Teorija programskih jezikov 2017–18

Homework 2: Trees and Polymorphism (TMH)

Version 1 of April 26, 2018

In the lectures we considered LLMH: an extension of MH with λ-expressions, let-expressions, lists, maybe types and monomorphic type inference. This homework concerns a further extension with a type of binary trees. We call this extension TMH.

The lexer, parser and several auxiliary functions are already implemented for you. Your task is to complete the `evaluate` function that implements operational semantics, and the `inferType` function that performs type inference. Both functions have already been written to work for all expression constructs in LLMH. In Part A of the homework, you are asked to add the cases for the new expression constructs of TMH. This requires completing the two files `TMH_Evaluator.hs` and `TMH_TypeInference.hs`.

Part B concerns changing the type inference algorithm to implement polymorphic type inference, as discussed in Lecture 7. For this part, you need to further modify the file `TMH_TypeInference.hs`.

Logistics

The folder for `Homework2` (which is on the course webpage in zipped format) contains the following thirteen files.

`TMH_Evaluator.hzs`	`TMH_Lex.hs`	`TMH_TypeEnvironments.hs`	`code1_tmh.hs`
`TMH_ExpSubst.hs`	`TMH_Lex.hs`	`TMH_TypeInference.hs`	`code2_tmh.hs`
`TMH_ExpType.hs`	`TMH_Parse.hs`		`code3_tmh.hs`
`TMH_Interpreter.hs`	`TMH_Parse.y`		

You are asked to complete the two files `TMH_Evaluator.hs` and `TMH_TypeInference` and submit *these two completed files only* before the deadline of **12.00** on **Friday 11th May 2018**.

Each of Part A and Part B of the homework is worth a potential 10% of your course grade. You may either submit a solution to Part A alone, meaning that the homework

will count a potential 10% towards your course grade, or submit a solution to both Parts A and B, meaning that the homework will count a potential 20%. In either case, a submission should consist of just two files: `TMH_Evaluator.hs` and `TMH_TypeInference`.

When working on your homework, you may sometimes find it convenient to alter the other files for the purposes of testing and debugging. However, if you do this, take care! At the end, your submitted code for `TMH_Evaluator.hs` and `TMH_TypeInference` is required to work in combination with all other files in their original state.

Running TMH

A Haskell module `TMH_Interpreter` is provided for you. This can be used as follows.

```
$ ghci
GHCi, version *.*.*: http://www.haskell.org/ghc/
Prelude> :load TMH_Interpreter
*TMH_Interpreter> runTMH "code1_tmh.hs"

neg :: Bool -> Bool
fib :: Integer -> Integer
times :: Integer -> Integer -> Integer
divides :: Integer -> Integer -> Bool
twice :: (b2 -> b2) -> b2 -> b2
hd :: [b4] -> Maybe b4
tl :: [b3] -> Maybe [b3]
nth :: Integer -> [Integer] -> Integer
from :: Integer -> [Integer]
fltr :: (Integer -> Bool) -> [Integer] -> [Integer]
notdivides :: Integer -> Integer -> Bool
sieve :: [Integer] -> [Integer]
primes :: [Integer]

TMH>  primes
Type: [Integer]
Value: 2:3:5:7:11:13:17:19:23:29:31:37:41:43:47:53: ...
```

Note that this above dialogue uses a file `code1_llmh.hs`, which is provided for you.

The file `code1_llmh.hs` uses only the language LLMH, and so can be run at the start of the homework. Two other test files are provided. The file `code2_llmh.hs` contains

functions that work with the tree datatype, but which do not require polymorphic type inference. This can be used as a test file for Part A of the homework. The file `code3_llmh.hs` includes functions that do require polymorphic type inference. This can be used as a test file for Part B. These files do not do exhaustive testing. You are encouraged to write your own test files too.

PART A [10 Marks]

A1 Completing the evaluator [5 marks]

TMH extends LLMH with a built-in datatype of binary trees, declared using

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

Because the type is built in, there is no need for the programmer to write the above `data` declaration. Instead, the programming language TMH already contains the `Tree` type constructor, the constructors `Leaf` and `Node`, and the relevant `case` statement for trees. (See `code2_llmh.hs` for examples of functions using this syntax.)

The lexing rules of TMH are implemented in `TMH_Lex.x`, and the grammar is implemented in `TMH_Parse.y`, which again make use of Haskell's `Alex` lexer and `Happy` parser respectively. These files use the data type `Exp` for the abstract syntax of expressions, which is declared in the Haskell module `TMH_ExpType`. This module also contains a data type `Type` for the abstract syntax of types.

The module `TMH_Evaluator` currently implements an evaluator for the language LLMH, based on its operational semantics. This is implemented by the function

```
evaluate :: Env -> Exp -> Exp
```

which computes the expression (which is necessarily a value) that results from evaluating the expression given as the second argument in the environment given as the first argument. The definition of `evaluate` makes use of the functions (such as substitution) for manipulating expressions and their variables, which are defined in the module `TMH_ExpSubst`.

Your task in this part is to extend the definition of the `evaluate` function in the file `TMH_Evaluator.hs`, to include cases for the additional expression constructions of TMH, associated with the `Tree` type constructor. Your operational semantics should follow the lazy approach to data types, from Note 6–7.

Here are a few hints on how to proceed with this part. First, check the data type declaration for `Exp` in `TMH_ExpType` to see what the missing cases are. Second, look at

the rules in `TMH_Parse.y`, which specify how the abstract syntax is generated from the concrete syntax. This should help you know how to correctly manipulate the abstract syntax. Third, pay particular attention to the following subtle issues.

- What is the correct abstract syntax for an expression matching a pattern of the form `Node` y z w in a `case` expression?
- Note that substitutions of the form $e[y, z, w := e'_1, e'_2, e'_3]$ in the operational semantics are *parallel substitutions*. (Each of the substitutions $[y := e'_1]$, $[z := e'_2]$, $[w := e'_3]$ is performed simultaneously on the expression e. This is different from performing the three substitutions in sequence. Exercise: why?) A function `parExpsubst` that computes parallel substitutions is available in the module `TMH_ExpSubst`.

A2 Type inference [5 marks]

As mentioned above, the Haskell module `LLMH_ExpType` contains a data type `Type` for the abstract syntax of types. (For example, if `t` is an LLMH type, then `List t` is the abstract syntax for the type `[t]`.) The module `TMH_TypeEnvironments` provides various functions for dealing with type substitutions, and type environments, including a function `mgu` that finds most general unifiers.

The module `TMH_TypeInference` currently implements type inference for the language LLMH. This is implemented by the function

```
inferType :: TypeEnv -> Exp -> [String] -> Maybe (TypeSubst, Type, [String])
```

This takes, as input, a type environment Γ, an expression e and a list of type variables B. It returns as output `Just`(θ_0, τ_0, B_0), where (θ_0, τ_0) is as described in Sections 2–3 of Note 5, and B_0 is a list of type variables. The role of the lists of type variables is to keep track of all the type variables used so far in the process of type inference. The list B is the list of all type variables already in use when the `inferType` function is called. This list is used by the function that finds 'fresh' type variables, so that it knows which type variables to avoid. The list B_0, returned by the function, is an updated list of type variables that extends the original B with any additional type variables used in the process of inferring (θ_0, τ_0). In order to facilitate the step to Part B of the homework, the type inference algorithm in `TMH_TypeInference` works with variables of the form `b0`, `b1`, `b2`, ... In Part B, these will be the *monomorphic* type variables. Note that the `inferType` function returns `Nothing` if type inference fails.

Your task is to extend the definition of the `inferType` function in the file `TMH_TypeInference.hs`, to include cases for the additional expression constructions of TMH, associated with the `Tree` type constructor. For this, adapt the type inference algorithm from Note 6–7.

In writing your answer to this part, you may find it convenient to follow the style of programming that is used in the existing parts of the `inferType` function. For example,

```
inferType tenv (Var x) bs =
  do t <- lookup x tenv
     return ([], t, bs)
```

The function application `lookup x tenv` looks up the type assigned to expression variable `x` in type environment `tenv`. The result of this function application has type `Maybe Type`. If the variable `x` appears in the type environment `tenv`, then `lookup x tenv` will return a value `Just typ`, where `typ` is the type assigned to `x` by `tenv`. The notation `t <- lookup x tenv` is a convenient way of assigning the type `typ` to the variable `t`. This notation has to be used within the scope of a `do ... return ...` construction, which allows us to work with `Maybe` types without explicitly pattern matching against the `Just` and `Nothing` constructors. As long as `lookup x tenv` does not evaluate to `Nothing`, we can make sense of `t <- lookup x tenv` as an assignment to the variable `t`. The `return` command then has the effect of packaging up the triple `([], t, bs)` as `Just ([], t, bs)`. If `lookup x tenv` does evaluate to `Nothing` then we cannot make sense of `t <- lookup x tenv` as an assignment to `t`. Instead, the whole `do ... return ...` construction evaluates to `Nothing`. (This is an example of Haskell's *monadic style* of programming, which is an advanced feature of Haskell. However, it is not necessary to understand the monadic style in general, in order to be able to program with it in the particular form used in the implementation `inferType`.)

PART B [10 Marks]

B Polymorphic type inference

The goal of this part of the homework is to perform *polymorphic* type inference, as discussed in Lecture 7. For this you will need to modify the definition of the `inferType` and `inferProg` functions, in the module `TMH_TypeInference`.

In the definition of `inferType`, the two cases that need changing are those for an expression variable `Var x`, and for a let expression `Let(x, exp1, exp2)`. For these, you should follow the algorithm on the slides for Lecture 7.

In order to implement this algorithm, you it will be helpful to write code for certain auxiliary functions. For example, in the case for an expression variable `Var x`, it will be useful to have a function that replaces polymorphic type variables in a type with fresh monomorphic type variables. And in the case for let expressions, it will be useful to have a function that turns relevant monomorphic type variables in a type into polymorphic type variables. In order to help you write these functions, the module `TMH_TypeEnvironments` contains functions `typeVars` and `polyTypeVars`, that respectively return lists of the monomorphic and polymorphic type variables that occur in a type; and also functions `freshtvar` and `freshptvar`, that respectively find fresh monomorphic and polymorphic type variables. The convention adopted is that polymorphic type variables are of the form `a0`, `a1`, `a2`, ..., and monomorphic type variables are of the form `b0`, `b1`, `b2`, ...

The `inferProg` performs type inference for the expression environment

$$\begin{aligned} x_1 &= exp_1 \\ &\ldots \\ x_n &= exp_n \end{aligned}$$

obtained from a program in which variables $x_1, \ldots, x_n$ are declared in that order. The current monomorphic algorithm needs adapting to assign polymorphic types to $exp_1, \ldots, exp_n$. In order to do this, it might be helpful to think of the program declarations as implementing the same behaviour as if the variables $x_1, \ldots, x_n$ were declared in a sequence of nested let-expression:

$$\texttt{let } x_1 = exp_1 \texttt{ in let } x_2 = exp_2 \texttt{ in } \ldots \texttt{ let } x_1 = exp_n \texttt{ in } \langle pending \rangle$$

The goal is then to generate a polymorphic type environment for $x_1, \ldots, x_n$, so that if any expression *exp* involving $x_1, \ldots, x_n$ is given to the interpreter for evaluation, then type inference will be carried out for *exp*, as if *exp* appears in the $\langle pending \rangle$ position in the nested let expression above.

One subtle point about the approach described is that, in order to make sense, it is necessary that the program declaration for any variable x_i involves an expression exp_i whose free variables all occur amongst the variables $x_1, \ldots, x_i$. That is, exp_i gives a (potentially) recursive definition of x_i making use of only previously declared variables. This is a real restriction on programs. For example, it means that programs can not include two variables defined *mutually recursively* in terms of each other. In this homework, we will live with this restriction. Indeed, a function `checkVars` is used in the `TMH_Interpreter` module to enforce this requirement on programs at the static analysis stage. (If this restriction were dropped, type inference for programs would involve a considerably more complex algorithm.)