Prototyping Generic Programming using Template Haskell

Ulf Norell and Patrik Jansson

Computing Science
Chalmers University of Technology

{ulfn,patrikj}@cs.chalmers.se

July 13, 2004

Overview

- Motivation
- Template Haskell
- A tiny generic programming language
- Implementation in Template Haskell
- Conclusions

Motivation

- Fact: Implementing a full-fledged generic programming language is *a lot of work*
- Fact: There is no obvious *right* way of designing a generic programming language

Motivation

- Fact: Implementing a full-fledged generic programming language is *a lot of work*
- Fact: There is no obvious *right* way of designing a generic programming language
- Conclusion: We need a light-weight approach to implementation of generic programming

Motivation

- Fact: Implementing a full-fledged generic programming language is *a lot of work*
- Fact: There is no obvious *right* way of designing a generic programming language
- Conclusion: We need a light-weight approach to implementation of generic programming
- Solution: Template Haskell

Template Haskell

- Sheard, Peyton Jones: *Template meta-programming for Haskell* (Haskell Workshop 2002)
- Extension to Haskell
  - available in GHC 6.0 and above.
- Enables meta-programming
  - code can be evaluated at compile-time
- Similar to Meta ML
  - but quoted code is untyped

Template Haskell

- Abstract syntax for Haskell
  - `Exp`, `Dec`, `Typ`, ...
- The quotation monad `Q`
  - handles fresh name generation
- We can *lift* Haskell code to abstract syntax using `[| |]`

Haskell code

```
absId :: Q Exp
absId = [| \x -> x |]
```

- or we can use combinators to construct the abstract syntax directly

Haskell code

```
absId :: Q Exp
absId = do x <- gensym "x"
           lamE [varP x] (varE x)
```

Template Haskell

- Yet another way is to use reification

Haskell code

```
reifyDecl Bool :: Q Dec
```

- We can *splice* abstract syntax into our program using `$( )`

Haskell code

```
five :: Int
five = $(absId) 5
```

- The code is type checked before lifting and after splicing

Generic Programming

- A generic function is parameterised by the structure of a datatype.

Pseudo code

```
sum<T> :: T -> Int
```

- Each datatype has a corresponding structure type

Pseudo code

```
data List  = Nil | Cons Int List
type ListS = 1 + Int * List
```

- and the generic function is defined by recursion on the structure type

Pseudo code

```
sum<a + b> (Left x)  = sum<a> x
sum<a + b> (Right y) = sum<b> y
sum<a * b> (x,y)     = sum<a> x + sum<b> y
sum<1>     ()        = 0
sum<Int>   n         = n
```

Structure types

The structure types exist on two levels

- Available for matching on in generic functions (+, ∗, 1, `Int`)
- As Haskell types used in the definition of the generic function

Pseudo code

```
type a + b = Either a b
type a * b = (a,b)
type 1     = ()
type Int   = Int
```

From structure types to datatypes

- We can convert between datatypes and structure types using `inn` and `out`.

Pseudo code

```
inn<List> :: ListS -> List
out<List> :: List -> ListS
```

- This allows us to define

Pseudo code

```
sum<List> :: List -> Int
sum<List> = sum<ListS> . out<List>
```

- Two approaches
  - Force the user to do this (today, PolyP)
  - Have the compiler do it (Generic Haskell)

Implementation

- A generic function is a function from a type structure to abstract syntax

Haskell code

```
sumS :: Struct -> Q Exp
```

- Type structures are modelled by a datatype

Haskell code

```
data Struct = Struct :+: Struct
            | Struct :*: Struct
            | Unit
            | TypeCon String
```

- We need a bit more though

Haskell code

```
type Name        = String
type Arity       = Int
type Constructor = (Name, Arity)
type Datatype    = (Name, [Constructor], Struct)
```

Implementation

- We need to construct `Datatype`s somehow

Haskell code

```
datatype :: Q Dec -> Datatype

listD = datatype (reifyDecl List)
```

- `inn` and `out` need to know constructor names and arities

Haskell code

```
inn, out :: [Constructor] -> Q Exp
```

ghci interaction

```
> printExp (inn (constructors listD))
\xs -> case xs of
          Left ()       -> []
          Right (x,xs) -> x : xs
```

Defining Generic Functions

- On structure types:

Haskell code

```
sumS :: Struct -> Q Exp
sumS s = case s of
    a :+: b        ->
        [| \z -> case z of
                    Left x  -> $(sumS a) x
                    Right y -> $(sumS b) y |]
    a :*: b        ->
        [| \ (x,y) -> $(sumS a) x + $(sumS b) y |]
    Unit           -> [| \ () -> 0 |]
    TypeCon "Int" -> [| id |]
    TypeCon t      -> varE (gName "sum" t)
```

- The function `gName g t` produces a suitable name for the generic function named `g` instantiated at the type named `t`.

Defining Generic Functions

- On datatypes:

Haskell code

```
sumD :: Datatype -> Q Exp
sumD (_,cons,s) = [| $(sumS s) . $(out cons) |]
```

- We also need to know the name of the generic function when instantiating

Haskell code

```
type Generic = (Name, Datatype -> Q Exp)

sum :: Generic
sum = ("sum", sumD)
```

Instantiating Generic Functions

- Instantiation generates a function declaration

Haskell code

```
instantiate :: Generic -> Datatype -> Q Dec
instantiate (gname, gfun) s@(tname, _, _) =
    funD (gName gname tname)
         [ clause [] ( normalB $ gfun s ) [] ]
```

ghci interaction

```
> printDec (instantiate sum listD)
sum__List = (\z -> case z of
        Left x  -> (\ () -> 0) x
        Right y -> (\ (x,y) -> id x + sum__List y) y
    ) . (\x -> case x of
              Nil       -> Left ()
              Cons x xs -> Right (x,xs) )
```

The paper

- In the paper
  - A general method for implementing generic programming in Template Haskell
  - Prototype implementations of PolyP and (a large subset of) Generic Haskell
  - Code available at `http://www.cs.chalmers.se/~ulfn`

Future and ongoing work

- Ongoing work
  - Optimizer, based on the ideas of the next speaker
  - Experimenting with the various design choices
- Future work
  - Implement other styles of generic programming (Boilerplate, Strafunski)

Conclusions

- Template Haskell gives you
  - Abstract syntax (`Exp`,`Dec`,..)
  - Parser (`[| |]`)
  - Pretty printer (`$( )`)
  - Fresh name generation (`Q`)
  - Smooth interaction with GHC
- But it doesn't give you
  - Type checking
  - Access to the entire program

Conclusions

- Great for experimenting with generic programming
  - Generic Haskell implementation $\approx$ 800 lines of code
- Not ready to replace *the real thing* (yet)
  - No dedicated syntax
  - No automatic instantiation
  - Not so nice error messages