Metody programowania 2017

Lista zadań na pracownię nr 5

Na poprzedniej liście zadań implementowaliśmy sprawdzanie typów oraz interpreter prostego języka, czyli wyrażeń arytmetyczno-logicznych. Teraz uczynimy ten język bardziej użytecznym, rozszerzając go o nowe konstrukcje: funkcje rekurencyjne (pierwszego rzędu), pary oraz listy.

Składnia

Większość definicji składni została podana w poprzednim zadaniu. Tutaj opiszemy tylko jej nowe elementy:

Typy teraz będą elementem składni i będziemy je oznaczać metazmiennymi τ, τ_1, τ_2,... Na typy składają się:

- *typy bazowe*: `int`, `bool` oraz `unit`,
- *typ pary*: τ_1 `*` τ_2,
- *typ list*: τ `list`.

Identyfikatory funkcji, podobnie jak zmienne zaczynają się literą lub podkreśleniem, po którym następuje ciąg znaków alfanumerycznych, podkreśleń i apostrofów. Identyfikatory funkcji będziemy oznaczać metazmiennymi f, f_1, f_2,...

Wyrażenia zawierają wszystkie konstrukcje wyrażeń z poprzedniego zadania. Dodatkowo poprawnym wyrażeniem jest:

- *aplikacja funkcji*: f e,
- *wyrażenie unit*: `()`,
- *para*: (`(`e_1`,` e_2`)`),
- *pierwsza projekcja*: `fst` e,
- *druga projekcja*: `snd` e,
- *list pusta*: `[]:` τ,
- *konstruktor listy niepustej (cons)*: e_1 `::` e_2,
- *dopasowanie wzorca dla listy*: `match` e `with []` `->` e_1 `|` x_1 `::` x_2 `->` e_2.

Zwróćmy uwagę, że lista pusta jest zawsze anotowana typem. Ułatwi to wyprowadzanie typów dla wyrażeń.

Aplikacja funkcji oraz obie projekcje wiążą silniej niż wszystkie operatory, natomiast cons wiąże silniej niż operatory porównania, ale słabiej niż operatory arytmetyczne. Dodatkowo parser ma zaimplementowany cukier syntaktyczny dla list: np. wyrażenie `[1, 2, 3]: int list` oznacza `1 :: 2 :: 3 :: []: int list`. W dopasowaniu wzorca pozwalamy również na występowanie znaku „`|`” od razu po słowie kluczowym `with`. Np. poprawnym wyrażeniem jest:

```
match l with
| []      -> 0
| x :: xs -> length xs + 1
```

Definicje funkcji mają postać `fun` f`(`x`:`τ_1`):`τ_2`=`e. Ciągi definicji funkcji będziemy oznaczać metazmienną F.

Programy składają się z (potencjalnie pustego) ciągu definicji funkcji, deklaracji zmiennych wejściowych oraz wyrażenia. Deklaracja zmiennych wejściowych jest opcjonalna i ma postać `input` x_1 ... x_n. Jeśli w programie występuje deklaracja zmiennych wejściowych lub definicja przynajmniej jednej funkcji, to definicje funkcji oraz deklaracja zmiennych wejściowych powinny być oddzielone od wyrażenia słowem kluczowym `in`. Poniżej przedstawiono przykładowy program liczący liczby Fibonacciego:

```
fun fib(n : int) : int =
  if n <= 1 then n
  else fib(n-1) + fib(n-2)

input n in fib(n)
```

Podobnie jak w poprzednim zadaniu, składnie abstrakcyjną będziemy reprezentować za pomocą algebraicznych typów danych. Szczegóły tej reprezentacji można znaleźć w pliku `AST.hs` zamieszczonym w serwisie SKOS.

System typów

Rozszerzyliśmy język o funkcje, w związku z tym relacja przypisująca typy do wyrażeń powinna też wiedzieć o funkcjach dostępnych w programie. Dlatego teraz będzie to relacja wiążąca ze sobą sekwencje definicji funkcji (F), środowisko (Γ), wyrażenie (e) oraz typ (τ), a stwierdzenie, że takie cztery elementy są w relacji będziemy zapisywać $F;\Gamma \vdash e :: \tau$. Jedyną zmienną reguł typowania z poprzedniego zadania jest obecność dodatkowego parametru F. Np. reguła dla wyrażenia `let` wygląda następująco:

$$\frac{F;\Gamma \vdash e_1 :: \tau_1 \qquad F;\Gamma[x \mapsto \tau_1] \vdash e_2 :: \tau_2}{F;\Gamma \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 :: \tau_2}.$$

Poniżej opiszemy reguły dla nowych konstrukcji w języku. Dodatkowy parametr F odgrywa ważną rolę jedynie w regule dla aplikacji funkcji:

$$\frac{(\texttt{fun } f\,(x:\tau_1):\tau_2\texttt{=}e') \in F \qquad F;\Gamma \vdash e :: \tau_1}{F;\Gamma \vdash f\ e :: \tau_2},$$

ponieważ z niego jesteśmy w stanie wyciągnąć typ argumentu (τ_1) oraz typ wartości (τ_2) funkcji. Wyrażenie `()` jest stałą typu `unit`:

$$\frac{}{F;\Gamma \vdash () :: \texttt{unit}}.$$

Dla operacji na parach oraz konstruktorów list mamy następujące reguły:

$$\frac{F;\Gamma \vdash e_1 :: \tau_1 \qquad F;\Gamma \vdash e_2 :: \tau_2}{F;\Gamma \vdash (e_1,\ e_2) :: \tau_1 \texttt{ * } \tau_2} \qquad \frac{F;\Gamma \vdash e :: \tau_1 \texttt{ * } \tau_2}{F;\Gamma \vdash \texttt{fst } e :: \tau_1} \qquad \frac{F;\Gamma \vdash e :: \tau_1 \texttt{ * } \tau_2}{F;\Gamma \vdash \texttt{snd } e :: \tau_2}$$

$$\frac{}{F;\Gamma \vdash [] : \tau \texttt{ list} :: \tau \texttt{ list}} \qquad \frac{F;\Gamma \vdash e_1 :: \tau \qquad F;\Gamma \vdash e_2 :: \tau \texttt{ list}}{F;\Gamma \vdash e_1 :: e_2 :: \tau \texttt{ list}}.$$

Zauważmy, że anotacja typowa przy liście pustej ułatwia implementację wyprowadzania typu: bez tej anotacji nie można jednoznacznie przypisać typu dla listy pustej, bez znajomości kontekstu w którym się ona znajduje. Ponadto, reguła wymaga, by ta anotacja typowa była postaci τ `list`. Wyrażenie `[]:int` jest poprawne składniowo, ale nie można mu przypisać typu.

Ostatnią regułą jest reguła dla dopasowania wzorca:

$$\frac{F;\Gamma \vdash e :: \tau_1 \texttt{ list} \qquad F;\Gamma \vdash e_1 :: \tau_2 \qquad F;\Gamma[x_1 \mapsto \tau_1][x_2 \mapsto \tau_1 \texttt{ list}] \vdash e_2 :: \tau_2}{F;\Gamma \vdash \texttt{match } e \texttt{ with [] -> } e_1 \texttt{ | } x_1 :: x_2 \texttt{ -> } e_2 :: \tau_2}.$$

Od całych programów wymagamy by parametry i ostateczna wartość były typu `int` oraz by wszystkie funkcje miały typ zgodny z ich anotacją. Wyrazimy to regułą:

$$\frac{\forall(\texttt{fun } f\,(x:\tau_1):\tau_2\texttt{=}e_f) \in F.\ F;[x \mapsto \tau_1] \vdash e_f :: \tau_2 \qquad F;[x_1 \mapsto \texttt{int}, \ldots, x_n \mapsto \texttt{int}] \vdash e :: \texttt{int}}{\vdash F \texttt{ input } x_1 \ldots x_2 \texttt{ in } e}.$$

Uwaga
W podanych regułach parametr F jest ciągiem definicji funkcji i jest używany tylko w regule dla aplikacji funkcji. Potrzebujemy go do tego by znając identyfikator funkcji wyznaczyć typy argumentu i wartości. Można więc traktować parametr F podobnie jak środowisko: jako funkcja częściowa z identyfikatorów funkcji w pary typów.

Semantyka

Wartościami w języku z poprzedniego zadania były tylko liczby i wartości boolowskie. Teraz pojawiły się nowe rodzaje wartości: wartość unit $\langle\rangle$, pary wartości (jeśli v_1 i v_2 są wartościami, to (v_1, v_2) też jest wartością) oraz listy wartości. Druga zmiana w stosunku do poprzedniego zadania jest podobna do tej, którą widzieliśmy dla systemu typów: aby zdefiniować semantykę wywołania funkcji, musimy znać definicje funkcji w programie. Zatem do relacji opisującej semantykę dodamy dodatkowy parametr F, który będzie ciągiem definicji funkcji. Poniżej przedstawimy reguły dla nowych konstrukcji w języku.

Dla wywołania funkcji najpierw liczymy jej argument, a potem interpretujemy ciało funkcji. Oznacza to, że nasz język jest gorliwy.

$$\frac{F;\rho \vdash e \Downarrow v_1 \qquad (\texttt{fun}\ f\,(x:\tau_1):\tau_2{=}e_f) \in F \qquad F;[x \mapsto v_1] \vdash e_f \Downarrow v_2}{F;\rho \vdash f\ e \Downarrow v_2}$$

Zauważmy, że ciało funkcji interpretujemy w środowisku, które definiuje tylko zmienną x — jest to jedyna zmienna widoczna w wyrażeniu e_f.
Wyrażenie `()` reprezentuje już wartość:

$$\frac{}{F;\rho \vdash () \Downarrow \langle\rangle}\ .$$

Dla operacji na parach oraz konstruktorów list mamy następujące reguły:

$$\frac{F;\rho \vdash e_1 \Downarrow v_1 \qquad F;\rho \vdash e_2 \Downarrow v_2}{F;\rho \vdash (e_1, e_2) \Downarrow (v_1, v_2)} \qquad \frac{F;\rho \vdash e \Downarrow (v_1, v_2)}{F;\rho \vdash \texttt{fst}\ e \Downarrow v_1} \qquad \frac{F;\rho \vdash e \Downarrow (v_1, v_2)}{F;\rho \vdash \texttt{snd}\ e \Downarrow v_2}$$

$$\frac{}{F;\rho \vdash [] : \tau \Downarrow []} \qquad \frac{F;\rho \vdash e_1 \Downarrow v_0 \qquad F;\rho \vdash e_2 \Downarrow [v_1, \ldots, v_n]}{F;\rho \vdash e_1 :: e_2 \Downarrow [v_0, v_1, \ldots, v_n]}\ .$$

Dla dopasowania wzorca, tak jak dla instrukcji warunkowej, wybieramy gałąź obliczeń w zależności od wartości pierwszego wyrażenia. Mamy więc dwie reguły:

$$\frac{F;\rho \vdash e \Downarrow [] \qquad F;\rho \vdash e_1 \Downarrow v}{F;\rho \vdash \texttt{match}\ e\ \texttt{with}\ []\ \texttt{->}\ e_1\ \texttt{|}\ x_1 :: x_2\ \texttt{->}\ e_2 \Downarrow v}$$

$$\frac{F;\rho \vdash e \Downarrow [v_0, v_1, \ldots, v_n] \qquad F;\rho[x_1 \mapsto v_1][x_2 \mapsto [v_1, \ldots, v_n]] \vdash e_2 \Downarrow v}{F;\rho \vdash \texttt{match}\ e\ \texttt{with}\ []\ \texttt{->}\ e_1\ \texttt{|}\ x_1 :: x_2\ \texttt{->}\ e_2 \Downarrow v}$$

Wartością całego programu F `input` $x_1, \ldots, x_m$ `in` e dla wartości zmiennych wejściowych $n_1, \ldots n_m$ będzie taka liczba n, że $F;[x_1 \mapsto n_1, \ldots, x_m \mapsto n_m] \vdash e \Downarrow n$.

Uwagi dotycząca implementacji w Haskellu
W podanych regułach informacja o funkcjach (F) jest ciągiem definicji funkcji, ale w implementacji interpretera wcale tak nie musi być. Może być to np.

1. funkcja częściowa z identyfikatorów funkcji w ich definicje:
   `Map FSym (FunctionDef p)`.
2. funkcja częściowa w elementy potrzebne do policzenia funkcji, czyli nazwę argumentu i ciało funkcji:
   `Map FSym (Var, Expr p)`.

3. funkcja częściowa w już zinterpretowane funkcje jako funkcje w Haskellu:
   `Map FSym (Value -> Maybe Value)`.

To ostanie podejście jest chyba najbardziej eleganckie, ale wymaga nieoczywistej rekursji wewnątrz definicji takiej funkcji częściowej.

Niektóre z wyrażeń mogą nie mieć wartości z dwóch powodów: występują w nich niezdefiniowane operacje (np. dzielenie przez zero) albo zawierają niekończące się obliczenia. Interpreter dla tych pierwszych powinien zwrócić błąd (`RuntimeError`), natomiast dla drugich może się zapętlić (w ogólności wykrywanie nieskończonych obliczeń jest nierozstrzygalne). A co jeśli w programie występują oba złe zachowania? Aby uniknąć niejednoznaczności, przyjmujemy kolejność obliczeń od lewej do prawej: jeśli do policzenia wartości wyrażenia e trzeba policzyć podwyrażenia e_1 oraz e_2, to najpierw liczymy to, które znajduje się bardziej po lewej stronie. Np. program

```
fun loop(u : unit) : int =
  loop ()
input x in
if x = 1 then loop() + 1/0 else 1/0 + loop()
```

dla danej wejściowej 1 powinien się zapętlić, a dla każdej pozostałej zgłosić błąd wykonania.

Zadanie, część 1.
Termin zgłaszania w serwisie SKOS: 29 maja 2017 6:00 AM CEST

Napisz zestaw testów dla sprawdzania typów i interpretowania przedstawionego języka. Należy posłużyć się następującym szablonem (znajdującym się również w serwisie SKOS):

```
-- Wymagamy, by moduł zawierał tylko bezpieczne funkcje
{-# LANGUAGE Safe #-}
-- Definiujemy moduł zawierający testy.
-- Należy zmienić nazwę modułu na {Imie}{Nazwisko}Tests gdzie za {Imie}
-- i {Nazwisko} należy podstawić odpowiednio swoje imię i nazwisko
-- zaczynające się wielką literą oraz bez znaków diakrytycznych.
module ImieNazwiskoTests(tests) where

-- Importujemy moduł zawierający typy danych potrzebne w zadaniu
import DataTypes

-- Lista testów do zadania
-- Należy uzupełnić jej definicję swoimi testami
tests :: [Test]
tests =
  [ Test "inc"      (SrcString "input x in x + 1") (Eval [42] (Value 43))
  , Test "undefVar" (SrcString "x")                TypeError
  ]
```

Znaczenia poszczególnych pól pojedynczego testu można znaleźć w pliku `DataTypes.hs` zamieszczonym w serwisie SKOS.

Wymogi formalne
Należy zgłosić pojedynczy plik o nazwie *imię_nazwisko*`_tests.tar.bz2` gdzie za *imię* i *nazwisko* należy podstawić odpowiednio swoje imię i nazwisko bez wielkich liter i znaków diakrytycznych. Nadesłany plik powinien być poprawnym skompresowanym archiwum `tar.bz2` nie zawierającym żadnego katalogu. W archiwum powinny znajdować się **tylko**:

- Plik źródłowy napisany w Haskellu o nazwie o nazwie *ImięNazwisko*`Tests.hs`, gdzie za *Imię* i *Nazwisko* należy podstawić odpowiednio swoje imię i nazwisko zaczynające się wielką literą oraz bez znaków diakrytycznych. Plik ten powinien być napisany w Haskellu przy użyciu podzbioru *SafeHaskell* i powinien definiować moduł eksportujący wartość `tests` typu `[Test]`.

- Wszystkie pliki źródłowe z programami w opisanym języku do których odwołują się testy (jeśli źródło programu podane jest za pomocą konstruktora `SrcFile`). Takie pliki powinny mieć rozszerzenie `.pp5`.

Rozwiązania nie spełniające wymogów formalnych nie będą oceniane!

Zadanie, część 2.
Termin zgłaszania w serwisie SKOS: 5 czerwca 2017 6:00 AM CEST

Napisz moduł eksportujący funkcje `typecheck` oraz `eval`, które odpowiednio sprawdzają typ oraz obliczają programy w opisanym języku. Należy posłużyć się następującym szablonem (znajdującym się również w serwisie SKOS):

```
-- Wymagamy, by moduł zawierał tylko bezpieczne funkcje
{-# LANGUAGE Safe #-}
-- Definiujemy moduł zawierający rozwiązanie.
-- Należy zmienić nazwę modułu na {Imie}{Nazwisko} gdzie za {Imie}
-- i {Nazwisko} należy podstawić odpowiednio swoje imię i nazwisko
-- zaczynające się wielką literą oraz bez znaków diakrytycznych.
module ImieNazwisko (typecheck, eval) where

-- Importujemy moduły z definicją języka oraz typami potrzebnymi w zadaniu
import AST
import DataTypes

-- Funkcja sprawdzająca typy
-- Dla wywołania typecheck fs vars e zakładamy, że zmienne występujące
-- w vars są już zdefiniowane i mają typ int, i oczekujemy by wyrażenia e
-- miało typ int
-- UWAGA: to nie jest jeszcze rozwiązanie; należy zmienić jej definicję.
typecheck :: [FunctionDef p] -> [Var] -> Expr p -> TypeCheckResult p
typecheck = undefined

-- Funkcja obliczająca wyrażenia
-- Dla wywołania eval fs input e przyjmujemy, że dla każdej pary (x, v)
-- znajdującej się w input, wartość zmiennej x wynosi v.
-- Możemy założyć, że definicje funckcji fs oraz wyrażenie e są dobrze
-- typowane, tzn. typecheck fs (map fst input) e = Ok
-- UWAGA: to nie jest jeszcze rozwiązanie; należy zmienić jej definicję.
eval :: [FunctionDef p] -> [(Var,Integer)] -> Expr p -> EvalResult
eval = undefined
```

Wymogi formalne
Należy zgłosić pojedynczy plik o nazwie *ImięNazwisko*`.hs` gdzie za *Imię* i *Nazwisko* należy podstawić odpowiednio swoje imię i nazwisko zaczynające się wielką literą oraz bez znaków diakrytycznych. Plik ten powinien być napisany w Haskellu przy użyciu podzbioru *SafeHaskell* i powinien definiować moduł eksportujący funkcje `typecheck` oraz `eval` tak jak opisano w załączonym szablonie. **Rozwiązania nie spełniające wymogów formalnych nie będą oceniane!**

Uwaga
W serwisie SKOS umieszczono plik `Prac5.hs` pozwalający uruchamiać napisane rozwiązanie na przygotowanych testach. Sposób jego uruchamiania znajduje się w komentarzu wewnątrz pliku.