PROGRAMMING LANGUAGE CONCEPTS (COMP 2212)

SEMESTER TWO, 2019

Home

exercise sheet four: writing a type checker for the λtoy language

The aim of this exercise class is for you to discover how to write a Type Checker for the Toy Language.

The demonstrators in the lab will help answer any questions you have about the tutorials you are reading or if you have difficulty understanding the solutions to the exercises.

We will build on your solutions from Sheet 3. What? Wait? What do you mean you haven't done that Sheet yet? No worries, you can use our pre-prepared solutions instead if you prefer Sheet 3 Solutions (zip).

Task One

Make sure you have an executable that allows you to interpret λToy programs. If you are using our provided solutions you can either

write λToy programs as text files and call `Toy` with the filename as argument, or use the interactive version `Toyi`.

Run the interpreter on a few small example test programs. You may want to keep a file containing the test programs somewhere handy. You should include both well-typed and ill-typed programs. Does the interpreter behave as expected?

Task Two

We are going to begin to implement a type checker for the language now. First, create a new Haskell module called "ToyTypes.hs" and import your parser module (ours is called `ToyGrammar`). This should import the data type(s) of abstract syntax trees for your language (ours are called `Expr` and `ToyType`).

Remind yourself of the type checking rules for λToy by referring to the lecture notes: they are on the final slide of Lecture 8. You will notice that the rules make use of a type environment, that is a mapping from variables to types. Define a type abbreviation to represent type environments called `TypeEnvironment` and write functions for looking up a variable in a type environment and adding a new binding in to a type

environment.

Finally, similar to `ToyEval.hs`, write a function `unparseType :: ToyType -> String` that creates a string output version of a type for printing.

Task Three

Write a function `typeOf :: TypeEnvironment -> Expr -> ToyType` that implements the type checking rules for the language. The function should be defined by pattern matching over the different `Expr` constructors. There is a type rule for each case. In each case, read the rule from bottom up, think about what the return type is, and about what conditions that need checking. Recursively call `typeOf` on subexpressions where needed.

Task Four

Modify `Toy.hs` and `Toyi.hs` (or your equivalents) to invoke the type checker as part of interpretation. Run your interpreter on your test programs from Task One. You should include both well-typed and ill-typed examples.

Task Five

If that was all too easy, extend the entire

language to include a primitive `Unit` data type and support for pairing types. You should include at least the projection operation on pair values.

Page maintained by Pawel Sobocinski, Julian Rathke.