Type theory behind GHC internals

Leap Workshop at LambdaConf 2018

Vitaly Bragilevsky

June 3, 2018

Boulder, CO

Review of GHC and its development process

- Haskell compiler
- Website: www.haskell.org/ghc/
- GHC Steering Committee and GHC proposals: github.com/ghc-proposals/ghc-proposals
- The Glasgow Haskell Team (committers & contributors): ghc.haskell.org/trac/ghc/wiki/TeamGHC
- Developed since 1990s

Key GHC developers

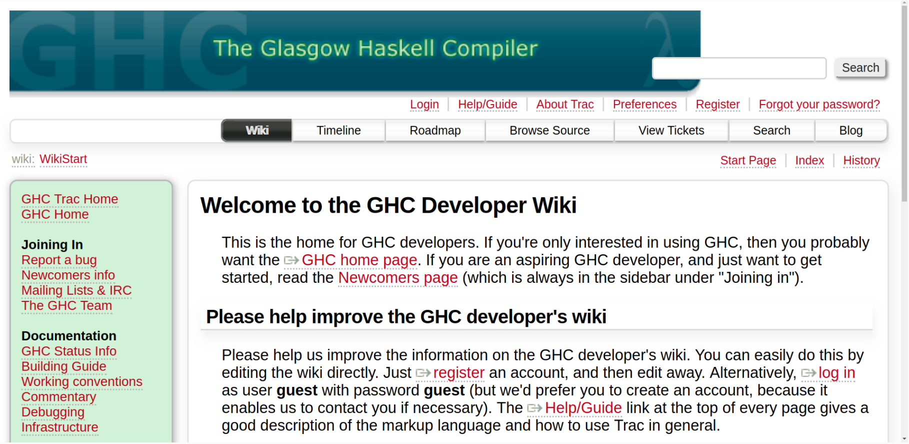
Simon Peyton Jones

Simon Marlow

The Glasgow Haskell Compiler, Marlow and Peyton Jones, 2012:

> *While the ultimate goal for us, the main developers of GHC, is to produce research rather than code, we consider developing GHC to be an essential prerequisite: the artifacts of research are fed back into GHC, so that GHC can then be used as the basis for further research that builds on these previous ideas.*

Developers Wiki (Trac)



ghc.haskell.org/trac/ghc/

What you can find in the Developers Wiki

- Newcomers info
- Building
- How to work on GHC
- Implementation details: The GHC Commentary
- Advice on debugging, testing, and profiling
- Issues (bugs, feature-requests, tasks)

- The GHC Reading List
- A Haskell Implementation Reading List (Stephen Diehl)
- Source code:
  - git.haskell.org/ghc.git
  - github.com/ghc/ghc (mirror)
- Phabricator, phabricator.haskell.org:
  - Sending patches, reviewing
  - Remote building and validating (Harbormaster)

Source code directories

```
bootstrapping
compiler
distrib
docs
driver
ghc
hadrian
includes
inplace
iserv
libffi
libffi-tarballs
libraries
mk
nofib
rts
rules
testsuite
utils
```

Haskell packages for building GHC itself (`libraries`)

- `ghc-prim`

Haskell packages for building GHC itself (`libraries`)

- `ghc-prim`
- `integer-gmp`, `integer-simple`

Haskell packages for building GHC itself (`libraries`)

- `ghc-prim`
- `integer-gmp`, `integer-simple`
- `base`

Haskell packages for building GHC itself (`libraries`)

- `ghc-prim`
- `integer-gmp`, `integer-simple`
- `base`
- array, binary, bytestring, Cabal, containers, deepseq, directory, dph, filepath, ghc-boot, ghc-boot-th, ghc-compact, ghci, haskeline, hpc, mtl, parallel, parsec, pretty, primitive, process, random, stm, template-haskell, terminfo, text, time, transformers, unix, vector, Win32, xhtml

Runtime system (`rts`)

- Memory management (garbage collection, STM primitives)
- Threads management
- Implementation of primitive operations
- Bytecode interpreter and GHCi dynamic linker

Runtime system (`rts`)

- Memory management (garbage collection, STM primitives)
- Threads management
- Implementation of primitive operations
- Bytecode interpreter and GHCi dynamic linker

The most of these components are implemented in C.

How to build GHC from sources

```
bravit@:ghc$ inplace/bin/ghc-stage2 -V
The Glorious Glasgow Haskell
Compilation System,
version 8.5.20180603
```

https://ghc.haskell.org/trac/ghc/wiki/Newcomers#Firststeps

Building steps

1. Prepare workstation (ghc, perl, gcc, make, happy, alex, autoconf, automake, python3, python-sphinx, libedit, ncurses, …)
2. Get sources
3. `./configure`
4. `make` (this builds stage1, stage2 [, stage3] GHC compilers)

Building steps

1. Prepare workstation (ghc, perl, gcc, make, happy, alex, autoconf, automake, python3, python-sphinx, libedit, ncurses, ...)
2. Get sources
3. `./configure`
4. `make` (this builds stage1, stage2 [, stage3] GHC compilers)

The rule of GHC directory tree reading:

if you see unknown file type, then it comes from build system (99%-guaranteed)

Alternative build system Hadrian

- Andrey Mokhov (Newcastle University, UK)
- github.com/snowleopard/hadrian
- Video: skillsmatter.com/skillscasts/8722-meet-hadrian-a-new-build-system-for-ghc
- Implemented in Haskell, uses Shake
  (Neil Mitchell, shakebuild.com/)

Newcomers info

- First steps
- How to rebuild modified code quickly
- How to find an issue (“low-hanging fruits”)
- Advice and links

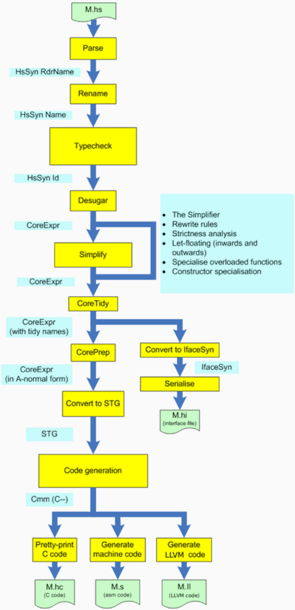
Don't get scared. GHC is a big codebase, but it makes sense when you stare at it long enough!

Haskell module compilation pipeline

GHC compiler (`compiler`)

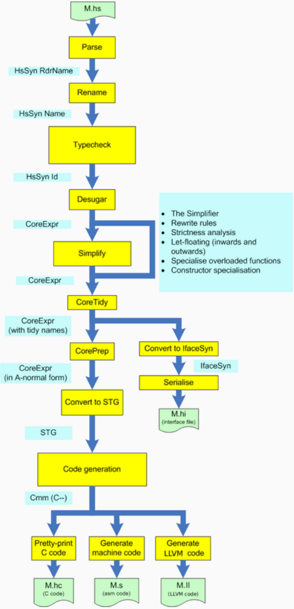
- Compilation manager
- The Haskell Compiler, `Hsc`
- Driver (composing `Hsc` with C preprocessor, assembler, linker, etc)

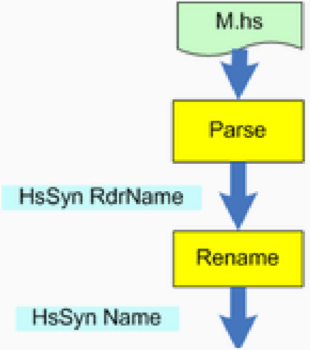
The Haskell Compiler



1. Parser (lexical and syntax analysis)
2. Renamer
3. Type checker
4. Desugarer
5. Optimizer
6. Code generator

Compilation pipeline (1)





- HsSyn – Abstract Syntax Tree (AST).
- RdrName, Name – names with additional info.

```haskell
data HsModule name
  = HsModule {
    hsmodName :: Maybe
                 (Located ModuleName),
    hsmodExports :: Maybe
                 (Located [LIE name]),
    hsmodImports :: [LImportDecl name],
    hsmodDecls :: [LHsDecl name],
    -- ...
   }
```

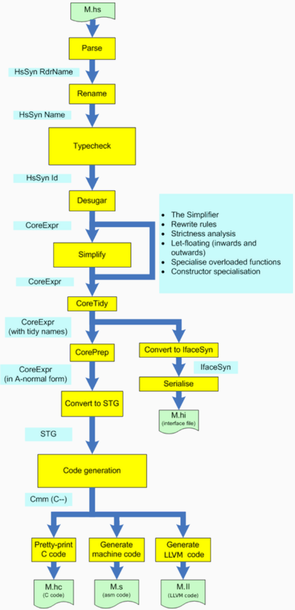
AST for Haskell source code (HsSyn), declarations

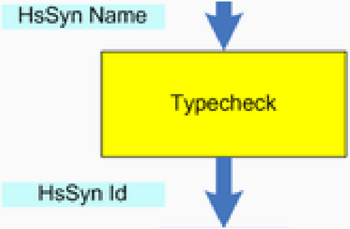
```haskell
type LHsDecl id = Located (HsDecl id)

data HsDecl id
  = TyClD         (TyClDecl id)
  | InstD         (InstDecl  id)
  | DerivD        (DerivDecl id)
  | ValD          (HsBind id)
  | SigD          (Sig id)
  | DefD          (DefaultDecl id)
  | ForD          (ForeignDecl id)
  ...
```

```
...
| WarningD     (WarnDecls id)
| AnnD         (AnnDecl id)
| RuleD        (RuleDecls id)
| VectD        (VectDecl id)
| SpliceD      (SpliceDecl id)
| DocD         (DocDecl)
| RoleAnnotD   (RoleAnnotDecl id)
```

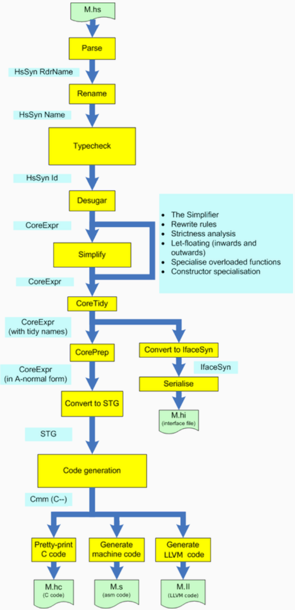
Compilation pipeline (2): type checking

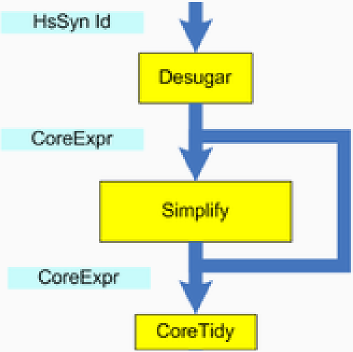




- Id—name with type (every expression has type at this step—either explicitly specified, or inferred)

Compilation pipeline (3): Core and optimisations





- Desugaring = converting to intermediate language (GHC Core)
- Optimisation

Type CoreExpr

```haskell
type CoreBndr = Var
type CoreExpr = Expr CoreBndr

data Expr b
  = Var   Id
  | Lit   Literal
  | App   (Expr b) (Arg b)
  | Lam   b (Expr b)
  | Let   (Bind b) (Expr b)
  | Case  (Expr b) b Type [Alt b]
  | Cast  (Expr b) Coercion
  | Tick  (Tickish Id) (Expr b)
  | Type  Type
  | Coercion Coercion
```

```haskell
type Arg b = Expr b
type Alt b = (AltCon, [b], Expr b)
data AltCon = DataAlt DataCon
            | LitAlt  Literal
            | DEFAULT
data Bind b = NonRec b (Expr b)
            | Rec [(b, (Expr b))]
```

GHC Core syntax: expressions and patterns

$$
\begin{array}{lcll}
t, e, u & ::= & x & \text{Variable} \\
& | & k & \text{Literal} \\
& | & \lambda x : \sigma . e & \text{Abstraction} \\
& | & e\ u & \text{Application} \\
& | & \mathrm{let}\ \overline{x : \tau = e}\ \mathrm{in}\ u & \text{Local binding} \\
& | & \mathrm{case}\ e\ \mathrm{of}\ \overline{p \to u} & \text{Case expression} \\
& | & e \rhd \gamma & \text{Type casting} \\
& | & \tau & \text{Type} \\
& | & \lfloor \gamma \rfloor & \text{Type coercion} \\
p & ::= & K\ \overline{c : \eta}\ \overline{x : \tau} & \text{Patterns}
\end{array}
$$

- Overline—list of subterms

Types

$$
\begin{array}{lcl}
\tau,\ \kappa,\ \sigma,\ \phi & ::= & \\
& | & n \\
& | & \tau_1\ \tau_2 \\
& | & T\ \overline{\tau_i}^{\,i} \\
& | & \tau_1 \to \tau_2 \\
& | & \forall n.\tau \\
& | & \mathsf{lit} \\
& | & \tau \triangleright \gamma \\
& | & \gamma
\end{array}
$$

GHC Core optimisation

- Simplifier
- Term rewriting rules
- Strictness analysis
- Overloading specialisation
- …

GHC Core example

```haskell
f a b = a + b
main = print $ f 2 3
```

GHC Core example

```haskell
f a b = a + b
main = print $ f 2 3
```

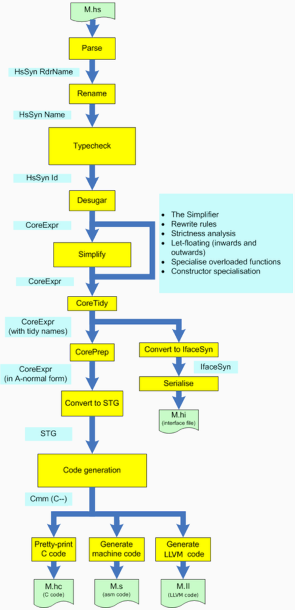
```
$ ghc -c ex.hs -ddump-simpl
```

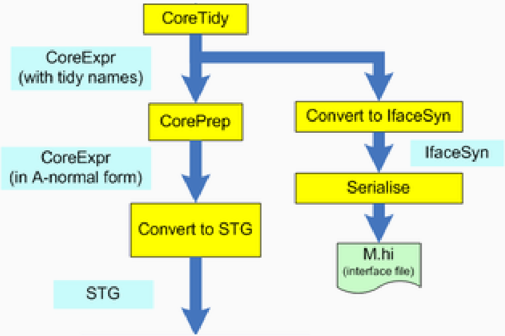
```
main :: IO ()
[GblId, Str=DmdType]
main =
 print
  @ Integer
  GHC.Show.$fShowInteger
  (+ @ Integer
       GHC.Num.$fNumInteger
       2 3)
```

Marlow and Peyton Jones, 2012:

> *In practice Core has been incredibly stable: over a 20-year time period we have added exactly one new major feature to Core (namely coercions and their associated casts). Over the same period, the source language has evolved enormously. We attribute this stability not to our own brilliance, but rather to the fact that Core is based directly on foundational mathematics:* ***bravo Girard!***

Compilation pipeline (4): prepare to code generation





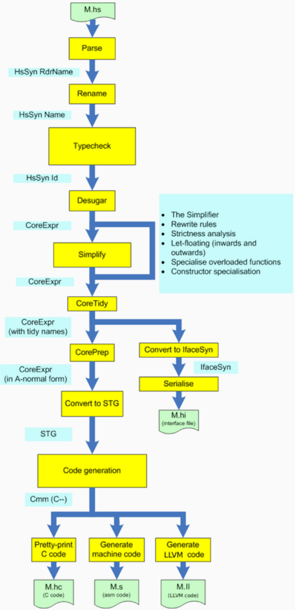
- Constructing *normal form*
- Generating interface files (for cross-module optimisations).
- STG — Spineless Tagless G-machine

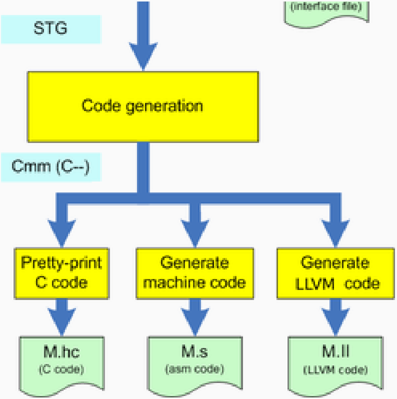
STG code fragment

```
sat_s10R :: GHC.Integer.Type.Integer
[LclId, Str=DmdType] =
    \u srt:SRT:[rp0 :-> GHC.Num.$fNumInteger] []
        let {
          sat_s10Q [Occ=Once] ::
                      GHC.Integer.Type.Integer
          [LclId, Str=DmdType] =
              NO_CCS GHC.Integer.Type.S#! [3#]; } in
        let {
          sat_s10P [Occ=Once] ::
                      GHC.Integer.Type.Integer
          [LclId, Str=DmdType] =
              NO_CCS GHC.Integer.Type.S#! [2#];
            } in  GHC.Num.+ GHC.Num.$fNumInteger
                                sat_s10P sat_s10Q;
```

```
Main.main :: GHC.Types.IO ()
[GblId, Str=DmdType] =
  \u srt:SRT:[0B :-> System.IO.print,
               rLs :-> GHC.Show.$fShowInteger,
               s10R :-> sat_s10R] []
  System.IO.print GHC.Show.$fShowInteger sat_s10R;
```

Compilation pipeline (5): code generation





- Cmm — low-level imperative language with explicit stack

Cmm code fragment

```
I64[(old + 24)] = stg_bh_upd_frame_info;
I64[(old + 16)] = _c10Y::I64;
I64[Hp - 24] = GHC.Integer.Type.S#_con_info;
I64[Hp - 16] = 3;
_c111::P64 = Hp - 23;
I64[Hp - 8] = GHC.Integer.Type.S#_con_info;
I64[Hp] = 2;
_c112::P64 = Hp - 7;
R2 = GHC.Num.$fNumInteger_closure;
I64[(old + 48)] = stg_ap_pp_info;
P64[(old + 40)] = _c112::P64;
P64[(old + 32)] = _c111::P64;
call GHC.Num.+_info(R2) args: 48, res: 0, upd: 24;
```

Code generation artifacts

- Bytecode
- Native code
- C-code
- LLVM IR (LLVM intermediate representation)

How to extend compiler

- User-defined rewriting rules
- Compiler plugins
- GHC as a library (GHC API)

Term rewriting rules

```
{-# RULES "fold/build"
  forall k z (g::forall b.
              (a->b->b) -> b -> b) .
       foldr k z (build g) = g k z
 #-}
```

- Semantically correct transformation (should be proved somehow outside GHC)

- Plugin is a single optimisation step iteration—a function from Core to Core
- Enabling by ghc option or source code pragma
- Plugin annotations to specify exactly when to run transformation

GHC as a library

- Steps modularity
- Every step is a function
- Compiler can be part of a user application

Extremely useful book

Simon L. Peyton Jones, The implementation of functional programming languages. 1987. 445 pp.

- Part 1. Compiling high-level functional languages
- Part 2. Graph reduction
- Part 3. Advanced graph reduction

Untyped λ-calculus

Types and programming languages, Pierce, 2002

Syntax

`t`	::=		*terms:*
	`x`		*variable*
	`λx.t`		*abstraction*
	`t t`		*application*
`v`	::=		*values:*
	`λx.t`		*value-abstraction*

Evaluation (operational semantics)

$\boxed{\mathtt{t} \to \mathtt{t}'}$

$$\frac{\mathtt{t}_1 \to \mathtt{t}_1'}{\mathtt{t}_1\ \ \mathtt{t}_2 \to \mathtt{t}_1'\ \ \mathtt{t}_2} \qquad \text{(E-App1)}$$

$$\frac{\mathtt{t}_2 \to \mathtt{t}_2'}{\mathtt{v}_1\ \ \mathtt{t}_2 \to \mathtt{v}_1\ \ \mathtt{t}_2'} \qquad \text{(E-App2)}$$

$$(\lambda\mathtt{x}.\mathtt{t}_{12})\ \ \mathtt{v}_2 \to [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \qquad \text{(E-AppAbs)}$$

Possible extensions

- Boolean constants with operations
- Natural numbers with operations
- Pairs, records, lists

Possible extensions

- Boolean constants with operations
- Natural numbers with operations
- Pairs, records, lists

Options

- Coding these notions in λ-calculus itself
- Adding new constructs

Untyped λ-calculus: Exercises

Repository:
`https://github.com/bravit/tt-ghc-exercises`

Repository:
https://github.com/bravit/tt-ghc-exercises

```
$ stack build
$ stack exec untyped -- untyped/test.f
```

Simply typed λ-calculus (STLC, $\lambda\!\to$)

Syntax

`t`	::=		*terms:*
	`x`		*variable*
	λ `x :T .t`		*abstraction*
	`t t`		*application*
`v`	::=		*values:*
	λ `x :T .t`		*value-abstraction*
`T`	::=		*types:*
	`T` $\rightarrow$ `T`		*function type*
Γ	::=		*contexts:*
	$\emptyset$		*empy context*
	Γ`, x:T`		*binding term variable*

Evaluation $\boxed{\mathtt{t} \to \mathtt{t}'}$

$$\frac{\mathtt{t}_1 \to \mathtt{t}_1'}{\mathtt{t}_1\ \ \mathtt{t}_2 \to \mathtt{t}_1'\ \ \mathtt{t}_2} \qquad \text{(E-App1)}$$

$$\frac{\mathtt{t}_2 \to \mathtt{t}_2'}{\mathtt{v}_1\ \ \mathtt{t}_2 \to \mathtt{v}_1\ \ \mathtt{t}_2'} \qquad \text{(E-App2)}$$

$$(\lambda\ \ \mathtt{x}\,\mathtt{:T}_{11}\,\mathtt{.t}_{12})\ \ \mathtt{v}_2 \to [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \qquad \text{(E-AppAbs)}$$

Typing

$$\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$$

$$\frac{\mathtt{x} : \mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-Var)}$$

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(T-App)}$$

- Basic types (Bool, Nat,…)
- Constants and operations for basic types (true, false, if/then/else, 0, succ, pred, iszero)
- Evaluation and typing rules

New typing rules

$\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\vdash \mathtt{true} : \mathtt{Bool} \qquad \text{(T-True)}$$

$$\vdash \mathtt{false} : \mathtt{Bool} \qquad \text{(T-False)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{Bool} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T} \qquad \Gamma \vdash \mathtt{t}_3 : \mathtt{T}}{\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{T}} \qquad \text{(T-If)}$$

$$\vdash \mathtt{0} : \mathtt{Nat} \qquad \text{(T-Zero)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{Nat}}{\Gamma \vdash \mathtt{succ}\ \mathtt{t}_1 : \mathtt{Nat}} \qquad \text{(T-Succ)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{Nat}}{\Gamma \vdash \mathtt{pred}\ \mathtt{t}_1 : \mathtt{Nat}} \qquad \text{(T-Pred)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{Nat}}{\Gamma \vdash \mathtt{iszero}\ \mathtt{t}_1 : \mathtt{Bool}} \qquad \text{(T-IsZero)}$$

Type inference trees: Example

$$\dfrac{\dfrac{\dfrac{\mathtt{x{:}Bool} \in \mathtt{x{:}Bool}}{\mathtt{x{:}Bool} \vdash \mathtt{x{:}Bool}}\,\text{T-Var}}{\vdash \lambda\mathtt{x{:}Bool.x} \;:\; \mathtt{Bool} \to \mathtt{Bool}}\,\text{T-Abs} \qquad \dfrac{}{\vdash \mathtt{true} \;:\; \mathtt{Bool}}\,\text{T-True}}{\vdash (\lambda\mathtt{x{:}Bool.x})\ \mathtt{true} \;:\; \mathtt{Bool}}\,\text{T-}$$

Typing relation properties

Safety = progress + preservation

Typing relation properties

Safety = progress + preservation

- Progress: correctly typed term is either value or can be further evaluated
- Preservation: if correctly typed term is evaluated then resulting term is correctly typed

Other typing relation properties

- Inversion lemmas (reversing typing rules)
- Type is uniquely identified
- Canonical forms (`succ (succ (succ 0))`)
- Preserving types in substitution
- Shuffling and weakening contexts
- Normalization
- Explicit and implicit typing

Erasing types in evaluation

Types are not used during evaluation so they can be erased

Erasing types in evaluation

Types are not used during evaluation so they can be erased

$$
\begin{aligned}
\mathit{erase}(\mathtt{x}) &= \mathtt{x} \\
\mathit{erase}(\lambda\mathtt{x}\!:\!\mathtt{T}_1.\mathtt{t}_2) &= \lambda\mathtt{x}.\quad \mathtt{erase}(\mathtt{t}_2) \\
\mathit{erase}(\mathtt{t}_1\ \mathtt{t}_2) &= \mathit{erase}(\mathtt{t}_1)\,\mathit{erase}(\mathtt{t}_2)
\end{aligned}
$$

Types are not used during evaluation so they can be erased

$$
\begin{aligned}
\mathit{erase}(\mathtt{x}) &= \mathtt{x} \\
\mathit{erase}(\lambda \mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2) &= \lambda \mathtt{x}. \quad \mathtt{erase}(\mathtt{t}_2) \\
\mathit{erase}(\mathtt{t}_1\ \mathtt{t}_2) &= \mathit{erase}(\mathtt{t}_1)\ \mathit{erase}(\mathtt{t}_2)
\end{aligned}
$$

Property: evaluation commutes with erasing

Repository:
`https://github.com/bravit/tt-ghc-exercises`

Repository:
`https://github.com/bravit/tt-ghc-exercises`

```
$ stack build
$ stack exec stlc -- stlc/test.f
```

The problem: code duplication in STLC

```
doubleNat  = λf:Nat → Nat.
                 λx:Nat. f (f x)
doubleBool = λf:Bool → Bool.
                 λx:Bool. f (f x)
doubleFun  =
  λf:(Nat → Nat) → (Nat → Nat).
     λx:Nat →  Nat. f (f x)
```

Henk Barendregt's λ-cube (1991)

$$
\begin{tikzcd}
 & \lambda\omega \arrow[rr] & & \lambda\Pi\omega \\
\lambda 2 \arrow[ur] \arrow[rr] & & \lambda\Pi 2 \arrow[ur] & \\
 & \lambda\underline{\omega} \arrow[uu] \arrow[rr] & & \lambda\Pi\underline{\omega} \arrow[uu] \\
\lambda{\to} \arrow[uu] \arrow[ur] \arrow[rr] & & \lambda\Pi \arrow[uu] \arrow[ur] &
\end{tikzcd}
$$

System F
polymorphic λ-calculus, $\lambda 2$

Polymorphic functions

```
id : ∀X. X ->  X
id = ΛX. λx:X. x
idNat = id [Nat]
```

Polymorphic functions

```
id : ∀X. X ->  X
id = ΛX. λx:X. x
idNat = id [Nat]

double : ∀X. (X ->  X) -> X -> X
double = ΛX. λf:X -> X. λa:X. f (f a)
doubleNat = double [Nat]
```

Polymorphic functions

```
id : ∀X. X ->  X
id = ΛX. λx:X. x
idNat = id [Nat]

double : ∀X. (X ->  X) -> X -> X
double = ΛX. λf:X -> X. λa:X. f (f a)
doubleNat = double [Nat]

quadruple = ΛX. double [X -> X]
                       (double [X])
```

Polymorphic functions

```
id : ∀X. X ->  X
id = ΛX. λx:X. x
idNat = id [Nat]
```

```
double : ∀X. (X ->  X) -> X -> X
double = ΛX. λf:X -> X. λa:X. f (f a)
doubleNat = double [Nat]
```

```
quadruple = ΛX. double [X -> X]
                       (double [X])
```

- Parametric polymorphism

Syntax (extends STLC)

t	::=		*terms:*
	x		*variable*
	λx:T.t		*abstraction*
	t t		*application*
	ΛX.t		*type abstraction*
	t [T]		*type application*
v	::=		*values:*
	λx:T.t		*value-abstraction*
	ΛX.t		*value-type abstraction*

$\mathtt{T}$	::=	*types:*
	$\boxed{\mathsf{X}}$	*type variable*
	$\mathtt{T} \to \mathtt{T}$	*function type*
	$\boxed{\forall \mathsf{X}.\mathsf{T}}$	*universal type*
Γ	::=	*contexts:*
	$\emptyset$	*empty context*
	$\Gamma, \mathtt{x} : \mathtt{T}$	*binding term variable*
	$\boxed{\Gamma, \mathsf{X}}$	*binding type variable*

Evaluation $\boxed{\mathtt{t} \rightarrow \mathtt{t}'}$

$$\frac{\mathtt{t}_1 \rightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \rightarrow \mathtt{t}_1'\ \mathtt{t}_2} \qquad \text{(E-App1)}$$

$$\frac{\mathtt{t}_2 \rightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \rightarrow \mathtt{v}_1\ \mathtt{t}_2'} \qquad \text{(E-App2)}$$

$$(\lambda \mathtt{x}:\mathtt{T}_{11}.\mathtt{t}_{12})\ \mathtt{v}_2 \rightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \qquad \text{(E-AppAbs)}$$

$$\frac{\mathtt{t}_1 \to \mathtt{t}_1'}{\mathtt{t}_1\ [\mathtt{T}_2] \to \mathtt{t}_1'\ [\mathtt{T}_2]} \qquad \text{(E-TApp)}$$

$$(\Lambda\mathtt{X}.\mathtt{t}_{12})\ [\mathtt{T}_2] \to [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{t}_{12} \qquad \text{(E-TappTabs)}$$

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{x} : \mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-Var)}$$

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2{:}\mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(T-App)}$$

$$\frac{\Gamma, \mathtt{X} \vdash \mathtt{t}_2 \ : \ \mathtt{T}_2}{\Gamma \vdash \Lambda\mathtt{X}.\mathtt{t}_2 \ : \ \forall\mathtt{X}.\mathtt{T}_2} \quad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 \ : \ \forall\mathtt{X}.\mathtt{T}_{12}}{\Gamma \vdash \mathtt{t}_1\ [\mathtt{T}_2] \ : \ [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{T}_{12}} \quad \text{(T-TApp)}$$

System F properties

- Progress and preservation

System F properties

- Progress and preservation
- Normalization

System F properties

- Progress and preservation
- Normalization
- We need `fix` to define non-normalized functions

System F properties

- Progress and preservation
- Normalization
- We need `fix` to define non-normalized functions
- Possibility of type erasure

System F properties

- Progress and preservation
- Normalization
- We need `fix` to define non-normalized functions
- Possibility of type erasure
- (!!!) type reconstruction is undecidable

- Limited System F: prenex polymorphism (type variables range over types without quantifiers), rank 2 polymorphism

- Limited System F: prenex polymorphism (type variables range over types without quantifiers), rank 2 polymorphism
- Impredicativity (quantified variable can be defined object itself)

Practising with System F: Haskell

```
double : ∀X. (X ->  X) -> X -> X
double = ΛX. λf:X -> X. λa:X. f (f a)

quadruple = ΛX. double [X -> X]
                       (double [X])
```

Practising with System F: Haskell

```
double : ∀X. (X ->  X) -> X -> X
double = ΛX. λf:X -> X. λa:X. f (f a)

quadruple = ΛX. double [X -> X]
                       (double [X])

{-# NOINLINE double #-}
double :: (a -> a) -> a -> a
double f x = f (f x)

{-# NOINLINE quadruple #-}
quadruple = double double
```

```
double_rpX :: forall a. (a -> a) -> a -> a
double_rpX
  = \ (@ a_ayq) (f_as1 :: a_ayq -> a_ayq)
                (x_as2 :: a_ayq) ->
      f_as1 (f_as1 x_as2)

quadruple_rrY :: forall a. (a -> a) -> a -> a
quadruple_rrY
  = \ (@ a_ayv) ->
       double_rpX @ (a_ayv -> a_ayv)
                    (double_rpX @ a_ayv)
```

System F_ω ($\lambda\omega$)

Functions over types, kinds

```
Id = ΛX. X
```

Functions over types, kinds

```
Id = ΛX. X

Id = ΛX::*. X
```

Functions over types, kinds

```
Id = ΛX. X

Id = ΛX::*. X

Pair = ΛA::*. ΛB::*.∀X.
            (A ->  B ->  X) -> X
```

Functions over types, kinds

```
Id = ΛX. X

Id = ΛX::*. X

Pair = ΛA::*. ΛB::*.∀X.
            (A ->  B ->  X) -> X

Pair :: * -> * -> *
```

Functions over types, kinds

```
Id = ΛX. X

Id = ΛX::*. X

Pair = ΛA::*. ΛB::*.∀X.
            (A ->  B ->  X) -> X

Pair :: * -> * -> *

PairNB = Pair [Nat] [Bool]
```

System F_ω

- Kinds: $*$, $* \to *$, $* \to * \to *$, ...
- Kinding relation (kind assignment) $\Gamma \vdash T :: K$
- Definitional equality ($S \equiv T$):

  $\texttt{Id Nat} \to \texttt{Id Bool} \equiv \texttt{Nat} \to \texttt{Bool}$
- Parallel reduction: directed version of definitional equality

Sample rules

$$\frac{\Gamma, X{::}K_1 \vdash \mathtt{T}_2{::}K_2}{\Gamma \vdash \Lambda X{::}K_1\,.\,\mathtt{T}_2 \;::\; K_1 \to \; K_2} \qquad \text{(K-Abs)}$$

$$\frac{\Gamma, X{::}K_1 \vdash \mathtt{T}_2{::}*}{\Gamma \vdash \forall X{::}K_1\,.\,\mathtt{T}_2 \;::\; *} \qquad \text{(K-All)}$$

$$\frac{S_2 \equiv \mathtt{T}_2}{\forall X{::}K_1\,.\,S_2 \equiv \forall X{::}K_1\,.\,\mathtt{T}_2} \qquad \text{(Q-All)}$$

Modifications in typing rules (fragment)

$$\frac{\Gamma, \mathtt{X} :: \mathtt{K}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{X} :: \mathtt{K}_1 . \mathtt{t}_2 : \forall \mathtt{X} :: \mathtt{K}_1 . \mathtt{T}_2} \quad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \forall \mathtt{X} :: \mathtt{K}_{11} . \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{T}_2 :: \mathtt{K}_{11}}{\Gamma \vdash \mathtt{t}_1\ [\mathtt{T}_2] : [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{T}_{12}} \quad \text{(T-TApp)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{S} \qquad \mathtt{S} \equiv \mathtt{T} \qquad \Gamma \vdash \mathtt{T} :: *}{\Gamma \vdash \mathtt{t} : \mathtt{T}} \quad \text{(T-Eq)}$$

- Untyped λ-calculus
- Simply typed λ-calculus
- System F
- System F_ω

For many years, GHC's intermediate language was essentially:

- *System Fw, plus*
- *algebraic data types (including existentials)*

But that is inadequate to describe GADTs and associated types. So in 2006 we extended GHC to support System FC, which adds

- *equality constraints and coercions*

GHC Commentary: The Compiler

Algebraic data types: Haskell

```haskell
data Point x = Point x x

{-# NOINLINE xCoord #-}

xCoord (Point x _) = x
yCoord (Point _ y) = y

main = print $ xCoord (Point 2 3)
```

Algebraic data types: GHC Core

```
xCoord_rq1 :: forall x. Point x -> x
xCoord_rq1
  = \ (@ x_a1dk) (ds_d1y8 :: Point x_a1dk) ->
        case ds_d1y8 of { Point x1_as6 ds1_d1ye
                                  -> x1_as6 }

main :: IO ()
main
  = print
      @ Integer
      GHC.Show.$fShowInteger
      (xCoord_rq1 @ Integer
                  (Main.Point @ Integer 2 3))
```

Intermediate language GHC Core in System F_ω period

t, e, u	::=	x	Variables
	|	K	Data constructors
	|	k	Literals
	|	$\lambda x : \sigma . e \mid e\ u$	Abstraction and application (values)
	|	$\Lambda a : \eta . e \mid e\ \varphi$	Abstraction and application (types)
	|	$\text{let } \overline{x : \tau = e} \text{ in } u$	Local binding
	|	$\text{case } e \text{ of } \overline{p \to u}$	Case expression
p	::=	$K\ \overline{c : \eta}\ \overline{x : \tau}$	Patterns

Generalized algebraic datatypes and associated types, System FC

Generalized algebraic datatypes (GADT)

[Peyton Jones et al., 2006; Schrijvers et al., 2009]

```
data Exp a where
  Zero :: Exp Int
  Succ :: Exp Int -> Exp Int
  Pair :: Exp b -> Exp c -> Exp (b, c)
```

```haskell
eval :: Exp a -> a
eval Zero = 0   -- Int
eval (Succ e) = eval e + 1   -- Int
eval (Pair x y) = (eval x, eval y)
                                  -- (b, c)
```

```haskell
eval :: Exp a -> a
eval Zero = 0   -- Int
eval (Succ e) = eval e + 1   -- Int
eval (Pair x y) = (eval x, eval y)
                                -- (b, c)

res = eval (Pair (Succ Zero) Zero)
```

```haskell
eval :: Exp a -> a
eval Zero = 0   -- Int
eval (Succ e) = eval e + 1   -- Int
eval (Pair x y) = (eval x, eval y)
                                -- (b, c)

res = eval (Pair (Succ Zero) Zero)

     -- (Int, Int)
```

[Sulzmann et al., 2007, rewritten in 2009–2011]

- Current context contains two syntactically different types (say, `a` and **`Int`**) which are in fact the same types (are *coercible*)

[Sulzmann et al., 2007, rewritten in 2009–2011]

- Current context contains two syntactically different types (say, `a` and **`Int`**) which are in fact the same types (are *coercible*)
- We introduce *coercion*, some evidence of *coercibility*, into the context

[Sulzmann et al., 2007, rewritten in 2009–2011]

- Current context contains two syntactically different types (say, `a` and **`Int`**) which are in fact the same types (are *coercible*)
- We introduce *coercion*, some evidence of *coercibility*, into the context
- Now it is safe to do type *cast*

```
data Exp a where
  Zero : ∀a.(a ∼ Int) => Exp a
  Succ : ∀a.(a ∼ Int) =>
                       Exp Int -> Exp a
  Pair : ∀abc.(a ∼ (b,c)) =>
             Exp b -> Exp c -> Exp a
```

- **`Zero`** has additional argument—evidence of coercibility of `a` to **`Int`**
- **`Exp`** is regular ADT now

```
zero : Exp Int
zero = Zero ???
```

```
zero : Exp Int
zero = Zero ???
```

- reflexivity: any type is a coercion to itself

```
eval : Exp a -> a
eval =
  Λa:*.λx:Exp a.
   case x of
    Zero (co : a∼Int) -> 0 ▶ sym co
    ...
```

- cast operator: ▶
- symmetry in coercion:
  (`sym co : Int` ∼ a)
- ▶ casts to the right hand side of ∼

```
eval : Exp a -> a
eval =
  Λa:*.λx:Exp a.
   case x of
    Zero (co : a∼Int) -> 0 ▶ sym co
    ...
```

- cast operator: ▶
- symmetry in coercion: (`sym co : Int` $\sim$ a)
- ▶ casts to the right hand side of $\sim$
- Why coercion is a type and not a value?

Associated types [Chakravarty et al., 2005]

```
class Collects c where
  type Elem c
  empty  :: c
  insert ::  Elem c -> c -> c

instance Eq e => Collects [e] where
  type Elem [e] = e
  ...
instance Collects BitSet where
  type Elem BitSet = Char
  ...
```

```haskell
foo :: Char -> BitSet
foo x = insert x empty
```

Original type classes and instances implementation

- Type class is translated into record (ADT with one constructor, dictionary)
- Every class method becomes record's field
- Instance is a value of this record, fields are methods' implementations
- Every function with this constraint receives additional argument—the dictionary with an implementation

Original type classes and instances implementation

- Type class is translated into record (ADT with one constructor, dictionary)
- Every class method becomes record's field
- Instance is a value of this record, fields are methods' implementations
- Every function with this constraint receives additional argument—the dictionary with an implementation
- Record cannot contain types!

Implementation in GHC Core (System FC)

Type class (Haskell)

```
class Collects c where
  type Elem c
  empty  :: c
  insert ::  Elem c -> c -> c
```

Abstract type constructor and type of dictionary (Core)

```
type Elem : * -> *
data CollectsDict c =
  Collects {empty : c;
     insert : Elem c -> c -> c}
```

Instance (Haskell)

```
instance Collects BitSet where
  type Elem BitSet = Char
  ...
```

Coercion axiom and dictionary (Core)

```
axiom elemBS : Elem BitSet ~ Char

dCollectsBS : CollectsDict Bitset
dCollectsBS = ...
```

Method call (Haskell)

```
foo :: Char -> BitSet
foo x = insert x empty
```

Function call (GHC Core)

```
foo : Char -> BitSet
foo =
  λx:Char. insert [BitSet] dCollectBS
        (x ▶ sym elemBS)
        (empty [BitSet] dCollectBS)
```

```
axiom elemBS : Elem BitSet ∼ Char
```

System FC [Sulzmann et al., 2007]

- Coercions as types and arguments passing in System F style, erasability
- Operators over coercions (sym etc.) and their normalization
- Abstract type constructors and axioms
- Typing and kinding rules
- Progress and preservation
- Operational semantics
- Erasure properties
- Rules for translation from Haskell

Bugs in type-level computations and System FC_2

Deriving instances for newtype

GHC extension GeneralizedNewtypeDeriving

```
newtype Age = MkAge { unAge :: Int }
class Idea a where
  good :: a -> a
instance Idea Int where
  good = (+1)
deriving instance Idea Age
```

GHC extension GeneralizedNewtypeDeriving

```
newtype Age = MkAge { unAge :: Int }
class Idea a where
  good :: a -> a
instance Idea Int where
  good = (+1)
deriving instance Idea Age
```

axiom CoMkAge : Age $\sim$ Int

```
axiom CoMkAge : Age ~ Int
```

Question

Let T is a function over types. What about

`T Age ~ T Int`?

Deriving instances in the presence of type-level functions

```haskell
newtype Age = MkAge { unAge :: Int }

type family Inspect x
type instance Inspect Age = Int
type instance Inspect Int = Bool

class BadIdea a where
  bad :: a -> Inspect a
instance BadIdea Int where
  bad = (> 0)
deriving instance BadIdea Age
```

Which type is it?

```
bad (MkAge 5)
```

Which type is it?

```
bad (MkAge 5) :: Int
```

Which type is it?

```
bad (MkAge 5) :: Int
```

What GND builds?

```
axiom CoMkAge : Age ∼ Int
```

Which type is it?

`bad (MkAge 5) :: Int`

What GND builds?

$\texttt{axiom CoMkAge : Age} \sim \texttt{Int}$

GND would use implementation `(>0)` from instance for **`Int`**

Which type is it?

`bad (MkAge 5) :: Int`

What GND builds?

$$\texttt{axiom CoMkAge : Age} \sim \texttt{Int}$$

GND would use implementation `(>0)` from instance for **`Int`** so we'd access internal **`Bool`** representation! Congrats!

Which type is it?

`bad (MkAge 5) :: Int`

What GND builds?

$\texttt{axiom CoMkAge : Age} \sim \texttt{Int}$

GND would use implementation `(>0)` from instance for **`Int`** so we'd access internal **`Bool`** representation! Congrats!

$\texttt{Inspect Age} \not\sim \texttt{Inspect Int}$

Which type is it?

`bad (MkAge 5) :: Int`

What GND builds?

$\texttt{axiom CoMkAge : Age} \sim \texttt{Int}$

GND would use implementation `(>0)` from instance for **`Int`** so we'd access internal **`Bool`** representation! Congrats!

$\texttt{Inspect Age} \nsim \texttt{Inspect Int}$

But $\texttt{Maybe Age} \sim \texttt{Maybe Int}$

Which type is it?

`bad (MkAge 5) :: Int`

What GND builds?

$\texttt{axiom CoMkAge : Age} \sim \texttt{Int}$

GND would use implementation `(>0)` from instance for **`Int`** so we'd access internal **`Bool`** representation! Congrats!

$\texttt{Inspect Age} \not\sim \texttt{Inspect Int}$

But $\texttt{Maybe Age} \sim \texttt{Maybe Int}$

Problem

When is it OK to lift $\sim$?

Roles idea

[Weirich et al., 2011; *Roles (GHC Dev wiki)*]

Type can play different roles:

- encoding
- representation

Roles idea

[Weirich et al., 2011; *Roles (GHC Dev wiki)*]

Type can play different roles:

- encoding
- representation

Here come different equalities:

- nominal equality (up to type synonyms)
- representational equality

Parametric and non-parametric type var occurences

```haskell
data List a = Nil | Cons a (List a)
data GADT a where
  GAge :: GADT Age
  GInt :: GADT Int
class C1 a where
  foo :: a -> List a
class C2 a where
  bar :: a -> GADT a
class BadIdea a where
  bad :: a -> Inspect a
```

Lifting $\sim$ and roles

- If type variable has parametric occurence then we may lift $\sim$ in case of representational equality
- Otherwise, lifting is allowed only in case of nominal equality

- If type variable has parametric occurence then we may lift $\sim$ in case of representational equality
- Otherwise, lifting is allowed only in case of nominal equality

Question

What is the right place for type roles?

Types and kinds in System FC_2

```
k  ::=  * | k' -> k          kinds
R  ::=  n | r                roles
k' ::=  k/R
```

Types and kinds in System FC_2

```
k  ::=  * | k' -> k           kinds
R  ::=  n | r                 roles
k' ::=  k/R
```

Example

```
Maybe   ::  */r -> *    parametric
Inspect ::  */n -> *    non-parametric
axiom CoMkAge : (Age ∼ Int)/r
```

Example

```
Maybe   ::  */r -> *  parametric
Inspect ::  */n -> *  non-parametric
axiom CoMkAge : (Age ~ Int)/r
```

$$\frac{\Gamma \vdash \gamma_1 : \varphi_1 \sim \varphi_2 \in (\eta_1/R_2 \to \eta_2)/R_1 \quad \Gamma \vdash \gamma_2 : \psi_1 \sim \psi_2 \in \eta_1/min(R_1, R_2)}{\Gamma \vdash \gamma_1\ \gamma_2 : \varphi_1\ \psi_1 \sim \varphi_2\ \psi_2 \in \eta_2/R_1} \quad \textsc{CApp}$$

- $n \leqslant r$
- With Maybe we can use whatever we like
- With Inspect—nominal role types only

System FC_2 [Weirich et al., 2011]

- Roles for differing type application contexts
- Roles as a kind component
- Progress and preservation
- Updated translation rules
- Simplifying System FC

“Typing” type-level computations and System $\mathbf{F}_C^{\uparrow}$

Vectors with GADTs: Example

Traditional implementation

```
data Zero
data Succ n

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

- `Zero :: *`
- `Succ :: * -> *`
- `Vec  :: * -> * -> *`

- `Zero :: *`
- `Succ :: * -> *`
- `Vec  :: * -> * -> *`

- No control at the type level!
- `Succ Bool` or `Vec Zero Int` are allowed!

[Giving Haskell a Promotion: Yorgey et al., 2012]

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

Enriching kinds system with promoted types

[Giving Haskell a Promotion: Yorgey et al., 2012]

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

- DataKinds, PolyKinds GHC extensions
- Type Nat becomes a kind
- Data constructors Zero and Succ become types

Computations over types (literally)

Addition over types

```
type family Plus (a :: Nat) (b :: Nat)
type instance Plus Zero b = b
type instance Plus (Succ a) b =
                    Succ (Plus a b)
```

Function over vectors

```
vconcat :: Vec a n -> Vec a m ->
                      Vec a (Plus n m)
vconcat VNil ys = ys
vconcat (VCons x xs) ys =
         VCons x (vconcat xs ys)
```

Kind polymorphism

Type equality *

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

- How to define equality for type constructors, say Maybe?

Type equality *

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

- How to define equality for type constructors, say Maybe?

PolyKinds implementation

```
data EqRefl (a :: X) (b :: X) where
  Refl :: ∀X . ∀(a :: X). EqRefl a a
```

Syntax of System $F_C^{\uparrow}$ [Yorgey et al., 2012]

$e,\ u ::=$		Expressions
	$\mid x$	Variables
	$\mid \lambda x{:}\,\tau.\ e \mid e_1\ e_2$	Abstraction/application
	$\mid \Lambda a{:}\,\kappa.\ e \mid e\ \tau$	Type abstraction/application
	$\mid \lambda c{:}\,\tau.\ e \mid e\ \gamma$	Coercion abstraction/application
	$\mid \Lambda \mathcal{X}.\ e$	Kind abstraction
	$\mid e\ \kappa$	Kind application
	$\mid K$	Data constructors
	$\mid \mathbf{case}\ e\ \mathbf{of}\ \overline{p \rightarrow u}$	Case analysis
	$\mid e \triangleright \gamma$	Casting

Syntax of System $F_C^{\uparrow}$ [Yorgey et al., 2012]

$e,\ u$	$::=$		Expressions
	$\mid$	x	Variables
	$\mid$	$\lambda x{:}\,\tau.\ e \mid e_1\ e_2$	Abstraction/application
	$\mid$	$\Lambda a{:}\,\kappa.\ e \mid e\ \tau$	Type abstraction/application
	$\mid$	$\lambda c{:}\,\tau.\ e \mid e\ \gamma$	Coercion abstraction/application
	$\mid$	$\Lambda \mathcal{X}.\ e$	Kind abstraction
	$\mid$	$e\ \kappa$	Kind application
	$\mid$	K	Data constructors
	$\mid$	$\textbf{case}\ e\ \textbf{of}\ \overline{p \to u}$	Case analysis
	$\mid$	$e \triangleright \gamma$	Casting

bnd	$::= q{:}\,\tau \mid w{:}\,\kappa \mid \mathcal{X}{:}\,\Box$	Bindings

ι	$::=$		Base kinds
	$\mid$	$\star$	Star
	$\mid$	$\boxed{\mathsf{Constraint}}$	Constraint kind
$\kappa,\ \eta$	$::=$		Kinds
	$\mid$	$\boxed{\mathcal{X}}$	Kind variables
	$\mid$	ι	Base kinds
	$\mid$	$\kappa_1 \to \kappa_2$	Arrow kinds
	$\mid$	$\boxed{\forall\, \mathcal{X}.\, \kappa}$	Kind polymorphism
	$\mid$	$\boxed{T\ \overline{\kappa}}$	Promoted type constant

System $F_C^{\uparrow}$ limitations

- Kinds can't be classified, there is only one "sort" BOX
- Only *-kinded type constructors are promoted
- No promotion for types with promoted types inside them
- No promotion for types with polymorphic kinds
- No promotion for functions!

Giving Haskell a Promotion [Yorgey et al., 2012]

- Enriching kind system
- Extending GHC Core to System $F_C^{\uparrow}$
- Every required type-theoretic property is guaranteed
- More optimisations available
- Changes in type inference algorithm for Haskell

Explicit kind equality

System FC with explicit kind equality

[Weirich, Hsu, and Eisenberg, 2013]

- Unifying types and kinds: `* :: *`
- Kind equality
- Kinds may depend on types
- Same type-theoretic properties
- Weakening System $F_C^{\uparrow}$ limitations
- Extension TypeInType (GHC 8.0, `Type :: Type`)

GHC Core by 2015

Expressions

[System FC, as implemented in GHC 2015]

$e,\ u$	$::=$		Expressions, *coreSyn/CoreSyn.lhs*:`Expr`
	$\mid$	n	`Var`: Variable
	$\mid$	lit	`Lit`: Literal
	$\mid$	$e_1\ e_2$	`App`: Application
	$\mid$	$\lambda n.e$	`Lam`: Abstraction
	$\mid$	$\mathbf{let}\ \mathit{binding}\ \mathbf{in}\ e$	`Let`: Variable binding
	$\mid$	$\mathbf{case}\ e\ \mathbf{as}\ n\ \mathbf{return}\ \tau\ \mathbf{of}\ \overline{\mathit{alt}_i}^{\,i}$	`Case`: Pattern match
	$\mid$	$e \triangleright \gamma$	`Cast`: Cast
	$\mid$	$e_{\{\mathit{tick}\}}$	`Tick`: Internal note
	$\mid$	τ	`Type`: Type
	$\mid$	γ	`Coercion`: Coercion

Types

$$
\begin{array}{lcl}
\tau,\ \kappa,\ \sigma,\ \phi & ::= & \\
& | & n \\
& | & \tau_1\ \tau_2 \\
& | & T\ \overline{\tau_i}^{\,i} \\
& | & \tau_1 \to \tau_2 \\
& | & \forall n.\tau \\
& | & \mathsf{lit} \\
& | & \tau \rhd \gamma \\
& | & \gamma
\end{array}
$$

GHC Core

- Typing rules
- Kinding rules
- Coercion operators
- Roles
- Core linting

Two theses

1. Adam Gundry. Type Inference, Haskell and Dependent Types. PhD thesis, University of Strathclyde, 2013 (advisor: Conor McBride).
2. Richard A. Eisenberg. Dependent Types in Haskell: Theory and Practice. PhD Thesis, University of Pennsylvania, 2016 (advisor: Stephanie Weirich).

Theses structure

- Outer language (inch — Dependent Haskell)
- Type inference in the outer language
- Inner language (language of evidence — PiCo)
- Translating outer language to inner language
- Programming with dependent types
- Implementation details

A Specification for Dependently-Typed Haskell

STEPHANIE WEIRICH, University of Pennsylvania

ANTOINE VOIZARD, University of Pennsylvania

PEDRO HENRIQUE AZEVEDO DE AMORIM, Ecole Polytechnique and University of Campinas

RICHARD EISENBERG, Bryn Mawr College

Presented in ICFP '2017

Main goal

Design Haskell extension such that:

- no difference between types and terms
- sharing semantics between compile-time and runtime computations

Achievements

- System DC as backward compatible with System FC
- System D as implicitly typed version of System DC:
  - erasure
  - reconstruction
  - source of ideas for type inference in outer language
- Fully formalized and proved in Coq (SSReflect).

```haskell
data Nat :: Type where
  O :: Nat
  S :: Nat -> Nat

data Vec :: Type -> Nat -> Type where
  Nil  :: Vec a O
  (:>) :: a -> Vec a m -> Vec a (S m)
```

Type for vector with full info about types

```
data Vec (a :: Type) (n :: Nat) :: Type where
  Nil :: (n ∼ 0) => Vec a n
  (:>) :: ∀ (m :: Nat). (n ∼ S m)
              => a -> Vec a m -> Vec a n
```

Type for vector with full info about types

```
data Vec (a :: Type) (n :: Nat) :: Type where
  Nil :: (n ∼ 0) => Vec a n
  (:>) :: ∀ (m :: Nat). (n ∼ S m)
             => a -> Vec a m -> Vec a n
```

Function over vectors

```
zip :: ∀ n a b. Vec a n -> Vec b n -> Vec (a,b)
zip Nil Nil = Nil
zip (x :> xs) (y :> ys) = (x, y) :> zip xs ys
```

- ∀ for erasable components
- It is enough to have two clauses

Function zip in SystemDC

```
zip =
 \-n:Nat. \-a:Type. \-b:Type.
 \xs:Vec a n. \ys:Vec b n.
    case xs of
      --ALT1
      --ALT2
```

```
--ALT1

Nil -> /\c1:(n ~ 0).
 case ys of
  Nil -> /\c2:(n ~ 0). Nil [a][n][c1]
  (:>) -> \-m:Nat. /\c2:(n ~ S m).
           \y:b. \ys:Vec b m.
               absurd [sym c1; c2]
```

```
--ALT2
(:>) ->
  \m:Nat. /\c1:(n ~ S m).
  \x:a. \xs:Vec a m. case ys of
   Nil -> /\c2:(n ~ 0).
               absurd [sym c1; c2]
   (:>) -> \-m:Nat. /\c2:(n ~ S m).
            \y:b. \ys:Vec b m.
              (:>) [a][n][m][c1]
                   ((,) [a][b] x y)
                   (zip [m][a][b] xs ys)
```

```
zip = \-n. \-a. \-b. \xs. \ys.
 case xs of
  Nil -> /\c1. case ys of
    Nil -> /\c2. Nil [][][]
    (:>) -> \-m. /\c2. \y. \ys.
                               absurd []
  (:>) -> \m. /\c1. \x. \xs. case ys of
    Nil -> /\c2. absurd []
    (:>) -> \-m. /\c2. \y. \ys.
            (:>) [][][][]
                 ((,) [][] x y)
                 (zip [][][] xs ys)
```

System D syntax

$$
\begin{array}{llcl}
\textit{terms, types} & a, b, A, B & ::= & \star \mid x \mid F \mid \lambda^{\rho} x.b \mid a\ b^{\rho} \mid \Box \\
 & & \mid & \Pi^{\rho} x\!:\!A \to B \mid \Lambda c.a \mid a[\gamma] \mid \forall c\!:\!\phi.A \\
\textit{propositions} & \phi & ::= & a \sim_A b \\
\textit{relevance} & \rho & ::= & + \mid - \\
\textit{coercions} & \gamma & ::= & \bullet \\
 & & & \\
\textit{values} & v & ::= & \lambda^{+} x.a \mid \lambda^{-} x.v \mid \Lambda c.a \mid \star \mid \Pi^{\rho} x\!:\!A \to B \mid \forall c\!:\!\phi.A \\
 & & & \\
\textit{contexts} & \Gamma & ::= & \varnothing \mid \Gamma, x : A \mid \Gamma, c : \phi \\
\textit{available set} & \Delta & ::= & \varnothing \mid \Delta, c \\
\textit{signature} & \Sigma & ::= & \varnothing \mid \Sigma \cup \{F \sim a : A\}
\end{array}
$$

Evaluation in System D

$$\frac{}{\vDash (\lambda^{\rho} x.b)\ a^{\rho} \leadsto b\{a/x\}}\ \textsc{E-AppAbs}$$

$$\frac{}{\vDash (\Lambda c.b)[\gamma] \leadsto b\{\gamma/c\}}\ \textsc{E-CAppCAbs}$$

$$\frac{F \sim a : A \in \Sigma_0}{\vDash F \leadsto a}\ \textsc{E-Axiom}$$

$$\frac{\vDash a \leadsto a'}{\vDash \lambda^{-} x.a \leadsto \lambda^{-} x.a'}\ \textsc{E-AbsTerm}$$

$$\frac{\vDash a \leadsto a'}{\vDash a\ b^{\rho} \leadsto a'\ b^{\rho}}\ \textsc{E-AppLeft}$$

$$\frac{\vDash a \leadsto a'}{\vDash a[\gamma] \leadsto a'[\gamma]}\ \textsc{E-CAppLeft}$$

Typing rules in System D

$\boxed{\Gamma \vDash a : A}$

$$\text{E-STAR}\quad \frac{\vDash \Gamma}{\Gamma \vDash \star : \star}$$

$$\text{E-VAR}\quad \frac{\vDash \Gamma \qquad x : A \in \Gamma}{\Gamma \vDash x : A}$$

$$\text{E-PI}\quad \frac{\Gamma, x : A \vDash B : \star}{\Gamma \vDash \Pi^{\rho} x{:}A \to B : \star}$$

$$\text{E-ABS}\quad \frac{\Gamma, x : A \vDash a : B \qquad \rho \lor x \notin a}{\Gamma \vDash \lambda^{\rho} x.a : \Pi^{\rho} x{:}A \to B}$$

$$\text{E-APP}\quad \frac{\Gamma \vDash b : \Pi^{+} x{:}A \to B \qquad \Gamma \vDash a : A}{\Gamma \vDash b\ a^{+} : B\{a/x\}}$$

$$\text{E-IAPP}\quad \frac{\Gamma \vDash b : \Pi^{-} x{:}A \to B \qquad \Gamma \vDash a : A}{\Gamma \vDash b\ \square^{-} : B\{a/x\}}$$

$$\text{E-CONV}\quad \frac{\Gamma \vDash a : A \qquad \Gamma; \widetilde{\Gamma} \vDash A \equiv B : \star}{\Gamma \vDash a : B}$$

$$\text{E-FAM}\quad \frac{\vDash \Gamma \qquad F \sim a : A \in \Sigma_0}{\Gamma \vDash F : A}$$

$$\text{E-CPI}\quad \frac{\Gamma, c : \phi \vDash B : \star}{\Gamma \vDash \forall c{:}\phi.B : \star}$$

$$\text{E-CABS}\quad \frac{\Gamma, c : \phi \vDash a : B}{\Gamma \vDash \Lambda c.a : \forall c{:}\phi.B}$$

$$\text{E-CAPP}\quad \frac{\Gamma \vDash a_1 : \forall c{:}(a \sim_A b).B_1 \qquad \Gamma; \widetilde{\Gamma} \vDash a \equiv b : A}{\Gamma \vDash a_1[\bullet] : B_1\{\bullet/c\}}$$

Typing rules in System D

$\boxed{\Gamma \vDash a : A}$

$$\textsc{E-Star}\ \frac{\vDash \Gamma}{\Gamma \vDash \star : \star} \qquad \textsc{E-Var}\ \frac{\vDash \Gamma \qquad x : A \in \Gamma}{\Gamma \vDash x : A} \qquad \textsc{E-Pi}\ \frac{\Gamma, x : A \vDash B : \star}{\Gamma \vDash \Pi^{\rho} x{:}A \to B : \star} \qquad \textsc{E-Abs}\ \frac{\Gamma, x : A \vDash a : B \qquad \rho \lor x \notin a}{\Gamma \vDash \lambda^{\rho} x.a : \Pi^{\rho} x{:}A \to B}$$

$$\textsc{E-App}\ \frac{\Gamma \vDash b : \Pi^{+} x{:}A \to B \qquad \Gamma \vDash a : A}{\Gamma \vDash b\ a^{+} : B\{a/x\}} \qquad \textsc{E-IApp}\ \frac{\Gamma \vDash b : \Pi^{-} x{:}A \to B \qquad \Gamma \vDash a : A}{\Gamma \vDash b\ \square^{-} : B\{a/x\}} \qquad \textsc{E-Conv}\ \frac{\Gamma \vDash a : A \qquad \Gamma; \widetilde{\Gamma} \vDash A \equiv B : \star}{\Gamma \vDash a : B} \qquad \textsc{E-Fam}\ \frac{\vDash \Gamma \qquad F \sim a : A \in \Sigma_0}{\Gamma \vDash F : A}$$

$$\textsc{E-CPi}\ \frac{\Gamma, c : \phi \vDash B : \star}{\Gamma \vDash \forall c{:}\phi.B : \star} \qquad \textsc{E-CAbs}\ \frac{\Gamma, c : \phi \vDash a : B}{\Gamma \vDash \Lambda c.a : \forall c{:}\phi.B} \qquad \textsc{E-CApp}\ \frac{\Gamma \vDash a_1 : \forall c{:}(a \sim_A b).B_1 \qquad \Gamma; \widetilde{\Gamma} \vDash a \equiv b : A}{\Gamma \vDash a_1[\bullet] : B_1\{\bullet/c\}}$$

$\boxed{\rho \lor x \notin A}$

$$\textsc{Rho-Rel}\ \frac{}{+ \lor x \notin A} \qquad \textsc{Rho-IrrRel}\ \frac{x \notin \mathsf{fv} A}{- \lor x \notin A}$$

System D: other components

- definitional equality (many rules)
- Progress and preservation

- System D plus explicit types in abstractions proofs for coercions
- Erasing to System D
- Decidable syntax-directed typing
- Uniqueness of typing
- Evaluation
- Progress and preservation
- No full System FC implementation

References

Pierce, Benjamin C. (2002). *Types and Programming Languages*.

Chakravarty, Manuel M. T. et al. (2005). “Associated Types with Class”. In: *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’05. Long Beach, California, USA: ACM, pp. 1–13.

Peyton Jones, Simon L. et al. (Sept. 2006). “Simple unification-based type inference for GADTs”. In: *International Conference on Functional Programming (ICFP)*. Portland, OR, USA, pp. 50–61.

Sulzmann, Martin et al. (2007). “System F with type equality coercions”. In: *TLDI ’07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*. Nice, Nice, France: ACM, pp. 53–66.

Schrijvers, Tom et al. (2009). “Complete and Decidable Type Inference for GADTs”. In: *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’09. Edinburgh, Scotland: ACM, pp. 341–352.

Weirich, Stephanie et al. (2011). “Generative Type Abstraction and Type-level Computation”. In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’11. Austin, Texas, USA: ACM, pp. 227–240.

Marlow, Simon and Simon Peyton Jones (2012). “The Glasgow Haskell Compiler”. In: *The Architecture of Open Source Applications*. Vol. 2. Chap. 5.

Yorgey, Brent A. et al. (2012). “Giving Haskell A Promotion”. In: *Seventh ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI ’12)*, pp. 53–66.

Weirich, Stephanie, Justin Hsu, and Richard A. Eisenberg (Sept. 2013). “System FC with explicit kind equality”. In: *Proceedings of The 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’13. Boston, MA, pp. 275–286.

System FC, as implemented in GHC (2015). url: https://github.com/ghc/ghc/blob/master/docs/core-spec/core-spec.pdf.

Weirich, Stephanie et al. (Aug. 2017). “A Specification for Dependent Types in Haskell”. In: *Proc. ACM Program. Lang.* 1.ICFP, 31:1–31:29.

GHC Commentary: The Compiler. url: https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler.

GHC Trac and Developer Wiki. url: https://ghc.haskell.org/trac/ghc.

Roles (GHC Dev wiki). url: https://ghc.haskell.org/trac/ghc/wiki/Roles.