Advanced Compiler Construction (CS 491): Exercise 2

Tristan & Andrew

Running it

To get the executable run `cabal install` while in the directory, and that should put an executable in your `~/.cabal/bin` directory. Otherwise use `cabal repl` and use the function `ghci> parseFile "filepath"`.

Assignment Additions

There were a few additions to the core language. We added `Unit` and `Char` along with a few functions and a test, but these were basically trivial as they did not add anything to `eval1`.

Records

`Record`s are product types and have multiple named elements of specified types. The constructor for `Record`s is

$$\text{record}(l_1 = t_1, l_2 = t_2, ..., l_n = t_n) : \text{Record}(l_1 = t_1, l_2 = t_2, ..., l_n = t_n)$$

and the eliminator is

$$\text{project}(\text{record}(l_1 = t_1, l_2 = t_2, ..., l_n = t_n).l_j) \to t_j$$

We used the small step semantics from TAPL. The type checking here was simple, as the internal structure of the values were:

```
data Type =  ...
          |  TypeRecord     [(Label, Type)]
data Term =  ...
          |  Record      [(Label, Term)]
          |  Project     Term Label
```

These let us use `lookup` to easily find the type a term should be, and the term that project needs from label. Since `lookup` returns a `Maybe` we just wrote a function `lookupOrElse` which returned an `Either` or our error system.

Variants

`Variant`s are sum types and have one of multiple possible named elements of specified types. The constructor for a `Variant` is

$$\text{tag}(l = t \text{ as Variant}(l_1 : T_1, l_2 : T_2, ..., l_n : T_n)) : \text{Variant}(l_1 : T_1, l_2 : T_2, ..., l_n : T_n)$$

and the eliminator is

$$
\begin{array}{l}
\text{case } t \\
\text{of } l_1 = x_1 \Rightarrow t_1 \\
\mid \ldots \\
\mid l_n = x_n \Rightarrow t_n \\
\text{esac}
\end{array}
$$

The implementation of `Variant` was significantly more work than `Record`.

```
data Type =  ...
          |  TypeVariant    [(Label, Type)]
data Term =  ...
          |  Tag          Label Term Type
          |  Case         Term [(Label, Var, Term)]
```

`Tag` was not too difficult, but `Case` proved to require more thoughtfulness since it requires that we do a lot in the type checking:

1. The labels in the case body and in the term being cased against need to match.
2. All terms in the case body need to evaluate to the same type
   - Additionally those terms need to be placed in a new context because of the variable that is bound to each one, but not all of those terms.

Here is the code without the `where` clause:

```
{- lib/Typing.hs -}
S.Case t1 lvt -> case typing gamma t1 of    -- match Case term
        Right tau1@(S.TypeVariant lts) -> do -- t1 is a Variant
                 -- same labels
                if sort labelsBody == sort labelsFocus
                       then isSameType
                       else Left "errMsg"
            where ...
```

After the type checking everything else was easy, as it was also just right out of TAPL.

Variants and records have all type checking incorporated into our type error framework.

Let

Implementing the small-step semantics of let bindings was not too difficult. In the case that we are substituting a value into our statement, we simply perform the substitution. If we are binding a non-value term to the variable to substitute, we first take one step of evaluation for the term. This is giving by the following clauses:

```haskell
eval_small (S.Let x t1 t2)
  | S.isValue t1 = Right (S.subst x t1 t2)  -- pg 124: E-LetV
  | otherwise = do t1' <- eval_small t1; Right (S.Let x t1' t2)  -- pg 124: E-Let
```

Fix

Similarly, our definition for fix is taken straight out of TAPL. It's definition is given by the following clauses:

```haskell
eval_small (S.Fix f@(S.Abs x _ t2)) = Right (S.subst x (S.Fix f) t2)  -- pg 144: E-FixBeta
eval_small (S.Fix t1) = do t1' <- eval_small t1; Right (S.Fix t1')  -- pg 144: E-Fix
```

Introducing Testing

We are using the cabal testing framework. To run the test suite use `$ cabal run test` in the cabal directory. We are using the `HUnit` module to make the tests. We have it set up so we can test a variety of aspects of a file:

- Does the file parse?
- Does the file have a certain list of free variables?
- Does the file type check?
- Does the file evaluate to something? (i.e. making sure it does not diverge or runtime error)
- Does the file evaluate to a specified value?

We encompass these in a list of the form:

```haskell
-- test/Test.hs
data TestType = NoParseError
              | NoTypeError
              | SolutionIs String
              | FreeVars [String]
              | EvalsToSomething

testAnswers :: [(FilePath, TestType)]
testAnswers = ...
```

Big-Step Semantics

Our implementation of Big Step Semantics does not implement the entire lamda-language due to time limitations (as we realized, we did not actually implement this on the day of presentations). That being said, the big-step semantics have been implemented for constants, records, variables, abstractions, applications, and primitive operations. Our first step in implementing big-step was to introduce a closure environment for term types. Following this, we implemented an alternative function called eval_big in the OperationalSemantics file. In essence, we implement the following rules:

$$e \vdash c \Rightarrow c \text{ (} c \text{ is a constant)}$$

$$e \vdash x \Rightarrow e(x)$$

$$e \vdash (\lambda x.t_1) \Rightarrow (\lambda x.t_1)[e]$$

$$(e \vdash t_1\ t_2, e \vdash t_1 \Rightarrow (\lambda x.t_1)[e'], e \vdash t_2 \Rightarrow v_2, (e' : x \rightarrow v_2) \vdash t_2 \Rightarrow v) \Rightarrow v$$

$$(e \vdash t_1 \Rightarrow Tru, e \vdash t_2 \Rightarrow v, e \vdash \text{ If } t_1 \text{ then } t_2 \text{ else } t_3) \Rightarrow v$$

$$(e \vdash t_1 \Rightarrow Fls, e \vdash t_3 \Rightarrow v, e \vdash \text{ If } t_1 \text{ then } t_2 \text{ else } t_3) \Rightarrow v$$

In our code, this looks as follows:

```
eval_big :: S.Term -> S.Environment -> Either String S.Term
eval_big t@(S.Const x) _ = Right t
eval_big t@(S.Record _) _ = Right t
eval_big (S.Var x) e = S.lookupEnv x e
eval_big t@(S.Abs _ _ _) e = Right (S.Closure t e)
eval_big t@(S.App t1 t2) e = case (eval_big t1 e) of
  Right (S.Closure (S.Abs x _ t12) e_prime) -> do
      v2 <- eval_big t2 e
      v <- eval_big t12 (S.Bind (x,v2) e_prime)
      return v
  err@(Left _) -> err
  _             -> Left ("Error evaluating " ++ (show t) ++ " with environment " ++ (show e)
eval_big t@(S.If t1 t2 t3) e = do
      v1 <- eval_big t1 e
      case v1 of
        (S.Const S.Tru) -> do v2 <- eval_big t2 e; return v2
        (S.Const S.Fls) -> do v3 <- eval_big t3 e; return v3
        _               -> Left ("Error evaluating " ++ (show t) ++ " with env " ++ (show e
eval_big t@(S.PrimApp op xs) e = do
      xs_val <- mapM ((flip $ eval_big) e) xs
      return (S.primOpEval op xs_val)
eval_big t e = Left ("Error evaluating term " ++ (show t) ++ " with environment " ++ (show e
```

We will implement the rest of the functionality before submitting exercise 3.