Overloaded Quotes for Template Haskell

Michael D. Adams

January 5, 2012

Caveat 1: I've only briefly skimmed TMPH. I think I understand the type system well enough to comment on how to change them, but I could easily be missing some detail.

Caveat 2: The following discussion only considers quotes and splices for expressions. Quotes and splices for types, declarations and patterns are ignored, but are a simple generalization of these ideas.

1 Changes to Figure 2 of "Template Meta-programming for Haskell"[1]

Figure 2 is unchanged except for the following definitions and rules:

- STATES: $s \subseteq C, B_\tau, S$
- BRACKET: $\dfrac{\Gamma \vdash^{n+1}_{B_m} e : \tau \qquad \text{Quasi } m}{\Gamma \vdash^{n}_{C,S} [|e|] : m\,\text{Exp}}$
- ESCB: $\dfrac{\Gamma \vdash^{n-1}_{S} e : m\,\text{Exp}}{\Gamma \vdash^{n}_{B_m} \$e : \tau}$

In the BRACKET rule, the notation "Quasi m" means that m must be an instance of the Quasi class.

2 Explanation

The system from "Template Meta-programming for Haskell" (TMPH) is largely unchanged. Conceptually the only difference is that the *Bracket* state is no longer a simple B but becomes B_m and carries a type, m, that determines the expected type for splices directly within that bracket. Specifically, if in state B_m then any splices inside it are expected to be of type $m\,\text{Exp}$. Compare this to TMPH where splices are of type $\text{Q}\,\text{Exp}$.

[1] Obtained from https://research.microsoft.com/en-us/um/people/simonpj/papers/meta-haskell/meta-haskell.pdf on January 5, 2011

As with TMPH there is only one way to enter state B_m and that is by the quote form. In TMPH all quotations are of type Q Exp, but we generalize this to allow quotations to have type m Exp for any m that is an instance of Quasi.

Splices from the Code (C) state (i.e. top level splices) still expect their body to be of type Q Exp just as in TMPH. This is so that it can run in the compiler's type-checker monad.

Nested splices, however, do not need run in the compiler's type-checker monad they simply run in whatever monad the containing bracket uses.

3 Example

(This code hasn't been checked by a compiler so it may contain typos.)

```
$(let nextInt :: StateT Int Q Exp = do c <- get
                                       put (c+1)
                                       return (LitE (IntegerL c))
      foo = [| $(nextInt) + $(nextInt) |]
      bar = runStateT foo 1
      baz = [| $(bar) + 5 |]
  in baz)
```

Here "foo" has type "StateT Int Q Exp", "bar" has type "Q Exp" and "baz" has type "Q Exp". Notice that "foo" does not have to be capable to running in the compiler's type-checker monad as long as the final result of the top-level splice is of type "Q Exp". In this code, that is done via "runStateT".

4 Desugaring

Conceptually a quote form (e.g. [| ...a... $(...b...) ...c... $(...d...) ...e...|]) behaves as a do-block (e.g. do b' <- b; d' <- d; return (...a... b' ...c... d' ...e...)) where each nested splice is run by the do-block and final statement of the do-block constructs the expression using the results of those splices.