Metody programowania 2017

Lista zadań na pracownię nr 6

Na tej liście zadań rozszerzymy język z poprzedniego zadania o funkcje wyższego rzędu oraz funkcje anonimowe. Tak jak wcześniej, omówimy tylko zmiany w stosunku do języka z poprzedniego zadania.

Składnia

Typy zawierają wszystkie konstrukcje z poprzedniego zadania. W nowym języku pozwalamy na funkcje wyższego rzędu (funkcje są wartościami), więc do gramatyki typów dodajemy jedną nową konstrukcję: typ funkcyjny τ_1 `->` τ_2. Przyjmujemy standardowe konwencje dotyczące łączności: strzałka łączy w prawo i wiąże słabiej niż wszystkie pozostałe konstrukcje w typach.

Identyfikatory funkcji oraz zmienne w tym języku nie są już rozdzielnymi przestrzeniami nazw. Teraz używamy zwykłych zmiennych do nazywania funkcji. Np. poniższy program nie jest poprawny (nie ma typu), choć był poprawny w poprzednim zadaniu:

```
fun f(x : int): int = x
input x in
  let f = x in f f
```

Wyrażenia zawierają wszystkie konstrukcje z poprzedniego zadania oraz jedną nową konstrukcję: lambda-abstrakcję `fn (`x `:` τ`) ->` e. Natomiast aplikacja funkcji ma teraz ogólniejszą postać: można zaaplikować dowolne wyrażenie do dowolnego wyrażenia (e_1 e_2). Standardowo, aplikacja łączy w lewo i wiąże silniej niż wszystkie operatory.

System typów

Z powodu, że nazwy funkcji traktowane są tak samo jak zmienne, to relacja typowania nie potrzebuje przyjmować sekwencji definicji funkcji F jako parametr — informację o typach funkcji będziemy trzymać w środowisku Γ. Przyjmujemy wszystkie reguły typowania z poprzedniego zadania (oprócz reguły dla aplikacji), tyle, że z pominięciem parametru F. Dodajemy następujące dwie nowe reguły dla aplikacji oraz lambda-abstrakcji.

$$\frac{\Gamma \vdash e_1 :: \tau_2 \texttt{ -> } \tau_1 \qquad \Gamma \vdash e_2 :: \tau_2}{\Gamma \vdash e_1\ e_2 :: \tau_1} \qquad \frac{\Gamma[x \mapsto \tau_1] \vdash e :: \tau_2}{\Gamma \vdash \texttt{fn (}x\texttt{:}\tau_1\texttt{) -> } e \texttt{ :: } \tau_1 \texttt{ -> } \tau_2}$$

Zwróćmy uwagę, że ciało lambda-abstrakcji ma do dyspozycji wszystkie zmienne lokalne widoczne w miejscu w którym została utworzona. Np. wyrażenie `fn(x:int)->fn(y:int)->x+y` jest poprawnym wyrażeniem typu `int->int->int`.

Reguła mówiąca o poprawności całych programów wygląda podobnie do tej z poprzedniego zadania. Jedyna istotna różnica polega na tym, że typy funkcji trzymane są teraz w środowisku Γ:

$$\frac{\Gamma[x_1 \mapsto \tau_1] \vdash e_1 :: \tau_1' \quad \ldots \quad \Gamma[x_n \mapsto \tau_n] \vdash e_n :: \tau_n' \qquad \Gamma[y_1 \mapsto \texttt{int}, \ldots, y_m \mapsto \texttt{int}] \vdash e :: \texttt{int}}{\vdash \texttt{fun } f_1\texttt{(}x_1\texttt{:}\tau_1\texttt{):}\tau_1'\texttt{=}e_1 \ \ldots \ \texttt{fun } f_n\texttt{(}x_n\texttt{:}\tau_n\texttt{):}\tau_n'\texttt{=}e_n \texttt{ input } y_1 \ldots y_m \texttt{ in } e}$$

gdzie $\Gamma = [f_1 \mapsto (\tau_1 \texttt{ -> } \tau_1'), \ldots, f_n \mapsto (\tau_n \texttt{ -> } \tau_n')]$.

Semantyka

Wszystkie reguły opisujące semantykę z poprzedniego zadania (za wyjątkiem reguły dla aplikacji) stosują się również w tym zadaniu, z tą różnicą, że podobnie jak dla systemu typów, nie potrzebujemy parametru F — informację o funkcjach będziemy trzymać w środowisku ρ. Dodatkowo w języku pojawia się nowy rodzaj wartości: funkcje. W semantyce rozróżnimy funkcje globalne (zdefiniowane słowem kluczowym `fun`) od anonimowych (zdefiniowanych słowem kluczowym `fn`). Jeśli w implementacji interpretera zareprezentujemy wartości funkcyjne jako funkcje Haskellowe (tak jak w uwadze do poprzedniego zadania), to takie rozróżnienie nie będzie potrzebne.

Funkcja anonimowa (np. `fn(y:int)->x+y`) jest już wartością, ale jej ciało (`x+y`) jest odroczonym obliczeniem. Do wykonania tego obliczenia potrzebujemy znać wartości zmiennych lokalnych. Wartość jednej ze zmiennych (`y`) dostaniemy jako parametr, ale wartości pozostałych powinny być takie, jak w miejscu utworzenia tej funkcji. Np. `let x=42 in fn(y:int)->x+y` oblicza się do funkcji, która dodaje 42, nawet w kontekście, gdzie zmienna `x` jest niezdefiniowana. Dlatego wartości odpowiadające funkcjom anonimowym będziemy reprezentowali jako *domknięcia*, czyli struktury zawierające nie tylko treść samej funkcji, ale też wartości zmiennych lokalnych w miejscu, gdzie została ona utworzona, czyli środowisko. Domknięcia będziemy zapisywali jako $[\rho; \texttt{fn}(x:\tau)\texttt{->}e]$.

Funkcje globalne są wzajemnie rekurencyjne, więc podczas wywołania takiej funkcji powinniśmy liczyć jej ciało w środowisku, w którym wszystkie funkcje globalne są zdefiniowane. Nie było to problemem w poprzednim zadaniu, bo cała relacja ewaluacji była parametryzowana dostępnymi definicjami funkcji, ale tym razem tego parametru nie mamy. Informację tę będziemy trzymać razem z funkcją. Wartość będącą funkcją globalną będziemy też reprezentowali jako pewną formę domknięcia: strukturę składającą się z definicji wszystkich funkcji globalnych oraz wyróżnionej definicji funkcji. Takie domknięcia będziemy zapisywali jako $[F; \texttt{fun}\ f\ (x:\tau_1):\tau_2\texttt{=}e]$. Dodatkowo zdefiniujmy operację tworzenia środowiska ρ_F z sekwencji definicji funkcji F:

$$\rho_F = [f_1 \mapsto [F; \texttt{fun}\ f_1\ (x_1:\tau_1):\tau_1'\texttt{=}e_1], \ldots, f_n \mapsto [F; \texttt{fun}\ f_n\ (x_n:\tau_n):\tau_n'\texttt{=}e_n]]$$

dla $F = \texttt{fun}\ (x_1:\tau_1):\tau_1'\texttt{=}e_1\ \ \ldots\ \ \texttt{fun}\ (x_n:\tau_n):\tau_n'\texttt{=}e_n$.

Mamy dwie reguły dla aplikacji oraz jedną dla lambda-abstrakcji:

$$\frac{\rho \vdash e_1 \Downarrow [\rho'; \texttt{fn}(x:\tau)\texttt{->}e'] \qquad \rho \vdash e_2 \Downarrow v \qquad \rho'[x \mapsto v] \vdash e' \Downarrow v'}{\rho \vdash e_1\ e_2 \Downarrow v'}$$

$$\frac{\rho \vdash e_1 \Downarrow [F; \texttt{fun}\ f\ (x:\tau_1):\tau_2\texttt{=}e'] \qquad \rho \vdash e_2 \Downarrow v \qquad \rho_F[x \mapsto v] \vdash e' \Downarrow v'}{\rho \vdash e_1\ e_2 \Downarrow v'}$$

$$\frac{}{\rho \vdash \texttt{fn}(x:\tau)\texttt{=}e \Downarrow [\rho; \texttt{fn}(x:\tau)\texttt{=}e]}$$

Zwróćmy uwagę, że w regułach dla aplikacji ciało funkcji jest liczone nie w bieżącym środowisku, ale w środowisku, które pochodzi z domknięcia. W przypadku lambda-abstrakcji jest to środowisko w którym została ona utworzona.

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 $\rho_F[x_1 \mapsto n_1, \ldots, x_m \mapsto n_m] \vdash e \Downarrow n$.

Zadanie, część 1.

Termin zgłaszania w serwisie SKOS: 16 czerwca 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 `.pp6`.

Rozwiązania nie spełniające wymogów formalnych nie będą oceniane!

Zadanie, część 2.
Termin zgłaszania w serwisie SKOS: 16 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 `Prac6.hs` pozwalający uruchamiać napisane rozwiązanie na przygotowanych testach. Sposób jego uruchamiania znajduje się w komentarzu wewnątrz pliku.