コンパイラ演習 第5回

2005/11/10

大山 恵弘　　佐藤 秀明

今回の内容

- レジスタ割り当て
  - Interference
  - Coalescing（レジスタ合わせ）
- 生きている変数のsave/restore
  - レジスタ溢れ(spilling)
  - 関数呼び出し
  - 条件分岐

相互に依存していて厄介！

講義方針

- そこそこの速さ&そこそこの易しさ
  - MinCaml方式
    - regAlloc.target-latespill.mlをもとに説明
- 各自アルゴリズムを工夫してみてください

レジスタ割り当て

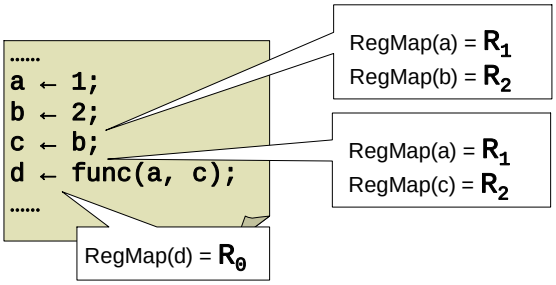
方針

1. Interference
   - 生きている変数を上書きしない
2. Coalescing(targeting)
   - 無駄なmovはできるだけ減らす

レジスタ割り当ての表現構造

- 変数からレジスタへの写像として実装
  - regenv(x) = R_n
- 処理の進行とともに写像を逐次更新

レジスタ割り当て写像の操作



生きている変数の save/restore

save命令、restore命令

- save(x): xの値をスタックに退避
  - xが他の値で上書きされるおそれのある時
- restore(x): xの値をスタックから復帰
  - いま使いたい変数がレジスタにない時

命令挿入の方針

- レジスタ溢れ
  - レジスタが足りないときに適当な変数をsave
- 関数呼び出し
  - 関数を呼び出す前に、生きている変数をsave
- 条件分岐
  - 各分岐先で異なるレジスタに割り当てられた変数は、合流後は改めてrestoreしてから使用

レジスタ溢れの例(1/2)

(汎用レジスタはR_0、R_1、R_2の3つだけとする)

```
let rec f a b =
  let x = –a in let y = –b in x – y – a – b
                        ↓
let rec f R1 R2 =
  let x = -R1 in let y = -R2 in x - y - R1 – R2
                        ↓
let rec f R1 R2 =
  let R0 = -R1 in let y = -R2 in R0 - y - R1 –
  R2
```

yに割り当てるレジスタは?

レジスタ溢れの例(2/2)

- 変数bを一時的にスタックへ退避

```
let rec f a b =
    let x = –a in let y = –b in x – y – a – b
                        ↓
let rec f a b =
    let x = –a in save(b); let y = –b in
    x – y – a – (restore(b); b)
```

関数呼び出しの例

```
let x = ... in
  let y = ... in
  let z = f x y in
  if z ≤ 0 then x – 1 else y – 2
                ↓
let x = ... in
  let y = ... in
  save(x); save(y); let z = f x y in
  if z ≤ 0  then restore(x); x – 1
   else restore(y); y – 2
```

条件分岐の例(1/2)

```
let a = if f () then x - y else y – x in a + x + y
                ↓
let a = save(x); save(y); if f ()
  then (restore(x); x) - (restore(y); y)
    (* regenv = { x→R1, y→R2 } *)
  else (restore(y); y) - (restore(x); x)
    (* regenv = { x→R2, y→R1 } *)
in a + x + y
```

合流後のレジスタ割り当ては?

条件分岐の例(2/2)

- 合流後にx、yを改めてrestore

```
let a = if f () then x - y else y – x in a + x + y
                ↓
let a = save(x); save(y); if f ()
  then (restore(x); x) - (restore(y); y)
    (* regenv = { x→R1, y→R2 } *)
  else (restore(y); y) - (restore(x); x)
    (* regenv = { x→R2, y→R1 } *)
in a + (restore(x); x) + (restore(y); y)
```

regAlloc.target-earlyspill.ml(1/3)

- regAlloc.mlと同様
- どうせ後でsaveする変数は定義直後にsave
  - ややこしいので参考程度に

regAlloc.target-earlyspill.ml(2/3)

- 変数xの退避が必要になったら、
  1. 現在の位置にForget命令を挿入
     - xのレジスタ割り当てを削除
  2. 式をToSpillコンストラクタに入れて返す
- 退避の必要な変数がなかったら、
  1. 式をレジスタ割り当て
  2. NoSpillコンストラクタに入れて返す

regAlloc.target-earlyspill.ml(3/3)

- ToSpillコンストラクタを受け取ったら、
  1. 退避する変数xの定義までさかのぼる
  2. 定義の直後にsave(x)を挿入
  3. 式のレジスタ割り当てをやり直す
- NoSpillコンストラクタを受け取ったら、
  - 式をそのまま返す

spillが多すぎるときは...

- ヒープポインタを（専用レジスタではなく）汎用レジスタにとる
  - 関数の引数や返値として付け加える
    - (x, h) ← CallCls(y, h, z_1, ..., z_n)
    - (x, h) ← CallDir(L_f, h, z_1, ..., z_n)
    - return(x, h)
- ヒープポインタをメモリ（固定）におく
  - MakeCls等が稀ならば得
- リターンアドレスを（専用レジスタではなく）汎用レジスタにとる
  - 関数からのreturnに「戻り先」として付け加える
    - return x to r

共通課題(1/2)

- 例にならい、次式へsave/restoreを挿入してみよ。どのように入れるのが「より良い」だろうか。

```
let x = ... in
let y = (if x ≤ 0 then f 1 else 2) in
let z = (if y ≤ 3 then x – 4 else g 5) in
x – y – z
```

共通課題(2/2)

- 適度に簡単な関数（フィボナッチ数列、アッカーマン関数、最大公約数など）に対し、例にならって手でレジスタ割り当てを行ってみよ。
  - 良いレジスタ割り当てをした結果と悪いレジスタ割り当てをした結果の両方を提出
    - レジスタ移動回数に差を作る
    - 少ないレジスタ数を仮定し、spill回数に差を作る
    - etc

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（11/24）の午後1時
- Subject: report 5 <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

$$
\begin{array}{lcl}
FV : \texttt{S.t} \to \texttt{SparcAsm.t} \to \texttt{S.t} & & \\
FV_s(x \leftarrow e; E) & = & s' = FV_s(E) \setminus \{x\} \text{ として } FV_{s'}(e) \\
FV_s(e) & = & FV_s(e) \\
 & & \\
FV : \texttt{S.t} \to \texttt{SparcAsm.exp} \to \texttt{S.t} & & \\
FV_s(c) & = & s \\
FV_s(\mathtt{L}_x) & = & s \\
FV_s(op(x_1, \ldots, x_n)) & = & \{x_1, \ldots, x_n\} \cup s \\
FV_s(\texttt{if } x = y \texttt{ then } E_1 \texttt{ else } E_2) & = & \{x, y\} \cup FV_s(E_1) \cup FV_s(E_2) \\
FV_s(\texttt{if } x \leq y \texttt{ then } E_1 \texttt{ else } E_2) & = & \{x, y\} \cup FV_s(E_1) \cup FV_s(E_2) \\
FV_s(x) & = & \{x\} \cup s \\
FV_s(\mathit{apply\_closure}(x, y_1, \ldots, y_n)) & = & \{x, y_1, \ldots, y_n\} \cup s \\
FV_s(\mathit{apply\_direct}(\mathtt{L}_x, y_1, \ldots, y_n)) & = & \{y_1, \ldots, y_n\} \cup s \\
FV_s(x.(y)) & = & \{x, y\} \cup s \\
FV_s(x.(y) \leftarrow z) & = & \{x, y, z\} \cup s \\
FV_s(\texttt{save}(x, y)) & = & \{x\} \cup s \\
FV_s(\texttt{restore}(y)) & = & s
\end{array}
$$

図 1: 命令の列 E および式 e において生きている変数の集合 $FV_s(E)$ および $FV_s(e)$。s は E や e の後で使われる変数の集合。以後の $FV(E)$ は $FV_\emptyset(E)$ の略記。

$\mathcal{R}$: SparcAsm.prog $\to$ SparcAsm.prog

$$\mathcal{R}((\{D_1,\ldots,D_n\},E)) = (\{\mathcal{R}(D_1),\ldots,\mathcal{R}(D_n)\},\mathcal{R}_\emptyset(E,x,())) \quad x \text{ はダミーの fresh な変数}$$

$\mathcal{R}$: SparcAsm.fundef $\to$ SparcAsm.fundef

$$\mathcal{R}(\mathtt{L}_x(y_1,\ldots,y_n)=E) = \mathtt{L}_x(\mathtt{R}_1,\ldots,\mathtt{R}_n)=\mathcal{R}_{x\mapsto\mathtt{R}_0,y_1\mapsto\mathtt{R}_1,\ldots,y_n\mapsto\mathtt{R}_n}(E,\mathtt{R}_0,\mathtt{R}_0)$$

$\mathcal{R}$: Id.t M.t $\to$ SparcAsm.t $\times$ Id.t $\times$ SparcAsm.t $\to$ SparcAsm.t $\times$ Id.t M.t

$$\begin{aligned}
\mathcal{R}_\varepsilon((x\leftarrow e;E),z_{\mathtt{dest}},E_{\mathtt{cont}}) = {} & E'_{\mathtt{cont}}=(z_{\mathtt{dest}}\leftarrow E;E_{\mathtt{cont}}),\\
& \mathcal{R}_\varepsilon(e,x,E'_{\mathtt{cont}})=(E',\varepsilon'),\\
& r\notin\{\varepsilon'(y)\mid y\in FV(E'_{\mathtt{cont}})\},\\
& \mathcal{R}_{\varepsilon',x\mapsto r}(E,z_{\mathtt{dest}},E_{\mathtt{cont}})=(E'',\varepsilon'') \text{ として}\\
& ((r\leftarrow E';E''),\varepsilon'') \qquad x \text{ がレジスタでない場合}\\
\mathcal{R}_\varepsilon((r\leftarrow e;E),z_{\mathtt{dest}},E_{\mathtt{cont}}) = {} & E'_{\mathtt{cont}}=(z_{\mathtt{dest}}\leftarrow E;E_{\mathtt{cont}}),\\
& \mathcal{R}_\varepsilon(e,r,E'_{\mathtt{cont}})=(E',\varepsilon'),\\
& \mathcal{R}_{\varepsilon'}(E,z_{\mathtt{dest}},E_{\mathtt{cont}})=(E'',\varepsilon'') \text{ として}\\
& ((r\leftarrow E';E''),\varepsilon'')\\
\mathcal{R}_\varepsilon(e,x,E_{\mathtt{cont}}) = {} & \mathcal{R}_\varepsilon(e,x,E_{\mathtt{cont}}) \qquad \text{(次図参照)}
\end{aligned}$$

図 2: 単純なレジスタ割り当て $\mathcal{R}(P)$, $\mathcal{R}(D)$ および $\mathcal{R}_\varepsilon(E,z_{\mathtt{dest}},E_{\mathtt{cont}})$。$\varepsilon$ は変数からレジスタへの写像、$z_{\mathtt{dest}}$ は E の結果をセットする変数、$E_{\mathtt{cont}}$ は E の後に実行される命令の列。$\mathcal{R}_\varepsilon(E,x,E_{\mathtt{cont}})$ の返り値はレジスタ割り当てされた命令の列 E' と、E の後のレジスタ割り当てを表す写像 ε' の組。[ファイル regAlloc.notarget-nospill.ml 参照]

$\mathcal{R}$: Id.t M.t $\to$ SparcAsm.exp $\times$ Id.t $\times$ SparcAsm.t $\to$ SparcAsm.t $\times$ Id.t M.t

$$
\begin{array}{lcl}
\mathcal{R}_\varepsilon(c, z_{\mathtt{dest}}, E_{\mathtt{cont}}) & = & (c, \varepsilon) \\
\mathcal{R}_\varepsilon(\mathtt{L}_x, z_{\mathtt{dest}}, E_{\mathtt{cont}}) & = & (\mathtt{L}_x, \varepsilon) \\
\mathcal{R}_\varepsilon(op(x_1, \ldots, x_n), z_{\mathtt{dest}}, E_{\mathtt{cont}}) & = & (op(\varepsilon(x_1), \ldots, \varepsilon(x_n)), \varepsilon) \\
\mathcal{R}_\varepsilon(\mathtt{if}\ x = y\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2, z_{\mathtt{dest}}, E_{\mathtt{cont}}) & = & \mathcal{R}_\varepsilon(E_1, z_{\mathtt{dest}}, E_{\mathtt{cont}}) = (E_1', \varepsilon_1), \\
 & & \mathcal{R}_\varepsilon(E_2, z_{\mathtt{dest}}, E_{\mathtt{cont}}) = (E_2', \varepsilon_2), \\
 & & \varepsilon' = \{z \mapsto r \mid \varepsilon_1(z) = \varepsilon_2(z) = r\}, \\
 & & \{z_1, \ldots, z_n\} = \\
 & & \qquad (FV(E_{\mathtt{cont}}) \setminus \{z_{\mathtt{dest}}\} \setminus dom(\varepsilon')) \cap dom(\varepsilon) \text{ として} \\
 & & ((\mathtt{save}(\varepsilon(z_1), z_1); \ldots; \mathtt{save}(\varepsilon(z_n), z_n); \\
 & & \ \mathtt{if}\ \varepsilon(x) \le \varepsilon(y)\ \mathtt{then}\ E_1'\ \mathtt{else}\ E_2'), \varepsilon') \\
\mathcal{R}_\varepsilon(\mathtt{if}\ x \le y\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2, z_{\mathtt{dest}}, E_{\mathtt{cont}}) & = & \text{同様} \\
\mathcal{R}_\varepsilon(x, z_{\mathtt{dest}}, E_{\mathtt{cont}}) & = & (\varepsilon(x), \varepsilon) \\
\mathcal{R}_\varepsilon(apply\_closure(x, y_1, \ldots, y_n), z_{\mathtt{dest}}, E_{\mathtt{cont}}) & = & \{z_1, \ldots, z_n\} = (FV(E_{\mathtt{cont}}) \setminus \{z_{\mathtt{dest}}\}) \cap dom(\varepsilon) \text{ として} \\
 & & ((\mathtt{save}(\varepsilon(z_1), z_1); \ldots; \mathtt{save}(\varepsilon(z_n), z_n); \\
 & & \ apply\_closure(\varepsilon(x), \varepsilon(y_1), \ldots, \varepsilon(y_n))), \emptyset) \\
\mathcal{R}_\varepsilon(apply\_direct(\mathtt{L}_x, y_1, \ldots, y_n), z_{\mathtt{dest}}, E_{\mathtt{cont}}) & = & \text{同様} \\
\mathcal{R}_\varepsilon(x.(y), z_{\mathtt{dest}}, E_{\mathtt{cont}}) & = & (\varepsilon(x).(\varepsilon(y)), \varepsilon) \\
\mathcal{R}_\varepsilon(x.(y) \leftarrow z, z_{\mathtt{dest}}, E_{\mathtt{cont}}) & = & (\varepsilon(x).(\varepsilon(y)) \leftarrow \varepsilon(z), \varepsilon) \\
\mathcal{R}_\varepsilon(\mathtt{save}(x, y), z_{\mathtt{dest}}, E_{\mathtt{cont}}) & = & (\mathtt{save}(\varepsilon(x), y), \varepsilon) \\
\mathcal{R}_\varepsilon(\mathtt{restore}(y), z_{\mathtt{dest}}, E_{\mathtt{cont}}) & = & (\mathtt{restore}(y), \varepsilon)
\end{array}
$$

図 3: 単純なレジスタ割り当て $\mathcal{R}_\varepsilon(e, z_{\mathtt{dest}}, E_{\mathtt{cont}})$。$\mathcal{R}_\varepsilon(e)$ の右辺で変数 x のレジスタ $\varepsilon(x)$ が定義されていない場合は、$\mathcal{R}_\varepsilon(e) = \mathcal{R}_\varepsilon(x \leftarrow \mathtt{restore}(x); e)$ とする。ただしレジスタ r については $\varepsilon(r) = r$ とする。[ファイル `regAlloc.notarget-nospill.ml` 参照]

$\mathcal{T}$: Id.t $\to$ SparcAsm.t $\times$ Id.t $\to$ bool $\times$ S.t

$$
\begin{array}{lcl}
\mathcal{T}_x((y \leftarrow e; E), z_{\mathtt{dest}}) & = & \mathcal{T}_x(e, y) = (c_1, s_1) \text{ として、もし } c_1 \text{ならば } (true, s_1) \\
 & & \text{そうでなければ } \mathcal{T}_x(E, z_{\mathtt{dest}}) = (c_2, s_2) \text{ として } (c_2, s_1 \cup s_2) \\
\mathcal{T}_x(e, z_{\mathtt{dest}}) & = & \mathcal{T}_x(e, z_{\mathtt{dest}})
\end{array}
$$

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

$$
\begin{array}{lcl}
\mathcal{T}_x(x, z_{\mathtt{dest}}) & = & (false, \{z_{\mathtt{dest}}\}) \\
\mathcal{T}_x(\mathtt{if}\ y = z\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2, z_{\mathtt{dest}}) & = & \mathcal{T}_x(E_1, z_{\mathtt{dest}}) = (c_1, s_1), \\
 & & \mathcal{T}_x(E_2, z_{\mathtt{dest}}) = (c_2, s_2) \text{ として} \\
 & & (c_1 \wedge c_2, s_1 \cup s_2) \\
\mathcal{T}_x(\mathtt{if}\ y \leq z\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2, z_{\mathtt{dest}}) & = & \text{同上} \\
\mathcal{T}_x(apply\_closure(y_0, y_1, \ldots, y_n), z_{\mathtt{dest}}) & = & (true, \{\mathtt{R}_i \mid x = y_i\}) \\
\mathcal{T}_x(apply\_direct(\mathtt{L}_y, y_1, \ldots, y_n), z_{\mathtt{dest}}) & = & \text{同上} \\
\mathcal{T}_x(e, z_{\mathtt{dest}}) & = & (false, \emptyset) \qquad \text{それ以外の場合}
\end{array}
$$

図 4: 変数 x に割り当てるレジスタ r を選ぶときに使う targeting $\mathcal{T}_x(E, z_{\mathtt{dest}})$ および $\mathcal{T}_x(e, z_{\mathtt{dest}})$。$E$ や e で関数呼び出しがあったかどうかを表す論理値 c と、x を割り当てると良いレジスタの集合 s の組を返す。前々図の「x がレジスタでない場合」において、$\mathcal{T}_x(E'_{\mathtt{cont}}, z_{\mathtt{dest}}) = (c, s)$ として、できれば $r \in s$ とする。[ファイル regAlloc.target-nospill.ml 参照]

$\mathcal{R}$: Id.t M.t $\to$ SparcAsm.t $\times$ Id.t $\times$ SparcAsm.t $\to$ SparcAsm.t $\times$ Id.t M.t

$$
\begin{array}{lcl}
\mathcal{R}_\varepsilon((x \leftarrow e; E), z_{\mathtt{dest}}, E_{\mathtt{cont}}) & = & E'_{\mathtt{cont}} = (z_{\mathtt{dest}} \leftarrow E; E_{\mathtt{cont}}), \\
 & & \mathcal{R}_\varepsilon(e, x, E'_{\mathtt{cont}}) = (E', \varepsilon'), \\
 & & y \in FV(E'_{\mathtt{cont}}), \\
 & & \mathcal{R}_{\varepsilon' \setminus \{y \mapsto \varepsilon'(y)\}, x \mapsto \varepsilon'(y)}(E, z_{\mathtt{dest}}, E_{\mathtt{cont}}) = (E'', \varepsilon'') \text{ として} \\
 & & ((\mathtt{save}(\varepsilon(y), y), r \leftarrow E'; E''), \varepsilon'') \\
 & & \quad x \text{ がレジスタでなく、} \\
 & & \quad r \notin \{\varepsilon'(y) \mid y \in FV(E'_{\mathtt{cont}})\} \text{ なる } r \text{ がない場合}
\end{array}
$$

図 5: spilling をするレジスタ割り当て $\mathcal{R}_\varepsilon(E, z_{\mathtt{dest}}, E_{\mathtt{cont}})$ [ファイル regAlloc.target-latespill.ml 参照]

```
open SparcAsm

(* for register coalescing *)
(* [XXX] Callがあったら、そこから先は無意味というか逆効果なので追わない。
         そのために「Callがあったかどうか」を返り値の第1要素に含める。 *)
let rec target' src (dest, t) = function
  | Mov(x) when x = src && is_reg dest ->
      assert (t <> Type.Unit);
      assert (t <> Type.Float);
      false, [dest]
  | FMovD(x) when x = src && is_reg dest ->
      assert (t = Type.Float);
      false, [dest]
  | IfEq(_, _, e1, e2) | IfLE(_, _, e1, e2) | IfGE(_, _, e1, e2)
  | IfFEq(_, _, e1, e2) | IfFLE(_, _, e1, e2) ->
      let c1, rs1 = target src (dest, t) e1 in
      let c2, rs2 = target src (dest, t) e2 in
      c1 && c2, rs1 @ rs2
  | CallCls(x, ys, zs) ->
      true, (target_args src regs 0 ys @
             target_args src fregs 0 zs @
             if x = src then [reg_cl] else [])
  | CallDir(_, ys, zs) ->
      true, (target_args src regs 0 ys @
             target_args src fregs 0 zs)
  | _ -> false, []
and target src dest = function (* register targeting (caml2html: regalloc_target
) *)
  | Ans(exp) -> target' src dest exp
  | Let(xt, exp, e) ->
      let c1, rs1 = target' src xt exp in
      if c1 then true, rs1 else
      let c2, rs2 = target src dest e in
      c2, rs1 @ rs2
  | Forget(_, e) -> target src dest e
and target_args src all n = function (* auxiliary function for Call *)
  | [] -> []
  | y :: ys when src = y -> all.(n) :: target_args src all (n + 1) ys
  | _ :: ys -> target_args src all (n + 1) ys

type alloc_result = (* allocにおいてspillingがあったかどうかを表すデータ型 *)
  | Alloc of Id.t (* allocated register *)
  | Spill of Id.t (* spilled variable *)
let rec alloc dest cont regenv x t =
  (* allocate a register or spill a variable *)
  assert (not (M.mem x regenv));
  let all =
    match t with
    | Type.Unit -> ["%g0"] (* dummy *)
    | Type.Float -> allfregs
    | _ -> allregs in
  if all = ["%g0"] then Alloc("%g0") else (* [XX] ad hoc optimization *)
  if is_reg x then Alloc(x) else
  let free = fv cont in
  try
    let (c, prefer) = target x dest cont in
    let live = (* 生きているレジスタ *)
      List.fold_left
        (fun live y ->
          if is_reg y then S.add y live else
          try S.add (M.find y regenv) live
          with Not_found -> live)
```



```
        S.empty
        free in
    let r = (* そうでないレジスタを探す *)
      List.find
        (fun r -> not (S.mem r live))
        (prefer @ all) in
    (* Format.eprintf "allocated %s to %s@." x r; *)
    Alloc(r)
  with Not_found ->
    Format.eprintf "register allocation failed for %s@." x;
    let y = (* 型の合うレジスタ変数を探す *)
      List.find
        (fun y ->
          not (is_reg y) &&
          try List.mem (M.find y regenv) all
          with Not_found -> false)
        (List.rev free) in
    Format.eprintf "spilling %s from %s@." y (M.find y regenv);
    Spill(y)

(* auxiliary function for g and g'_and_restore *)
let add x r regenv =
  if is_reg x then (assert (x = r); regenv) else
  M.add x r regenv

(* auxiliary functions for g' *)
exception NoReg of Id.t * Type.t
let find x t regenv =
  if is_reg x then x else
  try M.find x regenv
  with Not_found -> raise (NoReg(x, t))
let find' x' regenv =
  match x' with
  | V(x) -> V(find x Type.Int regenv)
  | c -> c

let rec g dest cont regenv = function (* 命令列のレジスタ割り当て (caml2html: regallo
c_g) *)
  | Ans(exp) -> g'_and_restore dest cont regenv exp
  | Let((x, t) as xt, exp, e) ->
      assert (not (M.mem x regenv));
      let cont' = concat e dest cont in
      let (e1', regenv1) = g'_and_restore xt cont' regenv exp in
      (match alloc dest cont' regenv1 x t with
      | Spill(y) ->
          let r = M.find y regenv in
          let r1 = M.find y regenv1 in
          let (e2', regenv2) = g dest cont (add x r1 (M.remove y regenv1)) e in
          (seq(Save(r, y), concat e1' (r, t) e2'), regenv2)
      | Alloc(r) ->
          let (e2', regenv2) = g dest cont (add x r regenv1) e in
          (concat e1' (r, t) e2', regenv2))
  | Forget(x, e) -> assert false
and g'_and_restore dest cont regenv exp = (* 使用される変数をスタックからレジスタへRestor
e (caml2html: regalloc_unspill) *)
  try g' dest cont regenv exp
  with NoReg(x, t) ->
    ((* Format.eprintf "restoring %s@." x; *)
     g dest cont regenv (Let((x, t), Restore(x), Ans(exp))))
and g' dest cont regenv = function (* 各命令のレジスタ割り当て (caml2html: regalloc_g
prime) *)
  | Nop | Set _ | SetL _ | Comment _ | Restore _ as exp -> (Ans(exp), regenv)
```

```
  | Mov(x) -> (Ans(Mov(find x Type.Int regenv)), regenv)
  | Neg(x) -> (Ans(Neg(find x Type.Int regenv)), regenv)
  | Add(x, y') -> (Ans(Add(find x Type.Int regenv, find' y' regenv)), regenv)
  | Sub(x, y') -> (Ans(Sub(find x Type.Int regenv, find' y' regenv)), regenv)
  | SLL(x, y') -> (Ans(SLL(find x Type.Int regenv, find' y' regenv)), regenv)
  | Ld(x, y') -> (Ans(Ld(find x Type.Int regenv, find' y' regenv)), regenv)
  | St(x, y, z') -> (Ans(St(find x Type.Int regenv, find y Type.Int regenv, find
' z' regenv)), regenv)
  | FMovD(x) -> (Ans(FMovD(find x Type.Float regenv)), regenv)
  | FNegD(x) -> (Ans(FNegD(find x Type.Float regenv)), regenv)
  | FAddD(x, y) -> (Ans(FAddD(find x Type.Float regenv, find y Type.Float regenv
)), regenv)
  | FSubD(x, y) -> (Ans(FSubD(find x Type.Float regenv, find y Type.Float regenv
)), regenv)
  | FMulD(x, y) -> (Ans(FMulD(find x Type.Float regenv, find y Type.Float regenv
)), regenv)
  | FDivD(x, y) -> (Ans(FDivD(find x Type.Float regenv, find y Type.Float regenv
)), regenv)
  | LdDF(x, y') -> (Ans(LdDF(find x Type.Int regenv, find' y' regenv)), regenv)
  | StDF(x, y, z') -> (Ans(StDF(find x Type.Float regenv, find y Type.Int regenv
, find' z' regenv)), regenv)
  | IfEq(x, y', e1, e2) as exp -> g'_if dest cont regenv exp ( fun e1' e2' -> IfE
q(find x Type.Int regenv, find' y' regenv, e1', e2')) e1 e2
  | IfLE(x, y', e1, e2) as exp -> g'_if dest cont regenv exp ( fun e1' e2' -> IfL
E(find x Type.Int regenv, find' y' regenv, e1', e2')) e1 e2
  | IfGE(x, y', e1, e2) as exp -> g'_if dest cont regenv exp ( fun e1' e2' -> IfG
E(find x Type.Int regenv, find' y' regenv, e1', e2')) e1 e2
  | IfFEq(x, y, e1, e2) as exp -> g'_if dest cont regenv exp ( fun e1' e2' -> IfF
Eq(find x Type.Float regenv, find y Type.Float regenv, e1', e2')) e1 e2
  | IfFLE(x, y, e1, e2) as exp -> g'_if dest cont regenv exp ( fun e1' e2' -> IfF
LE(find x Type.Float regenv, find y Type.Float regenv, e1', e2')) e1 e2
  | CallCls(x, ys, zs) as exp -> g'_call dest cont regenv exp ( fun ys zs -> Call
Cls(find x Type.Int regenv, ys, zs)) ys zs
  | CallDir(l, ys, zs) as exp -> g'_call dest cont regenv exp ( fun ys zs -> Call
Dir(l, ys, zs)) ys zs
  | Save(x, y) -> assert false
and g'_if dest cont regenv exp constr e1 e2 = (* ifのレジスタ割り当て (caml2html: re
galloc_if) *)
  let (e1', regenv1) = g dest cont regenv e1 in
  let (e2', regenv2) = g dest cont regenv e2 in
  let regenv' = (* 両方に共通のレジスタ変数だけ利用 *)
    List.fold_left
      ( fun regenv' x ->
        try
          if is_reg x then regenv' else
          let r1 = M.find x regenv1 in
          let r2 = M.find x regenv2 in
          if r1 <> r2 then regenv' else
          M.add x r1 regenv'
        with Not_found -> regenv')
       M.empty
       (fv cont) in
  (List.fold_left
     ( fun e x ->
       if x = fst dest || not (M.mem x regenv) || M.mem x regenv' then e else
       seq(Save(M.find x regenv, x), e)) (* そうでない変数は分岐直前にセーブ *)
     (Ans(constr e1' e2'))
     (fv cont),
   regenv')
and g'_call dest cont regenv exp constr ys zs = (* 関数呼び出しのレジスタ割り当て (caml
2html: regalloc_call) *)
  (List.fold_left
```



```
     ( fun e x ->
       if x = fst dest || not (M.mem x regenv) then e else
       seq(Save(M.find x regenv, x), e))
     (Ans(constr
            (List.map ( fun y -> find y Type.Int regenv) ys)
            (List.map ( fun z -> find z Type.Float regenv) zs)))
     (fv cont),
   M.empty)

let h { name = Id.L(x); args = ys; fargs = zs; body = e; ret = t } = (* 関数のレジ
スタ割り当て (caml2html: regalloc_h) *)
  let regenv = M.add x reg_cl M.empty in
  let (i, arg_regs, regenv) =
    List.fold_left
      ( fun (i, arg_regs, regenv) y ->
        let r = regs.(i) in
        (i + 1,
         arg_regs @ [r],
         ( assert ( not (is_reg y));
          M.add y r regenv)))
      (0, [], regenv)
      ys in
  let (d, farg_regs, regenv) =
    List.fold_left
      ( fun (d, farg_regs, regenv) z ->
        let fr = fregs.(d) in
        (d + 1,
         farg_regs @ [fr],
         ( assert ( not (is_reg z));
          M.add z fr regenv)))
      (0, [], regenv)
      zs in
  let a =
    match t with
    | Type.Unit -> Id.gentmp Type.Unit
    | Type.Float -> fregs.(0)
    | _ -> regs.(0) in
  let (e', regenv') = g (a, t) (Ans(Mov(a))) regenv e in
  { name = Id.L(x); args = arg_regs; fargs = farg_regs; body = e'; ret = t }

let f (Prog(data, fundefs, e)) = (* プログラム全体のレジスタ割り当て (caml2html: regall
oc_f) *)
  Format.eprintf "register allocation: may take some time (up to a few minutes, depending on the size of functi
ons)@.";
  let fundefs' = List.map h fundefs in
  let e', regenv' = g (Id.gentmp Type.Unit, Type.Unit) (Ans(Nop)) M.empty e in
  Prog(data, fundefs', e')
```

```
open SparcAsm

(* for register coalescing *)
(* [XXX] Callがあったら、そこから先は無意味というか逆効果なので追わない。
         そのために「Callがあったかどうか」を返り値の第1要素に含める。 *)
let rec target' src (dest, t) = function
  | Mov(x) when x = src && is_reg dest ->
      assert (t <> Type.Unit);
      assert (t <> Type.Float);
      false, [dest]
  | FMovD(x) when x = src && is_reg dest ->
      assert (t = Type.Float);
      false, [dest]
  | IfEq(_, _, e1, e2) | IfLE(_, _, e1, e2) | IfGE(_, _, e1, e2)
  | IfFEq(_, _, e1, e2) | IfFLE(_, _, e1, e2) ->
      let c1, rs1 = target src (dest, t) e1 in
      let c2, rs2 = target src (dest, t) e2 in
      c1 && c2, rs1 @ rs2
  | CallCls(x, ys, zs) ->
      true, (target_args src regs 0 ys @
             target_args src fregs 0 zs @
             if x = src then [reg_cl] else [])
  | CallDir(_, ys, zs) ->
      true, (target_args src regs 0 ys @
             target_args src fregs 0 zs)
  | _ -> false, []
and target src dest = function (* register targeting (cam12html: regalloc_target
) *)
  | Ans(exp) -> target' src dest exp
  | Let(xt, exp, e) ->
      let c1, rs1 = target' src xt exp in
      if c1 then true, rs1 else
      let c2, rs2 = target src dest e in
      c2, rs1 @ rs2
  | Forget(_, e) -> target src dest e
and target_args src all n = function (* auxiliary function for Call *)
  | [] -> []
  | y :: ys when src = y -> all.(n) :: target_args src all (n + 1) ys
  | _ :: ys -> target_args src all (n + 1) ys

type alloc_result = (* allocにおいてspillingがあったかどうかを表すデータ型 *)
  | Alloc of Id.t (* allocated register *)
  | Spill of Id.t (* spilled variable *)
let rec alloc dest cont regenv x t =
  (* allocate a register or spill a variable *)
  assert (not (M.mem x regenv));
  let all =
    match t with
    | Type.Unit -> ["%g0"] (* dummy *)
    | Type.Float -> allfregs
    | _ -> allregs in
  if all = ["%g0"] then Alloc("%g0") else (* [XX] ad hoc optimization *)
  if is_reg x then Alloc(x) else
  let free = fv cont in
  try
    let (c, prefer) = target x dest cont in
    let live = (* 生きているレジスタ *)
      List.fold_left
        (fun live y ->
          if is_reg y then S.add y live else
          try S.add (M.find y regenv) live
          with Not_found -> live)
```



```
        S.empty
        free in
    let r = (* そうでないレジスタを探す *)
      List.find
        (fun r -> not (S.mem r live))
        (prefer @ all) in
    (* Format.eprintf "allocated %s to %s@." x r; *)
    Alloc(r)
  with Not_found ->
    Format.eprintf "register allocation failed for %s@." x;
    let y = (* 型の合うレジスタ変数を探す *)
      List.find
        (fun y ->
          not (is_reg y) &&
          try List.mem (M.find y regenv) all
          with Not_found -> false)
        (List.rev free) in
    Format.eprintf "spilling %s from %s@." y (M.find y regenv);
    Spill(y)

(* auxiliary function for g and g'_and_restore *)
let add x r regenv =
  if is_reg x then (assert (x = r); regenv) else
  M.add x r regenv

type g_result = (* gやg'においてspillingがあったかどうかを表すデータ型 (cam12html: regall
oc_result) *)
  | NoSpill of t * Id.t M.t (* new regenv *)
  | ToSpill of t * Id.t list (* spilled variables *)

(* auxiliary functions for g' *)
exception NoReg of Id.t * Type.t
let find x t regenv =
  if is_reg x then x else
  try M.find x regenv
  with Not_found -> raise (NoReg(x, t))
let find' x' regenv =
  match x' with
  | V(x) -> V(find x Type.Int regenv)
  | c -> c
let forget_list xs e =
  List.fold_left
    (fun e x -> Forget(x, e))
    e
    xs
let insert_forget xs exp t =
  let a = Id.gentmp t in
  let m =
    match t with
    | Type.Unit -> Nop
    | Type.Float -> FMovD(a)
    | _ -> Mov(a) in
  ToSpill(Let((a, t), exp, forget_list xs (Ans(m))), xs)

let rec g dest cont regenv = function (* 命令列のレジスタ割り当て (cam12html: regallo
c_g) *)
  | Ans(exp) -> g'_and_restore dest cont regenv exp
  | Let((x, t) as xt, exp, e) ->
      assert (not (M.mem x regenv));
      let cont' = concat e dest cont in
      (match g'_and_restore xt cont' regenv exp with
      | ToSpill(e1, ys) -> ToSpill(concat e1 xt e, ys)
```

```
      | NoSpill(e1', regenv1) ->
          ( match alloc dest cont' regenv1 x t with
          | Spill(y) -> ToSpill(Let(xt, exp, Forget(y, e)), [y])
          | Alloc(r) ->
              match g dest cont (add x r regenv1) e with
              | ToSpill(e2, ys) when List.mem x ys ->
                  let x_saved = Let(xt, exp, seq(Save(x, x), e2)) in
                  ( match List.filter ( fun y -> y <> x) ys with
                  | [] -> g dest cont regenv x_saved
                  | ys_left -> ToSpill(x_saved, ys_left))
              | ToSpill(e2, ys) -> ToSpill(Let(xt, exp, e2), ys)
              | NoSpill(e2', regenv2) -> NoSpill(concat e1' (r, t) e2', regenv2)
))
  | Forget(x, e) ->
      ( match g dest cont (M.remove x regenv) e with
      | ToSpill(e1, ys) ->
          let x_forgotten = Forget(x, e1) in
          ( match List.filter ( fun y -> y <> x) ys with
          | [] -> g dest cont regenv x_forgotten
          | ys_left -> ToSpill(x_forgotten, ys_left))
      | NoSpill(e1', regenv1) -> NoSpill(e1', regenv1))
and g'_and_restore dest cont regenv exp = (* 使用される変数をスタックからレジスタへRestore (caml2html: regalloc_unspill) *)
  try g' dest cont regenv exp
  with NoReg(x, t) ->
    ( (* Format.eprintf "restoring %s@." x; *)
     g dest cont regenv (Let((x, t), Restore(x), Ans(exp))))
and g' dest cont regenv = function (* 各命令のレジスタ割り当て (caml2html: regalloc_gprime) *)
  | Nop | Set _ | SetL _ | Comment _ | Restore _ as exp -> NoSpill(Ans(exp), regenv)
  | Mov(x) -> NoSpill(Ans(Mov(find x Type.Int regenv)), regenv)
  | Neg(x) -> NoSpill(Ans(Neg(find x Type.Int regenv)), regenv)
  | Add(x, y') -> NoSpill(Ans(Add(find x Type.Int regenv, find' y' regenv)), regenv)
  | Sub(x, y') -> NoSpill(Ans(Sub(find x Type.Int regenv, find' y' regenv)), regenv)
  | SLL(x, y') -> NoSpill(Ans(SLL(find x Type.Int regenv, find' y' regenv)), regenv)
  | Ld(x, y') -> NoSpill(Ans(Ld(find x Type.Int regenv, find' y' regenv)), regenv)
  | St(x, y, z') -> NoSpill(Ans(St(find x Type.Int regenv, find y Type.Int regenv, find' z' regenv)), regenv)
  | FMovD(x) -> NoSpill(Ans(FMovD(find x Type.Float regenv)), regenv)
  | FNegD(x) -> NoSpill(Ans(FNegD(find x Type.Float regenv)), regenv)
  | FAddD(x, y) -> NoSpill(Ans(FAddD(find x Type.Float regenv, find y Type.Float regenv)), regenv)
  | FSubD(x, y) -> NoSpill(Ans(FSubD(find x Type.Float regenv, find y Type.Float regenv)), regenv)
  | FMulD(x, y) -> NoSpill(Ans(FMulD(find x Type.Float regenv, find y Type.Float regenv)), regenv)
  | FDivD(x, y) -> NoSpill(Ans(FDivD(find x Type.Float regenv, find y Type.Float regenv)), regenv)
  | LdDF(x, y') -> NoSpill(Ans(LdDF(find x Type.Int regenv, find' y' regenv)), regenv)
  | StDF(x, y, z') -> NoSpill(Ans(StDF(find x Type.Float regenv, find y Type.Int regenv, find' z' regenv)), regenv)
  | IfEq(x, y', e1, e2) as exp -> g'_if dest cont regenv exp ( fun e1' e2' -> IfEq(find x Type.Int regenv, find' y' regenv, e1', e2')) e1 e2
  | IfLE(x, y', e1, e2) as exp -> g'_if dest cont regenv exp ( fun e1' e2' -> IfLE(find x Type.Int regenv, find' y' regenv, e1', e2')) e1 e2
  | IfGE(x, y', e1, e2) as exp -> g'_if dest cont regenv exp ( fun e1' e2' -> IfG
```



```
E(find x Type.Int regenv, find' y' regenv, e1', e2')) e1 e2
  | IfFEq(x, y, e1, e2) as exp -> g'_if dest cont regenv exp ( fun e1' e2' -> IfF
Eq(find x Type.Float regenv, find y Type.Float regenv, e1', e2')) e1 e2
  | IfFLE(x, y, e1, e2) as exp -> g'_if dest cont regenv exp ( fun e1' e2' -> IfF
LE(find x Type.Float regenv, find y Type.Float regenv, e1', e2')) e1 e2
  | CallCls(x, ys, zs) as exp -> g'_call dest cont regenv exp ( fun ys zs -> Call
Cls(find x Type.Int regenv, ys, zs)) ys zs
  | CallDir(l, ys, zs) as exp -> g'_call dest cont regenv exp ( fun ys zs -> Call
Dir(l, ys, zs)) ys zs
  | Save(x, y) ->
      assert (x = y);
      assert ( not (is_reg x));
      try NoSpill(Ans(Save(M.find x regenv, x)), regenv)
      with Not_found -> NoSpill(Ans(Nop), regenv) (* must have already been save
d *)
and g'_if dest cont regenv exp constr e1 e2 = (* ifのレジスタ割り当て (caml2html: re
galloc_if) *)
  let (e1', regenv1) = g_repeat dest cont regenv e1 in
  let (e2', regenv2) = g_repeat dest cont regenv e2 in
  let regenv' = (* 両方に共通のレジスタ変数だけ利用 *)
    List.fold_left
      ( fun regenv' x ->
        try
          if is_reg x then regenv' else
          let r1 = M.find x regenv1 in
          let r2 = M.find x regenv2 in
          if r1 <> r2 then regenv' else
          M.add x r1 regenv'
        with Not_found -> regenv')
      M.empty
      (fv cont) in
  match
    List.filter
      ( fun x -> not (is_reg x) && x <> fst dest && not (M.mem x regenv'))
      (fv cont)
  with [] -> NoSpill(Ans(constr e1' e2'), regenv')
  | xs -> insert_forget xs exp (snd dest) (* そうでない変数は分岐以前にセーブ *)
and g'_call dest cont regenv exp constr ys zs = (* 関数呼び出しのレジスタ割り当て (caml
2html: regalloc_call) *)
  match
    List.filter (* セーブすべきレジスタ変数を探す *)
      ( fun x -> not (is_reg x) && x <> fst dest)
      (fv cont)
  with [] -> NoSpill(Ans(constr
                         (List.map ( fun y -> find y Type.Int regenv) ys)
                         (List.map ( fun z -> find z Type.Float regenv) zs)),
                     M.empty)
  | xs -> insert_forget xs exp (snd dest)
and g_repeat dest cont regenv e = (* Spillがなくなるまでgを繰り返す (caml2html: regal
loc_repeat) *)
    match g dest cont regenv e with
    | NoSpill(e', regenv') -> (e', regenv')
    | ToSpill(e, xs) ->
        g_repeat dest cont regenv
          (List.fold_left
             ( fun e x -> seq(Save(x, x), e))
             e
             xs)

let h { name = Id.L(x); args = ys; fargs = zs; body = e; ret = t } = (* 関数のレジ
スタ割り当て (caml2html: regalloc_h) *)
  let regenv = M.add x reg_cl M.empty in
```

```
  let (i, arg_regs, regenv) =
    List.fold_left
      ( fun (i, arg_regs, regenv) y ->
        let r = regs.(i) in
        (i + 1,
         arg_regs @ [r],
         ( assert ( not (is_reg y));
          M.add y r regenv)))
      (0, [], regenv)
      ys in
  let (d, farg_regs, regenv) =
    List.fold_left
      ( fun (d, farg_regs, regenv) z ->
        let fr = fregs.(d) in
        (d + 1,
         farg_regs @ [fr],
         ( assert ( not (is_reg z));
          M.add z fr regenv)))
      (0, [], regenv)
      zs in
  let a =
    match t with
    | Type.Unit -> Id.gentmp Type.Unit
    | Type.Float -> fregs.(0)
    | _ -> regs.(0) in
  let (e', regenv') = g_repeat (a, t) (Ans(Mov(a))) regenv e in
  { name = Id.L(x); args = arg_regs; fargs = farg_regs; body = e'; ret = t }

let f (Prog(data, fundefs, e)) = (* プログラム全体のレジスタ割り当て (caml2html: regall
oc_f) *)
  Format.eprintf "register allocation: may take some time (up to a few minutes, depending on the size of functi
ons)@.";
  let fundefs' = List.map h fundefs in
  let e', regenv' = g_repeat (Id.gentmp Type.Unit, Type.Unit) (Ans(Nop)) M.empty
 e in
  Prog(data, fundefs', e')
```