Haskell 言語拡張における型推論の追実装

目的

定理証明などの幅広い応用がある Haskell 言語拡張を追実装することで理解を深める

Haskell とは？

- > An advanced, purely functional programming language
- https://www.haskell.org/

```haskell
sort :: forall a. Ord a => [a] -> [a]
sort [] = []
sort (x:xs) = sort smaller ++ [x] ++ sort bigger
  where
    smaller = filter (< x) xs
    bigger = filter (>= x) xs
```

型の読み方

```
-- 関数名 :: 型変数の導入. 型制約 => 引数  -> 戻り値
   sort  :: forall a  . Ord a => [a]  -> [a]
```

- 変数は引数のない関数

```
one :: Int
```

- 大文字で始まるのは型コンストラクタ
  - 定義された型
  - 並べて書くとジェネリクスみたいなものの適用: `T a`
    - 型でパラーメタ化された型

```
map :: forall a b. (a -> b) -> Maybe a -> Maybe b
```

読み方

- `case` や定義を並べてパターンマッチ

```
-- 1つ目の引数（`[a]` 型）が `[]` だったら e1
sort [] = e1
-- 1つ目の引数（`[a]` 型）が `x : xs` だったら e2
sort (x:xs) = e2
```

```
-- 上と同じ意味
sort xs = case xs of
  [] -> e1
  (x : xs) -> e2
```

- 並べると関数適用

```
-- `sort` 関数に `bigger` 変数を渡す
sort bigger
```

導入: すごい Haskell

もしくは実装した Haskell 言語拡張の説明

配列の足し算

長さが違う配列は足せないが…

```
arr1 = np.array([1,2,3])
arr2 = np.array([1,1,1,1])
print(arr1 + arr2)
# ValueError: operands could not be broadcast together with shapes (3,) (4,)
```

普通に型をつけても問題は解決しない

- 型が**付いてしまう**→ 実行時エラー 😢

```
add :: [Int] -> [Int] -> [Int]
```

```
-- well-typed, but invalid
add [1,2,3] [1,1,1,1]
```

Lightweight dependent type

2 つの引数が同じ長さ `n` であることを**型の上で表現**

```
add :: Vec n -> Vec n -> Vec n
```

長さの違う配列を足そうとすると型が合わない

- コンパイル時エラー 🎉

```
let arr1 = VS 1 (VS 2 (VS 3 VZ)) in
let arr2 = VS 1 (VS 1 (VS 1 (VS 1 VZ))) in
add arr1 arr2  -- rejected!
```

型の上での計算

配列を結合

- 長さの足し算を型の上で表現: `Add n m`

```
append :: Vec n -> Vec m -> Vec (Add n m)
```

型の上で計算をして型を変える

```
let arr1 = VS 1 (VS 2 (VS 3 VZ)) in
let arr2 = VS 1 (VS 1 (VS 1 (VS 1 VZ))) in
-- well-typed, because 3 + 1 and 4 are known to be the same
add (append arr1 (VS 4 VZ)) arr2
```

型の上での計算についての事実

逆に足した結果 (`Add n m` と `Add m n`) は同じ？

- コンパイラにはわからない

```
-- rejected!
add (append v1 v2) (append v2 v1)
```

証明が必要

```
-- well-typed, given proof :: Add n m :~: Add m n
case proof of
  Refl -> add (append v1 v2) (append v2 v1)
```

半自動定理証明

- 定理証明: Curry-Howard 同型対応
- 半自動: 型推論によって推移律や対称律などによる書き換えは自動で推論
  - ↓ のように必要な定理と帰納法の仮定を並べてやれば証明できちゃったり

```
plusComm :: SNat n -> SNat m -> Add n m :~: Add m n
plusComm SZ m = case plusZRight m of Refl -> Refl
plusComm (SS n) m =
  case plusSnRight m n of
    Refl -> case plusComm n m of
      Refl -> Refl
```


```haskell
-- well-typed, given len :: Vec n -> SNat n
case plusComm (len v1) (len v2) of
  Refl -> add (append v1 v2) (append v2 v1)
```

ほかにも魅力的な応用が…

- typed `printf`

```
main = do
  name <- getLine
  age  <- read <$> getLine
  printf ("Hello, " `str` " (" `int` ")!\n") name age
```

- typed interpreter

```
data Expr t where
  Lit :: a -> Expr a
  Plus :: Expr Int -> Expr Int -> Expr Int
  IsZero :: Expr Int -> Expr Bool
  If :: Expr Bool -> Expr a -> Expr a -> Expr a

eval :: Expr t -> t
```

Haskell のライブラリとして effect system を実装した Extensible effects にも `GADTs` が応用されています

すごい

何？

Under the hood (1/1)

拡張機能: `GADTs`

- データ型の型パラメータを値の構造によって限定できる

```
data Vec n where
  VZ :: Vec Z
  VS :: Int -> Vec m -> Vec (S m)
```

長さが型に反映されている

```
VS 1 VZ :: Vec (S Z)
```

```
VS 1 (VS 2 (VS 3 VZ)) :: Vec (S (S (S Z)))
```

一般的に Generalized algebraic datatype と呼ばれる機能

Under the hood (2/2)

拡張機能: `TypeFamilies`

- 型の上の関数; 型を受け取って型を返す

```
type family Add n m where
  Add Z     m = m                -- (1)
  Add (S n) m = S (Add n m)      -- (2)
```

`Add (S Z) m` と `S m` は同じ型になる

```
Add (S Z) m
  => S (Add Z m)  -- by (2)
  => S m          -- by (1)
```

一般的に Type family と呼ばれる機能で、それは Associated type と呼ばれる機能に似ています

それだけ

この二つの拡張機能を含めた型推論で実現できる
→ 実装したい！（理論と実装に興味が出る）

拡張機能？

- Haskell は**プログラミング言語**
  - 仕様が策定されている: Haskell 98, Haskell 2010
- GHC (Glasgow Haskell Compiler) は Haskell の**処理系**（コンパイラ）
  - Haskell の仕様に沿って実装されている
- **GHC 拡張**は、Haskell の仕様を超えて GHC が独自に実装している拡張機能
  - `GADTs` `TypeFamilies` はこれです
- 今は GHC 以外に生きた処理系がないから GHC 拡張を使うのが当たり前
  - 😇

やったこと: 実装

実装をしました

- `GADTs` `TypeFamilies` と同様の型推論を再実装
- 簡単な文法を載せた小さな言語を作り、それに実装した
- アルゴリズム: OutsideIn(X)
  - GHC のベースになっている
- Vytiniotis, Dimitrios, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. "OutsideIn (X) Modular type inference with local assumptions." Journal of functional programming 21, no. 4-5 (2011): 333-412.
  - Simon Peyton Jones は Haskell のすごい人みたいな感じ

型推論の流れ

- 制約生成
  - プログラムに型が付くために満たす必要がある制約を集める
  - e.g. `id True` $\rightsquigarrow (\texttt{Bool} \to u_2) \sim (u_1 \to u_1)$
    - where `id :: forall a. a -> a`
- 制約解消
  - 制約を満たす型変数への代入を求める
  - e.g. $(\texttt{Bool} \to u_2) \sim (u_1 \to u_1) \rightsquigarrow u_1 := \texttt{Bool}, u_2 := \texttt{Bool}$

`GADTs` が居ると？

```haskell
map :: forall a b n. (a -> b) -> Vec n a -> Vec n b
map _ VZ = VZ
map f (VS x xs) = VS (f x) (map f xs)
```

- そのままでは `Vec n b` は作れない
  - `n` がわからない
- パターンマッチによって得た知識を元に `n` に制約が増えたおかげで `Vec n b` が作れるようになる
- でもその知識はのマッチの右辺でのみ有効
- `GADTs` でのパターンマッチでは、**その右辺でのみ有効な型制約**が登場する

`GADTs` が居ると？

```
map :: forall a b n. (a -> b) -> Vec n a -> Vec n b
map _ VZ = VZ
```

- `Vec n a` の引数が `VZ` なので、 `n ~ Z` だとわかる
  - そのため、戻すのは `Vec n b` ではなく `Vec Z b` で良い

```
map f (VS x xs) = VS (f x) (map f xs)
```

- `Vec n a` の引数が `VS x xs` なので、$\exists n'.\, \mathbf{n} \sim \mathbf{S}\, n'$ だとわかる
  - そのため、戻すのは `Vec n b` ではなく `Vec (S n') b` で良い
  - ここで `xs :: Vec n' a` なので `map f xs :: Vec n' b`
  - よって右辺が `Vec (S n') b` になり、無事型が付く

`GADTs` が居ると？

```haskell
map :: forall a b n. (a -> b) -> Vec n a -> Vec n b
map _ VZ = VZ
map f (VS x xs) = VS (f x) (map f xs)
```

- 出てくる制約

$$(\mathtt{n} \sim \mathtt{Z} \supset \mathtt{Vec\ Z}\ u_1 \sim \mathtt{Vec\ n\ b}) \wedge (\mathtt{n} \sim \mathtt{S\ n'} \supset \mathtt{Vec}\ (\mathtt{S\ n'})\ \mathtt{b} \sim \mathtt{Vec\ n\ b})$$

  - $\supset$ でローカルな型制約を表現 (**local implication**)

local implicationを含む制約解消

$$u_1 \sim \texttt{Bool} \supset u_2 \sim \texttt{Bool}$$

- これは $u_2 := \texttt{Bool}$ か $u_2 := u_1$ か？
  - どっちでもいい？
  - $u_1 \to u_2$
    - $u_1 \to \texttt{Bool}$
    - $u_1 \to u_1$
    - 違う型が付いてしまう！
    - どっちでもよくない

local implicationを含む制約解消: 課題

- implication constraint を含む型推論はたいへん

TODO

local implicationを含む制約解消: 方針

- OutsideIn(X)
  - GHCのベースになっている、多少アドホックだが実装が簡単なアプローチ
- 方針: implication constraint の中では**代入をしない**
  - 例外: implication constraint の右辺で完結している型変数
- 型推論は弱くなるけれど外側で代入をすればいいので致命的ではない

`TypeFamilies` での制約解消

- `TypeFamilies` がいない場合
  - $t_1 \sim t_2 \iff t_1$とt_2が構文的に同等
  - e.g. $\mathtt{T\ [Int]} \sim \mathtt{T\ Bool} \not\rightsquigarrow$
- `TypeFamilies` が居ると
  - $t_1 \sim t_2 \iff t_1$とt_2を型族の定義によって変形した結果が構文的に同等
  - e.g. $\mathtt{F\ [Int]} \sim \mathtt{F\ Bool} \rightsquigarrow \epsilon$

```
type family F a where
  F [x] = F x
  F Int = Bool
  F Bool = Bool
```

`TypeFamilies` での制約解消

- やみくもにやっていいわけではない
- $\mathtt{a} \sim [\mathtt{F\ a}] \supset \mathtt{G\ a} \sim \mathtt{Bool}$

```
type family G a where
  G [x] = Bool
```

- $\mathtt{G}\ [\mathtt{F\ a}] \sim \mathtt{Bool}$
  - NG: $\mathtt{G}\ [\mathtt{F}\ [\mathtt{F\ a}]] \sim \mathtt{Bool}$
    - 止まらない
  - OK: $\mathtt{Bool} \sim \mathtt{Bool}$
    - `G [x] = Bool` より

`TypeFamilies` での制約解消: 課題

- termination
  - インスタンス宣言に対する最低限の制約で制約解消が停止するようにしたい

`TypeFamilies` での制約解消: 方針

TODO

Evidence 生成: 制約生成

```
axiom ∀a. <F a> ~ a

let f :: ∀a. <F a> -> a
  = \x. x
```

```
axiom $F_a = ∀a. <F a> ~ a

let f :: ∀a. <F a> -> a
  = Λa. (\(x :: 'u0). x) ▷ %c1
```

```
%c1: ('u0 -> 'u0) ~ (<F a> -> a)
```

Evidence 生成: 制約解消

```
%c1: ('u0 -> 'u0) ~ (<F a> -> a)
(%c2: 'u0 ~ <F a>) ∧ (%c3: 'u0 ~ a)     (CANW) %c1 = (->) %c2 %c3
(%c4: <F a> ~ 'u0) ∧ (%c3: 'u0 ~ a)     (CANW) %c2 = sym %c4
(%c5: a ~ 'u0) ∧ (%c3: 'u0 ~ a)         (TOPG) %c4 = $F_a @a ∘ %c5
(%c6: 'u0 ~ a) ∧ (%c3: 'u0 ~ a)         (CANW) %c5 = sym %c6
(%c6: 'u0 ~ a) ∧ (%c7: a ~ a)           (INTW) %c3 = %c6 ∘ %c7
(%c6: 'u0 ~ a)                          (CANW) %c7 = <a>

θ: 'u0 ↦ a                                      %c6 = <a>
```

Evidence 生成: 置換

```
%c1 = (->) %c2 %c3
%c2 = sym %c4
%c4 = $F_a @a ∘ %c5
%c5 = sym %c6
%c3 = %c6 ∘ %c7
%c7 = <a>
%c6 = <a>
'u0 ↦ a
```

```
axiom $F_a = ∀a. <F a> ~ a

let f :: ∀a. <F a> -> a
  = Λa. (\(x :: 'u0). x) ▷ %c1
```

```
let f :: ∀a. <F a> -> a
  = Λa. (\(x :: a). x) ▷ (->) (sym ($F_a @a ∘ (sym <a>))) (<a> ∘ <a>)
```

Evidence 生成: 単純化

- `sym <t>` = `<t>`
- `T <t1> <t2> ..` = `<T t1 t2 ..>`
- `c ∘ <t2>` = `c`
  - where `c :: t1 ~ t2`
- とか

```
let f :: ∀a. <F a> -> a
  = Λa. (\(x :: a). x) ▷ (->) (sym ($F_a @a ∘ (sym <a>))) (<a> ∘ <a>)
```

```
let f :: ∀a. <F a> -> a
  = Λa. (\(x :: a). x) ▷ (->) (sym ($F_a @a)) <a>
```

実装

https://github.com/coord-e/impl-outsidein

Files	Lines	Code	Comments	Blanks
30	4312	3665	181	466

`evidence` ブランチに evidence 生成が実装されており、 `main` は型推論のみ

自動定理証明の例もちゃんと型推論できる

https://github.com/coord-e/impl-outsidein/blob/evidence/test/data/sample_nat_add_props.ok

吐き出した証明（自動でチェックできる）↓

```
axiom $a1 = ∀m. <Add Z m> ~ m
axiom $a0 = ∀n m. <Add (S n) m> ~ S <Add n m>

let plusComm :: ∀n. ∀m. SNat n → SNat m → Eq <Add n m> <Add m n> =
  Λn. Λm. λ(n :: SNat n). λ(m :: SNat m).
    case n → Eq <Add n m> <Add m n> {
      (%b255) SZ @n >('c239 :: n ~ Z) =>
        case plusZRight @m m → Eq m <Add m Z> {
          (%b252) Refl @<Add m Z> @m >('c246 :: <Add m Z> ~ m) =>
            Refl @m @m @<m> ▷ Eq <m> (sym 'c246)
        } ▷ sym (Eq (<Add 'c239 <m>> ∘ $a1 @m) <<Add m Z>>) ∘ sym (Eq <<Add n m>> <Add <m> 'c239>),
      (%b290) SS @n >'a257 >('c258 :: n ~ S 'a257) (n2 :: SNat 'a257) =>
        case plusSRight @m @'a257 m n2 → Eq (S <Add 'a257 m>) <Add m (S 'a257)> {
          (%b287) Refl @<Add m (S 'a257)> @(S <Add m 'a257>) >('c268 :: <Add m (S 'a257)> ~ S <Add m 'a257>) =>
            case plusComm @'a257 @m n2 m → Eq (S <Add 'a257 m>) (S <Add m 'a257>) {
              (%b284) Refl @<Add 'a257 m> @<Add m 'a257> >('c278 :: <Add 'a257 m> ~ <Add m 'a257>) =>
                Refl @(S <Add m 'a257>) @(S <Add m 'a257>) @<S <Add m 'a257>> ▷ Eq (sym (S 'c278)) <S <Add m 'a257>>
            } ▷ sym (Eq <S <Add 'a257 m>> 'c268)
        } ▷ sym (Eq (<Add 'c258 <m>> ∘ $a0 @'a257 @m) <<Add m (S 'a257)>>) ∘ sym (Eq <<Add n m>> <Add <m> 'c258>)
    }
```

型クラスも居るよ

```
data Pair :: ∀a b. ∃. ∃. a → b → Pair a b

type $d0 :: ∀a. ∀b. {Show} a → {Show} b → {Show} (Pair a b)
type $d1 :: {Show} Bool

type show :: ∀a. {Show} a → a → String
type parens :: String → String

let parensShow :: ∀a. {Show} a → a → String =
  Λa. λ('x20 :: {Show} a). λ(x :: a).
    parens (show @a 'x20 x)
```

型と coercion は消せるが型クラス制約は引数になるの、セマンティクスに影響するものがはっきりする感じがしてよくないですか

実装のおもしろポイント

- OutsideIn(X) と言っているくらいだから X はなんでもあり
  - なので、型推論器は X から独立して実装している

```
typeProgram ::
  forall x m.
  ( ConstraintDomain x,
    MonadLogger m,
    MonadError (TypeError x) m
  ) =>
  Program x ->
  m ()
```

- 外から `typeProgram @TypeClass` みたいに X を埋めて使える
  - 論文をそっくりそのまま実装できた感じがあって嬉しい

判明した OutsideIn(X) 論文の怪しいポイント

- 制約解消ルールの適用順によって解けたり解けなかったりする
  - 🤯
  - あるルールを優先させると解けるようになったり
  - confluence がない
- うまく解けないコーナーケースがある
  - $\mathtt{T}\ (\mathtt{F}\ \mathtt{a}) \sim tv$
  - 入れ替えるルールがあるはずが、条件の定義ミス？で引っかからない
    - 実装では修正した
  - 制約の意味がはっきりしてないので出力が適切な解なのかわかっていない
    - 😱
    - ソルバが常に適切な解を返すかどうか論じることもできない

これからやりたいこと

Constraint Handling Rules (CHR) での制約解消

- OutsideIn(X) の制約ソルバは怪しいところがある
- 制約を CHR に変換し、confluence や制約の意味をはっきりさせたい
- 拡張が容易な型クラス制約にも興味がある
- CHR + evidence 生成はまだ成されていないみたい
  - > The issue of evidence generation has not been fully addressed for CHR yet.

まとめ

- Haskell (GHC) 拡張で依存型がシミュレートできてすごい
- 意外と容易に実現できて → うれしい！
- 参考論文に怪しい部分があるのでもっとよくできそう！

スライドの PDF 👇

https://coord-e.github.io/slide-coins20lt-outsidein-type-inference/slide.pdf