Coercion Quantification

(Extended Abstract)

Ningning Xie
The University of Hong Kong, China
nnxie@cs.hku.hk

Richard A. Eisenberg
Bryn Mawr College, USA
rae@cs.brynmawr.edu

Dependent Haskell [3, 4, 7] has been desired in the community of Haskell programmers for a long time. However, compatibility with existing GHC features makes adding full-fledged dependent types into GHC very difficult. Thus, our goal of this project [1] is to make the core language of Haskell, known as System F_C [2, 5, 6, 8], dependently typed, as steps are taken towards dependent Haskell.

To this end, the first step we take is to embrace *homogeneous equality*, which means equality is between types of a same kind. Homogeneous equality simplifies meta-theory. More importantly, it enables us to prove the important lemma, *congruence*, for the dependently typed core. Adopting homogeneous equality is not straightforward. It requires us to make the type of primitive equality, ~#, homogeneous, and requires patches to the constraint solver.

This is a working-in-progress project. We are at the very beginning of the stage. As a small step towards our final goal, the focus of this talk is on *coercion quantification*. To understand the motivation, consider if we had homogeneous ~#, and in the source Haskell, we still want to provide programmers the ability to use heterogeneous equalities, then we define the heterogeneous equality, ~~, based on the homogeneous equality[1]:

```
data (~~) :: forall k1 k2. k1 -> k2 -> Constraint where
  MkHEq :: forall k1 k2 (a :: k1) (b :: k2).
    (k1 ~# k2) -> (a ~# b) -> a ~~ b
```

However this is definitely wrong because a ~# b is ill-kinded, as ~# is homogeneous! To correct it, we need to give the coercion a name, and use it to fix the kind:

```
data (~~) :: forall k1 k2. k1 -> k2 -> Constraint where
  MkHEq :: forall k1 k2 (a :: k1) (b :: k2).
    forall (co :: k1 ~# k2).  -- a name here...
      (a |> co ~# b) ->       -- and a cast here
      a ~~ b
```

Coercion quantification is interesting as:

[1]Note here *data* means a *dictionary*, the representation of a *type class* in core.

HIW'18, Sept. 23, 2018, St. Louis, MO, USA

1) For people working in core, the patch to core formalization is worth attention. Adding coercion quantification means now polymorphic quantifications (over both types and coercions) could have a coercion in their bindings. Refactor of those basic types has a significant impact to files in the compilation pipeline and introduces several subtleties involving binders, substitutions, representations of datatypes, etc.

2) For Haskell users, coercion quantification opens up new questions to the design space in *source* Haskell. For example, is the type of fun well-formed in source Haskell?

```
data SameKind :: k -> k -> *
fun :: forall k1 k2 (a::k1) (b::k2).
              (k1 ~ k2) => SameKind a b
```

In core, we now have the ability to name the coercion k1 ~ k2 and use it to cast a. But accepting the code in the source level requires the solver to be smart enough to generate a name and insert the cast. This requires non-trivial extension of the solver and we would want Haskell users to answer if this feature is ever desired in their development.

To sum up, in this talk, we would like to share the high-level story-line of the dependently typed core, our low-level progress in implementing coercion quantification, as well as the involving design space, and seek feedbacks from the broader community.

References

[1] 2018. Implementing Dependent Haskell, Phase 2. (2018). https://ghc.haskell.org/trac/ghc/wiki/DependentHaskell/Phase2
[2] Joachim Breitner, Richard A Eisenberg, Simon Peyton Jones, and Stephanie Weirich. 2016. Safe zero-cost coercions for Haskell. *Journal of Functional Programming* 26 (2016).
[3] Richard A. Eisenberg. 2016. *Dependent Types in Haskell: Theory and Practice.* Ph.D. Dissertation. University of Pennsylvania.
[4] Adam Michael Gundry. 2013. *Type Inference, Haskell and Dependent Types.* Ph.D. Dissertation. University of Strathclyde.
[5] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with Type Equality Coercions *(TLDI '07).*
[6] Stephanie Weirich, Justin Hsu, and Richard A Eisenberg. System FC with explicit kind equality *(ICFP '13).*
[7] Stephanie Weirich, Antoine Voizard, Pedro Henrique Avezedo de Amorim, and Richard A. Eisenberg. A Specification for Dependent Types in Haskell *(ICFP'17).*
[8] Brent A Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion *(TLDI '12).*