CS4201 – PLDI

Assignment: P1 – Lambda Calculus

Deadline: Monday 09 Oct 2023 *Credits*: 50% of coursework mark

MMS is the definitive source for deadline and credit details

You are expected to have read and understood all the information in this specification and any accompanying documents at least a week before the deadline. You must contact the lecturer regarding any queries well in advance of the deadline.

Aim / Learning objectives

The purpose of this assignment is four-fold:

- to increase your experience of the lambda calculus and type theory;
- to increase your understanding of the role type systems play in programming languages;
- to give you hands-on experience of implementing a programming language and type system; and
- to reinforce your understanding of types and semantics.

Requirements

You are required to use the Haskell programming language to implement a Lambda Calculus interpreter. The interpreter should allow lambda expressions (annotated with their types) to be defined with an evaluator that reduces those expressions to a normal form, using the rules of Alpha and Beta reduction. The interpreter should support a type inference and checking system. The interpreter comes supplied with a REPL (interpreter) system that you can use (and extend). One possible example of your interpreter might be the following:

```
ST> assume  (n :: *)
ST> assume (z :: n)
ST> assume (s :: n -> n)
ST> let id = z :: n
id :: n
ST> let zero = (\f x -> x) :: (n -> n) -> (n -> n)
zero :: n -> n -> n -> n
ST> let n0 = zero s z
n0 :: n
ST> n0
z :: n
```

In this output, note that the most general type of `n0` is inferred automatically by the system and displayed to the user. The evaluation of `n0` results in the normal form `z` with type `n`. The evaluator must be implemented using alpha and beta reduction as seen in the lecture slides.

Your implementation must be implemented **using only standard Haskell**. **You should not make use of any standard Haskell extensions or additional Hackage libraries.**

Your initial implementation can make use of the existing wrapper functions which map neatly to the different components of the system: `eval` for evaluation, `check` for type checking, `print` for pretty printing, etc. Most wrapper functions come with a **checkable** mode and an **inferable** mode. You need to extend these as appropriate with the correct functionality. You may need to implement additional data types and functions to capture the full functionality of the lambda calculus and its type system. Feel free to declare data types and functions as you need them.

Solutions should be simple, clean and elegant. Do not get distracted with implementing large and over-engineered solutions.

Getting started

Starter code can be found here:

https://staffres.cs.st-andrews.ac.uk/CS4201/StudRes/Coursework/P1-lambdacalc/lambdacalc/

Copy it to your practical folder and run the REPL with "`ghci Lambda/Main.hs`". When the `ghci` prompt Loads, type "`main`" to start the REPL. You can then type "`:help`" for a list of commands. The file "prelude.st" gives some start functions. These can be typed in to the interpreter, or loaded with `:load prelude.st`.

Keep in mind that Haskell programs are hard to debug. Write small bits of code and check that they are correct before continuing. The recommended approach for this practical is to:

1) Write a working interpreter by making use of the library wrappers for the simply typed lambda calculus, including variables, abstractions and applications. The first step should be to implement a simple evaluator, using the rules from the lectures on alpha and beta reduction, to reduce an expression to a normal form in the REPL (remember that there are different types of possible normal forms).
2) The interpreter should include type checking and inference of expressions, as in the example on Page 1. Some terms in the language are annotated with a type: the type of the term must match exactly the type specified in the annotation. If not, an error message should be reported to the user. Some expressions can have their type inferred (see the example on Page 1). These will require a modification to the typing rules from the lecture so that, instead of checking, a type can be inferred.
3) Extend the simply typed lambda calculus to include some additional standard types (and their corresponding terms): tuples, Either and if statements, for example. You will also need to extend the parser, but please do not create elaborate mechanisms to represent the terms.

Once you have a well-tested, stable, correct and well-written implementation accompanied by a good report, you may wish to implement further functionality, such as simply unary types and their corresponding terms, such as List and Maybe, which may be parameterized by a polymorphic or monomorphic parameter. You will need to think about how to model recursion over inductive types. This might include modeling inductive elimination using fix points. Again, you will need to modify the parser here. The idea is to have fun, to explore and experiment with different types and how they are checked, inferred and evaluated.

Submission

You are required to submit an archive file to the P1 slot on MMS. Your report should be in the ***PDF*** format. It must explain the design and implementation of your interpreter and it **must document your approach to testing and debugging**. Your report should also explain how you perform type inference, checking and evaluation, and how your implementation corresponds to the derivation rules given in the lectures; if you use any additional typing or evaluation rules (including for inference) then you must also state and formalize these in the report, with an explanation of their behaviour.

Assessment

Marking will follow the guidelines given in the school student handbook (see link at the end of this document). Some specific descriptors for this assignment are given below:

Mark range	Descriptor
1 - 6	A submission that does not compile or run, or a trivial implementation.
7 - 10	A solution which implements some of the requirements, such as allowing the user to form simple lambda calculus terms, but which is too unstable and buggy for real use.
11 - 13	A working implementation which is based on standard evaluation using alpha and beta reduction and type checking. Buggy and poorly tested submissions, and poorly written reports may also cause a submission to fall within this band.
14 - 16	A well-written implementation which also includes type inference and additional standard types. The implementation should be well-tested, well-written, well commented and stable, accompanied by a good report and additional examples.
17 - 18	An excellent implementation which goes beyond the original specification by defining additional unary types with examples and is accompanied by an excellent report. An example might be a complete, well-tested implementation of the simply typed calculus, with additional standard types, unary types and further. Practicals in the higher bracket should also include examples of e.g. List types, complete with examples.
19 – 20	An exceptional implementation showing independent reading and research, several extensions and an excellent quality of code, accompanied by an exceptional, insightful report. Submissions in this range would be expected to use a very large number of types, and implement complex behaviour such as eliminators, as well as show independent research and novel ideas.

Policies and Guidelines

Marking
See the standard mark descriptors in the School Student Handbook:
http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors

Lateness penalty
The standard penalty for late submission applies (Scheme A: 1 mark per 24 hour period, or part thereof):
http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

Good academic practice
The University policy on Good Academic Practice applies:
https://www.st-andrews.ac.uk/students/rules/academicpractice/