Advanced Compiler Construction (CS 491): Exercise 3

Tristan & Andrew Work Distribution:

Tristan - Reduction Semantics / CC Machine / De Bruijn

Andrew - Recursive Types / Debugging Evaluation

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

In this assignment, we added reduction semantics, the CC machine, de Bruijn notation, and recursive types. Furthermore, we added a more in-depth debugging evaluation mechanism, which is akin to a "proof" of evaluation results.

Recursive Types

Recursive types are introduced using the mu operator, and are interacted with using the fold/unfold syntax. The trouble we were running into when presenting was not actually due to any faults in our implementation, but rather we had an incorrect handling of substitution within record/variant types. In the typing world, we added two new types:

```
TypeVar TypeVar
TypeMu TypeVar Type
```

Furthermore, in the term world, we add two new terms:

```
Fold Type Term
Unfold Type Term
```

According to TAPL, we need to add two new typing rules to our core language:

```
S.Fold u t1 -- pg 276: T-Fld
   | S.TypeMu chi tau1 <- u -> enforceType t1 ((chi |-> u) tau1) gamma >> return u
   | otherwise -> Left $ "folding without mu operator in " ++ show t
S.Unfold u t1 -- pg 276: T-Unfld
   | S.TypeMu chi tau1 <- u -> enforceType t1 u gamma >> return ((chi |-> u) tau1)
   | otherwise -> Left $ "folding without a mu operator in " ++ show t
```

To do this, we needed to add a new "substT" function to be able to substitute into types. We added a new typeclass called "Substitutable" so we could reuse the arrow syntax.

Then, we also needed new evaluation rules, added as follows:

```
-- pg 276 E-Fld
    S.Fold tau1 t1
        | S.isNotValue t1 -> do t1' <- eval_small t1; return $ S.Fold tau1 t1'
        | otherwise       -> return t
-- pg 276 E-UnfldFld
    S.Unfold tau1 (S.Fold tau2 t1)
        | S.isValue t1 -> return t1
        | otherwise    -> do t1' <- eval_small t1; return $ S.Unfold tau1 (S.Fold tau2 t1')
-- pg 276 E-Unfld
    S.Unfold tau1 t1
        | S.isNotValue t1 -> do t1' <- eval_small t1; return $ S.Unfold tau1 t1'
```

Note, we have only implemented recursive types for operational semantics so far, but their extensions to other evaluation schemas (exluded De Bruijn) is straightforward.

Debugging Evaluation

To add debugging evaluation, we added a new evaluation stategy which adds in calls to trace to print every step of evaluation. Although the way we output it is informal, this trace is equivalent to a proof of evaluation. If we were to use a state monad to thread these steps through, we could easily print this more prettily. To avoid our code being convoluted, we used the traceM function to be able to incorporate tracing with our do syntax. As an example of a few lines from this new evaluation strategy:

```
S.App t1@(S.Abs x _ t12) v2
   | S.isValue v2 -> do traceM ("Substituting Value Into App" ++ "\n" ++ (show t) ++ "\n\n\n
                        return ((x |-> v2) t12)
S.App v1 t2
   | S.isValue v1 -> do traceM ("Evaluating Second Argument of Application" ++ "\n" ++ (show
                        t2' <- eval_small_trace t2
                        return (S.App v1 t2')
S.App t1 t2 ->       do traceM ("Evaluating First Argument of Application" ++ "\n" ++ (show
                        t1' <-  eval_small_trace t1
                        return (S.App t1' t2)
```