Assignment 9

CS 4400 Programming Languages

Start early and come to us with questions.

Due: 11pm on Friday, December 4, 2020

Submission:

1. Submit the following files via https://handins.ccs.neu.edu/courses/119:

   - `Assignment09.hs`
   - `Eval.hs`
   - `Syntax.hs`
   - `Types.hs`
   - `TypeCheck.hs`

2. An initial version of `Assignment09.hs` is included in the pack. You should add your tests to `allTest` and make sure `main` works with the rest of your code, preserving the functionality.
3. Every top-level definition must include a purpose statement (for functions) and a type signature, followed by one or more defining equations.
4. Double-check that you have named everything as required and that functions required by this assignment have the correct type signatures.
5. Make sure your file loads into GHCi or can be compiled by GHC without any errors. **Your grade might be reduced by up to 50% if your code does not compile and run.**

Purpose: To implement a typed version of protoScheme

Overview

In this assignment, we implement a simple type-checker for protoScheme. The finished type checker will take the abstract syntax of a program and return its type, if it has one. Our type system is based on Simply Typed Lambda Calculus. In particular, this means that lambdas will need to have type annotations on their arguments. Additionally, using only simple types means that we will have to restrict ourselves to integers for arithmetic operations and comparisons. We will keep booleans as the result of comparisons and for if-expressions and conditionals.

Our language has the following types, as defined in `Types.hs`:

```
<BaseType> ::= Integer
             | Real
             | Boolean

<Type> ::= <BaseType>
         | (-> <Type>+)
         | (Pair-of <Type> <Type>)
```

Take a look into `Types.hs` to see how these are represented. Note that our arrow types have a list of one or more types, not just a single input type and a single output type. This generalization directly corresponds to lambdas taking zero or more arguments. The last type of an arrow type is the return type. For example,

```
(-> Integer Boolean (-> Boolean Boolean) (Pair-of Integer Boolean))
```

is the type of functions which take an integer, a boolean and a function from booleans to booleans, and returns a pair containing an integer and a boolean.

State of the Union

The previous assignment introduced functional values and consolidated the syntax of `protoScheme`. At this point your code base should cover the following features:

1. Let-bindings and a variable references expressions
2. Boolean expressions, if expressions, conditionals
3. Pair values and selectors
4. Type predicates for integers, reals, numbers, pairs, and booleans
5. Function values: user-defined using `lambda` and built-in primitive operations
6. A base library of primitive operations, including arithmetic, comparisons, etc.
7. Global function and variable definitions. Function definitions are syntactic sugar for a global definition with a lambda. Global functions are mutually recursive

Assignment Pack

The starter code pack for this assignment contains the following:

`Parser.hs` As before or minor adjustments.

`SExpression.hs` As before or minor adjustments.

`Maps.hs` As before or minor adjustments.

`SimpleTests.hs` & `SimpleTestsColor.hs` As before or minor adjustments.

`Result.hs` As before or minor adjustments.

`Types.hs` The abstract syntax of protoScheme types with basic operations.

`TypeCheck.hs` Template for implementing the protoScheme type-checker. It contains some example equations which you will have to adjust to your abstract syntax.

`Assignment09.hs` Main assignment file. The main function allows passing the following arguments to the program:

- `--type file` type-check the file and print its type
- `--eval file` evaluate the given file and print the value
- `--tests` run the provided tests

Example programs (`example`*n*`.pss`) Example programs that your implementation should be able to type-check and evaluate.

`doc/index.html` Documentation generated from the source files in the pack. Contains an index of all exported names. Might come in handy.

Questions

Simple Types for protoScheme expressions

1. Extend `lambda`'s arguments with types, á la Simply-Typed Lambda Calculus. This means replacing the lambda syntax with the following:

   ```
   <Signature> ::= ( <Variable> : <Type> )

   <Expr> ::= ...
            | (lambda (<Signature>*) <Expr>)
   ```

   Update all relevant functions dealing with expressions. In case it is not clear, an s-expression representation of a signature will fit the following pattern:

   ```
   S.List [ name, S.Symbol ":", typeSExpr ]
   ```

   The `Types` module contains functions that will be useful for you. Check the provided documentation.

2. Complete the definition of the function `typeOf :: TyEnv -> Expr -> Result Type` in `TypeCheck.hs`. Some examples have been included in the definition, but you might need to change the names of constructors and handle their arguments fully. The individual constructs should follow the STLC examples given in lecture. You will need to generalize from the single

argument of the examples to multiple arguments we use in protoScheme. In addition to the rules presented in the lecture, the following should hold for `cond`.

The condition of each clause in a `cond` should have a boolean type. The expression parts of all clauses should have the same type. When this is satisfied, the result type of a `cond` expression should be the type of the expression parts of each clause. For example:

```
(cond ((> x 10) (+ x 1))
      ((and #t (= x 1)) x)
      (else 93))
```

should be well-typed (provided `x`'s type is `Integer`) and its result type should be `Integer`. On the other hand,

```
(cond ((> x 10) (and #t f))
      ((< x 10) (+ 1 2)))
```

should fail to type-check, due to the one clause returning a boolean, and the other returning an integer. Expressing the above as an inference rule, we get the following:

$$\frac{tenv \vdash e_{b1} : \texttt{Boolean} \quad tenv \vdash e_1 : t \quad \ldots \quad tenv \vdash e_{bn} : \texttt{Boolean} \quad tenv \vdash e_n : t}{tenv \vdash \texttt{(cond (}e_{b1}\ e_1\texttt{)} \ \ldots \ \texttt{(}e_{bn}\ e_n\texttt{))} : t}$$

The type of pairs, `Pair-of` is assigned to an expression returning pair values. Expression `(pair e1 e2)` has the type `(Pair-of ty1 ty2)` if `e1` has the type `ty1` and `e2` has the type `ty2`.

3. You will need to handle the type of pre-defined library operations. Our type language is quite limited and doesn't allow much flexibility, so we will assume most operations operate on `Integer`s whenever there is a choice between multiple valid types. This means:

   - `+`, `-`, `*`, `/` should have the type `(-> Integer Integer Integer)`
   - `<`, `>`, `<=`, `>=`, `=` should have the type `(-> Integer Integer Boolean)`
   - `not` should have the type `(-> Boolean Boolean)`

   The examples included with the pack respect these types.

   I suggest one of two ways of dealing with types of predefined operations:

   (a) Adding the type directly to the constructor or primitive operations in the syntax.

   (b) Not modifying the prim-ops themselves, but defining a base type environment containing the types of library operations.

   I personally prefer the second option, but you might find it easier to use (a).

Type-checking Definitions

4. Global definitions in a program will now come with type-signatures, similar to Haskell. Except, in protoScheme, signatures are obligatory. This means that every definition has to be *preceded* by a signature. Here is an example program:

```
(x : Integer)
(define x 32)

(pow : (-> Integer Integer Integer))
(defun pow (n m)
  (if (<= m 0)
      1
      (* n (pow n (- m 1)))))

(pow 2 x)
```

   Assume that in the list of s-expressions representing a program, a signature is always followed by a `defun` or `define` and, conversely, every `defun` or `define` is preceded by a signature. You will need to use this signature for two purposes:

   (a) When translating `defun` into `define`, the signature will provide you with the types for the lambda's arguments.

   (b) When type-checking a global definition, the signatures will provide you with the types of all available globals that the definition might use. This will allow type-checking mutually recursive functions.

   A definition whose actual type does not match their signature should fail. For example, the following should fail to type check:

```
(x : Boolean)
(define x 12)

(f : (-> Boolean Integer))
(defun f (x) (+ x 1))

(g : (-> Integer Boolean))
(defun g (x) (+ x 1))
```

   You will need to do a bit of parallel list processing for this question.

5. Finally, ensure the function `typeOfProgramSExpr :: [S.Expr] -> Result S.Expr` works correctly. It is used by `main` in `Assignment09.hs`, similarly to `runProgram`. Your evaluator should still work and be able to evaluate the examples included with this assignment.

You do not need to type-check type predicates (`integer?`, `boolean?`, etc.) To do this, we would need to introduce the concept of subtyping and a "super-type" (`Any`) as the type containing all types.