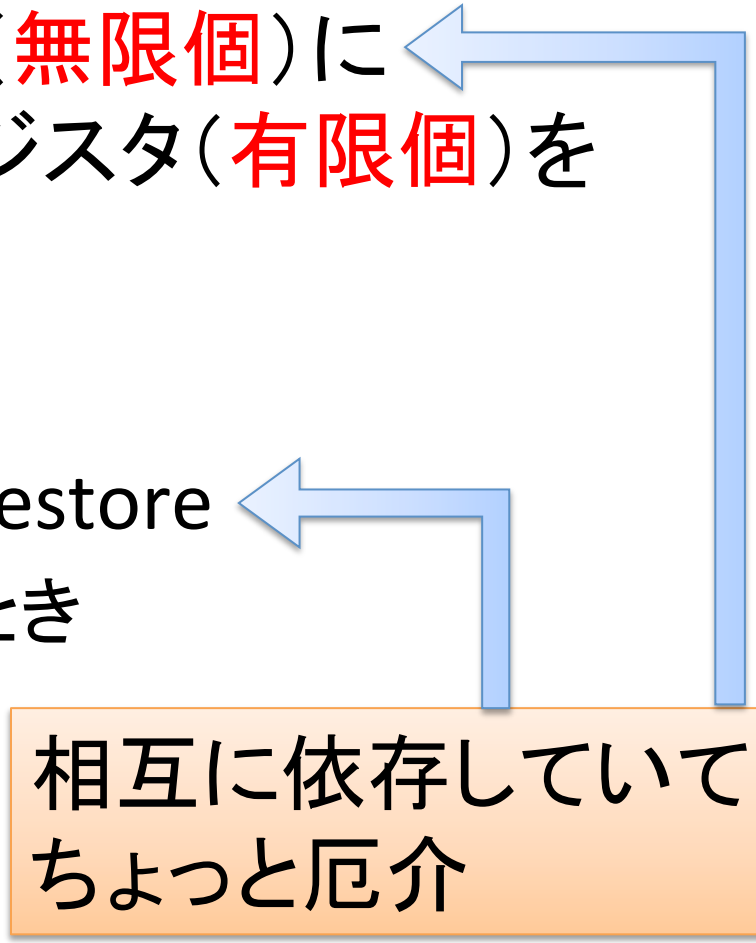
コンパイラ演習
第 5 回
(2011/11/10)

中村 晃一　野瀬 貴史　前田 俊行
秋山 茂樹　池尻 拓朗
鈴木 友博　渡邊 裕貴
潮田 資秀
小酒井 隆広
山下 諒蔵　佐藤 春旗
大山 恵弘　佐藤 秀明
住井 英二郎

今日の内容: レジスタ割り当て

- 仮想マシンコードの変数(無限個)に
  実際のハードウェアのレジスタ(有限個)を
  割り当てる

- 生きている変数の save/restore
  - レジスタ溢れ (spilling) のとき
  - 関数呼び出しのとき
  - 条件分岐のとき



レジスタ割り当てのポイント

- どうやって割り当てるレジスタを決めるのか？
  - 空いているレジスタを割り当てる？
    - 空いているレジスタって？
  - メモリアクセス回数をどのように最小化する？
    - レジスタ溢れ発生時に
      どの変数をどのタイミングで save すべき？
    - restore はどのタイミングで行うべき？

MinCaml の方針

- そこそこの速さ & そこそこの易しさ
  - regAlloc.ml
  - 各自アルゴリズムを工夫するとよい
    - グラフ彩色問題や整数線形計画問題に還元する
    - 関数間解析を行う
    - etc.

RegAlloc.f

- Asm.prog から Asm.prog への変換
  - レジスタマップを更新・参照しながら
    変数をレジスタに書き換えていく
  - ただし
    - 変数のsave/restore命令を適宜挿入する

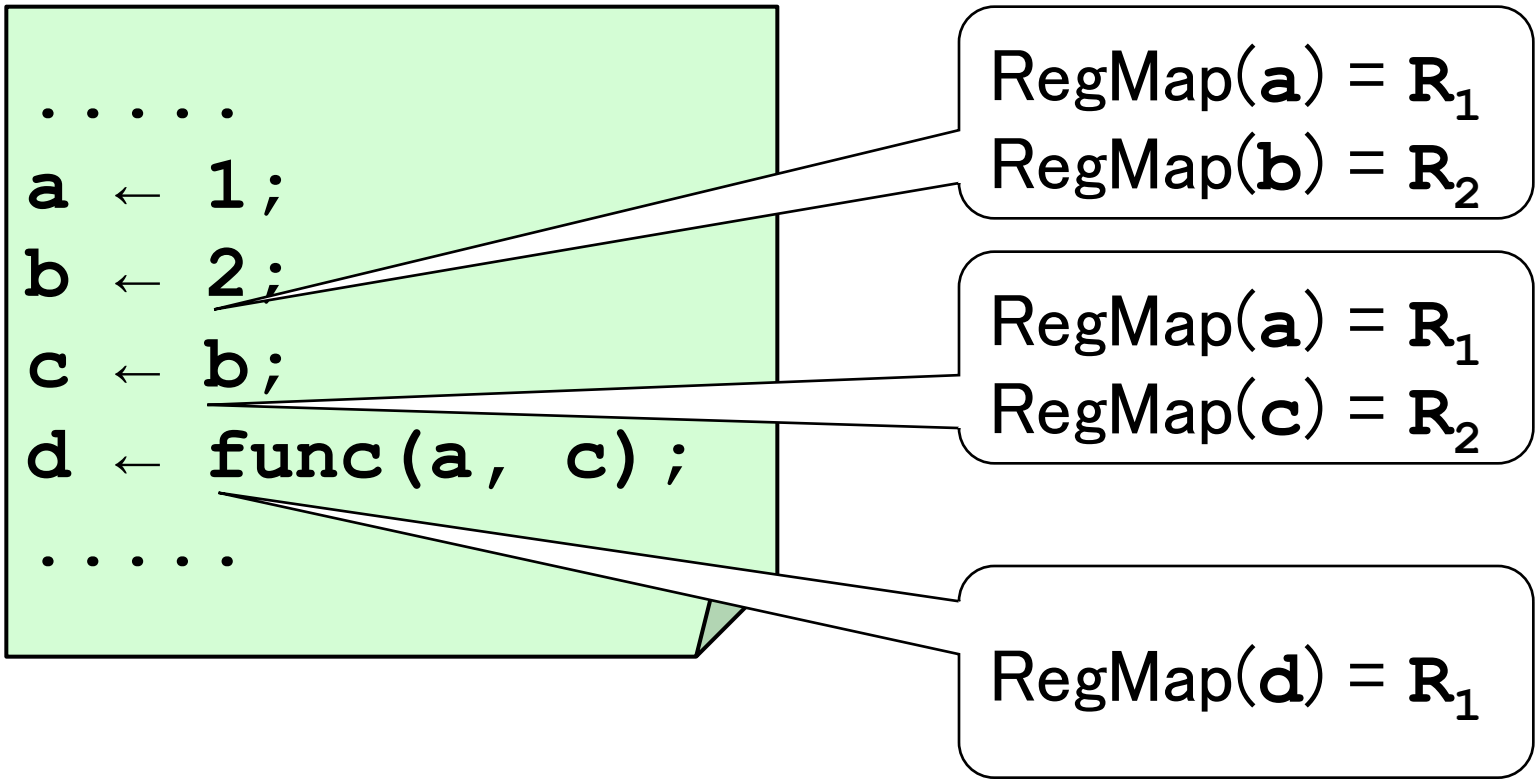
レジスタ割り当ての方針

- とりあえず適当に空いているレジスタを変数に割り当てていく
  - 「空いているレジスタ」
    = 生きている変数に割り当てられていないレジスタ
  - 「生きている変数」
    = これから使われうる変数
    = (後続の命令中の) 自由変数
- ただし、関数の引数やクロージャは決め打ち
  - 関数呼び出し規約に従う

レジスタ割り当ての表現

- 変数からレジスタへの写像として実装
  - RegMap($\mathbf{x}$) = $\mathbf{R_n}$
    - MinCaml のコード中では RegMap は regenv という変数で表される
- 割り当ての進行とともに写像を更新

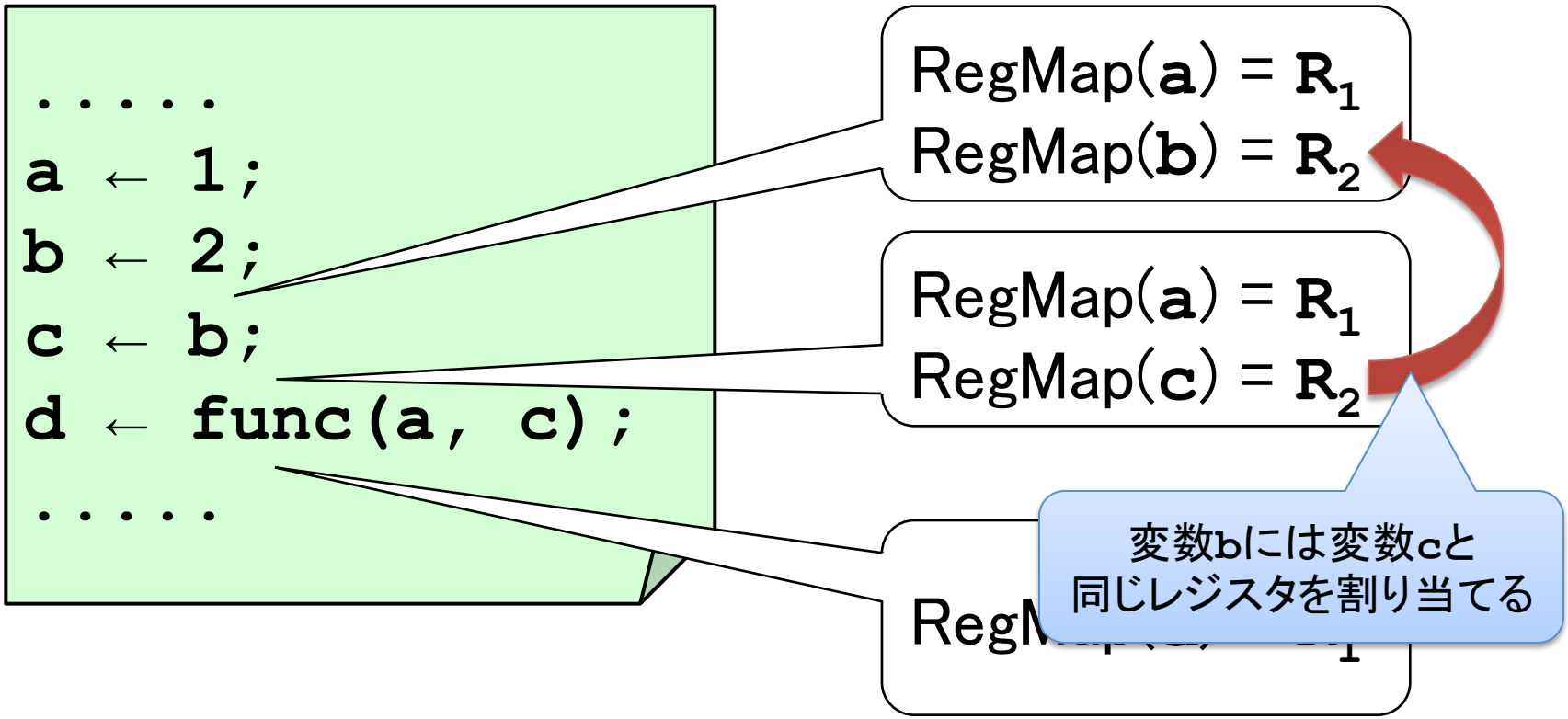
レジスタ割り当て写像の更新



Coalescing (targeting)

- 無駄な mov 命令をなるべく減らすように
  レジスタを割りつける
  - 関数呼び出し時に
    クロージャや関数の引数として使われる変数には
    予め関数呼び出し規約に従って割り当てる
  - mov 命令のソースとして使われる変数には
    予めターゲットと同じレジスタを割り当てる

レジスタ割り当て写像の更新: Coalescing (targeting)



変数の save/restore

- save(x) 命令
  - 変数 x の値をスタックに退避する擬似命令

- restore(x) 命令
  - 変数 x の値をスタックから復帰する擬似命令

save 命令の挿入方針

- save 命令の挿入
  - レジスタ溢れ時
    - レジスタが足りないときに、適当な変数を save
  - 関数呼び出し時
    - 関数を呼び出す前に、生きている変数を save
  - 条件分岐時
    - 2つの分岐先で異なるレジスタ割り当てをされた変数を、条件分岐前に save

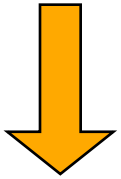
restore 命令の挿入方針

- restore 命令の挿入
  - レジスタ割り当てに存在しない変数を使用する前に、その変数を restore
    - 使用時に変数がレジスタ割り当てに含まれない
      = どこかで save されているということ
      - 変数の定義時にレジスタは割り当て済みのはずなので

レジスタ溢れの例 (1/2)

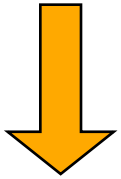
（汎用レジスタは R_0、R_1、R_2 の3つだけとする）

```
let rec f a b =
  let x = -a in let y = -b in x - y - a - b
```



まず、関数の引数に割り当てる

```
let rec f R1 R2 =
  let x = -R1 in let y = -R2 in x - y - R1 - R2
```



次に、最初の変数 x に割り当てる

```
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)

レジスタ割り当てが成功するように、
変数 b を一時的にスタックに退避するようにする

```
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)
```

```
let rec f R1 R2 =
  let R0 = -R1 in save(b); let R2 = -R2 in
  R0 - R2 - R1 - (restore(b); R0)
```

b を R_0 に restore した

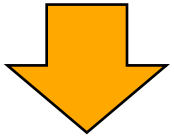
関数呼び出しの例

```
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)
    (* RegMap = { x → R1, y → R2 } *)
  else (restore(y); y) - (restore(x); x)
    (* RegMap = { x → R2, y → R1 } *)
in a + x + y
```

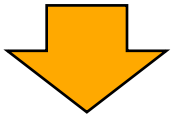
合流後のレジスタ割り当ては？

条件分岐の例 (2/2)

合流時にレジスタ割り当てが異なっている変数は RegMap から削除する

（合流後に改めて restore することになる）

```
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)
    (* RegMap = { x → R1, y → R2 } *)
  else (restore(y); y) - (restore(x); x)
    (* RegMap = { x → R2, y → R1 } *)
in a + (restore(x); x) + (restore(y); y)
```

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

- MinCaml で使われている別のアルゴリズム
- どうせ後で save することになる変数は、定義の直後に save してしまう
  - ややこしいので参考程度に…

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

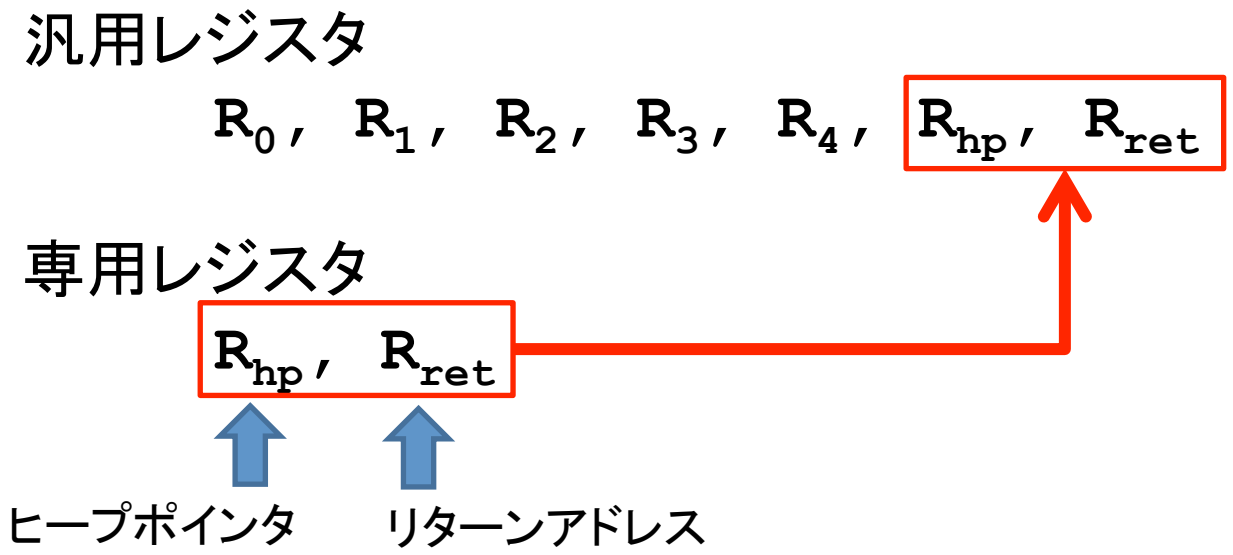
- 変数 x の退避が必要になったら、
  - 現在の位置に forget 命令を挿入
    - これにより、x のレジスタ割り当てを削除
  - 式を ToSpill コンストラクタに入れて返す
- 退避の必要な変数がなかったら、
  - 式のレジスタ割り当てを行う
  - 結果を NoSpill コンストラクタに入れて返す

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

- ToSpill コンストラクタを受け取ったら、
  - 退避する変数 **x** の定義までさかのぼる
  - 定義の直後に save(**x**) を挿入
  - 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…

- 専用レジスタを減らして汎用レジスタを増やす

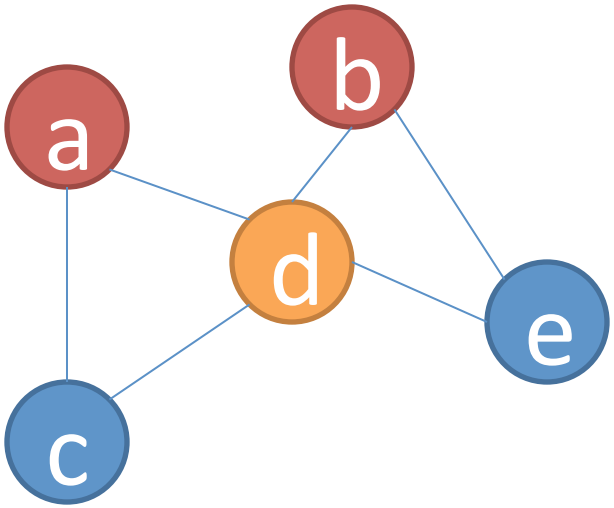


汎用レジスタを増やす方法

- 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, z1, …, zn)`
    - `return (x, h)`
- ヒープポインタをメモリ上におく
  - MakeCls 等が稀ならば得
- リターンアドレスを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`

グラフ彩色問題に還元する例

- 変数をノードとし、同時に生きている変数同士をエッジでつないだグラフを作る
- ノードに色 (=レジスタ) を割り当てる
  - 隣り合うノードが別の色を持つように



整数線型計画問題に還元する例

- 変数 i の生存区間を R_i とする
- R_i^j を以下のように定義する
  - R_i がレジスタ j に割り当てられたとき $R_i^j = 1$
  - そうでないとき $R_i^j = 0$
- 以下の式を満たす解を求める

$$\sum_{i \text{ s.t. } p \in R_i} R_i^j \leq 1 \text{ for all } j \text{ and } p$$

$$\sum_{j} R_i^j = 1 \text{ for all } i$$

ただし p はプログラムポイント

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？

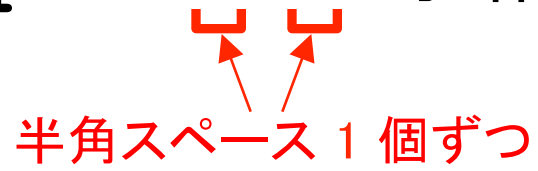
```
let x = … in
let y = (if x <= 0 then f 1 else 2) in
let z = (if y <= 3 then x - 4 else g 5) in
  x - y - z
```

共通課題 (2/2)

- 適度に簡単な再帰関数（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 わざと悪くレジスタ割り当てをした結果の両方を提出してください
    - 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-report-2011@yl.is.s.u-tokyo.ac.jp
- 締め切り: 2 週間後 (11/24) の午後 1 時 (JST)
- Subject: **`Report␣5␣`** <学籍番号: 5桁>



  - 例: **`Report 5 11099`**
- 本文にも氏名と学籍番号を明記のこと

◆ 質問は compiler-query-2011@yl.is.s.u-tokyo.ac.jp まで