Functional programming, Lecture No. 0
A bit of theoretical flashbacks

Danya Rogozin
Lomonosov Moscow State University,
Serokell OÜ
ITMO University
Information Technologies and Programming Faculty
The 6th of February, 2020

Introduction

General words on Haskell

- The language is named after Haskell Curry, an American logician
- The first implementation: 1990
- The current language standard: Haskell2010
- Default compiler: Glasgow Haskell Compiler (GHC)
- Haskell is a strongly-typed, polymorphic, and purely functional programming language

GHC

- GHC is an open-source project. Don’t hesistate to contribute!
- GHC is mostly implemented on Haskell
- GHC is developed under the GHC Steering committee control

GHC

- GHC is an open-source project. Don't hesitate to contribute!
- GHC is mostly implemented on Haskell
- GHC is developed under the GHC Steering committee control
- Very and very roughly and approximately, the compiling pipeline is arranged as follows:

$$\text{parsing} \Rightarrow \text{compile-time (type-checking)} \Rightarrow \text{runtime}$$

A bit of history

Lambda calculus and type theory. Historical notes

- At the end of the 1920-s, Alonzo Church provided an alternative approach to the foundations of mathematics. Here, the notion of a function is the primitive one
- Lambda calculus is a formal system that describes arbitrary abstract functions

Lambda calculus and type theory. Historical notes

- At the end of the 1920-s, Alonzo Church provided an alternative approach to the foundations of mathematics. Here, the notion of a function is the primitive one
- Lambda calculus is a formal system that describes arbitrary abstract functions
- Moreover, Church used lambda calculus to show that Peano arithmetic is undecidable.

Lambda calculus and type theory. Historical notes

- Kleene and Rosser showed that the initial version of lambda calculus is inconsistent. Initially, the idea of typing was the instrument of paradoxes avoiding.

Lambda calculus and type theory. Historical notes

- Kleene and Rosser showed that the initial version of lambda calculus is inconsistent. Initially, the idea of typing was the instrument of paradoxes avoiding.
- The first system of typed lambda calculus is a hybrid from lambda calculus and type theory developed by Bertrand Russell and Alfred North Whitehead (1910-s).

Lambda calculus and type theory. Historical notes

- After Church’s works, type theory as the branch of lambda calculus and combinatory logic was developed by Haskell Curry and William Howard from a perspective of proof theory (1950-1960-s)

Lambda calculus and type theory. Historical notes

- After Church's works, type theory as the branch of lambda calculus and combinatory logic was developed by Haskell Curry and William Howard from a perspective of proof theory (1950-1960-s)
- Polymorphic lambda calculus (John Reynolds and Jean-Yves Girard (1970-s))
- Polymorphic type inference (Roger Hindley, Robin Milner and Luis Damas (1970-1980-s))

Lambda calculus and type theory. Historical notes

- After Church's works, type theory as the branch of lambda calculus and combinatory logic was developed by Haskell Curry and William Howard from a perspective of proof theory (1950-1960-s)
- Polymorphic lambda calculus (John Reynolds and Jean-Yves Girard (1970-s))
- Polymorphic type inference (Roger Hindley, Robin Milner and Luis Damas (1970-1980-s))
- ML: the very first language with a polymorphic inferred type system (Robin Milner, 1973)

Lambda calculus and type theory. Historical notes

- After Church's works, type theory as the branch of lambda calculus and combinatory logic was developed by Haskell Curry and William Howard from a perspective of proof theory (1950-1960-s)
- Polymorphic lambda calculus (John Reynolds and Jean-Yves Girard (1970-s))
- Polymorphic type inference (Roger Hindley, Robin Milner and Luis Damas (1970-1980-s))
- ML: the very first language with a polymorphic inferred type system (Robin Milner, 1973)
- The language Haskell appeared at the beginning of 1990-s. Haskell desinged by Simon Peyton Jones, Philip Wadler, and others

General conceptual aspects

The notion of a function

Let us recall the ordinary notion of a function, the set-theoretical one.

The notion of a function

Let us recall the ordinary notion of a function, the set-theoretical one.

Definition
Let A, B be sets. A relation $f \subseteq A \times B$ is called functional, if $\langle x, y \rangle \in f$ and $\langle x, z \rangle \in f$ implies $y = z$.

The notion of a function

Let us recall the ordinary notion of a function, the set-theoretical one.

Definition
Let A, B be sets. A relation $f \subseteq A \times B$ is called functional, if $\langle x, y \rangle \in f$ and $\langle x, z \rangle \in f$ implies $y = z$.

A function is a triple $\langle A, B, f \rangle$, where f is a functional relation.
$f(x) = y \Leftrightarrow \langle x, y \rangle \in f$.

The notion of a function

- In such a set-theoretical approach, we identify a function and its graph
- In lambda calculus, the notion of a function is the primitive one
- Such an understanding of a function provides us a Turing-complete model of computation

Lambda calculus in a nutshell

The formal definition

Definition
Let $V = \{x, y, z, \dots\}$ *be the set of variables. The set of preterms is generated by the following grammar*

$$M ::= x \mid (\lambda x.M) \mid (MM)$$

The formal definition

Definition
Let $V = \{x, y, z, \dots\}$ *be the set of variables. The set of preterms is generated by the following grammar*

$$M ::= x \mid (\lambda x.M) \mid (MM)$$

Definition

- α*-conversion is an operation on preterms that renames bounded variables*
- $\equiv_\alpha$ *is a relfexive-transitive-symmetric closure of* α*-conversion*

The formal definition

Definition
Let $V = \{x, y, z, \dots\}$ be the set of variables. The set of preterms is generated by the following grammar

$$M ::= x \mid (\lambda x.M) \mid (MM)$$

Definition

- α*-conversion is an operation on preterms that renames bounded variables*
- $\equiv_\alpha$ *is a relfexive-transitive-symmetric closure of* α*-conversion*

Definition
The set of lambda terms is defined as the set of preterms modulo $\equiv_\alpha$.

$\Lambda = \Lambda_{pre} / \equiv_\alpha = \{[M]_N \mid M \equiv_\alpha N\}$

The reduction relation

Definition
An operational semantics is defined as the following rewriting rules:

Reduction rules

$$(\lambda x.M)N \to_\beta M[x := N]$$

$$\frac{M_1 \to_\beta M_2}{M_1 N \to_\beta M_2 N} \qquad \frac{M_1 \to_\beta M_2}{N M_1 \to_\beta N M_2}$$

$\twoheadrightarrow_\beta$ *is a multistep reduction, a reflexive-transitive closure of* $\to_\beta$

The reduction relation

Definition
An operational semantics is defined as the following rewriting rules:

Reduction rules

$$(\lambda x.M)N \to_\beta M[x := N]$$

$$\frac{M_1 \to_\beta M_2}{M_1 N \to_\beta M_2 N} \qquad \frac{M_1 \to_\beta M_2}{N M_1 \to_\beta N M_2}$$

$\twoheadrightarrow_\beta$ *is a multistep reduction, a reflexive-transitive closure of* $\to_\beta$

Definition

- *A term that has the form* $(\lambda x.M)N$ *is called* β*-redex*
- *A term is in normal form if it has no redexes in its subterms*

Reduction strategies

1. A term M is called weakly normalisable (WN), if there exists some halting reduction path that starts from M

Reduction strategies

1. A term M is called weakly normalisable (WN), if there exists some halting reduction path that starts from M
2. A term M is called strongly normalisable (SN), if any reduction path that starts from M terminates

Reduction strategies

1. A term M is called weakly normalisable (WN), if there exists some halting reduction path that starts from M
2. A term M is called strongly normalisable (SN), if any reduction path that starts from M terminates

It is clear, that SN implies WN, not vice versa. In other words, there exists a term that has an infinite reduction path, but it has a finite reduction path at the same time.

Reduction strategies

Let us consider the example of the following ridiculous term: $(\lambda xy.x)(\lambda z.z)((\lambda x.xx)(\lambda x.xx))$. One may reduce this term in two ways:

Reduction strategies

Let us consider the example of the following ridiculous term: $(\lambda xy.x)(\lambda z.z)((\lambda x.xx)(\lambda x.xx))$. One may reduce this term in two ways:

From the one hand

$$
\begin{aligned}
&(\lambda xy.x)(\lambda z.z)((\lambda x.xx)(\lambda x.xx)) \to_\beta \\
&(\lambda y.\lambda z.z)((\lambda x.xx)(\lambda x.xx)) \to_\beta \\
&(\lambda z.z)[y := (\lambda x.xx)(\lambda x.xx)] \to_\beta \\
&\lambda z.z
\end{aligned}
$$

Reduction strategies

Let us consider the example of the following ridiculous term: $(\lambda xy.x)(\lambda z.z)((\lambda x.xx)(\lambda x.xx))$. One may reduce this term in two ways:

From the one hand	**From the one hand**
$(\lambda xy.x)(\lambda z.z)((\lambda x.xx)(\lambda x.xx)) \to_\beta$	$(\lambda xy.x)(\lambda z.z)((\lambda x.xx)(\lambda x.xx)) \to_\beta$
$(\lambda y.\lambda z.z)((\lambda x.xx)(\lambda x.xx)) \to_\beta$	$(\lambda xy.x)(\lambda z.z)(xx)(x := [\lambda x.xx]) \to_\beta$
$(\lambda z.z)[y := (\lambda x.xx)(\lambda x.xx)] \to_\beta$	$(\lambda xy.x)(\lambda z.z)((\lambda x.xx)(\lambda x.xx)) \to_\beta$
$\lambda z.z$	$\ldots$

Reduction strategies

Thus, one needs to overview of the possible ways of application reduction. We have two chairs:

Reduction strategies

Thus, one needs to overview of the possible ways of application reduction. We have two chairs:

1. $(\lambda x_1 \dots x_n.M)N_1 \dots N_n$: we firtsly reduce $(N_i)_{i \in \{1,\dots,n\}}$
2. $(\lambda x_1 \dots x_n.M)N_1 \dots N_n$: reduce $(\lambda x_1 \dots x_n.M)N_1$ and go further from left to right

Reduction strategies

Thus, one needs to overview of the possible ways of application reduction. We have two chairs:

1. $(\lambda x_1 \ldots x_n.M)N_1 \ldots N_n$: we firtsly reduce $(N_i)_{i \in \{1,\ldots,n\}}$
2. $(\lambda x_1 \ldots x_n.M)N_1 \ldots N_n$: reduce $(\lambda x_1 \ldots x_n.M)N_1$ and go further from left to right

The first way is called an applicative order, the second one is a normal one. A normal order is much more better in a certain sense:

Reduction strategies

Thus, one needs to overview of the possible ways of application reduction. We have two chairs:

1. $(\lambda x_1 \dots x_n.M)N_1 \dots N_n$: we firtsly reduce $(N_i)_{i \in \{1,\dots,n\}}$
2. $(\lambda x_1 \dots x_n.M)N_1 \dots N_n$: reduce $(\lambda x_1 \dots x_n.M)N_1$ and go further from left to right

The first way is called an applicative order, the second one is a normal one. A normal order is much more better in a certain sense:

Theorem
Let M be a term such that M has a normal form M', then M might be reduced to M' via normal order

Lambda calculus in a nutshell. Call-by-value and call-by-name

- The applicative (normal) order is often called call-by-value (call-by-name)

Lambda calculus in a nutshell. Call-by-value and call-by-name

- The applicative (normal) order is often called call-by-value (call-by-name)
- The most mainstream programming languages you know (Java, Python, Kotlin, etc) have call-by-value semantics

Lambda calculus in a nutshell. Call-by-value and call-by-name

- The applicative (normal) order is often called call-by-value (call-by-name)
- The most mainstream programming languages you know (Java, Python, Kotlin, etc) have call-by-value semantics
- The Haskell reduction has a call-by-name strategy. Informally, such a stragety is called lazy. Laziness denotes that Haskell doesn't compute a value if it's not needed at the moment

Lambda calculus in a nutshell. Call-by-value and call-by-name

- The applicative (normal) order is often called call-by-value (call-by-name)
- The most mainstream programming languages you know (Java, Python, Kotlin, etc) have call-by-value semantics
- The Haskell reduction has a call-by-name strategy. Informally, such a stragety is called lazy. Laziness denotes that Haskell doesn't compute a value if it's not needed at the moment
- Call-by-name reduction reduces reducible terms to the bitter end, but it's not always optimal, unfortunately
- In Haskell, the reduction is arranged as call-by-name evaluation up to so called weak head normal form

Pure functions and side-effects

- A function is called pure if it yields the same value for the same argument each time

Pure functions and side-effects

- A function is called pure if it yields the same value for the same argument each time
- It means that, such a function has the same behaviour at every point. This principle is also called referential transparency

Pure functions and side-effects

- A function is called pure if it yields the same value for the same argument each time
- It means that, such a function has the same behaviour at every point. This principle is also called referential transparency
- A side-effect function is a function that may yield different value passing the same arguments. Mathematically, such a function is not function at all.
- Haskell functions are (mostly) pure ones, but Haskell isn't confluent as a version of lambda calculus

A couple of words on types

Motivation

- A type is a syntax construction that should be assigned to terms and values according to the list of rules
- Types define a sort of partial specification
- Type checking allows one to catch an enormous class of errors

Motivation

- A type is a syntax construction that should be assigned to terms and values according to the list of rules
- Types define a sort of partial specification
- Type checking allows one to catch an enormous class of errors
- The standard definition by Benjamin Peirce:

> *A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute*

General persective

Type theory has two perspectives that often intersect with each other.

- Type theory as the branch of proof theory and constructive mathematics
- Type theory as the branch of computer science and programming language theory

A landscape of typing from a bird's eye view

We may classify the possible ways of typing as follows:

- Strong and weak typing:
  - Strong typing: Java, Haskell, Ocaml, Rust, etc
  - Weak typing: JavaScript, e.g.
- Static and dynamic typing:
  - C, C++, Java, Haskell, etc
  - JavaScript, Ruby, PHP, etc
- Implicit and explicit typing:
  - JavaScript, Ruby, PHP, etc
  - C++, Java, etc
- Inferred typing:
  - Haskell, Standard ML, Ocaml, Idris, etc

A short reminder on simply typed lambda calculus

The typing rules are:

Axiom

$$\Gamma, x : \sigma \vdash x : \sigma$$

Lambda abstraction

$$\frac{\Gamma, x : \sigma \vdash M : \psi}{\Gamma \vdash (\lambda x.M) : \sigma \to \psi}$$

Application

$$\frac{\Gamma \vdash M : \sigma \to \psi \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \psi}$$

Higher order functions

- Higher-order functions are widely used in ordinary mathematics, such as differential operator, that has type $\mathbb{R}^{\mathbb{R}} \to \mathbb{R}^{\mathbb{R}}$

Higher order functions

- Higher-order functions are widely used in ordinary mathematics, such as differential operator, that has type $\mathbb{R}^{\mathbb{R}} \to \mathbb{R}^{\mathbb{R}}$
- In untyped lambda calculus, all functions are higher-order by default

Higher order functions

- Higher-order functions are widely used in ordinary mathematics, such as differential operator, that has type $\mathbb{R}^{\mathbb{R}} \to \mathbb{R}^{\mathbb{R}}$
- In untyped lambda calculus, all functions are higher-order by default
- In typed lambda calculus, a function of type $\varphi \to \psi$ is called higher-order, if $\varphi = \theta \to \delta$, for example:

$$\lambda fgx.g(fx) : (\varphi \to \psi) \to ((\psi \to \theta) \to (\varphi \to \theta))$$

Higher order functions

- Higher-order functions are widely used in ordinary mathematics, such as differential operator, that has type $\mathbb{R}^{\mathbb{R}} \to \mathbb{R}^{\mathbb{R}}$
- In untyped lambda calculus, all functions are higher-order by default
- In typed lambda calculus, a function of type $\varphi \to \psi$ is called higher-order, if $\varphi = \theta \to \delta$, for example:

$$\lambda fgx.g(fx) : (\varphi \to \psi) \to ((\psi \to \theta) \to (\varphi \to \theta))$$

- A function is a first-class object

Type systems and their metatheoretical properties

- Progress and preservation = safety
- Weak normalisation
- The type uniqueness
- The inversion property
- Weakening and permutation
- Canonicity
- Type preservation under substitution
- etc...

The type system classification

As you know, we have the following ways of dependency between terms and types:

- A term depends on type (polymorphism in system F)
- A type depends on type (so-called type operators in $\lambda_{\bar{\omega}}$)
- A type depends on terms (dependent types in the basic DT system called P and its extensions)

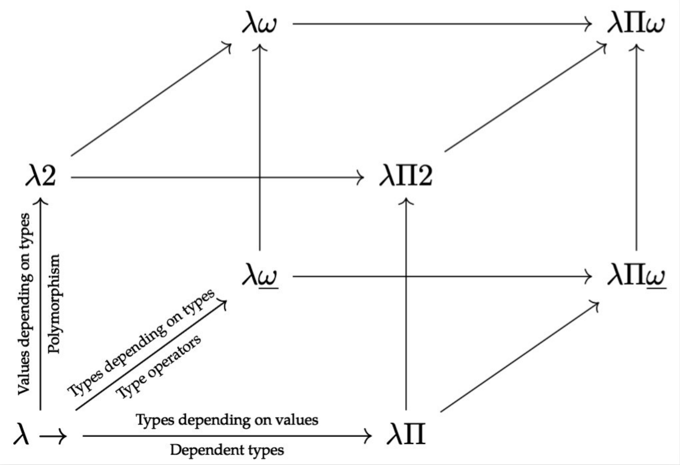
The type system classification

As you know, we have the following ways of dependency between terms and types:

- A term depends on type (polymorphism in system F)
- A type depends on type (so-called type operators in $\lambda_{\bar{\omega}}$)
- A type depends on terms (dependent types in the basic DT system called P and its extensions)

All possible combinations of these dependencies might be illustrated via Barendregt's lambda cube, the lattice of type theories

Lambda cube



λω
λΠω
λ2
λΠ2
λω
λΠω
λ →
λΠ
Values depending on types
Polymorphism
Types depending on types
Type operators
Types depending on values
Dependent types

Polymorphism

Motivation

- A polymorphism is a quite powerful tool in making a code more general and abstract
- Such an abstraction allows one to avoid a boilerplate. Let us take a look at the following functions:

```haskell
twiceInt :: (Int -> Int) -> Int -> Int
twiceInt f v = f (f v)

twiceBool :: (Bool -> Bool) -> Bool -> Bool
twiceBool f v = f (f v)
```

Motivation

- A polymorphism is a quite powerful tool in making a code more general and abstract
- Such an abstraction allows one to avoid a boilerplate. Let us take a look at the following functions:

```
twiceInt :: (Int -> Int) -> Int -> Int
twiceInt f v = f (f v)

twiceBool :: (Bool -> Bool) -> Bool -> Bool
twiceBool f v = f (f v)
```

- It is clear that all these functions do the same work.
- We don't want to reproduce the same pattern each time

System F

- System F is a polymorphic lambda calculus that was introduced by Jean-Yves Girard and John Reynolds in the 1970-s
- The initial motivation was a characterisation of computable functions that are provably recursive in second-order arithmetic

System F

- System F is a polymorphic lambda calculus that was introduced by Jean-Yves Girard and John Reynolds in the 1970-s
- The initial motivation was a characterisation of computable functions that are provably recursive in second-order arithmetic
- From an engineering perspective, System F is an axiomatic representation of parametric polymorpism
- The problem of type inference in the Curry-style typed System F is undecidable

System F. Typing rules

The Curry style typing rules:

Generalisation rule

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \forall \alpha . \sigma} \; \alpha \notin \mathrm{rng}(\Gamma)$$

Type instantiation

$$\frac{\Gamma \vdash M : \forall \alpha . \sigma}{\Gamma \vdash M : \sigma[\alpha := \psi]}$$

System F. Typing rules

The Church style typing rules:

Type abstraction

$$\frac{\Gamma, \alpha \vdash M : \sigma}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\sigma}$$

Explicit type instantiation

$$\frac{\Gamma \vdash M : \forall\alpha.\sigma}{\Gamma \vdash M[\psi] : \sigma[\alpha := \psi]}$$

System F. Typing rules

The Church style typing rules:

Type abstraction

$$\frac{\Gamma, \alpha \vdash M : \sigma}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\sigma}$$

Explicit type instantiation

$$\frac{\Gamma \vdash M : \forall\alpha.\sigma}{\Gamma \vdash M[\psi] : \sigma[\alpha := \psi]}$$

- One may enable the System F polymorphism in Haskell with the extension called `RankNTypes`
- The language extension `TypeApplications` provides an explicit type instantiation. In Haskell notation, `f @ a`

Polymorphism. Hindley-Milner type system

- Hindley-Milner type system provided a restricted polymorphism up to the one-rank quantifier depth.

Polymorphism. Hindley-Milner type system

- Hindley-Milner type system provided a restricted polymorphism up to the one-rank quantifier depth.

The inference rules are the following restricted Curry-style system F rules:

Type scheme introduction

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \forall \vec{\alpha}.\sigma} \ \vec{\alpha} \cap \mathrm{rng}(\Gamma) = \emptyset$$

Type instantiation

$$\frac{\Gamma \vdash M : \forall \alpha.\sigma}{\Gamma \vdash M : \sigma[\alpha := \psi]}$$

Type instantiation

$$\frac{\Gamma, x : \sigma \vdash M : \psi}{\Gamma, \vdash \lambda x.M : (\sigma \to \psi)} \ \sigma \text{ is quantifier-free}$$

Polymorphism. Hindley-Milner type system

This version of polymorphism is a "default" polymorphism in the basic Haskell. In fact, the following code:

```
const :: a -> b -> a
const x _ = x
```

denotes something like this

```
{-# LANGUAGE ExplicitForAll #-}
const :: forall a b. a -> b -> a
const x _ = x
```

A couple of words on kinds

Extending polymorpism with kinds

- We have already resigned to the fact that all objects are classified with types
- How can we classify types themselves?
- Types have kinds in the same sense as terms have types
- In particular, such a type classification provide way to writing type operators

Extending polymorpism with kinds. Some examples

$$\mathsf{Id} = \Lambda\sigma.\sigma$$

$$\mathsf{Id} = \Lambda\sigma : *.\sigma$$

Extending polymorpism with kinds. Some examples

$$\mathsf{Id} = \Lambda\sigma.\sigma$$

$$\mathsf{Id} = \Lambda\sigma : *.\sigma$$

$$\mathsf{Pair} = \Lambda\sigma : *.\Lambda\psi : *.\forall\vartheta.(\sigma \to \psi \to \vartheta) \to \vartheta$$

$$\mathsf{Pair} : * \to * \to *$$

Extending polymorpism with kinds. Some examples

$$\mathsf{Id} = \Lambda\sigma.\sigma$$

$$\mathsf{Id} = \Lambda\sigma : *.\sigma$$

$$\mathsf{Pair} = \Lambda\sigma : *.\Lambda\psi : *.\forall\vartheta.(\sigma \to \psi \to \vartheta) \to \vartheta$$

$$\mathsf{Pair} : * \to * \to *$$

$$\mathsf{Nat} : *$$

$$\mathsf{String} : *$$

$$\mathsf{PairNS} = \mathsf{Pair}\ [\mathsf{Nat}][\mathsf{String}]$$

System F_ω

The basic characteristics of the system F_ω are:

- Kinds: $\kappa, \mu ::= * \mid (\kappa \to \mu)$
- One needs to extend the system with kinding rules and kinding judgements that have the form $\Gamma \vdash \psi : \kappa$, where ψ is a type and κ is a kind

Let us take a look at some kinding rules:

Kinding abstraction

$$\frac{\Gamma, \varphi : \kappa_1 \vdash \psi : \kappa_2}{\Gamma \vdash \Lambda\varphi : \kappa_1.\psi : \kappa_1 \to \kappa_2}$$

Kinding generalisation

$$\frac{\Gamma, \varphi : \kappa \vdash \psi : *}{\Gamma \vdash \forall\varphi : \kappa.\psi : *}$$

The modified System F rules

The quantifier rules have the modified form:

Type abstraction with kinding

$$\frac{\Gamma, \alpha : \kappa \vdash M : \sigma}{\Gamma \vdash \Lambda\alpha : \kappa.M : \forall\alpha.\sigma}$$

Type instantiation with kinding

$$\frac{\Gamma \vdash M : \forall\alpha : \kappa_1.\sigma \qquad \Gamma \vdash \psi : \kappa_1}{\Gamma \vdash M[\psi] : \sigma[\alpha := \psi]}$$

System F_ω in Haskell

One may enable the extension called `TypeFamilies` to define type-level functions

```haskell
{-# LANGUAGE TypeFamilies #-}

type family G a where
  G Int = Bool
  G a   = Char

type family AnotherG a :: *
type instance AnotherG Int = Bool
type instance AnotherG String = Char
```

The relevant Haskell type system

System FC

- The system F_ω enriched with algebraic data types was the underlying Haskell type system till the mid-2000-s
- At the moment, F_ω is extended to the system FC. Let me drop the full definition of this system

System FC

- The system F_ω enriched with algebraic data types was the underlying Haskell type system till the mid-2000-s
- At the moment, F_ω is extended to the system FC. Let me drop the full definition of this system
- The features of *FC* are:
    1. Coercions and equality constraints
    2. Generalised algebraic data types
    3. et cetera

System FC

The example of a generalised algebraic data type is the following one:

```haskell
{-# LANGUAGE GADTs #-}

  data Term a where
    Lit    :: Int -> Term Int
    Succ   :: Term Int -> Term Int
    IsZero :: Term Int -> Term Bool
    If     :: Term Bool -> Term a -> Term a -> Term a
    Pair   :: Term a -> Term b -> Term (a,b)
```

System FC

```haskell
eval :: Term a -> a
eval (Lit i)       = i
eval (Succ t)      = 1 + eval t
eval (IsZero t)    = eval t == 0
eval (If b e1 e2) = if eval b then eval e1 else eval e2
eval (Pair e1 e2) = (eval e1, eval e2)
```

System FC

Formally, the type above is defined as follows:

```
{-# LANGUAGE ExistentialQuantification #-}
{-# LANGUAGE TypeFamilies #-}

data Term a =
  a ~ Int => Lit a |
  a ~ Int => Succ a |
 (a ~ Bool) => IsZero (Term Int) |
  If Bool (Term a) (Term a) |
  forall b c. (a ~ (b, c)) => Pair (Term b) (Term c)
```

Summary

Finally

- We discussed the general aspects of functional programming itself and its story
- We briefly and brutally reintroduced you to lambda calculus and the variety of type systems
- We overview the main directions of type theory influence on Haskell
- We said a couple of words on the underlying Haskell type system called System *FC*

Thank you for your kind attention!