Template Metaprogramming for Haskell

Tim Sheard and Simon Peyton Jones

Presented by John Chilton
March 5, 2007
http://www.jmchilton.net/index.html?page=geek:template_haskell

Outline

Haskell

See `haskell.hs` and `haskell-monad.hs`

Metaslide

The paper is out of date, a lot of code doesn't work and many names and concepts have changed slightly. I will present an up to date version of Template Haskell, with some modified and new examples that actually compile.

http://www.jmchilton.net/index.html?page=geek:template_haskell

Metaprogramming

What?

Metaprogramming facilities allows the programmer to analyze and compute parts of programs.

Why?

- Compile-time, programmable optimizations and analysis
- Mechanism for creating new abstractions
- Application: Embedded domain specific languages

Introducing the `printf` Example

The Goal

```
printf "Error:  %s on line %d." msg line
```

Problems

- Cannot be implemented in ordinary Haskell
- Interdependence of types
- Variable number of arguments

Template Haskell Solution

Generate appropriate procedure at compile time.

Introducing the printf Example

The Goal

```
printf "Error: %s on line %d." msg line
```

Problems

- Cannot be implemented in ordinary Haskell
- Interdependence of types
- Variable number of arguments

Template Haskell Solution

Generate appropriate procedure at compile time.

Introducing the printf Example

The Goal

`printf "Error: %s on line %d." msg line`

Problems

- Cannot be implemented in ordinary Haskell
- Interdependence of types
- Variable number of arguments

Template Haskell Solution

Generate appropriate procedure at compile time.

`$(...)` Syntax

`$(...)`

$\Downarrow$

Must contain valid Haskell expression

$\Downarrow$

Expression must yield Haskell syntax tree data

$\Downarrow$

Evaluated at runtime and inserted into program

Two Types of Spliceable Data

ExpQ Syntax trees corresponding to valid Haskell expressions (literals, variables, function application, conditionals, etc.). Called Expr in paper.

DecQ Syntax trees corresponding to a valid Haskell declarations (functions, values, data types, classes, etc.). Called Decl in paper.

Outline

The Problem

Example

```
> listSelect 2 [1,2,3,4,5]
3
> :type listSelect
Int -> [a] -> a
> sel 2 (1,2,3,4,5) --cannot be implemented
```

The Problem

No generic tuple, (a), type corresponding to list type, [a].

The Problem

Example

```
> listSelect 2 [1,2,3,4,5]
3
> :type listSelect
Int -> [a] -> a
> sel 2 (1,2,3,4,5) --cannot be implemented
```

The Problem

No generic tuple, (a), type corresponding to list type, [a].

sel

Example

```
> $(sel 3 5) (1,2,3,4,5)
3
> (\(a1,a2,a3,a4,a5) -> a3) (1,2,3,4,5)
3
```

Implementing sel

Example

```
(\(a1, a2, a3, a4) -> a2)
```

- `lamE :: [PatQ] -> ExpQ -> ExpQ`
- `tupP :: [PatQ] -> PatQ`
- `varP :: Name -> PatQ`
- `varE :: Name -> ExpQ`
- `mkName :: String -> Name`

Implementation

See `Sel.hs`

printf

Type

Implement printf with type - `String -> ExpQ`

Example

```
$(printf "Error:  %s on line %d.") msg line
```

Compile Time Expansion

```
(\s0 -> \n1 ->
   "Error:  " ++ s ++ " on line " ++ show n)
 msg line
```

printf

Type

Implement printf with type - `String -> ExpQ`

Example

```
$(printf "Error:  %s on line %d.") msg line
```

Compile Time Expansion

```
(\s0 -> \n1 ->
   "Error:  " ++ s ++ " on line " ++ show n)
 msg line
```

printf

Type

Implement printf with type - `String -> ExpQ`

Example

```
$(printf "Error:  %s on line %d.") msg line
```

Compile Time Expansion

```
(\s0 -> \n1 ->
   "Error:  " ++ s ++ " on line " ++ show n)
 msg line
```

```
appE ::  ExpQ -> ExpQ -> ExpQ
```

Example

```
showE ::  ExpQ -> ExpQ
showE e = appE (varE (mkName "show")) e
```

```
appE ::  ExpQ -> ExpQ -> ExpQ
```

Example

```
showE ::  ExpQ -> ExpQ
showE e = appE (varE (mkName "show")) e
```

```
infixE::(Maybe ExpQ) -> ExpQ -> (Maybe ExpQ) -> ExpQ
```

Haskell's Maybe Type

```
data Maybe a = Nothing | Just a
```

Example

```
> map (+ 2) [1..5]
[3,4,5,6,7]
```

concatE

```
concatE ::  ExpQ -> ExpQ -> ExpQ
concatE e1 e2 =
    infixE (Just e1) (varE (mkName "++")) (Just e2)
```

```
infixE::(Maybe ExpQ) -> ExpQ -> (Maybe ExpQ) -> ExpQ
```

Haskell's Maybe Type

```
data Maybe a = Nothing | Just a
```

Example

```
> map (+ 2) [1..5]
[3,4,5,6,7]
```

concatE

```
concatE ::  ExpQ -> ExpQ -> ExpQ
concatE e1 e2 =
    infixE (Just e1) (varE (mkName "++")) (Just e2)
```

```
infixE::(Maybe ExpQ) -> ExpQ -> (Maybe ExpQ) -> ExpQ
```

Haskell's Maybe Type

```
data Maybe a = Nothing | Just a
```

Example

```
> map (+ 2) [1..5]
[3,4,5,6,7]
```

concatE

```
concatE ::  ExpQ -> ExpQ -> ExpQ
concatE e1 e2 =
    infixE (Just e1) (varE (mkName "++")) (Just e2)
```

printf Implementation - See Printf1.hs

```
data Format = D | S | L String

printf ::  String -> ExpQ
printf s = gen (parse s) (litE (StringL ""))

parse ::  String -> [Format]
...
gen ::  [Format] -> ExpQ -> ExpQ
...
```

Example

```
> parse "Error:  %s on line %d."
 [L "Error:  ", S, L " on line ", D, L "."]
```

main1.hs

```
module Main where
import Printf1

errorAt :: String -> Int -> String
errorAt msg line =
    $(printf "Error: %s on line %d") msg line

main = putStrLn (errorAt "Undeclared variable" 314)
```

Example

```
@jl % ghc --make -fth main1.hs
@jl % main1
Error: Undeclared variable on line 314
```

main2.hs

```
module Main where
import Printf1

errorVar ::  String -> String -> String
errorVar msg var =
     $(printf "Error %s with variable %s") msg var

main = putStrLn (errorVar "Undeclared variable" "fo")
```

Example

```
@jl % ghc --make -fth main2.hs
@jl % main2
Error fo with variable fo
```

main2.hs

```
module Main where
import Printf1

errorVar ::  String -> String -> String
errorVar msg var =
     $(printf "Error %s with variable %s") msg var

main = putStrLn (errorVar "Undeclared variable" "fo")
```

Example

```
@jl % ghc --make -fth main2.hs
@jl % main2
Error fo with variable fo
```

main2.hs

```
module Main where
import Printf1

errorVar ::  String -> String -> String
errorVar msg var =
     $(printf "Error %s with variable %s") msg var

main = putStrLn (errorVar "Undeclared variable" "fo")
```

Problem - Generated Code

```
(\s -> \s ->
   "" ++ "Error:  " ++ s ++ " with variable " ++ s)
 msg var
```

gensym operator

- Common in Lisp & Scheme dialects
- Generates a fresh variable name, not clashing with any existing variables
- Can be used to simulate lexical scoping
- Template Haskell now calls it `qNewName`

```
qNewName :: String -> Q Name
```

Using gensym

Broken - Printf1.hs

```
gen (D : xs) x =
  let body = concatE x (showE (varE (mkName "n")))
  in lamE [varP (mkName "n")] (gen xs body)
```

Fixed - Printf2.hs

```
gen (D : xs) x =
  do { n <- qNewName "n";
       let body = concatE x (showE (varE n))
       in lamE [varP n] (gen xs body) }
```

main3.hs

```
module Main where
import Printf2

errorVar ::  String -> String -> String
errorVar msg var =
   $(printf "Error %s with variable %s") msg var

main = putStrLn (errorVar "Undeclared variable" "fo")
```

Example

```
@jl % ghc --make -fth main3.hs
@jl % main3
Error Undeclared variable with variable fo
```

Outline

1. Introduction
   - Introduction to Haskell, Metaprogramming, and `printf`
   - The `$(...)` Operator
2. Building Haskell Syntax Trees
   - Example I: `sel`
   - Example II: `printf`
3. Some Additonal Features
   - The Quasi-Quote
   - Splicing Declarations
   - Reification
4. Odds and Ends
   - Embedded DSLs and Metaprogramming
   - Scheme/Lisp Macros, MetaML, and Liskell
   - Questions

- `$( ) :: ExpQ -> "Haskell code"`
- `[| |] :: "Haskell Code" -> ExpQ`

Toy Example

Example

```
[| let x = 7
   in show x |]
```

Result

```
Q (LetE [ValD (VarP x_0)
              (NormalB (LitE (IntegerL 7)))
              []
        ]
        (AppE (VarE GHC.Show.show) (VarE x_0))
   )
```

More Examples

showE

```
showE ::  ExpQ -> ExpQ
showE e = [| show $(e) |]
```

concatE

```
concatE ::  ExpQ -> ExpQ -> ExpQ
concatE e1 e2 =
    [| $(e1) ++ $(e2) |]
```

printf with Quasi-Quotes

gen

```
gen [] x = x
gen (D : xs) x =
    [| \n -> $(gen xs [| $(x)++show n |]) |]
gen (S : xs) x =
    [| \s -> $(gen xs [| $(x)++s |]) |]
gen (L s :  xs) x =
    gen xs [| $(x) ++ $(lift s) |]
```

Scoping

Variables in the Quasi-Quote are statically scoped.

printf with Quasi-Quotes

gen

```
gen [] x = x
gen (D : xs) x =
    [| \n -> $(gen xs [| $(x)++show n |]) |]
gen (S : xs) x =
    [| \s -> $(gen xs [| $(x)++s |]) |]
gen (L s :  xs) x =
    gen xs [| $(x) ++ $(lift s) |]
```

Scoping

Variables in the Quasi-Quote are statically scoped.

Other Quasi-Quotes

- `$( ) :: ExpQ -> "Haskell code"`
- `[| |] :: "Haskell Code" -> ExpQ`
- `[d| |] :: "Haskell Code" -> DecQ`
- `[t| |] :: "Haskell Code" -> TypeQ`
- `[p| |] is not implemented, breaks a lot of code from the paper`

Splicing Declarations - gen_sels

mainsel2.hs

```
module Main where
import Sel

$(gen_sels 7)
-- Defines procedures sel1of7, sel2of7, ..., sel7of7

main = putStrLn (sel4of7 (1,"b",3,"d",sqrt,"f",1))
```

Example

```
@jl % ghc -fth --make mainsel2.hs
@jl % mainsel2
d
```

reify

```
reify ::  Name -> InfoQ
```

reify

Lookups up the compiler's information about Name. The name can be the name of a variable, data type, class, etc.

Note

Everything about reification in the paper is out of date. `reify` encapsulates many of the procedures mentioned, and other functionality such as `reifyOpt` or `reifyLocn` are not implemented.

Info (1 / 4)

The Info Data Type

data Info = *TyConI Dec* |
DataConI Name Type Name Fixity |
VarI Name Type (Maybe Dec) Fixity |...

Example

```
-- data Point = Pt Rational Rational
TyConI (DataD []
              Main.Point
              []
              [NormalC Main.Pt
                       [(NotStrict, ConT Rational),
                        (NotStrict, ConT Rational)]]
              [])
```

Info (2 / 4)

The Info Data Type

```
data Info = TyConI Dec |
            DataConI Name Type Name Fixity |
            VarI Name Type (Maybe Dec) Fixity |...
```

Example

```
-- data Point = Pt Rational Rational
DataConI Main.Pt
         (AppT (AppT ArrowT (ConT Rational))
               (AppT (AppT ArrowT (ConT Rational))
                     (ConT Main.Point)))
         Main.Point
         (Fixity 9 InfixL)
```

Info (3 / 4)

The Info Data Type

```
data Info = TyConI Dec |
            DataConI Name Type Name Fixity |
            VarI Name Type (Maybe Dec) Fixity |...
```

Example

```
-- x = 2.0
VarI Main.x
     (ConT Double)
     Nothing
     (Fixity 9 InfixL)
```

Info (4 / 4)

The Info Data Type

```
data Info = TyConI Dec |
            DataConI Name Type Name Fixity |
            VarI Name Type (Maybe Dec) Fixity |...
```

Example

```
-- foo x = x ++ "bar"
VarI Main.foo
     (AppT (AppT ArrowT (ConT GHC.Base.String))
           (ConT GHC.Base.String))
     Nothing
     (Fixity 9 InfixL)
```

Outline

Embedded DSLs

Template Haskell can be used to aid in the construction of embedded DSLs for Haskell.

- Automatic creation of data types and functions
- Changing semantics of Haskell code

Automatic creation of data types and functions (1/2)

Reify data types and classes and automatically generate domain specific code.

- Writing XML
- Network Transmission
- GUI Creation
- ...

Automatic creation of data types and functions (2/2)

At compile time you can generate data types and functions from domain specific declarations.

- Generate data type and functions from XML declarations (paper example)
- Generate data types and functions for interacting with a database from a DB schema (ala Ruby on Rails)
- Generate type safe functions for writing XML (tags and attributes) from declarative descriptions valid tags and attributes.

Changing semantics of Haskell code

Can use Quasi-quote to ensure type-safe syntactically valid Haskell is written, and then interpret it as you wish.

- "Compiling" portions of Haskell programs to other programming languages such as JavaScript
- Randomized Linear Perturbations

Randomized Linear Perturbations

Example

```
func ::  Double -> Double -> Double -> IO Double
func x y z = $(rlpTransform [| x + 2 * y * z |])
```

Spliced Body Is

```
do zRand <- randomUniform01
   yRand <- randomUniform01
   xRand <- randomUniform01
   return
     (let r = x + (2.0 * (y * z))
      in if not (r == (0%1)) then r
         else let r = xRand + ((2.0 * (yRand * z)) +
                                 (2.0 * (y * zRand)))
              in if not (r == (0%1)) then r
                 else let r = 2.0 * (yRand * zRand)
                      in if not (r == (0%1)) then r else 0%1)
```

Metaprogramming is a common paradigm in functional programming, and is spreading to other scripting languages such as Python and Ruby.

- Lisp/Scheme have builtin procedures, very integral parts of the languages
- MetaML is a metaprogramming extension to Standard ML
- Liskell is another extension to Haskell to allow metaprogramming

Lisp Programs are Lisp Data (1/2)

Example

```
[d| euclidDist (Pt x1 y1) (Pt x2 y2) =
      let dx = (x1 - x2)
          dy = (y1 - y2)
      in sqrt( dx * dx + dy * dy ) |]
```

Haskell Syntax Tree

```
[FunD euclidDist [Clause [ConP Main.Pt [VarP x1_0,VarP y1_1],ConP
Main.Pt [VarP x2_2,VarP y2_3]] (NormalB (LetE [ValD (VarP dx_5) (NormalB
(InfixE (Just (VarE x1_0)) (VarE GHC.Num.-) (Just (VarE x2_2)))) [],ValD
(VarP dy_4) (NormalB (InfixE (Just (VarE y1_1)) (VarE GHC.Num.-) (Just
(VarE y2_3)))) []] (AppE (VarE GHC.Float.sqrt) (InfixE (Just (InfixE
(Just (VarE dx_5)) (VarE GHC.Num.*) (Just (VarE dx_5)))) (VarE
GHC.Num.+) (Just (InfixE (Just (VarE dy_4)) (VarE GHC.Num.*) (Just (VarE
dy_4)))))))) []]]
```

Lisp Programs are Lisp Data (2/2)

Example

```
`(define (euclid-dist p1 p2)
   (let ((dx (- x1 x2))
         (dy (- y1 y2)))
     (sqrt (+ (* dx dx) (* dy dy)))))
```

Scheme Syntax Tree

```
(define (euclid-dist p1 p2) (let ((dx (- x1 x2)) (dy
(- y1 y2))) (sqrt (+ (* dx dx) (* dy dy)))))
```

Scheme/Lisp syntax trees are are trivial to parse, construct, interpret, and composed entirely of normal scheme data types.

Call Site Syntactic Baggage

Haskell

```
func x y z = $(rlpTransform [| (x + (2 * y * z)) |])
```

Scheme

```
(define (func x y z) (rlpTransform (+ x (* 2 y z))))
```

Freedom

Template Haskell, (`$(...)` and `[|...|]`), requires syntactically valid constructions. Scheme macros merely require parseablility. This allows for extension of the semantics of the language in many more intresting ways.

Haskell - requires syntactic correctness

```
func x y z = $(rlpTransform [| (x + (2 * y * z)) |])
```

Scheme

```
(define (func x y z) (rlpTransform x + 2 * y * z))
```

Scheme

```
(postfix (y z * 2 * x +))
```

MetaML

- ML has similar syntax to Haskell, but is not lazy and is not purely functional
- MetaML also features a Quasi-Quoting feature, type-safety, static scoping.
- MetaML can build and execute code at run-time.
- Template Haskell can generate type unsafe programs but they won't compile, MetaML will only generate type safe programs (Necessary for run-time type safe execution)

Liskell

http://clemens.endorphin.org/liskell - Syntax frontend over Haskell aimed at providing Lisp-like Syntax and metaprogramming facilities for Haskell programming.

Liskell Example

```
(defdata Point2DType (Point2D Rational Rational))

(define (distance2D (Point2D x1 y1) (Point2D x2 y2))
  (let ((dx (- x1 x2))
        (dy (- y1 y2)))
    (sqrt (+ (* dx dx) (* dy dy)))))
```

Questions

Questions...

Three Levels of Abstraction

Three levels of abstraction:

- Ordinary algebraic data types represents Haskell syntax trees.
- Q monad wrappers over algebraic data types for generating fresh names and interacting with the compilers symbol table
- Quasi-Quote

Example

```
data Exp = AppE Exp Exp | ...

appE ::  ExpQ -> ExpQ -> ExpQ
appE e1 e2 = do { a <- e1; b <- e2; return (AppE a b)
}
```

Three Levels of Abstraction

Three levels of abstraction:

- Ordinary algebraic data types represents Haskell syntax trees.
- Q monad wrappers over algebraic data types for generating fresh names and interacting with the compilers symbol table
- Quasi-Quote

Example

```
data Exp = AppE Exp Exp | ...

appE ::  ExpQ -> ExpQ -> ExpQ
appE e1 e2 = do { a <- e1; b <- e2; return (AppE a b)
}
```

Typing Template Haskell

Type checking and code execution must become interleaved.

Example

```
$(printf "Error:  %s on line %d.") msg line
```

When type checker encounters $, it does the following

- Type check body of splice
- Compile and execute body
- Splice in the resulting code
- Continue to type check result

Cross-stage Persistence

```
module T( genSwap ) where
  swap (a,b) = (b,a)
  genSwap x = [|swap x |]
```

```
module Foo where
  import T(genSwap)
   swap = True
  foo = $(genSwap (4,5))
```

swap

Quasi-Quote are lexically scoped, and swap will be bound to the swap in scope at occurrence site in T even though it is not exported or in scope while in module Foo.

Lift

```
module T( genSwap ) where
  swap (a,b) = (b,a)
  genSwap x = [|swap x |]
```

x

x is a free variable and is inferred to be of class `Lift`.

Lift

```
class Lift t where
  lift ::  t -> ExpQ
instance Lift Int
  lift n = litE (IntegerL t)
instance Lift (Lift a, Lift b) => Lift (a,b) where
  lift(a,b) = tupE [lift a, lift b]
```