Metaprogramming Tutorial: OCaml and Template Haskell

Jake Donham and Nicolas Pouillard

jake@donham.org nicolas.pouillard@gmail.com

CUFP 2010, Baltimore

Static metaprogramming

What is it?

- compile-time code generation
- transformation of syntax trees

What's it good for?

- for convenience (generate boilerplate or type-derived functions)
- for speed (generate first-order functions from higher-order templates)
- for EDSLs (embedded regexps or SQL)
- for language extensions

Static metaprogramming in OCaml and Haskell

Camlp4: preprocessing front-end to the OCaml compiler, AST transformations written as Camlp4 plugins

Template Haskell: compiler extensions to GHC, AST transformations embedded in Haskell code

Small example: map over a tuple

- Avoid boilerplate of mapping over elements of a tuple
- in OCaml,

  ```
  Tuple.map f (a, b, c, d)
  ```

  is transformed to

  ```
  (f a, f b, f c, f d)
  ```

- in Haskell,

  ```
  import qualified Data.Tuple.TH as T
  $(T.map 4) f (a,b,c,d)
  ```

  is transformed to

  ```
  (f a, f b, f c, f d)
  ```

Camlp4

ASTs in Camlp4:

```
type expr = ... and patt = ... and
ctyp = ... and str_item = ... and ...
```

-
- e.g. `ExInt` for an int expr, `TySum` for a sum type
- see `Camlp4Ast.partial.ml` for full def
- somewhat loose—easy to make invalid AST
- converted to OCaml AST; see `Camlp4Ast2OCamlAst.ml` for errors

OCaml quotations:

- a way to work with the AST using concrete syntax
- you can always fall back to AST constructors!
- e.g. `<:expr< 1, 2 >>` becomes

```
ExTup (_, (ExCom (_, (ExInt (_, "1")),
                     (ExInt (_, "2")))))
```

- `<:ctyp< int * int >>` becomes

```
TyTup (_, (TySta (_, (TyId (_, (IdLid (_, "int")))),
                     (TyId (_, (IdLid (_, "int"))))))))
```

- antiquotations: `<:expr< 1, $x$ >>`, `<:expr< 1, $‘int:x$ >>`
- see doc page of quotations / antiquotations

Working with the AST:

`Ast.map`

- object that maps over AST
- method for each syntactic class
- override to operate on AST

locations, `Ast.Loc.t`

- stores filename and position
- must provide one to construct AST nodes
- quotations use `_loc` by default
- `Loc.ghost`
- `Ast.loc_of_expr e`
- `<:expr@_loc< >>`
- `<:expr@here< >>`

Revised syntax:

- alternative concrete syntax for OCaml
- fixes some infelicities in OCaml syntax
- makes antiquotation easier (gives more context, bugs with original syntax)
- `list t` instead of `t list`
- `match [ patt -> expr | ... ]` instead of `match patt -> expr | ...`
- `True` instead of `true`
- see doc page for full details

Running Camlp4:

- `camlp4of [module.cmo]* [file.ml]`
- show loaded modules: `-loaded-modules`
- print original syntax: `-printer o`
- show AST for debugging: `-filter Camlp4AstLifter`
- take input from command line: `-str [input]`
- Ex.
```
camlp4of -printer o -filter Camlp4AstLifter \
    -str 'type t = Foo'
```

Debugging:

- don't know what AST to use?
- run example through camlp4of to see what AST is parsed
- quotations / antiquotations don't work?
- read parsers `:(` to see why (`Camlp4OCamlRevisedParser.ml`, `Camlp4OCamlParser.ml`)
- fall back to AST constructors
- errors converting Camlp4 to OCaml AST?
- read converter to see why (`Camlp4Ast2OCamlAst.ml`)
- use `-filter Camlp4AstLifter` to see what you're generating

Template Haskell

ASTs in Template Haskell

TH exposes data-types for expressions, patterns, declarations, types... (Exp, Pat, Dec, Type).

```
exE :: Exp
exE = ListE [_42, VarE 'succ `AppE` _42]
  where _42 = LitE (IntegerL 42)
```

Smart constructors, new names, and the Q monad

TH also exposes smart constructors for all constructors, to build programs in the Q monad.

```
apE :: ExpQ
apE = do x <- qNewName "x"
         y <- qNewName "y"
         lamE [varP x, varP y] (varE x `appE` varE y)
```

Generic quotations in Template Haskell

TH has a general mechanism for quotations.

```
[$sql| SELECT * FROM ‘users‘ |]
```

```
[$regex| (a|b)*b*(a|b)* |]
```

```
[$xml| <person><name>Foo</name><age>42</age></person> |]
```

Haskell quotations in TH

TH has a general mechanism for quotations.

```
[e| \f g x -> f (g x) |]
```

```
[t| Int -> (Bool, Char) |]
```

```
[d| data Foo = A | B | C |]
```

... and antiquotions for those

Using `$(...)` one can splice expressions, types... into one other.

```
[e| case $(a) of { [] -> $(b) ; x:xs -> $(c) x xs } |]
```

```
[t| Int -> ($(t), Char) |]
```

```
[d| data Foo = A | B $(t) | C |]
```

Exercises

Tuple map

Implement the tuple map syntax from the example.

Zipper types

The "zipper" representation of a value of type `t` is a subtree of type `t`, and a context of type `t'`, where `t'` is derived systematically from `t`. (see Huet)

A zipper type has:

- a `Top` arm
- for each arm containing `t`, an arm for each occurrence of `t` with that occurrence replaced with `t'`

For example:

```
type t = Leaf | Tree of t * t
```

has zipper type

```
type t' = Top | Tree0 of t' * t | Tree1 of t * t'
```

Implement a generator for zipper types.

Implementing quotations/antiquotations in Camlp4

Quotations are implemented in several phases:

- quotation is lexed to a `QUOTATION` token containing tag and body as strings
- expander for tag is looked up according to parse context (e.g. `expr` vs. `patt`)
- expander parses string to quotation AST with `FooAnt` nodes for antiquotations, containing tag and body
- expander lifts quotation AST to Camlp4 AST according to parse context
- expander parses antiquotation nodes as OCaml and applies conversions according to tag

JSON quotations in OCaml

We can define quotations for JSON:

```
<:json< [ 1, 2, 3 ] >>
<:json< { "foo" : true, "bar" : 17 } >>
```

And antiquotations:

```
<:json< [ 1, $int:x$, 3 ] >>
<:json< { "foo" : $bool:b$, "bar" : 17  } >>
<:json< [ 1, $list:y$, 3 ] >>
```

JSON quotations in Haskell

We can define quotations for JSON:

```
[$json| [ 1, 2, 3 ] |]
[$json| { "foo" : true, "bar" : 17 } |]
```

And antiquotations:

```
[$json| [ 1, $(js x), 3 ] |]
[$json| { "foo" : $(js b), "bar" : 17  } |]
[$json| [ 1, $(js y), 3 ] |]
```

Implement JSON quotations and antiquotations.