

Put Curry-Howard to Work

Maciej Buszka

Instytut Informatyki UWr

17.12.2018

Wprowadzenie

- Artykuł, na podstawie którego stworzona jest ta prezentacja pojawił się w 2005 roku.
- W tamtych czasach Haskell nie posiadał rozszerzeń opisanych w artykule.
- Dlatego autor opisuje je w języku Ω mega - podobnym do Haskell'a
- Aktualnie wszystkie rozszerzenia są dostępne w GHC (aczkolwiek niektóre mają inną implementację)
- Na tej prezentacji będę korzystał z Haskell'a

Typy i wartości

- W Haskellu mamy do dyspozycji typy wbudowane np. **Int**, **Char**
- Zdefiniowane w bibliotece standardowej np. **String**, **[a]**
- Oraz zdefiniowane przez użytkownika:

```
data Person = Person
  { name  :: String
  , age   :: Int
  }
```

```
data Maybe a
  = Just a
  | Nothing
```

Typy i wartości

Definicja **data** `Person = Person { ... }` wprowadza:

- Nową stałą typową `Person`
- Nowy konstruktor `Person :: String → Int → Person`

Natomiast **data** `Maybe a = Just a | Nothing` wprowadza:

- Nowy konstruktor typów **Maybe**
- Dwa konstruktory:
- **Just** `:: a → Maybe a`
- **Nothing** `:: Maybe a`

Rodzaje i typy

- Tak jak klasyfikujemy wartości za pomocą typów, typy są klasyfikowane poprzez rodzaje
- W standardowym Haskellu dostępna jest jedna stała rodzajowa `Type` opisująca typy wartości np: **`Int :: Type`**
- Oraz jeden konstruktor rodzajów \rightarrow
- Przykładowo konstruktor typów może mieć rodzaj **`Maybe :: Type \rightarrow Type`**
- Nazwę `Type` należy zaimportować z modułu `Data.Kind`
- Kiedyś `Type` nazywał się `*`

Rodzaje definiowane przez użytkownika

- Rozszerzenie DataKinds pozwala na użycie definicji danych jako definicji nowego rodzaju
- Przykładowo definicja

data Nat = Z | S Nat

wprowadza rodzaj 'Nat, stałą typową 'Z :: 'Nat oraz konstruktor typów 'S :: 'Nat → 'Nat

- Jeżeli jest to jednoznaczne można opuścić '
- Warto zauważyć, że takie promowane typy nie klasyfikują żadnych wartości

Klasyfikacje rodzajów

- Gdy już mamy ciekawszy język typów i rodzajów pojawia się naturalne pytanie jak je sklasyfikować
- Jednym z podejść jest wybrane w artykule, polegające na konstrukcji hierarchii rodzajów:
 - **Int** :: Type0, **Int** → **Bool** :: Type0,
 Maybe :: Type0 → Type0
 - Type0 :: Type1, Type0 → Type0 :: Type1
 - rodzaj z poziomu $n + 1$ klasyfikuje rzeczy z poziomu n
- Natomiast w Haskellu postanowiono dodać aksjomat
Type :: Type

GADTs

- Rozszerzenie GADTs pozwala na ogólniejsze definicje konstruktorów
- Korzystając z nowej składni możemy definiować algebraiczne typy danych jak wcześniej:

```
data Maybe a where  
  Just      :: a → Maybe a  
  Nothing   :: Maybe a
```


GADTs

- Rozszerzenie GADTs pozwala na ogólniejsze definicje konstruktorów
- Ale także:

```
data IntOrBool a where  
  AnInt  :: IntOrBool Int  
  ABool  :: IntOrBool Bool
```

- Takie ukonkretnienie typu jest widoczne podczas destrukcji:

```
check :: IntOrBool a → String  
check (AnInt i) = "An_Int_" ++ show (i + 42)  
check (ABool b) = "A_Bool_" ++ show (not b)
```

GADTs

Definicję postaci:

```
data IntOrBool a where  
  AnInt  :: IntOrBool Int  
  ABool  :: IntOrBool Bool
```

możemy równoważnie przepisać jako:

```
data IntOrBool a where  
  AnInt  :: (a ~ Int)  ⇒ IntOrBool a  
  ABool  :: (a ~ Bool) ⇒ IntOrBool a
```

gdzie \sim wprowadza nowe ograniczenie do rozwiązania przez system typów.

Rodziny typów

- Rozszerzenie TypeFamilies pozwala nam na definicje funkcji na poziomie typów
- Przykładowo:

```
type family Add n m :: Nat where  
  Add Z      m = m  
  Add (S n) m = S (Add n m)
```

definiuje funkcję dodającą do siebie dwie liczby naturalne zakodowane jako rodzaj Nat

Przykłady i wzorce

Teraz popatrzymy na różne przykłady i wzorce zastosowań