



Haskell,  
Type Function 派

shelarcy



# Type Function (型関数)

## って何？

- 「型」を取って「型」を返す関数
- GHC には専用の機能が用意されている
  - 型族 (Type Families)
  - 関連型 (Associated Types)

詳しい情報は

Oleg Kiselyov, Simon Peyton Jones  
and Chung-chieh Shan (單中杰):

“Fun with type functions”

(現在執筆中)

を読んでください、以上

で、話を終わらせる訳  
にはいいかないので、  
説明

# Haskell の関数

- 関数定義

$\text{add } n \ m = n + m$

$\text{factorial } 0 = 1$

$\text{factorial } n = n * \text{factorial } (n-1)$

- 型

`*ValueLevel> :type add`

`add :: (Num a) => a -> a -> a`

`*ValueLevel> :type factorial`

`factorial :: (Num t) => t -> t`

# 型族

- 関数定義

```
data Z
```

```
data S a
```

```
type family Add n m
```

```
type instance Add Z x = x
```

```
type instance Add (S x) y = S (Add x y)
```

- 型 (種)

```
*TypeLevel> :kind Add
```

```
Add :: * -> * -> *
```

# 種 (kind)

- 「\*」
- 「型」の「型」

```
*TypeLevel> :kind Int
```

```
Int :: *
```

```
*TypeLevel> :kind Z
```

```
Z :: *
```

```
*TypeLevel> :kind S
```

```
S :: * -> *
```

```
*TypeLevel> :kind (S Z)
```

```
(S Z) :: *
```

# 型族の使用例

- 長さ付きリスト (Sized List)

```
data Vec e len where -- bounded vector
```

```
  Nil :: Vec e Z
```

```
  Cons :: e -> Vec e len -> Vec e (S len)
```

```
vappend :: Vec e n -> Vec e m -> Vec e (Add n m)
```

```
vappend Nil l = l
```

```
vappend (Cons x xs) ys = Cons x (vappend xs ys)
```

# 長さ付きのデータ型？

- データ型の大きさを静的型として格納
  - 範囲外アクセスの防止
  - ただし、何らかの方法で「値（式）」から「型」を求める必要あり
    - e.g. 型クラス、GADT、  
Template Haskell、 etc ...

# 長さ付きリスト

- 長さを見る関数

```
vhead :: Vec e (S len) -> e
vhead (Cons x _) = x
```

- 使用例

```
*TypeLevel> vhead (Cons I Nil)
```

```
I
```

```
*TypeLevel> vhead Nil
```

```
<interactive>:1:6:
```

```
Couldn't match expected type `S len' against inferred type `Z'
```

```
Expected type:Vec e (S len)
```

```
Inferred type:Vec e Z
```

```
In the first argument of `vhead', namely `Nil'
```

```
In the expression: vhead Nil
```

# 長さ付きリスト

- 使用例

```
*TypeLevel> vhead $ vappend Nil Nil
```

```
<interactive>:1:8:
```

```
Couldn't match expected type `S len'  
  against inferred type `Add Z Z'
```

```
Expected type:Vec e (S len)
```

```
Inferred type:Vec e (Add Z Z)
```

```
In the second argument of `($)', namely `vappend Nil Nil'
```

```
In the expression: vhead $ vappend Nil Nil
```

```
*TypeLevel> vhead $ vappend Nil (Cons 1 Nil)
```

```
|
```

```
*TypeLevel> vhead $ vappend (Cons 1 Nil) Nil
```

```
|
```

```
*TypeLevel> vhead $ vappend (Cons 1 Nil) (Cons 2 Nil)
```

```
|
```

# 長さ付きリストの問題

- filter を定義できない
  - 結果の型を静的に決定できない
- Filter 型関数は定義できるけれど.....
  - どうやって値（式）レベルから型レベルに持っていくのか
    - GHC API ?、Template Haskell ?

話は戻って

# 型族の種類

- 型同義名族 (type synonym families)
  - 関数として働くのみ
  - e.g. Add 関数
- データ型族 (data type families)
  - データ型を定義
  - 関数適用後、データ型として残る

# データ型族

- 関数定義

```
data family TInteger n m
```

```
data instance TInteger Z x = Minus x
```

```
data instance TInteger (S x) Z = Plus x
```

- 型

```
*TypeLevel> :type Minus
```

```
Minus :: x -> TInteger Z x
```

```
*TypeLevel> :type Plus
```

```
Plus :: x -> TInteger (S x) Z
```

# 型族の種類(案)

- クラス族 (class families)
  - 型クラスを定義
  - アイデア段階
    - (採用されないかも.....)

# 関連型

- 元々は C++ から輸入したものの
- 型族の構文糖衣 (syntax sugar)
- 型クラスを使用
- 関数従属 (Functional Dependencies)同様に

# 関連型の例

```
class GMapKey k where
```

```
data GMap k :: * -> *
```

```
empty      :: GMap k v
```

```
lookup     :: k -> GMap k v -> Maybe v
```

```
insert     :: k -> v -> GMap k v -> GMap k v
```

```
instance GMapKey Double where
```

```
data GMap Double v = GMapDouble
```

```
(Data.Map.Map Double v)
```

```
empty      = GMapDouble Data.Map.empty
```

```
lookup k (GMapDouble m) = Data.Map.lookup k m
```

```
insert k v (GMapDouble m) = GMapDouble
```

```
(Data.Map.insert k v m)
```

# 関連型の例

```
instance GMapKey Int where
  data GMap Int v      = GMapInt
                        (Data.Map.IntMap Int v)
  empty                = GMapInt Data.Map.empty
  lookup k (GMapInt m) = Data.IntMap.lookup k m
  insert k v (GMapInt m) = GMapInt
                          (Data.IntMap.insert k v m)

instance (GMapKey a, GMapKey b) => GMapKey (a, b) where
  data GMap (a, b) v      = GMapPair (GMap a (GMap b v))
  empty                   = GMapPair empty
  lookup (a, b) (GMapPair gm) = lookup a gm >>= lookup b
  insert (a, b) v (GMapPair gm) = GMapPair $ case lookup a gm of
    Nothing -> insert a (insert b v empty) gm
    Just gm2 -> insert a (insert b v gm2 ) gm
```

# 比較

- **\*\*Monad** クラス
- 同値性制約による型付けの強制  
(equality coercion)
- ループを防ぐための制限
- 他の機能との連携

# \*\*\*Monad (関数従属)

```
class (Monad m) => MonadState s m | m -> s where  
  get :: m s  
  put :: s -> m ()
```

```
instance MonadState s (State s) where  
  get = State $ \s -> (s, s)  
  put s = State $ \_ -> ((), s)
```

```
-- Needs -XUndecidableInstances
```

```
instance (MonadState s m)  
  => MonadState s (ListT m) where  
  get = lift get  
  put = lift . put
```

# \*\*Monad (型族)

```
type family StateType (m :: * -> *)  
class (Monad m) => MonadState m where  
  get :: m (StateType m)  
  put :: StateType m -> m ()
```

```
type instance StateType (State s) = s  
instance MonadState (State s) where  
  get = State $ \s -> (s, s)  
  put s = State $ \_ -> ((), s)
```

```
type instance StateType (ListT m) = StateType m  
instance (MonadState m)  
  => MonadState (ListT m) where  
  get = lift get  
  put = lift . put
```

# \*\*Monad (関連型)

```
class (Monad m) => MonadState m where
  type StateType m
  get :: m (StateType m)
  put :: StateType m -> m ()

-- State = StateT Identity
instance (Monad m)
  => MonadState (Lazy.StateT s m) where
  type StateType (Lazy.StateT s m) = s
  get = Lazy.get
  put = Lazy.put

instance (MonadState m)
  => MonadState (ListT m) where
  type StateType (ListT m) = StateType m
  get = lift get
  put = lift . put
```

# 型付けの強制（関数従属）

- 関数従属を型付けの強制に使用

```
class Col a b | a->b where
  inCol :: a -> b
instance Col a [a] where
  inCol x = [x]
```

- a が Bool 型の場合には b は [Bool]

```
*Main> :t head $ inCol True
Bool
```

# 型付けの強制 (関数従属)

- a と b は同じ型

```
class TypeCast a b | a->b, b->a where
  typeCast :: a->b
instance TypeCast a a where
  typeCast = id
```

- TypeCast クラスを使った同様の定義

```
class Col' a b where
  inCol' :: a -> b
instance TypeCast a b
  => Col' a [b] where
  inCol' x = [typeCast x]
```

# 型付けの強制

## (型族、関連型)

- a と b は同じ型

a ~ b

- ただし、スーパークラスで同値性を示すための機能が未実装 (GHC 6.12?)

```
-- This doesn't work:  
class (a ~ b) => E a b  
cast :: E a b => a -> b  
cast a = a
```

# 制限

- 型推論時のループ発生を防ぐ
- 関数従属
  - 対応範囲条件 (Coverage Condition)
- 型族、関連型
  - 強い終了条件 (Strong Termination Condition)  
を緩和した終了条件 (Relaxed Condition)

# 制限を取り除きたい状況 (関数従属)

- \*\*Monadクラスのインスタンス
- 型レベルでプログラミングする時全般
- keigoいさんが色々知っているはず...

# 制限を取り除きたい状況 (型族)

- 緩和された終了条件でも定義不可能
  - e.g. 入れ子になった型関数定義

```
type family Times x y
type instance Times x Z = Z
-- Needs UndecidableInstances
type instance Times x ( S y ) = Add x ( Times x y )
```

# 別の解決策 (案)

- 値に対する関数なら検証可能なものも
- 値に対する関数と同じように、型関数の網羅性を検証すればいいのでは？
- 全体型族 (total type families)

# 全体型族 (案)

```
type Add n m where
  Add Z x = x
  Add (S x) y = S (Add x y)
```

```
type Times x y where
  Times x Z = Z
  Times x (S y) = Add x (Times x y)
```

- 最後の行に以下が書かれていると仮定

```
Add x y = VOID
Times x y = VOID
```

- *VOID* は型名ではなく、型検査で弾く  
という意味 (のはず.....)

# 他の機能との連携

- GADT
  - 関数従属では未対応
  - 型族、関連型では対応
- Template Haskell
  - 関数従属には対応済み
  - 型族、関連型への対応は GHC 6.12 から

# 型族の利点・欠点

- より直接的に記述できる
- 将来的には、より安全に利用できる機能になる可能性あり
- 現時点では未実装な機能も多い
  - 結果、表現力で関数従属に劣ることも...

# ライブラリ

- tfp: Type-level programming library using type families
- Salsa: A .NET Bridge for Haskell
- Fun with type functions の例
  - 関数従属を使った既存のライブラリを書き直す

# tfp

- Bool、比較、数値、List 演算を定義
  - SizedWord、SizedInt
  - Prelude の関数を網羅予定？
    - 後継 (typelib) の開発者次第
- Mac の GHC 6.10.1 ではリンクに失敗して使えませんでした.....

# Salsa

- darcs 版でないと GHC 6.10.x で利用できないので注意
- コンパイル時性能のバグ？
- .NET の型を Haskell に持っていくのに型レベルプログラミングを利用
- 暗黙の型変換 (implicit conversion) も実現

# Fun with type functions

- 色々面白い例が載るようです
- 公開を待ってください

# まとめ

- Haskell の型族と関連型は、型レベルプログラミングのための強力な道具
- 今後の GHC の進化に期待してください