Programowanie funkcyjne
Projekt - Haskell

Gabriela Białoskórska, Mikołaj Knysak, Ignacy Tekieli
Fizyka Techniczna

Grudzień 2020

Spis treści

1 Wprowadzenie

1.1 Cel pracy

Celem niniejszej pracy było stworzenie małej, lecz kompletnej biblioteki w Haskellu, w celu manipulowania danymi oraz ich serializacji. Dane te były w postaci JSON-a (*JavaScript Object Notation*).

1.2 Wstęp teoretyczny

1.2.1 Haskell

Haskell to czysto funkcyjny język programowania, nazwany na cześć Haskella Curry'ego. Ma on ogólne przeznaczenie. Powstał w celu połączenia wszystkich atutów programowania funkcyjnego w jednym eleganckim, silnym i ogólnie dostępnym języku programowania. Najważniejszą jego cechą jest to, że nie umożliwia występowania żadnych efektów ubocznych.

Inną ważną cechą języka Haskell jest bycie językiem „leniwym" („not-strict"). Oznacza to, że wartość żadnego wyrażenia nie jest wyznaczana dopóki nie jest konieczna. Pozwala to na wiele praktycznych i eleganckich rozwiązań mnóstwa problemów. Przykładem może być zdefiniowanie listy możliwych rozwiązań oraz przefiltrowanie jej, usuwając rozwiązania niedopuszczalne. Pozostała lista będzie zawierała rozwiązania dopuszczalne. Leniwe wartościowanie czyni tę operację bardzo przejrzystą.

Haskell jest również językiem stosującym silne typowanie. Oznacza to, że niemożliwa jest na przykład przypadkowa konwersja Double do Int. Prowadzi to do zmniejszenia liczby powstałych błędów; cecha ta bywa pomocna w zlokalizowaniu ich w kodzie.

W przeciwieństwie do wielu języków stosujących silne typowanie, typy w Haskellu są automatycznie rozpoznawane, wobec czego bardzo rzadko konieczne jest deklarowanie typu funkcji. Często takie definicje służą jedynie dokumentacji kodu. Haskell stara się wywnioskować typ z kontekstu w jakim zmienna została użyta. Następnie sprawdzana jest zgodność typów we wszystkich operacjach, w celu wykluczenia niedopasowania typów.

Program napisany w języku Haskell i skompilowany przy użyciu GHC wykonuje się z prędkością porównywalną do języków C czy C++. Różnice w szybkości są jednak tak małe, że niemalże nie mają znaczenia. Oczywiście nie dotyczy to aplikacji takich jak np. kodery MPEG, czy inne aplikacje wykonujące złożone obliczenia, które większość czasu działania spędzają na wykonywaniu małej części kodu. W takich przypadkach zdecydowanie lepszym rozwiązaniem jest zastosowanie języka C++. Należy również pamiętać, że optymalizacja algorytmu może dać lepsze rezultaty niż optymalizacja kodu. Jeśli aplikacja w Haskellu zostanie napisana w znacznie krótszym czasie niż w języku C++, pozostanie więcej czasu na pracę nad poprawą samego algorytmu.

1.2.2 JSON

JSON jest lekkim, tekstowym formatem wymiany danych komputerowych, bazującym na podzbiorze języka JavaScript. Pomimo nazwy, jest on formatem niezależnym od konkretnego języka. Wiele języków programowania obsługuje go poprzez zastosowanie dodatkowych pakietów bądź bibliotek.

JSON, podobnie jak inne struktury służące do przechowywania danych takich jak np. XML, ma szerokie zastosowanie. Najczęściej jednak jest wykorzystywany do przekazywania i odbierania danych z serwera przez aplikacje na stronie internetowej, na przykład kiedy użytkownik loguje się na do swojego konta w grze przeglądarkowej.

2 Materiały i metody

2.1 Wiadomości wstępne

2.1.1 Reprezentacja danych JSON w Haskellu

W celu rozpoczęcia pracy z danymi JSON w Haskellu, korzystaliśmy z danych algebraicznych, aby reprezentować zakres możliwych typów JSON:

```
-- file: ch05/SimpleJSON.hs
data JValue = JString String
            | JNumber Double
            | JBool Bool
            | JNull
            | JObject [(String, JValue)]
            | JArray [JValue]
              deriving (Eq, Ord, Show)
```

Każdemu typowi JSON dostarczyliśmy odrębny konstruktor wartości. Niektóre z tych konstruktorów mają parametry: jeśli chcemy skonstruować JSON string, musimy podać wartość String jako argument do konstruktora JString. Aby rozpocząć pracę z tym kodem, zapisaliśmy plik SimpleJSON.hs w naszym edytorze, a następnie przełączyliśmy się do okna ghci i wczytaliśmy plik do ghci:

```
ghci> :load SimpleJSON
[1 of 1] Compiling SimpleJSON         ( SimpleJSON.hs, interpreted )
Ok, modules loaded: SimpleJSON.
ghci> JString "foo"
JString "foo"
ghci> JNumber 2.7
JNumber 2.7
ghci> :type JBool True
JBool True :: JValue
```

Wiedzieliśmy w jaki sposób użyć konstruktora aby wartość normalną w Haskellu przekształcić w JValue. Aby odwrócić ten proces, dodaliśmy funkcję *SimpleJSON.hs*, która wyodrębnia dla nas string z wartości JSON. Jeśli ta wartość rzeczywiście zawiera string, wówczas funkcja "złapie" go konstruktorem `Just`. W przeciwnym wypadku zwróci `Nothing`.

```
-- file: ch05/SimpleJSON.hs
getString :: JValue -> Maybe String
getString (JString s) = Just s
getString _           = Nothing
```

Zapisany, zmodyfikowany plik źródłowy mogliśmy załadować ponownie w ghci i użyć nowej definicji. Polecenie `:reload` zapamiętuje ostatni wczytany przez nas plik.

```
ghci> :reload
Ok, modules loaded: SimpleJSON.
ghci> getString (JString "hello")
Just "hello"
ghci> getString (JNumber 3)
Nothing
```

Zastosowanie kilku dodatkowych funkcji pomocniczych sprawiło, że mieliśmy gotowy, podstawowy kod do pracy:

```
-- file: ch05/SimpleJSON.hs
getInt (JNumber n) = Just (truncate n)
getInt _           = Nothing

getDouble (JNumber n) = Just n
getDouble _           = Nothing

getBool (JBool b) = Just b
getBool _         = Nothing

getObject (JObject o) = Just o
getObject _           = Nothing

getArray (JArray a) = Just a
getArray _          = Nothing

isNull v            = v == JNull
```

Funkcja `truncate` zamienia liczbę zmiennoprzecinkową lub wymierną na całkowitą, pomijając cyfry po przecinku:

```
ghci> truncate 5.8
5
ghci> :module +Data.Ratio
ghci> truncate (22 % 7)
3
```

2.1.2 Budowa modułu w Haskellu

Moduł to zbiór powiązanych ze sobą funkcji, typów i klas. Język Haskell stanowi zbiór modułów, w którym moduł główny załącza pozostałe moduły w celu użycia zawartych w nich funkcji do określonego celu.

Plik źródłowy Haskella zawiera definicję pojedynczego modułu. Zaczyna się on od deklaracji modułu. Musi poprzedzać wszystkie inne definicje w pliku źródłowym:

```
-- file: ch05/SimpleJSON.hs
module SimpleJSON
    (
      JValue(..)
    , getString
    , getInt
    , getDouble
    , getBool
    , getObject
    , getArray
    , isNull
    ) where
```

Słowo `module` jest zarezerwowane. Po nim następuje nazwa modułu, która musi zaczynać się wielką literą. Plik źródłowy musi mieć taką samą *nazwę podstawową* (komponent przed przyrostkiem), jak nazwa modułu który zawiera. Dlatego nasz plik *SimpleJSON.hs* zawiera moduł o nazwie `SimpleJSON`.

Po nazwie modułu znajduje się lista *eksportów*, umieszczona w nawiasach. Słowo kluczowe `where` wskazuje, że następuje treść modułu. Lista eksportów określa, które nazwy w tym module są widoczne dla innych modułów. Specjalny zapis występujący po nazwie `JValue` wskazuje, że eksportujemy zarówno typ, jak i wszystkie jego konstruktory.

2.1.3 Kompilacja

Oprócz interpretera `ghci` dystrybucja GHC zawiera kompilator, `ghc`, który generuje kod natywny.

W celu skompilowania pliku źródłowego, w pierwszej kolejności otwieramy terminal lub okno wiersza poleceń, a następnie wywołujemy `ghc` z nazwą pliku źródłowego do kompilacji:

```
ghc -c SimpleJSON.hs
```

Opcja -c mówi `ghc`, aby generował tylko kod obiektowy.

Po zakończeniu działania `ghc`, jeśli wypiszemy zawartość katalogu, powinien on zawierać dwa nowe pliki: *SimpleJSON.hi* i *SimpleJSON.o*. Pierwszy to plik interfejsu, w którym `ghc` przechowuje informacje o nazwach wyeksportowanych z naszego modułu w postaci dostępnej do odczytu maszynowego. Ten ostatni jest plikiem obiektowym, który zawiera wygenerowany kod maszynowy.

2.1.4 Tworzenie programu i importowanie modułów

Mając sukcesywnie skompilowaną, niewielką bibliotekę, możemy napisać krótki program testowy. Tworzymy i zapisujemy plik o nazwie *Main.hs*:

```
-- file: ch05/Main.hs
module Main () where

import SimpleJSON

main = print (JObject [("foo", JNumber 1), ("bar", JBool False)])
```

Zwróćmy uwagę na dyrektywy `import`, które następują po deklaracji modułu. Oznacza to, że chcemy pobrać wszystkie nazwy wyeksportowane z modułu `SimpleJSON` i udostępnić je w naszym module. Wszelkie dyrektywy importu muszą pojawić się w grupie na początku modułu, po deklaracji modułu, ale przed całym innym kodem.

Wybór nazewnictwa dla pliku źródłowego i funkcji jest celowy. Aby utworzyć plik wykonywalny, `ghc` oczekuje modułu o nazwie `Main`, który zawiera funkcję o nazwie `main` (funkcja główna to ta, która zostanie wywołana, gdy uruchomimy program po jego zbudowaniu).

```
ghc -o simple Main.hs SimpleJSON.o
```

Tym razem pominęliśmy opcję -c w trakcie wywoływania `ghc`, więc spróbuje on wygenerować plik wykonywalny. Proces generowania pliku wykonywalnego nazywa się *łączeniem* (ang. linking). Jak sugeruje nasz wiersz poleceń, `ghc` jest w stanie doskonale zarówno skompilować pliki źródłowe, jak i połączyć plik wykonywalny w jednym wywołaniu.

Przekazaliśmy `ghc` nową opcję -o, która przyjmuje jeden argument: nazwę pliku wykonywalnego, który powinien utworzyć `ghc`. W tym przypadku zdecydowaliśmy się nazwać program *simple*.

Na koniec podaliśmy nazwę naszego nowego pliku źródłowego, *Main.hs*, oraz plik obiektowy, który już skompilowaliśmy, *SimpleJSON.o*. Musimy jawnie wymienić wszystkie nasze pliki, które zawierają kod, jaki powinien znaleźć się w pliku wykonywalnym. Jeśli zapomnimy o pliku źródłowym lub obiektowym, `ghc` będzie wskazywać na niezdefiniowane symbole, co oznacza, że niektóre definicje których potrzebuje, nie są zawarte w dostarczonych przez nas plikach.

Podczas kompilacji możemy przekazać `ghc` dowolną mieszankę plików źródłowych i obiektowych. Jeśli `ghc` zauważy, że już skompilował plik źródłowy do pliku obiektowego, przekompiluje plik źródłowy tylko wtedy, gdy go zmodyfikowaliśmy.

Gdy `ghc` zakończy kompilację i linkowanie naszego programu *simple*, możemy uruchomić go z wiersza poleceń.

2.1.5 Zwracanie danych JSON

Mając reprezentację Haskell dla typów JSON, mamy możliwość pobierania wartości Haskell i renderowania ich jako danych JSON. Można to zrobić na kilka sposobów. Najbardziej bezpośrednim z nich jest napisanie funkcji renderującej, która drukuje wartość w postaci JSON.

```
-- file: ch05/PutJSON.hs
module PutJSON where

import Data.List (intercalate)
import SimpleJSON

renderJValue :: JValue -> String

renderJValue (JString s)   = show s
renderJValue (JNumber n)   = show n
renderJValue (JBool True)  = "true"
renderJValue (JBool False) = "false"
renderJValue JNull         = "null"

renderJValue (JObject o) = "{" ++ pairs o ++ "}"
  where pairs [] = ""
        pairs ps = intercalate ", " (map renderPair ps)
        renderPair (k,v)   = show k ++ ": " ++ renderJValue v

renderJValue (JArray a) = "[" ++ values a ++ "]"
  where values [] = ""
        values vs = intercalate ", " (map renderJValue vs)
```

Dobry styl programowania w Haskellu opiera się na umiejętności oddzielenia czystego kodu od kodu wykonującego operacje I/O. Nasza funkcja `renderJValue` nie ma interakcji ze światem zewnętrznym, ale nadal musi być w stanie wydrukować `JValue`:

```
-- file: ch05/PutJSON.hs
putJValue :: JValue -> IO ()
putJValue v = putStrLn (renderJValue v)
```

Zwracanie wartości JSON jest już zatem proste.

2.2 Tworzenie projektu

2.2.1 Instalacja programu do stworzenia projektu

Program zainstalowaliśmy zgodnie z poleceniami dostępnymi na stronie:
`https://docs.haskellstack.org/en/stable/README/`
Jest to bardzo prosty proces. W terminalu użyliśmy komendy:

```
curl -sSL https://get.haskellstack.org/ | sh
```

a następnie: `stack upgrade`.
Tworzenie nowego projektu wygląda następująco:

```
stack new my-project
cd my-project
stack setup
stack build
stack exec my-project-exe
```

lub, jeśli chcemy uruchomić REPL, w wierszu poleceń wpisujemy `stack ghci`

2.2.2 Budowa i opis działania programu

Budowa naszego programu nieco różni się od tej opisanej powyżej (więcej informacji na ten temat znajduje się w podsumowaniu raportu), jednak cel oraz metodyka nie uległy zmianie - stąd ich rozległy opis we wstępie. Skupmy się zatem na najważniejszym fragmencie kodu, odpowiedzialnym za tzw. **validation**. Jest to kwintesecja wykonania zadania - program sprawdza, czy przewidywane wartości JSON zgadzają się z rzeczywistymi:

```
testJsonText =
  unlines
    [ "{"
    , "    \"hello\": [false, true, null, 42, \"foo\\n\\u1234\\\"\", [1, -2, 3.1415, 4e-6, 5E6, 0.123e+1]],"
    , "    \"world\": null"
    , "}"
    ]
expectedJsonAst =
  JsonObject
    [ ( "hello"
      , JsonArray
          [ JsonBool False
          , JsonBool True
          , JsonNull
          , JsonNumber 42
          , JsonString "foo\n\4660\""
          , JsonArray
              [ JsonNumber 1.0
              , JsonNumber (-2.0)
              , JsonNumber 3.1415
              , JsonNumber 4e-6
              , JsonNumber 5000000
              , JsonNumber 1.23
              ]
          ])
    , ("world", JsonNull)
    ]
```

Rysunek 1: *Sprawdzanie zgodności wpisanych wartości ("hello" oraz "world")*

W następnej sekcji raportu sprawdzimy w jaki sposób prezentują się możliwe warianty wykonania programu.

3 Wyniki

Rozważmy dwa przypadki. Pierwszy, przedstawiony powyżej - widzimy już w samym kodzie, że wartości są zgodne. Sprawdźmy to, wpisując w terminalu `stack ghci Main.hs` a następnie `main`:

```
*Main> main
[INFO] JSON:
{
    "hello": [false, true, null, 42, "foo\n\u1234\"", [1, -2, 3.1415, 4e-6, 5E6, 0.123e+1]],
    "world": null
}

[INFO] Parsed as: JsonObject [("hello",JsonArray [JsonBool False,JsonBool True,JsonNull,JsonNumber 42.0,JsonS
tring "foo\n\4660\"",JsonArray [JsonNumber 1.0,JsonNumber (-2.0),JsonNumber 3.1415,JsonNumber 4.0e-6,JsonNumb
er 5000000.0,JsonNumber 1.23]]),("world",JsonNull)]
[INFO] Remaining input (codes): [10]
[SUCCESS] Parser produced expected result.
```

Rysunek 2: *Wynik dla wartości zgodnych*

Zobaczmy co stanie się w przypadku, gdy wartości nie będą zgodne. Zauważmy że są one w formie Stringów, zatem nawet drobna zmiana, taka jak np. wielkość litery, stanowi kolosalną różnicę:

```
[INFO] JSON:
{
    "Hello": [false, true, null, 42, "foo\n\u1234\"", [1, -2, 3.1415, 4e-6, 5E6, 0.123e+1]],
    "World": null
}

[INFO] Parsed as: JsonObject [("Hello",JsonArray [JsonBool False,JsonBool True,JsonNull,JsonNumber 42.0,JsonString "foo\
n\4660\"",JsonArray [JsonNumber 1.0,JsonNumber (-2.0),JsonNumber 3.1415,JsonNumber 4.0e-6,JsonNumber 5000000.0,JsonNumbe
r 1.23]]),("World",JsonNull)]
[INFO] Remaining input (codes): [10]
[ERROR] Parser produced unexpected result. Expected result was: JsonObject [("hello",JsonArray [JsonBool False,JsonBool
True,JsonNull,JsonNumber 42.0,JsonString "foo\n\4660\"",JsonArray [JsonNumber 1.0,JsonNumber (-2.0),JsonNumber 3.1415,Js
onNumber 4.0e-6,JsonNumber 5000000.0,JsonNumber 1.23]]),("World",JsonNull)]
```

Rysunek 3: *Wynik dla wartości niezgodnych*

4 Podsumowanie

Kod źródłowy naszego programu jest nieco bardziej zaawansowany niż opisana początkowo metodyka działania, jednak zasada działania pozostaje ta sama. Rozdział 2.1 niniejszego raportu był wzorowany na dołączonej do wykładów literaturze - "Real World Haskell" autorstwa Bryana O'Sullivana, Dona Stewarta oraz Johna Goerzena. Postanowiliśmy wykorzystać ją do stworzenia wstępu, ponieważ zawierała w sobie kluczowe informacje opisane w streszczonej formie.

Zaprezentowany projekt powstał na podstawie innego źródła. Wzorowaliśmy się na pracy powstałej w trakcie streama - można ją znaleźć na YouTube: `https://www.youtube.com/watch?v=N9RUqGYuGfw` (w opisie filmu znajduje się link do repozytorium w `GitHub`) Nie różni się ona wiele od tej opisanej - w głównej mierze obsługuje więcej typów różnych liczb, co jest widoczne na załączonym fragmencie kodu. Zawiera nieco więcej opcji, stąd ilość kodu może wydawać się kolosalnie większa, jednak jak pokazaliśmy - opiera się na tej samej metodyce.

Zdecydowaliśmy się skorzystać z podanego filmu, ponieważ był on dla nas bardziej czytelny. Początkowo zetknęliśmy się z problemami dotyczącymi samej instalacji Stacka - wciąż nie wiemy dlaczego nie była ona możliwa na systemie macOS Big Sur, a powiodła się na macOS Catalina. Późniejsze pisanie w języku funkcyjnym również nie było dla nas intuicyjne, stąd forma obserwacji "krok po kroku" wykonania zadania na filmie najbardziej do nas przemawiała.

Korzystając z dostępnej literatury staraliśmy się jak najbardziej zrozumieć napisany kod (również dlatego zachowaliśmy w nim komentarze). Mamy nadzieję, że zdobytą wiedzę udało nam się udowodnić zarówno w niniejszym raporcie, jak i w trakcie prezentacji.

5 Bibliografia

1. Literatura z zajęć: *"Real World Haskell* - Bryan O'Sullivan, Don Stewart, John Goerzen (O'Reilly Media, Inc. 2008). Rozdział 5. *"Writing a library: working with JSON data"*

2. Instalacja *Haskell Tool Stack*:
`https://docs.haskellstack.org/en/stable/README/`

3. Film *JSON Parser From Scratch in Haskell*:
`https://www.youtube.com/watch?v=N9RUqGYuGfw`