Literature Review of GHC Core

Ningning Xie
xnningxie@gmail.com

June 25, 2018

1 Introduction

This document aims at giving an rough overview of the evolution of GHC Core, System FC, through publications. The motivation of this document is to aid developers who hack into GHC Core in gaining a theoretical understanding of each design choice involved in the type system.

Note that this document is not supposed to be a stand-alone literature; that is, it is impossible to understand all type systems solely by reading this document. Instead, it is supposed to be read along with the papers. It gives a summary of each type system's motivation, and highlights points that are important or different from previous type systems, which is expected to help the process of paper reading.

Assumed background: Types and Programming Languages (Pierce and Benjamin, 2002).

2 System FC

System F with Type Equality Coercions, TLDI'07 (Sulzmann et al., 2007).

2.1 Motivation

Language designers have begun to experiment with a variety of type systems that are difficult or impossible to translate into System F, such as functional dependencies, generalized algebraic data types (GADTs), and associated types.

This paper presents System FC, which extends System F with 1) explicit equality witnesses; 2) non-parametric type functions.

2.2 Notes

- The role of coercion in typing:

$$\frac{\Gamma \vdash_{\mathrm{e}} e : \sigma_1 \qquad \Gamma \vdash_{\mathrm{CO}} \gamma : \sigma_1 \sim \sigma_2}{\Gamma \vdash_{\mathrm{e}} e \blacktriangleright \gamma : \sigma_2} \; \textsc{fc-typing-Cast}$$

- The systems merges types and coercions.
  - Types have judgment $\Gamma \vdash_{\mathrm{TY}} \sigma : \kappa$
  - Coercions have judgment $\Gamma \vdash_{\mathrm{CO}} \gamma : \sigma_1 \sim \sigma_2$. Homogeneous.
- Kinds $\kappa ::= * \mid \kappa_1 \to \kappa_2 \mid \sigma_1 \sim \sigma_2$.
- Sorts $\delta ::= \mathrm{TY} \mid \mathrm{CO}$.
  - TY for kind $*$ and $\kappa_1 \to \kappa_2$;
  - CO for $\sigma_1 \sim \sigma_2$ and $\gamma_1 \sim \gamma_2$.
- Coercions γ are types, $\sigma_1 \sim \sigma_2$ are kinds, CO are sorts. $\gamma :: \sigma_1 \sim \sigma_2 :: \mathrm{CO}$.
- The meaning of type function is given by axioms.
- Type functions are required to be saturated.

3 System $F_C^{\uparrow}$

Giving Haskell a Promotion, TLDI'12 (Yorgey et al., 2012).

3.1 Motivation

The kind system in Haskell is too 1) permissive: type-level programming in Haskell is almost entirely untyped, because the type system has too few kinds ($*, * \to *$, and so on); 2) restrictive: It lacks polymorphism.

This paper presents System $F_C^{\uparrow}$, which extends System FC with

- Automatic promotion of datatypes to be kinds and data constructors to be types.

  ```
  data Nat = Zero | Succ Nat
  data Vec :: * -> Nat -> * where
    VNil  :: Vec a 'Zero
    VCons :: a -> Vec a n -> Vec a ('Succ n)
  ```

  Type `Nat` is used as a kind, and data constructors `Zero` and `Succ` are used as types, with a quote notation to avoid ambiguity.

- Kind polymorphism, for kinds, types, and terms.

  ```
  data EqRefl a b where
    Refl :: EqRefl a a
  ```

  Previously, `EqRefl`$:: * \to * \to *$, with kind polymorphism we have `EqRefl`$:: \forall \mathcal{X}. \mathcal{X} \to \mathcal{X} \to *$

3.2 Notes

- The formalization distinguish expressions, types, coercions and kinds, but in implementation they are *combined*.
- Expressions now include kind abstraction and kind application.
- Kinds $\kappa ::= * \mid \kappa_1 \to \kappa_2 \mid \boxed{\text{Constraint} \mid \mathcal{X} \mid \forall \mathcal{X}.\kappa \mid \text{T}\,\overline{\kappa}}$, where T is promoted type constant.
- Only one sort $\Gamma \vdash_{\text{k}} \kappa : \Box$
- Types $\sigma ::= \ldots \boxed{\mid \text{K} \mid \forall \mathcal{X}.\sigma \mid \sigma\,\kappa \mid \sim}$, where K is promoted data constructors, and $\sim$ is equality.
- Important rules for promotion:

  $$\frac{\text{K} : \sigma \in \Gamma \qquad \emptyset \vdash \sigma \rightsquigarrow \kappa}{\Gamma \vdash_{\text{TY}} \text{K} : \kappa}\ \textsc{fc-kind-KLift}$$

  In this rule, a data constructor K is treated as a type and has a kinding rule. $\emptyset \vdash \sigma \rightsquigarrow \kappa$ turns a type into a kind.

  $$\frac{\Gamma \vdash_{\text{k}} \kappa_1 : \Box \quad .. \quad \Gamma \vdash_{\text{k}} \kappa_n : \Box \qquad \emptyset \vdash_{\text{TY}} \text{T} : *^n \to *}{\Gamma \vdash_{\text{k}} \text{T}\,\overline{\kappa} : \Box}\ \textsc{kind-valid-KV-Lift}$$

  In this rule, a type constructor T is treated as a kind constructor. This rule is relatively restrictive since the type of T takes all arguments of kind $*$ and it needs to be fully saturated.
- System $F_C^{\uparrow}$ turns the equality from System FC into a type constructor with polymorphic kind:

  $$\frac{}{\Gamma \vdash_{\text{TY}} \sim : \forall \mathcal{X}.\mathcal{X} \to \mathcal{X} \to \text{Constraint}}\ \textsc{fc-kind-KEq}$$

- Coercions are homogeneous, having type $\sigma_1 \sim \sigma_2$, which has kind Constraint.
- Design principle: no kind equalities.

4 Deferred Type Errors

Equality Proofs and Deferred Type Errors, ICFP'12 (Vytiniotis et al., 2012).

4.1 Motivation

Based on System $F_C^{\uparrow}$, the coercion type $\sigma_1 \sim \sigma_2$ is now an ordinary type. Therefore, we can have ordinary values of this type, and the value can be passed to or returned from arbitrary terms. This proofs-as-values approach opens up an entirely new possibility, that of deferring type errors to runtime.

```
foo = (True, 'a' && False)

foo = let (c : Char ~ Bool) = error 'Cound't ...'
      in (True, (cast 'a' c) && False)
```

Here we manually define an evidence $c : \mathit{Char} \sim \mathit{Bool}$ which actually emits an error, which can be used to type check the program and defer the error to runtime.

4.2 Notes

- The original *erasable* type constructor $\sim$ is renamed to $\sim_{\#}$, and the kind Constraint is renamed to Constraint$_{\#}$.
- There are two kinds of coercions
  - $\sim_{\#}$, the type for primitive coercions γ. Erasable.
  - $\sim$, the type of evidence generated by the type inference engine. Cannot be erased. Defined as a GADT

    ```
    data a ~ b where
      Eq# :: (a ~# b) -> a ~ b

    ~ : forall X. X -> X -> *
    Eq# : forall X. forall (a : X). forall (b : X). (a ~# b) -> (a ~ b)
    ```

- Then we can define the function `cast`. Each use of cast forces evaluation of the coercion, via the `case` expression, which in the case of a deferred type error, triggers the runtime failure.

  ```
  cast : forall (a b : *). a -> (a ~ b) -> b
  cast = ∧(a b : *). \(x:a). \(eq:a ~ b).
    case eq of
      Eq# (c: a ~# b) -> x |> c
  ```

- The relation between $\sim_{\#}$ and $\sim$ is analogous to that between **int** and **int**$_{\#}$. Refer to Jones and Launchbury (1991) for more details.
- How it works: during constraint generation, we generate a type-equality constraint even for unifications that are *unsolvable*. We emit a warning, and create a value binding for the constraint valuable, which binds it to a call to error, applied to the error message string.
- The optimization uses wrapper, and re-boxing, so that most equality evidences can be optimized away.

5 Explicit Kind Equality

System FC with Explicit Kind Equality, ICFP'13 (Weirich et al., 2013).

5.1 Motivation

System FC lacks kind equality proofs, as mentioned in Section 3. This paper presents an approach based on dependent type systems with heterogeneous equality and the *Type-in-Type* axiom, yet it preserves the metatheoretic properties of FC.

It enables

- Kind-indexed GADT: the datatype is indexed by both kind and type information.

```
data TyRep :: \/ k. k -> * where
  TyInt    :: TyRep Int
  Ty Bool  :: TyRep Bool
  TyMaybe  :: TyRep Maybe
  TyApp    :: TyRep a -> TyRep b -> TyRep (a b)

zero :: \/ (a : *). TyRep a -> a
zero TyInt             = 0
zero TyBool            = False
zero (TyApp TyMaybe _) = Nothing
```

- Promoted GADT: GADT are allowed to be used an as index.

```
data Kind = Star | Arr Kind Kind

data Ty :: Kind -> * where
  TInt   :: Ty Star
  TBool  :: Ty Star
  TMaybe :: Ty (Arr Star Star)
  TApp   :: Ty (Arr k1 k2) -> Ty k1 -> Ty k2

data TyRep (k :: Kind) (t :: Ty k) where
  TyInt   :: TyRep Star TInt
  TyBool  :: TyRep Star TBool
  TyMaybe :: TyRep (Arr Star Star) TMaybe
  TyApp   :: TyRep (Arr k1 k2) a -> TyRep k1 b -> TyRep k2 (TApp a b)
```

- Kind family

```
kind family IK (k :: Kind)
kind instance IK Star = *
kind instance IK (Arr k1 k2) = IK k1 -> IK k2
```

5.2 Notes

- In this work, the syntax of types and kinds are unified, allowing us to reuse type coercions as kind coercions, with axiom $* : *$.
  $\sigma, \kappa ::= \forall a : \kappa.\sigma \mid \ldots \mid \forall c : \phi.\sigma \mid \sigma \blacktriangleright \gamma \mid \sigma\,\gamma$
  The type of coercion ϕ is now separated from types. Namely coercion abstractions are separated from arrow types.
  $\phi ::= \sigma_1 \sim \sigma_2$
- Equalities are now *heterogeneous*. In the definition of type equalities $\sigma_1 \sim \sigma_2$, the type σ_1 and σ_2 could have kinds κ_1 and κ_2 that have no syntactic relation to each other. A proof γ of $\sigma_1 \sim \sigma_2$ implies not only that σ_1 and σ_2 are equal, but also that their kinds are equal.
- Coercions are irrelevant to both the operational semantics and type equivalence.
- Important kinding rule:

  $$\frac{\Gamma \vdash_{\mathrm{TY}} \sigma : \kappa_1 \qquad \Gamma \vdash_{\mathrm{CO}} \gamma : \kappa_1 \sim \kappa_2 \qquad \Gamma \vdash_{\mathrm{TY}} \kappa_2 : *}{\Gamma \vdash_{\mathrm{TY}} \sigma \blacktriangleright \gamma : \kappa_2}\ \textsc{fc-kind-KCast}$$

  Important coercion coherence rule:

  $$\frac{\Gamma \vdash_{\mathrm{CO}} \gamma : \sigma_1 \sim \sigma_2 \qquad \Gamma \vdash_{\mathrm{TY}} \sigma_1 \blacktriangleright \gamma' : \kappa}{\Gamma \vdash_{\mathrm{CO}} \gamma \blacktriangleright \gamma' : \sigma_1 \blacktriangleright \gamma' \sim \sigma_2}\ \textsc{fc-co-CT-COH}$$

  The use of kind coercions can be ignored when proving type equalities.

  $$\frac{\Gamma \vdash_{\mathrm{CO}} \gamma : \sigma_1 \sim \sigma_2 \qquad \Gamma \vdash_{\mathrm{TY}} \sigma_1 : \kappa_1 \qquad \Gamma \vdash_{\mathrm{TY}} \sigma_2 : \kappa_2}{\Gamma \vdash_{\mathrm{CO}} \mathbf{kind}\,\gamma : \kappa_1 \sim \kappa_2}\ \textsc{fc-co-CT-EXT}$$

  The new coercion form $\mathbf{kind}\,\gamma$ extracts the proof of $\kappa_1 \sim \kappa_2$ from γ.
- A tricky `S_KPUSH` rule.

6 Safe Zero-cost Coercions for Newtypes

Safe Zero-cost Coercions for Haskell, ICFP'14 (Breitner et al., 2014), JFP'16 (Breitner et al., 2016).

This work itself is built on Generative Type Abstraction and Type-level Computation, POPL'11 (Weirich et al., 2011).

6.1 Motivation

Consider to define a newtype `HTML`

```
newtype HTML = MkHTML String

unHTML :: HTML -> String
unHTML (MkHTML s) = s

linesH :: HTML -> [HTML]
linesH h = map MkHTML (lines (unHTML h))
```

The runtime cost of `linesH` is inevitable. To avoid that, this paper describes safe coercions, with the constraint `Coercible` and the function

```
coerce :: Coercible a b => a -> b
```

Then `String -> [String]` and `HTML -> [HTML]` would be coercible and the function can be rewritted to

```
linesH :: HTML -> [HTML]
linesH = coerce lines
```

`Coercible` is translated into System FC, augmented with *roles*.

6.2 Notes

- `Coercible s t`: `s` and `t` have bit-for-bit identical run-time representation.
- For every type constructor, each type parameter has a role, determined by the way in which the parameter is used in the definition of the type constructor.
  - representational: type parameters of ordinary newtypes and datatypes
  - phantom: it does not occur in the definition of the type, or it occurs only as a phantom parameter of another type constructor.
  - nominal: parameters that possibly affect the run-time representation of a type, including parameters of a type/data family, non-uniform parameters to GADTs, type classes.
    Parameters of type variables are always nominal.
  - Users can specify the roles via annotation, and the compiler ensures that role annotations cannot violate type safety.

    ```
    type role Map nominal representational
    ```

- To decide whether two types are coercible:

  - *unwrapping* rule: for every `newtype NT = MkNT t`,
    we have `Coercible t NT`.
    This rule is available only if the corresponding newtype data constructor is in scope.
  - *lifting* rule: for every type constructor `TC r p n`,
    if `Coercible r1 r2`,
    we have `Coercible (TC r1 p1 n) (TC r2 p2 n)`.
  - Coercible is an equivalence relation: reflexivity, symmetry, transitivity.
  - *decomposition* rule: given non-newtype `T`,
    if `Coercible (T r1 p1 n1) (T r2 p2 n2)`,
    then `Coercible r1 r2`, and `n1 ∼ n2`.
  - *type application* rule: If `Coercible t1 t2`, where `t1, t2 :: k1 -> k2`,
    then `Coercible (t1 x) (t2 x)`.

- The type system in the paper is more like the one in Section 3: types and kinds are not unified and coercions are homogeneous.

- New form of coercion: $\Gamma \vdash_{\mathrm{CO}} \gamma : \sigma_1 \sim_{\rho}^{\kappa} \sigma_2$
  - Nominal equality, written $\sim_{\mathrm{N}}$.
    * The equality that the source Haskell type checker reasons about.
    * Type families introduce new nominal equalities.
  - Representational equality, written $\sim_{\mathrm{R}}$.
    * The equality holds between two types that share the same runtime representation.
    * A subset of nominal equality.
    * A `Coercible` constraint corresponds to a proposition of representational equality.
    * New types introduce new representational roles.
  - Phantom equality, written $\sim_{\mathrm{P}}$, holds between any two types.

- Important type-safe cast:

$$\frac{\Gamma \vdash_{\mathrm{e}} e : \sigma_1 \qquad \Gamma \vdash_{\mathrm{CO}} \gamma : \sigma_1 \sim_{\mathrm{R}}^{\kappa} \sigma_2}{\Gamma \vdash_{\mathrm{e}} e \blacktriangleright \gamma : \sigma_2}\ \textsc{fc-typing-Cast-R}$$

  The coercion γ must be a proof of representational equalities.

7 Levity Polymorphism

Levity Polymorphism, PLDI'17 (Eisenberg and Peyton Jones, 2017).

7.1 Motivation

```
bTwice :: forall a. Bool -> a -> (a -> a) -> a
bTwice b x f = case b of True -> f (f x)
                         False -> x
```

Polymorphic function is supposed to work for any type of argument `x`. However, the type of `x` influences the calling convention, and hence the executable code. For example, for list `x`, the function would be passed in a register pointing into the heap; for a float `x`, it would be passed in a special floating-point register.

One simple but very slow solution: represent every value uniformly, as a pointer to a heap-allocated object.

Most languages also support *unboxed values* that are represented by the value itself rather than a pointer. Haskell classifies types by kinds. Lifted types have kind `Type`, while unlifted types have kind `#`.

Current Instantiation Principle: all polymorphic type variables have kind `Type`. However it introduces several problems. For example,

```
-> :: Type -> Type -> Type
```

means `Int# -> Int# -> Int#` is ill-typed. The current design is sub-kinding:

```
Type <: OpenKind
#    <: OpenKind
->   :: OpenKind -> OpenKind -> Type
```

which is awkward and unprincipled.

The idea of this paper is to *replace sub-kinding with kind polymorphism.* The main idea is

```
-- primitive
TYPE :: Rep -> Type

-- definitions
type Rep      = [UnaryRep] -- a type-level list
data UnaryRep = PtrRep     -- boxed, lifted
              | UPtrRep    -- boxed, unlifed
              | IntRep     -- unboxed ints
              | FloatRep   -- unboxed floats
              | DoubleRep  -- unboxed doubles
              | ...etc...
type Lifted   = '[PtrRep]
type Type     = TYPE Lifted
```

7.2 Notes

	boxed: represented by a pointer into the heap	**unboxed**: represented by the value itself
lifted: lazy; has one extra element beyond the usual ones representing a non-terminating computation (call by need)	`Int`, `Bool`	(Haskell represents lazy computation as thunks, so lifted can only be boxed)
unlifted: strict (call by value)	`ByteArray#`	`Int#`, `Char#`

- Any type that classifies values has kind `TYPE r` for some `r :: Rep`. The type `Rep` specifies how a value of that type is represented, by giving a list of `UnaryRep`. A `UnaryRep` specifies how a single values is represented. Examples:

```
Int    :: TYPE '[PrtRep], TYPE Lifted, Type
Int#   :: TYPE '[IntRep]
Float# :: TYPE '[FloatRep]
(Int, Bool)  :: Type
Maybe Int    :: Type
Maybe        :: Type -> Type
```

- Levity polymorphism: an abstraction over only the levity (lifted vs. unlifted) of a type.

```
(->) :: forall (r1 :: Rep) (r2 :: Rep).
        TYPE r1 -> TYPE r2 -> Type
```

- Restrict the use of levity polymorphism so that it can be compiled:
  - Disallow levity-polymorphic binders.
  - Disallow levity-polymorphic function arguments.
- The correctness of levity polymorphism is proved by 1) defining $\mathcal{L}$: a variant of System F that supports levity polymorphism 2) defining $\mathcal{M}$: a λ-calculus in A-normal form, with operational semantics working with an explicit stack and heap; 3) a type-erasing compilation from $\mathcal{L}$ to $\mathcal{M}$, with correctness proofs.

8 Implementation

System FC, as implemented in GHC, Technical Report (Eisenberg, 2015).

References

Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. Safe zero-cost coercions for haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14. ACM, 2014.

Joachim Breitner, Richard A Eisenberg, Simon Peyton Jones, and Stephanie Weirich. Safe zero-cost coercions for haskell. *Journal of Functional Programming*, 26, 2016.

Richard A Eisenberg. System fc, as implemented in ghc. 2015.

Richard A Eisenberg and Simon Peyton Jones. Levity polymorphism. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 525–539. ACM, 2017.

Simon L Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Conference on Functional Programming Languages and Computer Architecture*, pages 636–666. Springer, 1991.

Benjamin C Pierce and C Benjamin. *Types and programming languages.* MIT press, 2002.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, pages 53–66. ACM, 2007.

Dimitrios Vytiniotis, Simon Peyton Jones, and José Pedro Magalhães. Equality proofs and deferred type errors: A compiler pearl. *ACM SIGPLAN Notices*, 47(9):341–352, 2012.

Stephanie Weirich, Dimitrios Vytiniotis, Simon Peyton Jones, and Steve Zdancewic. Generative type abstraction and type-level computation. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 227–240. ACM, 2011.

Stephanie Weirich, Justin Hsu, and Richard A Eisenberg. System fc with explicit kind equality. In *ACM SIGPLAN Notices*, volume 48, pages 275–286. ACM, 2013.

Brent A Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pages 53–66. ACM, 2012.