Coercion Quantification

Ningning Xie [1] Richard A. Eisenberg [2]

22 Sept. 2018

Haskell Implementor's Workshop (HIW'18)

[1]The University of Hong Kong

[2]Bryn Mawr College

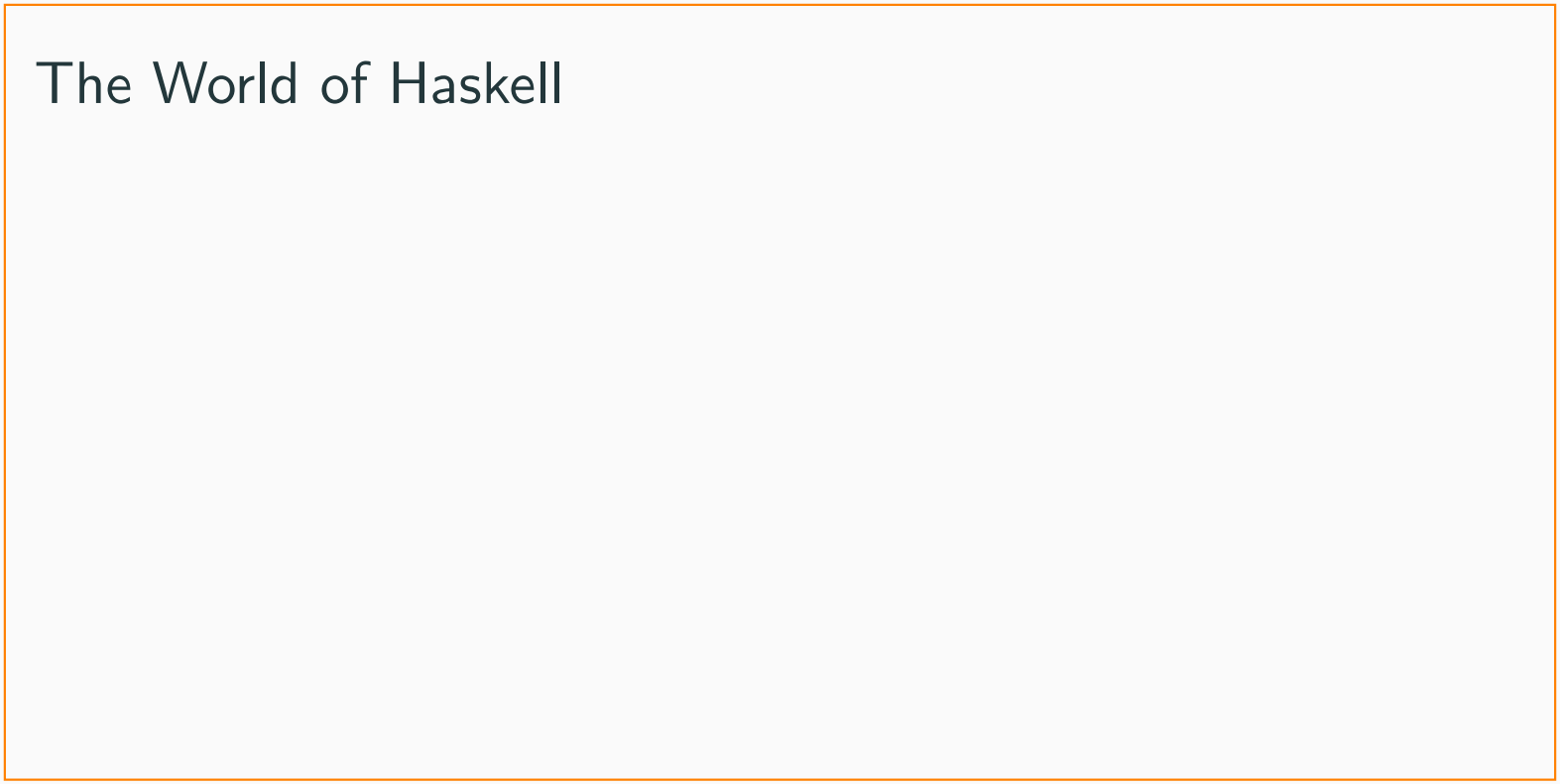
Motivation

Motivation

Our long-term goal is to have Dependent Haskell

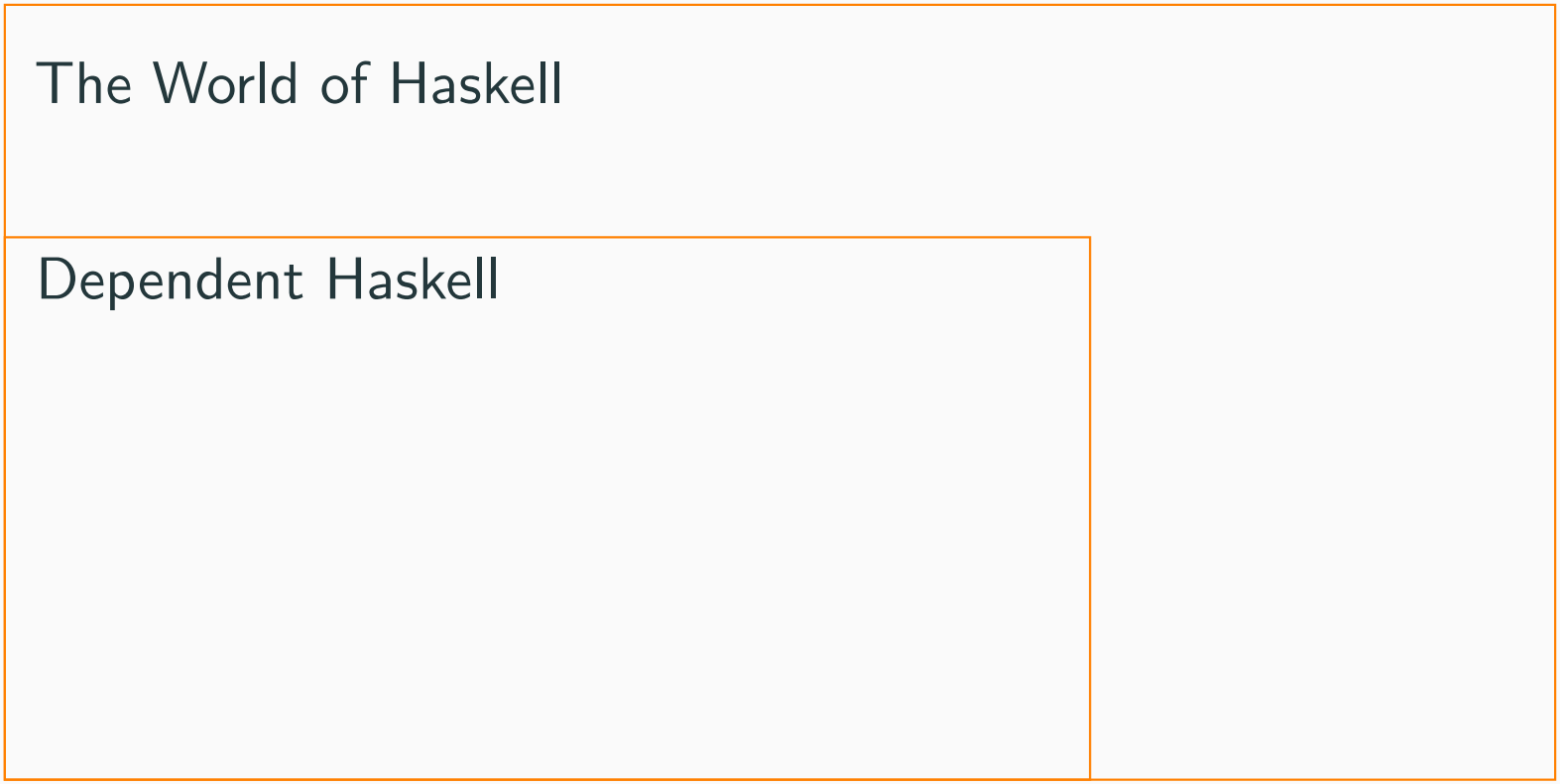
Motivation

Our long-term goal is to have Dependent Haskell



Motivation

Our long-term goal is to have Dependent Haskell

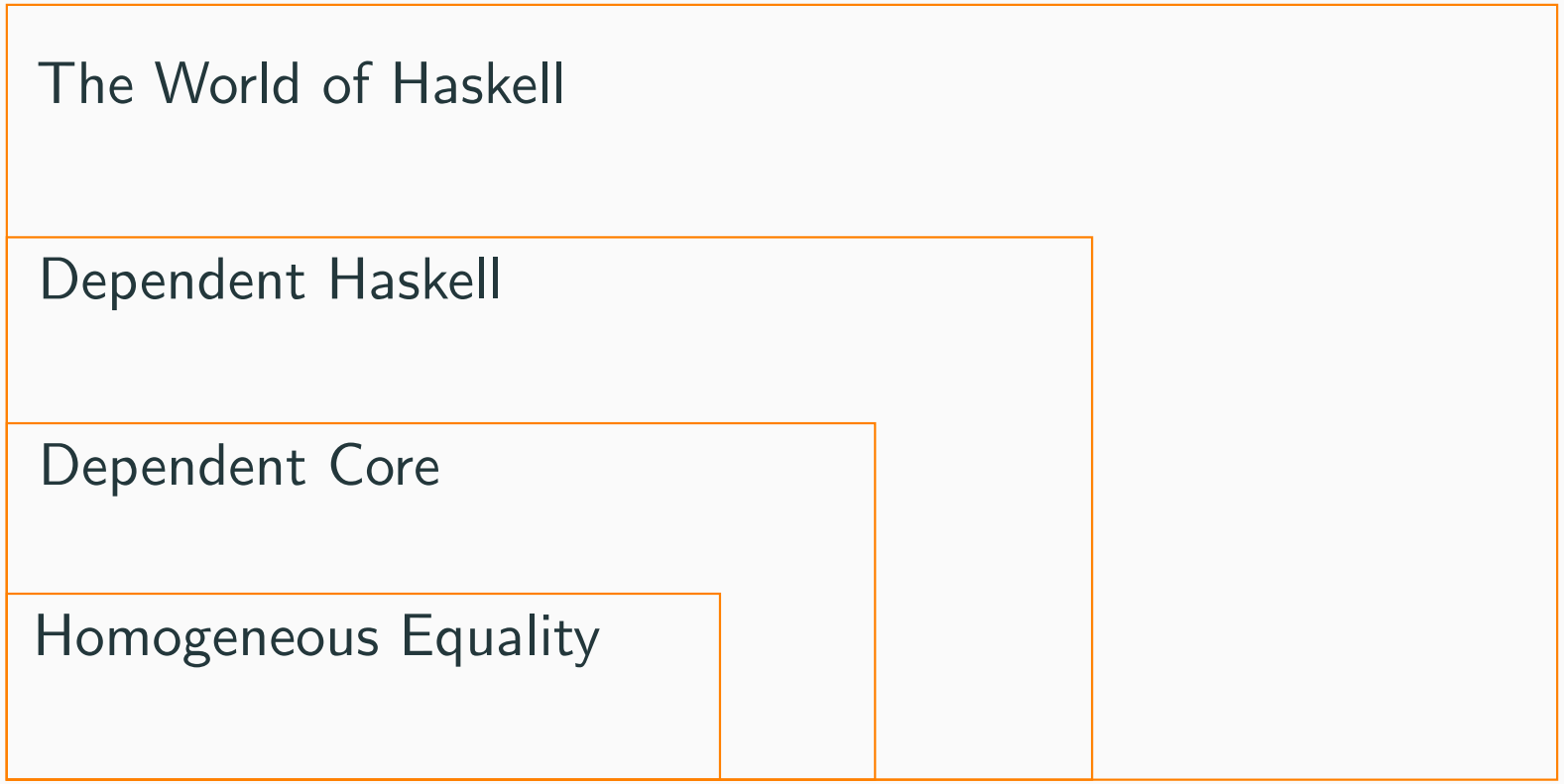


Motivation

Our long-term goal is to have Dependent Haskell

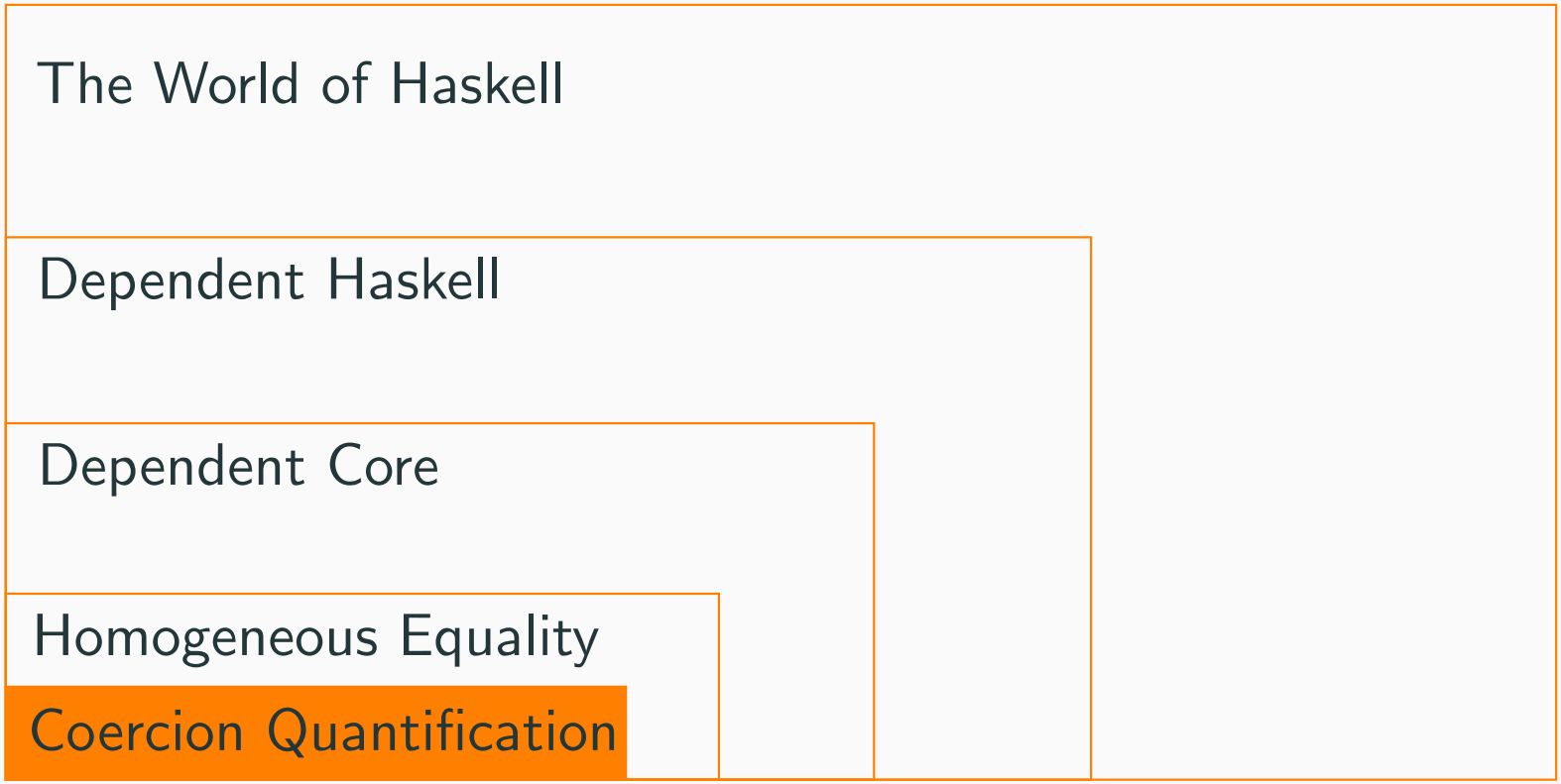
Motivation

Our long-term goal is to have Dependent Haskell



Motivation

Our long-term goal is to have Dependent Haskell



Outline

- From Haskell to Coercion Quantification
- Coercion Quantification
  - For Core Contributors: Theory and Implementation
  - For Haskell Users: Design Space in Source Haskell
- Future Work

From Haskell to Coercion Quantification

From Haskell to Coercion Quantification

- Why Dependent Types?

Why Dependent Types?

We want dependent types...Oh, do we?

- A language with dependent types may include references to programs inside of types.

Why Dependent Types?

We want dependent types...Oh, do we?

- A language with dependent types may include references to programs inside of types.
- Length-indexed vectors

```
data Nat = Zero | Succ Nat
data Vec :: Type → Nat → Type where
  Nil  :: Vec a Zero
  Cons :: a → Vec a n → Vec a (Succ n)
```

Why Dependent Types?

We want dependent types...Oh, do we?

- A language with dependent types may include references to programs inside of types.
- Length-indexed vectors

```
data Nat = Zero | Succ Nat
data Vec :: Type → Nat → Type where
  Nil  :: Vec a Zero
  Cons :: a → Vec a n → Vec a (Succ n)

-- vector of length 3
vec1 :: Vec Int (Succ (Succ (Succ Zero)))
vec1 = Cons 1 (Cons 2 (Cons 3 Nil))
```

Why Dependent Types?

We want dependent types...Oh, do we?

- A language with dependent types may include references to programs inside of types.
- Length-indexed vectors

```
data Nat = Zero | Succ Nat
data Vec :: Type → Nat → Type where
  Nil  :: Vec a Zero
  Cons :: a → Vec a n → Vec a (Succ n)

-- vector of length 3
vec1 :: Vec Int (Succ (Succ (Succ Zero)))
vec1 = Cons 1 (Cons 2 (Cons 3 Nil))

-- type error!
vec2 :: Vec Int (Succ (Succ Zero))
vec2 = Cons 1 (Cons 2 (Cons 3 Nil))
```

Why Dependent Types?

```
-- accepts only non-empty vector
vecHead :: Vec a (Succ n) → a
vecHead (Cons x xs) = x
```

Why Dependent Types?

```
-- accepts only non-empty vector
vecHead :: Vec a (Succ n) → a
vecHead (Cons x xs) = x

-- type error!
headOfEmpty = vecHead Nil
```

Why Dependent Types?

```
-- accepts only non-empty vector
vecHead :: Vec a (Succ n) → a
vecHead (Cons x xs) = x

-- type error!
headOfEmpty = vecHead Nil

-- addition in type-level
type family Plus (x::Nat) (y::Nat) :: Nat where
  Plus Zero y = y
  Plus (Succ x) y = Succ (Plus x y)
```

Why Dependent Types?

```
-- accepts only non-empty vector
vecHead :: Vec a (Succ n) → a
vecHead (Cons x xs) = x

-- type error!
headOfEmpty = vecHead Nil

-- addition in type-level
type family Plus (x::Nat) (y::Nat) :: Nat where
  Plus Zero y = y
  Plus (Succ x) y = Succ (Plus x y)

-- property of length is ensured in type-level
append :: Vec a n → Vec a m → Vec a (Plus n m)
append Nil v = v
append (Cons a v1) v2 = Cons a (append v1 v2)
```

Why Dependent Types?

- Dependent types help us eliminate erroneous programs
- Type-level computation
- Equivalence proofs
- ...

From Haskell to Coercion Quantification

- Why Dependent Types?
- Why Dependent Haskell?

Why Dependent Haskell?

A set of language extensions for GHC that provides the ability to program **as if** the language had dependent types[1]

```
{-# LANGUAGE DataKinds, TypeFamilies, PolyKinds,
    TypeInType, GADTs, RankNTypes, TypeOperators,
    FunctionalDependencies, ScopedTypeVariables,
    TypeApplications, Template Haskell,
    UndecidableInstances, InstanceSigs,
    TypeSynonymInstances, KindSignatures,
    MultiParamTypeClasses, TypeFamilyDependencies,
    AllowAmbiguousTypes, FlexibleContexts ... #-}
```

[1] Adapted from *Dependent Types in Haskell* by Stephanie Weirich at StrangeLoop'17

Why Dependent Haskell?

- There is no unified meta-theory for the extensions.

Why Dependent Haskell?

- There is no unified meta-theory for the extensions.
- Duplications for term-level and type-level functions.

```
-- term-level function
plus :: Nat → Nat → Nat
plus Zero m = m
plus (Succ n) m = Succ (plus n m)
```

Why Dependent Haskell?

- There is no unified meta-theory for the extensions.
- Duplications for term-level and type-level functions.

```
-- term-level function
plus :: Nat → Nat → Nat
plus Zero m = m
plus (Succ n) m = Succ (plus n m)
```

- Restrictions:
  - All applications of a type family must be fully saturated with respect to that arity;
  - Data families are not promoted;
  - ...

Why Dependent Haskell?

- There is no unified meta-theory for the extensions.
- Duplications for term-level and type-level functions.

```
-- term-level function
plus :: Nat → Nat → Nat
plus Zero m = m
plus (Succ n) m = Succ (plus n m)
```

- Restrictions:
  - All applications of a type family must be fully saturated with respect to that arity;
  - Data families are not promoted;
  - ...
- Plan: to extend GHC with full-spectrum dependent types in a way that is compatible with the current implementation, with the goal of simplifying and unifying many of GHC's extensions (Eisenberg, 2016; Gundry, 2013; Weirich et al., 2017).

From Haskell to Coercion Quantification

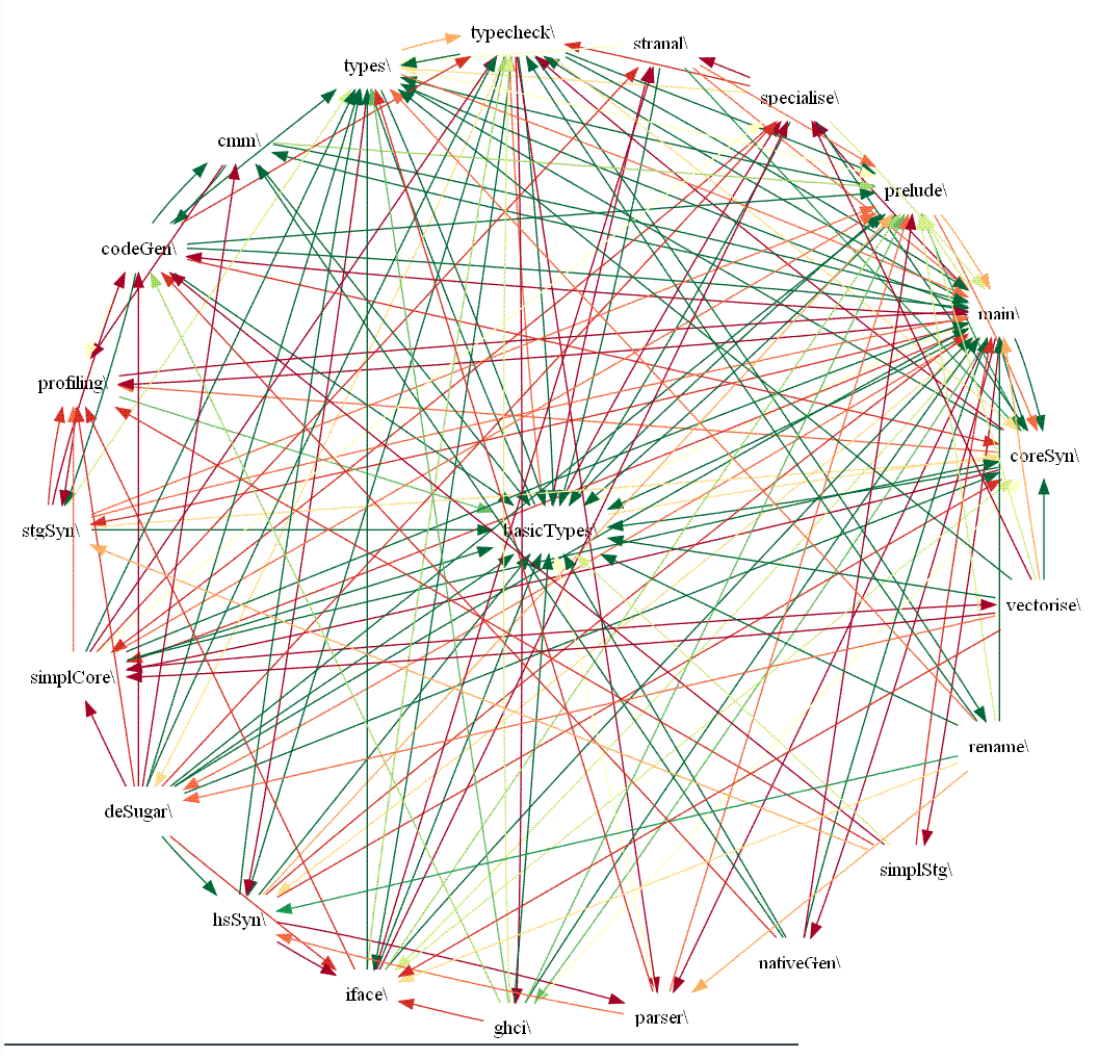
- Why Dependent Types?
- Why Dependent Haskell?
- Why Dependent Core?

Why Dependent Core?

Adding dependent types to GHC in one patch...

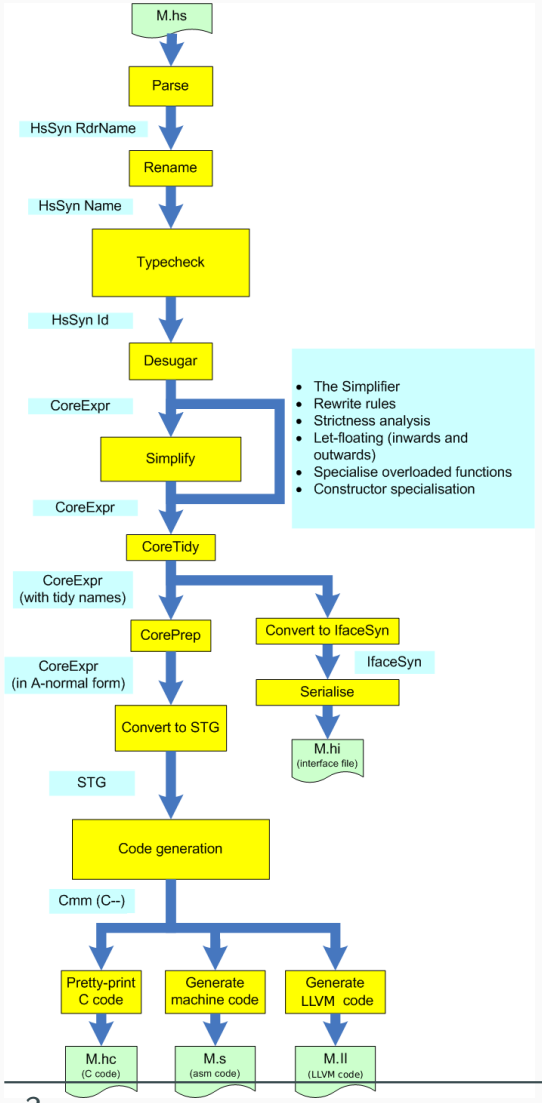
Why Dependent Core?

Adding dependent types to GHC in one patch... is very difficult [2].



[2] High-level Dependency Graph from `https://ghc.haskell.org/trac/ghc/wiki/Commentary/ModuleStructure`

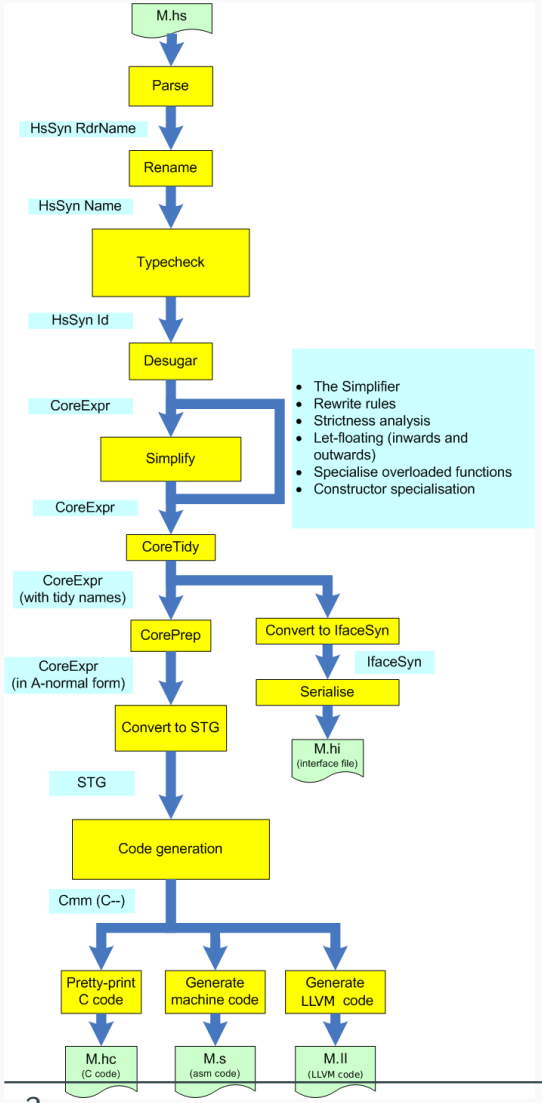
Why Dependent Core?



- GHC incorporates several compilation phases [3].

[3] Compiler Pipeline from `https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/HscPipe`

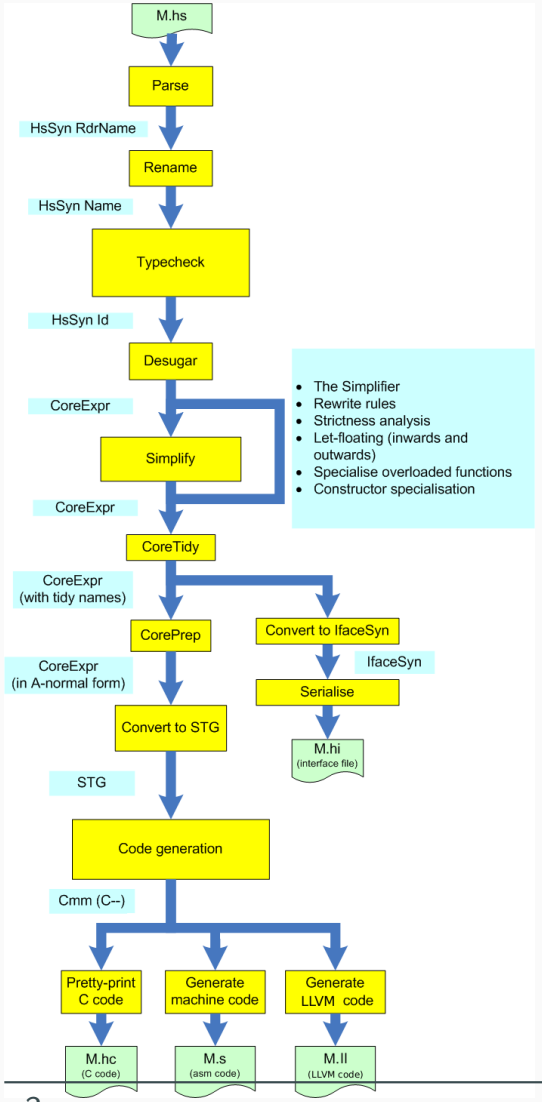
Why Dependent Core?



- GHC incorporates several compilation phases [3].
- Dependent Core, as steps are taken towards dependent Haskell (Weirich et al., 2017).

[3] Compiler Pipeline from `https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/HscPipe`

Why Dependent Core?



[3]Compiler Pipeline from

- GHC incorporates several compilation phases [3].
- Dependent Core, as steps are taken towards dependent Haskell (Weirich et al., 2017).
- Some discussions can be found in Haskell wiki[4].

`https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/HscPipe`

[4]`https://ghc.haskell.org/trac/ghc/wiki/DependentHaskell/Phase2`

From Haskell to Coercion Quantification

- Why Dependent Types?
- Why Dependent Haskell?
- Why Dependent Core?
- Why Homogeneous Equality?

Why Homogeneous Equality?

- Given an equality proposition $a : A \sim b : B$
  - **Homogeneous** Equality: A and B are definitionally equivalent types.
  - **Heterogeneous** Equality: A and B may be unrelated.

Why Homogeneous Equality?

- Given an equality proposition $a : A \sim b : B$
  - **Homogeneous** Equality: A and B are definitionally equivalent types.
  - **Heterogeneous** Equality: A and B may be unrelated.
- A Specification for Dependent Types in Haskell (Weirich et al., 2017).

  The most important change is that we show that the use of homogeneous equality propositions is compatible with explicit coercion proofs...by changing our treatment of equality propositions, we are able to simplify both the language semantics and the proofs of its metatheoretic properties with no cost to expressiveness...

From Haskell to Coercion Quantification

- Why Dependent Types?
- Why Dependent Haskell?
- Why Dependent Core?
- Why Homogeneous Equality?
- Why Coercion Quantification?

Why Coercion Quantification?

Suppose we have homogeneous equalities `A ~# B` in Core, then how can we define the kind-indexed GADTs? [5]

```
-- Source
data Rep :: ∀ k. k → Type where
  RepBool :: Rep Bool     -- instantiate k to Type
  RepMaybe :: Rep Maybe -- instantiate k to Type → Type
```

[5] Codes in Source Haskell are in black, and codes in Core are in blue.

Why Coercion Quantification?

Suppose we have homogeneous equalities `A ~# B` in Core, then how can we define the kind-indexed GADTs? [5]

```
-- Source
data Rep :: ∀ k. k → Type where
  RepBool :: Rep Bool      -- instantiate k to Type
  RepMaybe :: Rep Maybe -- instantiate k to Type → Type


-- Core
Rep :: ∀ k. k → Type
RepBool :: ∀ k (a :: k).
           (k ~# Type)
           → (a ~# Bool) -- ill-kinded
           → Rep k a
RepMaybe :: ∀ k (a :: k).
            (k ~# (Type → Type))
            → (a ~# Maybe) -- ill-kinded
            → Rep k a
```

[5] Codes in Source Haskell are in black, and codes in Core are in blue.

Why Coercion Quantification?

Suppose we have homogeneous equalities `A ~# B` in Core, then how can we define the kind-indexed GADTs? [5]

```
-- Source
data Rep :: ∀ k. k → Type where
  RepBool :: Rep Bool      -- instantiate k to Type
  RepMaybe :: Rep Maybe -- instantiate k to Type → Type


-- Core
Rep :: ∀ k. k → Type
RepBool :: ∀ k (a :: k).
           ∀ (cv :: k ~# Type). -- a name for kind co
           ((a ▷ cv) ~# Bool) -- kind cast
           → Rep k a
RepMaybe :: ∀ k (a :: k).
            ∀ (cv :: k ~# (Type → Type)).--kind co
            ((a ▷ cv) ~# Maybe) -- kind cast
            → Rep k a
```

[5] Codes in Source Haskell are in black, and codes in Core are in blue.

Coercion Quantification

Coercion Quantification

A quantification over a coercion variable.

Coercion Quantification

A quantification over a coercion variable.

- Forall-type over coercion variable.

```
RepBool :: ∀ k (a :: k).
           ∀ (cv :: k ~# Type). -- kind coercion
           ((a ▷ cv) ~# Bool) -- cast
           → Rep k a
```

Coercion Quantification

A quantification over a coercion variable.

- Forall-type over coercion variable.

```
RepBool :: ∀ k (a :: k).
           ∀ (cv :: k ∼# Type). -- kind coercion
           ((a ▷ cv) ∼# Bool) -- cast
           → Rep k a
```

- Forall-coercion over coercion variable.

```
∀ c: c1. c2 -- a coercion between two
            -- (Forall-types over coercion variable)
:: ∀ (co1:: k1 ∼# k2). A ∼# ∀ (co2::k3 ∼# k4). B
```

Coercion Quantification: Theory and Implementation

Theory

- Typing rules: roles, `NthCo`, `InstCo`; optimizations; visibility rules; ...

Coercion Quantification: Theory and Implementation

Theory

- Typing rules: roles, `NthCo`, `InstCo`; optimizations; visibility rules; ...

$$\frac{\begin{array}{c}\Gamma \vdash \gamma_1 : (t_1 \sim_r t_2) \sim_N (t_3 \sim_r t_4) \\ \Gamma, c : t_1 \sim_r t_2 \vdash \gamma_2 : t_5 \sim_{r2} t_6 \\ \eta_1 = \mathit{Nth}\ r\ 2\ (\mathit{downgradeRole}\ r\ N\ \gamma_1) :: t_1 \sim_r t_3 \\ \eta_2 = \mathit{Nth}\ r\ 3\ (\mathit{downgradeRole}\ r\ N\ \gamma_1) :: t_2 \sim_r t_4\end{array}}{\Gamma \vdash \forall c : \gamma_1.\gamma_2 :: (\forall c : t_1 \sim t_2.t_5) \sim_{r2} (\forall c : t_3 \sim t_4.t_6[c \mapsto \eta_1; c; \eta_2])}$$

Coercion Quantification: Theory and Implementation

Practice

- Merged patch: `https://phabricator.haskell.org/D5054`
- Both `ForAllTy` and `ForAllCo` can quantify over coercion variables, but only in **Core**.
- All relevant functions are updated accordingly.

Coercion Quantification: Theory and Implementation

Example: `compiler/basicTypes/DataCon.hs`[6]

```
data DataCon
= MkData { ...
    dcUnivTyVars         :: [TyVar]
    dcExTyVars           :: [TyVar]
    dcUserTyVarBinders   :: [TyVarBinder]
  ... }
-- old invariant : tyvars in dcUserTyVarBinders =
    dcUnivTyVars `union` dcExTyVars
```

[6] Simplified for presentation.

Example: `compiler/basicTypes/DataCon.hs`[6]

```
data DataCon
= MkData { ...
    dcUnivTyVars         :: [TyVar]
    dcExTyVars           :: [TyVar]
    dcUserTyVarBinders   :: [TyVarBinder]
  ... }
-- old invariant : tyvars in dcUserTyVarBinders =
    dcUnivTyVars `union` dcExTyVars

data DataCon
= MkData { ...
    dcUnivTyVars         :: [TyVar]
    dcExTyCoVars         :: [TyCoVar] -- covars allowed
    dcUserTyVarBinders   :: [TyVarBinder]
  ... }
-- new invariant : tyvars in dcUserTyVarBinders =
    dcUnivTyVars `union` (tyvars in dcExTyCoVars)
```

[6] Simplified for presentation.

Coercion Quantification: Design Space in Source Haskell

- So far, the user experience with Haskell should not be changed at all.

Coercion Quantification: Design Space in Source Haskell

- So far, the user experience with Haskell should not be changed at all.
- Question for Haskellers: do you want the compiler to accept the type of `fun`?

```
data SameKind :: k → k → *
fun :: ∀ k1 k2 (a::k1) (b::k2).
                (k1 ∼ k2) ⇒ SameKind a b
```

Coercion Quantification: Design Space in Source Haskell

- So far, the user experience with Haskell should not be changed at all.
- Question for Haskellers: do you want the compiler to accept the type of `fun`?

```
-- rejected!
data SameKind :: k → k → *
fun :: ∀ k1 k2 (a::k1) (b::k2).
                (k1 ∼ k2) ⇒ SameKind a b
```

Coercion Quantification: Design Space in Source Haskell

- So far, the user experience with Haskell should not be changed at all.
- Question for Haskellers: do you want the compiler to accept the type of `fun`?

```
-- rejected!
data SameKind :: k → k → *
fun :: ∀ k1 k2 (a::k1) (b::k2).
                (k1 ∼ k2) ⇒ SameKind a b
```

If the solver is smart enough (which implies non-trivial extensions), it should accept the program and produce

```
-- in Core
data SameKind :: k → k → *
fun :: ∀ k1 k2 (a:k1) (b:k2).
                (co :: k1 ∼# k2). -- generate a name
                SameKind (a ▷ co) b -- insert a cast
```

Future Work

Future Work

- Implementation of homogeneous equality in Core.
- Dependent Core: unified syntax for terms/types/kinds, elaboration, etc.
- Dependent Source Haskell.

References

Richard A. Eisenberg. 2016. *Dependent Types in Haskell: Theory and Practice*. PhD Dissertation. University of Pennsylvania.

Adam Michael Gundry. 2013. *Type Inference, Haskell and Dependent Types*. PhD Dissertation. University of Strathclyde.

Stephanie Weirich, Antoine Voizard, Pedro Henrique Avezedo de Amorim, and Richard A Eisenberg. 2017. A Specification for Dependent Types in Haskell. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 31.

Acknowledgments

Google Summer of Code

Summer of Haskell

Coercion Quantification

Ningning Xie [1] Richard A. Eisenberg [2]

22 Sept. 2018

Haskell Implementor's Workshop (HIW'18)

[1]The University of Hong Kong

[2]Bryn Mawr College

Backup Slides

Why Haskell?

Why Haskell? Why not any existing dependently typed language? [Eisenberg, 2016]

- Haskell is a general purpose functional programming language.
- Backward-compatible type inference.
- No termination or totality checking.

Different Equalities

Note [The equality types story] in `compiler/prelude/TysPrim.hs`

- `(∼#) :: ∀ k1 k2. k1 → k2 → #`
  - The Type Of Equality in GHC. This type is used in the solver for recording equality constraints.
  - We want this to be homogeneous.
- `(∼∼) :: ∀ k1 k2. k1 → k2 → Constraint`
  - Defined as if

    ```
    class a ∼# b ⇒ a ∼∼ b
    instance a ∼# b ⇒ a ∼∼ b
    ```

  - We want this to keep heterogeneous.

Strongly Typed STLC

- De Bruijn index

Strongly Typed STLC

- De Bruijn index

```
-- type: unit, or arrow
data Typ = U | Typ :→ Typ
 infixr 0 :→
```

Strongly Typed STLC

- De Bruijn index

```
-- type: unit, or arrow
data Typ = U | Typ :→ Typ
 infixr 0 :→

-- context is a list of types, e.g. [U, U → U, U]
```

Strongly Typed STLC

- De Bruijn index

```
-- type: unit, or arrow
data Typ = U | Typ :→ Typ
 infixr 0 :→

-- context is a list of types, e.g. [U, U → U, U]
-- variable in context
data Member :: a → [a] → * where
  First :: ∀ elm ls . Member elm (elm:ls)
  Next  :: ∀ elm ls x . Member elm ls → Member elm
  (x:ls)
```

Strongly Typed STLC

- De Bruijn index

```
-- type: unit, or arrow
data Typ = U | Typ :→ Typ
 infixr 0 :→

-- context is a list of types, e.g. [U, U → U, U]
-- variable in context
data Member :: a → [a] → * where
  First :: ∀ elm ls . Member elm (elm:ls)
  Next  :: ∀ elm ls x . Member elm ls → Member elm
  (x:ls)

test1 :: Member Int '[Int, Bool]
test1 = First

test2 :: Member Bool '[Int, Bool]
test2 = Next First

-- type error!
-- test3 :: Member Int '[Int, Bool]
-- test3 = Next First
```

Strongly Typed STLC

```
-- context [Typ], expr has type Typ
data Expr :: [Typ] → Typ → * where
  Unit :: ∀ ts. Expr ts U
  Var  :: ∀ ts t. Member t ts → Expr ts t
  Abs  :: ∀ ts dom ran. Expr (dom:ts) ran → Expr ts
    (dom :→ ran)
  App  :: ∀ ts dom ran. Expr ts (dom :→ ran) → Expr ts
    dom → Expr ts ran
```

Strongly Typed STLC

```
-- context [Typ], expr has type Typ
data Expr :: [Typ] → Typ → * where
  Unit :: ∀ ts. Expr ts U
  Var  :: ∀ ts t. Member t ts → Expr ts t
  Abs  :: ∀ ts dom ran. Expr (dom:ts) ran → Expr ts
    (dom :→ ran)
  App  :: ∀ ts dom ran. Expr ts (dom :→ ran) → Expr ts
    dom → Expr ts ran


-- \x. \y. x
fst' :: Expr '[]  (a :→ b :→ a)
fst' = Abs (Abs (Var (Next First)))
{-       ^                ---------
         |                    ^
         |                    |
         --------------------
-}
```

Strongly Typed STLC

```
-- context [Typ], expr has type Typ
data Expr :: [Typ] → Typ → * where
  Unit :: ∀ ts. Expr ts U
  Var  :: ∀ ts t. Member t ts → Expr ts t
  Abs  :: ∀ ts dom ran. Expr (dom:ts) ran → Expr ts
    (dom :→ ran)
  App  :: ∀ ts dom ran. Expr ts (dom :→ ran) → Expr ts
    dom → Expr ts ran


-- \x. \y. x
fst'' :: Expr '[]  (a :→ b :→ b) -- ill-typed!
fst'' = Abs (Abs (Var (Next First)))
{-      ^               ---------
        |                   ^
        |                   |
        --------------------
-}
```