Assignment 10

CS 4400 Programming Languages

Start early and come to us with questions.

Due: 11pm on Tuesday, December 15, 2020

Submission:

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

   - `Assignment10.hs`
   - `Eval.hs`
   - `Syntax.hs`
   - `Types.hs`
   - `TypeCheck.hs`

   Note that there will be two assignments on Handins: one for the regular part and one for the extra credit portion. Submit the regular part before updating or submitting the extra credit work.

2. An initial version of `Assignment10.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 type inference algorithm for typed protoScheme.

Assignment Pack

The starter code pack for this assignment contains the files below. The provided modules contain several useful functions for this assignment, so do check the documentation.

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

`Types.hs` The abstract syntax of protoScheme types with assorted useful operations on types. For this assignment, the type language is extended with type variables for the main portion of this assignment. Type schemes and infrastructure for the extra credit portion is also included. This file is to be submitted with the assignment. You can add functionality to the module, but you cannot remove or modify the functions and types already exported by this module, unless you check with us beforehand.

`Gensym.hs` Fresh variable generation. You will need to use this function when generating new type variables during constraint collection and possibly for other purposes. It is used as in the example lecture mentioned below. Note, that the code in this file is considered *unsafe* because it uses an escape hatch from the IO monad. For most uses in this assignment, it should work ok. Let me know if you run into trouble and have a good reason for suspecting it is caused by this module.

`Assignment10.hs` Main assignment file, as before. 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. For the main portion, you should make sure that examples 1-9 work. For the extra-credit portion, look at example 10 and higher (if applicable).

`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.

Questions

1. Following the example code from lecture on 12/01, implement type inference for protoScheme expressions. This means you will need to modify the type checker to produce a preliminary type together with a set of constraints. Use the function `unify` on the generated constraints to produce a substitution (list of mappings from type variables to types). To get the *principal type* of the expression you will need to apply this substitution on the preliminary type. The user should no longer be required to annotate lambda arguments with types.

2. Signatures for global definitions are still required and you will need to check that the inferred type corresponds to the given signature. Note that the type language now allows type variables in signatures. A signature can provide a more specific type than the inferred type, but not vice-versa. For example, the following is ok:

   ```
   (id : (-> Integer Integer))
   (defun id (x) x)
   ```

   But the following should fail to type check:

```
(f : (-> A A))
(defun f (x) (+ x 1))
```

3. Introduce lists to protoScheme. Lists are formed using constructors:

   - `nil` – constructs an empty list
   - `cons` – constructs a list from an element and a list

   Lists can be distinguished using predicates:

   - `list?` – returns true if the argument evaluates to a list
   - `cons?` – returns true if the argument evaluates to non-empty list
   - `nil?` – returns true if the argument evaluates to an empty list

   Finally, lists can be taken apart using destructors:

   - `head` – returns the first element of a non-empty list, fails otherwise
   - `tail` – returns all but the first element of a non-empty list and fails otherwise

```
<Expr> ::= ...
         | (cons <Expr> <Expr>)
         | nil
         | (list? <Expr>)
         | (cons? <Expr>)
         | (nil? <Expr>)
         | (head <Expr>)
         | (tail <Expr>)
```

   The type language is extended correspondingly:[1]

```
<Type> ::= ...
         | (List-of <Type>)
```

   The typing of lists is governed by the following rules:

   - `nil` has the type `(List-of A)` where `A` is some type
   - `(cons e1 e2)` has the type `(List-of A)` only if `e1` type-checks to `A` and `e2` type-checks to `(List-of A)`
   - `(list? e)`, `(cons? e)` and `(nil? e)` have the type `Boolean` if `e` type-checks to any type
   - `(head e)` has the type `A` only if `e` type-checks to `(List-of A)`
   - `(tail e)` has the type `(List-of A)` only if `e` type-checks to `(List-of A)`

   Both the type checker and the evaluator should be updated to handle list expressions. List values should be pretty-printed as a plain `S.List`, e.g., parsing and evaluating the following

```
(tail (cons 1 (cons 2 (cons 3 (cons 4 nil)))))
```

   should, after converting back to an s-expression, yield

```
(2 3 4)
```

   The pack contains several list-processing examples.

[1] `Types.hs` has already been extended with this new type. See the provided module documentation.

Extra credit work

This part contains work for extra credit. It will be worth approximately 1/3rd of the assignment grade, meaning that you can gain about 2% (possibly more) of the overall course grade by completing this part. It is to be submitted as

4. Implement so called let-polymorphism for protoScheme. The following description paraphrases TAPL[2], p. 333-334. This applies to type-checking a let-construct `(let (x e1) e2)` or a global definition `(define x e1) e2`[3].

    1. *Preliminary type + constraints.* We use `typeOf` to calculate a preliminary type and a set of associated constraints for the bound expression `e1`.

    2. *Principal type.* We use unification to find a substitution for the constraints generated above and apply the substitution to the preliminary type to obtain the *principal type* `ty` of the bound expression. We also update the type environment using this substitution.

    3. *Generalization.* We *generalize* any free type variables that remain in the principal type to obtain a *type scheme*. That means if `A1`, `...`, `An` are free type variables in the type T, we write `(All (A1 ... An) ty)` for the principal type scheme of `t1`.

       One caveat here is that we need to be careful not to generalize variables which are also mentioned in the current typing environment, since these correspond to real constraints between the type and its context.

    4. *Typing environment.* We extend the typing environment with a binding from `x` to the type scheme `(All (A1 ... An) ty)`, and use it to type-check the body `e2`.

       Consequence: the typing environment now associates variables with a *type scheme*, not just a type. Any ordinary type can be represented as a scheme with no variables, e.g., `(All () Integer)`.

    5. *Instantiation.* In the variable case of `typeOf`, we look up the corresponding type scheme `All (A1 ... An) ty`. We generate fresh type variables to *instantiate* the type scheme, which we will return as the type of the given variable.

5. Complete a base typing environment for all built-in operations, writing polymorphic types (type schemes) for operations, such as equality `=`, predicates, so that they can be type-checked with arbitrary input types. To get full credit for this question, (re)implement `pair`, `left`, `right` as built-ins (possibly reusing code from A8). Add `cons`, `nil`, `head`, `tail`, `list?`, `cons?`, and `nil?` to the library of built-ins in Eval, as well as the typing environment for type-checking.

For this part, submit a version of `Assignment10.zip` under **Assignment 10 (extra credit)** on Handins. You won't be able to submit before you submit the regular part of the assignment.

Note: You are not expected to be able to complete this part based on just this description. If you choose to attempt this extra credit, let me know and we can chat about the concepts in more detail. Before you do, make sure you are on track to completing most of the main part of this assignment. I will prioritize helping students working on the regular questions.

[2] Pierce, Benjamin C. *Types and Programming Languages*. The MIT Press, 2002. Available through our library.

[3] This is a slight abuse of notation since `e2` represents the rest of the program, not just the final expression. However, the principles are mostly the same.