プログラミング言語論 #06

TinyHaskell

2018-05-21

プログラミング言語に型推論機構を導入することでプログラマは型宣言を書く必要がなくなった．また複数の値をまとめたい状況に対応する新たな型を導入することでプログラムが書きやすくなった．

しかしこの二つの機能を両方とも導入すると型が一意に定まらない状況が存在し，型を表す変数の導入が避けられず，型推論はその対応が必要となった．

現在の実際のコンパイラではこのような状況にも問題なく対応できる型推論アルゴリズムを用いている．数学における連立方程式からの類推を通して，そのアルゴリズムがどのようなものかを理解する．

型変数を含む型推論と型検査

```
1  x ::          -- [a]
2  x = []
3
4  y :: [Int]
5  y = 0 : x    -- 定数Intと変
       数a
6
7  z :: [Bool]
8  z = True : x -- 定数と変数
```

変数 x の要素の型は与えられていないので 2 行目の定義から推論を行う．その結果，型変数 a を導入する．次に 5, 8 行目（更にそこで定義されている変数 y,z に関する 4,7 行目）における型に関する制約を挙げ，それぞれ成立することを検査する．
2 行目で x の型は定数化できない（単一解を持たない）ことがわかったが，型変数を導入することで全ての制約を満足させることができるので「x の要素の型はなんでもよい」という推論結果に問題ないことが検査できた．

型検査：

- 両辺が定数ならその値が等しいこと（等価）
- 定数と変数なら変数の値を定数とする（型代入）
- 両辺が変数なら，両者は同じ値を持つものとする（単一化）

以上を繰り返し行い，**全ての変数の型が単一の値（または単一の型変数）**となり，全ての型に関する等式が成立することを確認する．すなわち型に関する方程式が有解であることを確かめる．

Example 1

問題においては型推論優先

```
1  f1 ::
2  f1 x = True
```

- × エラーにならないように１行目の宣言を完成させよ
- ◯ 処理系による型推論の結果を１行目に書け

検算：ある型が Int になった． → 他の型では型エラーか．

Example 2

a と $[a]$ は違う

```
1 length []
2 length [1]
3 length [2]
4 length [True, False]
5 length ["A", "A", "A", "CC"]
6 length [[], [], [], []]
7 length 4
```

Example 3

```
1  f3 ::
2  f3 x = []
```

検算：使用する型変数の数はそれでいいか自問自答

応用例

```haskell
-- f2 ::
-- f2 x = [x]
--
-- f3 ::
-- f3 x = []

f4 ::
f4 x y = []

f5 ::
f5 x y = (x, y)

f6 ::
f6 x y = [x, y]
```

応用例 2

```
1  f7 ::
2  f7 x y = x : y : []   -- 右から実行 = (x : (y : []))
3
4  f8 ::
5  f8 x y = x : y
6
7  f9 ::
8  f9 x y = [(1,x), y]
9
10 f10 ::
11 f10 x y = ((x, "OK"), ("NG", y))
12
13 f11 ::
14 f11 x y = [(x, "OK"), ("NG", y), (y, x)]
15
16 f12 ::
17 f12 x = x : x
```

型推論の理解度に関する問題のルール

- 要求されているのは「処理系による型推論の結果」 back
- 「使ってよいものは，講義で説明した基本となる型及びそれらの組とリスト，型変数だけとする」
  - 基本となる型： `Bool`, `Int`, `String`
  - X 講義で説明してない基本となる型： Bxxx, Cxxx, Dxxx,···
  - X 講義で説明してない，型ですらないもの：題意より使用不可
  - 基本とならない型：組，リスト（基本型ではなく合成型）
- 「型エラーになる場合は *error* と書くこと」
  全ての型方程式が解を持つわけではない
- 右辺に出てくる関数，演算子：講義で用いたもの
- 部分点なし

型検査・型推論アルゴリズム

式の形に関する再帰的な型検査，型付けのアルゴリズムを示す

1. 式の形に基づく型検査の方針
2. 公式化
3. 導出規則
4. 型推論

式の木表現

式は再帰的に定義できる

定数式	1, 2, True, "AA" $\cdots$	3
変数式	x, y, $\cdots$	1
演算子式	式 1 + 式 1, 式 1 $\|\|$ 式 1, 式 1 == 式 1, $\cdots$	12
関数適用式	div 式 1 式 2, mod 式 1 式 2, not 式 1	3
型構成式	(式 1, 式 2, $\cdots$), [式 1, 式 2, $\cdots$]	2

式の型は部分式の型から決まる　　数学的には帰納法，技法的には再帰

20 種類 $\rightarrow$ 20 通りの場合分けで全ての式をカバーできる（似ているものをまとめると実質 5 通り）

導出形式の定式化

$$\frac{\Gamma \vdash t_1 :: \mathsf{Int} \qquad \Gamma \vdash t_2 :: \mathsf{Int}}{\Gamma \vdash t_1 + t_2 :: \mathsf{Int}} \quad (\textsc{Int}\text{-}+)$$

型環境 Γ の下で式 t_1 が Int 型を持ち，また同様に型環境 Γ の下で式 t_2 が Int 型を持つ場合，型環境 Γ の下で式 $t_1 + t_2$ が Int 型を持つことを導出（証明）できる．このルール（導出規則）に Int-+という名前を与える．
残り 19 通りも同じような形式で説明できる．一般に

$$\frac{\Gamma \vdash t_1 :: \tau_1 \qquad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash t_1 \oplus t_2 :: \tau} \quad (\textsc{Rule})$$

一般的な教科書での記法を少しだけ Haskell 学習者用に翻案した

型環境

型環境は変数とその型を保持するものなので以下で定義できる

Haskell で実現された型環境の型宣言

```
1  型環境 :: [(変数名を表すString, 型を表すデータ)]
```

- 型を表すデータは現在の Haskell の知識では無理なので，型をそのまま使うことにする（エラーになるがしょうがない）

例（Γ はこれらを表すメタ変数）

```
1  g    = []    -- 変数が一つも出現しない世界
2  ga   = [ ("x", Int), ("y", [Int]) ]
3  gam  = [ ("x", a), ("y", [b]) ] -- 型変数も型である
4  gamm = [ (1, Int), ("x", True), ("x + y", Int) ]  -- X
```

定義文の左辺で新規に変数が定義されるたびに型宣言文があればそれを，そうでなければ新規な型変数を割り当てる（それまでの変数定義と宣言を型環境に反映させたところからある式の型検査を始める）

型導出システム TinyHaskell/0 での導出規則の一部

第 3 回時点での言語の型検査に対応

$$\frac{}{\Gamma \vdash 0 :: \mathsf{Int}} \quad (\textsc{Int})$$

$$\frac{\Gamma \ni (x, \alpha)}{\Gamma \vdash x :: \alpha} \quad (\textsc{Var})$$

$$\frac{\Gamma \vdash t_1 :: \mathsf{Bool} \qquad \Gamma \vdash t_2 :: \mathsf{Bool}}{\Gamma \vdash t_1 \ \mathsf{\&\&} \ t_2 :: \mathsf{Bool}} \quad (\textsc{Bool-\&\&})$$

$$\frac{\Gamma \vdash t_1 :: \mathsf{Int} \qquad \Gamma \vdash t_2 :: \mathsf{Int}}{\Gamma \vdash t_1 \ * \ t_2 :: \mathsf{Int}} \quad (\textsc{Int-*})$$

$$\frac{\Gamma \vdash t_1 :: \mathsf{String} \qquad \Gamma \vdash t_2 :: \mathsf{String}}{\Gamma \vdash t_1 \ \mathsf{++} \ t_2 :: \mathsf{String}} \quad (\textsc{Str-++})$$

$$\frac{\Gamma \vdash t_1 :: a \qquad \Gamma \vdash t_2 :: b \qquad a = b}{\Gamma \vdash t_1 \ \mathsf{==} \ t_2 :: \mathsf{Bool}} \quad (\textsc{Rel-=})$$

厳密な型検査の議論例

Q：型導出システム TinyHaskell/0 を用いて型環境
$\Gamma = [(\text{"x"}, \text{Int}), (\text{"y"}, \text{Bool})]$ の下での下式の型検査を行え

なお，対象言語はこれまでの講義で示した内容に限定された「制限された Haskell」である．

```
1  [(x + 1, y), (x, y == True)]
```

A：導出過程は以下の導出図式で示される．これにより型環境 Γ の下で与式は型 `[(Int, Bool)]` を持つという結果を得る．

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \ni (\text{"}x\text{"}, \texttt{Int})}{\Gamma \vdash \texttt{x} :: \texttt{Int}}\,(\textsc{Var}) \quad \dfrac{}{\Gamma \vdash \texttt{1} :: \texttt{Int}}\,(\textsc{Int})}{\Gamma \vdash \texttt{x + 1} :: \texttt{Int}}\,(\textsc{Int-+}) \quad \dfrac{\Gamma \ni (\text{"}y\text{"}, \texttt{Bool})}{\Gamma \vdash \texttt{y} :: \texttt{Bool}}\,(\textsc{Var})}{\Gamma \vdash \texttt{(x + 1, y)} :: (\texttt{Int}, \texttt{Bool})}\,(\textsc{Pair}) \quad \dfrac{\dfrac{\Gamma \ni (\text{"}x\text{"}, \texttt{Int})}{\Gamma \vdash \texttt{x} :: Var}\,(\textsc{Int}) \quad \dfrac{\dfrac{\Gamma \ni (\text{"}y\text{"}, \texttt{Bool})}{\Gamma \vdash \texttt{y} :: \texttt{Bool}}\,(\textsc{Var}) \quad \dfrac{}{\Gamma \vdash \texttt{True} :: \texttt{Bool}}\,(\textsc{Bool})}{\Gamma \vdash \texttt{y == True} :: \texttt{Bool}}\,(\textsc{Rel-=})}{S\Gamma \vdash \texttt{(x, y == True)} :: (\texttt{Int}, \texttt{Bool})}\,(\textsc{Pair})}{\Gamma \vdash \texttt{[(x + 1, y), (x, y == True)]} :: [(\texttt{Int}, \texttt{Bool})]}\,(\textsc{List})$$

この導出図式は与えられた式の構造を木として逆向きに描いたものに一致する（偶然ではなく）．

型推論への拡張

要求：宣言がない変数への対応，型変数への対応

- 型環境 Γ に含まれる変数の中には，具体的な型（型定数）をもつものだけでなく，型変数を持つものも含まれる．
- ただし，導出規則の再帰的適用過程において，最初は型変数であったものが型定数へと具体化されるかもしれない．
- この場合，型環境 Γ の更新が必要となる．

これらに対応した型導出システムとして **TinyHaskell/1** を導入する．

型間の具体化関係 $\succeq$ と単一化

緩いものを厳しくする（または何もしない）事のみ可能

- 型変数を含む型 σ からより具体的な型 τ が「抽出」できることを $\sigma \succeq_S \tau$ で表す（具体的な置換内容を S で表す）
  - 型変数は型定数へ置換可能： $a \succeq_S \mathtt{Int}$, $a \succeq_S \mathtt{Bool}$, など
  - 型変数は型変数へ置換可能： $a \succeq_S a$, $a \succeq_S b$, など
  - 反射律，推移律成立： $\mathtt{Int} \succeq_S \mathtt{Int}$, $\mathtt{Bool} \succeq_S \mathtt{Bool}$, など
  - 詳細な方向への置換は可能： $a \succeq_S [b]$, $a \succeq_S (b, c)$, など
- $S\Gamma$ で型環境 Γ に置換 S を施した新しい型環境を表す

S のかたち（これ以外のものは置換ではない）

```
1  S:: [(型変数, 型)] -- [(Int, Int)]は実質無変換なので不要
```

- $\sigma_1 \succeq_S \tau, \sigma_2 \succeq_S \tau$ ならば σ_1， σ_2 は置換 S により同じ型 τ に単一化できることを意味する

型変数の存在を前提とした型推論

前提である部分式，帰結である式の間にある関係を $\succeq_S$ を用いて表現し，必要に応じて型環境を更新しながら（型推論），型検査を進める

例

```
1  x == 0      -- 型環境を Γ =[("x", a)]とする
```

1. (==) の両辺（部分式）は同じ型でなければならない
2. 右辺は Int であり，左辺は a である．単一化可能か
3. $S=$「型変数 a を型 Bool と置換」
4. 更新された型環境 $S\Gamma$ は [("x", Int)]

できなければ型エラー

型導出システム TinyHaskell/1 での導出規則の一部

$$\frac{\Gamma \ni (x, \alpha) \qquad \alpha \succeq_S \tau}{S\Gamma \vdash x :: \tau} \quad (\textsc{Var})$$

$$\frac{\Gamma \vdash t_1 :: a \qquad \Gamma \vdash t_2 :: b \qquad a \succeq_S \texttt{Bool} \qquad b \succeq_S \texttt{Bool}}{S\Gamma \vdash t_1 \texttt{ \&\& } t_2 :: \texttt{Bool}} \quad (\textsc{Bool-\&\&})$$

$$\frac{\Gamma \vdash t_1 :: a \qquad \Gamma \vdash t_2 :: b \qquad a \succeq_S \texttt{Int} \qquad b \succeq_S \texttt{Int}}{S\Gamma \vdash t_1 \texttt{ + } t_2 :: \texttt{Int}} \quad (\textsc{Int-+})$$

$$\frac{\Gamma \vdash t_1 :: a \qquad \Gamma \vdash t_2 :: b \qquad a \succeq_S \texttt{String} \qquad b \succeq_S \texttt{String}}{S\Gamma \vdash t_1 \texttt{ ++ } t_2 :: \texttt{String}} \quad (\textsc{Str-++})$$

$$\frac{\Gamma \vdash t_1 :: a \qquad \Gamma \vdash t_2 :: b \qquad a \succeq_S \tau \qquad b \succeq_S \tau}{S\Gamma \vdash t_1 \texttt{ == } t_2 :: \texttt{Bool}} \quad (\textsc{Rel-=})$$

$$\frac{\Gamma \vdash t_1 :: a \qquad \Gamma \vdash t_2 :: b \qquad \cdots}{\Gamma \vdash (t_1, t_2, \cdots) :: (a, b, \cdots)} \quad (\textsc{Tpl})$$

$$\frac{\Gamma \vdash t_1 :: a \qquad \Gamma \vdash t_2 :: b \qquad \cdots \qquad a \succeq_S \tau \qquad b \succeq_S \tau \qquad \cdots}{S\Gamma \vdash [t_1, t_2 \cdots] :: [\tau]} \quad (\textsc{List})$$

型推論における導出ステップの例

Q：型導出システム TinyHaskell/1 を用いて型環境 $\Gamma = [(\texttt{"x"}, b), (\texttt{"y"}, c)]$ の下での下式の型推論を行え

```
1  [(x + 1, y), (x, y == True)]
```

A：与えられた型環境の下で以下が成立する（とする：再帰的に証明すべき）：

$$\Gamma \vdash \texttt{(x + 1, y)} :: \texttt{(Int, c)} \qquad \Gamma \vdash \texttt{(x, y == True)} :: \texttt{(b, Bool)}$$

S を「型変数 b を Int へ，型変数 c を Bool への置換」とすると，

$$S\Gamma \vdash \texttt{(x + 1, y)} :: \texttt{(Int, Bool)} \qquad S\Gamma \vdash \texttt{(x, y == True)} :: \texttt{(Int, Bool)}$$

となり，以下のように導出規則 LIST が使える．

$$\frac{\Gamma \vdash \texttt{(x + 1, y)} :: (\texttt{Int}, b) \quad \Gamma \vdash \texttt{(x, y == True)} :: (c, \texttt{Bool}) \quad (\texttt{Int}, b) \succeq_S (\texttt{Int}, \texttt{Bool}) \quad (c, \texttt{Bool}) \succeq_S (\texttt{Int}, \texttt{Bool})}{S\Gamma \vdash \texttt{[(x + 1, y), (x, y == True)]} :: [(\texttt{Int}, \texttt{Bool})]} \ (\textsc{List})$$

従って具体的に S が存在し，与式は [(Int,Bool)] と型付けできる．

注釈

- `let` がない世界の型付けである
- 関数定義文全体を問うことはせず，定義式の左辺に対応する型環境を最初に与える

この設定により Tinyhaskell/1 には「型スキーム」は導入しない（型変数名の重複に触れてないのは．．．）

- 置換を繰り返すなら $S_3S_2S_1\Gamma$ のように書くこと．置換 S は具体的な値を持つ変数であり，単に置換したら $S\Gamma$ と書くというルールではない． S が導出規則に複数回出現すればそれらは全て同じ値を持っていると考えること．
- 同様に Γ も具体的な値を持つ変数である．様々な型環境を変数 Γ で代表させただけ．

$$
\begin{aligned}
\Gamma_0 &= [\cdots] \\
\Gamma_1 &= (\text{"i"}, \mathsf{Int}) : \Gamma_0 \\
\Gamma_2 &= S_1\Gamma_1 \\
\Gamma_3 &= S_2\Gamma_2 = S_2S_1\Gamma_1
\end{aligned}
$$

  Γ が導出規則に複数回出現すればそれらは全て同じ値を持っていると考えること．

- 構文エラーがない全ての式が型付けできるわけではない．それらは型エラーである．
- 式の構造に関する有限回の再帰で済む簡単なアルゴリズムのように見えるが，プログラマにとって便利な型の導入を続けると式の長さに関わらず導出ステップが無限回必要になり，停止性が保証されなくなる．

TinyHaskell/1 の実装方針

- 型環境は型検査のものと同じ

型環境

```
1  Γ:: [(変数名を表すString, その変数の型)]
```

- 定義文の左辺で新規に変数が定義されるたびに型宣言文があればそれを，そうでなければ新規な型変数を割り当てる（何も情報がないところから推論を始める）

変数追加に関する関数

```
1  defined :: 変数名 -> 型環境 -> Bool -- 定義済みかどうか
2  extend :: 新変数名 -> 型 -> 型環境 -> 更新された型環境
```

- それ以降（定義文の右辺）でその変数が使われる度に，その使い方から要求される型制約と登録された型情報とが矛盾を起こしていないかを検査する（型検査）．矛盾を起こしておらず，単一化できるならば，型環境の情報を更新する

実質的型推論関数：単一化，型検査，型環境の更新

```
1  unify::変数名 -> 要求される型 -> 今の型 -> 型環境 -> 型環境
```

参考文献


五十嵐淳，プログラミング言語の基礎概念，サイエンス社，2011.


Benjamin C. Pierce 著，住井英二郎監訳，型システム入門プログラミング – 言語と型の理論，オーム社， 2013.