Template Haskell ~~& Lenses~~

Advanced functional programming - Lecture 1

~~Wouter Swierstra~~

Lawrence Chonavel

Template Haskell

There is a language extension, *Template Haskell*, that provides metaprogramming support for Haskell.

This defines support for *quotating* code into data and *splicing* generated code back into your program.

Template Haskell

```haskell
import Language.Haskell.TH

three :: Int
three = 1 + 2

threeQ :: ExpQ
threeQ = [| 1 + 2 |]
```

Rather than Racket's `quote` function, we can enclose expressions in quotation brackets `[| ... |]`.

This turns code into a quoted expression, `ExprQ`.

Unquoting

```
> three
3
> $threeQ
3
```

We can *splice* an expression `e` back into our program by writing `$e` (note the lack of space between `$` and `e`)

This runs the associated metaprogram and replaces the occurrence `$e` with its result.

Inspecting code

What happens when we quote an expression?

Inspecting code

What happens when we quote an expression?

```
> runQ [| 1 + 2 |]
InfixE (Just (LitE (IntegerL 1)))
       (VarE GHC.Num.+)
       (Just (LitE (IntegerL 2)))
```

Template Haskell defines a data type `Exp` corresponding to Haskell expressions.

The type `ExpQ` is a synonym for `Q Exp` – a value of type `Exp` in the quotation monad `Q`.

The `runQ` function returns the quoted expression associated with `ExpQ`.

The Exp data type

```
data Exp =
  VarE Name -- variables
  | ConE Name   -- constructors
  | LitE Lit    -- literals such as 5 or 'c'
  | AppE Exp Exp -- application
  | ParensE Exp -- parentheses
  | LamE [Pat] Exp -- lambdas
  | TupE [Exp] -- tuples
  | LetE [Dec] Exp -- let
  | CondE Exp Exp Exp -- if-then-else
  ...
```

Not to mention unboxed tuples, record updates, record construction, lists, list comprehensions,...

Template Haskell: programming with expressions

```
incr : Int → Int
incr x = x + 1

incrE : Exp → Exp
incrE e = AppE (VarE 'incr) e)
```

- The first `incr` function increments a number;
- The second takes a quoted expression as argument and builds a new expression by passing its argument to `incr`.
- We can ‘quote’ variable names with the prefix quotation mark `'incr`

Template Haskell: programming with expressions

```
-- Let x = [| 1 + 2 |]
x : Exp
x = InfixE (Just (LitE (IntegerL 1)))...

y : Int
y = $(incrE x)
```

Question: What is the result of evaulating **y**?

Template Haskell: programming with expressions

```
-- Let x = [| 1 + 2 |]
x : Exp
x = InfixE (Just (LitE (IntegerL 1)))...

y : Int
y = $(incrE x)
```

Question: What is the result of evaulating **y**?

But this might go wrong...

Typing Template Haskell

What happens when we create ill-typed expressions?

```
-- Let x = [| "Hello world" |]
x : Exp
x = LitE (StringL "Hello World")

y : Int
y = $(incrE x)
```

Question: What is the result of this program?

Typing Template Haskell

What happens when we create ill-typed expressions?

```
-- Let x = [| "Hello world" |]
x : Exp
x = LitE (StringL "Hello World")

y : Int
y = $(incrE x)
```

Question: What is the result of this program?

We get a type error:

```
No instance for (Num [Char]) arising from a use of ‘incr’
In the expression: incr "Hello World"...
```

Typing Template Haskell

Template Haskell is a *staged* programming language.

The compiler starts by type checking the program – even the program fragments building up expressions:

```
x : Exp
x = LitE (StringL "Hello World")
```

But it ignores any splices:

```
y : Int
y = $(incrE x)
```

It does not yet know if ~~y~~ the splice is type correct or not...

Typing Template Haskell

Template Haskell is a *staged* programming language.

The compiler starts by type checking the program – even the program fragments building up expressions:

```
x : Exp
x = LitE (StringL "Hello World")
```

But it ignores any splices:

```
y : Int
y = $(incrE x)
```

It does not yet know if ~~`y`~~ the splice is type correct or not...

Template Haskell is a *staged* programming language.

Then it (safely) evaluates the meta-program

```
x : Exp
x = LitE (StringL "Hello World")
```

```
y : Int
y = $(incrE x)
```

Template Haskell is a *staged* programming language.

Then it (safely) evaluates the meta-program

```
x : Exp
x = LitE (StringL "Hello World")
```

```
y : Int
y = $
```

(AppE (VarE 'incr) (LitE (StringL "Hello World")))

Typing Template Haskell

Template Haskell is a *staged* programming language.

Then it un-quotes ('splices') the expression

```
x : Exp
x = LitE (StringL "Hello World")
```

```
y : Int
y = incr "Hello World"
```

Typing Template Haskell

Template Haskell is a *staged* programming language.

Then it type-checks the expression

```
x : Exp
x = LitE (StringL "Hello World")
```

```
y : Int
y = incr "Hello World"
```

Typing Template Haskell

Template Haskell is a *staged* programming language.

```
This expression was ill-typed
```

```
x : Exp
x = LitE (StringL "Hello World")
```

```
y : Int
y = incr "Hello World"
```

Typing Template Haskell

Template Haskell is a *staged* programming language.

This expression was ill-typed

```
x : Exp
x = LitE (StringL "Hello World")
```

```
y : Int
y = incr "Hello World"
```

Q: are 2 passes enough?

Template Haskell is a *staged* programming language.

This expression was ill-typed

```
x : Exp
x = LitE (StringL "Hello World")
```

```
y : Int
y = incr "Hello World"
```

Q: are 2 passes enough?

A: no, un-quoted code could contian more splices...

Type safety

Question: So is Template Haskell statically typed?

Type safety

Question: So is Template Haskell statically typed?

Yes: all generated code is type checked.

No: metaprograms are essentially untyped.

Type safe metaprograms

Template Haskell also provides a data type for 'typed expressions':

```
newtype TExp a = TExp { unType :: Exp }
```

The type variable is not used, but *tags* an expression with the type we expect it to have. This is sometimes known as a *phantom type*.

This can be used to give us some more type safety:

```
appE :: TExp (a → b) → TExp a → TExp b
appE (TExp f) (TExp x) = TExp (AppE f x)
```

Question: Where does this break?

Limited safety...

There are lots of ways to break this. Referring to variables is one way:

```
bogus :: TExp String
bogus = TExp (VarE 'incr)
```

But more generally, there are plenty of situations where we cannot easily figure out the types of the metaprogram that we generate.

Beyond expressions

The `Exp` data type is used to reflect expressions.

But Template Haskell also provides data types describing:

- Patterns
- Function definitions
- Data type declarations
- Class declarations
- Instance definitions
- Compiler pragmas
- ...

Together with the technology to reflect code into such data types.

Beyond expressions

As a result, we can generate *arbitrary* code fragments using Template Haskell:

- new type signatures;
- new data type declarations;
- new classes or class instances;
- ...

Any pattern in our code that we can describe programmatically can be automated throught Template Haskell.

Template Haskell: examples

There are several ways in which people use Template Haskell:

- Generalizing a certain pattern of functions:

```
zip :: [a] → [b] → [(a,b)]
zip3 :: [a] → [b] → [c] → [(a,b,c)]
zip4 :: [a] → [b] → [c] → [d] → [(a,b,c,d)]
...
```

- Automating boilerplate code (lenses);
- Including system information (`git-embed`).
- Interfacing safely to an external data source, such as a database, requires computing new marshalling/unmarshalling functions (`printf`).

Example: git-embed

Suppose that we want to include version information about our program.

```
> mytool --version
Built from the branch 'master' with hash 410d5264a
```

We could do this in numerous ways:

- maintain a `version` variable in our code manually;
- have a shell script that generates this information whenever we release code;
- splice this information into our code using Template Haskell.

Example: git-embed

There is a library using Template Haskell, git-embed, that provides precisely this functionality.

Using it is easy enough:

```
import Git.Embed

gitRev :: String
gitRev = $(embedGitShortRevision)

gitBranch :: String
gitBranch = $(embedGitBranch)
```

How is it implemented?

Example: git-embed ideas

Functions such as `embedGitBranch` need to compute a string, corresponding to the current git branch.

To do so:

1. We perform a bit of I/O, running `git branch` with suitable arguments;
2. The result of this command contains the information that we are after.
3. Quoting this result back into a string literal, yields the desired value.

Note: we can run `IO` computations while metaprogramming…

Example: git-embed implementation

```
embedGitBranch : ExpQ
embedGitBranch = embedGit
  ["rev-parse", "--abbrev-ref", "HEAD"]

embedGit :: [String] → ExpQ
embedGit args = do
    addRefDependentFiles
    gitOut <- runIO (readProcess "git" args "")
    return $ LitE (StringL gitOut)
```

The **addRefDependentFiles** adds the files from the .git directory as dependencies. If these files change, the module will be recompiled.

Quotation and I/O

This example illustrates that we can run I/O operations during quotation.

Question: Why should this be allowed? And what are the drawbacks?

Quotation and I/O

This example illustrates that we can run I/O operations during quotation.

Question: Why should this be allowed? And what are the drawbacks?

- Makes it possible to read data from a file, network, database, etc. – and use this information to generate new code or write new data to a file.
- The compiling code may have side effects! You can write a Haskell program that formats your hard-drive *when compiled*.

Example: printf

If you've ever done any debugging with C, you will have encountered **`printf`**:

```
char* userName;
x = ... ;
y = ... ;
printf("x,y, and user are now:")
printf("x=%d,y=%d,user=%s",x,y,userName);
```

The **`printf`** function takes a *variable* number of arguments: depending on the format string, it expects a different number of integers and strings.

What is its type?

Example: printf using Template Haskell

We'll sketch how to implement a `printf` function in Haskell:

```
> $(printf "x=%d,s=%s") 6 "Hello"
"x=6,s=Hello"
```

The key idea is to use the argument string to compute a function taking suitable arguments.

Splicing `$(printf "x=%d,s=%s")` will compute the term:

```
\n0 → \s1 → "x=" ++ show n0 ++ ",s=" ++ s1
```

Example: printf

```
printf :: String → ExpQ
printf s = gen (parse s)

data Format = D | S | L String

parse :: String → [Format]

gen :: [Format] → ExpQ
```

- The `printf` function maps a string to an expression;
- The `parse` function reads in the string and splits it into a series of format instructions – it doesn't use any Template Haskell and I won't cover it further.
- Depending on these instructions, the `gen` command will compute a different expression.

Example: printf

Let's try to figure out how the `gen` function works in several different steps.

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

gen :: [Format] → ExpQ
gen [] = LitE (StringL "")
```

If the list of formatting directives is empty, we compute the empty string – that was easy enough.

Example: printf

Now suppose we only ever have to worry about handling a single formatting directive:

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

gen :: [Format] → ExpQ
gen [D] = [| \n → show n |]
gen [S] = [| \s → s |]
gen [L str] = LitE (StringL str)
```

Each individual case generates the code that we would write by hand otherwise.

Example: printf

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

gen :: [Format] → Exp → Exp
gen [] e = e
gen (D:fmts) e =
  [| \n→ $(gen fmts [| $e ++ show n |]) |]
gen (S:fmts) e =
  [| \s→ $(gen fmts [| $e ++ s |]) |]
gen (L s:fmts) e = gen fmts [| $e ++ $(LitE (StringL s)) |]
```

The **gen** function is defined using an *accumulating parameter*. Initially, this is just the empty string.

Example: printf

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

gen :: [Format] → Exp → Exp
gen [] e = e
gen (D:fmts) e =
  [| \n→ $(gen fmts [| $e ++ show n |]) |]
gen (S:fmts) e =
  [| \s→ $(gen fmts [| $e ++ s |]) |]
gen (L s:fmts) e = gen fmts [| $e ++ $(LitE (StringL s)) |]
```

As we encounter more formatting directives, we add an additional lambda if necessary and perform a recursive call.

Note the subtle interplay between splicing and quoting.

Example: printf

This example shows how to compute *new expressions* from existing data.

This same pattern pops up whenever we want to interface with an external data source, such as database:

- Request information about the table layout;
- Parse the result and generate corresponding types;
- Generate functions to access the data.

Quasiquotation

As a last example, I want to briefly mention *quasiquotation*.

We've seen how to embed domain specific languages in Haskell using deep/shallow embeddings.

When we do so, we are constrained by Haskell's syntax and static semantics.

Racket shows how to use macros to write custom language dialects.

Haskell's *quasiquotation* framework borrows these ideas.

Quasiquotation: example

Suppose I'm writing a Haskell library for manipulating and generating C code.

But working with C ASTs directly is pretty painful:

```
add n =
   Func (DeclSpec [ ] [ ] (Tint Nothing))
   (Id "add")
     DeclRoot
     (Args
        [Arg (Just (Id "x"))
      ...
```

What I'd like to do is embed (a fragment of) C in my Haskell library.

Using quasiquotation

The quasiquoter allows me to do just that:

```
add n = [cfun |
  int add (int x )
  {
    return x + $int : n$;
  }
  |]
```

The `cfun` quasiquoter tells me how to turn a string into a suitable `Exp`.

Defining quasiquoters

A quasiquoter is nothing more than a series of parsers for expressions, patterns, types and declarations:

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

Whenever the Haskell parser encounters a quasiquotation `[ myQQ | ... |]` it will run the parser associated with the quasiquoter `myQQ` to generate the quoted expression/pattern/type/declaration.

Multiline strings

(built-in since GHC 9.12)

As a simple example, suppose we want to have multi-line string literals.

We can define a quasiquoter:

```
ml :: QuasiQuoter
ml = QuasiQuoter
  { quoteExp  = (\a →  LitE (StringL a)), ... }
```

And call it as follows:

```
example : String
example = [ml |
hello
beautiful
world|]
```

Quasiquoting

The quasiquoting mechanism allows you to embed arbitrary syntax within your Haskell program.

And still use Template Haskell's quotation and splicing to mix your object language with Haskell code.

This is a mix of the embedded and stand-alone approaches to domain specific languages that we saw over the last few weeks.

Drawbacks of metaprogramming

Metaprogramming has many applications, but several crucial drawbacks:

- Template Haskell AST and ‘real’ AST are often out of step.
- AST is much more complex than S-expressions – the Template Haskell library is huge!
- Debugging type errors in generated code is hard.
- Q monad allows arbitrary IO *during compilation* – which may be a security risk.
- Large computations can slow down compile times.

*** Cross-compilation becomes (even) harder**