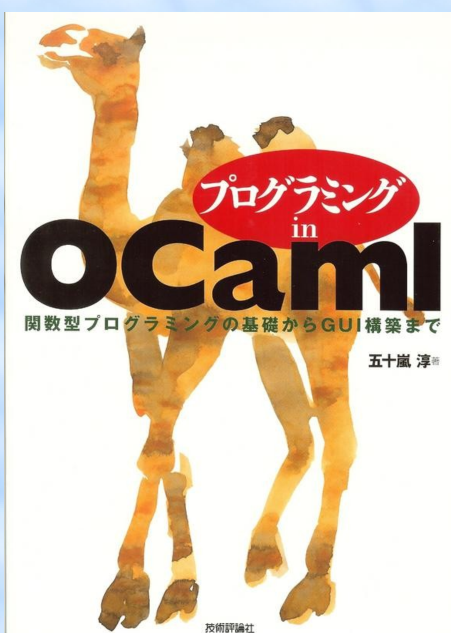
ML型推論の光と影

@平成廿一年東都大駱駝会

京都大学
五十嵐 淳

自己紹介

- 情報科学の研究をしています
- 専門はプログラミング言語とか型理論とか
- 研究のひとつはJavaの改良ですが、Javaでプログラムは書きません(けません)
- ML歴16年、OCaml歴は11年くらい
- 

の著者です

いきなり鶴亀算

OCamlプログラマとラクダが合わせて7匹いる。
足の数が合わせて20本である時、
OCamlプログラマとラクダはそれぞれ何匹いるか。

小学生の解法
全員ラクダだとすると足の数は 4 x 7 = 28 本
実際には20本あるから8本分ラクダが多い
ラクダ一匹をOCamlプログラマに置き換えると
足は2本減るから4人置き換えれば丁度よい
OCamlプログラマ 4匹、ラクダ 3 匹

いきなり鶴亀算

OCamlプログラマとラクダが合わせて7匹いる。
足の数が合わせて20本である時、
OCamlプログラマとラクダはそれぞれ何匹いるか。

中学生の解法

OCamlプログラマの数を x、ラクダの数を y とすると、

$$x + y = 7$$

$$2x + 4y = 20$$

これを解いて $x = 4, y = 3$

中学生の解法のエラいところ

- 未知数の導入
- 未知数を使った式で状況をとにかく表現
- 簡単に解ける問題(連立一次方程式)に帰着

今日のおはなし

ML型推論の仕組み･長所･短所を知る

- MLの型検査と型推論
- ML型推論の仕組み
  - 多相型などにはほとんど踏みこみません
- ML型推論礼賛
- ML型推論に毒づく
- 解毒剤
- まとめ

MLの型検査･型推論

MLの型検査

- プログラムの「つじつまが合っているか」の検査
  - if の条件部には真偽値がくるか
  - 関数の引数は適当か
- プログラム実行前に行われる(静的検査)
- 型検査を通過 ⇒ データの「種類」にまつわるエラーが実行時に発生しないことが**保証**される
  - 0での除算などのエラーはその限りではない

型安全性

型

- プログラム(断片)の分類のためのラベル
  - int型 … 実行結果が整数である(あろう)ような式
  - bool型 … 実行結果が真偽値であるような式
  - int→bool型 … 実行結果が「整数を引数として真偽値を返すような」関数値であるような式
- 分類にあてはまる式が「つじつまの合った式」
  - 1+1 は int 型の式
  - fun x → x > 3 は int → bool 型の式
  - if 3 then 4 else 5 は型が与えられない式

型付け規則

何がつじつまの合った式なのかを型を使って規定

- 整数定数式には int 型を与える
- 式e1とe2ともに int 型が与えられるなら、式 e1+e2 には int 型を与える
- 式e1とe2に、それぞれS→T型、S型が与えられるなら、適用式 e1 e2 にはT型を与える
- xはS型であるという仮定の下で式 e にT型が与えられるなら、fun x→ e にはS→T型を与える

型推論

- 変数に対する型宣言のないプログラムから
  - 変数の型
  - プログラムの型

を知る

fun x → x > 1 の型は?

x は int 型で、
全体は int→bool 型です!

ML型推論の仕組み

Hindley
の写真

J. Roger Hindley (b. 1938)

Milner
の写真

Robin Milner (b. 1934)

Luís Damas (b. 19??)
John Alan Robinson (b. 1930)

問題設定の確認

- 入力: 式(と、これまでに定義された関数などの型)
- 出力: 各変数の型と式の型、または、エラー

基本的なアイデア: 中学生の鶴亀算

- わからないことは変数で表す
  - 型変数(α, β,...)の導入
- 状況をとにかく式で表す
  - 状況: 部分式の型同士の関係
  - 型付け規則の役割
    - 各部分式の型の間に成立すべき関係の規定
    - 具体的な問題全てに共通する背景知識
- 簡単な問題に帰着
  - MLの場合: 一階の単一化(unification)問題

例題(1): fun f → f(3) + 2

- fの型は　α，　各部分式　3,　f(3),　2,　f(3)+2, fun f → f(3)+2 の型を$\beta 1,\ldots,\beta 5$とおく
- 型付け規則から方程式を立てる

整数定数式の型付け規則　⇒　$\beta 1 = \mathrm{int}$
関数呼び出し式の型付け規則 ⇒　$\alpha = \beta 1 \to \beta 2$
整数定数式の型付け規則　⇒　$\beta 3 = \mathrm{int}$
足し算式の型付け規則　⇒　$\beta 2 = \mathrm{int}$, $\beta 3 = \mathrm{int}$, $\beta 4 = \mathrm{int}$
funの型付け規則　⇒　$\beta 5 = \alpha \to \beta 4$

- 解く!
  - $\alpha = \mathrm{int}\to\mathrm{int}$, $\beta 1 = \ldots = \beta 4 = \mathrm{int}$
  - $\beta 5 = (\mathrm{int}\to\mathrm{int})\to\mathrm{int}$

例題(2): fun f → f(3) + f

- fの型は　α,　各部分式　3,　f(3),　f(3)+f, fun f → f(3)+f の型をβ1,...,β4とおく
- 型付け規則から方程式を立てる

整数定数式の型付け規則　⇒　$\beta 1 = \mathrm{int}$
関数呼び出し式の型付け規則 ⇒　$\alpha = \beta 1 \to \beta 2$
整数定数式の型付け規則　⇒　$\alpha = \mathrm{int}$
足し算式の型付け規則　⇒　$\beta 2 = \mathrm{int}$, $\alpha = \mathrm{int}$, $\beta 3 = \mathrm{int}$
funの型付け規則　⇒　$\beta 4 = \alpha \to \beta 3$

- 解く！ ⇒ 解けない！

鶴亀算でいうと

Ocamlプログラマとラクダが合わせて7匹いる。
足の数が合計で20本であった。
レントゲン写真を撮ってみると胃が14個見える。
Ocamlプログラマとラクダはそれぞれ何匹いるか。
ちなみにラクダには胃が3つある。
(生物学的には4つでひとつは退化しているらしい)

OCamlプログラマの数を x、ラクダの数を y とすると、

$$x + y = 7$$

$$2x + 4y = 20$$

$$x + 3y = 14$$

この方程式には解がない。

例題(2): fun f → f(3) + f

- …

整数定数式の型付け規則 ⇒ $\beta 1 = \mathrm{int}$
関数呼び出し式の型付け規則 ⇒ $\alpha = \beta 1 \to \beta 2$
整数定数式の型付け規則 ⇒ $\alpha = \mathrm{int}$
足し算式の型付け規則 ⇒ $\beta 2 = \mathrm{int},\ \alpha = \mathrm{int},\ \beta 3 = \mathrm{int}$
funの型付け規則 ⇒ $\beta 4 = \alpha \to \beta 3$

- 解く! ⇒ 解けない!
  - って、よく見たら f を関数として使ったり、整数として使ったりしてるじゃん
  - 型検査を通してはいけないプログラム

例題(3): fun f → f(3)

- fの型は α, 各部分式 3, f(3), fun f → f(3) の型を $\beta 1,\ldots,\beta 3$とおく
- 型付け規則から方程式を立てる

整数定数式の型付け規則 ⇒ $\beta 1 = \text{int}$
関数呼び出し式の型付け規則 ⇒ $\alpha = \beta 1 \to \beta 2$
funの型付け規則 ⇒ $\beta 3 = \alpha \to \beta 2$

- 解く! ⇒ 別解が無数にある!
  - 解1: $\alpha = \text{int} \to \text{int}$, $\beta 3 = (\text{int} \to \text{int}) \to \text{int}$
  - 解2: $\alpha = \text{int} \to \text{string}$, $\beta 3 = (\text{int} \to \text{string}) \to \text{string}$
  - 解3: $\alpha = \text{int} \to \text{int list}$, $\beta 3 = (\text{int} \to \text{int list}) \to \text{int list}$
  - ...

ふたたび鶴亀算でいうと

Ocamlプログラマとラクダが合わせて7匹いる。
目の数は合計で14個であった。
OCamlプログラマとラクダはそれぞれ何匹いるか。

OCamlプログラマの数を x、ラクダの数を y とすると、

$$x + y = 7$$

$$2x + 2y = 14$$

これを解いて $x = 7 - y$ (解のパラメータ表示)
この関係を満たす自然数 x, y なら**なんでもよい**

例題(3): fun f → f(3)

- …

整数定数式の型付け規則 ⇒ $\beta 1 = \mathrm{int}$
関数呼び出し式の型付け規則 ⇒ $\alpha = \beta 1 \rightarrow \beta 2$
funの型付け規則 ⇒ $\beta 3 = \alpha \rightarrow \beta 2$

- 解く!
  - fun式の型 $\beta 3 = (\mathrm{int} \rightarrow \beta 2) \rightarrow \beta 2$
  - 上の式を満たす$\beta 3$、$\beta 2$なら**なんでもよい**
  - ➡ **多相性!**

方程式の一般形と解法

- 型に関する等式の集合
  - 等式の例: $\beta \to$int = (bool * α) $\to$ δ
  - 両辺: int, bool などの基本型と型変数を→や*で繋いでいった型
- 一般的には「一階(first-order) の単一化問題」と呼ばれる
- (解ける場合には)解が必ず求まる方法(詳細は省略)がある![Robinson65]
  - しかも「最も一般的」な解が求まる!
    - 全ての解のパラメータ表示

ここまでのまとめ

MLの型推論は単一化問題に帰着できる

- 式が型付けできるかどうか ＝ 式から導かれる単一化問題の解の有無
- 解が複数ある場合：多相的なプログラム

ML型推論礼賛


プログラムに
いちいち型書かなくてもよくて、
しかも安全なんてちょー便利じゃん
だよね

ML型推論の性質

- 型推論の完全性(主要型の推論)

うまく変数の型を与えれば型検査に通るようなプログラムは必ず型推論に成功する

  - 型宣言はいつでも省略可
  - 最も一般的な型(主要型)を推論
- 型推論の健全性

型推論に成功したプログラムは必ず型エラーなく実行できる

主要型

- 式に与えうる型のバリエーション全てを網羅するような(多相)型
  - fun x → (x, x) に与えうる型
    - int → int*int
    - string → string*string
    - int list → int list * int list
    - …
  - 主要型は: $\alpha \rightarrow \alpha * \alpha$
- 「主要型の推論」は単一化を解くアルゴリズムの性質の系

ML型推論に毒づく

ここには
イギリスのロックバンド Black Sabbath の
アルバム Heaven and Hell のジャケット
があると思ってください

つーか、MLってプログラム見ても型書いてなくて、
何すんだかわかんねーんだけど

だよね

ここには
イギリスのロックバンド Black Sabbath の
アルバム Heaven and Hell のジャケット
があると思ってください

つーか、MLってプログラム見ても型書いてなくて、
何すんだかわかんねーんだけど

だよね

トップレベルの関数くらいは
型が宣言されていた方が
親切かもしれません
(特に一週間後のあなたに)

ここには
イギリスのロックバンド Black Sabbath の
アルバム Heaven and Hell のジャケット
があると思ってください

つーかMLってさあ、
エラーメッセージが腐ってない？
こないだもさぁ。。。

ここには
イギリスのロックバンド Black Sabbath の
アルバム Heaven and Hell のジャケット
があると思ってください

```
let f x ls = (List.fold_left x (+) ls, x + 1);;
Characters xx-yy:
  ... x + 1);;
      ^
This expression has type
(int -> int -> int) -> 'a -> int -> int -> int
but is here used with type int
```

「こ、この巨大な型はいったい!?」

(20分後、デバグを終えて)
「間違えてんのここじゃねーし!(怒)」

解毒剂

親切なエラーメッセージを出す研究

- 方程式を解く順序を工夫する
  - エラーは(型)変数消去ができなくなった時に発生する
- 型推論が「いかに失敗したか」をうまく説明
- 適当な経験則でエラー箇所を推測(&修正の提案)
  - 経験則の例:
    場合分けの各枝で型が合わない場合は多数決で(!)
    どの枝が悪いか決める
- エラーに関連するプログラムを抽出して見せる(プログラムスライシング)

方程式を解く順序を工夫する

```
fold_left : (α→β→α)→α→β list→ α
x : α1,       (+) : int→int→int

… fold_left x (+) ls, x + 1
```

```
fold_left x       ⇒ α1 = α → β → α
fold_left x (+)   ⇒ α = int → int → int
x + ...           ⇒ α1 = int
```

αを消去　　α1を消去　　即エラー

```
α1 = (int→int→int) → β → (int→int→int)
α1 = int
```

万能な順序付けは難しい

- 多くのML処理系では式の位置で解く順番が決まる
  - 例えばペアの要素順を変えるだけで、ぐっとわかりやすくなる

```
let f x ls = (x + 1, List.fold_left x (+) ls);;
Characters xx-yy:
  ... List.fold_left x (+) ls);;
                     ^
This expression has type int
   but is here used with type 'a -> 'b -> 'a
```

- 本当は、いつでも、「正しいあたり」から解き始めたい

難しい、というか、根本的な解決は無理な話では？

- 問題設定のどこ(個体数、足の数、胃の数)が間違っていたのか答える方法がないのと同じ？

Ocamlプログラマとラクダが合わせて7匹いる。
足の数が合計で20本であった。
レントゲン写真を撮ってみると胃が14個見える。
Ocamlプログラマとラクダはそれぞれ何匹いるか。

- 結局、大体うまくいく経験則に頼るしかない

まとめ

- 方程式はエラい
- ML型推論もかなりエラい
  - 実行時の安全性保証
  - 主要な型の推論
- 何が間違いなのかを知るのは難しい
- 経験則に基いた対策はいろいろ考えられている
  - 実用上充分な決定版はまだない
- ラクダの胃の数は3つ
  - 第三･第四の胃がくっついている

おまけ: 型推論実装演習のはなし

- 学生にML型推論を OCaml で実装させてます
  - 対象: 学部3回生(Schemeの経験はあり)
  - 期間: 週6コマの演習 x 6週間 のうち最後の2週間
  - 多相性はオプション
- 正しく実装するのは難しいです
  - 方程式の構成と単一化の解消を交互にやるのがミスりやすい
  - 結果: はじくべきプログラムをはじけない
  - 対策: テスト用の正例・反例両方を提供

学生の実装の傑作No.1

```
# let s x y z = x z (y z);;
(* 正しい結果 *)
val s : ('a → 'b → 'c) →
          ('a → 'b) → 'a → 'c = <fun>
(* 学生Xの提出したプログラムの出力 *)
val s : 'a → 'b → 'c → 'd = <fun>
```

Obj.magic か!?