Assignment 5

CS 4400 / CS 5400 Programming Languages

General

Due: Wednesday, December 4, 11:59pm.

Instructions:

1. Submissions are handled via the Khoury Handin server: https://handins.ccs.neu.edu/
2. Download the assignment pack from https://vesely.io/teaching/CS4400f19/m/cw/05/Assignment5.zip.
3. You can choose to complete this assignment as a pair. If you work as a pair, submit as a pair. Completing an assignment with a partner but submitting individually is considered cheating.
4. Complete the information in `Assignment5.hs` and submit `Assignment5.hs`, `StlcExt.hs`, and `Types.hs`.
5. You are free to add top-level definitions, but add them to the bottom of each file, below the separator line.
6. Each top-level function must include a type signature, followed by one or more defining equations.
7. Make sure your file loads into GHCi or can be compiled by GHC without any errors.
8. Use the provided `SimpleTests` module to write your own tests in the `tests` function. If appropriate, feel free to use QuickCheck for tests.
9. If some parts of the assignment are unclear, ask. Please do not wait until just before the submission deadline.

Grade: To calculate your grade, we will take the following into account:

a) Quality of submission: Does your code compile without errors? Did you follow the above steps?
b) Correctness: How well does it implement the specification?
c) QA: How well did you test your code?

Simply Typed Lambda Calculus with Extensions

Purpose: The purpose of this assignment is to practice reading typing rules and transcribing them as a type checker. Another aspect is, again, to try out programming in such a bare-bones language.

In the lecture, we looked at typing rules for arithmetic expressions, conditionals and simply typed lambda expressions. This assignment is about completing the type-checker for the basic language and adding extensions.

The assignment pack contains the following modules:

- `Assignment5`: the meta-file, which runs all tests and should contain extra information
- `Syntax`: the abstract syntax definitions for the language
- `Eval`: a complete evaluator for the language
- `Types`: contains an incomplete type checker for the language – to be completed by you (exercises 1-3)
- `StlcExt`: contains examples and space for completing exercises 4-7

Here is the syntax of the base language, followed by the typing rules.

```haskell
data Expr = Val Value
          -- basic Simply Typed Lambda
          | Var Variable           -- variable
          | Lam Variable Type Expr -- lambda abstraction
          | App Expr Expr          -- application
          | Fix Expr               -- fixed point operator
          -- Familiar extensions
          | Let Variable Expr Expr -- simple let binding
          | Add Expr Expr          -- arithmetic expressions
          | Sub Expr Expr
          | Mul Expr Expr
          | And Expr Expr          -- boolean expressions
          | Not Expr
          | Leq Expr Expr          -- less than or equal predicate
          | If Expr Expr Expr      -- conditionals
```

The `Fix` operator is used as follows. Unlike in Assignment 3, it is a construct in the language, not an expression to be applied.

```haskell
Fix (Lam "sum" (TyArrow TyInt TyInt) (Lam "n" TyInt (
  If (Leq (Var "n") (num 0))
     (num 0)
     (Add (Var "n") (App (Var "sum") (Sub (Var "n") (num 1)))))))
```

Here are the typing rules:

```
-------------------------------     -----------------------------
 tenv |- Val (Bool b) : TyBool       tenv |- Val (Num n) : TyInt


  t <- get x tenv                        add x t1 tenv |- e : t2
------------------                  -----------------------------------
 tenv |- Var x : t                   tenv |- Lam x t1 e : TyArrow t1 t2


 tenv |- e1 : TyArrow t2 t1    e2 : t2'   t2 == t2'
----------------------------------------------------
             tenv |- App e1 e2 : t1


 tenv |- e : TyArrow t t'   t == t'
------------------------------------
          tenv |- Fix e : t


 tenv |- e1 : t1    add x t1 env |- e2 : t2
--------------------------------------------
        tenv |- Let x e1 e2 : t2


 tenv |- e1 : TyInt    tenv |- e2 : TyInt
------------------------------------------  where OP is one of Add, Sub, Mul
        tenv |- OP e1 e2 : TyInt


 tenv |- e1 : TyBool    tenv |- e2 : TyBool
--------------------------------------------
        tenv |- And e1 e2 : TyBool


  tenv |- e : TyBool
------------------------
 tenv |- Not e : TyBool


 tenv |- e1 : TyInt    tenv |- e2 : TyInt
------------------------------------------
        tenv |- Leq e1 e2 : TyBool
 tenv |- e1 : TyBool    tenv |- e2 : t    tenv |- e3 : t
---------------------------------------------------------
                  tenv |- If e1 e2 e3 : t
```

Tip: While developing the type-checker, for debugging I recommend using `trace` (http://hackage.haskell.org/package/base-4.12.0.0/docs/Debug-Trace.html#g:1) (from `Debug.Trace`, which is imported for you in the files), or using `error` to throw an exception instead of returning Nothing.

To make things easier and faster, many places where you have to fill in code are marked with an underscore _, which, in Haskell parlance, is a "typed hole". Unlike for `undefined`, the compiler (and GHCi) will complain about such missing code and will also tell you its type.

For exercises 5-7, take a look at the `"sum"` example above and also the factorial example in `StlcExt`.

Exercises

Exercise 1 In `Types.hs`:

Finish implementing all the rules above. They should be familiar from the lecture. Some rules have been pre-implemented for you. Write

Exercise 2 – Pairs New constructors:

```
          ...
          | Pair Expr Expr         -- pairs
          | Fst Expr               -- select the left element of a pair
          | Snd Expr               -- select the right element of a pair
```

In `Types.hs`:

Implement typing rules for the pair constructor `Pair` and the pair selectors `Fst` and `Snd`. For the meaning of these operations, take a look at the corresponding cases in `eval`.

```
 tenv |- e1 : t1   tenv |- e2 : t2
-----------------------------------
 tenv |- Pair e1 e2 : TyPair t1 t2


 tenv |- e : TyPair t1 t2            tenv |- e : TyPair t1 t2
--------------------------          --------------------------
 tenv |- Fst e : t1                    tenv |- Snd e : t2
```

A pair is constructed using `Pair`:

```
Pair (num 1) (bool True)
Pair (num 2) (Pair (bool False) (Add (num 4) (num 3)))
```

Write both positive (well-typed expressions) and negative (failing expressions) test cases for your implementation.

Exercise 3 – Lists New constructors:

```
          ...
          | Cons Expr Expr          -- lists:
          | Nil Type                -- empty list
          | IsNil Expr              -- is the list empty?
          | Head Expr               -- head of the list
          | Tail Expr               -- tail of the list
```

In `Types.hs`:

Implement typing rules for the list constructors `Cons` and `Nil`, together with the operations `Head`, `Tail`, and the predicate `IsNil`. Notice that the `Nil` list constructor carries its type. Working with lists this way might be familiar to you from Racket. Again, I recommend taking a look at the corresponding cases in `eval` to see how they are evaluated.

```
                                  tenv |- e1 : t    tenv |- e2 : TyList t
--------------------------       -----------------------------------------
 tenv |- Nil t : TyList t                 tenv |- Cons e1 e2 : TyList t


 tenv |- e : TyList t             tenv |- e : TyList t
--------------------------       ---------------------------
 tenv |- Head e : t               tenv |- Tail e : TyList t


 tenv |- e : TyList t
--------------------------
 tenv |- IsNil e : TyBool
```

Lists are constructed as follows:

```
Nil TyInt
Cons (num 1) (Cons (num 2) (Cons (num 3) (Nil TyInt)))
Cons (Lam "x" TyInt (Leq (Var "x") (num 0))) (Nil (TyArrow TyInt TyBool))
```

Exercise 4 In StlcExt.hs:

```
untypedButOk1 :: Expr
untypedButOk2 :: Expr
untypedButOk3 :: Expr
```

Provide three interesting examples of expressions that do not typecheck, but evaluate correctly.

Exercise 5 In StlcExt.hs:

Complete

```
swapExpr :: Expr
```

by defining a well-typed function which take a pair of a boolean and an integer and swaps them.

Evaluation example:

```
> eval empty (App swapExpr (Pair (bool False) (num 3)))
Just (VPair (Num 3) (Bool False))
```

Show that the function type-checks by filling in its expected type and writing a corresponding test case.

Exercise 6 In StlcExt.hs:

Complete

```
boolListLengthExpr :: Expr
```

by implementing a well-typed function that counts the number of elements in a given list of booleans.

Evaluation examples:

```
> eval empty (
  App boolListLengthExpr
      (Cons (bool True) (Cons (bool False) (Cons (bool False) (Nil TyBool)))))
Just (Num 3)
```

```
> eval empty (App boolListLengthExpr (Nil TyBool))
`Just (Num 0)`
```

Show that the function type-checks by filling in its expected type and writing a corresponding test case.

Exercise 7 (required for CS5400, recommended for CS4400) In `StlcExt.hs`:

Complete

```
zipInt :: Expr
```

by implementing a well-typed function which takes a list of numbers and a list of booleans, and combines them by returning a list of pairs, where the first element is taken from the first (numeric) list and the second element is taken from the second (boolean) list. The resulting list should be have the same length as the shorter list.

You are free to choose whether to implement it as a curried function (taking one list at a time), or a function that takes a pair of lists.

Evaluation examples (using the pair version):

```
> list1 = Cons (num 1) (Cons (num 2) (Cons (num 3) (Cons (num 4) (Nil TyInt))))
> list2 = Cons (bool True) (Cons (bool False) (Cons (bool False) (Nil TyBool)))
> eval empty (App zipIntExpr (Pair list1 list2))

Just (VCons (VPair (Num 1) (Bool True))
       (VCons (VPair (Num 2) (Bool False))
         (VCons (VPair (Num 3) (Bool False)) VNil)))
```

Show that the function type-checks by filling in its expected type and writing a corresponding test case.