関数・論理型プログラミング実験
第７回

江口 慎悟
酒寄 健
塚田 武志
松下祐介

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

今日の参考資料

- Benjamin C. Pierce:
  Types and Programming Languages,
  The MIT Press, Cambridge, MA, 2002.
  - 特に 22章 Type Reconstruction
  - 邦訳:
    型システム入門 –プログラミング言語と型の理論–

型システム

型システム

- 式を「型」で分類することで、プログラムが実行時に不正な動作をしないことを検査・保障する仕組み
  - 「型」とは評価結果の値の分類
    - 例： 整数、真偽値、関数 など
- 「式が与えられた型を持つか」の検査を「型検査」と呼ぶ

ML系言語の型システムの特徴

- 静的（static）
  - 型検査は実行前に行われる
    → 実行時のオーバーヘッドがない
- 健全（sound）
  - 型検査が成功したら、そのプログラムは実行時に型エラーを生じない
- 型推論（type inference）
  - プログラマが型を明示せずとも、
    コンパイラが適切な型を推論してくれる

型推論の方法 （今回・次回で紹介するもの）

o 制約生成と制約解決の２ステップから成る

今回

- 制約生成：式から型の方程式（＝制約）を作る
  - 式に型が付くことと、方程式に解があることが同値

```
fun x y z -> (x z) (y z)
```

$$
\left\{
\begin{array}{rl}
X = X_1 \to X_2, & X_1 = Z \\
Y = Y_1 \to Y_2, & Y_1 = Z \\
X_2 = X_{21} \to X_{22}, & X_{21} = Y_2
\end{array}
\right\}
$$

- 制約解決：型の方程式（＝制約）を解く

型推論

素朴なアプローチと困難

型推論アルゴリズム

　　1) 制約の生成

　　2) 制約の解決（単一化アルゴリズム）

型付け規則

- 式がどういう型を持つか判定する規則

例：
- 整数リテラルは `int` 型を持つ
- 真偽値リテラルは `bool` 型を持つ
- 式「`e1 + e2`」は、
  `e1` と `e2` が共に `int` 型を持つなら、
  全体も `int` 型を持つ

簡単な型推論の例

- 式「0」の型は？
  - int
- 式「true」の型は？
  - bool
- 式「1 + 2」の型は？
  - int
- 式「1 + true」の型は？
  - 型を持たない（型エラー）

型推論

素朴なアプローチと困難

型推論アルゴリズム

1) 制約の生成

2) 制約の解決（単一化アルゴリズム）

素朴なアプローチ

- 例について見てみる
  - 定数・組み込み関数
  - `let` と変数
  - 関数適用と抽象

定数・組み込み関数の型推論

o 型は予め与えられている

例：

```
1     :  int 型

true  :  bool 型

not   :  bool -> bool 型
```

let 式の型推論

例： `let x = 1 in x + 2`

i. 1 は int 型。従って変数 x は int 型

ii. x を int 型とすると、x+2 は int 型

型環境

＝変数から型へのマッピング

cf. 評価に用いた環境

変数の型推論

o 型環境で与えられる

<u>例：</u>

{x=int} のとき、　x　:　int 型

{x=bool} のとき　x　:　bool 型

{x=bool} のとき　y　:　型エラー

束縛されてない変数が
出てきているから，

関数適用の型推論

<u>例１：</u> `not true`

i. not は bool -> bool 型。true は bool 型
ii. よって全体は bool 型

<u>例２：</u> `is_zero 1`

i. is_zero は int->bool 型。1 は int 型
ii. よって全体は bool 型

困難：関数抽象の型推論

<u>例：</u> `fun x -> x + 1`

「x+1」の型が関数の戻り値の型となるが、

「x+1」の型検査を始める段階では、

「x」の型が分からない！

変数の型が
わかれば
かんすうの型
わかるやったやん.

かいけつさく

型推論

素朴なアプローチと困難

型推論アルゴリズム

1) 制約の生成

2) 制約の解決（単一化アルゴリズム）

今回紹介する方法

- o 型変数の概念を導入する
- o 推論規則を変更し、型と型制約を返す
- o 制約を解くと、具体的な型が分かる

例：`fun x -> x + 1`

i. x の型は型変数 α

ii. 式全体の結果：

- 型：`α -> int`
- 制約：`α = int`

型推論の流れ

- ステップ１：制約の生成
  - 式の型と、型の満たすべき制約を式の構造に沿って求める
    - 例：式「fun x -> x + 1」に対して、
      型 α -> int と制約 { α = int } を求める

- ステップ２：制約の解決
  - 制約を解き、式の具体的な型を得る
    - 例：{ α = int }を解くと、[α := int]。
      これを α -> int に適用すると int -> int

型推論

素朴なアプローチと困難

型推論アルゴリズム

1) 制約の生成

2) 制約の解決（単一化アルゴリズム）

今回の
かだい

ステップ１：制約の生成

- 式の構造に従って定義
  - 定数
  - 変数
  - let 式
  - if 式
  - 関数抽象
  - 関数適用
  - 再帰関数

制約の生成：定数

- あらかじめ与えられた型、空の制約
  - 1
    - int 型、制約 { }
  - true
    - bool 型、制約 { }
  - not
    - bool -> bool 型、制約 { }

制約の生成：変数

o 環境で与えられた型、空の制約

- 型環境 { x=int } での x
  - int 型、制約 { }

- 型環境 { x=bool } での x
  - bool 型、制約 { }

- 型環境 { x=bool } での y
  - エラー
    - いわゆる Error: Unbound value y

制約の生成：let 式

- `let x = e`$_1$ `in e`$_2$
  - 現在の型環境 `env` で `e`$_1$ の型と制約を求める（それぞれ `t`$_1$ と `C`$_1$ とする）
  - `env` に `x` と `t`$_1$ の対応を追加した環境を `env'` とし、`env'` において `e`$_2$ の型と制約を求める（それぞれ `t`$_2$ と `C`$_2$ とする）
  - let 式全体の型と制約は、`t`$_2$ 型と制約 `C`$_1 \cup$ `C`$_2$

制約の生成：if 式

- if e_1 then e_2 else e_3
  - i = 1, 2, 3 について、
    現在の環境 env において型と制約を求める
    （それぞれ t_i と C_i とする）
  - if 式全体の型と制約は、
    t_2型、制約 $\{t_1 = \mathrm{bool},\ t_2 = t_3\} \cup C_1 \cup C_2 \cup C_3$

制約の生成：関数抽象

o `fun x -> e`

- 新たな型変数 α を導入
- 現在の型環境に x と α の対応を追加し env' とし、env' のもとで e の型と制約を求める（それぞれ t と C とする）
- fun 式全体の型と制約は、`α -> t` 型で制約 C

制約の生成：関数適用

$e_1\ e_2$

- i = 1, 2 について
  現在の型環境 env で e_i の型と制約を求める
  （それぞれ t_i と C_i とする）
- 新たな型変数 α を導入する
- 関数適用式全体の型と制約は、
  α型と、制約 $\{ t_1 = t_2 \text{->} \alpha \} \cup C_1 \cup C_2$

$t_1 = \alpha \to \beta$

$\alpha = t_2$

例：`fun x -> not x`

i. { x = α } を型環境に追加

ii. `not` は `bool -> bool` 型、制約 { }

iii. x は α 型、制約 { }

iv. `not x` は β型、制約 { bool->bool = α->β }

v. `fun x -> not x` は、
   α -> β 型、制約 { bool->bool = α->β }

制約の生成：再帰関数

- `let rec f x = e`$_1$ `in e`$_2$
  - 新たな型変数 α と β を導入する
  - 現在の型環境を env とする。
    型環境 env ∪ { f = α->β, x = α } のもとで
    e_1 の型と制約を求める（t_1型、制約C_1とする）
  - env ∪ { f = α -> β } のもとで e_2 の型と制約を求める（それぞれ t_2 と C_2 とする）
  - 式全体の型と制約は、 t_2 型、制約 $\{ t_1=\beta \} \cup C_1 \cup C_2$

$\alpha \to \beta = \alpha \to t_1$ が出てくる。

型推論

素朴なアプローチと困難

型推論アルゴリズム

1) 制約の生成

2) 制約の解決（単一化アルゴリズム）

ステップ２：制約の解決

o ステップ１で求まった型と制約に対し、
制約を解くことで具体的な型を求める

例：`fun x -> not x` の型と制約は、
α->β型と制約 { bool->bool = α->β }

o 単一化アルゴリズムを用いればよい

（次回に実装する）

まとめ

- ステップ１：制約の収集
  - 式の型と満たすべき制約を、
    式の構造に沿って求める

    例： fun x -> x + 1 に対し、α->int型と制約 { α=int } を得る

- ステップ２：制約の解決
  - 制約を解き、式の具体的な型を得る

    例： { α=int } を解くと、[α:=int]。
    これを α->int に適用して、型 int -> int を得る

補足

- 次のモジュールを参考として配布する
  - 型の構文に関する操作を与える TySyntax
  - 型の制約を解く ConstraintSolver
- mli, cmi, cmo を配布するが ml は配布しない
  - 実装はブラックボックスとして使う
    次回に実装を行う

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

- 次の式から生成される制約を（手で）求めよ

```
fun f -> fun g -> fun x -> f (g x)
```

- 配布した制約解消器を使って制約を解け

レポート課題 7

締切：**2019/6/18** 13:00(JST)

問１

- 第６回の問２のインタプリタを拡張し、
  式を評価する前に型推論を行い、
  型が付くものだけを評価するようにせよ

```
type constraints = (ty * ty) list

type tyenv = (name * ty) list
```
(コマンド実行後の型環境)

```
infer_expr :
    tyenv -> expr -> ty * constraints

infer_cmd :
    tyenv -> cmd -> ty * tyenv
```
←ちがうかも

発展１

Cでの Nullがうめてね みたいなしょ

うえに　g　nullだったら

みたいにすればよいけ?

- 型が付いてもよさそうだが、
  OCaml でも型が付かないプログラムを挙げよ。
  そのプログラムがどうして型が付くべきか、
  OCaml ではなぜ型が付かないかを論ぜよ。