コンパイラ演習 第6回

2005/11/17

大山 恵弘　　佐藤 秀明

今回の内容

- 実マシンコード生成
  - アセンブリ生成(emit.ml)
  - スタブ、ライブラリとのリンク
- 末尾呼び出し最適化
  - 関数呼び出しからの効率的なリターン (emit.ml)
  - [参考]CPS変換
- 種々の簡単な拡張
  - MinCamlにない機能

実マシンコード生成

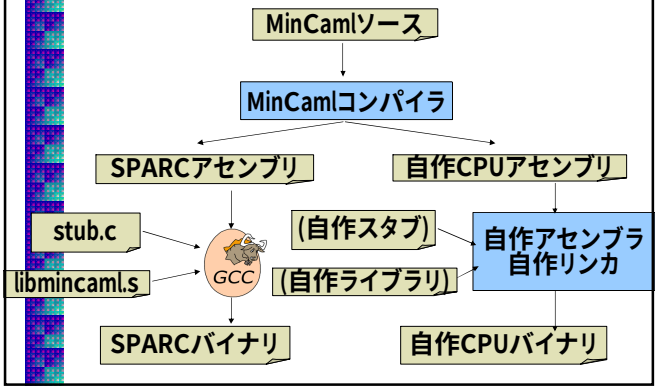
アセンブリ生成

- save/restoreをストア/ロードとして明示化
  - スタックの状態を追跡(stackmap, stackset)
    - if文の合流後は両方のスタックの積集合をとる
- 関数呼び出し規約の明示化
  - 引数を正しいレジスタにセット(shuffle関数)
  - リターンアドレスのsave/restore
  - スタックポインタの管理
- 条件分岐の明示化
  - 分岐用・合流用のラベルを導入
- 他は単純

外部ファイル

- スタブ(stub.c)
  - ヒープとスタックを確保
  - その後MinCamlのエントリポイントをcall
- ライブラリ(libmincaml.s)
  - 外部関数を定義
    - 入出力
    - 配列操作
    - 数値計算
- 自作CPU向けの外部ファイルも必要かも
  - アーキテクチャ次第

ビルドの流れ



末尾呼び出し最適化

末尾呼び出し

```
let rec fact x r =
  if x <= 1 then r
  else fact (x - 1) (r * x)
```

- 関数の最後の処理がcall

単純にコンパイルすると…

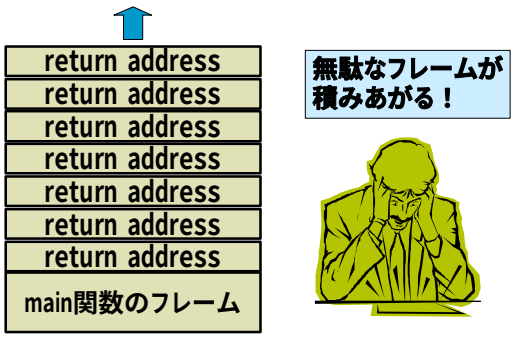
```
let rec fact x r =
  if x <= 1 then r
  else fact (x - 1) (r * x)
```

```
save(Rret)
add Rsp, 4, Rsp
call fact
nop
sub Rsp, 4, Rsp
restore(Rret)
retl
nop
```

返り番地だけからなるフレームを構成

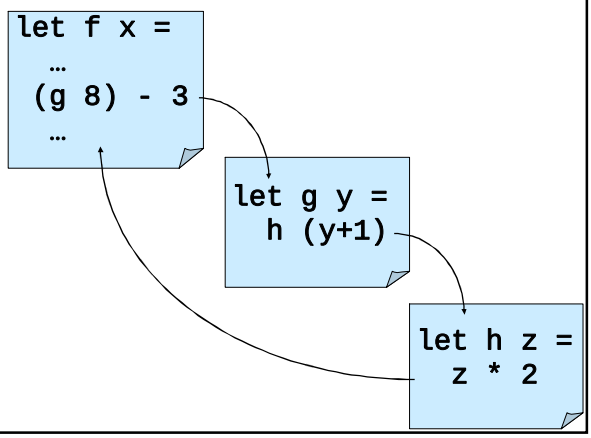
返り番地をpopして直ちにリターン

その結果



何がいけなかったのか? どうすればよいのか?

- 問題の元凶: 末尾呼び出し直後の地点に律義にリターン
  - 時間的に無駄
    - 何もしない地点へわざわざリターン
  - 空間的にも無駄
    - リターンするために、返り番地を余分に退避
- することがないなら、呼び出し元に直接返ればよい!
  - コンパイラが末尾呼び出しを認識



末尾呼び出し最適化

- 末尾呼び出し時の無駄なジャンプ・退避を除去
- cf. 末尾再帰
  - 関数の最後の処理が自分自身の再帰呼び出し
  - 末尾呼び出し最適化により、ループに変換

「ifの直後にreturn」する場合

- 合流する必要がない

```
    cmp x, y
    bg L1
    nop
    ... (then節) ...
    b L2
    nop
L1: ... (else節) ...
L2: retl
    nop
```

```
    cmp x, y
    bg L1
    nop
    ... (then節) ...
    retl
    nop
L1: ... (else節) ...
    retl
    nop
```

「callの直後にreturn」する場合

- callをただのgotoにできる

```
save(Rret)
add Rsp, n, Rsp
call Lf
nop
sub Rsp, n, Rsp
restore(Rret)
retl
nop
```

```
b Lf
nop
```

実装

- 変換中の式が末尾かどうかを管理
  - Tail: 末尾
    - 末尾呼び出し最適化を実行、または
    - 結果を返り値用のレジスタにセットしてリターン
  - NonTail(r): 末尾でない
    - 結果をレジスタrにセット

[参考]CPS変換

- すべてのcallやifをtailにしてしまう!
  - nontailな関数適用/条件分岐の継続を生成
    - 「その後にすること」をクロージャで表現
  - すべての関数定義/関数適用に仮引数/実引数として継続を追加
  - 関数の戻り値は継続に渡す
- α変換およびA正規化の完了したK正規形に対して行うと簡単

CPS変換のイメージ

```
let rec f x =
  let a = p 100 in
  let b = q 200 in
  x + a + b + r 300
```

この部分を実行する関数（クロージャ）を新たに導入。pの引数として渡す

pは実行が終わったら、引数にもらった関数（クロージャ）を呼び出すことにより「復帰」する

CPS変換の利点/欠点

- 利点:以降の処理が容易
  - 関数呼び出し時のsave/restoreが不要
  - 「リターンアドレス」の概念が不要
  - スタックも不要
- 欠点:クロージャが頻繁に生成/適用される
  - 効率的なヒープ管理の必要性
    - inter-proceduralなレジスタ割り当て
    - エスケープ解析
    - generational garbage collection

種々の簡単な拡張

レコード、Variant

- レコード: フィールドがアルファベット順に並んだtupleとみなす
  { foo = 3; bar = 7 }
  = { bar = 7; foo = 3 }
  ⇒ (7, 3)
- Variant: コンストラクタを整数で表し、それを第1要素とするtupleにする
  type α list = Nil | Cons of α * α list として
  Nil ⇒ (0)
  Cons(x, y) ⇒ (1, x, y)

λ抽象、部分適用

- λ抽象: let recに置換
  fun x → M ⇒ let rec f x = M in f
  (fはfreshな変数名)
- 部分適用: let recと完全適用に置換
  たとえば let rec f x y = x - y なら
  f 3 ⇒ let rec g y = f 3 y in g
  (gはfreshな変数名)
  - 関数の型情報が必要

共通課題(1/3)

- 以下のプログラムはどのようなアセンブリにコンパイルされるか、末尾呼び出し最適化をしない場合とする場合、それぞれについて説明せよ。
  - ヒント: 末尾呼び出し最適化をする場合、手続き型言語でループを用いて書いたgcdと同じアセンブリになる(はず)

```
let rec gcd m n =
  if m <= 0 then n else
  if m <= n then gcd m (n − m) else
  gcd n (m − n) in
print_int (gcd 21600 337500)
```

共通課題(2/3)

- 以下のプログラムを手動でCPS変換せよ。
  - K正規化はしてもしなくてもよい
  - 余裕があれば、CPS変換した場合としなかった場合でどのようなアセンブリにコンパイルされるか、両者を比較してみよう

```
let rec ack x y =
  if x <= 0 then y - -1 else
  if y <= 0 then ack (x - 1) 1 else
  ack (x - 1) (ack x (y - 1)) in
print_int (ack 3 10)
```

共通課題(3/3)

- 「種々の簡単な拡張」(の一部)を用いるMLプログラムを書け。それを既存のMLコンパイラがどうコンパイルするか調べ、解説せよ。
  - 同程度以上に複雑な他のプリミティブについて調べてもよい

課題の提出先と締め切り

- 提出先: compiler-enshu@yl.is.s.u-tokyo.ac.jp
- 締め切り: 2週間後 (12/1) の午後1時
- Subject: report 6 <学籍番号> <アカウント>
- 本文にも氏名と学籍番号を明記のこと

課題の提出についての注意

- プログラムだけでなく、説明・考察・感想なども書くこと
- 基本的にはメールの本文に解答を記述
- 多くのソースを送る必要がある課題では、ソースをtarファイルなどに固めてメールに添付のこと

今後の予定

- 次回からは応用編です
  - 11/24: 休講
  - 12/1: 休講
  - 12/8: Garbage Collection
  - 12/15: オブジェクト
  - 12/22: Polymorphism
  - 1/12: 例外処理
  - 1/19: エスケープ解析
  - 1/26: リージョン推論?

$\mathcal{S}$: `SparcAsm.prog` $\rightarrow$ `string`

$$
\begin{array}{lcl}
\mathcal{S}((\{D_1, \ldots, D_n\}, E)) & = & \texttt{.section ".text"} \\
 & & \mathcal{S}(D_1) \\
 & & \ldots \\
 & & \mathcal{S}(D_n) \\
 & & \texttt{.global min\_caml\_start} \\
 & & \texttt{min\_caml\_start:} \\
 & & \texttt{save \%sp, -112, \%sp} \\
 & & \mathcal{S}(E, \texttt{\%g0}) \\
 & & \texttt{ret} \\
 & & \texttt{restore}
\end{array}
$$

$\mathcal{S}$: `SparcAsm.fundef` $\rightarrow$ `string`

$$
\begin{array}{lcl}
\mathcal{S}(\mathtt{L}_x(y_1, \ldots, y_n) = E) & = & x\texttt{:} \\
 & & \mathcal{S}(E, \mathtt{R}_0) \\
 & & \texttt{retl} \\
 & & \texttt{nop}
\end{array}
$$

$\mathcal{S}$: `SparcAsm.t` $\times$ `Id.t` $\rightarrow$ `string`

$$
\begin{array}{lcl}
\mathcal{S}((x \leftarrow e; E), z_{\texttt{dest}}) & = & \mathcal{S}(e, x); \mathcal{S}(E, z_{\texttt{dest}}) \\
\mathcal{S}(e, z_{\texttt{dest}}) & = & \mathcal{S}(e, z_{\texttt{dest}})
\end{array}
$$

図 1: 単純なアセンブリ生成 $\mathcal{S}(P)$, $\mathcal{S}(D)$ および $\mathcal{S}(E, z_{\texttt{dest}})$

$\mathcal{S}$: SparcAsm.exp $\times$ Id.t $\to$ string

$\mathcal{S}(c, z_{\texttt{dest}})$	=	set $c, z_{\texttt{dest}}$
$\mathcal{S}(\texttt{L}_x, z_{\texttt{dest}})$	=	set $\texttt{L}_x, z_{\texttt{dest}}$
$\mathcal{S}(op(x_1, \ldots, x_n), z_{\texttt{dest}})$	=	$op\ x_1, \ldots, x_n, z_{\texttt{dest}}$
$\mathcal{S}(\texttt{if}\ x = y\ \texttt{then}\ E_1\ \texttt{else}\ E_2, z_{\texttt{dest}})$	=	cmp x, y
		bne b_1
		nop
		$\mathcal{S}(E_1, z_{\texttt{dest}})$
		b b_2
		nop
		b_1:
		$\mathcal{S}(E_2, z_{\texttt{dest}})$
		b_2:
$\mathcal{S}(\texttt{if}\ x \leq y\ \texttt{then}\ E_1\ \texttt{else}\ E_2, z_{\texttt{dest}})$	=	同様
$\mathcal{S}(x, z_{\texttt{dest}})$	=	mov $x, z_{\texttt{dest}}$
$\mathcal{S}(apply\_closure(x, y_1, \ldots, y_n), z_{\texttt{dest}})$	=	$shuffle((x, y_1, \ldots, y_n), (\texttt{R}_0, \texttt{R}_1, \ldots, \texttt{R}_n))$
		st $\texttt{R}_{\texttt{ra}}, [\texttt{R}_{\texttt{st}} + 4\#\varepsilon]$
		ld $[\texttt{R}_0], \texttt{R}_{n+1}$
		call $\texttt{R}_{n+1}$
		add $\texttt{R}_{\texttt{st}}, 4(\#\varepsilon + 1), \texttt{R}_{\texttt{st}}$! *delay slot*
		sub $\texttt{R}_{\texttt{st}}, 4(\#\varepsilon + 1), \texttt{R}_{\texttt{st}}$
		ld $[\texttt{R}_{\texttt{st}} + 4\#\varepsilon], \texttt{R}_{\texttt{ra}}$
		mov $\texttt{R}_0, z_{\texttt{dest}}$
$\mathcal{S}(apply\_direct(\texttt{L}_x, y_1, \ldots, y_n), z_{\texttt{dest}})$	=	$shuffle((y_1, \ldots, y_n), (\texttt{R}_1, \ldots, \texttt{R}_n))$
		st $\texttt{R}_{\texttt{ra}}, [\texttt{R}_{\texttt{st}} + 4\#\varepsilon]$
		call x
		add $\texttt{R}_{\texttt{st}}, 4(\#\varepsilon + 1), \texttt{R}_{\texttt{st}}$! *delay slot*
		sub $\texttt{R}_{\texttt{st}}, 4(\#\varepsilon + 1), \texttt{R}_{\texttt{st}}$
		ld $[\texttt{R}_{\texttt{st}} + 4\#\varepsilon], \texttt{R}_{\texttt{ra}}$
		mov $\texttt{R}_0, z_{\texttt{dest}}$
$\mathcal{S}(x.(y), z_{\texttt{dest}})$	=	ld $[x + y], z_{\texttt{dest}}$
$\mathcal{S}(x.(y) \leftarrow z, z_{\texttt{dest}})$	=	st $z, [x + y]$
$\mathcal{S}(\texttt{save}(x, y), z_{\texttt{dest}})$	=	もし $y \notin dom(\varepsilon)$ なら $\varepsilon \leftarrow (\varepsilon, y \mapsto 4\#\varepsilon)$ として
		st $x, [\texttt{R}_{\texttt{st}} + \varepsilon(y)]$
$\mathcal{S}(\texttt{restore}(y), z_{\texttt{dest}})$	=	ld $[\texttt{R}_{\texttt{st}} + \varepsilon(y)], z_{\texttt{dest}}$

図 2: 単純なアセンブリ生成 $\mathcal{S}(e, z_{\texttt{dest}})$。$\varepsilon$ はスタック位置を記憶するグローバル変数。$\#\varepsilon$ は ε の要素の個数。$shuffle((x_1, \ldots, x_n), (r_1, \ldots, r_n))$ は $x_1, \ldots, x_n$ を $r_1, \ldots, r_n$ に適切な順序で移動する命令。

$\mathcal{S} : \texttt{S.t} \to \texttt{SparcAsm.t} \times \texttt{Id.t} \to \texttt{S.t} \times \texttt{string}$

$$
\begin{array}{lcll}
\mathcal{S}_s((x \leftarrow e; E), z_{\texttt{dest}}) & = & \mathcal{S}_s(e, x) = (s', S), & \\
 & & \mathcal{S}_{s'}(E, z_{\texttt{dest}}) = (s'', S') \text{ として} & \\
 & & (s'', SS') & \\
\mathcal{S}_s(e, z_{\texttt{dest}}) & = & \mathcal{S}_s(e, z_{\texttt{dest}}) &
\end{array}
$$

$\mathcal{S} : \texttt{S.t} \to \texttt{SparcAsm.exp} \times \texttt{Id.t} \to \texttt{S.t} \times \texttt{string}$

$$
\begin{array}{lcll}
\mathcal{S}_s(\texttt{if } x = y \texttt{ then } E_1 \texttt{ else } E_2, z_{\texttt{dest}}) & = & \mathcal{S}_s(E_1, z_{\texttt{dest}}) = (s_1, S_1), & \\
 & & \mathcal{S}_s(E_2, z_{\texttt{dest}}) = (s_2, S_2) \text{ として} & \\
 & & (s_1 \cap s_2, & \\
 & & \texttt{cmp } x, y & \\
 & & \texttt{bne } b_1 & \\
 & & \texttt{nop} & \\
 & & S_1 & \\
 & & \texttt{b } b_2 & \\
 & & \texttt{nop} & \\
 & & b_1\texttt{:} & \\
 & & S_2 & \\
 & & b_2\texttt{:}) & \\
\mathcal{S}_s(\texttt{if } x \leq y \texttt{ then } E_1 \texttt{ else } E_2, z_{\texttt{dest}}) & = & \text{同様} & \\
\mathcal{S}_s(\texttt{save}(x, y), z_{\texttt{dest}}) & = & (s, \texttt{nop}) & y \in s \text{ の場合} \\
\mathcal{S}_s(\texttt{save}(x, y), z_{\texttt{dest}}) & = & \text{もし } y \notin dom(\varepsilon) \text{ なら } \varepsilon \leftarrow (\varepsilon, y \mapsto 4\#\varepsilon) \text{ として} & \\
 & & (s \cup \{y\}, \texttt{st } x, [\texttt{R}_{\texttt{st}} + \varepsilon(y)]) & y \notin s \text{ の場合} \\
\mathcal{S}_s(e, z_{\texttt{dest}}) & = & (s, \text{以前と同様}) & \text{上述以外の場合}
\end{array}
$$

図 3: 無駄な save を省略するアセンブリ生成 $\mathcal{S}_s(E, z_{\texttt{dest}})$ および $\mathcal{S}_s(e, z_{\texttt{dest}})$。$s$ はすでに save された変数の名前の集合。以前の $\mathcal{S}(E, z_{\texttt{dest}})$ は $\mathcal{S}_\emptyset(E, z_{\texttt{dest}}) = (s, S)$ として S の略記とする。

$\mathcal{S}$: SparcAsm.fundef $\rightarrow$ string

$\mathcal{S}(\mathtt{L}_x(y_1, \ldots, y_n) = E)$ = $\mathcal{S}_\emptyset(E, \mathtt{tail}) = (s, S)$ として
x:
S

$\mathcal{S}$: S.t $\rightarrow$ SparcAsm.exp $\times$ Id.t $\rightarrow$ S.t $\times$ string

$\mathcal{S}_s(\mathtt{if}\ x = y\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2, \mathtt{tail})$ = $\mathcal{S}_s(E_1, \mathtt{tail}) = (s_1, S_1)$,
$\mathcal{S}_s(E_2, \mathtt{tail}) = (s_2, S_2)$ として
$(\emptyset,$
cmp x, y
bne b
nop
S_1
b:
$S_2)$

$\mathcal{S}_s(\mathtt{if}\ x \leq y\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2, \mathtt{tail})$ = 同様

$\mathcal{S}_s(\mathit{apply\_closure}(x, y_1, \ldots, y_n), \mathtt{tail})$ = $(\emptyset,$
$\mathit{shuffle}((x, y_1, \ldots, y_n), (\mathtt{R}_0, \mathtt{R}_1, \ldots, \mathtt{R}_n))$
ld $[\mathtt{R}_0], \mathtt{R}_{n+1}$
jmp $\mathtt{R}_{n+1}$
nop)

$\mathcal{S}_s(\mathit{apply\_direct}(\mathtt{L}_x, y_1, \ldots, y_n), \mathtt{tail})$ = $(\emptyset,$
$\mathit{shuffle}((y_1, \ldots, y_n), (\mathtt{R}_1, \ldots, \mathtt{R}_n))$
b x
nop)

$\mathcal{S}_s(e, \mathtt{tail})$ = $\mathcal{S}_s(e, \mathtt{R}_0) = (s', S)$ として
$(\emptyset,$
S
retl
nop) 上述以外の場合

図 4: 末尾呼び出し最適化をするアセンブリ生成 $\mathcal{S}_s(D)$ および $\mathcal{S}_s(e, z_{\mathtt{dest}})$。$z_{\mathtt{dest}} = \mathtt{tail}$ の場合が末尾。

$$
\begin{array}{l}
e\ ::= \\
\quad c \\
\quad op(x_1, \ldots, x_n) \\
\quad \texttt{if}\ x = y\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
\quad \texttt{if}\ x \leq y\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
\quad \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \\
\quad x \\
\quad \texttt{let rec}\ x\ y_1\ \ldots\ y_n = e_1\ \texttt{in}\ e_2 \\
\quad x\ y_1\ \ldots\ y_n \\
\quad (x_1, \ldots, x_n) \\
\quad \texttt{let}\ (x_1, \ldots, x_n) = y\ \texttt{in}\ e \\
\quad x.(y) \\
\quad x.(y) \leftarrow z
\end{array}
$$

図 5: MinCaml の K 正規形（外部配列・外部関数適用は省略）

$\mathcal{C} : \texttt{Id.t} \rightarrow \texttt{KNormal.t} \rightarrow \texttt{KNormal.t}$

$$
\begin{array}{lcl}
\mathcal{C}_k(\texttt{if}\ x \leq y\ \texttt{then}\ e_1\ \texttt{else}\ e_2) & = & \texttt{if}\ x \leq y\ \texttt{then}\ \mathcal{C}_k(e_1)\ \texttt{else}\ \mathcal{C}_k(e_2) \\
\mathcal{C}_k(\texttt{if}\ x = y\ \texttt{then}\ e_1\ \texttt{else}\ e_2) & = & \texttt{if}\ x = y\ \texttt{then}\ \mathcal{C}_k(e_1)\ \texttt{else}\ \mathcal{C}_k(e_2) \\
\mathcal{C}_k(\texttt{let}\ (x_1, \ldots, x_n) = y\ \texttt{in}\ e) & = & \texttt{let}\ (x_1, \ldots, x_n) = y\ \texttt{in}\ \mathcal{C}_k(e) \\
\\
\mathcal{C}_k(\texttt{let}\ x = (\texttt{if}\ y \leq z\ \texttt{then}\ e_1\ \texttt{else}\ e_2)\ \texttt{in}\ e_3) & = & \texttt{let rec}\ k'\ x = \mathcal{C}_k(e_3)\ \texttt{in} \\
 & & \texttt{if}\ y \leq z\ \texttt{then}\ \mathcal{C}_{k'}(e_1)\ \texttt{else}\ \mathcal{C}_{k'}(e_2) \quad (k' \text{ は fresh}) \\
\mathcal{C}_k(\texttt{let}\ x = (\texttt{if}\ y = z\ \texttt{then}\ e_1\ \texttt{else}\ e_2)\ \texttt{in}\ e_3) & = & \texttt{let rec}\ k'\ x = \mathcal{C}_k(e_3)\ \texttt{in} \\
 & & \texttt{if}\ y = z\ \texttt{then}\ \mathcal{C}_{k'}(e_1)\ \texttt{else}\ \mathcal{C}_{k'}(e_2) \quad (k' \text{ は fresh}) \\
\mathcal{C}_k(\texttt{let}\ x = y\ z_1\ \ldots\ z_n\ \texttt{in}\ e) & = & \texttt{let rec}\ k'\ x = \mathcal{C}_k(e)\ \texttt{in}\ y\ k'\ z_1\ \ldots\ z_n \\
 & & (k' \text{ は fresh}) \\
\mathcal{C}_k(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) & = & \texttt{let}\ x = e_1\ \texttt{in}\ \mathcal{C}_k(e_2) \quad (\text{上述以外の場合}) \\
\\
\mathcal{C}_k(\texttt{let rec}\ f\ x_1\ \ldots\ x_n = e_1\ \texttt{in}\ e_2) & = & \texttt{let rec}\ f\ c\ x_1\ \ldots\ x_n = \mathcal{C}_c(e_1)\ \texttt{in}\ \mathcal{C}_k(e_2) \\
 & & (c \text{ は fresh}) \\
\mathcal{C}_k(x\ y_1\ \ldots\ y_n) & = & x\ k\ y_1\ \ldots\ y_n \\
\\
\mathcal{C}_k(e) & = & \texttt{let}\ x = e\ \texttt{in}\ k\ x \quad (\text{上述以外の場合。} x \text{ は fresh})
\end{array}
$$

図 6: [参考] α 変換および A 正規化の完了した K 正規形に対する CPS 変換 $\mathcal{C}_k(e)$。ただし k は e の継続。継続がないときは、$\texttt{let rec}\ k\ x = x\ \texttt{in}\ \mathcal{C}_k(e)$ のように恒等関数とする（メインルーチンなど）

```
open SparcAsm

external gethi : float -> int32 = "gethi"
external getlo : float -> int32 = "getlo"

let stackset = ref S.empty (* すでにSaveされた変数の集合 (caml2html: emit_stackset) *)
let stackmap = ref [] (* Saveされた変数の、スタックにおける位置 (caml2html: emit_stackmap) *)
let save x =
  stackset := S.add x !stackset;
  if not (List.mem x !stackmap) then
    stackmap := !stackmap @ [x]
let savef x =
  stackset := S.add x !stackset;
  if not (List.mem x !stackmap) then
    ( let pad =
      if List.length !stackmap mod 2 = 0 then [] else [Id.gentmp Type.Int] in
    stackmap := !stackmap @ pad @ [x; x])
let locate x =
  let rec loc = function
    | [] -> []
    | y :: zs when x = y -> 0 :: List.map succ (loc zs)
    | y :: zs -> List.map succ (loc zs) in
  loc !stackmap
let offset x = 4 * List.hd (locate x)
let stacksize () = align ((List.length !stackmap + 1) * 4)

let pp_id_or_imm = function
  | V(x) -> x
  | C(i) -> string_of_int i

(* 関数呼び出しのために引数を並べ替える(register shuffling) (caml2html: emit_shuffle) *)
let rec shuffle sw xys =
  (* remove identical moves *)
  let _, xys = List.partition ( fun (x, y) -> x = y) xys in
  (* find acyclic moves *)
  match List.partition ( fun (_, y) -> List.mem_assoc y xys) xys with
  | [], [] -> []
  | (x, y) :: xys, [] -> (* no acyclic moves; resolve a cyclic move *)
      (y, sw) :: (x, y) :: shuffle sw (List.map
                                         ( function
                                           | (y', z) when y = y' -> (sw, z)
                                           | yz -> yz)
                                        xys)
  | xys, acyc -> acyc @ shuffle sw xys

type dest = Tail | NonTail of Id.t (* 末尾かどうかを表すデータ型 (caml2html: emit_dest) *)
let rec g oc = function (* 命令列のアセンブリ生成 (caml2html: emit_g) *)
  | dest, Ans(exp) -> g' oc (dest, exp)
  | dest, Let((x, t), exp, e) ->
      g' oc (NonTail(x), exp);
      g oc (dest, e)
  | _, Forget _ -> assert false
and g' oc = function (* 各命令のアセンブリ生成 (caml2html: emit_gprime) *)
  (* 末尾でなかったら計算結果をdestにセット (caml2html: emit_nontail) *)
  | NonTail(_), Nop -> ()
  | NonTail(x), Set(i) -> Printf.fprintf oc "\tset\t%d, %s\n" i x
  | NonTail(x), SetL(Id.L(y)) -> Printf.fprintf oc "\tset\t%s, %s\n" y x
  | NonTail(x), Mov(y) when x = y -> ()
  | NonTail(x), Mov(y) -> Printf.fprintf oc "\tmov\t%s, %s\n" y x
```



```
  | NonTail(x), Neg(y) -> Printf.fprintf oc "\tneg\t%s, %s\n" y x
  | NonTail(x), Add(y, z') -> Printf.fprintf oc "\tadd\t%s, %s, %s\n" y (pp_id_or_imm z') x
  | NonTail(x), Sub(y, z') -> Printf.fprintf oc "\tsub\t%s, %s, %s\n" y (pp_id_or_imm z') x
  | NonTail(x), SLL(y, z') -> Printf.fprintf oc "\tsll\t%s, %s, %s\n" y (pp_id_or_imm z') x
  | NonTail(x), Ld(y, z') -> Printf.fprintf oc "\tld\t[%s + %s], %s\n" y (pp_id_or_imm z') x
  | NonTail(_), St(x, y, z') -> Printf.fprintf oc "\tst\t%s, [%s + %s]\n" x y (pp_id_or_imm z')
  | NonTail(x), FMovD(y) when x = y -> ()
  | NonTail(x), FMovD(y) ->
      Printf.fprintf oc "\tfmovs\t%s, %s\n" y x;
      Printf.fprintf oc "\tfmovs\t%s, %s\n" (co_freg y) (co_freg x)
  | NonTail(x), FNegD(y) ->
      Printf.fprintf oc "\tfnegs\t%s, %s\n" y x;
      if x <> y then Printf.fprintf oc "\tfmovs\t%s, %s\n" (co_freg y) (co_freg x)
  | NonTail(x), FAddD(y, z) -> Printf.fprintf oc "\tfaddd\t%s, %s, %s\n" y z x
  | NonTail(x), FSubD(y, z) -> Printf.fprintf oc "\tfsubd\t%s, %s, %s\n" y z x
  | NonTail(x), FMulD(y, z) -> Printf.fprintf oc "\tfmuld\t%s, %s, %s\n" y z x
  | NonTail(x), FDivD(y, z) -> Printf.fprintf oc "\tfdivd\t%s, %s, %s\n" y z x
  | NonTail(x), LdDF(y, z') -> Printf.fprintf oc "\tldd\t[%s + %s], %s\n" y (pp_id_or_imm z') x
  | NonTail(_), StDF(x, y, z') -> Printf.fprintf oc "\tstd\t%s, [%s + %s]\n" x y (pp_id_or_imm z')
  | NonTail(_), Comment(s) -> Printf.fprintf oc "\t! %s\n" s
  (* 退避の仮想命令の実装 (caml2html: emit_save) *)
  | NonTail(_), Save(x, y) when List.mem x allregs && not (S.mem y !stackset) ->
      save y;
      Printf.fprintf oc "\tst\t%s, [%s + %d]\n" x reg_sp (offset y)
  | NonTail(_), Save(x, y) when List.mem x allfregs && not (S.mem y !stackset) ->
      savef y;
      Printf.fprintf oc "\tstd\t%s, [%s + %d]\n" x reg_sp (offset y)
  | NonTail(_), Save(x, y) -> assert (S.mem y !stackset); ()
  (* 復帰の仮想命令の実装 (caml2html: emit_restore) *)
  | NonTail(x), Restore(y) when List.mem x allregs ->
      Printf.fprintf oc "\tld\t[%s + %d], %s\n" reg_sp (offset y) x
  | NonTail(x), Restore(y) ->
      assert (List.mem x allfregs);
      Printf.fprintf oc "\tldd\t[%s + %d], %s\n" reg_sp (offset y) x
  (* 末尾だったら計算結果を第一レジスタにセットしてret (caml2html: emit_tailret) *)
  | Tail, (Nop | St _ | StDF _ | Comment _ | Save _ as exp) ->
      g' oc (NonTail(Id.gentmp Type.Unit), exp);
      Printf.fprintf oc "\tretl\n";
      Printf.fprintf oc "\tnop\n"
  | Tail, (Set _ | SetL _ | Mov _ | Neg _ | Add _ | Sub _ | SLL _ | Ld _ as exp) ->
      g' oc (NonTail(regs.(0)), exp);
      Printf.fprintf oc "\tretl\n";
      Printf.fprintf oc "\tnop\n"
  | Tail, (FMovD _ | FNegD _ | FAddD _ | FSubD _ | FMulD _ | FDivD _ | LdDF _ as exp) ->
      g' oc (NonTail(fregs.(0)), exp);
      Printf.fprintf oc "\tretl\n";
      Printf.fprintf oc "\tnop\n"
  | Tail, (Restore(x) as exp) ->
      ( match locate x with
      | [i] -> g' oc (NonTail(regs.(0)), exp)
      | [i; j] when i + 1 = j -> g' oc (NonTail(fregs.(0)), exp)
      | _ -> assert false);
```

```
    Printf.fprintf oc "\tretl\n";
    Printf.fprintf oc "\tnop\n"
| Tail, IfEq(x, y', e1, e2) ->
    Printf.fprintf oc "\tcmp\t%s, %s\n" x (pp_id_or_imm y');
    g'_tail_if oc e1 e2 "be" "bne"
| Tail, IfLE(x, y', e1, e2) ->
    Printf.fprintf oc "\tcmp\t%s, %s\n" x (pp_id_or_imm y');
    g'_tail_if oc e1 e2 "ble" "bg"
| Tail, IfGE(x, y', e1, e2) ->
    Printf.fprintf oc "\tcmp\t%s, %s\n" x (pp_id_or_imm y');
    g'_tail_if oc e1 e2 "bge" "bl"
| Tail, IfFEq(x, y, e1, e2) ->
    Printf.fprintf oc "\tfcmpd\t%s, %s\n" x y;
    Printf.fprintf oc "\tnop\n";
    g'_tail_if oc e1 e2 "fbe" "fbne"
| Tail, IfFLE(x, y, e1, e2) ->
    Printf.fprintf oc "\tfcmpd\t%s, %s\n" x y;
    Printf.fprintf oc "\tnop\n";
    g'_tail_if oc e1 e2 "fble" "fbg"
| NonTail(z), IfEq(x, y', e1, e2) ->
    Printf.fprintf oc "\tcmp\t%s, %s\n" x (pp_id_or_imm y');
    g'_non_tail_if oc (NonTail(z)) e1 e2 "be" "bne"
| NonTail(z), IfLE(x, y', e1, e2) ->
    Printf.fprintf oc "\tcmp\t%s, %s\n" x (pp_id_or_imm y');
    g'_non_tail_if oc (NonTail(z)) e1 e2 "ble" "bg"
| NonTail(z), IfGE(x, y', e1, e2) ->
    Printf.fprintf oc "\tcmp\t%s, %s\n" x (pp_id_or_imm y');
    g'_non_tail_if oc (NonTail(z)) e1 e2 "bge" "bl"
| NonTail(z), IfFEq(x, y, e1, e2) ->
    Printf.fprintf oc "\tfcmpd\t%s, %s\n" x y;
    Printf.fprintf oc "\tnop\n";
    g'_non_tail_if oc (NonTail(z)) e1 e2 "fbe" "fbne"
| NonTail(z), IfFLE(x, y, e1, e2) ->
    Printf.fprintf oc "\tfcmpd\t%s, %s\n" x y;
    Printf.fprintf oc "\tnop\n";
    g'_non_tail_if oc (NonTail(z)) e1 e2 "fble" "fbg"
(* 関数呼び出しの仮想命令の実装 (caml2html: emit_call) *)
| Tail, CallCls(x, ys, zs) -> (* 末尾呼び出し (caml2html: emit_tailcall) *)
    g'_args oc [(x, reg_cl)] ys zs;
    Printf.fprintf oc "\tld\t[%s + 0], %s\n" reg_cl reg_sw;
    Printf.fprintf oc "\tjmp\t%s\n" reg_sw;
    Printf.fprintf oc "\tnop\n"
| Tail, CallDir(Id.L(x), ys, zs) -> (* 末尾呼び出し *)
    g'_args oc [] ys zs;
    Printf.fprintf oc "\tb\t%s\n" x;
    Printf.fprintf oc "\tnop\n"
| NonTail(a), CallCls(x, ys, zs) ->
    g'_args oc [(x, reg_cl)] ys zs;
    let ss = stacksize () in
    Printf.fprintf oc "\tst\t%s, [%s + %d]\n" reg_ra reg_sp (ss - 4);
    Printf.fprintf oc "\tld\t[%s + 0], %s\n" reg_cl reg_sw;
    Printf.fprintf oc "\tcall\t%s\n" reg_sw;
    Printf.fprintf oc "\tadd\t%s, %d, %s\t! delay slot\n" reg_sp ss reg_sp;
    Printf.fprintf oc "\tsub\t%s, %d, %s\n" reg_sp ss reg_sp;
    Printf.fprintf oc "\tld\t[%s + %d], %s\n" reg_sp (ss - 4) reg_ra;
    if List.mem a allregs && a <> regs.(0) then
      Printf.fprintf oc "\tmov\t%s, %s\n" regs.(0) a
    else if List.mem a allfregs && a <> fregs.(0) then
      (Printf.fprintf oc "\tfmovs\t%s, %s\n" fregs.(0) a;
       Printf.fprintf oc "\tfmovs\t%s, %s\n" (co_freg fregs.(0)) (co_freg a))
| NonTail(a), CallDir(Id.L(x), ys, zs) ->
    g'_args oc [] ys zs;
```



```
    let ss = stacksize () in
    Printf.fprintf oc "\tst\t%s, [%s + %d]\n" reg_ra reg_sp (ss - 4);
    Printf.fprintf oc "\tcall\t%s\n" x;
    Printf.fprintf oc "\tadd\t%s, %d, %s\t! delay slot\n" reg_sp ss reg_sp;
    Printf.fprintf oc "\tsub\t%s, %d, %s\n" reg_sp ss reg_sp;
    Printf.fprintf oc "\tld\t[%s + %d], %s\n" reg_sp (ss - 4) reg_ra;
    if List.mem a allregs && a <> regs.(0) then
      Printf.fprintf oc "\tmov\t%s, %s\n" regs.(0) a
    else if List.mem a allfregs && a <> fregs.(0) then
      (Printf.fprintf oc "\tfmovs\t%s, %s\n" fregs.(0) a;
       Printf.fprintf oc "\tfmovs\t%s, %s\n" (co_freg fregs.(0)) (co_freg a))
and g'_tail_if oc e1 e2 b bn =
  let b_else = Id.genid (b ^ "_else") in
  Printf.fprintf oc "\t%s\t%s\n" bn b_else;
  Printf.fprintf oc "\tnop\n";
  let stackset_back = !stackset in
  g oc (Tail, e1);
  Printf.fprintf oc "%s:\n" b_else;
  stackset := stackset_back;
  g oc (Tail, e2)
and g'_non_tail_if oc dest e1 e2 b bn =
  let b_else = Id.genid (b ^ "_else") in
  let b_cont = Id.genid (b ^ "_cont") in
  Printf.fprintf oc "\t%s\t%s\n" bn b_else;
  Printf.fprintf oc "\tnop\n";
  let stackset_back = !stackset in
  g oc (dest, e1);
  let stackset1 = !stackset in
  Printf.fprintf oc "\tb\t%s\n" b_cont;
  Printf.fprintf oc "\tnop\n";
  Printf.fprintf oc "%s:\n" b_else;
  stackset := stackset_back;
  g oc (dest, e2);
  Printf.fprintf oc "%s:\n" b_cont;
  let stackset2 = !stackset in
  stackset := S.inter stackset1 stackset2
and g'_args oc x_reg_cl ys zs =
  let (i, yrs) =
    List.fold_left
      ( fun (i, yrs) y -> (i + 1, (y, regs.(i)) :: yrs))
      (0, x_reg_cl)
      ys in
  List.iter
    ( fun (y, r) -> Printf.fprintf oc "\tmov\t%s, %s\n" y r)
    (shuffle reg_sw yrs);
  let (d, zfrs) =
    List.fold_left
      ( fun (d, zfrs) z -> (d + 1, (z, fregs.(d)) :: zfrs))
      (0, [])
      zs in
  List.iter
    ( fun (z, fr) ->
      Printf.fprintf oc "\tfmovs\t%s, %s\n" z fr;
      Printf.fprintf oc "\tfmovs\t%s, %s\n" (co_freg z) (co_freg fr))
    (shuffle reg_fsw zfrs)

let h oc { name = Id.L(x); args = _; fargs = _; body = e; ret = _ } =
  Printf.fprintf oc "%s:\n" x;
  stackset := S.empty;
  stackmap := [];
  g oc (Tail, e)
```

```
let f oc (Prog(data, fundefs, e)) =
  Format.eprintf "generating assembly...@.";
  Printf.fprintf oc ".section\t\".rodata\"\n";
  Printf.fprintf oc ".align\t8\n";
  List.iter
    ( fun (Id.L(x), d) ->
      Printf.fprintf oc "%s:\t! %f\n" x d;
      Printf.fprintf oc "\t.long\t0x%lx\n" (gethi d);
      Printf.fprintf oc "\t.long\t0x%lx\n" (getlo d))
    data;
  Printf.fprintf oc ".section\t\".text\"\n";
  List.iter ( fun fundef -> h oc fundef) fundefs;
  Printf.fprintf oc ".global\tmin_caml_start\n";
  Printf.fprintf oc "min_caml_start:\n";
  Printf.fprintf oc "\tsave\t%%sp, -112, %%sp\n"; (* from gcc; why 112? *)
  stackset := S.empty;
  stackmap := [];
  g oc (NonTail("%g0"), e);
  Printf.fprintf oc "\tret\n";
  Printf.fprintf oc "\trestore\n"
```