The University of Hong Kong

COMP3259: Principles of Programming Languages

Assignment 2

Tutor
Wan Qianyong `qywan@cs.hku.hk`

Instructor
Bruno Oliveira `bruno@cs.hku.hk`

Deadline: 23:59, 6 April 2025

Table of Contents

Note

Your solution must be submitted in a single zip file, named as *A2_XXX.zip*, with *XXX* replaced by your UID. Please submit your solution on Moodle before the deadline. Your code should be well-written (e.g. proper indentation, readable names, and type signatures). You can test your code using `stack test`. Please at least make sure your code compiles!

Please do this assignment on your own; if, for a small part of an exercise, you use something from the Internet or were advised by your classmate, please mark and attribute the source in a comment. Do not use publicly accessible code sharing websites for your assignment to avoid being suspected of plagiarism.

1 More Kinds of Data, Redux

Our tiny language has grown from having only Integers to supporting Boolean values. But you may still miss some compound data types you got used to in other languages, like *Arrays* and *Structs* (we'll call them *Records*.) So, let's go one step further!

We will add the following syntax:

- For Arrays, we use `[a, b, ...]` to represent its literal form, and use `!!` for indexing:

```
var arr = [19, 193, 10];
arr !! (1+1)
<- 10
```

- For Records, we denote it by `{ field1 = v1, field2 = v2, ... }`. To access a field, use the *projection* operator `.`:

```
var rec = { foo = 1, bar = 2 };
rec.foo
<- 1
```

To support these constructions, we need to extend the abstract syntax:

```
type Label = String

data Value = ...
           | RecordV [(Label, Value)] -- { l1 = v1, l2 = v2, ... }

data Type = ...
          | TArray Type -- [T]
          | TRecord [(Label, Type)] -- { l1: T1, l2: T2, ... }

data Exp = ...
         | Array [Exp] -- [e1, e2, ...]
         | Index Exp Exp -- e1 !! (e2)
         | Record [(Label, Exp)] -- { l1 = e1, l2 = e2, ... }
         | Proj Exp Label -- e.l
```

In this assignment, we assume a Record will always have disjoint labels for simplicity.

Question 1. (5 pts.) Your first task is to complete the `show` and `showExp` functions in the file `Declare.hs` to add a pretty printer for all the extensions above, according to the format given in the comments.

The parser is provided. Now, let's extend our evaluator.

Question 2. (10 pts.) Extend the `evaluate` (and `evaluate'`) function in the file `Interp.hs` to handle Records.

- For `Record`, evaluate each field to get a `RecordV`;
- For `Proj e l`, evaluate `e` and return the field corresponding to label `l`; throw an error if `e` does not evaluate to a `RecordV` or the field does not exist.

Now we have Records, but where are our Arrays? Well, they're already close at hand! If you are familiar with JavaScript, you'll probably agree that an Array is just a special kind of object with its labels being natural numbers.

In our language, we'll treat them as a syntactic sugar of Records. For example, the double-element Array `[true, false]` can be represented as:

```
{ "0" = true, "1" = false }
```

Question 3. (Program Translate, 10 pts.) Extend `evaluate` (and `evaluate'`) in the `Interp.hs` file to translate Arrays to Records.

- For `Array`, translate it into a `Record` and call `eval` (or `evaluate'`) on the result;
- When encountering `Index e1 e2`, evaluate `e2` to `v2` and wrap them into `Proj e1 v2`, then call `eval` (or `evaluate'`) on it; throw an error if `e2` does not evaluate to an `IntV`.

Hint: You may use the `show` function to convert an index (`Int`) to a `Label` (`String`).

Our interpreter can now support Records and Arrays, but we have not leveraged the power of the type system. So, the next task will be completing the type check.

Question 4. (15 pts.) Complete the definition of `tcheck` in the file `TypeCheck.hs`. The type inference rules are shown as follows:

$$\frac{\Gamma \vdash e_i : T_i \ \ \forall i \in 1 \cdots n}{\Gamma \vdash \{l_i = e_i \ ^{i \in 1 \cdots n}\} : \{l_i : T_i \ ^{i \in 1 \cdots n}\}} \text{ TRcd} \qquad \frac{\Gamma \vdash e : \{l_i : T_i \ ^{i \in 1 \cdots n}\}}{\Gamma \vdash e.l_i : T_i} \text{ TProj}$$

$$\frac{\Gamma \vdash e_i : T \ \ \forall i \in 1 \cdots n}{\Gamma \vdash [e_1, e_2, \cdots, e_n] : [T]} \text{ TArray} \qquad \frac{\Gamma \vdash e : [T] \qquad \Gamma \vdash i : \mathbb{Z}}{\Gamma \vdash e \ !! \ i : T} \text{ TIndex}$$

Note:

- For illegal conditions like non-existent projection labels, Array elements with different types, or non-Integer indices, etc., return `Nothing`;
- For an empty Array, you can treat its type as `[Int]`;
- Unfortunately, our type system cannot statically catch errors like Index Out of Bounds. A more powerful system supporting *Dependent Types* will be required to achieve this.

2 Recapping Top-Level Functions

So far, we have been playing with a small language in the tutorials that supports arithmetic and logic operations, variable declarations, and so on. However, the ability to define functions is still missing. In Lecture 7, you have learned how to define a top-level function. Once a function is defined, we can use it anywhere at any times without having to rewrite it. Functions provide an abstraction layer: it *encapsulates* many instructions into a single name, thus increasing the readability and code reuse of programs.

A function definition consists of a function name, a list of parameters, and a function body. As for the matter of implementation, the syntax of function definitions in our language is as follows:

```
function fname(p1: t1, p2: t2, ..., pn: tn) { body }
```

Note that each parameter name must be accompanied with a type `t`. The type annotations here will be used later in this assignment when we implement a type checker for this language.

The syntax of function calls is as follows:

```
fname(e1, e2, ..., en)
```

A program is a sequence of function definitions, followed by a main expression:

```
function_1
function_2
...
function_n
expression
```

Here is an example of a complete program:

```
function absolute(x: Int) { if (x > 0) x; else -x }
function max(x: Int, y: Int) { if (x > y) x; else y }
max(absolute(-5), 4)  // returns 5
```

The domain and range of defined functions are limited to non-function types, because these are the only types of expression. Such functions are called *first-order* functions, in contrast to *higher-order* functions, which allow functions as arguments and results of other functions (we will cover them in the future.)

In order to allow function definitions, we need to define a new datatype `Program`:

```
data Program = Program FunEnv Exp
```

Each program has a function environment, which holds all the function definitions:

```
type FunEnv = [(String, Function)]
```

The definition above is read as: a function environment consists of bindings between function names and definitions. Each function definition has a list of parameters and a body expression:

```
data Function = Function [(String, Type)] Exp
```

To be able to call functions, we need to extend the datatype `Exp`:

```
data Exp = ...
         | Call String [Exp]
```

If we represent the previous program using the abstract syntax, it will be:

```
Program [ ("absolute",
           Function [("x",TInt)]
                    (If (Bin GT
                             (Var "x")
                             (Lit (IntV 0)))
                        (Var "x")
                        (Unary Neg (Var "x"))))
        , ("max",
           Function [("x",TInt), ("y",TInt)]
                    (If (Bin GT
                             (Var "x")
                             (Var "y"))
                        (Var "x")
                        (Var "y")))
        ] (Call "max" [Call "absolute" [Lit (IntV (-5))], Lit (IntV 4)])
```

We've provided a pretty printer for you, so don't be scared!

2.1 Evaluate with Top-Level Functions

We need to extend our evaluator as well. We use a top-level execution function, which has been provided for you in `Interp.hs`:

```
execute :: Program -> Value
execute (Program funEnv main) = evaluate main [] funEnv
```

Compared to previous evaluators, the new environment-based evaluator takes one additional argument (i.e. the function environment):

```
evaluate :: Exp -> Env -> FunEnv -> Value
```

In the case for function calls, the evaluation performs the following steps:

1. Look up the function by name to get its parameters and body;
2. Evaluate the actual arguments to get a list of values;
3. Create a new environment by zipping together the parameter names with the actual argument values;
4. Evaluate the function body in the new environment.

For other expressions, you can follow the definition of `evaluate` function in previous assignments, and just pass the `FunEnv` parameter to the `evaluate` call.

Question 5. (10 pts.) Please implement the `evaluate` function in `Interp.hs` that supports function calls with the above evaluation steps.

An interesting thing to note is that our language supports *recursion* for free! For example, we can write a factorial function, which recursively calls itself in the function body.

```
function fact(x: Int) {
  if (x == 0) 1; else x * fact(x - 1)
}
fact(5)
```

If you implement your evaluation function in the correct way, the expression will eventually evaluate to 120, as expected.

3 Function Substitution

In Lecture 4, we've seen another style of implementing an evaluator: by *substitution*. Thus let's explore a mixed-style implementation of the evaluator, that is, we still use an environment to evaluate variables, but we substitute away all function calls.

The key insight is that adding top-level functions doesn't necessarily make our language more expressive. Our language with variable declarations is already expressive enough to handle function calls. All we need is to *transform each call site of some function to appropriate `Decl` expressions.* The benefit of this approach is that we don't need to change the core function of the evaluator in Tutorial 4 because all function calls disappear after substitution.

We start by introducing function substitution, written as:

$$e'[\![f \mapsto e[x]]\!]$$

Here, $e[x]$ denotes a function body where only one parameter x occurs (we only discuss single-parameter functions here, but you can easily generalize it to multiple-parameter functions.) The whole notation means that in expression e', we substitute the function name f with its function body $e[x]$. Function substitution is defined by induction on the structure of e' like ordinary substitution. Since f is only used in `Call` expressions, the only non-trivial case is:

$$(\texttt{Call}\ f\ e')[\![f \mapsto e[x]]\!] \quad = \quad \texttt{Decl}\ x\ (e'[\![f \mapsto e[x]]\!])\ e[x]$$

By function substitution, we replace the function call by a `Decl` expression that binds the parameter x to $e'[\![f \mapsto e[x]]\!]$, which expands any further calls to f within e'. With the parameter x declared, we can know the result of the function call from $e[x]$.

For example, if we want to substitute the function call `absolute(-5)`, the result after substitution will be:

```
var x = -5;
if (x > 0) x; else -x
```

Let's now turn to the implementation. In the file `Interp.hs`, there is a function called `fsubst` as follows:

```haskell
fsubst :: (String, Function) -> Exp -> Exp
fsubst (f, Function xs body) e = error "TODO"
```

It takes a function definition and an expression to be substituted, and returns another expression with no calls to `f`. What it does is just like ordinary substitution, except for function calls, where it compares the name of the callee function with `f`. If they match, then it returns an appropriate `Decl` expression as specified by the formula above, otherwise it recursively substitutes the sub-expressions.

Question 6. (15 pts.) Complete the definition of `fsubst` in the file `Interp.hs`.

Note: You are required to consider multiple-parameter functions here.

In the same file, you will see the definition of `evaluate'`. As you can see, we don't need to bother with function calls, as they're presumably eliminated by function substitution. We are still one step away from completion: we need to modify the top-level execution function:

```haskell
execute' :: Program -> Value
execute' (Program funEnv main) = error "TODO"
```

Before calling `evaluate'`, the new `execute'` needs to substitute away all the function definitions in the main expression.

Question 7. (5 pts.) Complete the definition of `execute'` in `Interp.hs`.

Hint: There is a subtlety: one function can call another function defined before it, so the order of function substitution in `funEnv` matters!

There are disadvantages to this approach: one is that, like every substitution-based evaluator, it makes the evaluation less efficient than its environment-based counterpart. Another disadvantage is that we lose the ability to write recursive functions, as the substitution will never terminate.

4 The Glorious Type Checker

The glorious type checker prevents us doing something erroneous, such as calling functions that do not exist, or passing the wrong arguments that are incompatible with the formal parameters.

To augment the type checker we have seen in the tutorials to support top-level functions, we need two steps:

1. type checking function definitions;
2. type checking function calls.

Let's first see how to type check function definitions. In order to do that, we have to define another data structure called function type environment to associate function names with their corresponding parameters and return types:

```
type TFunEnv = [(String, (TEnv, Type))]
```

After successfully type checking function definitions, we will get a `TFunEnv`, which is needed to type check function calls. So the type checker now takes one additional argument:

```
tcheck :: Exp -> TEnv -> TFunEnv -> Maybe Type
```

The function `checkFunEnv` is used to type check function definitions:

```
checkFunEnv :: FunEnv -> Maybe TFunEnv
checkFunEnv fds = checkFunEnv' fds []
```

For each function, we type check (by using `tcheck` as before) the function body in the old environments (i.e., the parameters and the function type environment). If the function body type checks, we will get a return type for that function, then we extend the function type environment by one element (the function name, the parameters and the return type). As a result, functions defined later can call the ones defined earlier.

Note that if any one of the function bodies fails to type check, `checkFunEnv` returns `Nothing`.

Question 8. (10 pts.) Finish the definition of `checkFunEnv'` in the file `TypeCheck.hs`.

As for function calls, we first look up the function name in the function type environment. If it is not found, we will get `Nothing`; otherwise, we will get a pair of corresponding parameters and the return type of the callee function. Then we check if the types of the arguments match the types of the parameters. If it's not the case, we will get `Nothing` as well; otherwise, we successfully get the return type.

Question 9. (10 pts.) Complete the definition of `tcheck (Call name args)`.

Now everything is ready to make the glorious type checker! We proceed by first type checking the function definitions (using `checkFunEnv`). If they type check, we continue with type checking the main expression (using `tcheck`). If it type checks, we will get `True`; otherwise, we will get `False`.

```haskell
checkProgram :: Program -> Bool
checkProgram (Program fds main) = error "TODO"
```

Question 10. (5 pts.) Finish the definition of `checkProgram` according to the explanation above.

Question 11. (5 pts.) Does the type checker support recursive function calls? If yes, please show an example with detailed type derivation; otherwise, try to explain why they are not supported and come up with a possible fix.

Feel free to give any thought you have; only the Yes/No part will be graded. Your answers can be written in any form (e.g. hand-written, typeset, or ASCII art). Please remember to pack a soft copy of your answer into the zip file you submit.

Finale: SNHello!

Congratulations if you have managed to finish the assignment!

The interpreter is fully fledged to some extent, but you may not know how to use it as an executable program. In the bundle, you can find a file named `Main.hs` under the `app` directory, inside which there is a `main` function. You don't need to fully understand the code. It essentially takes the command-line arguments, reads the specified file, does type checking, and finally evaluates the program in the file.

The executable file needs a name. We're happy to announce that our new-born language is called SNH, which is a recursive acronym for "SNH's Not Haskell". To compile SNH, run the following command in the terminal:

```
> stack build
```

Under the hood, GHC will compile the source file `Main.hs`, producing an object file `Main.o` and an interface file `Main.hi`, and then it will link the object file to the libraries that come with GHC to produce an executable. Of course, all of these are automatically managed by `stack`, so you won't see those intermediate files.

Now you can create a file, say `demo.txt`, inside which you can write a program in SNH, for example:

```
function absolute(x: Int) { if (x > 0) x; else -x }
function max(x: Int, y: Int) { if (x > y) x; else y }
max(absolute(-5), 4)
```

Save the file and then run the command:

```
> stack exec snh demo.txt
5
```

Hurray! (As a side note, if you want to run the executable `snh` from anywhere in your shell, run `stack install`, which will copy the executable to your local `bin` path.)