ノート: Haskell の型付けを Haskell で (Typing Hasell in Haskell)

Typing Haskell in Haskell[*1] を勉強したときのノート（紙に書いてた）を公開します。全訳するつもりではなかったのですが、結局ほとんど訳すことになってしまった…。

原文を読む際の助けになれば幸いです。また、誤りを発見された場合には、お知らせいただけると助かります。

- 原文が書かれた日： 2000 年 11 月 23 日
- この日本語ノートの公開日： 2016 年 12 月 6 日
- この日本語ノートを書いた（訳した）人：@unnohideyuki[*2]

1. 導入

（略）

2. Preliminaries

本稿では、識別子の実装には文字列 (String) を用いることとし、enumId 関数を使って自然数から識別子を単純に生成できるものとする。

```
type Id = String
enumId :: Int -> Id
enumId n = "v" ++ show n
```

3. Kinds (種≒型の型)

(Kind については、Haskell 2010 report の 4.1.1 や 4.6 も見よう。)

それらが有効であることを保証するために、Haskell における型構築子はそれぞれの種 (Kind) によって分類される。

`*` (「スター」と読む) は単純な型を表す種で、`Int` や `Char -> Bool` などの種はいずれも `*` である。`k1 -> k2` の形の種は、型引数をとるような型構築子を表す種。たとえば、リストや `Maybe`, `IO` は、いずれも `* -> *` の種を持つ。

```
data Kind = Star | Kfun Kind Kind
            deriving Eq
```

[*1] http://web.cecs.pdx.edu/~mpj/thih/

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

種の型構築子に対する役割は、型が値に対するものと本質的に同じであるが、種システムは型システムに比べると明らかにずっと単純である。

4. 型

糖衣構文を取り除いたあとの Haskell の型表現は以下のいずれか：

- 型変数
- 型定数
- 型を他の型に適用したもの（`k1 -> k2` の種をもつ型を `k1` 種の型に適用すると、`k2` 種の型になる）

```
data Type = TVar Id Kind
          | TCon Id Kind
          | TAp Type Type
          | TGen Int

data Tyvar = Tyvar Id Kind
             deriving Eq

data Tycon = Tycon Id Kind
             deriving Eq
```

`TGen` は Generic な、または、量化された (quantified) 型変数をあらわす。ここで、量化型変数をそれ以外の型変数と区別しているのは、後述する型代入が束縛変数を捕獲しないようにするためである（型代入の適用は、`TGen` を無視する）。

以下の例は、標準の組み込みデータ型がどのように型定数として表現されるのかを示している。

```
tUnit    = TCon (Tycon "()" Star)
tChar    = TCon (Tycon "Char" Star)
tInt     = TCon (Tycon "Int" Star)
tInteger = TCon (Tycon "Integer" Star)
tFloat   = TCon (Tycon "Float" Star)
tDouble  = TCon (Tycon "Double" Star)

tList    = TCon (Tycon "[]" (Kfun Star Star))
tArrow   = TCon (Tycon "(->)" (Kfun Star (Kfun Star Star)))
tTuple2  = TCon (Tycon "(,)" (Kfun Star (Kfun Star Star)))
```

Haskell のコンパイラ、または、インタプリターは、各型定数に付加的な情報を格納するかもしれない。例えば、代数データ型に対しては、構築関数のリストなどである。しかし、それらの詳細は型検査の間は必要でない。

より複雑な型は、TAp を用いて定数、変数から構築される。たとえば、Int -> [a] 型の表現は以下のようになる。

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

型シノニムに対する表現は用意しない。ここでは、それらは型検査に先だって、すべて展開されているものとする。例えば、String 型（これは [Char] のシノニム）は次のように表現される；

```
tString :: Type
tString = list tChar
```

コンパイラの実装がシノニムをこのように取り扱うことは、常に可能である。なぜなら、Haskell はシノニムを引数が不足しているまま用いる（部分適用）ことを禁じているからである。また、再帰的なシノニム定義も禁じられているため、処理が完了することも保証される。ただ、現実には、シノニムの展開はもっと後まで遅延されることが多いだろう。型エラーの診断メッセージは、展開後の型ではなくシノニムを示した方が分かりやすくなる。

この節の最後に、いくつかの便利なヘルパー関数を示す。最初の 3 つは、関数、リスト、および、ペアを構築する単純な方法を提供する：

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

list        :: Type -> Type
list t      = TAp tList t

pair        :: Type -> Type -> Type
pair a b    = TAp (TAp tTuple2 a) b
```

さらに、kind を多重定義して、型変数、型構築子、または、型表現の種を決定するのに用いる。

```
class HasKind t where
```

```
  kind :: t -> Kind
instance HasKind Tyvar where
  kind (Tyvar v k) = k
instance HasKind Tycon where
  kind (Tycon v k) = k
instance HasKind Type where
  kind (TCon tc) = kind tc
  kind (TVar u)  = kind u
  kind (TAp t _) = case (kind t) of
                     (Kfun _ k) -> k
```

ほとんどのケースは素直なものである。ただ、適用 (TAp t t') の種が最初の引数 t の種のみを用いることに注意。うまく構成された (well-formed) 型である限り、t' の種が k' であるなら、t は k' -> k の種を持たなければならず、そして適用全体の種は k である。このことは、型表現の種を計算するには、最も左の脊椎のみたどれば良いことを示している（？[*3]）。

5. 型代入 (substitution)

型代入とは、型変数から型への写像をあらわす有限関数である。有限関数とは、有限な定義域をもつ部分関数のことで、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る。型代入は、型推論において中心的な役割をはたす。

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

正しい形をした (well-formed) 型表現のみを扱うため、型代入は種を保存する (kind-preserving) ように構成する。つまり、型代入によって、型は同じ種をもつ型にのみ写される。

最も単純なのは、null 型代入 (`nullSubst = []`) であり、何もしないので明らかに kind-reserving である。

同じくらい単純な型代入に `u +-> t` がある。これは、ひとつの型変数 `u` を同じ種の型 `t` に写す。

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

この型代入が kind-preserving となるための必要十分条件は、 `kind u = kind t` を満たすことである。

型代入は型変数に適用できる。それだけでなく、実のところ、型を要素に含む他の値にも自然に適用できる。これの示すところは、型代入の適用演算 (`apply` 関数）を異なる対象につかえるように多重定義しておくのがいいということだ。

[*3] とりあえず訳してみましたが、この文の意味がよくわからなかった

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

いずれの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ある。引数中に現れる型変数の集合を返す関数 tv も用意する。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る。

Types 型クラスの Type インスタンス定義は次のようになる：

```
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 s t = t

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

リストにも適用できるようにしておくのは簡単だし、そうしておくと便利。

```
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用いられる。たとえば、apply (s1 @@ s2) が apply s1 . apply s2 と等しくなるような複合代入は次のように書ける：

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

これとは別に、代入の「並列」合成を s1 ++ s2 のように作ることもできる。しかし、この複合は左に偏っていて、s1 に含まれる対応が s2 よりも優先される。

もっと偏りのない合成としては、merge 関数を用いることにする。この関数は、二つの代入の意見があっていること、つまり、両方の定義域にある型変数に対して 2 つの代入がおなじ対応をあたえることをチェックする (apply (s1++s2) と apply (s2++s1) が等しいことを保証)。

この merge は明らかに部分関数になる（与えられ引数によっては失敗する）ので、モナドの中に値を返すようにして、関数が定義されないときには標準の fail で診断メッセージをつける。

```
merge :: Monad m => Subst -> Subst -> m Subst
merge s1 s2 = if agree then return (s1++s2) else faile "(略)"
   where agree = all (\v -> apply s1 (TVar v) == apply s2 (TVar v))
                     (fmap fst s1 `intersect` fmap fst s2)
```

なお、(@@) や merge が kind-preserving であることを確認するのは容易なので省略。

次の節では、これらの 2 つの複合代入のうち前者が単一化に、後者がマッチングに用いられることを述べる。

6. 単一化とマッチング

単一化 (unification) 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ある。たとえば、関数の適用において、関数の定義域の型と引数の型を一致させたいといった場合がある。

さらに、単一化においては、なるべく小さい代入を見つけることが大事で、それによって、より一般的な型を導く。

2 つの型 t1, t2 に対し、以下を満たす型代入を t1 と t2 の単一化子 (unifier) という。

```
apply s t1 = apply s t2
```

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： 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' がある。

Haskell の言語仕様は、 2 つの型が単一化子をもつときには必ず mgu が存在するように設計してある。

もし、型システムを拡張して高階にしたり（型についてもラムダ式を導入するなど)、リダクションをゆるすような仕組みをいれたり、型言語を変更するなどした場合には、単一化子が決定不能になったり、単一化できる場合にも mgu がないケース (non-unitary) が生じるかもしれない。たとえば、Haskell で型シノニムの部分適用（これは限定されたラムダ式にあたる）が許されていないのは、このためである。

```
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 unifiy."

varBind u t | t == Var u       = return nullSubst
            | u `elem` tv t    = fail "occures check fails."
            | kind u /= kind t = fail "kind do not match."
            | otherwise        = return (u +-> t)
```

また、次節以降では、単一化に近い演算である「マッチング」と呼ばれる操作も用いる。マッチングの目的は、2つの型 t1, t2 に対して `apply s t1 = t2` を満たす型代入 s を見つけることである。(s が片方のみに適用されるので、マッチングは「片方向マッチング」と呼ばれることもある)

```
match :: Monad m => Type -> Type -> m Subst
match (TAp l r) (TAp l' r') = do sl <- match l l'
                                 sr <- match r r'
                                 merge sl sr
match (TVar u) t | kind u == kind t = return (u +-> t)
match (TCon tc1) (TCon tc2) | tc1 == tc2 = return nullSubst
match _ _ = fail "types do not match"
```

mgu の定義とよく似ているが、こちらでは `(@@)` でなく `merge` を代入の合成につかう。

7. 型クラス、述語、および、修飾された型 (Qualified Types)

ML のような他の多相な静的型付け言語と比較したとき、Haskell の最も際立った特徴は、型クラスである。Wadler と Blott がいくつかのアドホックな多重定義[*4]を包含する機構として示して以来、型クラスはさまざまに用いられてきた。

本稿に示すコードの大半は、この型クラスを扱うためのものだといえる。

また、型システムを複雑にする要因には、他にも以下のようなものたちがある：

- 暗黙の型付けと明示的な型付けの混在
- 相互再帰束縛

[*4] アドホック多相とは、1967 年に Christopher Strachey が分類した2つの多相のうちの片方。3*3 と 3.14*3.14 の両方に同じ掛け算演算子 * を用いる場合のように、複数の型に対して同名の関数が定義され、かつ、関数は各々の型に対してことなる振る舞いを持つような多相のこと。もうひとつはパラメトリック多相で、こちらは、length :: [a] -> Int のように、ひとつの関数が複数の型に対して全くおなじ振る舞いをもつような多相である。

- パターンマッチング

これらは、理論を提示する際には省略されることも多いが、本稿ではすべて扱う。

7.1 型クラスの基本の定義

Haskell の型クラスは、ある特定のメンバ関数群をクラス定義の一部としてもつ、同じ種の型の集合とみなすことができる。

型クラスに含まれる型（インスタンスと呼ばれる）は、インスタンス宣言のあつまりによって定義される。

また、Haskell の型は、述語のリスト（空かもしれない）、つまり、クラス制約によって修飾することができ、そうすることで型変数の実体化方法を制限できる。

```
data Qual t = [Pred] :=> t
              deriving Eq
```

`ps :=> t` の形をした値において、ps を文脈 (context), t を頭部 (head) と呼ぶ。

述語には、クラス識別子と型が含まれる。`Isin i t` は t が i という名のクラスのメンバであることを表す。

```
data Pred = IsIn Id Type
            deriving Eq
```

(例) `Num a => a -> Int` を表す修飾された型は次のようになる：

```
[IsIn "Num" (TVar (Tyvar "a" Star))] :=> (TVar (Tyvar "a" Star) `fn` tInt)
```

なお、Types は容易に Qual や Pred に拡張できる：

```
instance Types t => Types (Qual t) where
  apply s (ps :=> t) = apply s ps :=> apply s t
  tv (pa :=> t) = tv ps `union` tv t

instance Types Pred where
  apply s (IsIn i t) = IsIn i (apply s t)
  tv (IsIn i t) = tv t
```

最汎単一化子やマッチング代入の計算も、自然に述語へと拡張できる。

```
mguPred, matchPred :: Monad m => Pred -> Pred -> m Subst

mguPred = lift mgu
matchPred = lift match

lift m (IsIn i t) (IsIn i' t')
  | i == i'   = m t t'
  | otherwise = fail "classes differ"
```

クラスはリストのペアで表すことにする。片方はスーパークラス名のリスト、他方はインスタンス宣言を含む。

```
type Class = ([Id], [Inst])
type Inst = Qual Pred
```

なお、ここでの定義には、型推論に必要な情報しか含まれていない。実際にコンパイラを実装するときには、メンバ関数のリストや、各クラスむけのメンバ関数定義のなかみも必要。

7.2 クラス環境

与えられたプログラムにおいて、クラスインスタンス宣言によってもたらされる情報は、次に示すようなデータ型をもつクラス環境に保持される：

```
data ClassEnv = ClassEnv { classes :: Id -> Maybe Class
                         , defaults :: [Type]
                         }
```

classes 要素は、識別子をクラス値へと写す部分関数[*5]（識別子に対応するクラスがなければ Nothing を返す)。スーパークラスやインスタンスを抽出するためのヘルパー関数も定義しておきます。

```
super :: ClassEnv -> Id -> [Id]
super ce i = case classes ce i of Just (is, _) -> is

insts :: ClassEnv -> Id -> [Inst]
insts ce i = case classes ce i of Just (_, its) -> its
```

[*5] 失敗したときにランタイムエラーを起こすようなものを部分関数といい、適切な値が計算されない場合にも値 (Nothing など）を返すような関数は部分関数とはいわないこともあるようです。この論文では、そういう Haskell でインプリするときの関数の型について述べているのではなく、もともとの写像の定義域のことをいっているんですよね。

これらのコードは、i が示す名前のクラスが定義済であることを前提にしています。型チェックの前段における静的解析によってこれが保証されることも多く、そうでないなら、次に示すような defined で検査できる：

```
defined :: Maybe a -> Bool
defined (Just _) = True
defined Noghing = False
-- Data.Maybe をつかえば defined = isJust
```

クラス環境を更新して、クラス値と識別子のあらたな束縛を反映していくためのヘルパー関数、modify を定義しておく[*6]。

```
modify :: ClassEnv -> Id -> Class -> ClassEnv
modify ce i c = ce{classes = \j -> if i == j then Just c else classes ce j}
```

ClassEnv の default 要素は、11.5.1 節で述べる defaulting 向けの型リストを提供します。Haskell では、default 宣言でこの内容を明に指定することができます。明に指定しなかった場合は

```
default (Integer, Double)
```

が仮定されます。

本節 (7.2) の残りの部分では、ほとんど空っぽのクラス環境から始めて、与えられたプログラムの適切なクラス環境をいかにつくるか、また、クラス宣言やインスタンス宣言に応じて拡張していくのかについて述べる。

初期のクラス環境は次のとおり：

```
initialEnv :: ClassEnv
initialEnv = ClassEnv{ classes = \i -> fail "class not defined."
                     , defaults = [tInteger, tDouble]
                     }
```

プログラム中のクラス宣言／インスタンス宣言を処理していくにつれて、この初期クラスを変換し、新しいエントリ（新しいクラスか、新しいインスタンス）を足していきます。このために、クラス環境の変換を環境変換子、 EnvTransformer の形で示していくことにします。

[*6] このように定義すると、ClassEnv を Show しづらくなるので、私は Data.Map をつかいました。

```
type EnvTransformer = ClassEnv -> Maybe ClassEnv
```

複数の変換子をつなげるのは、（前向きの）合成演算、(<:>) で書ける：

```
infixr 5 <:>
(<:>) :: EnvTransformer -> EnvTransformer -> EnvTransformer
(f <:> g) ce = do ce' <- f ce
                  g ce'
```

これは、Left-to-right Kleisli composition operator (Control.Monad.(>=>)) を特殊化したものになっている。

新しくクラスを環境に加えるには、同名クラスが既に存在していないこと、および、スーパークラスがすべて定義済であることをチェックしなければならない。

これは、クラス階層が非循環でなくてはならないという Haskell の制約を守るための単純なやり方である。もちろん、実際にこのやり方がうまくいくためには、プログラム中のクラス宣言を位相的に整列しておかなくてはならないだろうから、循環があればこの段階で発見されるのだけど。

```
addClass :: Id -> [Id] -> EnvTransformer
addClass i is ce
  | defined (classes ce i) = fail "class already defined."
  | any (not.defined.classes ce) is = fail "superclass not defined."
  | otherwise = return (modify ce i (is, []))

addPreludeClasses :: EnvTransformer
addPreludeClasses  = addCoreClasses <:> addNumClasses

addCoreClasses ::   EnvTransformer
addCoreClasses  =   addClass "Eq" []
                <:> addClass "Ord" ["Eq"]
                <:> addClass "Show" []
                <:> addClass "Read" []
                <:> addClass "Bounded" []
                <:> addClass "Enum" []
                <:> addClass "Functor" []
                <:> addClass "Monad" []

addNumClasses  ::   EnvTransformer
addNumClasses   =   addClass "Num" ["Eq", "Show"]
```

```
        <:> addClass "Real" ["Num", "Ord"]
        <:> addClass "Fractional" ["Num"]
        <:> addClass "Integral" ["Real", "Enum"]
        <:> addClass "RealFrac" ["Real", "Fractional"]
        <:> addClass "Floating" ["Fractional"]
        <:> addClass "RealFloat" ["RealFrac", "Floating"]
```

新たにインスタンスを加えるときには、そのインスタンスに適用される型クラスが定義されていること、および、新規インスタンスが既存のものとオーバーラップしていないことをチェックしなければならない。

```
addInst                          :: [Pred] -> Pred -> EnvTransformer
addInst ps p@(IsIn i _) ce
 | not (defined (classes ce i)) = fail "no class for instance"
 | any (overlap p) qs           = fail "overlapping instance"
 | otherwise                    = return (modify ce i c)
   where its = insts ce i
         qs  = [ q | (_ :=> q) <- its ]
         c   = (super ce i, (ps:=>p) : its)
```

1 つのクラスに属する 2 つのインスタンスは、次を満たすときに「オーバーラップしている」という。

```
overlap       :: Pred -> Pred -> Bool
overlap p q    = defined (mguPred p q)
```

オーバーラップは、単に同じインスタンス宣言を二回行ってしまったケースも当然含む（例：Eq Int を二回宣言）のだが、他にも Eq [Int] と Eq [a] とか Eq (a, Bool) と Eq (Int, b) のように、より興味深いものもある。このような例は意味論の曖昧さを現しており、2 つのうちどちらを用いればいいのか、明らかな理由がない（ので、エラーにする）。

Haskell report は、上述の addClass や addInst がチェックしていないような制約を規定している。例えば：

- あるクラスと、そのスーパークラスの種は同じでなくてはならない
- インスタンスコンテクストにおける、あらゆる述語のパラメータは型変数であるそして、いずれも頭部に出現する
- インスタンスの頭部に出現する型は、異なる型変数を引数にとる型構築子を含む

これらをチェックするのは難しくはない上に煩雑なので、ここでは静的チェックがあらかじめなされる前提で、省略する。

7.3 Entailment (帰結)

この節では、あるクラスのインスタンスはどんな型なのかという問いにクラス環境を用いて答える方法について述べる。より一般的にいうと、帰結 (entailment) の扱いを考えていくことになる。述語 p と述語のリスト ps について、ps がすべて満たされるなら p が成り立つのかどうかを決めるのが目的である。p = IsIn i t かつ ps = [] の特別なケースは、t がクラス i のインスタンスであることに相当する。型修飾の理論においては、このような仮定は型判断 $ps \Vdash p$ によってとらえられるのだが、本稿ではそれを ential 関数をつかって、クラス環境への依存が明らかになるような形で示す。

はじめに、スーパークラスやインスタンスに関する情報が、それぞれどのように帰結の理由づけに用いられるのかを考えてみよう。たとえば、ある型が、あるクラス i のインスタンスであれば、その型は i のスーパークラスのインスタンスでもあるはずである。よって、スーパークラス情報のみを用いて、次のことがいえる：もし与えられた述語 p が成り立つなら、bySuper p のリストにあるすべての述語もなりたつ

```
bySuper :: ClassEnv -> Pred -> [Pred]
bySuper ce p@(IsIn i t)
 = p : concat [ bySuper ce (IsIn i' t) | i' <- super ce i ]
```

bySuper ce p のリストには重複はあるかもしれないが、常に有限である。このことは、クラス階層が非循環であることによって保証される。

次に、インスタンス情報の使われ方について。もちろん、p = IsIn i t という述語に対しては、insts ce i とすることよって、直接関係するインスタンスを環境から得ることが出来る。既にみたように、個々のインスタンス宣言は、ps :=> h の形の節に対応づけられる。頭部述語 h は、その宣言によって構築されるインスタンスの一般的な形を示しており、このインスタンスをある述語 p に適用可能かどうかは、matchPred を用いて調べることができる。適用可能なときには、この matching は代入 u を返すので、残る下位目標は map (apply u) ps の要素である。次の関数は、このアイディアを用いて、与えられた述語 p に対する下位目標のリストを決定する。

```
byInst                    :: ClassEnv -> Pred -> Maybe [Pred]
byInst ce p@(IsIn i t)     = msum [ tryInst it | it <- insts ce i ]
 where tryInst (ps :=> h) = do u <- matchPred h p
                               Just (map (apply u) ps)
```

msum は標準モナドライブラリにあるもので、Maybe 型のリストから、最初の定まった要素を返す。任意に与えられた述語 p に対し、適用可能なインスタンスは高々 1 つであるため、ここで msum が返す要素は、唯一の要素でもある。

bySuper と byInst は組み合わせて一般的な帰結関数に用いられる。

ここでやりたいことは、クラス環境 ce が与えられたときに、entail ce ps p が True となるための必要十分条件が「ps すべてが成り立つなら、いつでも p が成り立つ」となるようにすること。

```
entail        :: ClassEnv -> [Pred] -> Pred -> Bool
entail ce ps p = any (p `elem`) (map (bySuper ce) ps) ||
                 case byInst ce p of
                   Nothing -> False
                   Just qs -> all (entail ce ps) qs
```

まずスーパークラスのみを用いて ps から p が演繹できるかを調べ、失敗したら、マッチするインスタンスを探し述語のリスト qs を生成して、今度は qs が満たされるかどうかを調べている。

Haskell Report の規定（クラス階層が非循環で、型やインスタンス宣言は頭部よりも「小さい」）により、entail の計算は停止する。また、完全性についても、Haskell Report の制約によって保証される。

ここでは証明はしないが、このアルゴリズムの健全性、完全性、および、停止保証を信じることにしよう。

7.4 文脈の削減（Context Reduction）

クラス環境が、Haskell の型システムにおいて担うもう一つ重要な役割が、文脈の削減 (context reduction) である。文脈を削減することの基本的な目的は、述語のリストを等価な、しかし、何らかの意味でより単純なリストまで削減すること。

述語のリストを単純化するための 1 つの方法は、個々の述語に含まれる型コンポーネントを単純化することである。

(例) Eq [a], Eq (a, a) や Ea ([a], Int) はすべて Eq a に置き換えできる。

なぜなら、もとの述語が成立する $\leftrightarrow$ Eq a だから

この手の型コンポーネント単純化は、Eq (a, b) が (Eq a, Eq b) になるというように、述語の個数を増やすことがある。

このような単純化が、修飾型システムにおいてどの程度用いられているかは、実システムの実装や、多重定義の性能に影響する。

Haskell では、head-normal-form になるまで述語を単純化しなければならないように、言語に制約を加えている。この語は、λ 計算の head-normal-form との類似から来ている。より正確にいうと、Haskell では、クラス引数が v t1 ... tn （v は型変数、t1 ... tn (n>=0) は型）の形であることが要求される。述語がこの制約を満たしているかどうかは、次の関数でもとまる：

```
inHnf       :: Pred -> Bool
inHnf (IsIn c t) = hnf t
 where hnf (TVar v)  = True
       hnf (TCon tc) = False
       hnf (TAp t _) = hnf t
```

このパタンに合わない述語は byInst でばらす必要がある。ときには、述語はすっかり削除されてしまうかもしれない。言い換えると、byInst の失敗は述語に関する条件が充足不可であることを示すので、診断メッセー

ジをだすきっかけとなる。

```
toHnfs      :: Monad m => ClassEnv -> [Pred] -> m [Pred]
toHnfs ce ps = do pss <- mapM (toHnf ce) ps
                  return (concat pss)

toHnf                  :: Monad m => ClassEnv -> Pred -> m [Pred]
toHnf ce p | inHnf p   = return [p]
           | otherwise = case byInst ce p of
                           Nothing -> fail "context reduction"
                           Just ps -> toHnfs ce ps
```

述語のリストを単純化するためのもう一つの方法は、リストに含まれる要素の数を減らすことである。これを達成できそうな方法にはいくつかある：

- 重複をとりのぞく（eg. [Eq a, Eq a] を Eq a に）
- 成立することがわかっている述語をのぞく（eg. Num Int を削る）
- 親クラスの情報をつかう（eg. [Eq a, Ord a] を Ord a に)

いずれにおいても、削減の前後で（述語のリストを充足する場合が同じという意味で）等価である。

我々が用いるアルゴリズムは、述語のリスト (p:ps) に含まれる述語 p は、ps から p が帰結されるときには取り除けるという洞察に基づく。特別な場合として、重複した述語もこれで取り除かれる。

```
simplify   :: ClassEnv -> [Pred] -> [Pred]
simplify ce = loop []
 where loop rs []                             = rs
       loop rs (p:ps) | entail ce (rs++ps) p = loop rs ps
                      | otherwise            = loop (p:rs) ps
```

以上を用いて Haskell で用いられている文脈削減を、toHnfs, simplify の組み合わせで書ける。

```
reduce      :: Monad m => ClassEnv -> [Pred] -> m [Pred]
reduce ce ps = do qs <- toHnfs ce ps
                  return (simplify ce qs)
```

技術的に細かい話になるが、上の redude は冗長である。qs はすでに head-normal-form であることが保証されており、Haskell の制約下では instance 宣言にマッチしない。なので、simplify は、スーパークラスのみを参照するような entail の亜種をつかってもよい。

```
scEntail        :: ClassEnv -> [Pred] -> Pred -> Bool
scEntail ce ps p = any (p `elem`) (map (bySuper ce) ps)
```

8. 型スキーム (Type Scheme)

型スキームは多相型を表すのに用いられ、種のリストと修飾された型で表現される。

```
data Scheme = Forall [Kind] (Qual Type)
              deriving Eq
```

Haskell 文法には Forall に直接対応するものはないが、自由型変数を束縛するような、暗黙の量化子が挿入されていると考えることができる[*7]。

型スキーム、Forall ks qt において、修飾された型 qt に出現する TGen n 型は、総称型（または全称型）変数を表しており、その種は ks !! n で与えられる。また、TGen 型が許されるのはここだけである。この制約を（型や型スキームの表現を注意深くえらぶなどで）うまく静的に表せたらいいなと思っていたのですが、これを強制するうまいやり方がみつからず、で、よく考えた結果、今の表現に落ち着いた。こうすることで、型修飾や代入を単純に実装できるからである。

たとえば、型への代入は TGen 値を無視する。これにより、以下の定義において、変数捕獲の問題はおきない。

```
instance Types Scheme where
  apply s (Forall ks qt) = Forall ks (apply s qt)
  tv (Forall ks qt)      = tv qt
```

型スキームは、修飾型 qt を、型変数のリスト vs について量化することで構築される。

```
quantify      :: [Tyvar] -> Qual Type -> Scheme
quantify vs qt = Forall ks (apply s qt)
 where vs' = [ v | v <- tv qt, v `elem` vs ]
       ks  = map kind vs'
       s   = zip vs' (map TGen [0..])
```

[*7] Haskell の型シグネチャは暗黙のうちに量化されている。たとえば、g :: a -> a は g :: ∀ a . a -> a の意味。GHC では、-XExplicitForAll を用いると g :: forall a . a -> a と書ける

ks における種のならび順は、vs ではなく tv qt における型変数の出現順であることに注意。なので、例えば、型スキーム中の最も左の量化変数はいつも TGen 0 になる。型スキームをこのやり方でつくる限り、型スキームはユニークな正準形になる。このことは、重要である。なぜなら、これによって、単純な比較で型スキーム同士がおなじかどうかを判定できるからである（より複雑な α 等価性などを実装しなくていい)。

量化変数なしで型を型スキームに変換したいこともある：

```
toScheme      :: Type -> Scheme
toScheme t     = Forall [] ([] :=> t)
```

型スキームの説明を完成させるためには、型スキームにおける量化変数を実体化できるようにならねばならない。だが、型推論のためだけであれば、単に fresh な型変数で型変数を実体化する、特殊ケースだけで足りる。そのため、実体化については、10 節で後述する（型推論モナドで fresh な型変数が導入されてから)。

9 仮定 (Assumptions)

変数の型についての仮定は Assump データ型で表現される。各仮定は、変数名と型スキームのペアである。

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

ここでも Types クラスのインスタンスにして代入を適用できるようにする：

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

Types はリストに対しても定義してあるので、apply や tv は Assump のリストに対しても使えることになる。Assump のリストは、型推論中にプログラム変数を記録するのに用いられる。

仮定の集合から、特定の変数の型を探すための関数も用意しておく：

```
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
```

この実装は、未束縛のケースを許しているが、実際には、未束縛の変数は型推論よりも早い段階で検出されるだろう。

10 型推論モナド

今日では、モナドを用いてある種の「配管」を隠し、プログラム設計のもっと重要な面に注意を向けるのは、とても普通のことになっている。型推論でも、その用途で状態モナドを使う。

```
newtype TI a = TI (Subst -> Int -> (Subst, Int, a))

instance Monad TI where
  return x   = TI (\s n -> (s,n,x))
  TI f >>= g = TI (\s n -> case f s n of
                             (s',m,x) -> let TI gx = g x
                                         in  gx s' m)

runTI       :: TI a -> a
runTI (TI f) = x where (s,n,x) = f nullSubst 0
```

現在の代入を返す：

```
getSubst   :: TI Subst
getSubst    = TI (\s n -> (s,n,s))
```

引数の最汎単一化子で、現在の代入を拡張する：

```
unify      :: Type -> Type -> TI ()
unify t1 t2 = do s <- getSubst
                 u <- mgu (apply s t1) (apply s t2)
                 extSubst u

extSubst   :: Subst -> TI ()
extSubst s' = TI (\s n -> (s'@@s, n, ()))
```

全体的にみて、現在の代入を TI モナドに隠すことにしたことは、型推論の表現を明瞭にするのに役立っている。とくに apply を拡張が計算される度に何度も用いなくてもいい点が大きい。

状態のうち整数要素を用いる唯一のプリミティブが以下のもの。これは enumID と組み合わせて指定された種の新しい型変数を生成する。

```
newTVar    :: Kind -> TI Type
newTVar k   = TI (\s n -> let v = Tyvar (enumId n) k
                          in  (s, n+1, TVar v))
```

newTVar が用いられる場所のひとつは、型スキームを、適切な種の新しい型変数でもって実体化するとき：

```
freshInst               :: Scheme -> TI (Qual Type)
freshInst (Forall ks qt) = do ts <- mapM newTVar ks
                              return (inst ts qt)

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 ts t         = t
instance Instantiate a => Instantiate [a] where
  inst ts = map (inst ts)
instance Instantiate t => Instantiate (Qual t) where
  inst ts (ps :=> t) = inst ts ps :=> inst ts t
instance Instantiate Pred where
  inst ts (IsIn c t) = IsIn c (inst ts t)
```

11. 型推論

ほとんどの型付け規則 (typing rules) は次の型の関数として表される。

```
type Infer e t = ClassEnv -> [Assump] -> e -> TI ([Pred], t)
```

理論よりなことを言うと、この規則が型判断 $\Gamma\ ;\ P \mid A \vdash e : t$ の形（ここで Γ はクラス環境、P は述語の集合、A は仮定の集合、e は式、そして t はその型）をしていることは驚くにはあたらない。このような型判断は 5 つ組と考えることができ、型付け規則も 5 箇所関係に対応づけられる。Infer e t にも同じ関係がみられるのだが、関数にしたことで、入出力の区別が明確になっている。

11.1 リテラル

```
data Literal = LitInt  Integer
             | LitChar Char
             | LitRat  Rational
             | LitStr  String

tiLit            :: Literal -> TI ([Pred],Type)
tiLit (LitChar _) = return ([], tChar)
tiLit (LitInt _)  = do v <- newTVar Star
                       return ([IsIn "Num" v], v)
tiLit (LitStr _)  = return ([], tString)
tiLit (LitRat _)  = do v <- newTVar Star
                       return ([IsIn "Fractional" v], v)
```

11.2 パターン

パターンは、ラムダ抽象、関数とパターンの束縛、リスト内包、do 記法、および case 式においてデータ値を検査したり分解したりするのに用いられる。

```
data Pat        = PVar Id
                | PWildcard
                | PAs  Id Pat
                | PLit Literal
                | PNpk Id Integer
                | PCon Assump [Pat]
```

コンストラクタを表すデータには Assump を用いた。型推論に本当に必要なのは型であって名前はいらないのだが。

ほとんどの Haskell のパターンは Pat で表現できるが、labeled field をつかったパターンを扱うには拡張がいる。これは難しくはないのだが、いくらか煩雑になるので、ここでは省く。

パターンの型推論には 2 つのゴールがある。

- 束縛される各変数の型を計算する
- パターン全体にマッチするような値の型を決める

```
tiPat :: Pat -> TI ([Pred], [Assump], Type)
```

Assump を渡さなくても良いことに注意。パターン中の変数は、外にある変数を隠す。
　変数パターン PVar i に対しては、単に新しい仮定を返す：

```
tiPat (PVar i) = do v <- newTVar Star
                    return ([], [i :>: toScheme v], v)
```

Haskell はひとつのパターン中に、変数を何度も用いることを許していないので、これが、このパターンにおける i の唯一の出現であると考えてよい。
　ワイルドカードに対するものは、これと似ているが、新しい仮定をつくらない。

```
tiPat PWildcard   = do v <- newTVar Star
                       return ([], [], v)
```

As パターン PAs i pat に対しては、pat の仮定と型を計算して、さらに、それを i に束縛するような仮定をつけくわえる。

```
tiPat (PAs i pat) = do (ps, as, t) <- tiPat pat
                       return (ps, (i:>:toScheme t):as, t)
```

リテラルパターン

```
tiPat (PLit l) = do (ps, t) <- tiLit l
                    return (ps, [], t)
```

コンストラクタパターンのときには少し複雑。

```
tiPat (PCon (i:>:sc) pats) = do (ps,as,ts) <- tiPats pats
                                t'         <- newTVar Star
                                (qs :=> t) <- freshInst sc
                                unify t (foldr fn t' ts)
                                return (ps++qs, as, t')
```

1. 後述する tiPats で、各部分パターンの型と、仮定リスト、述語リストを求める。

2. 新しい型変数 t' を、この時点では未知であるパターン全体の型をつかまえるためにつくる。
3. コンストラクタの型は、以上より、t1 -> (t2 -> ... -> (tn -> t'))) と予想される。これは foldr fn t' [t1, t2, ..., tn] で求まる。
4. 予想された型のチェックは、sc を実体化されたものと単一化することで可能。

```
tiPats     :: [Pat] -> TI ([Pred], [Assump], [Type])
tiPats pats = do psasts <- mapM tiPat pats
                 let ps = concat [ ps' | (ps',_,_) <- psasts ]
                     as = concat [ as' | (_,as',_) <- psasts ]
                     ts = [ t | (_,_,t) <- psasts ]
                 return (ps, as, ts)
```

tiPats は tiPat の中で使われている以外のも、関数の左辺にパターンのリストが出現したときなどにも使う。

11.3 式

```
data Expr = Var   Id
          | Lit   Literal
          | Const Assump
          | Ap    Expr Expr
          | Let   BindGroup Expr
```

Haskell は、これらよりずっと豊かな式の文法をもつ（λ抽象、case 式、条件式、リスト内包や do 記法）が、これらはすべて単純な翻訳で Expr にできる。

（例） \x -> e は let f x = e in f (f は新しい変数)

式の型推論はいたって素直に書ける：

```
tiExpr                        :: Infer Expr Type
tiExpr ce as (Var i)           = do sc         <- find i as
                                    (ps :=> t) <- freshInst sc
                                    return (ps, t)
tiExpr ce as (Const (i:>:sc)) = do (ps :=> t) <- freshInst sc
                                    return (ps, t)
tiExpr ce as (Lit l)           = do (ps,t) <- tiLit l
                                    return (ps, t)
tiExpr ce as (Ap e f)          = do (ps,te) <- tiExpr ce as e
                                    (qs,tf) <- tiExpr ce as f
                                    t       <- newTVar Star
```

```
                                    unify (tf `fn` t) te
                                    return (ps++qs, t)
tiExpr ce as (Let bg e)       = do (ps, as') <- tiBindGroup ce as bg
                                    (qs, t)   <- tiExpr ce (as' ++ as) e
                                    return (ps ++ qs, t)
```

11.4 Alternatives

後続の節では、関数束縛を表すのに Alt を用いる。

```
type Alt = ([Pat], Expr)
```

Alt は、関数宣言の左辺と右辺を規定する。より完全な Expr 文法では、Alt はλ式や Case 式にも用いられるだろう。

```
tiAlt                :: Infer Alt Type
tiAlt ce as (pats, e) = do (ps, as', ts) <- tiPats pats
                           (qs,t)  <- tiExpr ce (as'++as) e
                           return (ps++qs, foldr fn t ts)
```

実際には、Alt のリスト alts に対して型検査を行い、その結果が既知の型 t に合うかを調べたいことがよくある。この過程は次のようにまとめることができる：

```
tiAlts             :: ClassEnv -> [Assump] -> [Alt] -> Type -> TI [Pred]
tiAlts ce as alts t = do psts <- mapM (tiAlt ce as) alts
                         mapM (unify t) (map snd psts)
                         return (concat (map fst psts))
```

ここではやらなかったが、最後ではなく各 Alt のチェックの中に t との検査をいれてしまうこともできる。Hugs のように二階多相をやるには、そうするのが大切になってくるし、拡張しない場合でも、そうした方がエラーメッセージをまともにできる可能性がある。

もちろん tiAlts を、いちから型を推論するためにも使える。そのときには、fresh な型変数を用意して t として渡す。そして返ってきたら、現在の代入における v を調べればよい。

11.5 型から型スキームへ

これまでは、型推論を通じて述語が積み重ねられていく様子を述べてきたが、この先は、述語の列が、推論される型の構築に用いられる様子を述べる。

この過程は、しばしば generalization と呼ばれる。なぜなら、常に最も一般的な型を求めようとするからである。標準的な Hindley-Milner システムでは、普通は、関連する型変数で仮定の集合に現れないものすべてを量化することで、最も一般的な型を計算できる。本節では、この過程も改めて Haskell の述語向けにするやり方を示す。

基本的な問題を理解するために、型検査器を関数の本体 h に対して走らせ、述語のリスト ps と型 t を得ることを考える。

この時点では、最も一般的な結果として、h の型としては修飾型 qt = (ps :=> t) を形成し、そして、qt 中の仮定にない全ての変数を量化することで推論する。

これは、修飾型の理論によって許されるのだが、実際にやるには最善とはいえない。たとえば :

- ps しばしば単純化できる（7.4 節で述べた文脈削減で)。また、これにより、head-normal-form が要求されるので、Haskell の言語制約も守れる。
- ps には、すでに定まった (fixed) 変数（つまり仮定に含まれる変数）を含むものもあるかもしれない。それによる、推論された型での制約は使われないので、そういう述語は外側の束縛まで遅らされるべきである。
- ps におけるいくつかの述語は、曖昧であるという結果になり得る。これは、エラーを避けるために dafaulting (11.5.1 で述べる) を要する。

これらの問題に対処するため、split と呼ばれる関数を使う。関数 h に対する推論された型 t、述語のリスト ps が手に入ったとき、split を用いて ps を書き換え、(ds, rs) の組に分けることができる (ds: deferred rs: retained)。

rs は推論型 (rs :=> t) を形作るのに使われ、ds は外側のスコープを制約するために渡される。

```
split :: Monad m => ClassEnv -> [Tyvar] -> [Tyvar] -> [Pred]
                      -> m ([Pred], [Pred])
split ce fs gs ps = do ps' <- reduce ce ps
                       let (ds, rs) = partition (all (`elem` fs) . tv) ps'
                       rs' <- defaultedPreds ce (fs++gs) rs
                       return (ds, rs \\ rs')
```

述語のリスト ps に加えて、split は 2 つの型変数リストを引数としてとる。最初の fs は fixed な変数のリストである。これは、仮定の中に自由に出現する変数のリストである。 2 つめの gs は、我々が量化しようとしている変数の集合を指定する。上の例では単に (tv t \\fs) になるだろう。ps は、fs にも gs にも含まれない変数を含んでいるかもしれない。11.5.1 節でみるが、これは曖昧さを示している。split は 3 つの段階にわかれているが、これは前に上げた 3 つの点に対応している。

1. reduce を用いた文脈の削減
2. partition 関数を用いて、fs にある fixed 変数を含むものだけが ps' からわけられ ds に
3. rs の各述語は Haskell の defaulting によって削除されるかどうかが決められる。取り除くべきものが rs' にリストされ、最後に rs \\rs' によって引かれる。

11.5.1 曖昧性とデフォルト

Haskell の用語では、型スキーム ps :=> t は、t に出現しない全称変数が ps にあるとき曖昧であるという。

この条件は重要である。なぜなら、理論的な研究によると、一般に項の最も一般的な型が曖昧でないときに限り、well-formed なセマンティクス (semantics) が定義されるため。

なので、曖昧な型をもつ式は、Haskell においては型付け異常として静的なエラーとなる。

（例）stringInc x = show (read x + 1)

stringInc "1.5" は、x が浮動小数点数とすれば "2.5" になるが、整数だとすると構文エラーになる。どちらを選ぶかでセマンティクスが変わる。

stringInc x :: (Read a, Num a) => String -> String

※ Show a がないのは、Show が Num のスーパークラスだから。

次のようにすれば直せる：

stringInc x = show (read x + 1 :: Int)

経験上、この手の曖昧性はまれにしか実際のコードではみかけないし、曖昧性が検出されたときには、そのことが、プログラムの本当の問題をプログラマに気付かせる場合も多い。

しかし、Haskell の設計者は、数値型と、おそらくは多相な数値リテラルによって、あまりに多くの曖昧性が生じてプログラマをうんざりさせるのではないかと思ったのだろう。そこで、実用主義的な妥協策として、Haskell の default が導入された。

これは、便利だが危険なものである。default は、もしそれがなければ曖昧になっていた型を自動的に選んでくれるのだが、それにより明示的でない方法でセマンティクスが選ばれてしまうからだ。

だから、default の使用は、ごく限定的、かつ、よく理解された場合にとどめるべきである。

本節の残りの部分で、Haskell プログラムの曖昧性がどのように検出されるか、そして、適切なときには、それが defaulting によって 取り除かれる様子を詳しくみていく。

第一歩は、曖昧さの元を特定することである。

修飾型の述語 ps と、既知のすべての変数のリスト vs (fixed, generic 両方）を想定したとき、ps にあって vs にない型変数（つまり tv ps \\vs) があるときに、曖昧性が生じる。

defaulting の目的は、各々の曖昧な変数 v を単一型 t に結び付けること。t は、v に t を代入したときに ps がすべて充足されるように選ばれなければならない。

以下では、曖昧な変数を、各々 default を選ぶ際に満たすべき述語のリストとペアにして求める方法を示す。

```
type Ambiguity       = (Tyvar, [Pred])

ambiguities         :: ClassEnv -> [Tyvar] -> [Pred] -> [Ambiguity]
ambiguities ce vs ps = [ (v, filter (elem v . tv) ps) | v <- tv ps \\ vs ]
```

Haskell Report (§ 4.3.4) にあるように、(v, qs) ペアの集合が与えられたとき、以下の条件が満たされる場合に限って defaulting が許可される。

- qs に含まれるすべての述語が IsIn c (TVar v) の形である（c は何らかのクラス）
- qs に関わるクラスのうち、少なくとも一つは標準の数値型（または IsString)
- qs に関わるすべてのクラスは Prelude または標準ライブラリで定義される標準クラスである。

包含する (enclosing) モジュールのデフォルト型リストの中に、qs で言及されたすべてのクラスのインスタンスであるような型が 1 つ以上あったときに、リストの中で最初のものがデフォルトとして選ばれる。デフォルト型のリストは、7.2 節で述べた通り、defaults 関数によってクラス環境から取り出せる。

これらの条件は、以下の定義によって、もっと簡潔に検査される。これを、我々は、特定の曖昧性を解決するような候補を計算するために用いる。

```
numClasses :: [Id]
numClasses  = ["Num", "Integral", "Floating", "Fractional",
               "Real", "RealFloat", "RealFrac"]

stdClasses :: [Id]
stdClasses  = ["Eq", "Ord", "Show", "Read", "Bounded", "Enum", "Ix",
               "Functor", "Monad", "MonadPlus"] ++ numClasses

candidates           :: ClassEnv -> Ambiguity -> [Type]
candidates ce (v, qs) = [ t' | let is = [ i | IsIn i t <- qs ]
                                   ts = [ t | IsIn i t <- qs ],
                               all ((TVar v)==) ts,
                               any (`elem` numClasses) is,
                               all (`elem` stdClasses) is,
                               t' <- defaults ce,
                               all (entail ce []) [ IsIn i t' | i <- is ] ]
```

candidates が空のリストを返した場合、デフォルト型はその変数には適用できず、したがって、曖昧性は回避できない。一方、空でないリスト ts が返るときには、head ts を v に代入することが可能で、ps から述語の集合 qs を取り除ける。

デフォルト代入のための計算、および、それによって取り除かれる述語リストのための計算は、次に示すような、高階関数を用いた単純なパターンで表せる：

```
withDefaults :: Monad m => ([Ambiguity] -> [Type] -> a)
                  -> ClassEnv -> [Tyvar] -> [Pred] -> m a
withDefaults f ce vs ps
```

```
    | any null tss  = fail "cannot resolve ambiguity"
    | otherwise     = return (f vps (map head tss))
      where vps = ambiguities ce vs ps
            tss = map (candidates ce) vps
```

withDefaults 関数が適切なデフォルトの抜きだしや、曖昧性の削除が可能かどうかの検査をするので、defaulting が成功したときには、取り除かれる述語のリストは、各 Ambiguity の述語を連結することで得られる。

```
defaultedPreds :: Monad m => ClassEnv -> [Tyvar] -> [Pred] -> m [Pred]
defaultedPreds  = withDefaults (\vps ts -> concat (map snd vps))
```

同様に、デフォルト代入は、変数のリストとデフォルトリストを zip すれば得られる。

```
defaultSubst   :: Monad m => ClassEnv -> [Tyvar] -> [Pred] -> m Subst
defaultSubst    = withDefaults (\vps ts -> zip (map fst vps) ts)
```

このデフォルト代入がなぜ有用なのか疑問に感じる人もいるかもしれない。もし仮定や推論された型のどこにも曖昧な変数が出現しなければ、この代入には効果がない。

実際、defaultSubst が必要になるのは、1 つのモジュール全体の型推論が完了したトップレベルにおいてのみである。

このケースでは、Haskell の悪名高い ”monomorphizm 制約” によって、いくつかの型変数の一般化が妨げられる可能性がある。しかし Haskell は未束縛の型変数をトップレベル関数の型については許していない。その代わりに、残った型変数は、推論された型の中にあるものでも曖昧とみなされる。そこで、defaultSubst が必要とされるのだ。

11.6 バインディング・グループ (Binding Groups)

残す技術的挑戦は、バインディング・グループに対する型検査について述べることである。この領域は、型推論の理論的な扱いでは、ほとんどの場合に省略され、しばしば、基本的な考え方を拡張する単純な演習問題とされる。が、少なくとも、Haskell に関する限り、それは全く正しくない！

多重定義、多相再帰および明示型と暗黙型の混在、これらの相互作用は、型推論における最も複雑で微妙な要素である。我々はまず、型を明示した束縛と、明示しない束縛を別々に述べ、そののち、それらをいかに統合するかについて述べる。

11.6.1 明示型付き束縛

最も単純なケースは、明に型付けられた束縛に対するもので、各々は定義される関数の名前、宣言された型スキーム、および、定義ににおける alternatives のリストで記述される。

```
type Expl = (Id, Scheme, [Alt])
```

Haskell は、ある 1 つの識別子の定義における各々の Alts が同じ個数の左辺引数をもつことを要求するが、ここではこれを強制していない。

型が明示されている束縛の型推論は、かなり簡単である。我々は、単に、宣言された型が有効であることを検査すればよい。第一の原則から順を追って型を推論する必要はない。

多相再帰をサポートするために、以下の関数を呼ぶ前には、i の宣言された型は仮定のリストに含まれていなければならない。

```
tiExpl :: ClassEnv -> [Assump] -> Expl -> TI [Pred]
tiExpl ce as (i, sc, alts)
        = do (qs :=> t) <- freshInst sc
             ps         <- tiAlts ce as alts t
             s          <- getSubst
             let qs'     = apply s qs
                 t'      = apply s t
                 fs      = tv (apply s as)
                 gs      = tv t' \\ fs
                 sc'     = quantify gs (qs':=>t')
                 ps'     = filter (not . entail ce qs') (apply s ps)
             (ds,rs)    <- split ce fs gs ps'
             if sc /= sc' then
                 fail "signature too general"
               else if not (null rs) then
                 fail "context too weak"
               else
                 return ds
```

このコードでは、まず宣言された型スキーム sc が実体化され、その結果の型 t で alternatives を検査する。すべての alternative が処理されたあと、推論された型は qs' :=> t' となる。型宣言が正しければ、これは、全称変数のリネーミングを除けば元の型 qs :=> t と同じになる。もし、型シグネチャが一般的過ぎたときには、計算された sc' は sc よりも特殊になるので、そのときにはエラーを報告する。

その間、alternative のリストを検査している間に生成された述語は排出しなければならない。文脈 qs' によって帰結される述語は、さっさと削除できる。残った述語は ps' にあつめられ、適切に用意された fixed および generic 変数とともに split に渡される。文脈削減後もなお残った述語があった場合には、文脈が弱すぎる旨のエラーとなる。

11.6.2 暗黙型付き束縛

明に型付けされていない束縛の扱いに関しては、2つの複雑な問題がある。

1つめは、相互再帰束縛は、各々を別々に推論するのではなく、ひとまとまりに扱わなければならないこと。2つめは、Haskell の monomorphism restriction で、これは、あるケースでの多重定義を制限する。

1つの、明に型付けされていない束縛は、変数名と alternative のリストのペアで表される。

```
type Impl   = (Id, [Alt])
```

Monomorphism restriction は、明に型付けされていない束縛において、1つ以上のエントリが単純であるとき、すなわち、左辺パターンのない alternative をもつときに発動する。

```
restricted   :: [Impl] -> Bool
restricted bs = any simple bs
 where simple (i,alts) = any (null . fst) alts
```

相互再帰の、明に型付けされていない束縛に対する型推論は次の通りである。

```
tiImpls         :: Infer [Impl] [Assump]
tiImpls ce as bs = do ts <- mapM (\_ -> newTVar Star) bs
                      let is    = map fst bs
                          scs   = map toScheme ts
                          as'   = zipWith (:>:) is scs ++ as
                          altss = map snd bs
                      pss <- sequence (zipWith (tiAlts ce as') altss ts)
                      s   <- getSubst
                      let ps'     = apply s (concat pss)
                          ts'     = apply s ts
                          fs      = tv (apply s as)
                          vss     = map tv ts'
                          gs      = foldr1 union vss \\ fs
                      (ds,rs) <- split ce fs (foldr1 intersect vss) ps'
                      if restricted bs then
                          let gs'  = gs \\ tv rs
                              scs' = map (quantify gs' . ([]:=>)) ts'
                          in return (ds++rs, zipWith (:>:) is scs')
                        else
```

```
        let scs' = map (quantify gs . (rs:=>)) ts'
        in return (ds, zipWith (:>:) is scs')
```

この仮定のはじめの部分では、bs で定義された各識別子を新しい型変数に束縛するような仮定で as を拡張する。そして、それを各束縛の alts を型検査するのに用いる。これは、定義しようとしている束縛において、各型変数が同じ型に対して用いられることを確かめるために必要。

次に、split を使って、推論された述語 ps' を deferred な述語 ds と残る述語 rs にわける。gs は、推論型 ts' に出現する全称型変数であって、fixed 変数 fs にはないもののリスト。split には、これとは異なるリストが渡されることに注意。すべての推論型に含まれる変数だけが渡される。これは、これらの型だけが最終的に同じ述語集合によって修飾されるはずであって、結果の型が曖昧であることは望まれていないためである。

最後の段階では、まず monomorphism restriction を適用すべきかどうかが検査される。そして、引き続き定義された各値の主型を含む仮定の計算を行う。unrestricted binding については、単純に、残った述語 rs を修飾し、全称型変数 gs を量化すればよい。バインディング・グループが restricted であるときには、ds 同様に rs も遅延させ、rs に出現しない gs の要素のみを量化する。

11.6.3 結合されたバインディング・グループ

トップレベルのプログラムであれ、ローカル定義であれ、複数の束縛を、相互再帰するような最小のグループに分割して、かつ、後続のグループには依存しないように並び替えるような依存性解析を、Haskell は要求する。

これは、最も一般的な型が可能となるために必要である。たとえば、次のコード片において、

```
foldr f a (x:xs) = f x (foldr f a xs)
foldr f a []     = a
and xs = foldr (&&) True xs
```

もし、これらの定義をすべて 1 つのバインディング・グループに入れてしまったら、foldr の最も一般的な型を得ることができない。このバインディング・グループにおいて変数のすべての出現は同じ型をもたなければならないため、foldr の型は次のように導かれてしまうだろう：

```
(Bool -> Bool -> Bool) -> Bool -> [Bool] -> Bool
```

この問題を避けるには、foldr の定義が、とにかく (&&) に依存しないことだけ気をつければよい。結果、2 つの関数を別のバインディング・グループに配置し、先に foldr の最も一般的な型を求め、次に and の型を正しくする。

明示的な型付けもあるので、この依存性解析には、すこし修正がいる。

例えば、次の束縛の組を考える：

```
f :: Eq a => a -> Bool
f x = (x == x) || g True
g y = (y <= y) || f True
```

これは相互に再帰してはいるが、f と g を同時に型推論する必要はない。そうではなく、f の型宣言を用いて、次の型を推論できる。

```
g :: Ord a => a -> Bool
```

そして、これを f 本体のチェックに用い、宣言された型が正しいことを確認する。

以上の観察をもとに、Haskell のバインディング・グループを次のデータ型で表すことにする。

```
type BindGroup  = ([Expl], [[Impl]])
```

各ペアの第 1 要素は、グループ中の明に型付けされた束縛のリストである。第 2 要素は、明に型付けされない束縛のリストを、さらに小さいリストに分解し依存順に並べたものである。

1 つのバインディング・グループ (es, [is1, ..., isn]) において、各 isi に含まれる明に型付けされていない束縛は、es, is1, ..., isi のみに依存する。j>i であるような isj には依存しない。

（es 内の束縛は、グループ内のどの束縛に依存してもよいが、少なくとも isn に含まれるものには依存するはずである。そうでないなら、グループはおそらく最小になっていない。また、es が空なら n=1 になる）

この表現を選ぶにあたって、我々は、型検査に先立って依存解析が行われ、BindGroup 値が適切に作られていることを前提とする（参考：強連結成分分解）。

特に、明示型付け束縛と、非明示のものを分離することで、推論される型をより多相的にできる可能性がある。

さらに、グループが restricted なら、同じバインディング・グループ内の明示型付け束縛のいずれも、その型に述語を含まないことを保証しなければならない。後知恵だが、Haskell のこの制約は不要なので、とってしまいたいのだが。

pat = expr の形の束縛を直接扱う必要はない。単純な変換により、本稿で述べた枠組みにあてはめられるから。たとえば、

```
(x, y) = expr
```

は、次のように書き直せる。

```
nv = expr
x = fst nv
```

```
y = snd nv
```
(nv は新しい変数)

この変換を用いる限り、パターンに対する monomorphism restriction も結果的に正しく判定される。

ついに、バインディング・グループの型推論を示す用意ができた：

```
tiBindGroup :: Infer BindGroup [Assump]
tiBindGroup ce as (es,iss) =
  do let as' = [ v:>:sc | (v,sc,alts) <- es ]
     (ps, as'') <- tiSeq tiImpls ce (as'++as) iss
     qss        <- mapM (tiExpl ce (as''++as'++as)) es
     return (ps++concat qss, as''++as')
```

まず、明示的な型付けに対応する仮定 (as') を作り、それを用いて、非明示型束縛の各グループの検査をする。そのとき仮定は、後の段階のために拡張しつつ。最後に、明示型付け束縛にもどって各宣言型が受理可能かどうかを検証する。

非明示型のバインディング・グループリストを扱うために、次に示すユーティリティ関数を用いる。これは、バインディング・グループのリストを型検査しつつ、仮定を累積していく。

```
tiSeq                  :: Infer bg [Assump] -> Infer [bg] [Assump]
tiSeq ti ce as []       = return ([],[])
tiSeq ti ce as (bs:bss) = do (ps,as')  <- ti ce as bs
                             (qs,as'') <- tiSeq ti ce (as'++as) bss
                             return (ps++qs, as''++as')
```

11.6.4 トップレベルのバインディング・グループ

トップレベルにおいて、Haskell のプログラムはバインディンググループのリストとみなせる。

type Program = [BindGroup]

クラス宣言やインスタンス宣言におけるメンバ関数でさえ、この表現に含めることができる。それらは、トップレベルの明示型束縛に翻訳できる。

プログラムの型推論は、組み込み型についての仮定を引数にとり、すべての変数についての仮定を返す。

```
tiProgram :: ClassEnv -> [Assump] -> Program -> [Assump]
tiProgram ce as bgs = runTI $
                      do (ps, as') <- tiSeq tiBindGroup ce as bgs
                         s         <- getSubst
```

```
rs        <- reduce ce (apply s ps)
s'        <- defaultSubst ce [] rs
return (apply (s'@@s) as')
```

これにて、Haskell の型システムは述べられました（おわり）。