型システムを学ぼう！

書いた日： 2017 年 12 月 15 日[*1]、書いた人：@unnohideyuki[*2]、

はじめに

これは Haskell Advent Calendar 2017 (その 1)[*3] 15 日目の記事です。この記事では、以下のような構成で Hindley/Milner の型推論について述べたいと思います。

1. 型システム： 構文、意味論、および、型付け規則
2. Hindly/Milner の型推論
3. Haskell による実装

最初の節では、とても小さな言語を題材にして型システムを形式的に定義してみます。ここでは、型とはなにか、また、その役割や能力について確認します。この節のもうひとつの目的は、型システムを形式的に記述するやりかたに慣れることです。そして次に、Hindley/Milner の型推論について述べ、最後に Haskell による実装を示します。

1. 型システム： 構文、意味論、および、型付け規則

ここではちょっと手間をかけて、自然数と真理値、および、条件式のみからなるような小さな言語の型システムを形式的に定義し、その性質について調べてみます。

1.1 構文

ここで扱う言語の構文を以下に示します。

```
t ::=                          項：
      0                        定数ゼロ
      true                     定数真
      false                    定数偽
      succ t                   後者値
      pred t                   前者値
      iszero t                 ゼロ判定
      if t then t else t       条件式
```

[*1] SW8 の公開日！

[*2] https://twitter.com/unnohideyuki

[*3] https://qiita.com/advent-calendar/2017/haskell

これは、次のような意味になります。

- 0, true, false は、いずれも項 (term) である。
- t1 が項であるなら、succ t1, pred t1, iszero t1 はいずれも項である。
- t1, t2, t3 がいずれも項であるなら、if t1 then t2 else t3 も項である。
- この言語は、以上の 3 つの条件のうち、いずれかを満たす項のみからなる。

以下に、この言語におけるいくつかのプログラム例をみてみましょう。

```
if false then 0 else (succ 0)
```

```
iszero (pred (succ 0))
```

なお、この構文では、`succ true` や `if 0 then 0 else 0` のような項も許容されることに注意してください。

1.2 意味論

言語の構文を定義したので、次は、項がどう評価されるかを定義します。

最初に着目したいのは、項の中でも、そのものが値を示しているような特別な項です。値には、真理値 (bv; boolean value) をあらわす、true と false があります。

```
bv := true
      false
```

さらに、数値 (nv; numeric vaule) を表す項として以下のものがあります。0、もしくは、数値に後者関数 succ を適用したものが数値です。

```
nv := 0
      succ nv
```

両者をまとめるて、この言語における値 (v; vaule) は以下のように定義できます。

```
v := bv
     nv
```

次に、値以外の項の評価規則を定義していきます。評価関係は t → t' のように書かれ、「項 t が項 t' に評価される」とよみます。また、前提条件のない評価は t → t' そのものの形をとり、前提条件がある場合には、以下のように前提となる評価関係と、適用される評価関係を重ねた推論規則の形をとります。

$$\frac{\texttt{t1} \rightarrow \texttt{t1'}}{\texttt{t2} \rightarrow \texttt{t2'}}$$

では、if 文に関わる規則から。

$$\texttt{if true then t2 else t3} \rightarrow \texttt{t2} \quad \text{(E-IfTrue)}$$

$$\texttt{if false then t2 else t3} \rightarrow \texttt{t3} \quad \text{(E-IfFalse)}$$

$$\frac{\texttt{t1} \rightarrow \texttt{t1'}}{\texttt{if t1 then t2 else t3} \rightarrow \texttt{if t1' then t2 else t3}} \quad \text{(E-If)}$$

ひとつめの規則は、条件式において、検査対象の項が true そのものであった場合には t3 を評価せずに捨てて、条件式全体を t2 に起きかえることを意味しています。ふたつめは、検査対象が false だった場合で、この場合は条件式の評価結果が t3 となります。

3 つめは、検査対象の項が値でなかったケースに対応するもので、他の規則によって t1 が t1' に評価されるときには、条件式全体が if t1' then t2 else t3 へ評価されることを示しています。

残りの規則は以下の通り、算術式に関する評価規則です：

$$\frac{\texttt{t1} \rightarrow \texttt{t'}}{\texttt{succ t1} \rightarrow \texttt{succ t1'}} \quad \text{(E-Succ)}$$

$$\texttt{pred 0} \rightarrow \texttt{0} \quad \text{(E-PredZero)}$$

$$\texttt{pred (succ nv1)} \rightarrow \texttt{nv1} \quad \text{(E-PredSucc)}$$

$$\frac{\texttt{t1} \rightarrow \texttt{t'}}{\texttt{pred t1} \rightarrow \texttt{pred t1'}} \quad \text{(E-Pred)}$$

$$\texttt{iszero 0} \rightarrow \texttt{true} \quad \text{(E-IszeroZero)}$$

$$\texttt{iszero (succ nv1)} \rightarrow \texttt{false} \quad \text{(E-IszeroSucc)}$$

$$\frac{\texttt{t1} \rightarrow \texttt{t'}}{\texttt{iszero t1} \rightarrow \texttt{iszero t1'}} \quad \text{(E-IsZero)}$$

前節で示したプログラム例について、評価の様子を確認してみましょう。まずは、`if false then 0 else (succ 0)` から：

```
if false then 0 else (succ 0) → succ 0
```

E-IfFalse 規則で `if false then 0 else (succ 0)` は `succ 0` に評価されました。`succ 0` は値なので、そこで評価はおしまいです。

もう一つの例、`iszero (pred (succ 0))` については以下のようになります。最初に E-PredSucc および E-IsZero 規則によって `iszero (pred (succ 0))` → `iszero 0` のように評価され、さらに、E-IszeroZero 規則によって `true` へ評価されます。

```
iszero (pred (succ 0)) → iszero 0 → true
```

では、`succ true` や `if 0 then 0 else 0` についてはどうでしょうか。これまでの定義にてらしてみると、この 2 つはいずれも値ではなく、かといって、これらに当てはまる評価規則も存在しません。このように、値でなく、かつ、当てはまる評価規則がないような状態を **行き詰まり状態** といいます。

1.3 型付け

項を評価すると、その結果が値として得られるか、または、ある段階でどの評価規則も適用できないような**行き詰まり状態**になるかのどちらかとなります。行き詰まり状態は、実行時エラーに対応するものであり、できれば項を実際に評価せずに、その項の評価が行き詰まり状態にならないことを保証したいところです。

行き詰まり状態になるケースを観察すると、`pred`, `succ`, `iszero` などの引数に数値以外を与えようとした場合や、if 文の検査対象に真理値以外を置いてしまったケースがあることがわかります。そこで、数値に評価される項と真理値に評価される項とを、ふたつの型 `Nat`, `Bool` に分類することにします。

そういうわけで、ここでの型は Bool (真理値型）または、Nat (自然数型）です：

```
T ::= Bool
      Nat
```

「項 t が型 T をもつ」という型付け関係を `t:T` と書くこととし、この言語における型付け規則を、項に型を割り当てる推論規則の集合として定義しましょう。

$$\texttt{true} : \texttt{Bool} \quad \text{(T-True)}$$

$$\texttt{false} : \texttt{Bool} \quad \text{(T-False)}$$

$$\frac{\texttt{t1} : \texttt{Bool} \quad \texttt{t2} : \texttt{T} \quad \texttt{t3} : \texttt{T}}{\texttt{if t1 then t2 else t3} : \texttt{T}} \quad \text{(T-If)}$$

$$\texttt{0} : \texttt{Nat} \quad \text{(T-Zero)}$$

$$\frac{\texttt{t1} : \texttt{Nat}}{\texttt{succ t1} : \texttt{Nat}} \quad \text{(T-Succ)}$$

$$\frac{\texttt{t1} : \texttt{Nat}}{\texttt{pred t1} : \texttt{Nat}} \quad \text{(T-Pred)}$$

$$\frac{\texttt{t1} : \texttt{Nat}}{\texttt{iszero t1} : \texttt{Bool}} \quad \text{(T-IsZero)}$$

規則 T-True, T-False は、真理値定数 true と false に Bool 型を割り当てています。規則 T-If は、部分項 t1 の型が Bool であり、かつ、t2, t3 に同じ型 T が割り当てられる場合に限って、if 文全体に型 T が割り当てられることを主張しています。

これらの型付け規則によって、たとえば項 `if true then false else false` は Bool 型を持ち、`pred(succ(pred(succ 0)))` は Nat 型を持つことが、項の値を評価することなくわかりました。このようなに「項の値を実際に計算することなく」行う分析のことを、**静的な**分析といいます。

なお、実際に値を計算せずに行う型分析は、保守的になります。つまり、`if (iszero 0) then 0 else false` や `if true then 0 else false` といった項は、実際には行き詰まり状態にはならないにも関わらず、どんな型も持てません。

1.4 安全性

型システムの目的は、正しく型付けされた項はおかしくならない、つまり、行き詰まり状態にならないことを保証することです。この性質のことを**安全性**といいます。この性質は、**進行**と**保存**という 2 つのステップに分けて証明できることができます[*4]。

- **進行**とは、正しく型付けされた項は行き詰まり状態ではない（値であるか、評価規則によって評価を進められる）ということ
- **保存**とは、正しく片づけされた項が評価できるならば、評価後の項も正しく型付けされるということ

これらの性質をあわせると、正しく型付けされた項は評価において行き詰まり状態になりえないことがいえます。

[*4] 各々の証明は、あとで別頁に書くかもしれません。または、TaPL の 8.3 節を読もう！

2 Hindley/Milner の型推論

前の章では、算術と条件式からなる小さな言語に対する型検査を通じて、型システムの概要をみてきました。つぎに、本章では、もう少し大きな言語に対する型推論について述べます。

2.1 構文

本章で扱うプログラミング言語の構文を以下に示します。

この言語の項 t を次のように定めます。ただし x は変数を表します。

`t ::=`	項：
`x`	変数
`0`	定数ゼロ
`true`	定数真
`false`	定数偽
`succ t`	後者値
`if t then t else t`	条件式
`[]`	空リスト
`t : t`	リスト構築
`null t`	空リストの判定
`head t`	リストの先頭の要素
`tail t`	リストの先頭要素を除いた残り
`let x = t in t`	let 束縛
`\x -> t`	ラムダ抽象
`t t`	関数適用

前章で扱ったのと同様の算術と条件式に加えて、リスト、let 束縛、ラムダ抽象、および、関数適用があります。

2.2 操作的意味論

この言語における値には、以下に示すものがあります：

`bv ::=`	真理値：
`true`	真
`false`	偽
`nv ::=`	数値：
`0`	ゼロ
`succ nv`	後者値

v ::=	値：
bv	真理値
nv	数値
\x -> t	ラムダ抽象値
[]	空のリスト
v : v	リスト構築子

次に、評価規則です。条件式と算術に関するものは、前章と共通です。

$$\texttt{if true then t2 else t3} \rightarrow \texttt{t2} \quad \text{(E-IfTrue)}$$

$$\texttt{if false then t2 else t3} \rightarrow \texttt{t3} \quad \text{(E-IfFalse)}$$

$$\frac{\texttt{t1} \rightarrow \texttt{t1'}}{\texttt{if t1 then t2 else t3} \rightarrow \texttt{if t1' then t2 else t3}} \quad \text{(E-If)}$$

$$\frac{\texttt{t1} \rightarrow \texttt{t'}}{\texttt{succ t1} \rightarrow \texttt{succ t1'}} \quad \text{(E-Succ)}$$

次に、新しい評価規則です。まずは、リストに関するものです。

$$\frac{\texttt{t1} \rightarrow \texttt{t1'}}{\texttt{t1 : t2} \rightarrow \texttt{t1' : t2}} \quad \text{(E-Cons1)}$$

$$\frac{\texttt{t2} \rightarrow \texttt{t2'}}{\texttt{v : t2} \rightarrow \texttt{v : t2'}} \quad \text{(E-Cons2)}$$

$$\texttt{null []} \rightarrow \texttt{true} \quad \text{(E-IsnilNil)}$$

$$\texttt{null (v : v)} \rightarrow \texttt{false} \quad \text{(E-IsnilCons)}$$

$$\frac{\texttt{t1} \rightarrow \texttt{t1'}}{\texttt{null t1} \rightarrow \texttt{null t1'}} \quad \text{(E-Isnil)}$$

$$\texttt{head (v1 : v2)} \rightarrow \texttt{v1} \quad \text{(E-HeadCons)}$$

$$\frac{\texttt{t1} \rightarrow \texttt{t1'}}{\texttt{head t1} \rightarrow \texttt{head t1'}} \quad \text{(E-Head)}$$

$$\texttt{tail (v1 : v2)} \rightarrow \texttt{v2} \quad \text{(E-TailCons)}$$

$$\frac{\texttt{t1} \rightarrow \texttt{t1'}}{\texttt{tail t1} \rightarrow \texttt{tail t1'}} \quad \text{(E-Tail)}$$

残りの評価規則を示す前に、代入を表す記法を導入します。変数 x へ s を代入することを $[\texttt{x} \mapsto \texttt{s}]$ と書くこととします。これを用いると、$[\texttt{x} \mapsto \texttt{s}]\texttt{t}$ は、項 t における変数 x の出現を s に置き換えたものを示します。

たとえば、$[\texttt{x} \mapsto \texttt{0}]\texttt{(succ x)}$ は、succ 0 となります。

なお、代入については、束縛変数の扱いについて考慮が必要なのですが、煩雑なのでここでは省略します。

では、残りの、let 束縛と関数に関わる評価規則を示しましょう。

$$\texttt{let x=v1 in t2} \to [\texttt{x} \mapsto \texttt{v1}]\texttt{t2} \quad \text{(E-LetV)}$$

$$\frac{\texttt{t1} \to \texttt{t1'}}{\texttt{let x=t1 in t2} \to \texttt{let x=t1' in t2}} \quad \text{(E-Tail)}$$

$$\frac{\texttt{t1} \to \texttt{t1'}}{\texttt{t1 t2} \to \texttt{t1' t2}} \quad \text{(E-App1)}$$

$$\frac{\texttt{t2} \to \texttt{t2'}}{\texttt{v1 t2} \to \texttt{v1 t2'}} \quad \text{(E-App2)}$$

$$(\lambda\texttt{x -> t1})\ \texttt{v2} \to [\texttt{x} \mapsto \texttt{v2}]\texttt{t1} \quad \text{(E-AppAbs)}$$

この言語においても、項を評価すると、結果的に値が得られるか、または、行き詰まり状態になるかのどちらかとなります。

たとえば、`tail false` は値ではなく、かつ、当てはまる評価規則がないので行き詰まり状態です。

2.3 制約に基づいた型付け

まず、この言語では、変数が登場するため、型付け関係を t:T の二項関係から、$\Gamma \vdash \texttt{t:T}$ の三項関係に変更されます。ここで、Γ は t に現れる自由変数の型に関する仮定です。

前章での型付け規則は、型を検査して、項の型を決めるような規則の集まりでした。例えば (T-If) は、t1 の型が Bool で、t2, t3 が同じ型 T であることを要請しつつ、条件式全体の型を T と出力する規則だったといえます。

一方で、型推論システムにおいては、型に関する規則は、次のような形をとります。

$$\frac{\Gamma \vdash \texttt{t1} : \texttt{T1} \mid C_1 \quad \Gamma \vdash \texttt{t2} : \texttt{T2} \mid C_2 \quad \Gamma \vdash \texttt{t3} : \texttt{T3} \mid C_3}{\Gamma \vdash \texttt{if t1 then t2 else t3} : \texttt{T2} \mid C_1 \cup C_2 \cup C_3 \cup \{T1 = Bool, T2 = T3\}} \quad \text{(CT-If)}$$

新しく登場した、縦棒の後ろの C は、制約の集合です。この制約型付け規則 CT-IF は、前提条件としては、t1, t2, t3 の型を T1, T2, T3 としており、つまり、これらが特定の型であることを要請していません。その代わり、規則を適用した結果として、T1 = Bool, T2 = T3 という 2 つの新たな制約を、制約集合に追加します。

（未稿。この言語の制約型付け規則をここに示す。）

2.4 単一化

制約集合の解を計算するには、Hindley と Milner のアイディアを用います。これは、単一化を用いて解の集合が空でないことを検査し、空でないなら、「最良の」解を見つける方法です。

（二章は以下未稿。。。[*5]）

[*5] 今年中に書けるかな。

3. Haskell による実装

3.1 モジュール宣言、ライブラリのインポート

簡単のために、型推論のコードは単一のモジュールとして提示します。

また、プログラムにおいては Haskell の豊富な語彙のうちでも基本的なものだけを使いました。標準ライブラリ Prelude に含まれるもの以外では、`Data.List` より `nub`, `(\\)`, `union` の 3 関数、そして、型推論モナドのために State ライブラリより `State`, `runState`, `get`, `put` の 4 関数のみを用いています。

```
module HMTI where
import Data.List (nub, (\\), union)
import Control.Monad.State.Strict (State, runState, get, put)
```

識別子を表現するためには `String` を用いました。また、整数値から識別子を生成するための方法を用意しておきます。以下に示す `enumId` がこれで、後述する `newTVar` 関数から用いられます。

```
type Id = String

enumId :: Int -> Id
enumId n = "a" ++ show n
```

3.2 項

2.1 節で示した抽象構文を表現するための `Term` は以下のようになります。ここでは、簡単のために、定数を定義済みの変数として扱うことにしています。

```
data Term = Var Id
          | Lam Id Term
          | App Term Term
          | Let Id Term Term
          | If Term Term Term
          deriving Show
```

3.3 型

次に、型の表現を定義します。型には、型変数 (TVar)、型コンストラクタ (TCon)、および、一つの型に他のもう一つの型を適用したもの (TAp) があります。

```
data Type = TVar Tyvar
          | TCon Tycon
          | TAp Type Type
          | TGen Int
          deriving (Show, Eq)

data Tyvar = Tyvar Id deriving (Show, Eq)
data Tycon = Tycon Id deriving (Show, Eq)
```

型定義には、量化された型変数をあらわす TGen n が含まれていますが、この TGen は、後に示す Scheme の中においてのみ用いられます。

以下に、いくつか型コンストラクタの実例を示します。

```
tBool :: Type
tBool = TCon (Tycon "Bool")

tInt :: Type
tInt = TCon (Tycon "Int")

tList :: Type
tList = TCon (Tycon "[]")

tArrow :: Type
tArrow = TCon (Tycon "->")
```

より複雑な型は、TAp を用いて定数や変数から構築されます。たとえば、Int -> [a] 型の表現は次のようになります[*6]。

```
TAp (TAp tArrow tInt) (TAp tList (TVar (Tyvar "a")))
```

このような関数型の構築は度々行うため、次に示すようなヘルパー関数を用意しておきます。これを使うと、上の例は `tInt `fn` TAp tList (TVar (Tyvar "a"))` のように書けます。

[*6] Scala By Example では Type 定義のなかに、特別な型として関数型を定義してありました。ここでは Typing Haskell in Haskell と同様に、"->" を型コンストラクタとすることで、関数型も一般的に扱っています。

```
infixr 4 `fn`
fn :: Type -> Type -> Type
a `fn` b = TAp (TAp tArrow a) b
```

3.4 型代入 (Substitution)

型代入は、型変数から型への写像をあらわす有限関数（有限な定義域をもつ部分関数）であり、第一要素に重複がないようなペアのリストとして表現できます。

```
type Subst = [(Tyvar, Type)]

nullSubst :: Subst
nullSubst = []

(+->) :: Tyvar -> Type -> Subst
u +-> t = [(u, t)]
```

最も単純なのは、null 代入で、つぎに単純な代入は、ひとつの型変数 `u` を型 `t` に移す `u +-> t` です。

型は型変数に適用します。ですが、それだけでなく、型を要素に含む（つまり型変数を含み得る）他の値にも適用できるようにしておくのが便利です。これは、つまり、型代入の適用関数 `apply` を異なる対象に使えるようにしておくと便利だということですね。

```
class Types t where
  apply :: Subst -> t -> t
  tv :: t -> [Tyvar]
```

いずれの対象に用いる場合でも型代入を適用する目的は同じで、型代入の定義域にある型変数の出現を対応する型に置き換えることです。さらに、引数中に現れる型変数の集合を返す関数 `tv` も用意します。これは、出現する順（左から右）に重複のないリストとして返されます。

まず、型を対象にした定義。

```
instance Types Type where
  apply s (TVar u) = case lookup u s of
                       Just t -> t
                       Nothing -> TVar u
  apply s (TAp l r) = TAp (apply s l) (apply s r)
  apply _ t = t
```

```
  tv (TVar u) = [u]
  tv (TAp l r) = tv l `union` tv r
  tv _ = []
```

リストに適用できるようにしておくのは簡単で、かつ便利です。

```
instance Types a => Types [a] where
  apply s = fmap $ apply s
  tv = nub.concat.fmap tv
```

apply 関数は、より複雑な型代入をつくるときにも使われます。例えば、apply (a1 @@ s2) が apply s1 . apply s2 と等しくなるような複合関数はつぎのように書けます。

```
infixr 4 @@
(@@) :: Subst -> Subst -> Subst
s1 @@ s2 = [(u, apply s1 t) | (u, t) <- s2] ++ s1
```

3.5 型スキーム

多相型をあらわすために、型スキームを定義します。Forall m t において、t に出現する TGen n は量化された（全称型の）型変数です。m は t に含まれる TGen の個数を示します。この情報は型 t を分析すれば得られるため冗長なのですが、後に、型スキームを実体化する際に便利なため、型スキームの定義に含めました。

```
data Scheme = Forall Int Type
            deriving Show

instance Types Scheme where
  apply s (Forall n t) = Forall n (apply s t)
  tv (Forall _ t) = tv t
```

たとえば、型スキーム $\forall a \forall b.a \to b$ は、次のように書けます。

```
Forall 2 (TGen 0 `fn` TGen 1)
```

TGen がゆるされるのは、型スキームの中だけです。また、量化された型変数は型スキームにおいて束縛されているので、型代入の影響を受けません。これらをどうにか強制できないかということで、このような定義になっています。

3.6 仮定 (Assumption)

変数の型に関する仮定は Assump データ型で表現されます。仮定の各要素は、変数名と型スキームのペアです。

```
data Assump = Id :>: Scheme
            deriving Show
```

これも、Types クラスのインスタンスにして、型代入を適用できるようにしておきます。

```
instance Types Assump where
  apply s (i :>: sc) = i :>: (apply s sc)
  tv (_ :>: sc) = tv sc
```

仮定に関しても、いくつかヘルパー関数を用意します。引数の型に含まれる型変数をすべて量化して型スキームをつくる gen 関数と、仮定の集合から、特定の変数の型を探すための find 関数です。

```
gen :: [Assump] -> Type -> Scheme
gen as t = Forall n (apply s t)
  where gs = tv t \\ tv as
        n = length gs
        s = zip gs (map TGen [0..])

find :: Monad m => Id -> [Assump] -> m Scheme
find i [] = fail $ "unbound identifier: " ++ i
find i ((i':>:sc):as) = if i==i' then return sc else find i as
```

3.7 単一化

単一化の目的は、２つの型が一致するような型代入を見つけることです。単一化においては、なるべく小さい型代入を見つけることが重要で、そうすることで、より一般的な型を得ることができます。

２つの型 t1, t2 に対し、`apply s t1` と `apply s t2` が等しくなるような型代入 s を t1 と t2 の単一化子 (unifier) といいます。

さらに、最汎単一化子 (most general unifier, mgu) とは、２つの型の単一化子 u であって、かつ、次の性

質をもつものです[*7]：

- その２つの型の任意の単一化子 s に対して、s が s'@@u と等しくなるような s' が存在する。

今回対象とする言語の mgu は以下のように計算されます：

```
mgu :: Monad m => Type -> Type -> m Subst
varBind :: Monad m => Tyvar -> Type -> m Subst

mgu (TAp l r) (TAp l' r') = do s1 <- mgu l l'
                               s2 <- mgu (apply s1 r) (apply s1 r')
                               return (s2@@s1)
mgu (TVar u) t = varBind u t
mgu t (TVar u) = varBind u t
mgu (TCon tc1)(TCon tc2) | tc1 == tc2 = return nullSubst
mgu _ _ = fail "types do not unify"

varBind u t | t == TVar u   = return nullSubst
            | u `elem` tv t = fail "occurs check fails"
            | otherwise     = return (u +-> t)
```

3.8 型推論モナド

モナドは、ある種の「配管」を隠すことで、プログラム設計のより重要な面に注意を向ける用途で広く用いられています。今回の型推論でも、その目的で状態モナドを用います。

型推論モナドは、現在の型代入と「フレッシュな」変数を生成するための整数値の２つを状態として持ちます。

```
type TI a = State (Subst, Int) a

runTI :: TI a -> a
runTI ti = x where (x, _) = runState ti (nullSubst, 0)
```

状態モナドから、現在の代入を返すのが getSubst 関数です。

```
getSubst :: TI Subst
```

[*7] 平たくいうと、単一化子のなかでも最も小さいもの、つまり、最低限の置き換えしかしないような型代入が mgu です。

```
getSubst = do (s, _) <- get
              return s
```

引数として与えた２つの型の最汎単一化子で、現在の型代入を拡張するのが unify 関数です。

```
unify :: Type -> Type -> TI ()
unify t1 t2 = do (s, n) <- get
                 u <- mgu (apply s t1) (apply s t2)
                 put (u@@s, n)
```

もうひとつの状態変数を用いて、あたらしい型変数を生成する関数が newTVar です。

```
newTVar :: TI Type
newTVar = do (s, n) <- get
             put (s, n+1)
             return (TVar (Tyvar (enumId n)))
```

newTVar が用いられる場面のひとつが、型スキームを新しい型変数とともに実体化するときです。

```
freshInst :: Scheme -> TI Type
freshInst (Forall n t) = do ts <- sequence $ replicate n newTVar
                            return (inst ts t)

class Instantiate t where
  inst :: [Type] -> t -> t

instance Instantiate Type where
  inst ts (TAp l r) = TAp (inst ts l) (inst ts r)
  inst ts (TGen n) = ts!!n
  inst _ t = t
```

3.9 型推論

型推論の主な仕事を tiTerm 関数に実装していきます。この関数は、仮定 as, 項 e, 型のプロトタイプ t を引数にとり、現在の型代入[*8]を拡張することで $\sigma\Gamma \vdash e : \sigma t$ を満たすような型代入 σ を計算します。そのような型代入が存在しない場合には、モナド標準の fail 関数で計算が失敗します。

```
tiTerm :: [Assump] -> Term -> Type -> TI ()

tiTerm as (Var x) t = do sc <- find x as
                         t' <- freshInst sc
                         unify t' t

tiTerm as (Lam x e) t = do a <- newTVar
                           b <- newTVar
                           unify (a `fn` b) t
                           let as' = (x :>: Forall 0 a):as
                           tiTerm as' e b

tiTerm as (App e1 e2) t = do a <- newTVar
                             tiTerm as e1 (a `fn` t)
                             tiTerm as e2 a

tiTerm as (Let x e1 e2) t = do a <- newTVar
                               tiTerm as e1 a
                               s <- getSubst
                               let as' = (x :>: gen as (apply s a)):as
                               tiTerm as' e2 t

tiTerm as (If e1 e2 e3) t = do tiTerm as e1 tBool
                               a <- newTVar
                               tiTerm as e2 a
                               tiTerm as e3 a
                               s <- getSubst
                               unify (apply s a) t
```

次に示す typeOf 関数は、tiTerm を用いて、与えられた仮定 as のもとで項 e の型を計算します。

[*8] 「現在の型代入」は、型推論モナドが状態として保持しています。

```
typeOf :: [Assump] -> Term -> TI Type
typeOf as e = do a <- newTVar
                 tiTerm as e a
                 s <- getSubst
                 return (apply s a)
```

なお、型推論を行うにあたっては、組み込み定数に関する仮定をひとまとめに定義しておくと便利です。

```
prims :: [Assump]
prims = [ "true"   :>: gen' tBool
        , "false"  :>: gen' tBool
        , "zero"   :>: gen' tInt
        , "succ"   :>: gen' (tInt `fn` tInt)
        , "pred"   :>: gen' (tInt `fn` tInt)
        , "iszero" :>: gen' (tInt `fn` tBool)
        , "[]"     :>: gen' listTy
        , ":"      :>: gen' (a `fn` listTy `fn` listTy)
        , "null"   :>: gen' (listTy `fn` tBool)
        , "head"   :>: gen' (listTy `fn` a)
        , "tail"   :>: gen' (listTy `fn` listTy)
        ]
  where gen' = gen []
        a = TVar (Tyvar "a")
        listTy = (TAp tList a)
```

最後に、これまで定義した関数の使い方の例として、与えられた項の型を返す関数を示します。

```
testInfer :: Term -> Type
testInfer e = runTI $ typeOf prims e
```

試しに、ghci を起動して実行してみましょう：

```
$ ghci HMTI.hs
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
```

```
Loading package base ... linking ... done.
[1 of 1] Compiling HMTI             ( HMTI.hs, interpreted )
Ok, modules loaded: HMTI.
*HMTI> testInfer (Lam "x" (App (App (Var ":") (Var "x")) (Var "[]")))
TAp (TAp (TCon (Tycon "->")) (TVar (Tyvar "a4"))) (TAp (TCon (Tycon "[]")) (TVar (Tyvar "a4")))
*HMTI>
```

ちょっとわかりにくいですが、`\x -> (x:[])` の型として、`a4->[a4]` が計算されたことがわかります。

まとめと参考文献など

Hindley/Milner の型推論について、その理論と Haskell による実装を解説してみようと思ったのですが、ちょっと荷が重かったかも。

そういうわけで、型システムを学ぶには、以下の教科書を読むのがいいです。

- 型システム入門　プログラミング言語と型の理論（通称 TaPL）、Benjamin C. Pierce 著、オーム社
- 情報科学における論理、小野 寛晰著、日本評論社

後者は、型理論の教科書ではないのですが、形式体系による推論になじみのないひとで、どんな教科書で勉強するのかわからん！という人にはオススメです。

型推論の実装については、Scala By Example[*9] の 16 章を元にしました。ただ、それを Haskell に「移植」するのではなくて、Typing Haskell in Haskell の設計やプログラミングスタイルにあわせたつもりです。コードの全体は GitHub に置いてあります。[*10]

なんか、長いわりに不完全な記事になってしまいましたが、ひとまずおしまいです[*11]。

次は、Typing Haskell in Haskell に再挑戦する前に、How to make ad-hoc polymorphism less ad hoc の appendix を読みたい！今なら読めるかも知れない。

では。

[*9] `http://www.scala-lang.org/docu/files/ScalaByExample-ja_JP.pdf`

[*10] `https://github.com/unnohideyuki/typeinference`

[*11] ちょっとずつ書き足すかも知れません。