Programowanie Funkcyjne 2020

Lista zadań nr 13 (bonusowa) dla grupy mabi, mbu, ppo i efes

Na zajęcia 2 i 3 lutego 2021

Zadania na ostatnie zajęcia mają charakter bonusowy. Oprócz poniższych zadań można również zgłaszać *ładnie rozwiązane* zadania o grach, interpreterze Brainfucka i wyrażeniach regularnych z poprzednich list.

Tegotygodniowe zadania mają charakter dodatkowy i istotnie wykraczają poza treści omawiane na wykładnie. Zachęcam jednak do próby ich rozwiązania — rozwiązania częściowe i różnego rodzaju próby rozwiązań są bardzo mile widziane!

W poniższych zadaniach zajmiemy się bardzo współczesnym podejściem do programowania z efektami w czystych językach, które jest całkowicie odmienne od monad — efektami algebraicznymi. Efekty algebraiczne są dość nowym pomysłem (ostatnie dziesięciolecie) i nie doczekały się jeszcze implementacji w żadnym produkcyjnym języku programowania, dlatego musimy użyć jakiegoś eksperymentalnego języka programowania. Poniższe przykłady podaję w języku *Helium*[1], który jest rozwijany w naszym instytucie, ale zadanie można rozwiązać również w innym podobnym języku, np. *Koka*[2], *Eff*[3] albo *Frank*[4].

Najnowsza wersja Heliuma wprowadza dużo istotnych zmian w stosunku do poprzednich wersji i nie wszystkie jej elementy są jeszcze dopracowane, w szczególności funkcje rekurencyjne nie mogą być niejawnie polimorficzne, a *pretty-printer* używany w komunikatach o błędach nie odpowiada w pełni składni konkretnej. Dlatego zalecam użycie starszej, lepiej dopracowanej wersji oznaczonej tagiem `popl19`. W tym celu najłatwiej będzie sklonować całe repozytorium i skompilować odpowiednią wersję.

```
$ git clone https://bitbucket.org/pl-uwr/helium.git
$ cd helium
$ git checkout popl19
$ make
```

Istotną cechą języków programowania z efektami algebraicznymi jest to, że możemy sami definiować własne efekty obliczeniowe podając ich interfejs: zbiór operacji które faktycznie wykonują jakiś efekt (odpowiadający operacjom o które rozszerzaliśmy monady w ostatnich tygodniach). Na przykład możemy zdefiniować efekt `Reader`, który ma jedną operację, odpytującą się o wartość jakiegoś ukrytego parametru (typu `Int`).

```
effect Reader =
  { ask : Unit => Int
  }
```

Od teraz możemy używać operacji `ask` jako zwykłej funkcji, np.

```
let foo x = ask () + x
```

Jednak co robi funkcja `foo`? Ma ona typ `Int ->[Reader] Int`, co oznacza, że jej wykonanie może spowodować efekt `Reader`, więc można ją uruchomić tylko w kontekście, w którym efekt `Reader` (więc i operacja `ask`) ma jakieś znaczenie. Możemy lokalnie nadawać znaczenie efektom za pomocą specjalnej konstrukcji, zwanej handlerem, która działa podobnie do dopasowania wzorca, tyle, że konkretne klauzule odpowiadają na dopasowanie się do operacji z danego efektu, które wystąpiły podczas wykonania obliczenia. Np. możemy napisać funkcję, która wywołuje funkcję `foo` w kontekście w którym operacja `ask` zawsze odpowiada wartością 42.

[1] `https://bitbucket.org/pl-uwr/helium`
[2] `https://bithub.com/koka-lang/koka`
[3] `https://www.eff-lang.org`
[4] `https://github.com/frank-lang/frank`

```
let bar x =
  handle foo x with
  | ask () => resume 42
  end
```

Handlery zachowują się podobnie do handlerów wyjątków: kod poszczególnych klauzul handlera wykonuje się w kontekście handlera, a nie konkretnej operacji (wyjątku), którą obsługujemy. W przeciwieństwie do wyjątków możemy wznowić przerwane obliczenie, dzięki niejawnie związanej zmiennej `resume`, która zawiera kontynuację przerwanego obliczenia. Podana funkcja oblicza $x + 42$. Możemy jednak napisać funkcję

```
let bar2 x =
  handle foo x with
  | ask () => 42
  end
```

która jest funkcją stałą zwracającą 42, albo nawet funkcję

```
let bar3 x =
  handle foo x with
  | ask () => resume 42 + resume 13
  end
```

która oblicza $2x + 55$.

Okazuje się, że taki mechanizm sterowania pozwala na wyrażenie dowolnego efektu obliczeniowego. Np. obliczenia używające mutowalnego stanu zadanego przez dwie operacje `put` i `get`:

```
effect State X =
  { put : X => Unit
  ; get : Unit => X
  }
```

można za pomocą handlera zinterpretować jako funkcje, które są bezpośrednio zaaplikowane do bieżącej wartości stanu.

```
let hState st0 c =
  handle c () with
  | put s    => fn _ => resume () s
  | get ()   => fn s => resume s  s
  | return x => fn _ => x
  end st0
```

Tu pojawia się nowy element handlerów: klauzula *return*, która mówi co zrobić z ostatecznym wynikiem `x` obliczenia wewnątrz handlera (`c ()`). W przypadku stanu zwracamy funkcję, która porzuci swój argument (bieżącą wartość stanu) i zwróci ostateczny wynik.

W starej implementacji Heliuma (tag `popl19`), funkcję przyjmującą inną funkcję o argumencie typu `Unit` można traktować jako handler, więc możemy napisać wyrażenie

```
handle
  let x = 13 in
  put 42;
  x + get ()
with hState 0
```

które obliczy się do wartości 55. W nowszej wersji Heliuma do reprezentacji handlerów służy osobny typ.

Efekty algebraiczne mają wiele przewag nad monadami. Po pierwsze, łatwiej jest używać wielu niezależnych efektów naraz: funkcje mogą mieć więcej niż jeden efekt, np. `fn () => get () + ask ()` ma typ `Unit ->[Reader, State Int] Int`. Po drugie mamy ten sam język do zapisu czystych obliczeń (takich co nie mają efektów), i nieczystych, zatem nie potrzebujemy notacji `do`. Komunikacja ze światem zewnętrznym też jest prosta: robimy to bezośrednio, tak jak np. w OCamlu. Wtedy obliczenia, które to

robią mają abstrakcyjny efekt `IO`, obsługiwany przez środowisko uruchomieniowe języka. Więcej informacji o programowaniu w Heliumie możesz znaleźć na wiki języka [5].

Zadanie 1 (5pkt). W SKOSie znajduje się plik `BF.he` (w dwóch wersjach) z implementacją parsera dla programów w języku *Brainfuck*, oraz implementacją kilku przydatnych funkcji. Rozwiąż problem braku modularności, z którym zetknęliśmy się na poprzedniej liście tj. zdefiniuj trzy niezależne efekty: do obsługi taśmy, czytania wejścia i pisania na wyjście (jeden z tych efektów masz już zdefiniowany). Następnie zdefiniuj interpreter, jako funkcję z `List BF` w `Unit`, która ma wszystkie trzy efekty. Na koniec napisz funkcję, o typie `String ->[IO] Unit`, która uruchomi podany podany program w języku *Brainfuck*, korzystając ze standardowego wejścia-wyjścia. Pamiętaj, że Helium pozwala obsłużyć tylko jeden efekt w handlerze, więc będziesz musiał zagnieździć handlery.

Zadanie 2 (5 pkt). Helium pozwala wyrazić obliczenia z nawrotami za pomocą następującego efektu:

```
effect BT =
  { flip : Unit => Bool // Wraca dwa razy
  ; fail : Unit => a    // Nie wraca nigdy
  }
```

Standardowy handler zbierający wyniki obliczeń na listę wygląda następująco:

```
let hBTList =
  handle
  | flip ()  => resume True ++ resume False
  | fail ()  => []
  | return x => [x]
  end
```

Używając efektów niedeterminizmu i mutowalnego stanu (efekt `State` zdefiniowany wyżej) zaimplementuj program rozwiązujący zadanie o wyrażeniach regularnych z poprzedniej listy.

Zadanie 3 (2pkt). W ubiegłym tygodniu próbowaliśmy wziąć pierwszy możliwy wynik obliczeń z nawracaniem za pomocą monady `Maybe`, ale nie przyniosło to zamierzonych rezultatów. Sprawdź, czy handlery również nas rozczarują. W tym celu zaimplementuj handler o następującej sygnaturze.

```
val hBTOpt : (Unit ->[BT|e] a) ->[|e] Option a
```

Następnie porównaj jego zachowanie z monadą `Maybe`.

[5] `https://bitbucket.org/pl-uwr/helium/wiki/Home`