Staging in Haskell

What is Staging

What does it Mean

Using Template Haskell

Example reduction

– (power 2)
  - unfold the definition

– (fn x => if 2=0 then 1 else x * (power (2-1) x))
  - perform the if, under the lambda

– (fn x => x * (power (2-1) x))
  - unfold power again

```
–(fn x => x * ((fn x => if 1=0
–                        then 1
–                        else x * (power (1-1) x))
–               x))
```
  - use the beta rule to apply the explicit lambda to x

Example (cont.)

– (fn x => x * (if 1=0 then 1 else x * (power (1-1) x)))
  - perform the if

– (fn x => x * (x * (power (1-1) x)))
  - unfold power again
  -

– (fn x => x * (x * ((fn x => if 0=0
–   then 1
–   else x * (power (0-1) x)))
–   x))
  - use the beta rule to apply the explicit lambda to x

– (fn x => x * (x * (if 0=0 then 1
–   else x * (power (0-1) x))))
  - perform the if

– (fn x => x * (x * 1))

Theory

- Develop a theory
- See how it applies in practice
- How does it work in Template Haskell?

Solution - Use richer annotations

- Brackets: `[| e |]`
  - no reductions allowed in **e**
  - delay computation
  - if `e:t` then `[|e|] : [|t|]` (pronounced code of t)
- Escape: `$ e`
  - relax the no reduction rule of brackets above
  - Escape may only occur inside Brackets
  - splice code together to build larger code
- Run: `run e`
  - remove outermost brackets
  - force computations which have been delayed

Calculus

- A calculus describes equivalences between program fragments.
- The rules of a calculus can be applied in any order.
- An implementation applies the rules in some fixed order.

- Traditional rules
  - beta – (\ x -> e) v → e[v/x]
  - if – if true then x else y → x
  - – if false then x else y → y
  - delta – 5 + 2 → 7

Rules for code

- Introduction rule for code
  - [| e |]

- 1st elimination rule for code (escape-bracket elim)
  - [| … $[|e|] … |] ---> [| … e … |]
    - $[|e|] must appear inside enclosing brackets
    - e must be escape free
    - [|e|] must be at level 0

- 2nd elimination rule for code (run-bracket elim)
  - run [|e|] ---> e
    - provided e has no escapes
    - the whole expression is at level 0

Power example revisited

```
power :: int -> [|int|] -> [|int|]
power n  x =
        if n=0
           then [|1|]
          else [| $x * $(power (n-1) x) |]


ans :: [| int -> int |]
ans = [| \ z -> $(power 2 [|z|]) |];
```

```
[|  \ z -> $ (power 2 [|z|]) |]
```

```
[|  \ z ->
       $(if 2=0
          then [|1|]
          else [| $[|z|] * $(power (2-1) [|z|]) |])|]
```

```
[|  \ z -> $[| $[|z|] * $(power (2-1) [|z|])  |] |])
```

```
[|  \ z -> $[| z * $(power (2-1) [|z|]) |]|])
```

```
[|  \ z ->
   $[| z * $(if 1=0
                then [|1|]
                else [| $[|z|] *
                        $(power (1-1) [|z|]) |]) |]|])
```

```
[|  \ z -> $[| z * $[| $[|z|] *
                           $(power (1-1) [|z|]) |]|]|]
[|  \ z -> $[| z * $[| z *
                           $(power (1-1) [|z|]) |]|]|]

[|  \ z -> $[| z * $[| z *
                           $(power 0 [|z|]) |]|]|]

 [|  \ z -> $[| z * $[| z * $[|1|] |]|]|]


  [|  \ z -> $[| z * $[| z * 1 |]|]|]


  [|  \ z -> $[| z * z * 1 |]|]


   [|  \ z -> z * z * 1|]
```

Meta-programming

- Programs that write programs
  - What Infrastructure is possible in a language designed to help support the algorithmic construction of other programs?
- Advantages of meta-programs
  - capture knowledge
  - efficient solutions
  - design ideas can be communicated and shared

Staging

```
inc x = x + 1
c1a = [| 4 + 3 |]

c2a = [| \ x -> x + $c1a |]


c3 = [| let f x = y - 1
             where y = 3 * x
        in f 4 + 3 |]
c4 = [| inc 3 |]
c5 = [| [| 3 |] |]
c6 = [| \ x -> x |]
```

brackets build code

The escape $, splices previously existing code (c1a) into the hole in the brackets marked by $c1a

An example

- `count 0 = []`
- `count n = n: count (n-1)`

- `count' 0 = [| [] |]`
- `count' n = [| $(lift n) : $(count' (n-1)) |]`

Exercise 18

- The traditional staged function is the power function. The term **`(power 3 x)`** returns x to the third power. The unstaged power function can be written as:

```
power:: Int -> Int -> Int
power 0 x = 1
power n x = x * power (n-1) x
```

Write a staged power function:

```
pow:: Int -> [| Int |] -> [| Int |]
```

such that **`(pow 3 [|99|])`** evaluates to

`[| 99 * 99 * 99 * 99 * 1 |]`.

This can be written simply by placing staging annotations in the unstaged version.

A simple object-language

```
data Exp:: * where
  Variable:: String  -> Exp
  Constant:: Int -> Exp
  Plus:: Exp
  Less:: Exp
  Apply:: Exp -> Exp -> Exp
  Tuple:: [Exp] -> Exp

-- exp1  represents “x+y”
exp1 = Apply Plus
            (Tuple [Variable "x"
                   ,Variable "y"])
```

A simple value domain

```
data Value :: * where
   IntV:: Int -> Value
   BoolV:: Bool -> Value
   FunV:: (Value -> Value) -> Value
   TupleV :: [Value] -> Value
```

Values are a disjoint sum of many different semantic things, so they will all have the same type. We say the values are tagged.

A simple semantic mapping

```
eval:: (String -> Value) -> Exp -> Value
eval env (Variable s) = env s
eval env (Constant n) = IntV n
eval env Plus = FunV plus
  where plus (TupleV[IntV n ,IntV m]) = IntV(n+m)
eval env Less = FunV less
  where less (TupleV[IntV n ,IntV m]) = BoolV(n < m)
eval env (Apply f x) =
  case eval env f of
    FunV g -> g (eval env x)
eval env (Tuple xs) = TupleV(map (eval env) xs)
```

Compared to a compiler, a mapping has two forms of overhead

- Interpretive overhead
- tagging overhead

Removing Interpretive overhead

- We can remove the interpretive overhead by the use of staging.
- I.e. for a given program, we generate a meta language program (here that is Template Haskell) that when executed will produce the same result.
- Staged programs often run 2-10 times faster than un-staged ones.

A staged semantic mapping

```
-- operations on values
plus (TupleV[IntV n ,IntV m]) = IntV(n+m)
less (TupleV[IntV n ,IntV m]) = BoolV(n < m)
apply (FunV g) x = g x

stagedEval:: (String -> [| Value |]) -> Exp -> [| Value |]

stagedEval env (Variable s) = env s
stagedEval env (Constant n) = lift(IntV n)
stagedEval env Plus = [| FunV plus |]
stagedEval env Less = [| FunV less |]
stagedEval env (Apply f x) =
   [| apply $(stagedEval env f) $(stagedEval env x) |]
stagedEval env (Tuple xs) =
   [| TupleV $(mapLift (stagedEval env) xs) |]
 where mapLift f [] = lift []
       mapLift f (x:xs) = [| $(f x) : $(mapLift f xs) |]
```

Observe

```
ans = stagedEval f exp1
  where f "x" = lift(IntV 3)
        f "y" = lift(IntV 4)


[| %apply (%FunV %plus)
          (%TupleV [IntV 3,IntV 4])
|] : [| Value |]
```

Removing tagging

- Consider the residual program

```
[| %apply (%FunV %plus)
          (%TupleV [IntV 3,IntV 4])
|]
```

The **FunV**, **TupleV** and **IntV** are tags.

They make it possible for integers, tuples, and functions to have the same type (**Value**)

But, in a well typed object-language program they are superfluous.

Typed object languages

- We will create an indexed term of the object language.
- The index will state the type of the object-language term being represented.

```
data Term:: * -> * where
  Const :: Int -> Term Int                    -- 5
  Add:: Term ((Int,Int) -> Int)               -- (+)
  LT:: Term ((Int,Int) -> Bool)               -- (<)
  Ap:: Term(a -> b) -> Term a -> Term b  -- (+) (x,y)
  Pair:: Term a -> Term b -> Term(a,b)   -- (x,y)
```

- Note there are no variables in this object language

The value domain

- The value domain is just a subset of Haskell values.
- No tags are necessary.

A tag less interpreter

```
evalTerm :: Term a -> a
evalTerm (Const x) = x
evalTerm Add = \ (x,y) -> x+y
evalTerm LT = \ (x,y) -> x < y
evalTerm (Ap f x) =
    evalTerm f (evalTerm x)
evalTerm (Pair x y) =
    (evalTerm x,evalTerm y)
```

Exercise 1

- In the object-languages we have seen so far, there are no variables. One way to add variables to a typed object language is to add a variable constructor tagged by a name and a type. A singleton type representing all the possible types of a program term is necessary. For example, we may add a **`Var`** constructor as follows (where the **`Rep`** is similar to the **`Rep`** type from Exercise 9).

```
data Term:: * -> * where
  Var:: String -> Rep t -> Term t          -- x
  Const :: Int -> Term Int                  -- 5
..
```

- Write a GADT for **`Rep`**. Now the evaluation function for **`Term`** needs an environment that can store many different types. One possibility is use existentially quantified types in the environment as we did in Exercise 21. Something like:

```
data Env where
  Empty :: Env
  Extend :: String -> Rep t -> t -> Env -> Env

eval:: Term t -> Env -> t
```

- Write the evaluation function for the **`Term`** type extended with variables. You will need a function akin to **`test`** from the lecture on GADTs, recall it has type: **`test:: Rep a -> Rep b -> Maybe(Equal a b).`**

Typed Representations for languages with binding.

- The type (`Term a`) tells us it represents an object-language term with type a
- If our language has variables, what type would (`Var “x”`) have?
- It depends upon the context.
- We need to reflect the type of the variables in a term, in an index of the term, as well as the type of the whole term itself.
- E.g. **`t :: Term {`a=Int,`b=Bool} Int`**

Exercise

- A common use of labels is to name variables in a data structure used to represent some object language as data. Consider the GADT and an evaluation function over that object type.

```
data Expr:: * where
  VarExpr :: Label t -> Expr
  PlusExpr:: Expr -> Expr -> Expr

valueOf:: Expr -> [exists t .(Label t,Int)] -> Int
valueOf (VarExpr v) env = lookup v env
valueOf (PlusExpr x y) env =
    valueOf x env + valueOf y env
```

- Write the function:

```
lookup:: Label v -> [exists t .(Label t,Int)] -> Int
```

hint: don't forget the use of **"Ex"** .

Languages with binding

```
data Lam:: Row Tag * -> * -> *  where
  Var      :: Label s -> Lam (RCons s t env) t
  Shift    :: Lam env t -> Lam (RCons s q env) t
  Abstract :: Label a ->
              Lam (RCons a s env) t ->
              Lam env (s -> t)
  App      :: Lam env (s -> t) ->
              Lam env s ->
              Lam env t
```

A tag-less interpreter

```
data Record :: Row Tag * -> * where
 RecNil :: Record RNil
 RecCons :: Label a -> b ->
            Record r -> Record (RCons a b r)
```

```
eval:: (Lam e t) -> Record e -> t
eval (Var s) (RecCons u x env) = x
eval (Shift exp) (RecCons u x env) =
   eval exp env
eval (Abstract s body) env =
   \ v -> eval body (RecCons s v env)
eval (App f x) env = eval f env (eval x env)
```

Exercise

- Another way to add variables to a typed object language is to reflect the name and type of variables in the meta-level types of the terms in which they occur. Consider the GADTs:

```
data VNum:: Tag -> * -> Row Tag * -> * where
  Zv:: VNum l t (RCons l t row)
  Sv:: VNum l t (RCons a b row) ->
              VNum l t (RCons x y (RCons a b row))
 deriving Nat(u)  -- 0u = Zv,  1u = Sv Zv,  2u = Sv(Sv Zv), etc

data Exp2:: Row Tag * -> * -> * where
  Var:: Label v -> VNum v t e -> Exp2 e t
  Less:: Exp2 e Int -> Exp2 e Int -> Exp2 e Bool
  Add:: Exp2 e Int -> Exp2 e Int -> Exp2 e Int
  If:: Exp2 e Bool -> Exp2 e t -> Exp2 e t -> Exp2 e t
```

- What are the types of the terms **`(Var `x 0u), (Var `x 1u)`**, and **`(Var `x 2u),`** Now the evaluation function for **`Exp2`** needs an environment that stores both integers and booleans. Write a datatype declaration for the environment, and then write the evaluation function. One way to approach this is to use existentially quantified types in the environment as we did in the previous exercise. Better mechanisms exist. Can you think of one?

A compiler = A staged, tag-less interpreter

```
data SymTab:: Row Tag * -> * where
  Insert :: Label a -> [| b |] -> SymTab e ->
            SymTab (RCons a b e)
  Empty :: SymTab RNil

compile:: (Lam e t) -> SymTab e -> Code t
compile (Var s) (Insert u x env) = x
compile (Shift exp) (Insert u x env) =
   compile exp env
compile (Abstract s body) env =
   [| \ v -> $(compile body (Insert s [|v|] env)) |]
compile (App f x) env =
   [| $(compile f env) $(compile x env) |]
```

Exercise

- A staged evaluator is a simple compiler. Many compilers have an optimization phase. Consider the term language with variables from a previous Exercise.

```
data Term:: * -> * where
  Var:: String -> Rep t -> Term t
  Const :: Int -> Term Int                    -- 5
  Add:: Term ((Int,Int) -> Int)               -- (+)
  LT:: Term ((Int,Int) -> Bool)               -- ([|)
  Ap:: Term(a -> b) -> Term a -> Term b  -- (+) (x,y)
  Pair:: Term a -> Term b -> Term(a,b)    -- (x,y)
```

- Can you write a well-typed staged evaluator the performs optimizations like constant folding, and applies laws like *(x+0) = x* before generating code?

Template Haskell

- All this is fine, but how do we do it in Haskell
- The notes above are done in Omega (very similar) to Haskell.
- Relating the notes to Template Haskell

Template Haskell is not Typed

- The type [| t |]   i.e.   Code of t, does not exist
- Instead we use a monad, Q, called the quoting monad, that respects scoping but not types.
- Q Exp is essentially equivalent to [| t |]
  - Note the type information has been lost.

Example

```
data Nat = Zero | Succ Nat
```

```
-- nat :: Int -> [| Nat |]
nat :: Int -> Q Exp
nat 0 = [e| Zero |]
nat n = [e| Succ $(nat (n-1)) |]
```

Inspecting generated code

```
class PPr  t where
  ppr:: t -> Doc

sh:: Ppr a => Q a -> IO ()
sh x =
  do str <- runQ(do { a <- x
                    ; return(show(ppr a))})
     putStrLn str

*S> sh (nat 4)
S.Succ (S.Succ (S.Succ (S.Succ S.Zero)))
```

More TH examples

```
sumf 0 x = x
sumf n x =
  [e| \ y -> $(sumf (n-1) [e| $x + y |]) |]

pow :: Int -> Q Exp -> Q Exp
pow 0 x = [| 1 |]
pow 1 x = x
pow n x = [e| $x * $(pow (n-1) x) |]

power n = [e| \ x -> $(pow n [e| x |]) |]
```