Putting Curry-Howard to Work

Maciej Buszka

Instytut Informatyki UWr

17.12.2018

Curry-Howard vs programowanie

- Izomorfizm Curry'ego-Howarda mówi nam o odpowiedniości typów i twierdzeń oraz dowodów i programów
- Jednak twierdzenia które dowodzimy pisząc w klasycznym języku programowania nie są specjalnie ciekawe
- Chcielibyśmy aby typy stanowiły twierdzenia o właściwościach programów które piszemy
- W tym celu musimy rozszerzyć język typów

Uwagi techniczne

- 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 (Tim Sheard) opisuje je w języku Ωmega - podobnym do Haskella
- Aktualnie wszystkie rozszerzenia są dostępne w GHC (aczkolwiek niektóre mają inną implementację)
- Na tej prezentacji będę korzystał z Haskella, kompilowanego `GHC 8.6.x`

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

$$\textbf{data}\ \ \mathsf{Nat} = \mathsf{Z} \mid \mathsf{S}\ \ \mathsf{Nat}$$

  wprowadza dodatkowo rodzaj $'\mathsf{Nat}$, stałą typową $'\mathsf{Z} :: '\mathsf{Nat}$ oraz konstruktor typów $'\mathsf{S} :: '\mathsf{Nat} \to '\mathsf{Nat}$
- 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** $\rightarrow$ **Bool** :: Type0, **Maybe** :: Type0 $\rightarrow$Type0
  - Type0 :: Type1, Type0 $\rightarrow$ 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

- przykłady z artykułu, przetłumaczone na Haskella
- GLambda autorstwa Richarda Eisenberga
- Dependently typed Haskell autorstwa Stephanie Weirich
- biblioteka singletons pozwalająca na automatyczne generowanie singletonów
- Kod z artykułów Justina Le

Bibliografia


Tim Sheard
Putting Curry-Howard to Work


Richard A. Eisenberg
Dependently Typed Programming with Singletons.


Justin Le
Introduction to Singletons.


GLambda


Dependently typed Haskell


singletons