RecCalc: A Simply Typed Functional Programming Language
CSCI-3300 (Fall 2014) Project
Groups of 1 or 2
Total Points: 200 (215 possible)

1 Introduction

To fully understand the theory of programming languages (PLs) it is important to understand the design aspect of PLs, but it is equally important to understand how to implement them. **The goal of this project is to implement – in Haskell – a simple but powerful programming language called RecCalc.** The RecCalc PL is based off of a simply typed λ-calculus called Gödel's system T, which will be introduced at some point during the semester. It is very powerful, in fact, any amount of arithmetic can be carried out with in it.

2 RecCalc's Specification

The RecCalc syntax is defined as follows:

$$\begin{array}{rl} \text{(Types)} & T, A, B, C ::= \mathsf{Nat} \mid A \to B \\ \text{(Terms)} & t ::= x \mid 0 \mid \mathsf{suc}\, t \mid \mathsf{fun}\, x : T \Rightarrow t \mid \mathsf{app}\, t_1\, \mathsf{to}\, t_2 \mid \mathsf{rec}\, t\, \mathsf{with}\, t_1 \parallel t_2 \\ \text{(Term Contexts)} & \Gamma ::= \cdot \mid x : T \mid \Gamma_1, \Gamma_2 \end{array}$$

Substitution is denoted:

$$\mathsf{replace}\, x\, \mathsf{with}\, t\, \mathsf{in}\, t'$$

This syntax has been implemented in Syntax.hs. Compare the definition of the concrete syntax above with the abstract syntax defined in that file.

Now we define the type checking algorithm:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{T\_Var} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \mathsf{fun}\, x : T_1 \Rightarrow t : T_1 \to T_2}\ \text{T\_Lam}$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash \mathsf{app}\, t_1\, \mathsf{to}\, t_2 : T_2}\ \text{T\_App} \qquad \frac{}{\Gamma \vdash 0 : \mathsf{Nat}}\ \text{T\_Zero} \qquad \frac{\Gamma \vdash t : \mathsf{Nat}}{\Gamma \vdash \mathsf{suc}\, t : \mathsf{Nat}}\ \text{T\_Suc}$$

$$\frac{\begin{array}{l}\Gamma \vdash t : \mathsf{Nat} \\ \Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T \to \mathsf{Nat} \to T\end{array}}{\Gamma \vdash \mathsf{rec}\, t\, \mathsf{with}\, t_1 \parallel t_2 : T}\ \text{T\_Rec}$$

At this point we have seen the syntax and how terms are typed using the type checking rules next we define the evaluation rules:

$$\frac{}{\mathsf{app}\,(\mathsf{fun}\, x : T \Rightarrow t)\,\mathsf{to}\, t' \rightsquigarrow \mathsf{replace}\, x\, \mathsf{with}\, t'\, \mathsf{in}\, t}\ \text{E\_Beta} \qquad \frac{}{\mathsf{rec}\, 0\, \mathsf{with}\, t_1 \parallel t_2 \rightsquigarrow t_1}\ \text{E\_RecBase}$$

$$\frac{}{\mathsf{rec}\,(\mathsf{suc}\, t)\,\mathsf{with}\, t_1 \parallel t_2 \rightsquigarrow \mathsf{app}\,(\mathsf{app}\, t_2\, \mathsf{to}\,(\mathsf{rec}\, t\, \mathsf{with}\, t_1 \parallel t_2))\,\mathsf{to}\, t}\ \text{E\_RecStep}$$

$$\frac{t \rightsquigarrow t'}{\mathsf{fun}\, x : T \Rightarrow t \rightsquigarrow \mathsf{fun}\, x : T \Rightarrow t'}\ \text{E\_Lam} \qquad \frac{t_1 \rightsquigarrow t_1'}{\mathsf{app}\, t_1\, \mathsf{to}\, t_2 \rightsquigarrow \mathsf{app}\, t_1'\, \mathsf{to}\, t_2}\ \text{E\_App1}$$

$$\frac{t_2 \rightsquigarrow t_2'}{\mathsf{app}\, t_1\, \mathsf{to}\, t_2 \rightsquigarrow \mathsf{app}\, t_1\, \mathsf{to}\, t_2'}\ \text{E\_App2} \qquad \frac{t \rightsquigarrow t'}{\mathsf{suc}\, t \rightsquigarrow \mathsf{suc}\, t'}\ \text{E\_Suc}$$

$$\frac{t \rightsquigarrow t'}{\mathsf{rec}\, t\, \mathsf{with}\, t_1 \parallel t_2 \rightsquigarrow \mathsf{rec}\, t'\, \mathsf{with}\, t_1 \parallel t_2}\ \text{E\_Rec1} \qquad \frac{t_1 \rightsquigarrow t_1'}{\mathsf{rec}\, t\, \mathsf{with}\, t_1 \parallel t_2 \rightsquigarrow \mathsf{rec}\, t\, \mathsf{with}\, t_1' \parallel t_2}\ \text{E\_Rec2}$$

$$\frac{t_2 \rightsquigarrow t_2'}{\mathsf{rec}\, t\, \mathsf{with}\, t_1 \parallel t_2 \rightsquigarrow \mathsf{rec}\, t\, \mathsf{with}\, t_1 \parallel t_2'}\ \text{E\_Rec3}$$

Again, the goal is to implement this language which amounts to taking the above algorithm descriptions and translating them in Haskell. The syntax and parser are provided. The next section gives an overview of what is given to you as an initial code base. A few more functions that are provided to you are given in the description of each project task given below.

3 Overview of the Initial Code Base

Located in the course git repository is a directory called Project, and within it are several Haskell files:

Syntax.hs	The implementation of RecCalc's syntax.
Parser.hs	The RecCalc parser.
Pretty.hs	The file that will contain the pretty printer.
TypeCheck.hs	The file that will contain the type checker.
Eval.hs	The file that will contain the evaluator.
Main.hs	The file that will contain the main functions.

The syntax and parser for RecCalc have already been implemented and tested, these implementations are provided. Everyone should first read through the Syntax.hs file before attempting any of the project tasks. It is important to understand the definition of terms and types.

3.1 Syntax.hs

The syntax is implemented using a Haskell library called Unbound. It is not important for us to fully understand this library, but only understand what is necessary for this project. The unbound library provides Haskell with support for handling binding when implementing programming languages. There are three functions it provides that we will need to use:

`bind`	Takes a term and a name, and then returns a term with the name bound in it.
`unbind`	Takes a term with a name bound in it, and returns a pair of the name and term.
`aeq`	Tests two terms for α-equivalence.

This file also has several functions that will be needed to handle replacing variables in terms (substitution function). These are as follows:

`replace`	Takes a name and two terms, and replaces the name in the second term with the first.

More details of each of the above functions will be given in class.

3.2 Parser.hs

The goal of a parser is to take a string representation of a program and turn it into a program in the abstract syntax. Our abstract syntax is the language defined in Syntax.hs. This file provides four parsers:

`parseTerm`	Parses a string into a term.
`parseType`	Parses a string into a type.
`parseCtx`	Parses a string into a list of pairs of variable names and their corresponding types.

The initial code base complies right away. Each project task (given in the next section) are already present in their respective files. They have simply been left undefined. For example, in the file TypeCheck.hs there are the following lines:

```
typeCheck :: Fresh m => Ctx -> Term -> ErrorT String m Type
typeCheck ctx t = undefined
```

It will then be your job to fill in these functions with their respective definitions using the definitions given in Section 2. Thus, you will not be responsible for filling in the types of the functions, but only their definitions. In addition, every external library module needed for this project has already been imported.

4 Project Tasks

This section simply lists the required work.

4.1 Pretty Printer (40 pts.)

The syntax of terms (or programs) of RecCalc are represented by the datatype called `Term`, and the syntax of types are represented by the datatype called `Type` in the file Syntax.hs. Now this is the internal representation of terms and types – often called the abstract syntax – and not the representation we as users use to program in RecCalc – often called the concreate syntax (see Section 2).

A pretty printer takes a program or type in the abstract syntax and outputs its equivalent form in the concreate syntax. Lets consider a couple of examples:

Input	$\Rightarrow$	Output
`Fun Nat (bind (s2n "y") (Var (s2n "y")))`	$\Rightarrow$	$\mathsf{fun}\ y : \mathsf{Nat} \Rightarrow y$
`Arr Nat Nat`	$\Rightarrow$	$\mathsf{Nat} \to \mathsf{Nat}$

So your task – if you choose to accept it – is to fill in the following two functions:

```
prettyType :: Fresh m => Type -> m String
prettyType = undefined

prettyTerm :: Fresh m => Term -> m String
prettyTerm = undefined
```

The first takes a `Type` and needs to output its equivalent form in the concrete syntax as a string. Then the second does the same thing, but for terms (programs).

4.2 Type Checker (60 pts.)

This task simply asks you to implement the type construction algorithm outlined in Section 2. Thus, you must implement the following function:

```
typeCheck :: Fresh m => Ctx -> Term -> ErrorT String m Type
typeCheck = undefined
```

4.3 Evaluator (60 pts.)

Similarly, this task asks you to implement the evaluation algorithm given in Section 2. This requires you to implement the following:

```
eval :: Fresh m => Term -> m Term
eval = undefined
```

4.4 Type Preservation (20 pts.)

This task requires you to implement the following function:

```
typePres :: Term -> Bool
typePres = undefined
```

The type preservation function should take in a term as input, and then first construct the input terms type using `runTypeChecker` (provided), and then evaluate the input term using `runEval` (provided). Finally, construct the type of the term returned by the evaluator (`runEval`), and then output `True` if it is the same type as the original input, and output `False` otherwise.

Extra credit (15 pts.) Can you give an argument – does not have to be a formal proof – that the function `typePres` will always output `True` for any well-formed term? Put your argument in a comment above the `typePres` function in Eval.hs.

4.5 Main Functions (20 pts.)

There are three main functions you must implement:

```
mainCheck :: IO ()
mainCheck = undefined

mainEval :: IO ()
mainEval = undefined

mainPres :: IO ()
mainPres = undefined
```

The first, `mainCheck`, should do the following:

- Ask the user for a context as a string,
- Ask the user for a term as a string,
- Parse the input using `parseCtx` and `parseTerm`,
- Type check (using `runTypeChecker`) the input term using the input context, and
- Finally, output the resulting type using `runPrettyType` (provided).

Note that typing contexts are to be entered by the user as comma separated lists of variable names, that is, in the form `x : T1, y : T2, ..., z : Ti`, where `x`, `y`, and `z` are term variable names, and `T1` through `Ti` are a bunch of types in the concreate syntax.

The second, `mainEval`, should do the following:

- Ask the user for a term,
- Parse the input using `parseTerm`,
- Evaluate the input term using the evaluator (`runEval`), and
- Output the resulting term from the evaluator using `runPrettyTerm`.

Finally, the third, `mainPres`, should do the following:

- Ask the user for a term,
- Parse the input using `parseTerm`,
- Test the term for type preservation using `typePres`, and
- Output the result of `typePres` using `show`.

5 Groups

This project may be done in groups of **at most two** students. If two students wish to work together on this project, then **both students must attend in person** my office hours to declare that they are going to collaborate on this project. I will then take note of this.

Each student must turn in a solution, and that solution must be checked into their git repos. Furthermore, **each student must do their part of the project within their own repo**. So if student A and B are working together, and student A is going to write the evaluator, and student B is going to implement the pretty printer, then the evaluator must be written by student A in their repo, and a patch must be sent to student B, applied, and checked in. Likewise, for student B and their pretty printer. This is a more real-world approach to collaboration, and allows me to be able to keep track of all of what each student does more easily. **If this is not adhered to then points will be deducted**.