barkhauseninstitut.org

Foundations of Certified Programming Language and Compiler Design

Dr.-Ing. Sebastian Ertel

Composable Operating Systems Group, Barkhausen Institute

Outline

Lecture	Logic	Formalisms	PL
1	Propositional and first-order logic		
2			Functional programming
3		Syntax and Semantics	
4			The untyped lambda calculus
5		Types	
6			The typed lambda calculus
7			Polymorphism
8		Curry-Howard	
9			Higher-order types
10			Dependent types

Goals

- At first, we have a look at an important second-order type system.
- Then, we resolve all long-standing mysteries of data types and
- introduce type-level computation to
- finally express “more interesting” propositions
- (and prove them).

From First to Second-order

- If we remove the top-level restriction for quantification, we
- enter second-order logic ,i.e.,
- we allow to quantify over predicates.

- The system that we arrive when doing so is called *System F/Polymorphic Lambda Calculus*.
- System F is the foundation for the rich and powerful type system of Haskell.

System F/Polymorphic Lambda Calculus

Syntax:

t	$::=$		terms:
	$\vert$	x	variable
	$\vert$	$\lambda x : T.t$	abstraction
	$\vert$	$t\ t$	application
	$\vert$	$\lambda X.t$	type abstraction
	$\vert$	$t\ [T]$	type application
v	$::=$		values:
	$\vert$	$\lambda x : T.t$	abstraction value
	$\vert$	$\lambda X.t$	type abstraction value

T	$::=$		types:
	$\vert$	X	type variable
	$\vert$	$T \to T$	type of functions
	$\vert$	$\forall X.T$	universal type
Γ	$::=$		contexts:
	$\vert$	$\varnothing$	empty context
	$\vert$	$\Gamma, x : T$	term variable binding
	$\vert$	Γ, X	type variable binding

System F

Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}\ \text{E-App1}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'}\ \text{E-App2}$$

$$\frac{}{(\lambda x : T.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12}}\ \text{E-AppAbs}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]}\ \text{E-TApp}$$

$$\frac{}{(\lambda X.t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12}}\ \text{E-TAppTAbs}$$

System F

Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{E-APP1}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{E-APP2}$$

$$\frac{}{(\lambda x : T.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12}} \quad \text{E-APPABS}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]} \quad \text{E-TAPP}$$

$$\frac{}{(\lambda X.t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12}} \quad \text{E-TAPPTABS}$$

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

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{T-VAR}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2 : T_1 \to T_2} \quad \text{T-ABS}$$

$$\frac{\Gamma, t_1 : T_{11} \to T_{12} \quad t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{T-APP}$$

$$\frac{\Gamma, X \vdash t_2 : T_2}{\Gamma \vdash \lambda X.t_2 : \forall X.T_2 \to T_2} \quad \text{T-TABS}$$

$$\frac{\Gamma, t_1 : \forall X.T_{12}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}} \quad \text{T-TAPP}$$

System F/Polymorphic Lambda Calculus

History

1972 – Jean-Yves Girard , a logician, discovers *System F* in the context of proof theory.

1974 – John Reynolds , a computer scientist, discovers the *polymorphic lambda calculus* that has a type system with the same expressive power.

Encoding Existentials

- With 2nd-order types, we can encode:

Encoding Existentials

- With 2nd-order types, we can encode:

$$\{\exists X, T\} \stackrel{\text{def}}{=} \forall Y.\, (\forall X.\, T \to Y) \to Y$$

Encoding Existentials

- With 2nd-order types, we can encode:

$$\{\exists X, T\} \stackrel{\text{def}}{=} \forall Y.\,(\forall X.\,T \to Y) \to Y$$

- ...and we can remove the explicit forms for existentials from the language definition.

Encoding Existentials

- With 2nd-order types, we can encode:

$$\{\exists X, T\} \stackrel{\text{def}}{=} \forall Y.\, (\forall X.\, T \to Y) \to Y$$

- ...and we can remove the explicit forms for existentials from the language definition.
- Pack: – A package is an abstraction that needs a result type Y and a continuation f.

$$\{*S, t\} \ \texttt{as} \ \{\exists X, T\} \stackrel{\text{def}}{=} \lambda Y.\, \lambda f : (\forall X.\, T \to Y).\ \ldots$$

A package can only be an abstraction.

Encoding Existentials

- With 2nd-order types, we can encode:

$$\{\exists X, T\} \stackrel{\text{def}}{=} \forall Y.\, (\forall X.\, T \to Y) \to Y$$

- ...and we can remove the explicit forms for existentials from the language definition.
- Pack: – A package is an abstraction that needs a result type Y and a continuation f.

$$\{*S, t\} \texttt{ as } \{\exists X, T\} \stackrel{\text{def}}{=} \lambda Y.\, \lambda f : (\forall X.\, T \to Y).\ \ldots$$ A package can only be an abstraction.

$$\stackrel{\text{def}}{=} \lambda Y.\, \lambda f : (\forall X.\, T \to Y).\ f\ [S]\ \ldots$$ The continuation needs the type of the existential. (See T-PACK)

Encoding Existentials

- With 2nd-order types, we can encode:

$$\{\exists X, T\} \stackrel{\text{def}}{=} \forall Y.\, (\forall X.\, T \to Y) \to Y$$

- ...and we can remove the explicit forms for existentials from the language definition.
- Pack: – A package is an abstraction that needs a result type Y and a continuation f.

$$\{*S, t\} \texttt{ as } \{\exists X, T\} \stackrel{\text{def}}{=} \lambda Y.\, \lambda f : (\forall X.\, T \to Y).\ \ldots$$

A package can only be an abstraction.

$$\stackrel{\text{def}}{=} \lambda Y.\, \lambda f : (\forall X.\, T \to Y).\ f\ [S]\ \ldots$$

The continuation needs the type of the existential. (See T-PACK)

$$\stackrel{\text{def}}{=} \lambda Y.\, \lambda f : (\forall X.\, T \to Y).\ f\ [S]\ t$$

The continuation needs the existential value.

Encoding Existentials

- With 2nd-order types, we can encode:

$$\{\exists X, T\} \stackrel{\text{def}}{=} \forall Y.\, (\forall X.\, T \to Y) \to Y$$

- ...and we can remove the explicit forms for existentials from the language definition.
- Pack: – A package is an abstraction that needs a result type Y and a continuation f.

$$\begin{aligned}\{*S, t\}\ \texttt{as}\ \{\exists X, T\} &\stackrel{\text{def}}{=} \lambda Y.\, \lambda f : (\forall X.\, T \to Y).\ \ldots && \text{A package can only be an abstraction.}\\ &\stackrel{\text{def}}{=} \lambda Y.\, \lambda f : (\forall X.\, T \to Y).\ f\,[S]\ \ldots && \text{The continuation needs the type of the existential. (See T-PACK)}\\ &\stackrel{\text{def}}{=} \lambda Y.\, \lambda f : (\forall X.\, T \to Y).\ f\,[S]\,t && \text{The continuation needs the existential value.}\end{aligned}$$

- Unpack: – Unpack with application to a continuation.

Encoding Existentials

- With 2nd-order types, we can encode:

$$\{\exists X, T\} \stackrel{\text{def}}{=} \forall Y.\, (\forall X.\, T \to Y) \to Y$$

- ...and we can remove the explicit forms for existentials from the language definition.
- Pack: – A package is an abstraction that needs a result type Y and a continuation f.

$$\{*S, t\}\ \texttt{as}\ \{\exists X, T\} \stackrel{\text{def}}{=} \lambda Y.\, \lambda f : (\forall X.\, T \to Y).\ \ldots$$
A package can only be an abstraction.

$$\stackrel{\text{def}}{=} \lambda Y.\, \lambda f : (\forall X.\, T \to Y).\ f\ [S]\ \ldots$$
The continuation needs the type of the existential. (See T-PACK)

$$\stackrel{\text{def}}{=} \lambda Y.\, \lambda f : (\forall X.\, T \to Y).\ f\ [S]\ t$$
The continuation needs the existential value.

- Unpack: – Unpack with application to a continuation.

$$\texttt{let}\ \{X, x\} = t_1\ \texttt{in}\ t_2 \stackrel{\text{def}}{=} t_1\ \ldots$$
Remember: an existential is an abstraction.

$$\stackrel{\text{def}}{=} t_1\ [T_2]\ \ldots$$
The result type of the continuation.

Encoding Existentials

- With 2nd-order types, we can encode:

$$\{\exists X, T\} \stackrel{\text{def}}{=} \forall Y.\, (\forall X.\, T \to Y) \to Y$$

- ...and we can remove the explicit forms for existentials from the language definition.
- Pack: – A package is an abstraction that needs a result type Y and a continuation f.

$$\begin{aligned} \{*S, t\}\ \texttt{as}\ \{\exists X, T\} &\stackrel{\text{def}}{=} \lambda Y.\, \lambda f : (\forall X.\, T \to Y).\ \ldots && \text{A package can only be an abstraction.} \\ &\stackrel{\text{def}}{=} \lambda Y.\, \lambda f : (\forall X.\, T \to Y).\ f\ [S]\ \ldots && \text{The continuation needs the type of the existential. (See T-PACK)} \\ &\stackrel{\text{def}}{=} \lambda Y.\, \lambda f : (\forall X.\, T \to Y).\ f\ [S]\ t && \text{The continuation needs the existential value.} \end{aligned}$$

- Unpack: – Unpack with application to a continuation.

$$\begin{aligned} \texttt{let}\ \{X, x\} = t_1\ \texttt{in}\ t_2 &\stackrel{\text{def}}{=} t_1\ \ldots && \text{Remember: an existential is an abstraction.} \\ &\stackrel{\text{def}}{=} t_1\ [T_2]\ \ldots && \text{The result type of the continuation.} \\ &\stackrel{\text{def}}{=} t_1\ [T_2]\ (\lambda X.\, \lambda x : T_{11}.\, t_2) && \text{The continuation itself.} \end{aligned}$$

Impredicativity

- System F implements *impredicative* polymorphism.
- $T = \forall X. X \to X$ ranges over all types, even T itself!

System F in Haskell

- Haskell implements System FC[1], an extension of system F.
- Haskell has all the described features, even visible type applications, ...
- ...and still provides type inference based on an extension of HM![2]

[1] Martin Sulzmann et al. "System F with type equality coercions". In: *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*. 2007.

[2] Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. "Visible Type Application". In: *Programming Languages and Systems*. Springer Berlin Heidelberg, 2016.

Higher-Order Types
Intuitions

- We have seen (both in the lecture and in the exercise) types such as

Higher-Order Types

Intuitions

- We have seen (both in the lecture and in the exercise) types such as
  - `Bool`, `Pair a b`, `List a`,...

Higher-Order Types
Intuitions

- We have seen (both in the lecture and in the exercise) types such as
  - `Bool`, `Pair a b`, `List a`,...
- Typing them in System F and Haskell (and also in Coq, Lean, etc.) would yield:

Higher-Order Types
Intuitions

- We have seen (both in the lecture and in the exercise) types such as
  - `Bool`, `Pair a b`, `List a`, ...
- Typing them in System F and Haskell (and also in Coq, Lean, etc.) would yield:
  - $\mathtt{CBool} = \forall X. X \to X \to X$

Higher-Order Types
Intuitions

- We have seen (both in the lecture and in the exercise) types such as
  - `Bool`, `Pair a b`, `List a`, ...
- Typing them in System F and Haskell (and also in Coq, Lean, etc.) would yield:
  - $\texttt{CBool} = \forall X. X \to X \to X$
  - $\texttt{tru} = \lambda X.\ \lambda x : X.\ \lambda y : X.\ x : \texttt{CBool}$ $\qquad$ $\texttt{fls} = \lambda X.\ \lambda x : X.\ \lambda y : X.\ y : \texttt{CBool}$

Higher-Order Types

Intuitions

- We have seen (both in the lecture and in the exercise) types such as
  - `Bool`, `Pair a b`, `List a`, ...
- Typing them in System F and Haskell (and also in Coq, Lean, etc.) would yield:
  - $\texttt{CBool} = \forall X. X \to X \to X$
  - $\texttt{tru} = \lambda X.\ \lambda x : X.\ \lambda y : X.\ x : \texttt{CBool}$ $\quad$ $\texttt{fls} = \lambda X.\ \lambda x : X.\ \lambda y : X.\ y : \texttt{CBool}$
  - `data Bool = True | False`

Higher-Order Types

Intuitions

- We have seen (both in the lecture and in the exercise) types such as
  - `Bool`, `Pair a b`, `List a`,...
- Typing them in System F and Haskell (and also in Coq, Lean, etc.) would yield:
  - $\texttt{CBool} = \forall X. X \to X \to X$
  - $\texttt{tru} = \lambda X.\, \lambda x : X.\, \lambda y : X.\, x : \texttt{CBool}$ $\quad$ $\texttt{fls} = \lambda X.\, \lambda x : X.\, \lambda y : X.\, y : \texttt{CBool}$
  - `data Bool = True | False`
  - $\texttt{Pair}\ X\ Y = \forall Z.\, (X \to Y \to Z) \to Z$

Higher-Order Types
Intuitions

- We have seen (both in the lecture and in the exercise) types such as
  - `Bool`, `Pair a b`, `List a`, ...
- Typing them in System F and Haskell (and also in Coq, Lean, etc.) would yield:
  - $\texttt{CBool} = \forall X. X \to X \to X$
  - $\texttt{tru} = \lambda X.\, \lambda x : X.\, \lambda y : X.\, x : \texttt{CBool}$ $\qquad$ $\texttt{fls} = \lambda X.\, \lambda x : X.\, \lambda y : X.\, y : \texttt{CBool}$
  - `data Bool = True | False`
  - $\texttt{Pair}\; X\; Y = \forall Z.\, (X \to Y \to Z) \to Z$
  - `data Pair a b = Pair a b`

- We have seen (both in the lecture and in the exercise) types such as
  - `Bool`, `Pair a b`, `List a`, ...
- Typing them in System F and Haskell (and also in Coq, Lean, etc.) would yield:
  - $\texttt{CBool} = \forall X. X \to X \to X$
  - $\texttt{tru} = \lambda X.\, \lambda x : X.\, \lambda y : X.\, x : \texttt{CBool}$ $\quad$ $\texttt{fls} = \lambda X.\, \lambda x : X.\, \lambda y : X.\, y : \texttt{CBool}$
  - `data Bool = True | False`
  - $\texttt{Pair}\; X\; Y = \forall Z.\, (X \to Y \to Z) \to Z$
  - `data Pair a b = Pair a b`
- Abbreviations:

Higher-Order Types

Intuitions

- We have seen (both in the lecture and in the exercise) types such as
  - `Bool`, `Pair a b`, `List a`,...
- Typing them in System F and Haskell (and also in Coq, Lean, etc.) would yield:
  - $\texttt{CBool} = \forall X.X \to X \to X$
  - $\texttt{tru} = \lambda X.\, \lambda x : X.\, \lambda y : X.\, x : \texttt{CBool}$ $\quad$ $\texttt{fls} = \lambda X.\, \lambda x : X.\, \lambda y : X.\, y : \texttt{CBool}$
  - `data Bool = True | False`
  - $\texttt{Pair}\; X\; Y = \forall Z.\, (X \to Y \to Z) \to Z$
  - `data Pair a b = Pair a b`
- Abbreviations:

  Simple `CBool`

Higher-Order Types
Intuitions

- We have seen (both in the lecture and in the exercise) types such as
  - `Bool`, `Pair a b`, `List a`, ...
- Typing them in System F and Haskell (and also in Coq, Lean, etc.) would yield:
  - $\texttt{CBool} = \forall X. X \to X \to X$
  - $\texttt{tru} = \lambda X.\, \lambda x : X.\, \lambda y : X.\, x : \texttt{CBool}$ $\quad$ $\texttt{fls} = \lambda X.\, \lambda x : X.\, \lambda y : X.\, y : \texttt{CBool}$
  - `data Bool = True | False`
  - $\texttt{Pair}\ X\ Y = \forall Z.\, (X \to Y \to Z) \to Z$
  - `data Pair a b = Pair a b`
- Abbreviations:

Simple	$\texttt{CBool}$
Parametric	$\texttt{Pair} = \lambda X.\, \lambda Y.\, \forall Z.\, (X \to Y \to Z) \to Z$

Higher-Order Types

Intuitions

- We have seen (both in the lecture and in the exercise) types such as
  - `Bool`, `Pair a b`, `List a`, ...
- Typing them in System F and Haskell (and also in Coq, Lean, etc.) would yield:
  - $\texttt{CBool} = \forall X. X \to X \to X$
  - $\texttt{tru} = \lambda X.\, \lambda x : X.\, \lambda y : X.\, x : \texttt{CBool}$ $\quad$ $\texttt{fls} = \lambda X.\, \lambda x : X.\, \lambda y : X.\, y : \texttt{CBool}$
  - `data Bool = True | False`
  - $\texttt{Pair}\ X\ Y = \forall Z.\, (X \to Y \to Z) \to Z$
  - `data Pair a b = Pair a b`
- Abbreviations:

  Simple $\texttt{CBool}$

  Parametric $\texttt{Pair} = \lambda X.\, \lambda Y.\, \forall Z.\, (X \to Y \to Z) \to Z$

- `Pair` is essentially a type-level function, called a *type operator*.

Higher-Order Types
Intuitions

- We have seen (both in the lecture and in the exercise) types such as
  - `Bool`, `Pair a b`, `List a`,...
- Typing them in System F and Haskell (and also in Coq, Lean, etc.) would yield:
  - $\texttt{CBool} = \forall X. X \to X \to X$
  - $\texttt{tru} = \lambda X.\, \lambda x : X.\, \lambda y : X.\, x : \texttt{CBool}$ $\quad$ $\texttt{fls} = \lambda X.\, \lambda x : X.\, \lambda y : X.\, y : \texttt{CBool}$
  - `data Bool = True | False`
  - $\texttt{Pair}\ X\ Y = \forall Z.\, (X \to Y \to Z) \to Z$
  - `data Pair a b = Pair a b`
- Abbreviations:

Simple	$\texttt{CBool}$
Parametric	$\texttt{Pair} = \lambda X.\, \lambda Y.\, \forall Z.\, (X \to Y \to Z) \to Z$

- `Pair` is essentially a type-level function, called a *type operator*.
- To make sure that these type-level functions are well-typed, we will lift STLC into the type-level!

Higher-Order Types
Intuitions

- We have seen (both in the lecture and in the exercise) types such as
  - `Bool`, `Pair a b`, `List a`,...
- Typing them in System F and Haskell (and also in Coq, Lean, etc.) would yield:
  - $\texttt{CBool} = \forall X.X \to X \to X$
  - $\texttt{tru} = \lambda X.\, \lambda x : X.\, \lambda y : X.\, x : \texttt{CBool}$ $\qquad \texttt{fls} = \lambda X.\, \lambda x : X.\, \lambda y : X.\, y : \texttt{CBool}$
  - `data Bool = True | False`
  - $\texttt{Pair}\ X\ Y = \forall Z.\, (X \to Y \to Z) \to Z$
  - `data Pair a b = Pair a b`
- Abbreviations:

$$\begin{array}{rl} \text{Simple} & \texttt{CBool} \\ \text{Parametric} & \texttt{Pair} = \lambda X.\, \lambda Y.\, \forall Z.\, (X \to Y \to Z) \to Z \end{array}$$

- `Pair` is essentially a type-level function, called a *type operator*.
- To make sure that these type-level functions are well-typed, we will lift STLC into the type-level!
- The types of types are referred to as *kinds*.

From Types to Kinds

Terms

From Types to Kinds

5

Terms

From Types to Kinds

Terms

5

$(\lambda x : \texttt{Nat}.\, x)\ 5$

From Types to Kinds

5 $\lambda x : \mathtt{Nat}.x$

$(\lambda x : \mathtt{Nat}.\, x)\ 5$

Terms

From Types to Kinds

Terms

5

$\lambda x : \texttt{Nat}.x$

$(\lambda x : \texttt{Nat}.\ x)\ 5$

$\lambda X.\ \lambda x : X.\ x$

From Types to Kinds

5 $\qquad \lambda x : \texttt{Nat}. x$ $\qquad \texttt{pair}\ [\texttt{Nat}]\ [\texttt{Bool}]\ 5\ \texttt{false}$

$(\lambda x : \texttt{Nat}.\ x)\ 5$ $\qquad \lambda X.\ \lambda x : X.\ x$

Terms

From Types to Kinds

Terms

5 $\quad \lambda x : \texttt{Nat}. x$ $\quad \texttt{pair}\ [\texttt{Nat}]\ [\texttt{Bool}]\ 5\ \texttt{false}$

$(\lambda x : \texttt{Nat}.\ x)\ 5$ $\quad \lambda X.\ \lambda x : X.\ x$ $\quad (\lambda x : \texttt{Nat}.\ x)\ \texttt{true}$

From Types to Kinds

Types

Terms

5 $\quad \lambda x : \texttt{Nat}. x$ $\quad \texttt{pair}\ [\texttt{Nat}]\ [\texttt{Bool}]\ 5\ \texttt{false}$

$(\lambda x : \texttt{Nat}.\ x)\ 5$ $\quad \lambda X.\ \lambda x : X.\ x$ $\quad (\lambda x : \texttt{Nat}.\ x)\ \texttt{true}$

From Types to Kinds

$\texttt{Nat}$

Types

Terms

5

$\lambda x : \texttt{Nat}. x$

$\texttt{pair}\ [\texttt{Nat}]\ [\texttt{Bool}]\ 5\ \texttt{false}$

$(\lambda x : \texttt{Nat}.\ x)\ \texttt{true}$

$(\lambda x : \texttt{Nat}.\ x)\ 5$

$\lambda X.\ \lambda x : X.\ x$

proper type

From Types to Kinds

$\texttt{Nat}$

$\texttt{Nat} \longrightarrow \texttt{Nat}$

Types

Terms

5

$\lambda x : \texttt{Nat}. x$

$\texttt{pair}\ [\texttt{Nat}]\ [\texttt{Bool}]\ 5\ \texttt{false}$

$(\lambda x : \texttt{Nat}.\ x)\ \texttt{true}$

$(\lambda x : \texttt{Nat}.\ x)\ 5$

$\lambda X. \lambda x : X.\ x$

proper type

From Types to Kinds

Types

$\texttt{Nat}$

$\texttt{Nat} \to \texttt{Nat}$

$(\lambda X.\, X \to X)\ \texttt{Nat}$

Terms

5

$\lambda x : \texttt{Nat}. x$

$\texttt{pair}\ [\texttt{Nat}]\ [\texttt{Bool}]\ 5\ \texttt{false}$

$(\lambda x : \texttt{Nat}.\, x)\ \texttt{true}$

$(\lambda x : \texttt{Nat}.\, x)\ 5$

$\lambda X.\, \lambda x : X.\, x$

proper type

From Types to Kinds

Types

$\texttt{Nat}$

$\texttt{Nat} \to \texttt{Nat}$

$\forall X.\, X \to X$

$(\lambda X.\, X \to X)\ \texttt{Nat}$

Terms

5

$\lambda x : \texttt{Nat}.x$

$\texttt{pair}\ [\texttt{Nat}]\ [\texttt{Bool}]\ 5\ \texttt{false}$

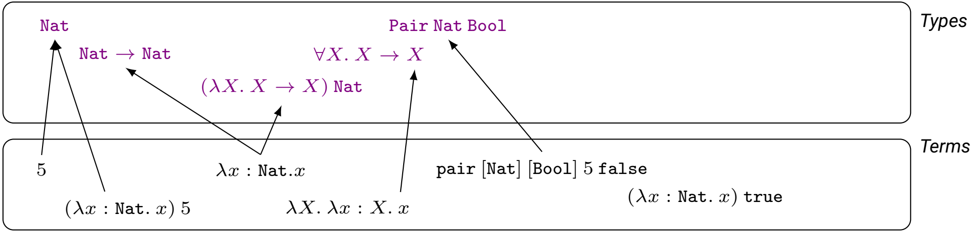
$(\lambda x : \texttt{Nat}.\, x)\ \texttt{true}$

$(\lambda x : \texttt{Nat}.\, x)\ 5$

$\lambda X.\, \lambda x : X.\, x$

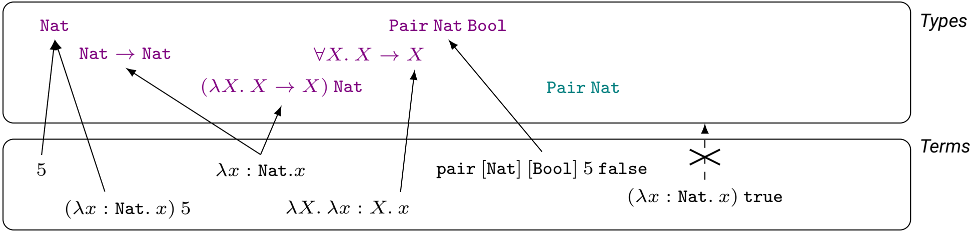
proper type

From Types to Kinds



proper type

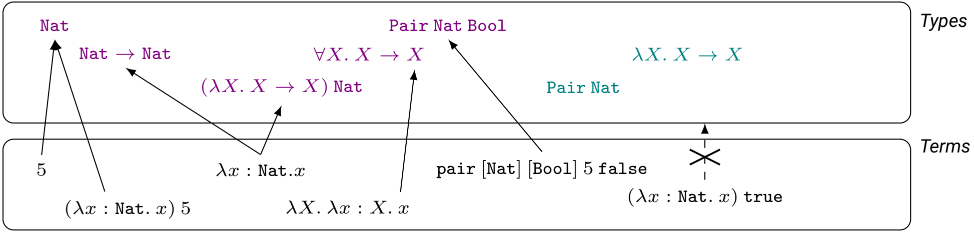
From Types to Kinds



proper type

type operator

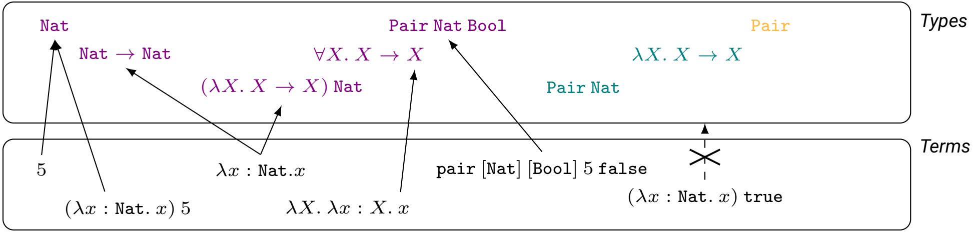
From Types to Kinds



proper type

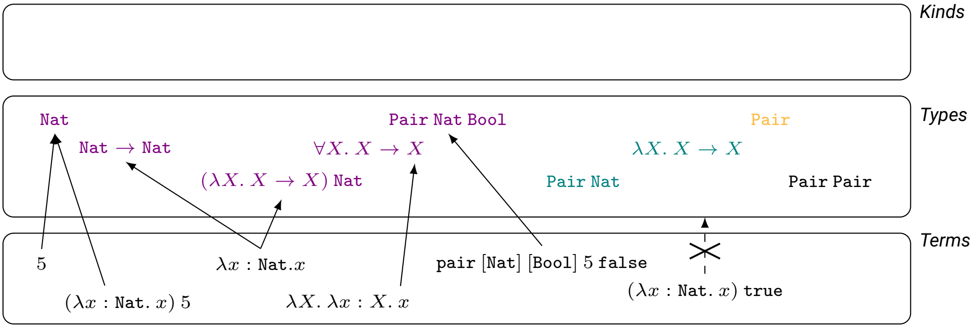
type operator

From Types to Kinds



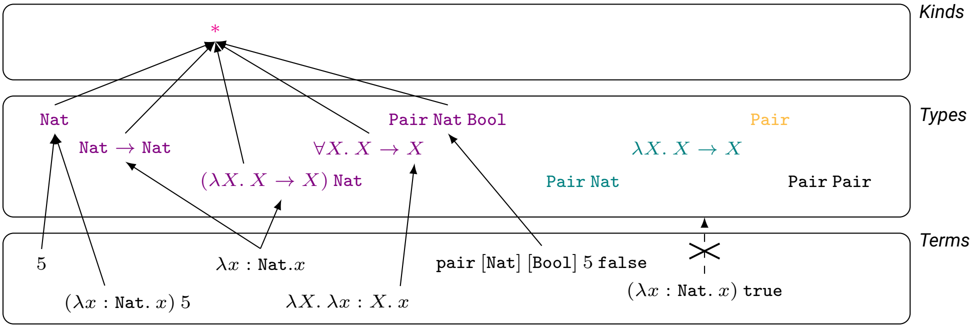
proper type   type operator   function from proper type to type operator

From Types to Kinds



proper type   type operator   function from proper type to type operator

From Types to Kinds



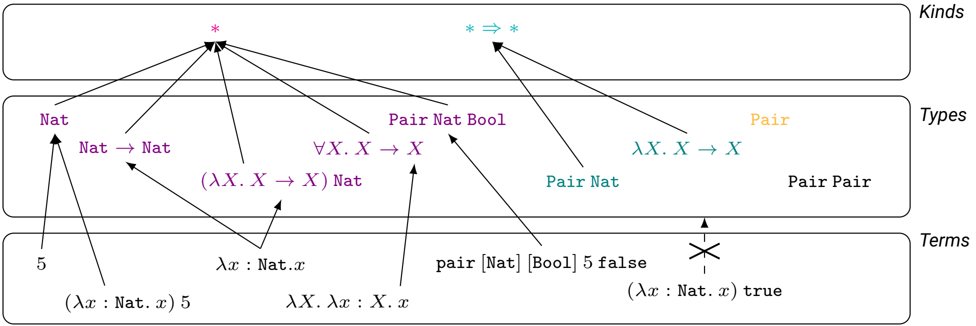
proper type

type operator

function from proper type to type operator

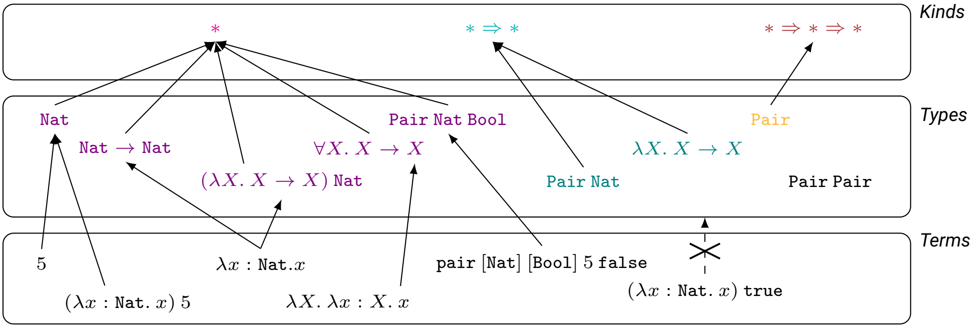
kind of proper types

From Types to Kinds



proper type
type operator
function from proper type to type operator
kind of proper types
kind of type operators

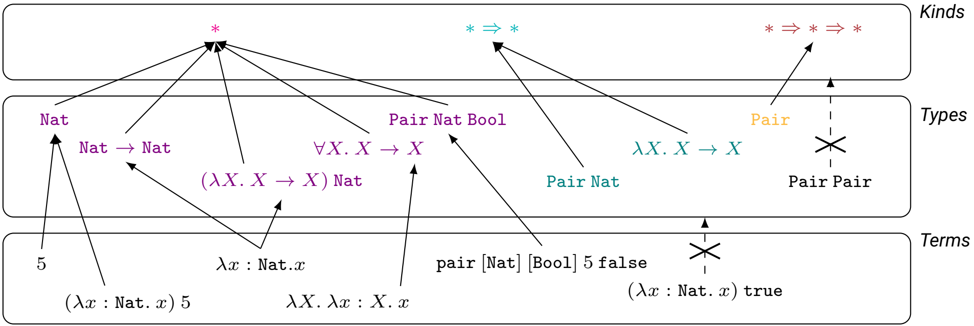
From Types to Kinds



proper type type operator function from proper type to type operator

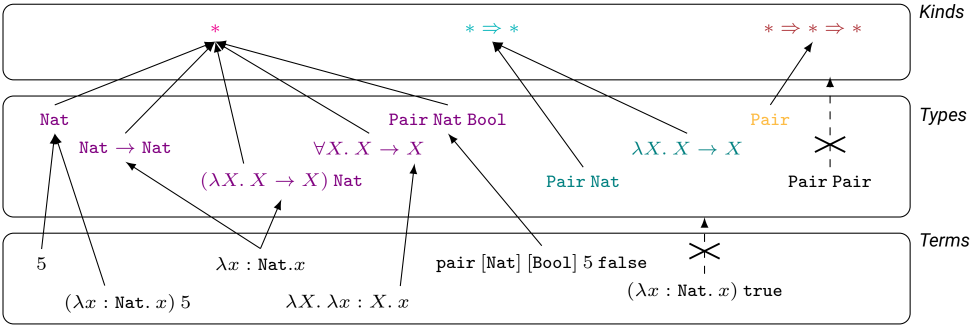
kind of proper types kind of type operators kind of two-argument type operators

From Types to Kinds



proper type
type operator
function from proper type to type operator
kind of proper types
kind of type operators
kind of two-argument type operators

From Types to Kinds



proper type | type operator | function from proper type to type operator

kind of proper types | kind of type operators | kind of two-argument type operators

- Note the uninhabited types in this figure!

The Systems of Todays Lecture

λ_ω	Monomorphic kinds (,i.e., kinding without quantifiers)
System F_ω	Polymorphic kinds

Syntax

- We extend STLC with type operators:

t	$::=$		terms:
	$\vert$	x	variable
	$\vert$	$\lambda x : T.t$	abstraction
	$\vert$	$t\ t$	application
v	$::=$		values:
	$\vert$	$\lambda x : T.t$	abstraction value

T	$::=$		types:
	$\vert$	X	type variable
	$\vert$	$\lambda X :: K.T$	operator abstraction
	$\vert$	$T\ T$	operator application
	$\vert$	$T \to T$	type of functions
Γ	$::=$		contexts:
	$\vert$	$\varnothing$	empty context
	$\vert$	$\Gamma, x : T$	term variable binding
	$\vert$	$\Gamma, X :: K$	type variable binding
K	$::=$		kinds:
	$\vert$	$*$	kind of proper types
	$\vert$	$K \Rightarrow K$	kind of operators

- For conciseness, we will not assume base kinds. But we will kinds later.

Evaluation

$$\boxed{t \longrightarrow t'}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \ \longrightarrow\ t_1'\ t_2} \text{ E-APP1} \qquad \frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \ \longrightarrow\ v_1\ t_2'} \text{ E-APP2}$$

$$\frac{}{(\lambda x : T.t_{12})\ v_2 \ \longrightarrow\ [x \mapsto v_2]t_{12}} \text{ E-APPABS}$$

Typing and Kinding

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

Typing and Kinding

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

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \textsc{T-Var} \qquad \frac{\Gamma \vdash T_1 :: * \qquad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2}\ \textsc{T-Abs} \qquad \frac{\Gamma, t_1 : T_{11} \to T_{12} \qquad t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}\ \textsc{T-App}$$

Typing and Kinding

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

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ T-VAR} \qquad \frac{\Gamma \vdash T_1 :: * \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2} \text{ T-ABS} \qquad \frac{\Gamma, t_1 : T_{11} \to T_{12} \quad t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \text{ T-APP}$$

$$\frac{\Gamma \vdash t : S \quad S \equiv T \quad \Gamma \vdash T :: *}{\Gamma \vdash t : T} \text{ T-EQ}$$

Typing and Kinding

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

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ T-VAR} \qquad \frac{\Gamma \vdash T_1 :: * \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2 : T_1 \to T_2} \text{ T-ABS} \qquad \frac{\Gamma, t_1 : T_{11} \to T_{12} \quad t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \text{ T-APP}$$

$$\frac{\Gamma \vdash t : S \quad S \equiv T \quad \Gamma \vdash T :: *}{\Gamma \vdash t : T} \text{ T-EQ}$$

$\boxed{\Gamma \vdash T :: K}$

λ_ω

Typing and Kinding

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

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ T-VAR} \qquad \frac{\Gamma \vdash T_1 :: * \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2} \text{ T-ABS} \qquad \frac{\Gamma, t_1 : T_{11} \to T_{12} \quad t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \text{ T-APP}$$

$$\frac{\Gamma \vdash t : S \quad S \equiv T \quad \Gamma \vdash T :: *}{\Gamma \vdash t : T} \text{ T-EQ}$$

$\boxed{\Gamma \vdash T :: K}$

$$\frac{X :: K \in \Gamma}{\Gamma \vdash X :: K} \text{ T-TVAR}$$

Typing and Kinding

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

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ T-Var} \qquad \frac{\Gamma \vdash T_1 :: * \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2} \text{ T-Abs} \qquad \frac{\Gamma, t_1 : T_{11} \to T_{12} \quad t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \text{ T-App}$$

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

$\boxed{\Gamma \vdash T :: K}$

$$\frac{X :: K \in \Gamma}{\Gamma \vdash X :: K} \text{ T-TVar} \qquad \frac{\Gamma, X :: K_1 \vdash T_2 :: K_2}{\Gamma \vdash \lambda X :: K_1 . T_2 :: K_1 \Rightarrow K_2} \text{ K-Abs}$$

Typing and Kinding

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

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ T-VAR} \qquad \frac{\Gamma \vdash T_1 :: * \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2} \text{ T-ABS} \qquad \frac{\Gamma, t_1 : T_{11} \to T_{12} \quad t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \text{ T-APP}$$

$$\frac{\Gamma \vdash t : S \quad S \equiv T \quad \Gamma \vdash T :: *}{\Gamma \vdash t : T} \text{ T-EQ}$$

$\boxed{\Gamma \vdash T :: K}$

$$\frac{X :: K \in \Gamma}{\Gamma \vdash X :: K} \text{ T-TVAR} \qquad \frac{\Gamma, X :: K_1 \vdash T_2 :: K_2}{\Gamma \vdash \lambda X :: K_1 . T_2 :: K_1 \Rightarrow K_2} \text{ K-ABS} \qquad \frac{\Gamma, T_1 :: K_{11} \Rightarrow K_{12} \quad T_2 : K_{11}}{\Gamma \vdash T_1\ T_2 :: K_{12}} \text{ K-APP}$$

λ_ω

Typing and Kinding

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

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \textsc{T-Var} \qquad \frac{\Gamma \vdash T_1 :: * \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2}\ \textsc{T-Abs} \qquad \frac{\Gamma, t_1 : T_{11} \to T_{12} \quad t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}\ \textsc{T-App}$$

$$\frac{\Gamma \vdash t : S \quad S \equiv T \quad \Gamma \vdash T :: *}{\Gamma \vdash t : T}\ \textsc{T-Eq}$$

$\boxed{\Gamma \vdash T :: K}$

$$\frac{X :: K \in \Gamma}{\Gamma \vdash X :: K}\ \textsc{T-TVar} \qquad \frac{\Gamma, X :: K_1 \vdash T_2 :: K_2}{\Gamma \vdash \lambda X :: K_1 . T_2 :: K_1 \Rightarrow K_2}\ \textsc{K-Abs} \qquad \frac{\Gamma, T_1 :: K_{11} \Rightarrow K_{12} \quad T_2 : K_{11}}{\Gamma \vdash T_1\ T_2 :: K_{12}}\ \textsc{K-App}$$

λ_ω

Typing and Kinding

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

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \textsc{T-Var} \qquad \frac{\Gamma \vdash T_1 :: * \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2}\ \textsc{T-Abs} \qquad \frac{\Gamma, t_1 : T_{11} \to T_{12} \quad t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}\ \textsc{T-App}$$

$$\frac{\Gamma \vdash t : S \quad S \equiv T \quad \Gamma \vdash T :: *}{\Gamma \vdash t : T}\ \textsc{T-Eq}$$

$\boxed{\Gamma \vdash T :: K}$

$$\frac{X :: K \in \Gamma}{\Gamma \vdash X :: K}\ \textsc{T-TVar} \qquad \frac{\Gamma, X :: K_1 \vdash T_2 :: K_2}{\Gamma \vdash \lambda X :: K_1 . T_2 :: K_1 \Rightarrow K_2}\ \textsc{K-Abs} \qquad \frac{\Gamma, T_1 :: K_{11} \Rightarrow K_{12} \quad T_2 : K_{11}}{\Gamma \vdash T_1\ T_2 :: K_{12}}\ \textsc{K-App}$$

$$\frac{\Gamma \vdash T_1 :: * \quad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \to T_2 :: *}\ \textsc{K-Arrow}$$

Type Equivalence
Intuition

- In a type checker for STLC, we would implement:

Type Equivalence
Intuition

- In a type checker for STLC, we would implement:

```
typeOf ctxt (TmApp t1 t2) =
  let tyT1 = typeOf ctxt t1
      tyT2 = typeOf ctxt t2
  in case tyT1 of
       (TyArr tyT11 tyT12) ->
         if tyT2 == tyT11
         then tyT2
         else error "type mismatch"
```

Type Equivalence
Intuition

- In a type checker for STLC, we would implement:

```
typeOf ctxt (TmApp t1 t2) =
  let tyT1 = typeOf ctxt t1
      tyT2 = typeOf ctxt t2
  in case tyT1 of
       (TyArr tyT11 tyT12) ->
         if tyT2 == tyT11
         then tyT2
         else error "type mismatch"
```

Type Equivalence
Intuition

- In a type checker for STLC, we would implement:

```
typeOf ctxt (TmApp t1 t2) =
  let tyT1 = typeOf ctxt t1
      tyT2 = typeOf ctxt t2
  in case tyT1 of
       (TyArr tyT11 tyT12) ->
         if tyT2 == tyT11
         then tyT2
         else error "type mismatch"
```

$$\boxed{\Gamma \vdash T \equiv T}$$

Axioms

$$\frac{}{\Gamma \vdash \texttt{Nat} \equiv \texttt{Nat}} \quad \text{QT-NAT}$$

$$\frac{}{\Gamma \vdash \texttt{Bool} \equiv \texttt{Bool}} \quad \text{OT-BOOL}$$

Congruence

$$\frac{\Gamma \vdash T_1 \equiv S_1 \qquad \Gamma \vdash T_2 \equiv S_2}{\Gamma \vdash T_1 \to T_2 \equiv S_1 \to S_2} \quad \text{OT-ABS}$$

- (Remember, in HM, we would not do this comparison directly.)

Type Equivalence

Intuition

- In a type checker for STLC, we would implement:

```
typeOf ctxt (TmApp t1 t2) =
  let tyT1 = typeOf ctxt t1
      tyT2 = typeOf ctxt t2
  in case tyT1 of
       (TyArr tyT11 tyT12) ->
         if tyT2 == tyT11
         then tyT2
         else error "type mismatch"
```

$$\boxed{\Gamma \vdash T \equiv T}$$

Axioms

$$\frac{}{\Gamma \vdash \texttt{Nat} \equiv \texttt{Nat}} \text{ QT-NAT}$$

$$\frac{}{\Gamma \vdash \texttt{Bool} \equiv \texttt{Bool}} \text{ OT-BOOL}$$

Congruence

$$\frac{\Gamma \vdash T_1 \equiv S_1 \quad \Gamma \vdash T_2 \equiv S_2}{\Gamma \vdash T_1 \to T_2 \equiv S_1 \to S_2} \text{ OT-ABS}$$

- (Remember, in HM, we would not do this comparison directly.)
- We want to *define* that $\texttt{Nat} \to \texttt{Nat} \equiv (\lambda X.\, X \to X)\ \texttt{Nat}$.

λ_ω

Type Equivalence

- *Definitional equivalence*:

$$\boxed{S \equiv T}$$

Type Equivalence

- *Definitional equivalence*:

$$\boxed{S \equiv T}$$

$$\frac{}{T \equiv T}\ \text{Q-Refl}$$

Type Equivalence

- *Definitional equivalence*:

$$\boxed{S \equiv T}$$

$$\frac{}{T \equiv T}\ \textsc{Q-Refl} \qquad \frac{T \equiv S}{S \equiv T}\ \textsc{Q-Symm}$$

Type Equivalence

- *Definitional equivalence*:

$\boxed{S \equiv T}$

$$\frac{}{T \equiv T}\ \textsc{Q-Refl} \qquad \frac{T \equiv S}{S \equiv T}\ \textsc{Q-Symm} \qquad \frac{S \equiv U \quad U \equiv T}{S \equiv T}\ \textsc{Q-Trans}$$

Type Equivalence

- *Definitional equivalence*:

$$\boxed{S \equiv T}$$

$$\frac{}{T \equiv T}\ \text{Q-Refl} \qquad \frac{T \equiv S}{S \equiv T}\ \text{Q-Symm} \qquad \frac{S \equiv U \quad U \equiv T}{S \equiv T}\ \text{Q-Trans} \qquad \frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 \to S_2 \equiv T_1 \to T_2}\ \text{Q-Arrow}$$

Type Equivalence

- *Definitional equivalence*:

$$\boxed{S \equiv T}$$

$$\frac{}{T \equiv T}\ \text{Q-REFL} \qquad \frac{T \equiv S}{S \equiv T}\ \text{Q-SYMM} \qquad \frac{S \equiv U \quad U \equiv T}{S \equiv T}\ \text{Q-TRANS} \qquad \frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 \to S_2 \equiv T_1 \to T_2}\ \text{Q-ARROW}$$

$$\frac{S_2 \equiv T_2}{\lambda X :: K_1 . S_2 \equiv \lambda X :: K_1 . T_2}\ \text{Q-ABS}$$

Type Equivalence

- *Definitional equivalence*:

$$\boxed{S \equiv T}$$

$$\frac{}{T \equiv T}\ \text{Q-Refl} \qquad \frac{T \equiv S}{S \equiv T}\ \text{Q-Symm} \qquad \frac{S \equiv U \quad U \equiv T}{S \equiv T}\ \text{Q-Trans} \qquad \frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 \to S_2 \equiv T_1 \to T_2}\ \text{Q-Arrow}$$

$$\frac{S_2 \equiv T_2}{\lambda X :: K_1.S_2 \equiv \lambda X :: K_1.T_2}\ \text{Q-Abs} \qquad \frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1\ S_2 \equiv T_1\ T_2}\ \text{Q-App}$$

Type Equivalence

- *Definitional equivalence*:

$$\boxed{S \equiv T}$$

$$\frac{}{T \equiv T}\ \text{Q-Refl} \qquad \frac{T \equiv S}{S \equiv T}\ \text{Q-Symm} \qquad \frac{S \equiv U \quad U \equiv T}{S \equiv T}\ \text{Q-Trans} \qquad \frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 \to S_2 \equiv T_1 \to T_2}\ \text{Q-Arrow}$$

$$\frac{S_2 \equiv T_2}{\lambda X :: K_1.S_2 \equiv \lambda X :: K_1.T_2}\ \text{Q-Abs} \qquad \frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1\ S_2 \equiv T_1\ T_2}\ \text{Q-App}$$

$$\frac{}{(\lambda X :: K_{11}.T_{12})\ T_2 \equiv [X \mapsto T_2]T_{12}}\ \text{Q-AppAbs}$$

No more mysteric data types

- We were mumbling about the syntax of these two data type definitions (to encode relations):

No more mysteric data types

- We were mumbling about the syntax of these two data type definitions (to encode relations):

(recursive) ADT

```
data Nat = O | S Nat
```

GADT

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

parameterized

```
data List a :: Type where
    Nil :: List a
    Cons :: a -> List a -> List a
```

annotated/indexed

```
data Ev :: Nat -> Type where
    EvO :: Ev 'O
    EvSS :: Ev n -> Ev ('S ('S n))
```

No more mysteric data types

- We were mumbling about the syntax of these two data type definitions (to encode relations):

(recursive) ADT

```
data Nat = O | S Nat
```

GADT

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

parameterized

```
data List a :: Type where
    Nil :: List a
    Cons :: a -> List a -> List a
```

annotated/indexed

```
data Ev :: Nat -> Type where
    EvO :: Ev 'O
    EvSS :: Ev n -> Ev ('S ('S n))
```

- Now that we know what kinds are and what type operators are, we see that:

No more mysteric data types

- We were mumbling about the syntax of these two data type definitions (to encode relations):

(recursive) ADT	`data Nat = O | S Nat`

GADT	`data Nat where`
	`    O :: Nat`
	`    S :: Nat -> Nat`

parameterized	`data List a :: Type where`
	`    Nil :: List a`
	`    Cons :: a -> List a -> List a`

annotated/indexed	`data Ev :: Nat -> Type where`
	`    EvO :: Ev 'O`
	`    EvSS :: Ev n -> Ev ('S ('S n))`

- Now that we know what kinds are and what type operators are, we see that:
  - There is nothing special about their syntax!

No more mysteric data types

- We were mumbling about the syntax of these two data type definitions (to encode relations):

(recursive) ADT	`data Nat = O | S Nat`	`type Nat :: Type`

GADT	`data Nat where`
	`    O :: Nat`
	`    S :: Nat -> Nat`
		`type Nat :: Type`

parameterized	`data List a :: Type where`
	`    Nil :: List a`
	`    Cons :: a -> List a -> List a`
		`type List :: Type -> Type`

annotated/indexed	`data Ev :: Nat -> Type where`
	`    EvO :: Ev 'O`
	`    EvSS :: Ev n -> Ev ('S ('S n))`
		`type Ev :: Nat -> Type`

- Now that we know what kinds are and what type operators are, we see that:
  - There is nothing special about their syntax!
  - They just have a different kind!

No more mysteric data types

- We were mumbling about the syntax of these two data type definitions (to encode relations):

(recursive) ADT	`data Nat = O | S Nat`	`type Nat :: Type`

GADT	`data Nat where`
	`    O :: Nat`
	`    S :: Nat -> Nat`
		`type Nat :: Type`

parameterized	`data List a :: Type where`
	`    Nil :: List a`
	`    Cons :: a -> List a -> List a`
		`type List :: Type -> Type`

annotated/indexed	`data Ev :: Nat -> Type where`
	`    EvO :: Ev 'O`
	`    EvSS :: Ev n -> Ev ('S ('S n))`
		`type Ev :: Nat -> Type`

- Now that we know what kinds are and what type operators are, we see that:
  - There is nothing special about their syntax!
  - They just have a different kind!
  - (See code!)

No more mysteric data types

- We were mumbling about the syntax of these two data type definitions (to encode relations):

(recursive) ADT	`data Nat = O | S Nat`	`type Nat :: Type`

GADT	`data Nat where`
	`    O :: Nat`
	`    S :: Nat -> Nat`
		`type Nat :: Type`

parameterized	`data List a :: Type where`
	`    Nil :: List a`
	`    Cons :: a -> List a -> List a`
		`type List :: Type -> Type`

annotated/indexed	`data Ev :: Nat -> Type where`
	`    EvO :: Ev 'O`
	`    EvSS :: Ev n -> Ev ('S ('S n))`
		`type Ev :: Nat -> Type`

- Now that we know what kinds are and what type operators are, we see that:
  - There is nothing special about their syntax!
  - They just have a different kind!
  - (See code!)
- (Don't worry, we will unlock the rest of the mysteries in a couple of slides.)

F_ω

- F_ω is a combination of System F and λ_ω.
- To directly read System F terms in System F_ω, we abbreviate $\forall X :: *.T$ as $\forall X.T$.

F_ω

Syntax

t	$::=$		terms:
	$\mid$	x	variable
	$\mid$	$\lambda x : T.t$	abstraction
	$\mid$	$t\ t$	application
	$\mid$	$\lambda X :: K.t$	type abstraction
	$\mid$	$t\ [T]$	type application
v	$::=$		values:
	$\mid$	$\lambda x : T.t$	abstraction value
	$\mid$	$\lambda X :: K.t$	type abstraction value

T	$::=$		types:
	$\mid$	X	type variable
	$\mid$	$T \to T$	type of functions
	$\mid$	$\forall X :: K.T$	universal type
	$\mid$	$\lambda X :: K.T$	operator abstraction
	$\mid$	$T\ T$	operator application
Γ	$::=$		contexts:
	$\mid$	$\emptyset$	empty context
	$\mid$	$\Gamma, x : T$	term variable binding
	$\mid$	$\Gamma, X :: K$	type variable binding
K	$::=$		kinds:
	$\mid$	$*$	kind of proper types
	$\mid$	$K \Rightarrow K$	kind of operators

F_ω

Evaluation

$$\boxed{t \longrightarrow t'}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \text{ E-APP1} \qquad \frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \text{ E-APP2} \qquad \frac{}{(\lambda x : T.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12}} \text{ E-APPABS}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]} \text{ E-TAPP} \qquad \frac{}{(\lambda X :: K_{11}.t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12}} \text{ E-TAPPTABS}$$

Typing

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

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ T-VAR} \qquad \frac{\Gamma \vdash T_1 :: * \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2} \text{ T-ABS} \qquad \frac{\Gamma, t_1 : T_{11} \to T_{12} \quad t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \text{ T-APP}$$

$$\frac{\Gamma \vdash t : S \quad S \equiv T \quad \Gamma \vdash T :: *}{\Gamma \vdash t : T} \text{ T-EQ}$$

F_ω
Typing

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

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \textsc{T-Var} \qquad \frac{\Gamma \vdash T_1 :: * \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2}\ \textsc{T-Abs} \qquad \frac{\Gamma, t_1 : T_{11} \to T_{12} \quad t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}\ \textsc{T-App}$$

$$\frac{\Gamma, X :: K_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda X :: K_1 . t_2 : \forall X :: K_1 . T_2}\ \textsc{T-TAbs} \qquad \frac{\Gamma, t_1 : \forall X :: K_{11} . T_{12} \quad \Gamma \vdash T_2 :: K_{11}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2] T_{12}}\ \textsc{T-TApp}$$

$$\frac{\Gamma \vdash t : S \quad S \equiv T \quad \Gamma \vdash T :: *}{\Gamma \vdash t : T}\ \textsc{T-Eq}$$

Kinding

$$\boxed{\Gamma \vdash T :: K}$$

$$\frac{X :: K \in \Gamma}{\Gamma \vdash X :: K} \text{ T-TVAR} \qquad \frac{\Gamma, X :: K_1 \vdash T_2 :: K_2}{\Gamma \vdash \lambda X :: K_1 . T_2 :: K_1 \Rightarrow K_2} \text{ K-ABS} \qquad \frac{\Gamma, t_1 : T_{11} \to T_{12} \quad t_2 : T_{11}}{\Gamma \vdash T_1 \; T_2 :: K_{12}} \text{ K-APP}$$

$$\frac{\Gamma \vdash T_1 :: * \quad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \to T_2 :: *} \text{ K-ARROW}$$

F_ω

Kinding

$\boxed{\Gamma \vdash T :: K}$

$$\frac{X :: K \in \Gamma}{\Gamma \vdash X :: K} \textsc{T-TVar}$$

$$\frac{\Gamma, X :: K_1 \vdash T_2 :: K_2}{\Gamma \vdash \lambda X :: K_1.T_2 :: K_1 \Rightarrow K_2} \textsc{K-Abs}$$

$$\frac{\Gamma, t_1 : T_{11} \to T_{12} \quad t_2 : T_{11}}{\Gamma \vdash T_1\ T_2 :: K_{12}} \textsc{K-App}$$

$$\frac{\Gamma \vdash T_1 :: * \quad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \to T_2 :: *} \textsc{K-Arrow}$$

Kinding

$$\boxed{\Gamma \vdash T :: K}$$

$$\frac{X :: K \in \Gamma}{\Gamma \vdash X :: K} \ \text{T-TVar}$$

$$\frac{\Gamma, X :: K_1 \vdash T_2 :: K_2}{\Gamma \vdash \lambda X :: K_1 . T_2 :: K_1 \Rightarrow K_2} \ \text{K-Abs}$$

$$\frac{\Gamma, t_1 : T_{11} \to T_{12} \quad t_2 : T_{11}}{\Gamma \vdash T_1\ T_2 :: K_{12}} \ \text{K-App}$$

$$\frac{\Gamma \vdash T_1 :: * \quad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \to T_2 :: *} \ \text{K-Arrow}$$

$$\frac{\Gamma, X :: K_1 \vdash T_2 :: *}{\Gamma \vdash \forall X :: K_1 . T_2 :: *} \ \text{K-All}$$

F_ω

Type Equivalence

$$\boxed{S \equiv T}$$

$$\frac{}{T \equiv T}\ \text{Q-Refl} \qquad \frac{T \equiv S}{S \equiv T}\ \text{Q-Symm} \qquad \frac{S \equiv U \quad U \equiv T}{S \equiv T}\ \text{Q-Trans} \qquad \frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 \to S_2 \equiv T_1 \to T_2}\ \text{Q-Arrow}$$

$$\frac{S_2 \equiv T_2}{\lambda X :: K_1.S_2 \equiv \lambda X :: K_1.T_2}\ \text{Q-Abs} \qquad \frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1\ S_2 \equiv T_1\ T_2}\ \text{Q-App}$$

$$\frac{}{(\lambda X :: K_{11}.T_2)\ T_{11} \equiv [X \mapsto T_{11}]T_2}\ \text{Q-AppAbs}$$

F_ω
Type Equivalence

$$\boxed{S \equiv T}$$

$$\frac{}{T \equiv T}\ \text{Q-Refl} \qquad \frac{T \equiv S}{S \equiv T}\ \text{Q-Symm} \qquad \frac{S \equiv U \quad U \equiv T}{S \equiv T}\ \text{Q-Trans} \qquad \frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 \to S_2 \equiv T_1 \to T_2}\ \text{Q-Arrow}$$

$$\frac{S_2 \equiv T_2}{\forall X :: K_1.S_2 \equiv \forall X :: K_1.T_2}\ \text{Q-All} \qquad \frac{S_2 \equiv T_2}{\lambda X :: K_1.S_2 \equiv \lambda X :: K_1.T_2}\ \text{Q-Abs} \qquad \frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1\ S_2 \equiv T_1\ T_2}\ \text{Q-App}$$

$$\frac{}{(\lambda X :: K_{11}.T_2)\ T_{11} \equiv [X \mapsto T_{11}]T_2}\ \text{Q-AppAbs}$$

Dependently-typed Programming in Haskell

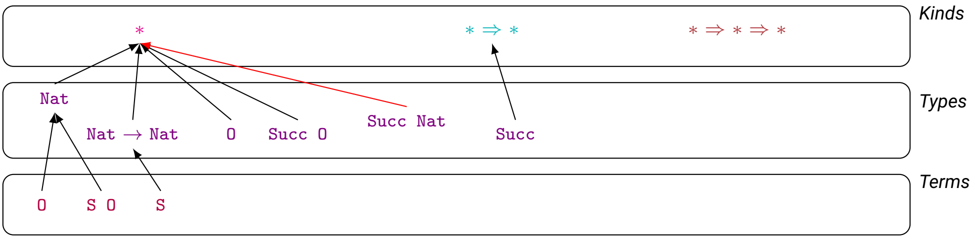
- We have worked so hard this semester to reach this point. Congratulations!
- Let's see the Curry-Howard Correspondence in dependently-typed Haskell in action and prove some theorems! (See code!)

Language Extensions in Haskell

- To perform type-level programming, we need to be able to
- create new types that are well-kinded and
- type-level functions that are total.

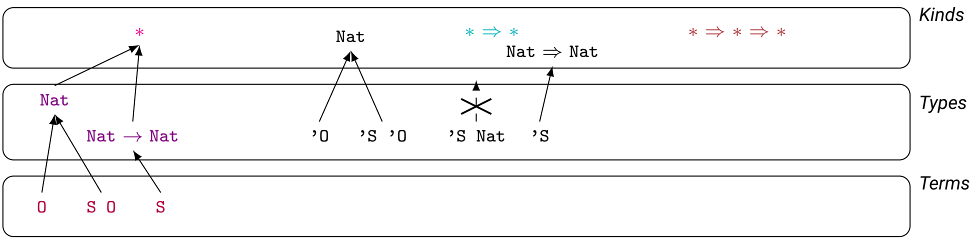
Promoted Data Types and Type Families in Haskell

- Data types abstract over the concrete data constructors.
- But for type-level programming, we actually need this information. How else would we express something like $1 + 0$, i.e., `S 0`?
- Let's create some unique (uninhabitated) types.



Promoted Data Types and Type Families in Haskell

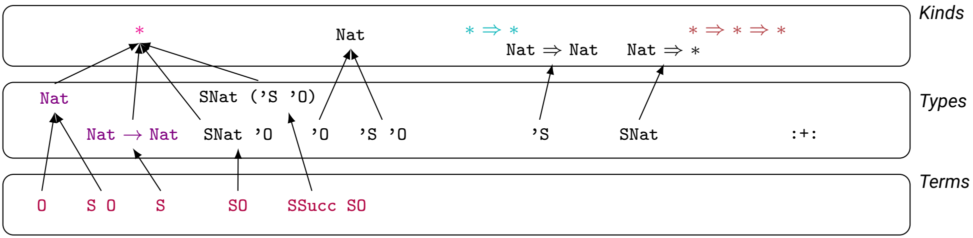
- The language extension `{# LANGUAGE DataKinds #}` ...
- promotes (data) types to the kind-level and
- constructors to the type-level.[1]



[1] Brent A Yorgey et al. "Giving Haskell a promotion". In: *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. 2012, pp. 53–66.

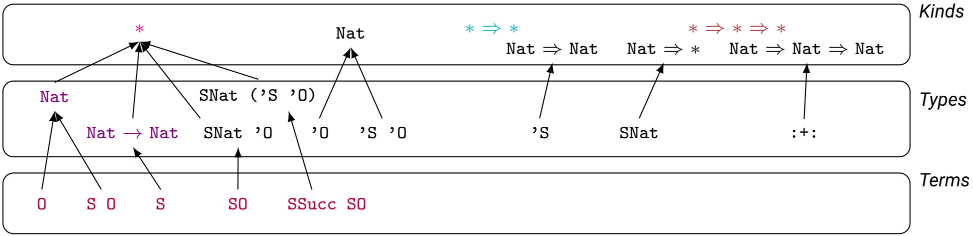
Promoted Data Types and Type Families in Haskell

- Remember: proofs are programs!
- Hence, we need to connect terms and types.
- We need a term-level representation for every type: `'O`, `'S 'O`, `'S ('S 'O)`, ...
- We do this by creating a *singleton* data type, i.e., a type that is index by the types of kind `Nat`.



Promoted Data Types and Type Families in Haskell

- Language extension: `{# LANGUAGE TypeFamilies #}`
- Promoting *closed* data types, i.e., ADTs and GADTs, enables
- closed type families.[1]
- (Where "closed" here means: not extensible.)



[1] Richard A Eisenberg et al. "Closed type families with overlapping equations". In: *ACM SIGPLAN Notices* (2014).

Further Reading on Dependently-Typed Programming in Haskell

Hasochism describes the benefits and pain points of dependent programming in Haskell.[1]

Singletons a library-based approach to create singleton types and promote term-level functions to the type-level.[2]

Stitch describes a very elegant implementation of a language and its compiler in Haskell that promotes all the type-checking of the implemented language into GHCs type system.[3]

DH Make sure to stay tuned for Dependently-typed Haskell.[4]

[1] Sam Lindley and Conor McBride. "Hasochism: The Pleasure and Pain of Dependently Typed Haskell Programming". In: *SIGPLAN Not.* (2013).

[2] Richard A Eisenberg and Stephanie Weirich. "Dependently typed programming with singletons". In: *ACM SIGPLAN Notices* (2012).

[3] Richard A Eisenberg. "Stitch: the sound type-indexed type checker (functional pearl)". In: *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*. 2020, pp. 39–53.

[4] Stephanie Weirich, Justin Hsu, and Richard A Eisenberg. "Towards dependently typed Haskell: System FC with kind equality". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP*.

What we have learned

- This lecture marks the end of our path towards understanding the type system of Haskell.
- We have learned that
  - kinds are types of the types and
  - most importantly that proofs are just programs (that produce a proof witness for their result type, i.e., their proposition)!
- Type-level functions provide the possibility to
  - express powerful propositions and
  - in Haskell translate them into adding equations for the HM type inference algorithm.