Compile-time Metaprogramming

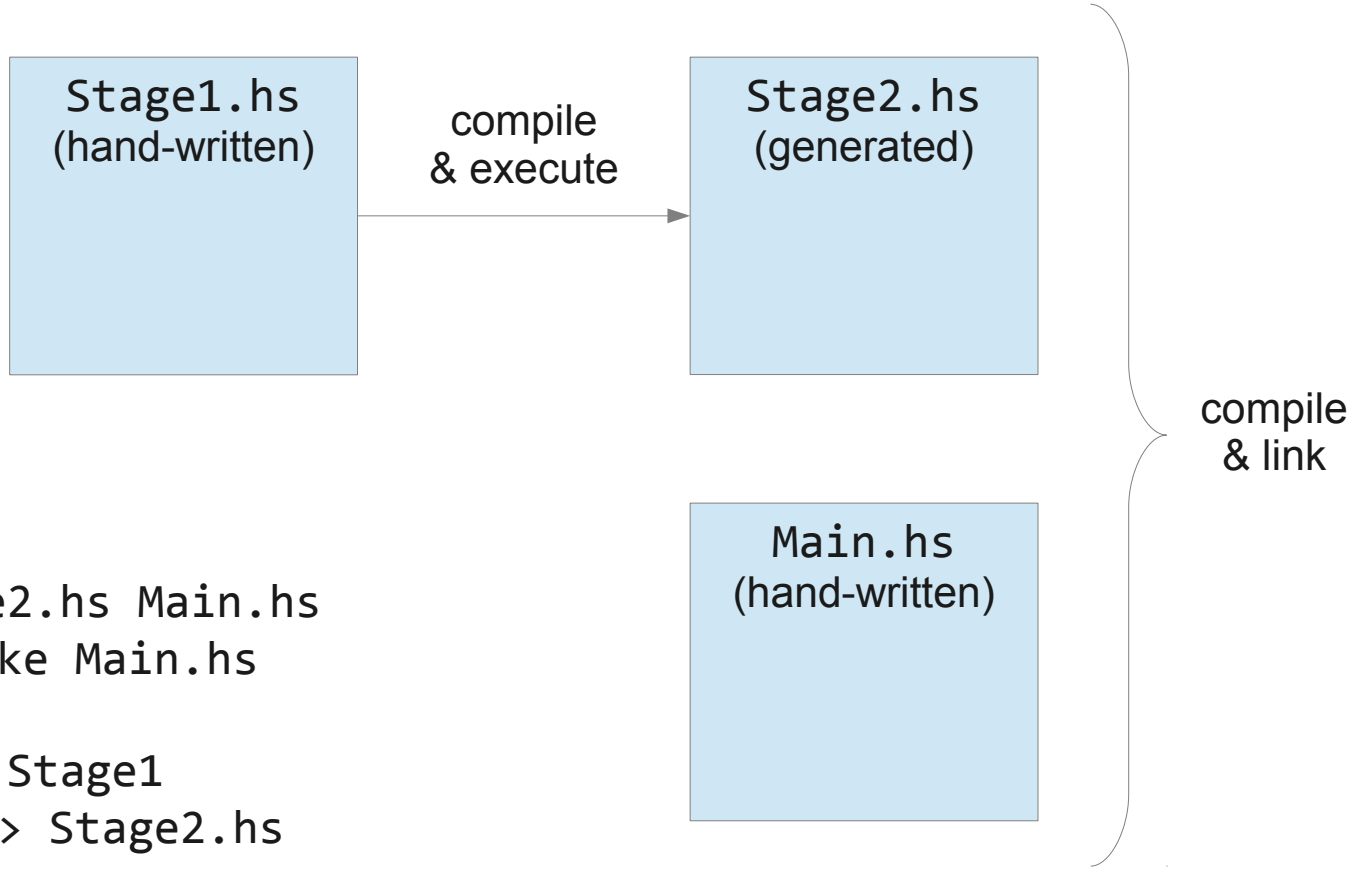
with

Template Haskell and Quasiquotation

Matthew Brecknell

m.brck.nl/bfpg6

Staged programming (with a Makefile)




```
Main: Stage2.hs Main.hs
    ghc –make Main.hs

Stage2.hs: Stage1
    Stage1 > Stage2.hs

Stage1: Stage1.hs
    ghc –make Stage1.hs
```

Template Haskell

- Abstract Syntax
- TH Quotes
- Quotation monad
- Splices
- QuasiQuotes

Abstract Syntax

Types for manipulating Haskell code as data

```
data Name -- names of variables, types, etc.
data Dec  -- declarations of functions, classes, instances, etc.
data Exp  -- expressions
data Type -- types
```

etc.

Example: encode this

```
\x → x + 1
```

As abstract syntax:

```
LamE [VarP (mkName "x")]
     (InfixE (Just (VarE (mkName "x")))
             (VarE (mkName "GHC.Num.+"))
             (Just (LitE (IntegerL 1)))) :: Exp
```

TH Quotes

Where possible, quote the code you want to generate:

```
[| \x → x + 1 |] :: Q Exp
```

(`Q` is the quotation monad – more on this soon.)

```
ghci> runQ [| \x → x + 1 |]
LamE [VarP x_0] (InfixE (Just (VarE x_0)) ...)
```

TH Quotes

Expression quotes

```
[| \x → x + 1 |] :: Q Exp
```

Type quotes

```
[t| Int → Int |] :: Q Type
```

Pattern quotes

```
[p| xs@(x:r) |] :: Q Pat
```

Declaration quotes

```
[d| data Pair a = Pair a a |] :: Q [Dec]
```

Quoting Names

```
'foldr  :: Name  -- value
'Just   :: Name  -- data constructor
''Maybe :: Name  -- type constructor
```

The Quotation Monad (Q)

Provides access to the compilation environment

- Unique name generation

```
newName :: String → Q Name
```

- Reification

```
reify :: Name → Q Info
```

- Error reporting

```
location :: Q Loc
report :: Bool → String → Q ()
```

- Arbitrary IO

```
runIO :: IO a → Q a
```

Splicing $(…)

Execute stage 1 code-generation during stage 2 compilation.

```
id :: a → a
id = $( do x ← newName "x"; lamE [varP x] (varE x) )
```

Quoting and splicing are inverse:

```
$( [| ... |] )  =  ...
[| $( ... ) |]  =  ...
```

Top-level splices can omit the $(…) notation

```
data Foo = …
deriveShow ''Foo
```

```
data Foo = …
$(deriveShow ''Foo)
```

Example: printf

```
data Format = Dec | Str | Lit String

gen :: [Format] -> Q Exp -> Q Exp
gen (Dec : fs) q = [| \n -> $(gen fs [| $q ++ show n |]) |]
gen (Str : fs) q = [| \s -> $(gen fs [| $q ++ s |]) |]
gen (Lit s : fs) q = gen fs [| $q ++ s |]
gen [] q = q

parse :: String → [Format]
parse = ...

printf :: String → Q Exp
printf s = gen (parse s) [ "" ]



ghci> $(printf "%d: %s") 1 "One"
"1: One"
```

QuasiQuotes

```
data QuasiQuoter = QuasiQuoter {
  quoteExp  :: String → Q Exp,
  quotePat  :: String → Q Pat,
  quoteType :: String → Q Type,
  quoteDec  :: String → Q [Dec]
  }
```

Example: regular expressions

```
regex = QuasiQuoter {
  quoteExp  = regexExp,  -- assume suitable implementation
  quotePat  = regexPat,
  quoteType = error "...",
  quoteDec  = error "...",
  }

evenCs :: String → Match
evenCs = [regex|^[^c]*(c[^c]*c[^c]*)*$|]

evenCs' = $(regexExp "^[^c]*(c[^c]*c[^c]*)*$")
```