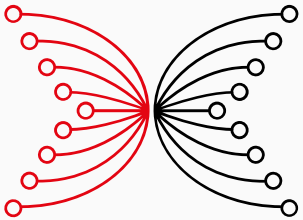
Template Haskell

Haskell and Cryptocurrencies

Dr. Lars Brünjes, IOHK
Alejandro Garcia, IOHK
Dr. Andres Löh, Well-Typed LLP

2020-09-20



INPUT | OUTPUT

Goals

- Introduce Template Haskell.
- Typed and untyped metaprogramming.
- Quotes and splices.
- Stage restrictions.

The “Hello World” of meta-programming

The `power` function

A somewhat clever function for computing the `n`-th power of `x`:

```
power :: Int -> Int -> Int
power n x
   | n <= 0      = 1
   | even n      = let
                     r = power (n `div` 2) x
                   in
                     r * r
   | otherwise = x * power (n - 1) x
```

Examples

```
GHCi>  power 5 2
 32
GHCi>  power 10 2
 1024
```

Performance

Consider:

```haskell
power10 :: Int -> Int
power10 x =
  let
    p10 = p5 * p5
    p5  = x * p2 * p2
    p2  = x * x
  in
    p10
```

Performance

Consider:

```
power10 :: Int -> Int
power10 x =
  let
    p10 = p5 * p5
    p5  = x * p2 * p2
    p2  = x * x
  in
    p10
```

Which of the two is more efficient?

```
power 10 42
power10 42
```

Benchmark

```haskell
module Main where

import Gauge.Main  -- or Criterion.Main

main :: IO ()
main =
  defaultMain
    [ bench "power 10 42" $ nf (power 10) 42
    , bench "power10 42"  $ nf power10    42
    , bench "pow 10 42"   $ nf (pow 10)   42
    ]

pow :: Int -> Int -> Int
pow = (^)

 ...
```

Results

- The unrolled version is significantly faster than `power`.
- Even just using the built-in operator is somewhat faster than `power`.

Situation

- We have an algorithm for better exponentation if we “know” the first argument.
- We can apply the algorithm manually to obtain better performance.
- But trying to implement the algorithm leads to worse performance.

Why?

Situation

- We have an algorithm for better exponentation if we “know” the first argument.
- We can apply the algorithm manually to obtain better performance.
- But trying to implement the algorithm leads to worse performance.

Why?

Because GHC does not unroll recursive functions, so the static knowledge of the first argument to `power` is not exploited.

(Typed) Template Haskell

Template Haskell

A metaprogramming extension for Haskell:

- Introduced by Tim Sheard and Simon Peyton Jones in the Haskell Workshop 2002 paper “Template Meta-programming for Haskell”.
- Extended with quasiquoting by Geoffrey Mainland in the Haskell Symposium 2007 paper “Why It’s Nice to be Quoted: Quasiquoting for Haskell”.
- Extended with typed quotes and splices by Geoffrey Mainland in 2013.

Goals of Template Haskell

- Give the programmer more fine-grained control over what happens at compile time and what at run time.
- Allow to eliminate boilerplate code by generating code algorithmically at compile time.

Typed quotes

The quote construct build a “piece of syntax”:

- Rather than to evaluate the quoted code, we just build a representation of the code (a syntax tree).
- A typed quote is still typechecked. We cannot build representations of ill-typed or ill-scoped code.

Example

```haskell
exp1 :: Int
exp1 = 1 + 1  -- represents the integer 2
```

Example

```haskell
exp1 :: Int
exp1 = 1 + 1  -- represents the integer 2
```

```haskell
exp2 :: Q (TExp Int)
exp2 = [|| 1 + 1 ||]  -- represents the code 1 + 1
```

Example

```
exp1 :: Int
exp1 = 1 + 1  -- represents the integer 2
```

```
exp2 :: Q (TExp Int)
exp2 = [|| 1 + 1 ||]  -- represents the code 1 + 1
```

You can read `Q` as “quoted”, and `TExp Int` as “typed expression of type `Int`”.

But for the time being, we can better view these two as a unit:

```
type Code a = Q (TExp a)
```

Splices

A *splice* can be used to insert a piece of code into another.

The syntax tree that is being spliced is grafted into the syntax tree of the quote where the splice occurs.

Typed splices are again type-checked.

Example

```
exp3 :: Code Int
exp3 = [|| $$exp2 * $$exp2 ||]
```

Represents `(1 + 1) * (1 + 1)`.

Note: We are building abstract syntax, not concrete syntax – even though `*` binds stronger than `+`, a splice is always implicitly grouped.

Example

```
exp3 :: Code Int
exp3 = [|| $$exp2 * $$exp2 ||]
```

Represents `(1 + 1) * (1 + 1)`.

Note: We are building abstract syntax, not concrete syntax – even though `*` binds stronger than `+`, a splice is always implicitly grouped.

```
exp4 :: Code Int
exp4 = [|| let x = $$exp2 in x * x ||]
```

Represents `let x = 1 + 1 in x * x`.

Parameterised quotes

```
mul :: Code Int -> Code Int
mul x = [|| $$x + $$x ||]
```

Parameterised quotes

```
mul :: Code Int -> Code Int
mul x = [|| $$x + $$x ||]
```

```
exp5 :: Code Int
exp5 = mul [|| 1 + 1 ||]
```

Represents `(1 + 1) * (1 + 1)`.

Let-binding to avoid duplication

```
mul' :: Code Int -> Code Int
mul' x = [|| let y = $$x in y * y ||]
```

Let-binding to avoid duplication

```
mul' :: Code Int -> Code Int
mul' x = [|| let y = $$x in y * y ||]
```

```
exp6 :: Code Int
exp6 = mul' [|| 1 + 1 ||]
```

Represents `let y = 1 + 1 in y * y`.

Local names hygiene

Recall:

```
mul' :: Code Int -> Code Int
mul' x = [|| let y = $$x in y * y ||]
```

```
exp7 :: Code Int
exp7 = mul' (mul' [|| 1 + 1 ||])
```

Represents:

```
let y1 = (let y2 = 1 + 1 in y2 * y2) in y1 * y1
```

The machinery takes care of avoiding name capture.

Staging the power function

Staged code

The process of rewriting programs with quotes and splices to make use of statically known arguments and to control the generated code is called staging.

We usually work with two stages:

- Everything that is statically known is represented using normal Haskell types and will ultimately run at compile time.
- Everything that is only dynamically known is represented using `Code` types and will ultimately run at run time.

In principle, more than two stages can exist, but for our purposes, two are sufficient.

Simplified power

The following function just unrolls the multiplication:

```
simplepower :: Int -> Code Int -> Code Int
simplepower n x
   | n <= 0     = [|| 1 ||]
   | otherwise =
    [|| $$x * $$(simplepower (n - 1) x) ||]
```

Note: We have to use parentheses if we want to splice a compound expression.

Staged power

```
spower :: Int -> Code Int -> Code Int
spower n x
  | n <= 0      = [|| 1 ||]
  | even n      = [|| let
                      r = $$(spower (n `div` 2) x)
                    in
                      r * r ||]
  | otherwise = [|| $$x * $$(spower (n - 1) x) ||]
```

Actually generating code

A missing piece

We currently can:

- construct pieces of code (syntax trees) via quoting,
- insert pieces of code into other pieces of code via splicing.

But we cannot actually do anything with the pieces of code.

Top-level splices

A top-level splice (i.e., a splice that does not occur within a quote) generates the given piece of code at compile time and inserts it into the program that is just being compiled.

Top-level splices

A top-level splice (i.e., a splice that does not occur within a quote) generates the given piece of code at compile time and inserts it into the program that is just being compiled.

Stage restriction for top-level splices

Definitions referred to from top-level splices must be defined in a different module and imported from there.

Invoking staged functions

Examples:

```
simplerpower10 :: Int -> Int
simplerpower10 x =
  $$(simplepower 10 [|| x ||])

spower10 :: Int -> Int
spower10 x =
  $$(spower 10 [|| x ||])
```

Debugging the generated code

It is often useful to see the generated code.

We can tell GHC to show it during compilation using the `-ddump-splices` flag.

```
  simplepower 10 [|| x_a7di ||]
======>
  (x_a7di
     * (x_a7di
          * (x_a7di
               * (x_a7di
                    * (x_a7di
                         * (x_a7di * (x_a7di * (x_a7di * (x_a7di * (x_a7di * 1))))))))))

  spower 10 [|| x_a7dg ||]
======>
  let
    r_a7gv
      = (x_a7dg
           * (let r_a7gw = let r_a7gx = (x_a7dg * 1) in (r_a7gx * r_a7gx)
              in (r_a7gw * r_a7gw)))
  in (r_a7gv * r_a7gv)
```

Revisiting the benchmark

It is easy to add `spower10` to our performance comparison, and unsurprisingly, the result is that it is identical in performance to `power10`.

- In this case, we see that the spliced code for `spower10` is very nearly equivalent to the hand-written code for `power10`.
- Generated code is still subject to GHC optimisation afterwards, therefore simple artifacts of code generation (such as the extra multiplication by `1` here) do usually not have to be removed.

Comparing staged and unstaged power function

```
power :: Int -> Int -> Int
power n x
  | n <= 0     = 1
  | even n     = let
                   r = power (n `div` 2) x
                 in
                   r * r
  | otherwise = x * power (n - 1) x
```

```
spower :: Int -> Code Int -> Code Int
spower n x
  | n <= 0     = [|| 1 ||]
  | even n     = [|| let
                       r = $$(spower (n `div` 2) x)
                     in
                       r * r ||]
  | otherwise = [|| $$x * $$(spower (n - 1) x) ||]
```

Observations

- Removing quotes and splices (and `Code` in the types) yields the original code.
- In general, removing quotes and splices should yield equivalent unstaged code.
- However, in some cases we must transform the code more substantially in order to obtain a good staged version.

Intermediate Summary

Using Typed Template Haskell, we can write functions that can use statically known information to produce good code at compile time.

We can use this to define abstractions at no run-time cost that otherwise would have one.

Exercises

1. Could we define `spower` to have type `Int -> Code (Int -> Int)` instead? This would make the top-level splice easier.
2. In general, are `Code (a -> b)` and `Code a -> Code b` equivalent?

Another example / exercise

Consider `lookup` on binary search trees:

```
data BST a =
    Node Int a (BST a) (BST a)
  | Leaf

lookup :: Int -> BST a -> Maybe a
lookup _ Leaf            = Nothing
lookup i (Node j a l r) =
  case compare i j of
    LT -> lookup i l
    EQ -> Just a
    GT -> lookup i r
```

Stages, lifting, and cross-stage persistence

Example

What about the following code?

```
strange :: Code (Code Int -> Int)
strange = [|| \x -> 1 + $$x ||]
```

Example

What about the following code?

```
strange :: Code (Code Int -> Int)
strange = [|| \x -> 1 + $$x ||]
```

We get a stage error: "`x` is bound at stage `2` but used at stage `1`".

The problem is:

- The `x` will be bound to a concrete value only at run time.
- But we need to have that value available at compile time, because we use it in a splice.

Counting stages

The compiler considers the top-level of a module to be stage `1`.

- Every quote increases the stage by `1`.
- Every splice increases the stage by `1`.

Counting stages

The compiler considers the top-level of a module to be stage `1`.

- Every quote increases the stage by `1`.
- Every splice increases the stage by `1`.

A variable used at the same stage as it is bound is fine (that is the normal situation).

Counting stages

The compiler considers the top-level of a module to be stage `1`.

- Every quote increases the stage by `1`.
- Every splice increases the stage by `1`.

A variable used at the same stage as it is bound is fine (that is the normal situation).

A variable used at a stage before it is bound is an error (as just observed).

Counting stages

The compiler considers the top-level of a module to be stage `1`.

- Every quote increases the stage by `1`.
- Every splice increases the stage by `1`.

A variable used at the same stage as it is bound is fine (that is the normal situation).

A variable used at a stage before it is bound is an error (as just observed).

What about a variable used at a later stage than it is bound?

Example

What about this code?

```
persist :: Int -> Code Int
persist x = [|| 3 * x ||]
```

Example

What about this code?

```
persist :: Int -> Code Int
persist x = [|| 3 * x ||]
```

Here, `x` is bound at stage `1` but used at stage `2`.

If stage `1` runs at compile time, then we need to build code representing `x` at run time.

For an `Int`, this isn't a problem: evaluate the `Int` and insert the result as a literal.

Lifting

```
persist (1 + 1) :: Code Int
```

Represents `3 * 2`, and not `3 * (1 + 1)`!

Lifting

```
persist (1 + 1) :: Code Int
```

Represents `3 * 2`, and not `3 * (1 + 1)`!

In general, this requires some form of serialisation, which is governed by the type class `Lift`:

```
persist' :: Lift a => a -> Code a
persist' x = [|| x ||]
```

This would fail without the `Lift` constraint.

The `Lift` class

```haskell
class Lift a where
  lift :: a -> Q Exp
  liftTyped :: a -> Q (TExp a)  -- or: a -> Code a
```

This is somewhat similar to `Show`.

The `DeriveLift` language extension enables deriving this automatically for most types.

But “as usual”, types such as functions or `IO` are not an instance of `Lift`.

Lifting explicitly

```
persist' :: Lift a => a -> Code a
persist' x = [|| 3 * x ||]
```

This can be rewritten to use `liftTyped` explicitly:

```
persist'' :: Lift a => a -> Code a
persist'' x = [|| 3 * $$(liftTyped x) ||]
```

Now the stage where `x` is bound and the stage where `x` is used are the same!

A hand-written `Lift` instance

```
data BST a =
    Node Int a (BST a) (BST a)
  | Leaf
```

```
instance Lift a => Lift (BST a) where
  liftTyped (Node i x l r) = [|| Node i x l r ||]
  liftTyped Leaf           = [|| Leaf ||]
```

Here, we use implicit lifting on all four arguments of `Node`!

Cross-stage persistence

We can always invoke top-level definitions in splices.

These are considered to be cross-stage persistent, i.e., they are available at any stage.

Exercise

Define a static memoisation function:

```
smemo :: [Int] -> (Int -> Int) -> Code (Int -> Int)
              -> Code (Int -> Int)
```

The idea is that `smemo xs sf f` calls `sf` statically on all `xs`, stores the results in some static data structure, and generates code that, given an argument `x`, first tries to find the `x` in the static memo table, and otherwise executes `f` dynamically.

Here, `f` and `sf` are supposed to refer to the same function. We are working around the fact that functions do not have a `Lift` instance.

Untyped Template Haskell

Going past the typed world

- Sometimes, we might want to generate code fragments that cannot easily be typed in isolation.
- Or what if we want to generate patterns, or declarations, or types?

For such purposes, there are untyped quotes and splices.

Untyped quotes

Recall:

```
exp1 :: Int
exp1 = 1 + 1  -- represents the integer 2
```

Untyped quotes

Recall:

```
exp1 :: Int
exp1 = 1 + 1  -- represents the integer 2
```

```
exp2 :: Q (TExp Int)
exp2 = [|| 1 + 1 ||]  -- represents the code 1 + 1
```

Untyped quotes

Recall:

```
exp1 :: Int
exp1 = 1 + 1  -- represents the integer 2
```

```
exp2 :: Q (TExp Int)
exp2 = [|| 1 + 1 ||]  -- represents the code 1 + 1
```

```
exp2' :: Q Exp
exp2' = [| 1 + 1 |]  -- also represents the code 1 + 1
```

Untyped quotes always have the type `Q Exp`.

Type-incorrect quotes

There is nothing preventing us from building type-incorrect quotes:

```
te :: Q Exp
te = [| head True |]  -- represents the code head True
```

Type-incorrect quotes

There is nothing preventing us from building type-incorrect quotes:

```
te :: Q Exp
te = [| head True |]  -- represents the code head True
```

However, top-level splices of untyped code are still type checked, so if `te` is every top-level spliced, it will still yield an error then.

Untyped splices

Untyped splices work much like typed splices. The spliced code must just be of type `Q Exp` and can be spliced anywhere within another quote.

Untyped splices

Untyped splices work much like typed splices. The spliced code must just be of type `Q Exp` and can be spliced anywhere within another quote.

Untyped top-level splices look the same.

Untyped splices

Untyped splices work much like typed splices. The spliced code must just be of type `Q Exp` and can be spliced anywhere within another quote.

Untyped top-level splices look the same.

All staging restrictions for untyped quotes and splices are the same as for their typed versions.

Quoting other syntactic categories

In untyped Template Haskell, we cannot quote just expressions

...

Patterns:

```
[p| (True, _) |] :: Q Pat
```

Declarations:

```
[d| foo x = x + 1 |] :: Q [Dec]
```

Types:

```
[t| Int -> Int |] :: Q Type
```

Splicing other syntactic categories

Splices can also appear in places where patterns or types are expected.

However, declaration splices are only allowed at the top-level. Declaration top-level splices can be written as naked expressions. This is the form most commonly seen (e.g. for deriving optics).

Building syntax directly

Sometimes, we need to step outside the syntax we have available in quotes:

- what if we want to build a tuple of flexible size?
- what if we want to build a `let` with a flexible number of declarations?
- ...

For this, we have the entire Haskell abstract syntax available.

Example abstract syntax for patterns

```
data Pat =
    LitP Lit                    -- literals
  | VarP Name                   -- variables
  | TupP [Pat]                  -- tuples
  | ConP Name [Pat]             -- constructor applications
  | InfixP Pat Name Pat         -- infix constructors
  | TildeP Pat                  -- irrefutable patterns
  | BangP Pat                   -- strict patterns
  | AsP Name Pat                -- as-patterns
  | WildP                       -- wildcard pattern (underscore)
  | RecP Name [FieldPat]        -- record syntax patterns
  | ListP [Pat]                 -- literal list patterns
  | SigP Pat Type               -- pattern type signatures
  |  ...
```

Smart constructors

There are type synonyms and smart constructors for moving everything into the `Q` type:

```
type PatQ = Q Pat

litP :: Lit -> PatQ
varP :: Name -> PatQ
tupP :: [PatQ] -> PatQ
conP :: Name -> [PatQ] -> PatQ
 ...
```

Examples

```haskell
onesExp :: Int -> ExpQ
onesExp n = tupE (replicate n [| 1 |])

onesPat :: Int -> PatQ
onesPat n = tupP (replicate n [p| 1 |])
```

Why `Q`?

- It is a monad.
- It allows generating fresh names (this has to be done manually once we construct binders directly, and not via quotes).
- It allows to reify information about existing Haskell entities, getting hold of information on how they are defined.
- It allows performing `IO` actions (which means we can run arbitrary `IO` at compile time, which is quite controversial).

There are also ways to construct `Name`s from existing Haskell entities.

A glimpse at `reify` and `Info`

```
reify :: Name -> Q Info

data Info =
    ClassI Dec [InstanceDec]
  | ClassOpI Name Type ParentName
  | TyConI Dec
  | FamilyI Dec [InstanceDec]
  | PrimTyConI Name Arity Unlifted
  | DataConI Name Type ParentName
  | PatSynI Name PatSynType
  | VarI Name Type (Maybe Dec)
  | TyVarI Name Type
```

More examples

Demo.

Summary / Outlook

- Typed Template Haskell allows to get more control over evaluation order and can be used to write powerful high-level abstractions that still generate efficient code.
- Untyped Template Haskell is extremely powerful and allows to generate code for various tasks. It is, however, difficult to debug and to maintain, and its power can be abused.
- The quotation/splicing mechanisms that are used by Template Haskell can be further extended via *quasiquotes*. These allow a user-defined syntax to be used in place of various syntactic categories in Haskell.