazy な実装では，k がエスケープしない場合フレームをスタックに確保したままにする。この状態で shift が実行されると，スタックに残されている余分なフレームまで切り取られることになる。これを避けるために，shp を用意している。shp はスタックに余分なフレームが残されるたびに更新し，常に残されているフレームの上を指す。そして shift が実行された際には，スタックトップから shp (の 1 つ上) までのフレームを切り取る継続に対応するフレームにする。こうすることで，スタックにフレームが残っていても正しくフレームを対応させられるようになる。

この方針は，末尾位置に k の呼び出しがない場合のものだが，末尾位置に k の呼び出しがある場合は残っているフレームを再利用するための特殊な処理が必要となる。この場合，スタックに残されているフレームは余分なフレームではなく，末尾位置での k の呼び出しを表す，必要なフレームになる。したがって，末尾位置での k の呼び出しがあるときには shp を更新せず，後で shift が実行されたときに切り取られるべきフレームに含まれるようにする。加えて，末尾位置での k の呼び出しを何もしない命令に書き換える。

例えば，1 + reset (2 * shift (fun k -> k (shift (fun h -> k 3 + h 4)))) という式の実行を考えると，図 8 のようになる。ここで，ひとつめの shift では k が末尾位置で呼び出されているため，スタックに k のフレームを確保した上で，末尾位置での k の呼び出しのときにはそのフレームを再利用する。したがって，この時点では shift の印の保存は行わない。この状態でふたつめの shift を実行すると，h に対応するフレームは左から 2 番目の図のように shift の印までとなる。これで正しく h に k [] という継続を取れていることがわかる。h は末尾位置では呼び出されていないので，ふたつめの shift を呼び出した時点で shift の印が保存される (左から 2 番目の図)。

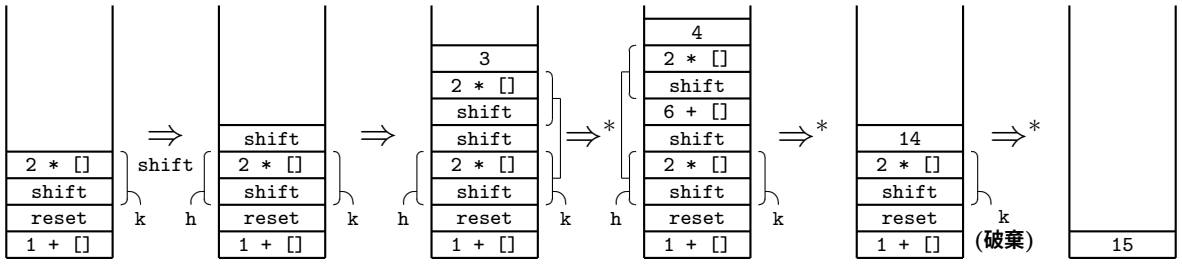

図 8: 1 + reset (2 * shift (fun k -> k (shift (fun h -> k 3 + h 4)))) の実行

shift の呼び出しが終了したときには最も近い reset の残りの計算に移る。3 章の実装では，shift の引数の関数の実行が終了した時点で，スタックトップに rp と戻り番地だけが残っていたのでそれを復活すればよかったが，lazy な実装ではスタックに shift のときに移動させなかったフレームが残っていることがある。末尾位置での k の呼び出しがない場合，このフレームは不要なので破棄する必要がある。そのとき，どこまでのフレームを破棄すればよいのかという印に用いているのが rp である。rp は reset のときと k の呼び出しのときにしか保存しないので，rp までのフレームを破棄し，その下に保存されていた戻り番地へジャンプすれば，reset の残りの計算に移れる (図 8 の左から 5 番目の図)。末尾位置での k の呼び出しがある場合は，スタックに残されていたフレームを使って末尾位置での k の呼び出しの実行をするので，フレームの破棄は不要になる。以下，4 通りの shift の処理をそれぞれ見ていく。

- k がエスケープせず，必ず末尾位置での呼び出しがある場合 (図 9)

  ① shift を呼び出した時点での戻り番地を使って関数 k_{lazy} のクロージャを作る
  フレームはそのままスタックに残し，ヒープにコピーはしない
  ② k_{lazy} を第 1 引数として shift の引数の関数 s に (戻り番地を更新せずに) ジャンプ

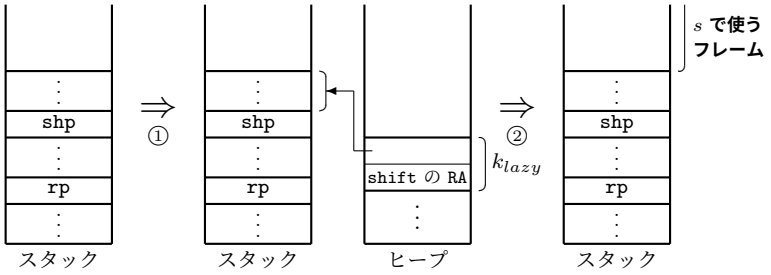

図 9: k がエスケープせず，必ず末尾位置での呼び出しがある場合のスタックとヒープの変化

- k がエスケープせず，末尾位置での呼び出しがないかも知れない場合 (図 10)

  ① shift を呼び出した時点での戻り番地を使って関数 k_{lazy} のクロージャを作る
  フレームはそのままスタックに残し，ヒープにコピーはしない
  ② shp をスタックに保存 (rp と戻り番地の保存はしない)
  ③ k_{lazy} を第 1 引数として shift の引数の関数 s を呼び出す
  ④ rp の 2 つ上までのフレームを破棄
  ⑤ 戻り番地と rp，shp を復活 (rp の上下には必ず shp と戻り番地が保存されている)
  ⑥ 戻り番地へジャンプ

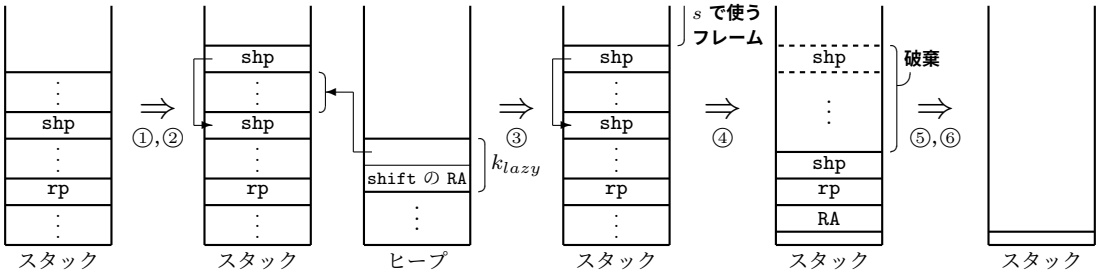

図 10: k がエスケープせず，末尾位置での呼び出しがないかも知れない場合のスタックとヒープの変化

- k がエスケープし，必ず末尾位置での呼び出しがある場合 (図 11)

  ① 現在の shp の 1 つ上までのフレームをヒープにコピー
  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 に (戻り番地を更新せずに) ジャンプ

- k がエスケープし，末尾位置での呼び出しがないかも知れない場合 (図 12)

  ① 現在の shp 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

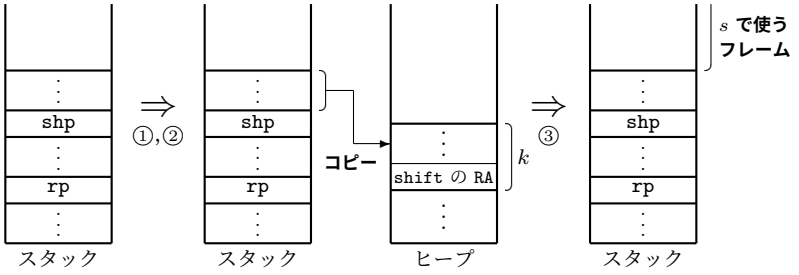

図 11: k がエスケープし，必ず末尾位置での呼び出しがある場合のスタックとヒープの変化

④ rp の 2 つ上までのフレームを破棄

⑤ 戻り番地と rp，shp を復活 (rp の上下には必ず shp と戻り番地が保存されている)

⑥ 戻り番地へジャンプ

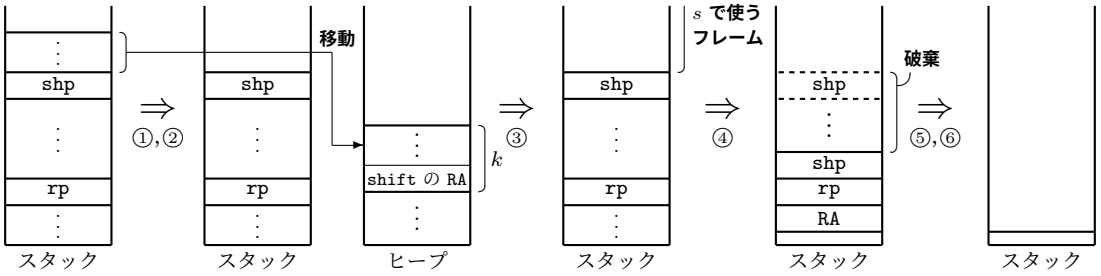

図 12: k がエスケープし，末尾位置での呼び出しがないかも知れない場合のスタックとヒープの変化

3 章の実装から考えると，末尾位置での k の呼び出しがないかも知れない shift を抜けるときには，rp の 1 つ上までのフレームを破棄することになる。しかし，lazy な実装では reset のときに shp の保存もしているため，rp の 2 つ上までのフレームを破棄する。

k，k_{lazy} 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 のクロージャに保存するのは 3 章と同様，shift でヒープに移動したフレームの情報，および shift を呼び出した時点での戻り番地である。また k_{lazy} のクロージャには，現在の shp の値とスタックトップの位置 (その間のフレームが k_{lazy} に対応する)，および shift を呼び出した時点での戻り番地を保存する。実際に k が呼び出されたときには，

① k が呼び出された時点での戻り番地と rp，shp をスタックに保存

② k に対応するフレームをヒープからスタックトップにコピー

③ 保存されていた，shift を呼び出した時点での戻り番地へジャンプ

同じように，k_{lazy} が呼び出されたときには，

① k_{lazy} が呼び出された時点での戻り番地と rp，shp をスタックに保存

② k_{lazy} に対応するフレームをスタックからスタックトップにコピー

③ 保存されていた，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。k と k_{lazy} の違いは，フレームをヒープからコピーするか，スタックからコピーするかだけである。また，末尾位置での k の呼び出しはスタックに必要なフレームが残っているので，関数呼び出しの形ではなく単に戻り番地へジャンプする格好に変換する。

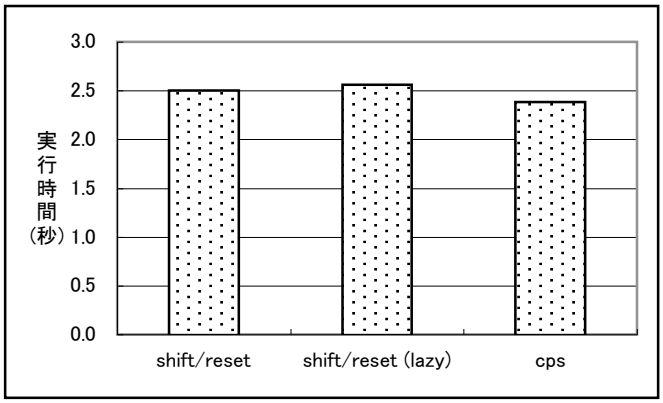

図 13: prefix の実行時間

5 評価

本章では，3 章と 4 章の実装の評価を行う。

- 5.1 節　k がエスケープせず，末尾位置での呼び出しがない例
- 5.2 節　k がエスケープせず，末尾位置での呼び出しがある例と
  k がエスケープし，末尾位置での呼び出しがある例
- 5.3 節　例外処理の例

いずれも実行環境は PowerPC G4 500MHz，メモリは 1.28 GB SDRAM である。また，実行時間の図における “normal” は CPS 形式でなく，`shift/reset` を使わないプログラムの実行時間，“cps” は CPS 形式のプログラムの実行時間，“shift/reset” は `shift/reset` を用いたプログラムの実行時間，“shift/reset (lazy)” は同じプログラムをフレームを lazy にコピーする 4 章の実装で実行した場合の実行時間である。

5.1 prefix

受け取ったリストのプリフィックスのリストを返す関数 `prefix` を考える。例えば `prefix [1; 2; 3]` ⇒ `[[1]; [1; 2]; [1; 2; 3]]` となる。

`shift/reset` を使って書くと，以下のようになる。リストの先頭のほうの要素は多くのプリフィックスに含まれるが，その部分を `shift` を使って継続の形で取り出し，取り出された継続 k を複数回呼び出すことによって，同じコードで複数のプリフィックスに加えることを実現している。

```
let rec visit lst = match lst with
  | [] -> shift (fun k -> [])
  | a :: rest -> a :: shift (fun k -> (k []) :: (reset (k (visit rest)))) in
let prefix lst = reset (visit lst)
```

`shift` の引数の関数に末尾位置での k の呼び出しはなく，k はエスケープしない。この関数と，付録 A の CPS 形式の `prefix` 関数に適当な長さのリストを渡して実行したときの実行速度を図 13 に示す。

“cps” と “shift/reset” の実行時間に大きな差はなく，`shift/reset` を使っても CPS 形式のプログラムと同じくらいの効率を達成出来ている。したがって，効率を犠牲にすることなく，より書きやすい直接形式で

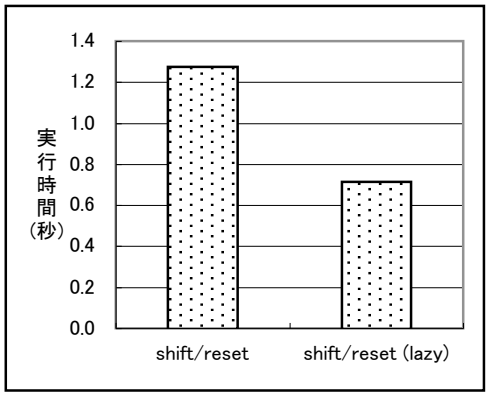

図 14: queen の実行時間

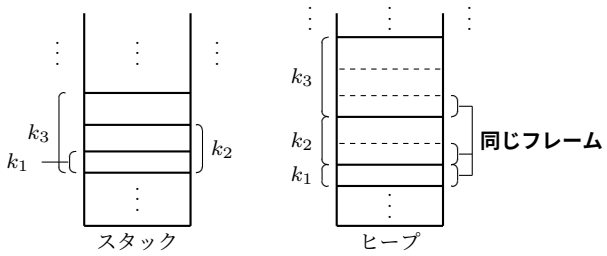

図 15: 末尾位置における k の呼び出しがあるが，k がエスケープする場合のスタックとヒープの状態

書けるようになっている。shift のたびにフレームのコピー回数が 1 回少ないにも関わらず “shift/reset” と “shift/reset (lazy)” の実行時間がほとんど同じなのは，コピーするフレームが小さいこと，また lazy にコピーする場合のオーバーヘッドが多少影響していると考えられる。

5.2 queen

自然数 N を与えられて N-Queen 問題の解のリストを返す関数 queen を考える。N-Queen 問題とは，N×N のマスに N 個の Queen を各 Queen の縦，横，斜めに別の Queen が入らないように配置し，そのすべての可能なパターンを求める問題である。ループを 2 重に回せば解けるが，以下のような関数 choice を使うと，非決定的に選択肢を選ぶ形でプログラムを書くことが出来る (付録 B)。

```
let rec choice j =
  if j = 1 then 1 else shift (fun k -> k j; k (choice (j - 1)))
```

choice は，現在の継続 k を切り取ってきて，k に値 j を渡した後，さらに k に j - 1 以下の値について再帰した値を渡すので，実質的に j から 1 までの数を非決定的に返す関数とみなせる。shift の引数の関数には末尾位置での k の呼び出しがあり，k はエスケープしない。この queen 関数で 11-Queen 問題を解いたプログラムの実行時間と，同じプログラムを lazy な実装で実行した実行時間を図 14 に示す。

普通の実装では choice を 1 回呼び出すたびに，shift の時点と 2 回の k の呼び出しのときの，計 3 回フレームをコピーする。一方，lazy な実装では末尾位置でない k の呼び出し k j のときにのみコピーするので，フレームのコピーは 1 回で済む。2 回のフレームのコピー回数の違いにより，lazy にコピーするほうが速くなったと考えられる。このようにある程度大きなフレームをコピーする場合や，末尾位置での k の呼び出しがある場合は，lazy な手法が効果的である。一方色々な場合を検討するうちに，lazy な手法のほうがヒープの使用量が増える場合があることもわかった。具体的には同じプログラムの choice の shift 式を shift (fun k -> a.(0) <- k; k j; k (choice (j - 1))) と変更する。ただし，これは人為的に k をエスケープさせているのであって，プログラム上，意味がある変更ではない。この場合，shift の引数の関数には末尾位置での k の呼び出しがあり，k はエスケープする。すると，shift でヒープに shp までの部分をコピーし，shp は保存せずに実行を進めるので，スタックとヒープは図 15 のような状態になる。3 章の方法では，フレームは shift のたびにヒープに移動されるので，同じフレームが複数回スタックからヒープへコピーされることはない。しかし lazy な実装ではフレームをスタックに残し，なおかつ末尾位置での呼び出しがある場合はフレームの共有をするので，同じフレームが複数回コピーされることがある。これにより，lazy な実装のほうがヒープを多く消費してしまう。

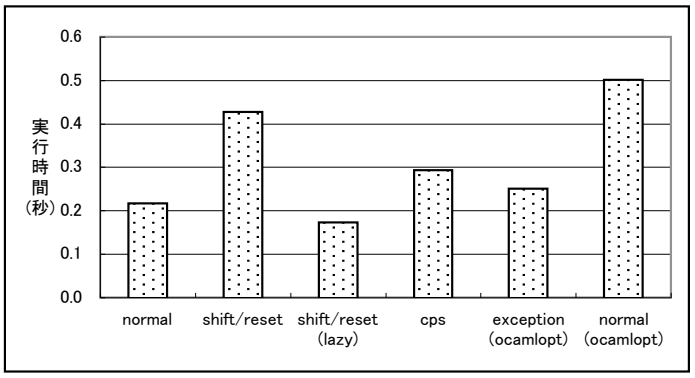

図 16: times の実行時間

同じフレームの共有がなされないためヒープの使用量は増えるが，これによって実行時間がすぐに増えるわけではない。lazy でない実装では k の呼び出し時に独立してフレームをコピーする必要があるのに対し，lazy な実装ではまとめてコピーするので，全体としては速くなる場合が多い。実際，queen の例で実測してみたところ，ヒープ使用量，全体のコピー量ともに普通の実装より多いにも関わらず，実行速度は普通の実装の 80%程度であった。

5.3 times

受け取ったリストの要素の積を求める関数 times を考える。

```
let rec times lst = match lst with       (* normal *)
  | [] -> 1
  | a :: rest -> if a = 0 then 0 else a * times rest
```

要素に 0 がある場合は必ず 0 になるので無条件に 0 を返したいが，上の定義では times [1; 2; 0; 3] ⇒ 1 * 2 * 0 ⇒ 0 と，1 * 2 * 0 = 0 の計算は実行する。例外の機構があるプログラミング言語ならば要素に 0 があった時点で例外を発生させ，呼び出し側で例外を捕らえて 0 を返せばよいが，継続が扱えると同じような処理が出来る。

```
let rec times lst k = match lst with     (* cps *)
  | [] -> k 1
  | a :: rest -> if a = 0 then 0 else times rest (fun k -> k (x * a))
```

上の times 関数はプログラム全体を CPS 形式にして書いたが，shift/reset を使うとはじめの定義 (normal) の then 部分を変更するだけでよい。

```
let rec times lst = match lst with       (* shift/reset *)
  | [] -> 1
  | a :: rest -> if a = 0 then shift (fun k -> 0) else a * times rest
```

これらのプログラムのそれぞれに，末尾に 0 が入っている長いリストを渡すプログラムを実行した実行時間と，normal の times 関数と例外処理を使った times 関数 (付録 C) をそれぞれ ocamlopt でコンパイルした実行時間を図 16 に示す。

CPS 形式のプログラムでは，残りの計算を捨てて全く計算をせずに 0 を返すだけだが，“normal” より遅くなっている。これは，引数として用意するクロージャを作る影響と考えられる。shift/reset を使えばクロージャを用意する必要がないので CPS 形式のものよりも速くなると思われるが，shift の時点で不要なコピーをする分，遅くなってしまう。lazy な実装では最終的に shift を抜ける時点にフレームを破棄するだけで，OCaml で例外処理を用いた場合と同程度の速度が実現出来ている。

6 関連研究

Gasbichler らは control と shift/reset の直接実装法を示し，実際に Scheme48 上で実装した [7] 。それにより，call/cc を用い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できること，および実行効率の向上がもたらされることを示した。Gasbichler らの実装は，Scheme 48 1.0 の仮想機械を記述する PreScheme という低レベルな Scheme で書かれているが，本研究はそれを PowerPC の機械語で直接実装した形となっている。

Ugawa らはスタックベースの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 [14] 。遅延スタックコピー法では，call/cc *proc* という呼び出しの時点ではフレームをコピーせず，*proc* を抜ける時点でアクティブでまだコピーしていないフレームをコピーする。フレームをスタックに残し，必要になるまでコピーしないというアイディアはこの遅延スタックコピー法による。しかし MinCaml は型付き言語なので，本研究では型推論の際にエスケープするかを判定しており，実行時のチェックによるオーバーヘッドはかからない。

7 まとめと今後の課題

本論文では，shift/reset の直接実装について述べた。また実装の改良として，フレームを lazy にコピーする実装方法についても述べた。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。5 章で見たように，shift/reset を使っても CPS 形式のプログラムと同程度の効率を達成しており，本研究は今後の shift/reset の使用を推し進めると期待される。また，ここで shift/reset の直接実装法を詳細に記述したことで，他の言語に shift/reset を導入する際にも助けになると思われる。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，そして直接実装法の正当性の証明などが挙げられる。

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] K. Asai, “Logical Relations for Call-by-value Delimited Continuations”, In *Trends in Functional Programming*, Vol. 6, pp. 63–78, Intellect (2007).

[2] K. Asai, and Y. Kameyama, “Polymorphic Delimited Continuations”, In Z. Shao editor, *5th Asian Symposium on Programming Languages and Systems (APLAS 2007)*, pp. 239–254 (November 2007).

[3] O. Danvy, and A. Filinski, “Abstracting Control”, In *Proceeding of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 151–160 (June 1990).

[4] O. Danvy, and A. Filinski, “Representing Control: A Study of the CPS transformation”, In *Mathematical Structures in Computer Science*, Vol. 2, No. 4, pp. 361–391 (December 1992).

[5] M. Felleisen, “The Theory and Practice of First-Class Prompts”, In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pp. 180–190 (January 1988).

[6] A. Filinski, “Representing Manads”, In *Conference Records of the 21st Annual ACM Symposium on Principles of Programming Languages*, pp. 446–457 (January 1994).

[7] M. Gasbichler, and M. Sperber, “Final Shift for Call/cc: Direct Implementation of Shift and Reset”, In *International Conferrence on Functional Programming (ICFP 2002)*, pp. 271–281 (October 2002).

[8] J. Hannan, “A Type-based Escape Analysis for Functional Languages”, In *Journal of Functional Programming*, Vol. 8, No. 3, pp. 239–273 (May 1998).

[9] Y. Kameyama, O. Kiselyov, and C. Shan. “Shifting the Stage: Staging with Delimited Control”, In *Proceeding of the ACM SIGPLAN Symposium on Partial Evaluation and Program Manipulation (PEPM 2009)*, pp. 111–120 (January 2009).

[10] K. Sasada, “Ruby Continuation”, presented at Continuation Fest, Tokyo (April 2008).

[11] M. Sperber, R. Kent Dybvig, M. Flatt, and A. van Straaten, “Revised^6 report on the algorithmic language Scheme”, http://www.r6rs.org/, 2007.

[12] 住井英二郎, 東京大学理学部情報科学科, 情報科学実験 II, 講義資料, 2002.

[13] E. Sumii, “MinCaml: A Simple and Efficient Compiler for a Minimal Functional Language”. In *ACM SIGPLAN Workshop on Functional and Declarative Programming in Education (FDPE 2005)*, pp. 27–38 (September 2005).

[14] T. Ugawa, N. Minagawa, T. Komiya, M. Yasugi, and T. Yuasa. “Lazy Stack Copying and Stack Copy Sharing for the Efficient Implementation of Continuations”. In *Proceeding of the First Asian Symposium on Programming Languages and Systems (APLAS 2003)*, pp. 410–426 (October 2003).

A prefix

CPS 形式の prefix 関数。

```
let rec prefix lst k = match lst with
  | [] -> []
  | a :: rest -> (k [a]) :: (prefix rest (fun x -> k (a :: x)))
```

B queen

shift/reset を使った queen 関数。ただし，print_solution は解のリストを出力する関数，is_safe は縦，横，斜めに 2 つ以上 Queen が入っていないかチェックする関数。

```
let queen n =
  let rec loop i solution =
    if i = 0 then print_solution solution
    else let j = choice n in
         let solution2 = j :: solution in
           if is_safe solution2 then loop (i - 1) solution2 in
reset (loop n [])
```

C times

例外処理を用いた times 関数。

```
exception Zero
let rec times0 lst = match lst with
  | [] -> 1
  | a :: rest -> if a = 0 then raise Zero else a * times0 rest in
let times lst = try times0 lst with
  | Zero -> 0
```