Exercise 1 Report

Andrew & Tristan

FV/SUBSTITUTION

Our definition of the free variable function was based on the definition provided in TAPL, with appropriate modifications to deal with If statements, Constants, and Primitive Operations. In the case of If statements and Primitive Operations, we simply take the union of free variables for all subterms, and constants are assumed to contain no free variables. Otherwise, a variable has free variables equal to itself, and the free variables of an abstraction consists of the free variables of the subterm minus the variable being abstracted. All this functionality is implemented using lists and the bind operator (»=).

For substitution, we base our implementation of the definition in the book. However, unlike the book, we did not implement variable renaming using alpha-equivalence in order to deal with cases of shadowing. We assume that our programs will not contain such issues. Otherwise, our definition of substitution matches that of the book.

Parser

For parsing, we have utilized the Parsec library. We first implement some primitives to parse identifiers, tokens, whitespace, comments, etc. After implementing some parsing primitives, implementing the BNF grammar provided is straightforward. In case of a parse error, we propagate the error upwards.

The parser uses the applicative notation to make it easy to read. In general for reading the parser here are the meanings:

```haskell
parser :: Parser a

parser = parser1 *> parser2
-- we ignore the parsed output of parser1, both must still succeed

parser = parser1 <* parser2
-- we ignore the parsed output of parser2, both must still succeed

parser = function <$> parser1 <*> parser2
-- create the parser that fmaps 'function' into the Parser Monad add 'ap's
-- it with the outputs of parser1 and parser2, both must succeed

parser = try parser1 <|> parser2
-- we can either read parser1 OR parser2, but parser1 will be attemped first
```

Type Checker

We expanded the type checker in two aspects. Starting off, we expanded the type checker to properly deal with Primitive application by expanding the pattern matching cases. Furthermore, we replaced the Maybe type with the Either type in the type declaration, and added informative type error messages using it.

We also added a new type constructor for `Type`: `TypeError String` so we call the type of the term that fails the type checker is still a type instance. That way we still have full control over what happens and we never need Haskell to throw an exception.

Semantics/Evaluator

Our only addition to the operational semantics of our lambda calculus has been implementing the semantics of a primitive operation application, which involves evaluating all the subterms, evaluating the operation, and then applying the evaluated operator onto the evaluated subterms. To evaluate a primitive application, we updated the operational semantics to contain

```
S.PrimApp op xs ->
  do
    xs' <- (sequence $ fmap eval1 xs)
    Just (S.primOpEval op xs')
```

The evaluator will still run if the program is a `TypeError String`. If the type error comes from a free variable in the program then the evaluator will evaluate until it runs into a free variable, which is nice for reducing the term a bit. If the type error is a mismatched type to function then Haskell will throw an exception, but we hope to handle this more effectively later. We might want to separate these cases so a `TypeError` only exists if the type is a known mismatch, and if there is a free variable then it might just assume the type is whatever it needs to be.

Main Function

Our main function runs each component implemented and displays information about it. We parse command line arguments, read the file named by then, and then runs the parser, type checker, and evaluator in order. For each component, we display the relevant output: entire program for the parser, type or type error for the type checker, and normal form of the program for the evaluator. At the moment, we do not stop evaluation in the case of a type error. This can be easily fixed, but for the moment it allows us to evaluate programs that do not terminate in a value.

Running It

The executable is `./dist-newstyle/build/x86_64-linux/ghc-9.0.2/adv-compilers-0.1.0.0/x/adv-compilers/build/`

run with `./dist-newstyle/build/x86_64-linux/ghc-9.0.2/adv-compilers-0.1.0.0/x/adv-compilers/build/adv-com`
`test_types`

or replace `test_types` with whatever file you want to run.

Examples

```
GIVEN PROGRAM: app(app(abs(f: ->(Int,Int). abs(x: Bool. app(f, x))), abs(y: Int. *(y, y))), 4)
Free Variables: []
Typechecker: Type Error: Expected a 'Int,' but given the 'Bool' x in: "app(f, x)"
Evaluator: 16
```

```
GIVEN PROGRAM: if if true then true else false fi then if true then if true
  then true else false fi else false fi else false fi
Free Variables: []
Typechecker: Bool
Evaluator: true
```