コンパイラ実験
第２回

江口 慎悟
押川 広樹
塚田 武志

今日の内容

- 型推論（type inference）
- K正規化（K-normalisation）
- α変換（alpha-conversion）
- いくつかの最適化
  - インライン化（inlining）
  - 定数畳み込み（constant folding）
  - 不要な束縛の除去

今日の内容

- 型推論（type inference）
- K正規化（K-normalisation）
- α変換（alpha-conversion）
- いくつかの最適化
  - インライン化（inlining）
  - 定数畳み込み（constant folding）
  - 不要な束縛の除去

MinCaml における型推論

- typing.ml で定義
- 部分式の型を積極的に単一化
  - 例：`e1 = e2`
    - `e1` と `e2` の型を単一化。式全体は `bool`
- `let` および `let rec` では型環境を拡張

MinCaml 特有の型推論

- 型環境に含まれない変数は外部変数
  - typing.ml における Var(x) の処理
- 残った型変数は int とする
  - typing.ml における deref_typ
- 単相型（＝型多相はない）
  - `let f x = x in (f 1, f 1.2)` は型エラー

今日の内容

- 型推論（type inference）
- K正規化（K-normalisation）
- α変換（alpha-conversion）
- いくつかの最適化
  - インライン化（inlining）
  - 定数畳み込み（constant folding）
  - 不要な束縛の除去

K正規形

- 計算の中間結果が
  すべて変数として明示化された形

```
let a = 1 + 2 in
let b = a – 3 in
let c = b + 4 in
let d = c – 5 in
  d
```

(cf. `1 + 2 – 3 + 4 - 5`)

- ML Kit というコンパイラの中間形式

K正規化

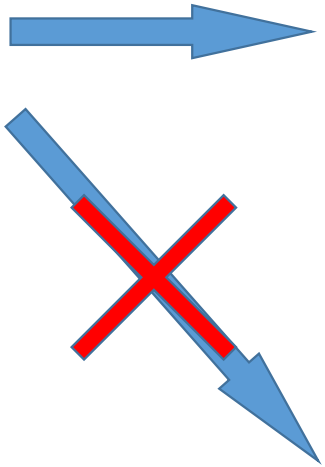
- 部分式を明示的に変数に束縛するように変換
  - MLコードをアセンブリコードに近づける
  - MinCaml では kNormal.ml

MinCaml における工夫

o 最初から変数になっている部分は そのまま

- insert_let 関数を参照

```
123 - x
```



```
let i1 = 123 in
  i1 - x
```

```
let i1 = 123 in
let i2 = x in
  i1 - i2
```

今日の内容

- 型推論（type inference）
- K正規化（K-normalisation）
- α変換（alpha-conversion）
- いくつかの最適化
  - インライン化（inlining）
  - 定数畳み込み（constant folding）
  - 不要な束縛の除去

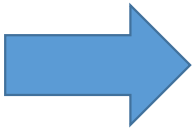
α変換

- 束縛変数の名前変え

$$\lambda x.\, M \quad \longmapsto \quad \lambda y.\, (M[y/x])$$

- (今の文脈では) 束縛変数を異なるものにすること
  - 以降の処理が楽になる

```
let x = 1 in
let x = x + 1 in
  x
```



```
let x0 = 1 in
let x1 = x0 + 1 in
  x1
```

今日の内容

- 型推論（type inference）
- K正規化（K-normalisation）
- α変換（alpha-conversion）
- いくつかの最適化
  - インライン化（inlining）
  - 定数畳み込み（constant folding）
  - 不要な束縛の除去

インライン化の目的

- 関数呼び出しのオーバーヘッドの削減
  - オーバーヘッド

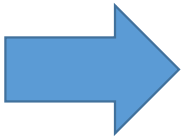
    [呼び出し時]
    　　レジスタ退避、スタックポインタ更新、ジャンプ

    [戻り時]
    　　ジャンプ、スタックポインタ更新、レジスタ復帰
- 「不要な変数の解析」等が精緻に (or 容易に)

インライン化

o 関数呼び出しを その関数の本体で置換

```
let f x y = M in
  … (f a b) …
```



```
let f x y = M in
  … (M [a/x, b/y]) …
```

- 適切にα変換をする必要あり

o むやみなインライン化にはデメリットも

- コードサイズの爆発
- コンパイルの非停止

今日の内容

- 型推論（type inference）
- K正規化（K-normalisation）
- α変換（alpha-conversion）
- いくつかの最適化
  - インライン化（inlining）
  - 定数畳み込み（constant folding）
  - 不要な束縛の除去

定数畳み込み

o 演算のオペランドが「明らかに」定数なら
コンパイル時に計算

```
let x = 1 in
let y = 2 in
  x + y
```

```
let x = 1 in
let y = 2 in
  3
```

- インライン化で適用が促進される

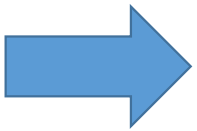
今日の内容

- 型推論（type inference）
- K正規化（K-normalisation）
- α変換（alpha-conversion）
- いくつかの最適化
  - インライン化（inlining）
  - 定数畳み込み（constant folding）
  - 不要な束縛の除去

不要な束縛の除去（関数）

o 無駄な関数定義を除去

```
let rec f x y = M in L
```



```
L
```

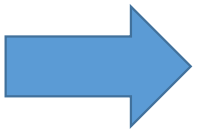
- f の定義が無駄とは…
  - L 中に出現しない

インライン化、定数畳み込みで適用が促進される

不要な束縛の除去（変数）

o 無駄な let を除去

```
let x = M in L
```



```
L
```

- x の定義が無駄とは…
  - L 中に出現しない
  - M は副作用を持たない
    - 副作用の有無を正確に判定することは困難（実は不可能）
    - 「配列への書き込み」「関数呼び出し」がないことが十分条件
  - M は停止する

インライン化、定数畳み込みで適用が促進される

今日の内容

- 型推論（type inference）
- K正規化（K-normalisation）
- α変換（alpha-conversion）
- いくつかの最適化
  - インライン化（inlining）
  - 定数畳み込み（constant folding）
  - 不要な束縛の除去

最適化の繰り返し

- 最適化により変化がなくなるまで最適化処理を繰り返す
  - ただし一定の回数を超えたら打ち切る

```
let rec iter n e = Format.eprintf "iteration %d@." n;
  if n = 0 then e else
  let e' = Elim.f (ConstFold.f
                     (Inline.f (Assoc.f (Beta.f e))))
  if e = e' then e else iter (n - 1) e'
```

各処理の詳細

- 住井英二郎「MinCaml の擬似コード」
  - http://esumii.github.io/min-caml/min-caml.pdf

- コードを読むのもよい勉強になる

レポート課題２

締切：2018/10/18 13:00(JST)

共通課題

３問中２問以上を解答のこと

問 1

束縛変数の名前が全て異なるように

o 下の式をK正規化し適切にα変換せよ

- 前述の「MinCamlにおける工夫」以外は **最適化を行わないこと**

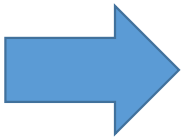
o 同じ式をA正規化（次ページ参照）し適切にα変換をせよ

```
let rec x x =
  let x = let x = x - (-x) in
          x - (let x = -x in x - (-x)) in
  x - (-x) in
let x = x 125 in x - (- x)
```

A正規形

- 式を計算する順序に並べたもの
  - let のネストも許さない
- K正規化の後に let を平坦化して得られる

```
let x =
  (let y = M in N)
in
  L
```



```
let y = M in
let x = N in
  L
```

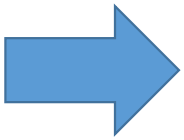
  - ただし L は y の自由な出現を含まない
    - 適宜α変換する

Flanagan, Sabry, Duba, Feleisen.
"The Essence of Compiling with Continuations". In PLDI 1993.

問2

- 共通部分式削除を実装せよ
  - K正規形の式に同じ式が現れたら最初の式を利用
  - この処理がプログラムの意味を変えないための条件を考察せよ

```
let a = x + y in
let b = x + y in
  L
```



```
let a = x + y in
let b = a in
  L
```

問２：追加条件

- 共通部分式削除が行われたことを確認できるようにせよ
  - 第１回課題のように中間結果を出力する

- この時点では最適化の影響の考察は不要
  - 処理速度の向上に寄与するかは考えなくてよい

問３

再帰的に定義されている変数の

- インライン化の際にα変換を行わないと
  プログラムの意味が変わる例を挙げよ
  - 変換元のプログラムは
    既に適切にα変換されているとする
- インライン化、定数畳み込み等の最適化を
  繰り返し行った場合について
  「コードサイズは増えないが
  回数制限がないと止まらない」
  ような例を挙げよ
  - 最適化前のプログラムの評価は停止するものに限る