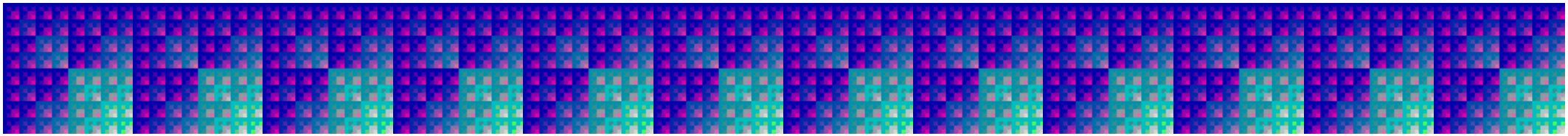
コンパイラ演習
第11回



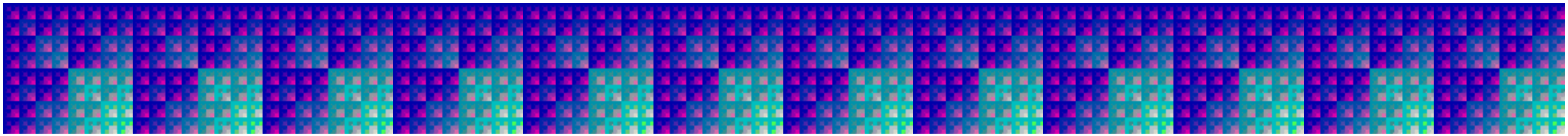
2006/1/19

大山 恵弘　　佐藤 秀明

今回の内容

- バリアント/レコード
  - 表現方法
  - 型付け
- パターンマッチ
  - 型付け
  - switch文への変換
  - 簡単な最適化
  - マッチング漏れ
  - 以降のフェーズでの処理
  - 発展
    - exhaustiveness informationの利用
    - パターン式の拡張

バリアント/レコード



バリアントのメモリレイアウト

- 先頭にタグを追加したタプルのように配置
  - タグ名は整数にエンコード
    - 異なるtypeのタグは同じ整数を使用してもよい
  - 要素数は可変

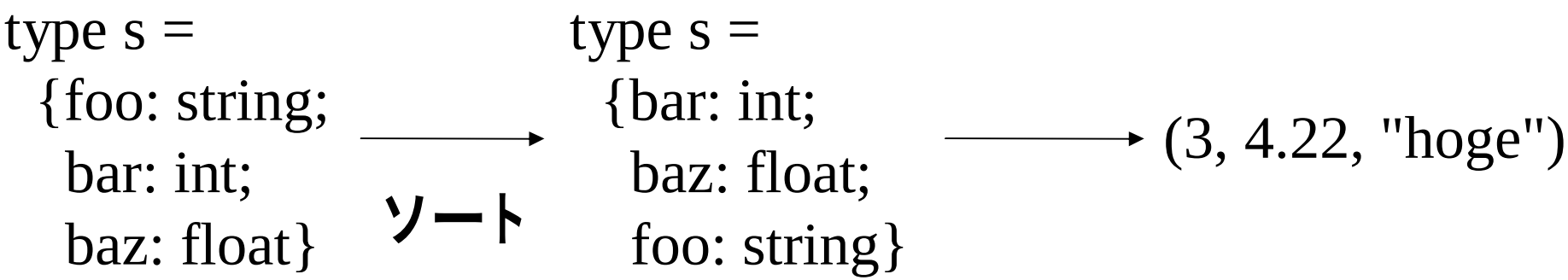
```
type t = A of string * int
       | B
       | C of string
       | ...
```

A → 0
B → 1
C → 2
...

(0, “str1”, 13)
または (1)
または (2, “str2”)
または...

レコードのメモリレイアウト

- ラベル名でソートしたタプルのように配置



中間コード上での表現

- 新たにプリミティブを追加
  - バリアント: 生成/タグ名取得/要素取得
  - レコード: 生成/要素取得
- 既存のプリミティブに吸収してもよい
  - 損をすることもある
    - かえって実装が大変
    - 最適化の精度が落ちる
- 実装方法は各自の判断に任せます

型付け

- タグ名/ラベル名と型情報との関連を管理
  - 外部関数としてまとめておくとよい
  - タグ名/ラベル名から型は一意に決定
    - 複数のtypeで同じ名が使用される場合はエラー

```
type t = A of string * int
       | B
       | C of string
       | ...
```

→

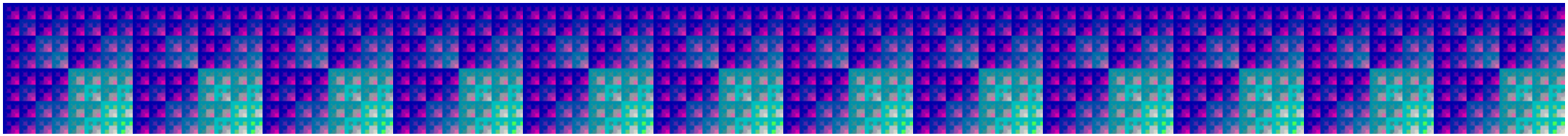
```
TagEnv(A) = (t, [string; int])
TagEnv(B) = (t, [])
TagEnv(C) = (t, [string])
```

```
type s =
  {bar: int;
   baz: float;
   foo: string}
```

→

```
LabelEnv(bar) = LabelEnv(baz) = LabelEnv(foo)
    = (s, [bar→int; baz→float; foo→string])
```

パターンマッチ



パターン式の導入

- 基本的な定義
  - 拡張は後述

p ::=	(パターン式)
| _	(ワイルドパターン)
| c	(定数パターン)
| x	(変数パターン)
| $(p_1, \ldots, p_n)$	(タプルパターン)
| c p	(バリアントパターン)
| $\{l_1=p_1; \ldots; l_n=p_n\}$	(レコードパターン)

パターン式の型付け(1/2)

- 同じ名前の変数はパターン式の中に2回以上出現できない
  - 型システムで保証すると楽

$$\Gamma \mid\!- p : \tau , (B)$$

Γ : 前提となる型環境

p: 型付けするパターン式

τ : pの型

B: pの中で束縛される変数の型環境

パターン式の型付け(2/2)

(変数パターン)

$$\Gamma \vdash x : \tau, (x : \tau)$$

パターン式で束縛された変数の型環境

(タプルパターン)

$$\frac{\Gamma \vdash p_1 : \tau_1, (B_1) \quad \ldots \quad \Gamma \vdash p_n : \tau_n, (B_n) \qquad \text{for all } i \neq j,\ \mathrm{varname}(B_i) \cap \mathrm{varname}(B_j) = 0}{\Gamma \vdash (p_1, \ldots, p_n) : \tau_1 \times \ldots \times \tau_n, (B_1, \ldots, B_n)}$$

タプルの各要素で束縛された変数の名前が重複していないことを確認

match文の導入

- **基本的な定義**
  - **拡張は後述**

e ::= (**式**)

...

match e with (**パターンマッチ**)

p_1 -> e_1

| ...

| p_n -> e_n

match文の型付け

- パターン式が束縛した変数の型環境を利用

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash p_1 : \tau, (B_1) \qquad \Gamma, B_1 \vdash e_1 : \tau' \qquad \cdots \qquad \Gamma \vdash p_n : \tau, (B_n) \qquad \Gamma, B_n \vdash e_n : \tau'}{\Gamma \vdash \text{match } e \text{ with } p_1 \text{->} e_1 \mid \ldots \mid p_n \text{->} e_n : \tau'}$$

switch文への変換

- match文を低レベルなプリミティブに分解
  - 定数とのマッチング→switch
  - 変数束縛→let
- 一見うまくいきそうだが…

```
e ::=                          (式)
    ...
    switch e with              (switch)
      c1 -> e1
    | ...
    | cn -> en
    | _ -> e'
```

ナイーブに変換すると

```
match e with
  (1, 2, v, 4) -> e1
| (_, _, _, _) -> e2
```



```
let (x1, x2, x3,x4) = e in
switch x1 with
    1 ->
     (switch x2 with
        2 ->
          let v = x3 in
          (switch x4 with
             4 -> e1
           | _ -> error)
      | _ -> error)
| _ -> e2
```

- errorならばe2に実行が移ることをどう表現するか?
  - 処理の巻き戻し

いろいろな案

- マッチングの成否をoption型で表現して返す
  - match e_1 with Success(result) -> result |Failure -> e_2
  - バリアントの生成/展開を毎回行うコスト
- マッチングの失敗を例外として処理
  - try e_1 with MatchFailure -> e_2
  - 例外処理によるオーバヘッド増加
- errorの後にすることを継続として渡す
  - クロージャ作成/関数呼び出しが重い
- errorの後にすることをインライン展開
  - errorが複数存在するときはコード量爆発
    - 最適化のフェーズでやるべきことかも

解決策: 専用プリミティブの追加

- fail_match
  - マッチングに失敗
- handle_match e_1 e_2
  - e_1の評価中fail_matchに到達したらe_2の評価にジャンプ
- 手続き型的な挙動を表現
  - 大域的脱出みたいなもの
  - ただのジャンプだから軽い

```
let (x1, x2, x3,x4) = e in
handle_match
  (switch x1 with
      1 ->
        (switch x2 with
            2 ->
              let v = x3 in
              (switch x4 with
                  4 -> e1
                | _ -> fail_match)
          | _ -> fail_match)
    | _ -> fail_match)
e2
```

バリアント/レコードの処理

- タプル同様に展開して各要素をマッチング
  - バリアントの場合はタグも展開してマッチング
    - タグは整数定数と同じように扱える

```
match e with
    {a=1; b=2} -> e1
  | {a=2; b=3} -> e2
```



```
let (a, b) = fields(e) in
switch a with
    1 -> (switch b with
              2 -> e1
            | _ -> fail_match)
  | 2 -> ...
  | _ -> fail_match
```

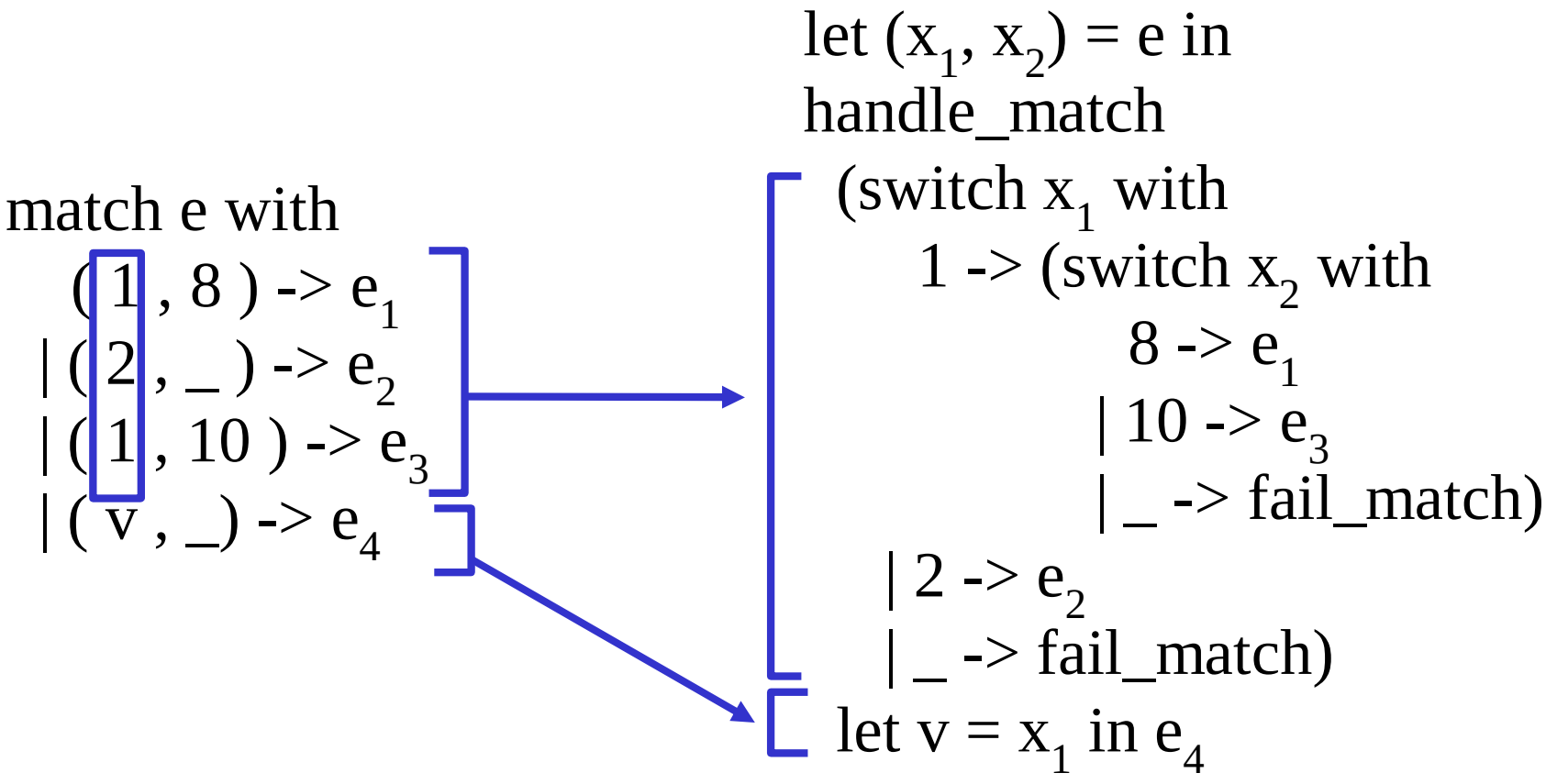
```
match e with
    A(1, 2) -> e1
  | B -> e2
```



```
let t = tag(e) in
switch t with
    A -> let (x1, x2) = fields(e) in
           (switch x1 with ...)
  | B -> e2
  | _ -> fail_match
```

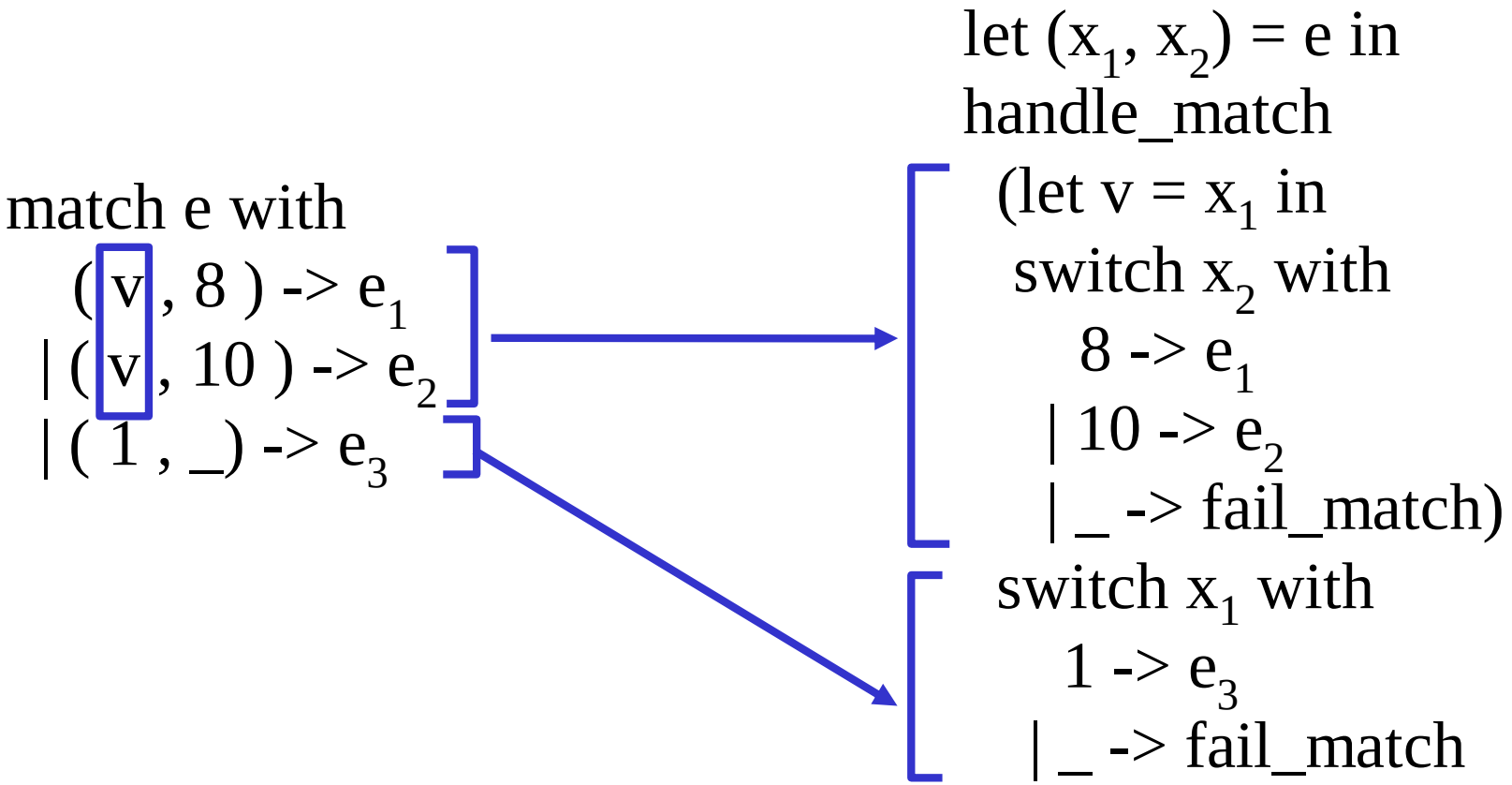
簡単なマッチング最適化(1/2)

- 先頭パターンから定数が連続するとき
  - 連続する定数の種類ごとにまとめてマッチング
  - 定数が連続する部分の候補に結局マッチしなかったらそれ以降の候補にジャンプ



簡単なマッチング最適化(2/2)

- 先頭パターンから同じ名前の変数が連続するとき
  - まとめて変数束縛
    - 連続するのがワイルドパターンなら束縛すら不要



マッチング漏れ

- handle_matchで捕捉されないfail_match
  =マッチするパターンがない場合
- 対処法の例
  - コンパイル時に警告だけ表示
    - 実行時の異常終了を実現するしくみが必要
    - 例外処理の枠組みで対処してもよい
  - マッチング漏れがあればコンパイルを通さない

以降のfail_matchの扱い

- 生きている変数
  - FV(fail_match)=(以降の処理で生きている変数)
- スタック/レジスタ割り当ての状態
  - fail_matchに到達したら破棄してよい
  - 次の処理を追跡する前に状態を巻き戻す
- アセンブリ生成
  - その後に実行するコードへジャンプ

switchのアセンブリ生成

- caseの数による
  - 2〜3個程度ならif-then-elseの連鎖でよい
  - それ以上ならジャンプテーブルを作成

```
switch x with
    0 -> e0
  | 1 -> e1
  | 2 -> e2
  | ...
```



```
  mul x, 4, y
  ld [table+y], z
  b z
  nop
   ...
table:
  .word label_e0
  .word label_e1
  .word label_e2
   ...
label_e0:
  (e0の処理)
label_e1:
  (e1の処理)
label_e2:
  (e2の処理)
   ...
```

発展: exhaustiveness information

- 型情報やマッチングの文脈から
  いろいろなことが分かる
  - マッチングが必ず成功するパターン
  - マッチングが必ず失敗するパターン
  - 絶対に使用されないパターン
  - 評価結果を変えない判定順序入れ替え
    - 行: 各マッチング候補間の判定順序
    - 列: パターン中の各要素間の判定順序
- コンパイル時に活用可能
  - コード最適化
  - 冗長/不完全なmatch式について警告を出す

コード最適化の例(1/3)

- 最適化しないと

```
match x0, y0 with
   [], _ -> 1
 | _, [] -> 2
 | _::_, _::_ -> 3
```



```
let (x, y) =
  tag(x0), tag(y0) in
handle_match
  (switch x with
      [] -> 1
    | _ -> fail_match)
  (handle_match
    (switch y with
        [] -> 2
      | _ -> fail_match)
    (switch x with
        (::) ->
          (switch y with
              (::) -> 3
            | _ -> fail_match)
      | _ -> fail_match))
```


リストは[]、(::)をタグにもつバリアントとみなす

コード最適化の例(2/3)

match文の第2・第3
パターンを入れ替え

絶対に使われない
パターンを削除

```
let (x, y) =
  tag(x0), tag(y0) in
handle_match
  (switch x with
       [] -> 1
     | (::) ->
         (switch y with
              (::) -> 3
            | _ -> fail_match)
     | _ -> fail_match)
  (switch y with
       [] -> 2
     | _ -> fail_match)
```

```
let (x, y) =
  tag(x0), tag(y0) in
handle_match
  (switch x with
       [] -> 1
     | (::) ->
         (switch y with
              (::) -> 3
            | _ -> fail_match))
  (switch y with
       [] -> 2)
```

コード最適化の例(3/3)

冗長なマッチング判定を削除 → **サイズの小さい式をインライン展開** → **不要なhandle_matchを除去**

```
let (x, y) =
  tag(x0), tag(y0) in
handle_match
  (switch x with
      [] -> 1
    | _ ->
       (switch y with
           (::) -> 3
         | _ -> fail_match))
  2
```

```
let (x, y) =
  tag(x0), tag(y0) in
handle_match
  (switch x with
      [] -> 1
    | _ ->
       (switch y with
           (::) -> 3
         | _ -> 2))
  2
```

```
let (x, y) =
  tag(x0), tag(y0) in
switch x with
    [] -> 1
  | _ ->
      (switch y with
          (::) -> 3
        | _ -> 2)
```

最適化の適用例:

最大反復回数を定め、変化がなくなるまで各フェーズを反復

発展: パターン式の拡張

- $p_1 \mid \cdots \mid p_n$
  - $p_1, \ldots, p_n$ はすべて同じ型/変数束縛を持つ
    - 型システムで保証するとよい
- p when e
  - eが不成立ならfail_match
- p_1 as p_2
- let p = e 、let rec f $p_1 \ldots p_n$ = e など

共通課題

- 次のmatch文をswitch文に変換せよ。
  - 「最良」と思われる変換結果を提出せよ
    - 「最良」の定義は各自で設定すること
  - 「処理の巻き戻し」をどう表現するかは自由

```
match e with
    (_, A(4)) -> e1
  | (_, A(5)) -> e2
  | (1, B) -> e3
  | (3, B) -> e4
  | (1, A(2)) -> e5
  | (1, A(x)) -> e6
  | (y, _) -> e7
```

ただし第2要素の持ちうるタグはAとBのみ

コンパイラ係用選択課題

- パターンマッチを自作コンパイラに実装せよ。
  - 以下のパターンを扱えるようにすること
    - ワイルドパターン
    - 定数パターン
    - 変数パターン
    - バリアントパターン
  - マッチング漏れに「対処」すること
  - その他の実装方針は自由に決めてよい

課題の提出先と締め切り

- 提出先: compiler-enshu@yl.is.s.u-tokyo.ac.jp
- 共通課題の締め切り:
  2週間後 (2/2) の午後1時
- コンパイラ係用課題の締め切り:
  2006年3月31日
- Subject: report 11 <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

参考文献

- Caml Lightでのパターンマッチ実装法
  - “The ZINC experiment: an economical implementation of the ML language”
    - http://pauillac.inria.fr/~xleroy/publi/ZINC.ps.gz
- OCamlが採用したパターンマッチ最適化の手法
  - “Optimizing Pattern-Matching”
    - http://pauillac.inria.fr/~maranget/papers/opt-pat.ps.gz
- パターンマッチ最適化のheuristics
  - “When Do Match-compilation Heuristics Matter?”
    - http://www.cs.virginia.edu/~techrep/CS-2000-13.pdf