"A Gentle Introduction to Multi-stage Programming" Discussion Sheet

Problems in Building Program Generators

- We can represent program fragments by:
  1. Strings
  2. data types, "abstract syntax trees"
- Strings do not ensure syntactic correctness!
  - E.g., `"f (,y)"` is statically a valid string, but (usually) not a valid program.
- AST's yield well-formed syntax, but do not guarantee well-typed programs!
  - E.g., `Add (MkInt 3) (MkBool True)` may be a valid AST value, but is not well-typed (without coercions!)
- Both approaches require the programmer to ensure that there are no name clashes or variable capture!

MSP languages statically ensure that any generator only produces
- syntactically well-formed and well-typed programs;
- and ensure inadvertent naming clashes do not occur!

The practical appeal of staged interpreters lies in that they can be almost as simple as interpreter-based language implementations and at the same time be as efficient as compiler-based ones.

This is possible because the staged interpreter becomes effectively a translator from the DSL to the host language (in this case Haskell) and such translations happen at compile time!

A Staged Program = A Conventional Program + Staging Annotations

The Two Basic MSP Constructs

We can change the order of evaluation of terms using *splicing* and *quoting* thereby possibly reducing computational costs.

`[|...|]` *Quoting brackets* delay the execution of a term by turning it into code.

`$(...)` *Splice* allows the combination of smaller delayed values to construct larger ones.

```
> let a = 1 + 2       :: Int
> let a = [| 1 + 2 |] :: ExpQ

> runQ a
InfixE (Just (LitE (IntegerL 1))) (VarE GHC.Num.+) (Just (LitE (IntegerL 2)))

> let b = [| $a * $a |] :: ExpQ

> $b
9

> let aa = [|| 1 + 2 ||] :: Q (TExp Int)
```

Notice the type of the code fragment, in `aa`, is reflected in the type of the value. This is our typing guarantee! Typed splicing is via `$$(...)`.

Question What does MetaOCaml's `!` corresponds to in Template Haskell?

Basic Equivalences `$([| e |]) = e` and `[| $c |] = c`.

- Or so I claim…

Basic Integer Language

This language supports integer arithmetic, conditionals, and recursive functions.

```
data Exp  = Int Int
          | Var String
          | App String Exp
          | Add Exp Exp
          | Sub Exp Exp
          | Mul Exp Exp
          | Div Exp Exp
          | Ifz Exp Exp Exp
  deriving (Show)
```

A *functional program* consists of a sequence of declarations with a special one designed the start point.

`Declaration f x b` denotes $f = (\lambda x \to b)$; i.e., a top level item.

```
data Declaration = Declaration { name :: String , parameter :: String , body :: Exp}
```

The main method is the expression to be evaluated, given auxiliary definitions.

```
data Program = Program {supercombinators :: [Declaration] , mainMethod :: Exp}
```

Factorial `Program`

One possible encoding,

```
fact :: Program
fact = let  = Var "x" in Program
 {supercombinators = [ Declaration { name      = "fact"
                                   , parameter = "x"
                                   , body      = Ifz
                                              {- then -} (Int 1)
                                              {- else -} ( `Mul`(App "fact" ( `Sub` Int 1)))
                                   }
                    ]
 , mainMethod      = App "fact" (Int 10)
 }
```

Which corresponds to the direct presentation,

```
fact_direct :: Int
fact_direct = let f n = if n == 0 then 1 else n * f (n - 1) in f 10
```

A Simple Interpreter –§3.3

We have states corresponding to the names of variables and functions, semantically yielding integers and functions on integers.

```
type VariableTable = String -> Int
type FunctionTable = String -> Int -> Int
```

Function patching: $f[x := v]$ behaves like f but now goes to v at position x.

```
patch :: Eq a => (a -> b) -> a -> b -> (a -> b)
patch f x v = \y -> if x == y then v else f y
```

Evaluate an expression by refying the primitive symbols and using state lookup for identifiers.

```
eval :: Exp -> VariableTable -> FunctionTable -> Int
eval (Int i) env fenv = i
eval (Var v) env fenv = env v
eval (App name  exp) env fenv = fenv name $ eval exp env fenv
eval (Add e1 e2) env fenv   =  eval e1 env fenv  +     eval e2 env fenv
```

Lift a declaration into a function.

```
deval :: Declaration -> VariableTable -> FunctionTable -> (Int -> Int)
deval (Declaration name var body) env fenv = this
  where this = \x -> eval body (patch env var x) (patch fenv name this)
```

Interpret a declaration as a function and add it to our function table, if there are no more declarations then try to evaluate the main expression.

```
peval :: Program -> VariableTable -> FunctionTable -> Int
peval (Program []     exp) env fenv = eval exp env fenv
peval (Program (d:ds) exp) env fenv = peval (Program ds exp) env fenv'
  where fenv' = patch fenv (name d) (deval d env fenv)
```

~20,000 byte difference in size!

The Simple Interpreter Staged –§3.4

We now need metaprogramming support.

```
import Language.Haskell.TH        hiding (Exp)
import Language.Haskell.TH.Syntax hiding (Exp)
```

A convenient alias for readability,

```
type Code a = Q (TExp a)
```

Now state holds code rather than values,

```
type VariableTable = String -> Code Int
type FunctionTable = String -> Code (Int -> Int)
```

Quote annotations being inserted,

```
eval :: Exp -> VariableTable -> FunctionTable -> Code Int
eval (Int i) env fenv = [|| i ||]
eval (Var v) env fenv = env v
eval (App name  exp) env fenv = [|| $$(fenv name) $$(eval exp env fenv) ||]
eval (Add e1 e2) env fenv     = [|| $$(eval e1 env fenv)  + $$(eval e2 env fenv) ||]
  ...

deval :: Declaration -> VariableTable -> FunctionTable -> Code (Int -> Int)
deval (Declaration name var body) env fenv =
  [|| let this = \x -> $$(eval body (patch env var [|| x ||] ) (patch fenv name [|| this ||])
      in this
  ||]
```

~2000 byte difference in size!

Including Error Handling –§3.5

A function simply may not be defined at some point; or may “not terminate”.

```
type VariableTable = String -> Code Int
type FunctionTable = String -> Code (Int -> Maybe Int)
```

A missed opportunity to use a monad transformer ;-)

```
eval :: Exp -> VariableTable -> FunctionTable -> Code (Maybe Int)
eval (Int i) env fenv = [|| Just i ||]
eval (Var v) env fenv = [|| Just $$(env v) ||]
eval (App name  exp) env fenv =
  [||
  let f   = $$(fenv name)         :: Int -> Maybe Int
      arg = $$(eval exp env fenv) :: Maybe Int
  in
      arg >>= f
  ||]
eval (Add e1 e2) env fenv     =
  [||
  do l <- $$(eval e1 env fenv)
     r <- $$(eval e2 env fenv)
     return (l + r)
  ||]
...
```

~10,000 byte difference in size!

Controlled Inlining –§3.7

Let’s do this together!