

関数・論理型プログラミング実験 第9回

江口 慎悟
押川 広樹
塚田 武志

講義のサポートページ

<http://www.kb.is.s.u-tokyo.ac.jp/~tsukada/cgi-bin/m/>

- 講義資料等が用意される
- レポートの提出先
- 利用にはアカウントが必要
- 名前/学籍番号/希望アカウント名をメールを
tsukada@kb.is.s.u-tokyo.ac.jp
までメールしてください。
 - 件名は「FL/LP実験アカウント申請」
 - アカウント名/パスワードを返信
 - PCからのメールを受け取れるように

インタプリタを作る（全5回）

第5回 基本的なインタプリタの作成

- 字句解析・構文解析、変数の扱い方

第6回 関数型言語への拡張

- 関数定義・呼び出し機構の作成

第7回 型システムと単純型推論

- 単純型検査器

第8回 単一化、let多相

- 単一化の定義とアルゴリズム、let多相

第9回 様々な拡張

- パターンマッチング

今日の内容

○ パターンマッチング

- 評価
- 型検査

○ 評価戦略

- 値呼び、名前呼び、必要呼び

パターンマッチング

評価

型推論

制約の生成

制約の解決

式の構文

○パターンマッチ式

$\text{match } e \text{ with } p_1 \rightarrow e_1 \mid \dots \mid p_n \rightarrow e_n$

○パターン (整数、真偽値、変数、組、リスト)

$p ::= \theta \mid 1 \mid 2 \mid \dots$
 $\mid \text{true} \mid \text{false}$
 $\mid x$
 $\mid (p, p)$
 $\mid [] \mid p :: p$

型

- 組型とリスト型を加える

$$\text{ty} ::= \text{Int} \mid \text{Bool} \mid \text{ty} \rightarrow \text{ty} \mid \alpha$$
$$\mid \text{ty} * \text{ty} \mid \text{ty list}$$

パターンマッチング

評価

型推論

制約の生成

制約の解決

match式の評価

○ $\text{match } e \text{ with } p_1 \rightarrow e_1 \mid \dots \mid p_n \rightarrow e_n$

1. 式 e を評価し、値 v を得る
2. v とパターン p_1 を照合：
 - 照合すれば環境に結果を追加し e_1 を評価
3. ...
4. v とパターン p_n を照合：
 - 照合すれば環境に結果を追加し e_n を評価
5. いずれとも照合しなかったので、エラー

パターン照合

入力：パターンと値

出力：照合の成否と（成功した場合）束縛

- 束縛は、変数から値へのマッピング

例：

- 1 と 0 \Rightarrow 失敗
- 1 と 1 \Rightarrow 成功 { }
- x と 1 \Rightarrow 成功 { x=1 }
- (x, y) と (1, (2, 3)) \Rightarrow 成功 { x=1, y=(2,3) }
- (x, 1) と (2, 3) \Rightarrow 失敗

パターンマッチング

評価

型推論

制約の生成

制約の解決

制約の生成

- $\text{match } e \text{ with } p_1 \rightarrow e_1 \mid \dots \mid p_n \rightarrow e_n$
 - 式 e の型 t と制約 c を求める
 - 各 i について
 - パターン p_i の型と制約 c_i 、および追加される型環境 Γ_i (後述) を求める
 - 現在の型環境に Γ_i を追加した型環境の下で、 e_i の型 t'_i と制約 c'_i を求める
 - 新しい型変数 α を導入
 - match 式の型は α 、制約は $\{ t=t_1=\dots=t_n, \alpha=t'_1=\dots=t'_n \} \cup c \cup c_1 \cup \dots \cup c_n \cup c'_1 \cup \dots \cup c'_n$

制約の生成：定数パターン

- 定数パターン 1

- 型 int、制約 {}, 追加される型環境 {}

- 定数パターン true

- 型 bool、制約 {}, 追加される型環境 {}

制約の生成：変数パターン

○ 変数パターン x

- 新しい型変数 α を導入する
- 型 α 、制約 $\{ \}$ 、追加される型環境 $\{ x = \alpha \}$

制約の生成：組パターン

○ 組パターン (p_1, p_2)

- パターン p_i の型を t_i 、制約を c_i 、追加される型環境を Γ_i とする
- 型 $t_1 * t_2$ 、制約 $c_1 \cup c_2$ 、追加される型環境 $\Gamma_1 \cup \Gamma_2$

制約の生成：リストパターン

○ ニルパターン []

- 型変数 α を導入
- 型 α list、制約 $\{ \}$ 、追加される型環境 $\{ \}$

○ コンスパターン $p_1 :: p_2$

- パターン p_i の型を t_i 、制約を C_i 、追加される型環境を Γ_i とする
- 新しい型変数 α を導入
- 型 α list、制約 $\{ \alpha = t_1, \alpha \text{ list} = t_2 \} \cup C_1 \cup C_2$
追加される型環境 $\Gamma_1 \cup \Gamma_2$

パターンマッチング

評価

型推論

制約の生成

制約の解決

制約の解決

- 前回と同じ unification algorithm で行える
- ただし、次のケースを追加

$$\text{unify} (\{ s*t = s'*t' \}UC) = \text{unify} (\{ s=s', t=t' \}UC)$$

$$\text{unify} (\{ t \text{ list} = t' \text{ list} \}UC) = \text{unify} (\{ t=t' \}UC)$$

評価戦略

値呼び (Call-by-Value)

名前呼び (Call-by-Name)

必要呼び (Call-by-Need)

評価可能な部分式

- 一般的には複数個ある

fst (1+2, 3+4)

- どの順序で評価すべきか？
これを決めるのが**評価戦略**

評価戦略の影響

- 性質の良い式については
評価戦略は評価結果に影響しない
 - 性質が良い = 副作用がなく、評価が止まる
- 逆に言うと、評価戦略が影響するのは
 - 評価が止まるかどうか
 - 副作用の起こる順序
 - 副作用によっては評価結果も変わる

主な評価戦略

- 値呼び (call by value)
- 名前呼び (call by name)
- 必要呼び (call by need)
 - 他の戦略
 - normal order, applicative order など

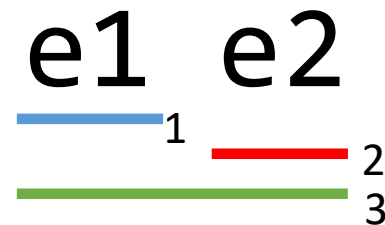
評価戦略

値呼び (Call-by-Value)

名前呼び (Call-by-Name)

必要呼び (Call-by-Need)

値呼び



- 関数適用の前に引数を評価

$(\text{fun } x \rightarrow x * x) (2 + 3)$

$\longrightarrow (\text{fun } x \rightarrow x * x) 5$

$\longrightarrow 5 * 5$

$\longrightarrow 25$

- 今まで実装していたのはこれ

値呼びの利点

- 関数呼び出しにオーバーヘッドがない
 - 特に加算や減算などの組み込み演算
 - 整数値や浮動小数点値などを直接使える
- 評価順序が分かりやすい
 - 副作用を扱いやすい
 - 参照、I/O, ...

値呼びの欠点

- 他の評価戦略なら計算が止まるのに値呼びだと止まらない場合がある
 - 例： `fst (5, loop ())`
- `if` や `||`, `&&` を関数で表現できない
- 展開・折り畳みによる最適化と相性が悪い
 - `fst (x, y) = x` という定義であっても `fst (e1, e2)` を `e1` に変換することができない

評価戦略

値呼び (Call-by-Value)

名前呼び (Call-by-Name)

必要呼び (Call-by-Need)

名前呼び

e1 e2
1 2

- 関数適用を引数より先に評価

`(fun x -> x * x) (2 + 3)`

$\longrightarrow (2 + 3) * (2 + 3)$

$\longrightarrow 5 * (2 + 3)$

$\longrightarrow 5 * 5$

$\longrightarrow 25$

- 式は必要になるまで評価しない

名前呼びの利点

- 計算が止まりやすい
 - 他の評価戦略で止まるなら、名前呼びで止まる
- if や ||、&& を関数として実現できる
 - 「マクロ」的な機能を実現可
- 不必要な計算を避けられる
 - 例：hd (sort xs) が $O(\text{xs の長さ})$ で計算できる
- 展開・折り畳みによる最適化と相性が良い

名前呼びの欠点

- 関数呼び出しのオーバーヘッドが大きい
 - 次のスライド（実装法）も参照
- 同じ式を何度も評価
 - 例： $(\text{fun } x \rightarrow x * x) (2 + 3)$ の $(2+3)$
- 式の評価順序やタイミングの制御が困難
 - 副作用との相性が悪い

名前呼びの実装 (1/2)

- サンク (thunk) を使う
 - サンクとは 式 と 環境 の組 (cf. クロージャ)
- 環境には値ではなくサンクを保存
 - 変数に束縛されるのは、値ではなく、式だから

```
type value = ...  
  and env = (var * thunk) list  
  and thunk = Thunk of expr * env
```

名前呼びの実装 (2/2)

- 関数適用 $e1\ e2$ の環境 env での評価
 - 環境 env で $e1$ を評価し $v1$ とする
 - $v1$ はクローージャ $\langle \text{fun } x \rightarrow e, env' \rangle$ であるはず
 - サंक $\langle e2, env \rangle$ を作る
 - 式 e を環境 $(x, \langle e2, env \rangle) :: env'$ の下で評価
- 変数 x の環境 env での評価
 - x に対応するサंक $\langle e, env' \rangle$ を env から探す
 - 式 e を環境 env' の下で評価する

評価戦略

値呼び (Call-by-Value)

名前呼び (Call-by-Name)

必要呼び (Call-by-Need)

必要呼び

e1 e2
1 2

- 式は必要になるまで評価しないが
評価結果は共有する

(fun x -> x * x) (2 + 3)

→ x * x {x=2+3}

→ x * x {x=5}

→ 5 * 5 {x=5}

→ 25

- 2 + 3 の評価は一度だけ

必要呼びの利点・欠点

- 必要呼びの利点 =
「名前呼び」の利点
+ 評価回数が高々「値呼び」程度
- 必要呼びの欠点 =
「名前呼び」の欠点
- 同じ式を何度も評価
+ オーバーヘッドがさらに大きく

必要呼びの注意

○ 同じ式を二度評価しないわけではない

- 例：素朴な fib は指数時間

```
let rec fib n =  
  if n < 2 then 1  
  else fib (n-1) + fib (n-2)
```

- 二度評価しないのは
変数を通して共有されている式のみ

- OK: `let x = 2 + 3 in let y = x in y * x`
- NG: `(2 + 3) * (2 + 3)`

必要呼びの実装 (1/2)

- 環境で「サンクまたは値」への参照を保存
 - 最初はサンクで保存
 - サンクが評価されたら、評価結果の値に書き換え

```
type value = ...  
  and env = (var * dval ref) list  
  and dval = DThunk of expr * env  
             | DVal   of value
```

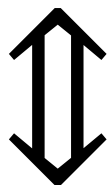
必要呼びの実装 (2/2)

- 関数適用 $e1\ e2$ の環境 env での評価
 - 環境 env で $e1$ を評価し $v1$ とする
 - $v1$ はクローージャ $\langle \text{fun } x \rightarrow e, env' \rangle$ であるはず
 - サンクへの参照 ref ($DThunk(e2, env)$) を作る
 - これを r とする
 - 式 e を環境 $(x, r) :: env'$ の下で評価
- 変数 x の環境 env での評価
 - x に対応する「サンクまたは値」への参照を探す
 - 参照の中身が値なら、その値を返す
 - 参照の中身がサンクなら、評価して値 v を得て、 v で参照を書き替え、 v を返す

補足：let 式の評価順序

- 関数適用の評価順序と同じになるようにする

`let x = e1 in e2`



`(fun x -> e2) e1`

補足

- 紹介した評価戦略は主に関数適用の話
 - 例えば組 $(e1, e2)$ の評価順序は別の話
- `if e1 then e2 else e3` の評価順序は
(紹介した三つの) どの評価戦略でも同じ
- `match` 式も同様
 - ただし `match e with ..` が `e` をどこまで評価するかには、いくつかの選択肢がある
 - `match (print "hello"; 1), 2 with (x,y) -> y`
が文字列 "heloo" を出力するかどうか

例題

理解の確認をするための課題です

課題提出システム上での提出の必要はありません

例題を解きTAに見せることで出席とします

分からないことがあったら、積極的に質問しましょう

例題

- 値を表す型を以下のように定める

```
type name = string
```

```
type value =
```

```
| VInt      of int
| VBool     of bool
| VFun      of name * expr * env
| VRecFun   of name * name * expr * env
| VPair     of value * value
| VNil
| VCons     of value * value
```

```
and env = (name * value) list
```

例題

- パターンを表す型を以下のように定める

```
type pattern = PInt of int | PBool of bool
              | PVar of name
              | PPair of pattern * pattern
              | PNil | PCons of pattern * pattern
```

- パターンを値と照合する関数

```
find_match : pattern -> value ->
              (name * value) list option
```

を与えよ

レポート課題 9

締切：2018/6/26 13:00(JST)

問 1

- 前回の課題のインタプリタを拡張し、組型とリスト型の値を扱えるようにせよ
 - 構文は以下をサポートすること
 - (式, 式)
 - []
 - 式 :: 式
 - 参考資料の ex1 に適当な eval.ml を与えればよい
 - 型推論は問 3

問 2

- 問 1 のインタプリタを改良し、
パターンマッチングを扱えるようにせよ
- 参考資料の ex2 に適当な eval.ml を与えればよい

問 3

- 問 2 のインタプリタに型推論を導入せよ

問 4

- インタプリタの評価戦略を名前呼びにせよ
 - パターンマッチはサポートしなくてよい

発展 1

- インタプリタの評価戦略を必要呼びにせよ
 - パターンマッチはサポートしなくてよい
 - 式が繰り返し評価されていないことを確認せよ
- 例えば

`(fun x -> (fun y -> x * y) x) (2+3)`

における `(2+3)` の評価は一度だけ

発展 2

- インタプリタの機能拡張を試みよ。例えば
 - 便利な糖衣構文の追加
 - 例外機構
 - 第一級継続 (call/cc)
 - rank-2 多相

注意

- 問および発展はまとめて提出してもよい
 - どの問に答えているかは明らかにすること
 - 考察はそれぞれ行うこと
- OCaml の標準ライブラリは自由に用いてよい
- ビルド方法の記述は忘れないように
- Conflict は可能な限り消すこと
- 参考資料のコードを使う必要はない