barkhauseninstitut.org

barkhausen institut

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

- Terms in STLC:
  - $\texttt{idNat} = \lambda x : \texttt{Nat}.\, x, \quad \texttt{idBool} = \lambda x : \texttt{Bool}.\, x, \quad \ldots$
- Let's
  - increase re-usability by
  - enabling *polymorphic abstactions*.

From Base Types to Type Variables

- So far, we define the notion of a *(uninterpreted) base type* without any specific functionality.
- Intuitively, base types are just placeholders for some type (that we do not care about).
- From now on, we treat base types as *type variables* that can be substituted and *instantiated*.

Type Variables formally

Type substitution is a mapping σ from type variables to types, e.g., $\sigma = [X \mapsto \texttt{Nat}, Y \mapsto \texttt{Bool}]$.

Type Variables formally

Type substitution is a mapping σ from type variables to types, e.g., $\sigma = [X \mapsto \texttt{Nat}, Y \mapsto \texttt{Bool}]$.

Applying substitution σ to type T to obtain an instance σT is defined as:

Type Variables formally

Type substitution is a mapping σ from type variables to types, e.g., $\sigma = [X \mapsto \texttt{Nat}, Y \mapsto \texttt{Bool}]$.

Applying substitution σ to type T to obtain an instance σT is defined as:

$$
\begin{array}{lcl}
\sigma(X) & = & \begin{cases} T & \text{if } (X \mapsto T) \in \sigma \\ X & \text{if } X \notin dom(\sigma) \end{cases} \\
\sigma(\texttt{Nat}) & = & \texttt{Nat} \\
\sigma(T_1 \to T_2) & = & \sigma T_1 \to \sigma T_1
\end{array}
$$

Type Variables formally

Type substitution is a mapping σ from type variables to types, e.g., $\sigma = [X \mapsto \texttt{Nat}, Y \mapsto \texttt{Bool}]$.

Applying substitution σ to type T to obtain an instance σT is defined as:

$$
\begin{array}{lcll}
\sigma(X) & = & \begin{cases} T & \text{if } (X \mapsto T) \in \sigma \\ X & \text{if } X \notin dom(\sigma) \end{cases} \\
\sigma(\texttt{Nat}) & = & \texttt{Nat} \\
\sigma(T_1 \to T_2) & = & \sigma T_1 \to \sigma T_1
\end{array}
$$

- When $\sigma = [X \mapsto U]$ then we also write $[X \mapsto U]T$.

Two perspectives on type variables

1. "Are *all* substitution instances of t well-typed?" ($\forall \sigma . \exists T . \sigma\Gamma \vdash \sigma t : T$)

Two perspectives on type variables

1. "Are *all* substitution instances of t well-typed?" ($\forall \sigma . \exists T . \sigma\Gamma \vdash \sigma t : T$)
    - Keeps the type variables abstract.

Two perspectives on type variables

1. "Are *all* substitution instances of t well-typed?" ($\forall \sigma . \exists T . \sigma \Gamma \vdash \sigma t : T$)
    - Keeps the type variables abstract.
    - Example: $\lambda f : X \to X .\ \lambda a : X .\ f\ (f\ a) : (X \to X) \to X \to X$

Two perspectives on type variables

1. "Are *all* substitution instances of t well-typed?" ($\forall \sigma . \exists T . \sigma \Gamma \vdash \sigma t : T$)
   - Keeps the type variables abstract.
   - Example: $\lambda f : X \to X.\ \lambda a : X.\ f\ (f\ a) : (X \to X) \to X \to X$
   - Replacing X with T, then $\lambda f : T \to T.\ \lambda a : T.\ f\ (f\ a)$ is well-typed.

Two perspectives on type variables

1. “Are *all* substitution instances of t well-typed?” ($\forall \sigma . \exists T . \sigma \Gamma \vdash \sigma t : T$)
    - Keeps the type variables abstract.
    - Example: $\lambda f : X \to X .\ \lambda a : X .\ f\ (f\ a) : (X \to X) \to X \to X$
    - Replacing X with T, then $\lambda f : T \to T .\ \lambda a : T .\ f\ (f\ a)$ is well-typed.
    - Terms can be used in many different contexts which leads to *parametric polymorphism*.

Two perspectives on type variables

1. “Are *all* substitution instances of t well-typed?” ($\forall \sigma . \exists T . \sigma \Gamma \vdash \sigma t : T$)
    - Keeps the type variables abstract.
    - Example: $\lambda f : X \to X.\ \lambda a : X.\ f\ (f\ a) : (X \to X) \to X \to X$
    - Replacing X with T, then $\lambda f : T \to T.\ \lambda a : T.\ f\ (f\ a)$ is well-typed.
    - Terms can be used in many different contexts which leads to *parametric polymorphism*.
2. “Is *some* substitution instance of t well-typed?” $\exists \sigma . \exists T . \sigma \Gamma \vdash \sigma t : T$

Two perspectives on type variables

1. "Are *all* substitution instances of t well-typed?" ($\forall \sigma . \exists T . \sigma\Gamma \vdash \sigma t : T$)
   - Keeps the type variables abstract.
   - Example: $\lambda f : X \to X.\ \lambda a : X.\ f\ (f\ a) : (X \to X) \to X \to X$
   - Replacing X with T, then $\lambda f : T \to T.\ \lambda a : T.\ f\ (f\ a)$ is well-typed.
   - Terms can be used in many different contexts which leads to *parametric polymorphism*.
2. "Is *some* substitution instance of t well-typed?" $\exists \sigma . \exists T . \sigma\Gamma \vdash \sigma t : T$
   - Can the term t be instantiated to a well typed term when choosing appropriate concrete types for its type variables?

Two perspectives on type variables

1. “Are *all* substitution instances of t well-typed?” ($\forall \sigma. \exists T. \sigma\Gamma \vdash \sigma t : T$)
    - Keeps the type variables abstract.
    - Example: $\lambda f : X \to X.\ \lambda a : X.\ f\ (f\ a) : (X \to X) \to X \to X$
    - Replacing X with T, then $\lambda f : T \to T.\ \lambda a : T.\ f\ (f\ a)$ is well-typed.
    - Terms can be used in many different contexts which leads to *parametric polymorphism*.
2. “Is *some* substitution instance of t well-typed?” $\exists \sigma. \exists T. \sigma\Gamma \vdash \sigma t : T$
    - Can the term t be instantiated to a well typed term when choosing appropriate concrete types for its type variables?
    - Example: $\lambda f : Y.\ \lambda a : X.\ f\ (f\ a)$ is not even typable.

Two perspectives on type variables

1. "Are *all* substitution instances of t well-typed?" ($\forall \sigma . \exists T . \sigma\Gamma \vdash \sigma t : T$)
   - Keeps the type variables abstract.
   - Example: $\lambda f : X \rightarrow X.\ \lambda a : X.\ f\ (f\ a) : (X \rightarrow X) \rightarrow X \rightarrow X$
   - Replacing X with T, then $\lambda f : T \rightarrow T.\ \lambda a : T.\ f\ (f\ a)$ is well-typed.
   - Terms can be used in many different contexts which leads to *parametric polymorphism*.
2. "Is *some* substitution instance of t well-typed?" $\exists \sigma . \exists T . \sigma\Gamma \vdash \sigma t : T$
   - Can the term t be instantiated to a well typed term when choosing appropriate concrete types for its type variables?
   - Example: $\lambda f : Y.\ \lambda a : X.\ f\ (f\ a)$ is not even typable.
   - But replacing Y with $\texttt{Nat} \rightarrow \texttt{Nat}$ and X with $\texttt{Nat}$ gives well-typed term $\lambda f : \texttt{Nat} \rightarrow \texttt{Nat}.\ \lambda a : \texttt{Nat}.\ f\ (f\ a)$.

Two perspectives on type variables

1. "Are *all* substitution instances of t well-typed?" ($\forall \sigma . \exists T . \sigma \Gamma \vdash \sigma t : T$)
   - Keeps the type variables abstract.
   - Example: $\lambda f : X \to X.\ \lambda a : X.\ f\ (f\ a) : (X \to X) \to X \to X$
   - Replacing X with T, then $\lambda f : T \to T.\ \lambda a : T.\ f\ (f\ a)$ is well-typed.
   - Terms can be used in many different contexts which leads to *parametric polymorphism*.
2. "Is *some* substitution instance of t well-typed?" $\exists \sigma . \exists T . \sigma \Gamma \vdash \sigma t : T$
   - Can the term t be instantiated to a well typed term when choosing appropriate concrete types for its type variables?
   - Example: $\lambda f : Y.\ \lambda a : X.\ f\ (f\ a)$ is not even typable.
   - But replacing Y with $\texttt{Nat} \to \texttt{Nat}$ and X with $\texttt{Nat}$ gives well-typed term $\lambda f : \texttt{Nat} \to \texttt{Nat}.\ \lambda a : \texttt{Nat}.\ f\ (f\ a)$.
   - Also replacing Y with $X \to X$ gives well-typed term $\lambda f : X \to X.\ \lambda a : X.\ f\ (f\ a)$.

Two perspectives on type variables

1. "Are *all* substitution instances of t well-typed?" ($\forall\sigma.\exists T.\sigma\Gamma \vdash \sigma t : T$)
   - Keeps the type variables abstract.
   - Example: $\lambda f : X \to X.\ \lambda a : X.\ f\ (f\ a) : (X \to X) \to X \to X$
   - Replacing X with T, then $\lambda f : T \to T.\ \lambda a : T.\ f\ (f\ a)$ is well-typed.
   - Terms can be used in many different contexts which leads to *parametric polymorphism*.
2. "Is *some* substitution instance of t well-typed?" $\exists\sigma.\exists T.\sigma\Gamma \vdash \sigma t : T$
   - Can the term t be instantiated to a well typed term when choosing appropriate concrete types for its type variables?
   - Example: $\lambda f : Y.\ \lambda a : X.\ f\ (f\ a)$ is not even typable.
   - But replacing Y with $\mathtt{Nat} \to \mathtt{Nat}$ and X with $\mathtt{Nat}$ gives well-typed term $\lambda f : \mathtt{Nat} \to \mathtt{Nat}.\ \lambda a : \mathtt{Nat}.\ f\ (f\ a)$.
   - Also replacing Y with $X \to X$ gives well-typed term $\lambda f : X \to X.\ \lambda a : X.\ f\ (f\ a)$.
   - Considered the most general instance.

Two perspectives on type variables

1. "Are *all* substitution instances of t well-typed?" ($\forall \sigma . \exists T . \sigma \Gamma \vdash \sigma t : T$)
    - Keeps the type variables abstract.
    - Example: $\lambda f : X \to X.\ \lambda a : X.\ f\ (f\ a) : (X \to X) \to X \to X$
    - Replacing X with T, then $\lambda f : T \to T.\ \lambda a : T.\ f\ (f\ a)$ is well-typed.
    - Terms can be used in many different contexts which leads to *parametric polymorphism*.
2. "Is *some* substitution instance of t well-typed?" $\exists \sigma . \exists T . \sigma \Gamma \vdash \sigma t : T$
    - Can the term t be instantiated to a well typed term when choosing appropriate concrete types for its type variables?
    - Example: $\lambda f : Y.\ \lambda a : X.\ f\ (f\ a)$ is not even typable.
    - But replacing Y with $\mathtt{Nat} \to \mathtt{Nat}$ and X with $\mathtt{Nat}$ gives well-typed term $\lambda f : \mathtt{Nat} \to \mathtt{Nat}.\ \lambda a : \mathtt{Nat}.\ f\ (f\ a)$.
    - Also replacing Y with $X \to X$ gives well-typed term $\lambda f : X \to X.\ \lambda a : X.\ f\ (f\ a)$.
    - Considered the most general instance.
    - Looking for valid instantiations leads to *type reconstruction/type inference*.

Forms of Polymorphism

Parametric Polymorphism generalizes over a specific type using variables and allows to instantiate them with concrete types. (The focus of this lecture.) No concrete type information is present in the abstraction.

Forms of Polymorphism

Parametric Polymorphism generalizes over a specific type using variables and allows to instantiate them with concrete types. (The focus of this lecture.) No concrete type information is present in the abstraction.

Example `id :: forall x. x -> x`

Forms of Polymorphism

Parametric Polymorphism generalizes over a specific type using variables and allows to instantiate them with concrete types. (The focus of this lecture.) No concrete type information is present in the abstraction.

Example `id :: forall x. x -> x`

Ad-hoc Polymorphism associates a polymorphic value with different behaviors (terms), e.g., *overloading* associates one function symbol with multiple implementations that are specialized for a concrete type.

Forms of Polymorphism

Parametric Polymorphism generalizes over a specific type using variables and allows to instantiate them with concrete types. (The focus of this lecture.) No concrete type information is present in the abstraction.

: **Example** `id :: forall x. x -> x`

Ad-hoc Polymorphism associates a polymorphic value with different behaviors (terms), e.g., *overloading* associates one function symbol with multiple implementations that are specialized for a concrete type.

: **Representatives** Haskell's type classes, Interface-based programming, Trait-based programming, `instanceof` in Java etc.

Forms of Polymorphism

Parametric Polymorphism generalizes over a specific type using variables and allows to instantiate them with concrete types. (The focus of this lecture.) No concrete type information is present in the abstraction.

  Example `id :: forall x. x -> x`

Ad-hoc Polymorphism associates a polymorphic value with different behaviors (terms), e.g., *overloading* associates one function symbol with multiple implementations that are specialized for a concrete type.

  Representatives Haskell's type classes, Interface-based programming, Trait-based programming, `instanceof` in Java etc.

Subtype Polymorphism associates a single term with several other types, that may refine the type or "forget" information about it.

Forms of Polymorphism

Parametric Polymorphism generalizes over a specific type using variables and allows to instantiate them with concrete types. (The focus of this lecture.) No concrete type information is present in the abstraction.

Example `id :: forall x. x -> x`

Ad-hoc Polymorphism associates a polymorphic value with different behaviors (terms), e.g., *overloading* associates one function symbol with multiple implementations that are specialized for a concrete type.

Representatives Haskell's type classes, Interface-based programming, Trait-based programming, `instanceof` in Java etc.

Subtype Polymorphism associates a single term with several other types, that may refine the type or "forget" information about it.

Representatives LiquidHaskell's subset types, Coq's sigma types, (Inheritance in object-oriented programming)

Hindley-Milner
History

1969 – J. Roger Hindley , a logician, discovers a method to derive a *principal type scheme* for a term in combinatory logic.

1978 – Robin Milner , a computer scientist, redicovers this method to infer the concrete type of a *polymorphic type* in a functional programming languages

- A language that implements HM is *implicitly-typed*, i.e., there are no type annotations in terms.

Hindley-Milner

History

1969 – J. Roger Hindley , a logician, discovers a method to derive a *principal type scheme* for a term in combinatory logic.

1978 – Robin Milner , a computer scientist, redicovers this method to infer the concrete type of a *polymorphic type* in a functional programming languages

- A language that implements HM is *implicitly-typed*, i.e., there are no type annotations in terms.
- The type checker *infers* the types.

Hindley-Milner

History

1969 – J. Roger Hindley , a logician, discovers a method to derive a *principal type scheme* for a term in combinatory logic.

1978 – Robin Milner , a computer scientist, redicovers this method to infer the concrete type of a *polymorphic type* in a functional programming languages

- A language that implements HM is *implicitly-typed*, i.e., there are no type annotations in terms.
- The type checker *infers* the types.
- Foundation of type systems for ML and Haskell.

Hindley-Milner

History

1969 – J. Roger Hindley , a logician, discovers a method to derive a *principal type scheme* for a term in combinatory logic.

1978 – Robin Milner , a computer scientist, redicovers this method to infer the concrete type of a *polymorphic type* in a functional programming languages

- A language that implements HM is *implicitly-typed*, i.e., there are no type annotations in terms.
- The type checker *infers* the types.
- Foundation of type systems for ML and Haskell.
- Most important property: type inference is decidable.

Hindley-Milner

History

1969 – J. Roger Hindley , a logician, discovers a method to derive a *principal type scheme* for a term in combinatory logic.

1978 – Robin Milner , a computer scientist, redicovers this method to infer the concrete type of a *polymorphic type* in a functional programming languages

- A language that implements HM is *implicitly-typed*, i.e., there are no type annotations in terms.
- The type checker *infers* the types.
- Foundation of type systems for ML and Haskell.
- Most important property: type inference is decidable.
- Disclaimer: we restrict the presentation here to universal quantification.

t	$::=$		terms:	T	$::=$		monotypes:
	$\mid$	x	variable		$\mid$	X	type variable
	$\mid$	$\lambda x.t$	abstraction		$\mid$	$T \to T$	type of functions
	$\mid$	$t\ t$	application	P	$::=$		polytypes:
	$\mid$	$\texttt{let}\ x = t\ \texttt{in}\ t$			$\mid$	T	monotype
v	$::=$		values:		$\mid$	$\forall X.P$	type scheme
	$\mid$	$\lambda x.t$	abstraction value	Γ	$::=$		contexts:
					$\mid$	$\varnothing$	empty context
					$\mid$	$\Gamma, x : P$	term variable binding

Hindley Milner
Examples

- $\texttt{id} = \lambda x.\, x$ has type $\texttt{id} : \forall X.\, X \to X$

[1] Of course `List` is something that we can not yet express.

Hindley Milner
Examples

- $\texttt{id} = \lambda x.\, x$ has type $\texttt{id} : \forall X.\, X \to X$
- $\texttt{double} = \lambda f.\, \lambda a.\, f\ (f\ a)$ has type $\texttt{double} : \forall X.\, (X \to X) \to X \to X$

[1] Of course `List` is something that we can not yet express.

Hindley Milner

Examples

- $\texttt{id} = \lambda x.\, x$ has type $\texttt{id} : \forall X.\, X \to X$
- $\texttt{double} = \lambda f.\, \lambda a.\, f\ (f\ a)$ has type $\texttt{double} : \forall X.\, (X \to X) \to X \to X$
- $\texttt{doubleZero} = \texttt{double}\ 0$

[1] Of course `List` is something that we can not yet express.

Hindley Milner
Examples

- $\texttt{id} = \lambda x.\, x$ has type $\texttt{id} : \forall X.\, X \to X$
- $\texttt{double} = \lambda f.\, \lambda a.\, f\ (f\ a)$ has type $\texttt{double} : \forall X.\, (X \to X) \to X \to X$
- $\texttt{doubleZero} = \texttt{double}\ 0$
- $\texttt{map} : \forall X.\, \forall Y.\, (X \to Y) \to \texttt{List}\ X \to \texttt{List}\ Y$[1]

[1] Of course `List` is something that we can not yet express.

- $\texttt{id} = \lambda x.\, x$ has type $\texttt{id} : \forall X.\, X \to X$
- $\texttt{double} = \lambda f.\, \lambda a.\, f\ (f\ a)$ has type $\texttt{double} : \forall X.\, (X \to X) \to X \to X$
- $\texttt{doubleZero} = \texttt{double}\ 0$
- $\texttt{map} : \forall X.\, \forall Y.\, (X \to Y) \to \texttt{List}\ X \to \texttt{List}\ Y$[1]
- Note, universal quantification can only appear at the top-level!

[1] Of course `List` is something that we can not yet express.

Hindley Milner

Examples

- $\texttt{id} = \lambda x.\, x$ has type $\texttt{id} : \forall X.\, X \to X$
- $\texttt{double} = \lambda f.\, \lambda a.\, f\ (f\ a)$ has type $\texttt{double} : \forall X.\, (X \to X) \to X \to X$
- $\texttt{doubleZero} = \texttt{double}\ 0$
- $\texttt{map} : \forall X.\, \forall Y.\, (X \to Y) \to \texttt{List}\ X \to \texttt{List}\ Y$[1]
- Note, universal quantification can only appear at the top-level!
- $\texttt{map}' : \forall X.\, \forall Y.\, (\forall Z.\, Z \to Y) \to \texttt{List}\ X \to \texttt{List}\ Y$ is not supported by the grammar. (We will support this when talking about higher-order types.)

[1] Of course `List` is something that we can not yet express.

Hindley-Milner

Type system

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

$$\frac{x : P \in \Gamma \quad P \sqsubseteq T}{\Gamma \vdash x : T}\ \textsc{T-Var}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x.\, t_2 : T_1 \to T_2}\ \textsc{T-Abs}$$

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

$$\frac{\Gamma, x : \forall_{\Gamma} T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : T_2}\ \textsc{T-Let}$$

where

Hindley-Milner
Type system

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

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

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

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

$$\frac{\Gamma, x : \forall_\Gamma T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : T_2} \text{ T-LET}$$

where

- $P_1 \sqsubseteq P_2$ states that P_1 is more general than P_2 or P_2 specializes P_1

Typing

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

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

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

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

$$\frac{\Gamma, x : \forall_\Gamma T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : T_2} \text{ T-LET}$$

where

- $P_1 \sqsubseteq P_2$ states that P_1 is more general than P_2 or P_2 specializes P_1

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

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

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

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

$$\frac{\Gamma, x : \forall_\Gamma T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\; x = t_1 \;\texttt{in}\; t_2 : T_2} \text{ T-LET}$$

where

- $P_1 \sqsubseteq P_2$ states that P_1 is more general than P_2 or P_2 specializes P_1

$$\forall X.X \to X \sqsubseteq \forall Y.(Y \to Y) \to (Y \to Y)$$

Hindley-Milner

Type system

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

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

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

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

$$\frac{\Gamma, x : \forall_{\Gamma} T_1 \vdash t_2 : T_2}{\Gamma \vdash \mathtt{let}\ x = t_1\ \mathtt{in}\ t_2 : T_2} \text{ T-LET}$$

where

- $P_1 \sqsubseteq P_2$ states that P_1 is more general than P_2 or P_2 specializes P_1

$$\begin{aligned} \forall X.X \rightarrow X &\sqsubseteq \forall Y.(Y \rightarrow Y) \rightarrow (Y \rightarrow Y) \\ &\sqsubseteq \mathtt{Bool} \rightarrow \mathtt{Bool} \end{aligned}$$

Hindley-Milner

Type system

Typing

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

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

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

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

$$\frac{\Gamma, x : \forall_\Gamma T_1 \vdash t_2 : T_2}{\Gamma \vdash \mathtt{let}\ x = t_1\ \mathtt{in}\ t_2 : T_2} \text{ T-LET}$$

where

- $P_1 \sqsubseteq P_2$ states that P_1 is more general than P_2 or P_2 specializes P_1
$$\begin{aligned} \forall X.X \to X &\sqsubseteq \forall Y.(Y \to Y) \to (Y \to Y) \\ &\sqsubseteq \mathtt{Bool} \to \mathtt{Bool} \end{aligned}$$
- $\forall_\Gamma T = \forall X_1 . \ldots . X_n . T$ with $FV(T) \setminus FV(\Gamma) = X_1, \ldots, X_n$ is called the *generalization of* T.

Hindley-Milner

Type system

Typing

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

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

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

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

$$\frac{\Gamma, x : \forall_\Gamma T_1 \vdash t_2 : T_2}{\Gamma \vdash \mathtt{let}\ x = t_1\ \mathtt{in}\ t_2 : T_2} \text{ T-LET}$$

where

- $P_1 \sqsubseteq P_2$ states that P_1 is more general than P_2 or P_2 specializes P_1

$$\begin{aligned} \forall X.X \to X &\sqsubseteq \forall Y.(Y \to Y) \to (Y \to Y) \\ &\sqsubseteq \mathtt{Bool} \to \mathtt{Bool} \end{aligned}$$

- $\forall_\Gamma T = \forall X_1. \ldots X_n.T$ with $FV(T) \setminus FV(\Gamma) = X_1, \ldots, X_n$ is called the *generalization of* T.

Hindley-Milner
Type system

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

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

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

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

$$\frac{\Gamma, x : \forall_\Gamma T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : T_2} \text{ T-LET}$$

where

- $P_1 \sqsubseteq P_2$ states that P_1 is more general than P_2 or P_2 specializes P_1

$$\begin{aligned} \forall X.X \to X &\sqsubseteq \forall Y.(Y \to Y) \to (Y \to Y) \\ &\sqsubseteq \texttt{Bool} \to \texttt{Bool} \end{aligned}$$

- $\forall_\Gamma T = \forall X_1. \ldots .X_n.T$ with $FV(T) \setminus FV(\Gamma) = X_1, \ldots, X_n$ is called the *generalization of* T.

$$FV(\forall X_1. \ldots .X_n.T) = FV(T) \setminus \{X_1, \ldots, X_n\}$$

Hindley-Milner
Type system

Typing

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

$$\frac{x : P \in \Gamma \quad P \sqsubseteq T}{\Gamma \vdash x : T}\ \textsc{T-Var}$$

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

$$\frac{\Gamma, x : \forall_\Gamma T_1 \vdash t_2 : T_2}{\Gamma \vdash \mathtt{let}\ x = t_1\ \mathtt{in}\ t_2 : T_2}\ \textsc{T-Let}$$

where

- $P_1 \sqsubseteq P_2$ states that P_1 is more general than P_2 or P_2 specializes P_1
$$\begin{aligned} \forall X.X \to X &\sqsubseteq \forall Y.(Y \to Y) \to (Y \to Y) \\ &\sqsubseteq \mathtt{Bool} \to \mathtt{Bool} \end{aligned}$$
- $\forall_\Gamma T = \forall X_1 .\ldots.X_n.T$ with $FV(T) \setminus FV(\Gamma) = X_1, \ldots, X_n$ is called the *generalization of* T.
$$FV(\forall X_1 .\ldots.X_n.T) = FV(T) \setminus \{X_1, \ldots, X_n\}$$
$$FV(\Gamma) = \bigcup_{i=1}^{n} FV(P_i) \text{ for a context } \Gamma = x : P_1, \ldots, x : P_n$$

Algorithm W

- The type inference algorithm for HM type systems is called *Algorithm W*.

Algorithm W

- The type inference algorithm for HM type systems is called *Algorithm W*.
- If Algorithm W can derive a type for a term then the term is guaranteed to be well-typed.

Algorithm W

- The type inference algorithm for HM type systems is called *Algorithm W*.
- If Algorithm W can derive a type for a term then the term is guaranteed to be well-typed.
- The algorithm records *type constraints* instead of directly checking them:

Algorithm W

- The type inference algorithm for HM type systems is called *Algorithm W*.
- If Algorithm W can derive a type for a term then the term is guaranteed to be well-typed.
- The algorithm records *type constraints* instead of directly checking them:
  Type checking: On $t_1\ t_2$ with $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash t_2 : T_2$,

Algorithm W

- The type inference algorithm for HM type systems is called *Algorithm W*.
- If Algorithm W can derive a type for a term then the term is guaranteed to be well-typed.
- The algorithm records *type constraints* instead of directly checking them:

  Type checking: On $t_1\ t_2$ with $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash t_2 : T_2$,

  1. Check immediately whether $T_1 = T_2 \rightarrow T_{12}$.

Algorithm W

- The type inference algorithm for HM type systems is called *Algorithm W*.
- If Algorithm W can derive a type for a term then the term is guaranteed to be well-typed.
- The algorithm records *type constraints* instead of directly checking them:

  Type checking: On $t_1\ t_2$ with $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash t_2 : T_2$,

  1. Check immediately whether $T_1 = T_2 \rightarrow T_{12}$.
  2. Return T_{12}.

Algorithm W

- The type inference algorithm for HM type systems is called *Algorithm W*.
- If Algorithm W can derive a type for a term then the term is guaranteed to be well-typed.
- The algorithm records *type constraints* instead of directly checking them:

  Type checking: On $t_1\ t_2$ with $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash t_2 : T_2$,

  1. Check immediately whether $T_1 = T_2 \rightarrow T_{12}$.
  2. Return T_{12}.

  Constraint-based typing: On $t_1\ t_2$ with $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash t_2 : T_2$

Algorithm W

- The type inference algorithm for HM type systems is called *Algorithm W*.
- If Algorithm W can derive a type for a term then the term is guaranteed to be well-typed.
- The algorithm records *type constraints* instead of directly checking them:

  Type checking: On $t_1\ t_2$ with $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash t_2 : T_2$,

  1. Check immediately whether $T_1 = T_2 \to T_{12}$.
  2. Return T_{12}.

  Constraint-based typing: On $t_1\ t_2$ with $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash t_2 : T_2$

  1. Choose a frest type variable X.

Algorithm W

- The type inference algorithm for HM type systems is called *Algorithm W*.
- If Algorithm W can derive a type for a term then the term is guaranteed to be well-typed.
- The algorithm records *type constraints* instead of directly checking them:

Type checking: On $t_1\ t_2$ with $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash t_2 : T_2$,

1. Check immediately whether $T_1 = T_2 \to T_{12}$.
2. Return T_{12}.

Constraint-based typing: On $t_1\ t_2$ with $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash t_2 : T_2$

1. Choose a frest type variable X.
2. Record $T_1 = T_2 \to X$ in the set of constraints.

Algorithm W

- The type inference algorithm for HM type systems is called *Algorithm W*.
- If Algorithm W can derive a type for a term then the term is guaranteed to be well-typed.
- The algorithm records *type constraints* instead of directly checking them:

Type checking: On $t_1\ t_2$ with $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash t_2 : T_2$,

1. Check immediately whether $T_1 = T_2 \to T_{12}$.
2. Return T_{12}.

Constraint-based typing: On $t_1\ t_2$ with $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash t_2 : T_2$

1. Choose a frest type variable X.
2. Record $T_1 = T_2 \to X$ in the set of constraints.
3. Return X

Algorithm W

- The type inference algorithm for HM type systems is called *Algorithm W*.
- If Algorithm W can derive a type for a term then the term is guaranteed to be well-typed.
- The algorithm records *type constraints* instead of directly checking them:

  Type checking: On $t_1\ t_2$ with $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash t_2 : T_2$,
  1. Check immediately whether $T_1 = T_2 \to T_{12}$.
  2. Return T_{12}.

  Constraint-based typing: On $t_1\ t_2$ with $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash t_2 : T_2$
  1. Choose a frest type variable X.
  2. Record $T_1 = T_2 \to X$ in the set of constraints.
  3. Return X
- On the recorded constraint set C, Algorithm W tries to find a substitution σ such that σ *unifies* every equation in C, i.e., $\forall (S = T) \in C.\ \sigma S = \sigma T$.

Algorithm W

- The type inference algorithm for HM type systems is called *Algorithm W*.
- If Algorithm W can derive a type for a term then the term is guaranteed to be well-typed.
- The algorithm records *type constraints* instead of directly checking them:

  Type checking: On $t_1\ t_2$ with $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash t_2 : T_2$,

  1. Check immediately whether $T_1 = T_2 \to T_{12}$.
  2. Return T_{12}.

  Constraint-based typing: On $t_1\ t_2$ with $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash t_2 : T_2$

  1. Choose a frest type variable X.
  2. Record $T_1 = T_2 \to X$ in the set of constraints.
  3. Return X

- On the recorded constraint set C, Algorithm W tries to find a substitution σ such that σ *unifies* every equation in C, i.e., $\forall (S = T) \in C.\ \sigma S = \sigma T$.
- We leave Algorithm W for the interested student to explore.

Algorithm W

- The type inference algorithm for HM type systems is called *Algorithm W*.
- If Algorithm W can derive a type for a term then the term is guaranteed to be well-typed.
- The algorithm records *type constraints* instead of directly checking them:

  Type checking: On $t_1\ t_2$ with $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash t_2 : T_2$,

  1. Check immediately whether $T_1 = T_2 \to T_{12}$.
  2. Return T_{12}.

  Constraint-based typing: On $t_1\ t_2$ with $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash t_2 : T_2$

  1. Choose a frest type variable X.
  2. Record $T_1 = T_2 \to X$ in the set of constraints.
  3. Return X

- On the recorded constraint set C, Algorithm W tries to find a substitution σ such that σ *unifies* every equation in C, i.e., $\forall (S = T) \in C.\ \sigma S = \sigma T$.
- We leave Algorithm W for the interested student to explore.
- We will see HM type inference in Haskell in action at the end of the lecture. (See code.)

Type schemes with explicit types

- Let's take the concept of a type scheme and
- construct a type system with explicit types (again).
- (For conciseness, we drop the `let` form.)

Universal quantification
Syntax

Syntax:

$$
\begin{array}{llll}
t & ::= & & \text{terms:} \\
 & | & x & \text{variable} \\
 & | & \lambda x : T.t & \text{abstraction} \\
 & | & t\ t & \text{application} \\
 & | & \lambda X.t & \text{type abstraction} \\
 & | & t\ [T] & \text{type application} \\
 \\
v & ::= & & \text{values:} \\
 & | & \lambda x : T.t & \text{abstraction value} \\
 & | & \lambda X.t & \text{type abstraction value}
\end{array}
\qquad
\begin{array}{llll}
T & ::= & & \text{monotypes:} \\
 & | & X & \text{type variable} \\
 & | & T \to T & \text{type of functions} \\
P & ::= & & \text{polytypes:} \\
 & | & T & \text{monotype} \\
 & | & \forall X.P & \text{type scheme} \\
 \\
\Gamma & ::= & & \text{contexts:} \\
 & | & \varnothing & \text{empty context} \\
 & | & \Gamma, x : T & \text{term variable binding} \\
 & | & \Gamma, X & \text{type variable binding}
\end{array}
$$

Universal quantification

Examples

- $\texttt{id} = \lambda X.\, \lambda x : X.\, x$ has type $\texttt{id} : \forall X.\, X \to X$

Universal quantification
Examples

- $\texttt{id} = \lambda X.\, \lambda x : X.\, x$ has type $\texttt{id} : \forall X.\, X \to X$
- $\texttt{double} = \lambda X.\, \lambda f : X \to X.\, \lambda a : X.\, f\ (f\ a)$ has type $\texttt{double} : \forall X.\, (X \to X) \to X \to X$

Universal quantification
Examples

- $\texttt{id} = \lambda X.\, \lambda x : X.\, x$ has type $\texttt{id} : \forall X.\, X \to X$
- $\texttt{double} = \lambda X.\, \lambda f : X \to X.\, \lambda a : X.\, f\ (f\ a)$ has type $\texttt{double} : \forall X.\, (X \to X) \to X \to X$
- $\texttt{doubleNat} = \texttt{double}\ [\texttt{Nat}]$

Universal quantification

Examples

- $\texttt{id} = \lambda X.\, \lambda x : X.\, x$ has type $\texttt{id} : \forall X.\, X \to X$
- $\texttt{double} = \lambda X.\, \lambda f : X \to X.\, \lambda a : X.\, f\ (f\ a)$ has type $\texttt{double} : \forall X.\, (X \to X) \to X \to X$
- $\texttt{doubleNat} = \texttt{double}\ [\texttt{Nat}]$
- $\texttt{doubleBool} = \texttt{double}\ [\texttt{Bool}]$

Universal quantification

Semantics and Typing

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

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

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

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

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

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

Universal quantification

Semantics and Typing

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

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

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

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

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

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

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

$$\frac{x : P \in \Gamma \quad P \sqsubseteq T}{\Gamma \vdash x : T} \ \text{T-Var}$$

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

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

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

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

Existential Types

Syntax

New Syntatic Forms:

t	$::=$	$\ldots$	terms:
	$\mid$	$\{*T, t\}\ \texttt{as}\ T$	packing
	$\mid$	$\texttt{let}\ \{X, x\} = t\ \texttt{in}\ t$	unpacking
v	$::=$	$\ldots$	values:
	$\mid$	$\{*T, v\}\ \texttt{as}\ T$	packaged value
T	$::=$	$\ldots$	types:
	$\mid$	$\{\exists X, T\}$	existential type

Existential Types

Intuition

- (Operational) intuition:

Existential Types

Intuition

- (Operational) intuition:

  Universal quantifiers An element of $\forall X.T$ is a function that maps a type S to a specialized term $[X \mapsto S]T$.

Existential Types

Intuition

- (Operational) intuition:

  Universal quantifiers An element of $\forall X.T$ is a function that maps a type S to a specialized term $[X \mapsto S]T$.

  Existential quantifiers An element of $\{\exists X, T\}$ is a *pair*, written $\{*S, t\}$, of type S and a term t of type $[X \mapsto S]T$.

Existential Types
Intuition

- (Operational) intuition:

  Universal quantifiers An element of $\forall X.T$ is a function that maps a type S to a specialized term $[X \mapsto S]T$.

  Existential quantifiers An element of $\{\exists X, T\}$ is a *pair*, written $\{*S, t\}$, of type S and a term t of type $[X \mapsto S]T$.
- Here, we use a tuple representation rather than the most standard notation $\exists X.T$.

Existential Types
Intuition

- (Operational) intuition:

  Universal quantifiers An element of $\forall X.T$ is a function that maps a type S to a specialized term $[X \mapsto S]T$.

  Existential quantifiers An element of $\{\exists X, T\}$ is a *pair*, written $\{*S, t\}$, of type S and a term t of type $[X \mapsto S]T$.

- Here, we use a tuple representation rather than the most standard notation $\exists X.T$.
- Concrete intuition: An existential value $\{*S, t\}$ of type $\{\exists X, T\}$ is a package or module with

- (Operational) intuition:

  Universal quantifiers An element of $\forall X.T$ is a function that maps a type S to a specialized term $[X \mapsto S]T$.

  Existential quantifiers An element of $\{\exists X, T\}$ is a *pair*, written $\{*S, t\}$, of type S and a term t of type $[X \mapsto S]T$.

- Here, we use a tuple representation rather than the most standard notation $\exists X.T$.
- Concrete intuition: An existential value $\{*S, t\}$ of type $\{\exists X, T\}$ is a package or module with
  - a *hidden* type component, the *witness type of the package* and

Existential Types

Intuition

- (Operational) intuition:

  Universal quantifiers An element of $\forall X.T$ is a function that maps a type S to a specialized term $[X \mapsto S]T$.

  Existential quantifiers An element of $\{\exists X, T\}$ is a *pair*, written $\{*S, t\}$, of type S and a term t of type $[X \mapsto S]T$.

- Here, we use a tuple representation rather than the most standard notation $\exists X.T$.
- Concrete intuition: An existential value $\{*S, t\}$ of type $\{\exists X, T\}$ is a package or module with
  - a *hidden* type component, the *witness type of the package* and
  - a term component.

Existential Types
Intuition

- (Operational) intuition:

  Universal quantifiers An element of $\forall X.T$ is a function that maps a type S to a specialized term $[X \mapsto S]T$.

  Existential quantifiers An element of $\{\exists X, T\}$ is a *pair*, written $\{*S, t\}$, of type S and a term t of type $[X \mapsto S]T$.
- Here, we use a tuple representation rather than the most standard notation $\exists X.T$.
- Concrete intuition: An existential value $\{*S, t\}$ of type $\{\exists X, T\}$ is a package or module with
  - a *hidden* type component, the *witness type of the package* and
  - a term component.
- Existentials have applications in module system and abstract data types.

Existential Types
Examples

- $p = \{*\texttt{Nat}, \{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}\}$ with type $\{\exists X, \{a : \texttt{Nat}, f : X \to \texttt{X}\}\}$

Existential Types

Examples

- $p = \{*\texttt{Nat}, \{a = 5, f = \lambda x : \texttt{Nat}.\ \texttt{succ}(x)\}\}$ with type $\{\exists X, \{a : \texttt{Nat}, f : X \to \texttt{X}\}\}$
- ($\{a = 5, f = \lambda x : \texttt{Nat}.\ \texttt{succ}(x)\}$ is a record, i.e., an extension of a tuple with named elements (/fields) and according accessors (eliminators).)

Existential Types
Examples

- $p = \{*\texttt{Nat}, \{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}\}$ with type $\{\exists X, \{a : \texttt{Nat}, f : X \to \texttt{X}\}\}$
- $(\{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}$ is a record, i.e., an extension of a tuple with named elements (/fields) and according accessors (eliminators).)
- But p also has type $\{\exists X, \{a : X, f : X \to \texttt{Nat}\}\}$

Existential Types
Examples

- $p = \{*\texttt{Nat}, \{a = 5, f = \lambda x : \texttt{Nat}.\ \texttt{succ}(x)\}\}$ with type $\{\exists X, \{a : \texttt{Nat}, f : X \to \texttt{X}\}\}$
- ($\{a = 5, f = \lambda x : \texttt{Nat}.\ \texttt{succ}(x)\}$ is a record, i.e., an extension of a tuple with named elements (/fields) and according accessors (eliminators).)
- But p also has type $\{\exists X, \{a : X, f : X \to \texttt{Nat}\}\}$
- $\Rightarrow$ Type reconstruction is not possible. The programmer has to provide the according type (via ascription):

Existential Types
Examples

- $p = \{*\texttt{Nat}, \{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}\}$ with type $\{\exists X, \{a : \texttt{Nat}, f : X \to \texttt{X}\}\}$
- ($\{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}$ is a record, i.e., an extension of a tuple with named elements (/fields) and according accessors (eliminators).)
- But p also has type $\{\exists X, \{a : X, f : X \to \texttt{Nat}\}\}$
- $\Rightarrow$ Type reconstruction is not possible. The programmer has to provide the according type (via ascription):
- $p = \{*\texttt{Nat}, \{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}\}$ as $\{\exists X, \{a : X, f : X \to X\}\}$

Existential Types
Examples

- $p = \{*\texttt{Nat}, \{a = 5, f = \lambda x : \texttt{Nat}.\ \texttt{succ}(x)\}\}$ with type $\{\exists X, \{a : \texttt{Nat}, f : X \rightarrow \texttt{X}\}\}$
- ($\{a = 5, f = \lambda x : \texttt{Nat}.\ \texttt{succ}(x)\}$ is a record, i.e., an extension of a tuple with named elements (/fields) and according accessors (eliminators).)
- But p also has type $\{\exists X, \{a : X, f : X \rightarrow \texttt{Nat}\}\}$
- $\Rightarrow$ Type reconstruction is not possible. The programmer has to provide the according type (via ascription):
- $p = \{*\texttt{Nat}, \{a = 5, f = \lambda x : \texttt{Nat}.\ \texttt{succ}(x)\}\}$ `as` $\{\exists X, \{a : X, f : X \rightarrow X\}\}$
- $p : \{\exists X, \{a : X, f : X \rightarrow X\}\}$

Existential Types

Examples

- $p = \{*\texttt{Nat}, \{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}\}$ with type $\{\exists X, \{a : \texttt{Nat}, f : X \to \texttt{X}\}\}$
- ($\{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}$ is a record, i.e., an extension of a tuple with named elements (/fields) and according accessors (eliminators).)
- But p also has type $\{\exists X, \{a : X, f : X \to \texttt{Nat}\}\}$
- $\Rightarrow$ Type reconstruction is not possible. The programmer has to provide the according type (via ascription):
- $p = \{*\texttt{Nat}, \{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}\}$ as $\{\exists X, \{a : X, f : X \to X\}\}$
- $p : \{\exists X, \{a : X, f : X \to X\}\}$
- $p' = \{*\texttt{Nat}, \{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}\}$ as $\{\exists X, \{a : X, f : X \to \texttt{Nat}\}\}$

Existential Types
Examples

- $p = \{*\texttt{Nat}, \{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}\}$ with type $\{\exists X, \{a : \texttt{Nat}, f : X \to \texttt{X}\}\}$
- $(\{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}$ is a record, i.e., an extension of a tuple with named elements (/fields) and according accessors (eliminators).)
- But p also has type $\{\exists X, \{a : X, f : X \to \texttt{Nat}\}\}$
- $\Rightarrow$ Type reconstruction is not possible. The programmer has to provide the according type (via ascription):
- $p = \{*\texttt{Nat}, \{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}\}$ as $\{\exists X, \{a : X, f : X \to X\}\}$
- $p : \{\exists X, \{a : X, f : X \to X\}\}$
- $p' = \{*\texttt{Nat}, \{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}\}$ as $\{\exists X, \{a : X, f : X \to \texttt{Nat}\}\}$
- $p' : \{\exists X, \{a : X, f : X \to \texttt{Nat}\}\}$

Existential Types
Examples

- $p = \{*\texttt{Nat}, \{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}\}$ with type $\{\exists X, \{a : \texttt{Nat}, f : X \to \texttt{X}\}\}$
- ($\{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}$ is a record, i.e., an extension of a tuple with named elements (/fields) and according accessors (eliminators).)
- But p also has type $\{\exists X, \{a : X, f : X \to \texttt{Nat}\}\}$
- $\Rightarrow$ Type reconstruction is not possible. The programmer has to provide the according type (via ascription):
- $p = \{*\texttt{Nat}, \{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}\}$ `as` $\{\exists X, \{a : X, f : X \to X\}\}$
- $p : \{\exists X, \{a : X, f : X \to X\}\}$
- $p' = \{*\texttt{Nat}, \{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}\}$ `as` $\{\exists X, \{a : X, f : X \to \texttt{Nat}\}\}$
- $p' : \{\exists X, \{a : X, f : X \to \texttt{Nat}\}\}$
- Note that *different* packages may have the *same* type:

Existential Types
Examples

- $p = \{*\texttt{Nat}, \{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}\}$ with type $\{\exists X, \{a : \texttt{Nat}, f : X \to \texttt{X}\}\}$
- $(\{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}$ is a record, i.e., an extension of a tuple with named elements (/fields) and according accessors (eliminators).)
- But p also has type $\{\exists X, \{a : X, f : X \to \texttt{Nat}\}\}$
- $\Rightarrow$ Type reconstruction is not possible. The programmer has to provide the according type (via ascription):
- $p = \{*\texttt{Nat}, \{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}\}$ as $\{\exists X, \{a : X, f : X \to X\}\}$
- $p : \{\exists X, \{a : X, f : X \to X\}\}$
- $p' = \{*\texttt{Nat}, \{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}\}$ as $\{\exists X, \{a : X, f : X \to \texttt{Nat}\}\}$
- $p' : \{\exists X, \{a : X, f : X \to \texttt{Nat}\}\}$
- Note that *different* packages may have the *same* type:
- $\{*\texttt{Nat}, 0\}$ as $\{\exists X, X\}$

Existential Types
Examples

- $p = \{*\texttt{Nat}, \{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}\}$ with type $\{\exists X, \{a : \texttt{Nat}, f : X \to \texttt{X}\}\}$
- $(\{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}$ is a record, i.e., an extension of a tuple with named elements (/fields) and according accessors (eliminators).)
- But p also has type $\{\exists X, \{a : X, f : X \to \texttt{Nat}\}\}$
- $\Rightarrow$ Type reconstruction is not possible. The programmer has to provide the according type (via ascription):
- $p = \{*\texttt{Nat}, \{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}\}$ `as` $\{\exists X, \{a : X, f : X \to X\}\}$
- $p : \{\exists X, \{a : X, f : X \to X\}\}$
- $p' = \{*\texttt{Nat}, \{a = 5, f = \lambda x : \texttt{Nat}.\, \texttt{succ}(x)\}\}$ `as` $\{\exists X, \{a : X, f : X \to \texttt{Nat}\}\}$
- $p' : \{\exists X, \{a : X, f : X \to \texttt{Nat}\}\}$
- Note that *different* packages may have the *same* type:
- $\{*\texttt{Nat}, 0\}$ `as` $\{\exists X, X\}$
- $\{*\texttt{Bool}, \texttt{true}\}$ `as` $\{\exists X, X\}$

Existential Types
Semantics and Typing

- *New Evaluation Rules*

- *New Evaluation Rules*

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

$$\frac{}{\texttt{let}\ \{X, x\} = (\{T_{11}, v_{12}\}\ \texttt{as}\ T_1)\ \texttt{in}\ t_2 \longrightarrow [X \mapsto T_{11}][x \mapsto v_{12}]t_2} \quad \text{E-UNPACKPACK}$$

$$\frac{t_{12} \longrightarrow t'_{12}}{\{*T_{11}, t_{12}\}\ \texttt{as}\ T_1 \longrightarrow \{*T_{11}, t'_{12}\}\ \texttt{as}\ T_1} \quad \text{E-PACK}$$

$$\frac{t_1 \longrightarrow t'_1}{\texttt{let}\ \{X, x\} = t_1\ \texttt{in}\ t_2 \longrightarrow \texttt{let}\ \{X, x\} = t'_1\ \texttt{in}\ t_2} \quad \text{E-UNPACK}$$

Existential Types
Semantics and Typing

- *<u>New</u> Evaluation Rules*

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

$$\frac{}{\texttt{let}\ \{X, x\} = (\{T_{11}, v_{12}\}\ \texttt{as}\ T_1)\ \texttt{in}\ t_2 \longrightarrow [X \mapsto T_{11}][x \mapsto v_{12}]t_2}\ \textsc{E-UnpackPack}$$

$$\frac{t_{12} \longrightarrow t'_{12}}{\{*T_{11}, t_{12}\}\ \texttt{as}\ T_1 \longrightarrow \{*T_{11}, t'_{12}\}\ \texttt{as}\ T_1}\ \textsc{E-Pack}$$

$$\frac{t_1 \longrightarrow t'_1}{\texttt{let}\ \{X, x\} = t_1\ \texttt{in}\ t_2 \longrightarrow \texttt{let}\ \{X, x\} = t'_1\ \texttt{in}\ t_2}\ \textsc{E-Unpack}$$

- *<u>New</u> Typing Rules*

Existential Types

Semantics and Typing

- *New Evaluation Rules*

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

$$\frac{}{\texttt{let}\ \{X, x\} = (\{T_{11}, v_{12}\}\ \texttt{as}\ T_1)\ \texttt{in}\ t_2 \longrightarrow [X \mapsto T_{11}][x \mapsto v_{12}]t_2}\ \text{E-UnpackPack}$$

$$\frac{t_{12} \longrightarrow t'_{12}}{\{*T_{11}, t_{12}\}\ \texttt{as}\ T_1 \longrightarrow \{*T_{11}, t'_{12}\}\ \texttt{as}\ T_1}\ \text{E-Pack}$$

$$\frac{t_1 \longrightarrow t'_1}{\texttt{let}\ \{X, x\} = t_1\ \texttt{in}\ t_2 \longrightarrow \texttt{let}\ \{X, x\} = t'_1\ \texttt{in}\ t_2}\ \text{E-Unpack}$$

- *New Typing Rules*

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

$$\frac{\Gamma \vdash t_2 : [X \mapsto U]T_2}{\Gamma \vdash \{*U, t_2\}\ \texttt{as}\ \{\exists X, T_2\} : \{\exists X, T_2\}}\ \text{T-Pack}$$

$$\frac{\Gamma \vdash t_1 : \{\exists X, T_{12}\} \quad \Gamma, X, x : T_{12} \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ \{X, x\} = t_1\ \texttt{in}\ t_2 : T_2}\ \text{T-Unpack}$$

Existential Types

Semantics and Typing

- *New Evaluation Rules*

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

$$\frac{}{\texttt{let}\ \{X, x\} = (\{T_{11}, v_{12}\}\ \texttt{as}\ T_1)\ \texttt{in}\ t_2 \longrightarrow [X \mapsto T_{11}][x \mapsto v_{12}]t_2}\ \textsc{E-UnpackPack}$$

$$\frac{t_{12} \longrightarrow t'_{12}}{\{*T_{11}, t_{12}\}\ \texttt{as}\ T_1 \longrightarrow \{*T_{11}, t'_{12}\}\ \texttt{as}\ T_1}\ \textsc{E-Pack}$$

$$\frac{t_1 \longrightarrow t'_1}{\texttt{let}\ \{X, x\} = t_1\ \texttt{in}\ t_2 \longrightarrow \texttt{let}\ \{X, x\} = t'_1\ \texttt{in}\ t_2}\ \textsc{E-Unpack}$$

- *New Typing Rules*

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

$$\frac{\Gamma \vdash t_2 : [X \mapsto U]T_2}{\Gamma \vdash \{*U, t_2\}\ \texttt{as}\ \{\exists X, T_2\} : \{\exists X, T_2\}}\ \textsc{T-Pack}$$

$$\frac{\Gamma \vdash t_1 : \{\exists X, T_{12}\} \quad \Gamma, X, x : T_{12} \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ \{X, x\} = t_1\ \texttt{in}\ t_2 : T_2}\ \textsc{T-Unpack}$$

- Note that the existential package does not expose the concrete type U.

What we have learned

We have extended our type systems:

- We introduced (untyped) type level computation, i.e., variables, abstraction and application.

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

Let’s connect

- propositional logic (in NJ) with
- the simply typed lambda calculus.

Term Erasure

Typing relation of the STLC[1]

Implicational fragment of propositional logic in the NJ system

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \textsc{T-Var} \quad \equiv \quad \frac{}{\Gamma, x : T, \Gamma' \vdash x : T}\ \textsc{T-Var}, \text{ax}$$

[1] *Disclaimer*: I greatly omitted the discourse on the subtleties of contexts (as lists vs. sets) in this lecture.

Term Erasure

Typing relation of the STLC[1]

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \textsc{T-Var} \quad \equiv \quad \frac{}{\Gamma, x : T, \Gamma' \vdash x : T}\ \textsc{T-Var}, \text{ax}$$

Implicational fragment of propositional logic in the NJ system

$$\frac{}{\Gamma, T, \Gamma' \vdash T}\ \text{ax}$$

[1] *Disclaimer*: I greatly omitted the discourse on the subtleties of contexts (as lists vs. sets) in this lecture.

Term Erasure

Typing relation of the STLC[1]

$$\frac{x : T \in \Gamma}{\Gamma \vdash {\color{green}x} : T}\ \textsc{T-Var} \quad \equiv \quad \frac{}{\Gamma, {\color{green}x} : T, \Gamma' \vdash {\color{green}x} : T}\ \textsc{T-Var}, \text{ax}$$

$$\frac{\Gamma, {\color{green}x} : T_1 \vdash {\color{green}t_2} : T_2}{\Gamma \vdash {\color{green}\lambda x : T_1 . t_2} : T_1 \to T_2}\ \textsc{T-Abs}, \to_I$$

Implicational fragment of propositional logic in the NJ system

$$\frac{}{\Gamma, T, \Gamma' \vdash T}\ \text{ax}$$

[1] *Disclaimer*: I greatly omitted the discourse on the subtleties of contexts (as lists vs. sets) in this lecture.

Term Erasure

Typing relation of the STLC[1]

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ T-VAR} \quad \equiv \quad \frac{}{\Gamma, x : T, \Gamma' \vdash x : T} \text{ T-VAR, ax}$$

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

Implicational fragment of propositional logic in the NJ system

$$\frac{}{\Gamma, T, \Gamma' \vdash T} \text{ ax}$$

$$\frac{\Gamma, T_1 \vdash T_2}{\Gamma \vdash T_1 \Rightarrow T_2} \Rightarrow_I$$

[1] *Disclaimer*: I greatly omitted the discourse on the subtleties of contexts (as lists vs. sets) in this lecture.

Term Erasure

Typing relation of the STLC[1]

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{T-VAR} \quad \equiv \quad \frac{}{\Gamma, x : T, \Gamma' \vdash x : T}\ \text{T-VAR, ax}$$

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

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

Implicational fragment of propositional logic in the NJ system

$$\frac{}{\Gamma, T, \Gamma' \vdash T}\ \text{ax}$$

$$\frac{\Gamma, T_1 \vdash T_2}{\Gamma \vdash T_1 \Rightarrow T_2}\ \Rightarrow_I$$

[1] *Disclaimer*: I greatly omitted the discourse on the subtleties of contexts (as lists vs. sets) in this lecture.

Term Erasure

Typing relation of the STLC[1]

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{T-VAR} \quad \equiv \quad \frac{}{\Gamma, x : T, \Gamma' \vdash x : T}\ \text{T-VAR, ax}$$

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

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

Implicational fragment of propositional logic in the NJ system

$$\frac{}{\Gamma, T, \Gamma' \vdash T}\ \text{ax}$$

$$\frac{\Gamma, T_1 \vdash T_2}{\Gamma \vdash T_1 \Rightarrow T_2}\ \Rightarrow_I$$

$$\frac{\Gamma, T_{11} \to T_{12} \quad \Gamma \vdash T_{11}}{\Gamma \vdash T_{12}}\ \Rightarrow_E$$

[1] *Disclaimer*: I greatly omitted the discourse on the subtleties of contexts (as lists vs. sets) in this lecture.

This correspondence is the foundation for proof assistants such as Coq and Lean and dependently-typed languages such as Agda.

Theorem (Curry-Howard Correspondence)

Given a context Γ and a type T, the term erasing procedure gives a one-to-one correspondence between

- *λ-terms of type T in context Γ, i.e., $\Gamma \vdash t : T$, and*
- *proofs in the implicational fragment of NJ of $\Gamma \vdash T$.*

The Curry-Howard Correspondence

History[1]

1934 **Haskell Curry** – mathematician

- Correspondence between the implicational fragement of NJ and the simply typed lambda calculus (STLC).
- Curry and Feys: correspondence not only between propositions and types but also between proofs and terms.

[1] Philip Wadler. "Propositions as Types". In: *Commun. ACM* (2015).

Proof of surjectivity from proofs to terms.

Given a proof of the form: the corresponding typing derivation is:

Case *ax*: $\dfrac{}{\Gamma, T, \Gamma' \vdash T}\ \text{ax}$

Case *intro*:

Case *elim*:

$\square$

Proof of surjectivity from proofs to terms.

Given a proof of the form: the corresponding typing derivation is:

Case *ax*:

$$\frac{}{\Gamma, T, \Gamma' \vdash T}\ \text{ax} \qquad \frac{}{\Gamma, x : T, \Gamma' \vdash x : T}\ \text{T-Var, ax}$$

Case *intro*:

Case *elim*:

$\square$

Proof of surjectivity from proofs to terms.

Given a proof of the form: the corresponding typing derivation is:

Case *ax*:

$$\frac{}{\Gamma, T, \Gamma' \vdash T}\ \text{ax}$$

$$\frac{}{\Gamma, x : T, \Gamma' \vdash x : T}\ \text{T-VAR, ax}$$

Case *intro*:

$$\frac{\dfrac{\pi}{\Gamma, T_1 \vdash T_2}}{\Gamma \vdash T_1 \Rightarrow T_2} \Rightarrow_I$$

Case *elim*:

□

Proof of surjectivity from proofs to terms.

	Given a proof of the form:	the corresponding typing derivation is:
Case *ax*:	$\dfrac{}{\Gamma, T, \Gamma' \vdash T}\ \text{ax}$	$\dfrac{}{\Gamma, x : T, \Gamma' \vdash x : T}\ \text{T-VAR, ax}$
Case *intro*:	$\dfrac{\dfrac{\pi}{\Gamma, T_1 \vdash T_2}}{\Gamma \vdash T_1 \Rightarrow T_2} \Rightarrow_I$	$\dfrac{\dfrac{\vdots}{\Gamma, x : T_1 \vdash t_2 : T_2}}{\Gamma \vdash \lambda x : T_1.t_2 : T_1 \to T_2}\ \text{T-ABS}, \to_I$
Case *elim*:		

□

Proof of surjectivity from proofs to terms.

	Given a proof of the form:	the corresponding typing derivation is:
Case *ax*:	$\dfrac{}{\Gamma, T, \Gamma' \vdash T}\ \text{ax}$	$\dfrac{}{\Gamma, x : T, \Gamma' \vdash x : T}\ \text{T-VAR, ax}$
Case *intro*:	$\dfrac{\dfrac{\pi}{\Gamma, T_1 \vdash T_2}}{\Gamma \vdash T_1 \Rightarrow T_2} \Rightarrow_I$	$\dfrac{\dfrac{\vdots}{\Gamma, x : T_1 \vdash t_2 : T_2}}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2}\ \text{T-ABS}, \to_I$
Case *elim*:	$\dfrac{\dfrac{\pi}{\Gamma, T_{11} \to T_{12}} \quad \dfrac{\pi'}{\Gamma \vdash T_{11}}}{\Gamma \vdash T_{12}} \Rightarrow_E$	

$\square$

Proof of surjectivity from proofs to terms.

Given a proof of the form: the corresponding typing derivation is:

Case *ax*:

$$\frac{}{\Gamma, T, \Gamma' \vdash T}\text{ax} \qquad \frac{}{\Gamma, x : T, \Gamma' \vdash x : T}\text{T-VAR, ax}$$

Case *intro*:

$$\frac{\dfrac{\pi}{\Gamma, T_1 \vdash T_2}}{\Gamma \vdash T_1 \Rightarrow T_2} \Rightarrow_I \qquad \frac{\dfrac{\vdots}{\Gamma, x : T_1 \vdash t_2 : T_2}}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2}\text{T-ABS}, \to_I$$

Case *elim*:

$$\frac{\dfrac{\pi}{\Gamma, T_{11} \to T_{12}} \quad \dfrac{\pi'}{\Gamma \vdash T_{11}}}{\Gamma \vdash T_{12}} \Rightarrow_E \qquad \frac{\dfrac{\vdots}{\Gamma, t_1 : T_{11} \to T_{12}} \quad \dfrac{\vdots}{\Gamma \vdash t_2 : T_{11}}}{\Gamma \vdash t_1\ t_2 : T_{12}}\text{T-APP}, \to_E$$

$\square$

Proof of injectivity from typed terms to proofs.

$\square$

Proof of injectivity from typed terms to proofs.

1. The *uniqueness of types* property assures that there is exactly one typing derivation for a typed term.

□

Proof of injectivity from typed terms to proofs.

1. The *uniqueness of types* property assures that there is exactly one typing derivation for a typed term.
2. Using the term erasure gives a proof $\Gamma \vdash T$ for every $\Gamma \vdash t : T$.

$\square$

Proof of injectivity from typed terms to proofs.

1. The *uniqueness of types* property assures that there is exactly one typing derivation for a typed term.
2. Using the term erasure gives a proof $\Gamma \vdash T$ for every $\Gamma \vdash t : T$.

$\square$

Typable λ-terms are proof *witnesses*.

The Curry-Howard Correspondence

History[1]

1934 **Haskell Curry** – mathematician

- Correspondence between the implicational fragement of NJ and the simply typed lambda calculus (STLC).
- Curry and Feys: correspondence not only between propositions and types but also between proofs and terms.

1969 **William A. Howard** – logician

- Correspondence extends to the other propositional connectives of NJ and the STLC with product, sum and unit types.
- Proof simplification corresponds to term evaluation!

[1] Philip Wadler. "Propositions as Types". In: *Commun. ACM* (2015).

Term Substitution and Proof Substitution

Proof substitution

Substitution Lemma for typed terms
(Preservation of types under substitution)

The Quest for the Shortest Proof

- Proofs sometimes perform "useless" work, i.e., they take a detour.
- Consider these examples:

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash A_1} \quad \dfrac{\pi'}{\Gamma \vdash A_2}}{\Gamma \vdash A_1 \wedge A_2}\,(\wedge_I)}{\Gamma \vdash A_1}\,(\wedge_E^l) \qquad\qquad \dfrac{\dfrac{\dfrac{\pi}{\Gamma, A_1 \vdash A_2}}{\Gamma \vdash A_1 \Rightarrow A_2}\,(\Rightarrow_I) \quad \dfrac{\pi'}{\Gamma \vdash A_1}}{\Gamma \vdash A_2}\,(\Rightarrow_E)$$

- We are interested in defining a procedure that transforms a proof into a proof without detours.
- In some sense, such a procedure "executes" a proof.

Cuts

- In general, a *cut* is the use of a lemma inside another proof.
- But, a *cut* in a proof is an elimination rule whose principal (leftmost) premise is proven via an introduction rule of the same connective.

$$
\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash A_1} \quad \dfrac{\pi'}{\Gamma \vdash A_2}}{\Gamma \vdash A_1 \wedge A_2}\,(\wedge_I)}{\Gamma \vdash A_1}\,(\wedge^l_E)
\qquad\qquad
\dfrac{\dfrac{\dfrac{\pi}{\Gamma, A_1 \vdash A_2}}{\Gamma \vdash A_1 \Rightarrow A_2}\,(\Rightarrow_I) \quad \dfrac{\pi'}{\Gamma \vdash A_1}}{\Gamma \vdash A_2}\,(\Rightarrow_E)
$$

- Lemmas provide proof modularity and foster reuse!
- But lemmas are often more general than what we are actually trying to prove.
- Hence, we are interested in a transformation that removes cuts.

Proof Substitution

Proof substitution: replacing axioms with proofs.

Example: Consider the following two proofs:

$$\pi = \cfrac{\cfrac{\cfrac{\cfrac{}{\Gamma, A_1 \vdash A_1}\,(ax)}{\Gamma, A_1, A_2 \vdash A_1}\,(wk) \quad \cfrac{\cfrac{}{\Gamma, A_1 \vdash A_1}\,(ax)}{\Gamma, A_1, A_2 \vdash A_1}\,(wk)}{\Gamma, A_1, A_2 \vdash A_1 \wedge A_1}\,(\wedge_I)}{\Gamma, A_1 \vdash A_2 \Rightarrow A_1 \wedge A_1}\,(\Rightarrow_I)$$

$$\xrightarrow{\text{substitute } \pi'}$$

$$\cfrac{\cfrac{\cfrac{\cfrac{\pi'}{\Gamma \vdash A_1}}{\Gamma, A_2 \vdash A_1}\,(wk) \quad \cfrac{\cfrac{\pi'}{\Gamma \vdash A_1}}{\Gamma, A_2 \vdash A_1}\,(wk)}{\Gamma, A_2 \vdash A_1 \wedge A_1}\,(\wedge_I)}{\Gamma \vdash A_2 \Rightarrow A_1 \wedge A_1}\,(\Rightarrow_I)$$

$$\pi' = \cfrac{\vdots}{\Gamma \vdash A_1}$$

Proposition (Proof substitution)

Given provable sequents

$$\frac{\pi}{\Gamma, A_1, \Gamma' \vdash A_2} \quad \textit{and} \quad \frac{\pi'}{\Gamma \vdash A_1}$$

the sequent $\Gamma, \Gamma' \vdash A_2$ *is provable by*

$$\frac{\pi[A_1 \longmapsto \pi']}{\Gamma, \Gamma' \vdash A_2}$$

(The proof is by induction on π.)

$$\frac{\Gamma \vdash A_1 \quad \Gamma, A_1, \Gamma' \vdash A_2}{\Gamma, \Gamma' \vdash A_2}\ (cut)$$

Cut Elimination

Definition (Cut Elimination Property)

A logic system has the *cut elimination property* if for every provable formula there exists a cut-free proof.

- Generally, we not only want to know whether there exists such a cut-free proof but we want a procedure that transforms any proof into a cut-free one.
- First introduced by Gentzen by the name *Hauptsatz*.

Cut Elimination Rules

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma, A_1 \vdash A_2}}{\Gamma \vdash A_1 \Rightarrow A_2}\,(\Rightarrow_I) \qquad \dfrac{\pi'}{\Gamma \vdash A_1}}{\Gamma \vdash A_2}\,(\Rightarrow_E) \quad \rightsquigarrow \quad \dfrac{\pi[A_1 \longmapsto \pi']}{\Gamma \vdash A_2}$$

Cut Elimination Rules

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma, A_1 \vdash A_2}}{\Gamma \vdash A_1 \Rightarrow A_2}\,{\scriptstyle(\Rightarrow_I)} \qquad \dfrac{\pi'}{\Gamma \vdash A_1}}{\Gamma \vdash A_2}\,{\scriptstyle(\Rightarrow_E)} \qquad \rightsquigarrow \qquad \dfrac{\pi[A_1 \longmapsto \pi']}{\Gamma \vdash A_2}$$

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash A_1} \qquad \dfrac{\pi'}{\Gamma \vdash A_2}}{\Gamma \vdash A_1 \wedge A_2}\,{\scriptstyle(\wedge_I)}}{\Gamma \vdash A_1}\,{\scriptstyle(\wedge_E^l)} \qquad \rightsquigarrow \qquad \dfrac{\pi}{\Gamma \vdash A_1}$$

Cut Elimination Rules

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma, A_1 \vdash A_2}}{\Gamma \vdash A_1 \Rightarrow A_2}\,{\scriptstyle(\Rightarrow_I)} \qquad \dfrac{\pi'}{\Gamma \vdash A_1}}{\Gamma \vdash A_2}\,{\scriptstyle(\Rightarrow_E)} \quad\rightsquigarrow\quad \dfrac{\pi[A_1 \longmapsto \pi']}{\Gamma \vdash A_2}$$

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash A_1} \qquad \dfrac{\pi'}{\Gamma \vdash A_2}}{\Gamma \vdash A_1 \wedge A_2}\,{\scriptstyle(\wedge_I)}}{\Gamma \vdash A_1}\,{\scriptstyle(\wedge_E^l)} \quad\rightsquigarrow\quad \dfrac{\pi}{\Gamma \vdash A_1}$$

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash A_1} \qquad \dfrac{\pi'}{\Gamma \vdash A_2}}{\Gamma \vdash A_1 \wedge A_2}\,{\scriptstyle(\wedge_I)}}{\Gamma \vdash A_2}\,{\scriptstyle(\wedge_E^r)} \quad\rightsquigarrow\quad \dfrac{\pi'}{\Gamma \vdash A_2}$$

Cut Elimination Rules
Continued

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash A_1}}{\Gamma \vdash A_1 \vee A_2}\,(\vee_I^l) \qquad \dfrac{\pi'}{\Gamma, A_1 \vdash A_3} \qquad \dfrac{\pi''}{\Gamma, A_2 \vdash A_3}}{\Gamma \vdash A_3}\,(\vee_E) \quad \rightsquigarrow \quad \dfrac{\pi'[A_1 \longmapsto \pi]}{\Gamma \vdash A_3}$$

Cut Elimination Rules
Continued

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash A_1}}{\Gamma \vdash A_1 \vee A_2}\,(\vee_I^l) \qquad \dfrac{\pi'}{\Gamma, A_1 \vdash A_3} \qquad \dfrac{\pi''}{\Gamma, A_2 \vdash A_3}}{\Gamma \vdash A_3}\,(\vee_E) \quad \rightsquigarrow \quad \dfrac{\pi'[A_1 \longmapsto \pi]}{\Gamma \vdash A_3}$$

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash A_2}}{\Gamma \vdash A_1 \vee A_2}\,(\vee_I^r) \qquad \dfrac{\pi'}{\Gamma, A_1 \vdash A_3} \qquad \dfrac{\pi''}{\Gamma, A_2 \vdash A_3}}{\Gamma \vdash A_3}\,(\vee_E) \quad \rightsquigarrow \quad \dfrac{\pi''[A_2 \longmapsto \pi]}{\Gamma \vdash A_3}$$

Term Substitution and Proof Substitution

Proof substitution

Given provable sequents

$$\frac{\pi}{\Gamma, S, \Gamma' \vdash T} \quad \text{and} \quad \frac{\pi'}{\Gamma \vdash S},$$

the sequent $\Gamma, \Gamma' \vdash T$ is provable by

$$\frac{\pi[S \longmapsto \pi']}{\Gamma, \Gamma' \vdash T}.$$

Substitution Lemma for typed terms (Preservation of types under substitution)

If $\Gamma, x : S \vdash t : T$ and $\Gamma \vdash s : S$ then $\Gamma \vdash [x \mapsto s]t : T$.

Assumption: Preservation of types (under substitution).

Preservation of Types under β-Reduction

Lemma (Preservation of Types under Substitution)

If $\Gamma, x : S \vdash t : T$ *and* $\Gamma \vdash s : S$, *then* $\Gamma \vdash [x \mapsto s]t : T$.

Preservation

Proof

- The proof is by induction on the typing derivation for $\Gamma, x : S \vdash t : T$.
- Cases:

Case	*Rule with* $\Gamma, x : S \vdash t : T$	*Proof*

- The proof is by induction on the typing derivation for $\Gamma, x : S \vdash t : T$.
- Cases:

Case	*Rule with* $\Gamma, x : S \vdash t : T$	*Proof*
T-VAR	$\dfrac{}{\Gamma, x : S \vdash z : T}$ T-VAR	There are two cases to consider:

Preservation
Proof

- The proof is by induction on the typing derivation for $\Gamma, x : S \vdash t : T$.
- Cases:

Case	*Rule with* $\Gamma, x : S \vdash t : T$	*Proof*
T-VAR	$\dfrac{}{\Gamma, x : S \vdash z : T}$ T-VAR	There are two cases to consider: $z = x$ such that $[x \mapsto s]z = s$ and $\Gamma \vdash s : S$ is an assumption of the lemma.

Preservation
Proof

- The proof is by induction on the typing derivation for $\Gamma, x : S \vdash t : T$.
- Cases:

Case	*Rule with* $\Gamma, x : S \vdash t : T$	*Proof*
T-VAR	$\dfrac{}{\Gamma, x : S \vdash z : T}$ T-VAR	There are two cases to consider: $z = x$ such that $[x \mapsto s]z = s$ and $\Gamma \vdash s : S$ is an assumption of the lemma. $z \neq x$ such that $[x \mapsto s]z = z$ and $\Gamma \vdash z : T$ is immediate.

Preservation
Proof

- The proof is by induction on the typing derivation for $\Gamma, x : S \vdash t : T$.
- Cases:

Case	*Rule with* $\Gamma, x : S \vdash t : T$	*Proof*

Preservation
Proof

- The proof is by induction on the typing derivation for $\Gamma, x : S \vdash t : T$.
- Cases:

Case	*Rule with* $\Gamma, x : S \vdash t : T$	*Proof*
T-ABS	$\dfrac{\Gamma, x : S, y : T_2 \vdash t_1 : T_1}{\Gamma, x : S \vdash \lambda y : T_2.t_1 : T_2 \to T_1}$ T-ABS	By alpha conversion, $x \neq y$ and $y \notin FV(s)$.

- The proof is by induction on the typing derivation for $\Gamma, x : S \vdash t : T$.
- Cases:

Case	*Rule with* $\Gamma, x : S \vdash t : T$	*Proof*
T-Abs	$$\dfrac{\Gamma, x : S, y : T_2 \vdash t_1 : T_1}{\Gamma, x : S \vdash \lambda y : T_2.t_1 : T_2 \to T_1}\ \text{T-Abs}$$	By alpha conversion, $x \neq y$ and $y \notin FV(s)$. **Now we have:** If $\Gamma, x : S, y : T_2 \vdash t_1 : T_1$ and $\Gamma \vdash s : S$, then ...

Preservation

Proof

- The proof is by induction on the typing derivation for $\Gamma, x : S \vdash t : T$.
- Cases:

Case	*Rule with* $\Gamma, x : S \vdash t : T$	*Proof*
T-ABS	$\dfrac{\Gamma, x : S, y : T_2 \vdash t_1 : T_1}{\Gamma, x : S \vdash \lambda y : T_2.t_1 : T_2 \to T_1}$ T-ABS	By alpha conversion, $x \neq y$ and $y \notin FV(s)$. **Now we have:** If $\Gamma, x : S, y : T_2 \vdash t_1 : T_1$ and $\Gamma \vdash s : S$, then ... **But we need:** If $\Gamma, x : S \vdash t_1 : T_1$ and $\Gamma \vdash s : S$, then ... .

Preservation
Proof

- The proof is by induction on the typing derivation for $\Gamma, x : S \vdash t : T$.
- Cases:

Case	*Rule with* $\Gamma, x : S \vdash t : T$	*Proof*
T-ABS	$\dfrac{\Gamma, x : S, y : T_2 \vdash t_1 : T_1}{\Gamma, x : S \vdash \lambda y : T_2.t_1 : T_2 \to T_1}$ T-ABS	By alpha conversion, $x \neq y$ and $y \notin FV(s)$.

Now we have: If $\Gamma, x : S, y : T_2 \vdash t_1 : T_1$ and $\Gamma \vdash s : S$, then ...

But we need: If $\Gamma, x : S \vdash t_1 : T_1$ and $\Gamma \vdash s : S$, then ... .

By permutation, we get $\Gamma, y : T_2, x : S$.

Preservation
Proof

- The proof is by induction on the typing derivation for $\Gamma, x : S \vdash t : T$.
- Cases:

Case	*Rule with* $\Gamma, x : S \vdash t : T$	*Proof*
T-ABS	$\dfrac{\Gamma, x : S, y : T_2 \vdash t_1 : T_1}{\Gamma, x : S \vdash \lambda y : T_2.t_1 : T_2 \to T_1}$ T-ABS	By alpha conversion, $x \neq y$ and $y \notin FV(s)$. **Now we have:** If $\Gamma, x : S, y : T_2 \vdash t_1 : T_1$ and $\Gamma \vdash s : S$, then ... **But we need:** If $\Gamma, x : S \vdash t_1 : T_1$ and $\Gamma \vdash s : S$, then ... . By permutation, we get $\Gamma, y : T_2, x : S$. By weakening, we get $\Gamma, y : T_2$.

Preservation
Proof

- The proof is by induction on the typing derivation for $\Gamma, x : S \vdash t : T$.
- Cases:

Case	*Rule with* $\Gamma, x : S \vdash t : T$	*Proof*

Preservation

Proof

- The proof is by induction on the typing derivation for $\Gamma, x : S \vdash t : T$.
- Cases:

Case	*Rule with* $\Gamma, x : S \vdash t : T$	*Proof*
T-ABS	$\dfrac{\Gamma, x : S, y : T_2 \vdash t_1 : T_1}{\Gamma, x : S \vdash \lambda y : T_2.t_1 : T_2 \to T_1}$ T-ABS	By definition of substitution: $[x \mapsto s](\lambda y : T_2.t_1) = \lambda y : T_2.[x \mapsto s]t_1$

Preservation
Proof

- The proof is by induction on the typing derivation for $\Gamma, x : S \vdash t : T$.
- Cases:

Case	*Rule with* $\Gamma, x : S \vdash t : T$	*Proof*
T-ABS	$\dfrac{\Gamma, x : S, y : T_2 \vdash t_1 : T_1}{\Gamma, x : S \vdash \lambda y : T_2.t_1 : T_2 \to T_1}$ T-ABS	By definition of substitution: $[x \mapsto s](\lambda y : T_2.t_1) = \lambda y : T_2.[x \mapsto s]t_1$ By induction hypothesis on T-ABS, we have that $\lambda y : T_2.[x \mapsto s]t_1$ is well-typed:

- The proof is by induction on the typing derivation for $\Gamma, x : S \vdash t : T$.
- Cases:

Case	*Rule with* $\Gamma, x : S \vdash t : T$	*Proof*
T-ABS	$\dfrac{\Gamma, x : S, y : T_2 \vdash t_1 : T_1}{\Gamma, x : S \vdash \lambda y : T_2.t_1 : T_2 \to T_1}$ T-ABS	By definition of substitution: $[x \mapsto s](\lambda y : T_2.t_1) = \lambda y : T_2.[x \mapsto s]t_1$ By induction hypothesis on T-ABS, we have that $\lambda y : T_2.[x \mapsto s]t_1$ is well-typed: $\dfrac{\Gamma, x : S, y : T_2 \vdash [x \mapsto s]t_1 : T_1}{\Gamma, x : S \vdash \lambda y : T_2.[x \mapsto s]t_1 : T_2 \to T_1}$ T-ABS

Preservation

Proof

- The proof is by induction on the typing derivation for $\Gamma, x : S \vdash t : T$.
- Cases:

Case	*Rule with* $\Gamma, x : S \vdash t : T$	*Proof*

Preservation
Proof

- The proof is by induction on the typing derivation for $\Gamma, x : S \vdash t : T$.
- Cases:

Case	*Rule with* $\Gamma, x : S \vdash t : T$	*Proof*
T-APP	$\dfrac{\Gamma, x : S \vdash t_1 : T_2 \to T_1 \quad \Gamma, x : S \vdash t_2 : T_2}{\Gamma, x : S \vdash t_1\ t_2 : T_1}$ T-APP	By definition of substitution: $[x \mapsto s](t_1\ t_2) = [x \mapsto s]t_1\ [x \mapsto s]t_2$

Preservation

Proof

- The proof is by induction on the typing derivation for $\Gamma, x : S \vdash t : T$.
- Cases:

Case	*Rule with* $\Gamma, x : S \vdash t : T$	*Proof*
T-App	$$\frac{\Gamma, x : S \vdash t_1 : T_2 \to T_1 \quad \Gamma, x : S \vdash t_2 : T_2}{\Gamma, x : S \vdash t_1\ t_2 : T_1}\ \text{T-App}$$	By definition of substitution: $[x \mapsto s](t_1\ t_2) = [x \mapsto s]t_1\ [x \mapsto s]t_2$ By definition of T-App, $[x \mapsto s]t_1$ and $[x \mapsto s]t_2$ are well-typed:

Preservation
Proof

- The proof is by induction on the typing derivation for $\Gamma, x : S \vdash t : T$.
- Cases:

Case	*Rule with* $\Gamma, x : S \vdash t : T$	*Proof*
T-APP	$\dfrac{\Gamma, x : S \vdash t_1 : T_2 \to T_1 \quad \Gamma, x : S \vdash t_2 : T_2}{\Gamma, x : S \vdash t_1\ t_2 : T_1}$ T-APP	By definition of substitution: $[x \mapsto s](t_1\ t_2) = [x \mapsto s]t_1\ [x \mapsto s]t_2$ By definition of T-APP, $[x \mapsto s]t_1$ and $[x \mapsto s]t_2$ are well-typed: $\dfrac{\Gamma, x : S \vdash [x \mapsto s]t_1 : T_2 \to T_1 \quad \Gamma, x : S \vdash [x \mapsto s]t_2 : T_2}{\Gamma, x : S \vdash [x \mapsto s](t_1\ t_2) : T_1}$ T-APP

Preservation
Proof

- The proof is by induction on the typing derivation for $\Gamma, x : S \vdash t : T$.
- Cases:

Case	*Rule with* $\Gamma, x : S \vdash t : T$	*Proof*
T-APP	$$\dfrac{\Gamma, x : S \vdash t_1 : T_2 \to T_1 \quad \Gamma, x : S \vdash t_2 : T_2}{\Gamma, x : S \vdash t_1\ t_2 : T_1}\ \text{T-App}$$	By definition of substitution: $[x \mapsto s](t_1\ t_2) = [x \mapsto s]t_1\ [x \mapsto s]t_2$ By definition of T-APP, $[x \mapsto s]t_1$ and $[x \mapsto s]t_2$ are well-typed: $$\dfrac{\Gamma, x : S \vdash [x \mapsto s]t_1 : T_2 \to T_1 \quad \Gamma, x : S \vdash [x \mapsto s]t_2 : T_2}{\Gamma, x : S \vdash [x \mapsto s](t_1\ t_2) : T_1}\ \text{T-App}$$ $\square$

Preservation

Theorem (Preservation)

If $\Gamma \vdash t : T$ *and* $t \longrightarrow t'$*, then* $\Gamma \vdash t' : T$

- The proof is by induction on the typing derivation for $\Gamma, x : S \vdash t : T$.
- The most interesting case is this:

Case	*Rule with* $\Gamma, x : S \vdash t : T$	*Proof*

Preservation

Theorem (Preservation)

If $\Gamma \vdash t : T$ *and* $t \longrightarrow t'$, *then* $\Gamma \vdash t' : T$

- The proof is by induction on the typing derivation for $\Gamma, x : S \vdash t : T$.
- The most interesting case is this:

Case	*Rule with* $\Gamma, x : S \vdash t : T$	*Proof*
T-App	$\dfrac{\Gamma, t : T_{11} \vdash t_1 : T_{11} \to T_{12} \quad \Gamma, \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}$ T-App	By E-AppAbs, we have: $\dfrac{}{(\lambda x : T_{11}.t_{12})\ v \longrightarrow [x \mapsto v]t_{12}}$ E-AppAbs

Preservation

Theorem (Preservation)

If $\Gamma \vdash t : T$ *and* $t \longrightarrow t'$*, then* $\Gamma \vdash t' : T$

- The proof is by induction on the typing derivation for $\Gamma, x : S \vdash t : T$.
- The most interesting case is this:

Case	*Rule with* $\Gamma, x : S \vdash t : T$	*Proof*
T-APP	$\dfrac{\Gamma, t : T_{11} \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma, \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}$ T-APP	By E-APPABS, we have: $\dfrac{}{(\lambda x : T_{11}.t_{12})\ v \longrightarrow [x \mapsto v]t_{12}}$ E-APPABS By the substitution lemma, we know that $\Gamma \vdash [x \mapsto v]t_{12} : T_{12}$.

Preservation

Theorem (Preservation)

If $\Gamma \vdash t : T$ *and* $t \longrightarrow t'$ *, then* $\Gamma \vdash t' : T$

- The proof is by induction on the typing derivation for $\Gamma, x : S \vdash t : T$.
- The most interesting case is this:

Case	*Rule with* $\Gamma, x : S \vdash t : T$	*Proof*
T-APP	$$\dfrac{\Gamma, t : T_{11} \vdash t_1 : T_{11} \to T_{12} \quad \Gamma, \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}\ \text{T-APP}$$	By E-APPABS, we have: $$\dfrac{}{(\lambda x : T_{11}.t_{12})\ v \longrightarrow [x \mapsto v]t_{12}}\ \text{E-APPABS}$$ By the substitution lemma, we know that $\Gamma \vdash [x \mapsto v]t_{12} : T_{12}$.

β-Reduction and Cut Elimination

- Assume types $T_{11} = S$ and $T_{12} = T$ with the respective terms $t_{11} = s$ and $t_{12} = t$.

β-Reduction and Cut Elimination

- Assume types $T_{11} = S$ and $T_{12} = T$ with the respective terms $t_{11} = s$ and $t_{12} = t$.
- Let's have a look at at these two steps in combination again:

β-Reduction and Cut Elimination

- Assume types $T_{11} = S$ and $T_{12} = T$ with the respective terms $t_{11} = s$ and $t_{12} = t$.
- Let's have a look at at these two steps in combination again:

β-Reduction

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, x : S \vdash t : T}}{\Gamma \vdash (\lambda x : S.t) : S \to T}\ \text{T-ABS}, \to_I \qquad \dfrac{\vdots}{\Gamma \vdash y : S}}{\Gamma \vdash t\ y : T}\ \text{T-APP}, \to_E \quad \xrightarrow{\text{E-APPABS}} \quad \overline{\Gamma \vdash [x \mapsto y]t : T}$$

β-Reduction and Cut Elimination

- Assume types $T_{11} = S$ and $T_{12} = T$ with the respective terms $t_{11} = s$ and $t_{12} = t$.
- Let's have a look at at these two steps in combination again:

Cut Elimination

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma, S \vdash T}}{\Gamma \vdash S \Rightarrow T}\,(\Rightarrow_I) \quad \dfrac{\pi'}{\Gamma \vdash S}}{\Gamma \vdash T}\,(\Rightarrow_E) \quad \rightsquigarrow \quad \dfrac{\pi[S \longmapsto \pi']}{\Gamma \vdash T}$$

β-Reduction

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, x : S \vdash t : T}}{\Gamma \vdash (\lambda x : S.t) : S \to T}\,\text{T-ABS},\to_I \quad \dfrac{\vdots}{\Gamma \vdash y : S}}{\Gamma \vdash t\ y : T}\,\text{T-APP},\to_E \quad \xrightarrow{\text{E-APPABS}} \quad \dfrac{}{\Gamma \vdash [x \mapsto y]t : T}$$

β-Reduction and Cut Elimination

- Assume types $T_{11} = S$ and $T_{12} = T$ with the respective terms $t_{11} = s$ and $t_{12} = t$.
- Let's have a look at at these two steps in combination again:

Cut Elimination

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma, S \vdash T}}{\Gamma \vdash S \Rightarrow T}\,(\Rightarrow_I) \quad \dfrac{\pi'}{\Gamma \vdash S}}{\Gamma \vdash T}\,(\Rightarrow_E) \quad \rightsquigarrow \quad \dfrac{\pi[S \longmapsto \pi']}{\Gamma \vdash T}$$

β-Reduction

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, x : S \vdash t : T}}{\Gamma \vdash (\lambda x : S.t) : S \to T}\,\text{T-ABS},\to_I \quad \dfrac{\vdots}{\Gamma \vdash y : S}}{\Gamma \vdash t\ y : T}\,\text{T-APP},\to_E \quad \xrightarrow{\text{E-APPABS}} \quad \dfrac{}{\Gamma \vdash [x \mapsto y]t : T}$$

- Notice the correspondence of proofs and terms.

Term Substitution and Proof Substitution

Proof substitution

Given provable sequents

$$\frac{\pi}{\Gamma, S, \Gamma' \vdash T} \quad \text{and} \quad \frac{\pi'}{\Gamma \vdash S} \;,$$

the sequent $\Gamma, \Gamma' \vdash T$ is provable by

$$\frac{\pi[S \longmapsto \pi']}{\Gamma, \Gamma' \vdash T} \;.$$

Substitution Lemma for typed terms
(Preservation of types under substitution)

If $\Gamma, x : S \vdash t : T$ and $\Gamma \vdash s : S$
then $\Gamma \vdash [x \mapsto s]t : T$.

Term Substitution and Proof Substitution

Proof substitution

Given provable sequents

$$\frac{\pi}{\Gamma, S, \Gamma' \vdash T} \quad \text{and} \quad \frac{\pi'}{\Gamma \vdash S},$$

the sequent $\Gamma, \Gamma' \vdash T$ is provable by

$$\frac{\pi[S \longmapsto \pi']}{\Gamma, \Gamma' \vdash T}.$$

Substitution Lemma for typed terms
(Preservation of types under substitution)

If $\Gamma, x : S \vdash t : T$ and $\Gamma \vdash s : S$
then $\Gamma \vdash [x \mapsto s]t : T$.

$$\frac{\Gamma, x : S \vdash t : T \qquad \Gamma \vdash s : S}{\Gamma \vdash [x \mapsto s]t : T}$$

Term Substitution and Proof Substitution

Proof substitution

Given provable sequents

$$\frac{\pi}{\Gamma, S, \Gamma' \vdash T} \quad \text{and} \quad \frac{\pi'}{\Gamma \vdash S},$$

the sequent $\Gamma, \Gamma' \vdash T$ is provable by

$$\frac{\pi[S \longmapsto \pi']}{\Gamma, \Gamma' \vdash T}.$$

$$\frac{\Gamma, S \vdash T \quad \Gamma \vdash S}{\Gamma \vdash T} \text{ (cut)}$$

Substitution Lemma for typed terms
(Preservation of types under substitution)

If $\Gamma, x : S \vdash t : T$ and $\Gamma \vdash s : S$
then $\Gamma \vdash [x \mapsto s]t : T$.

$$\frac{\Gamma, x : S \vdash t : T \quad \Gamma \vdash s : S}{\Gamma \vdash [x \mapsto s]t : T}$$

The Curry-Howard Correspondence

History[1]

1934 **Haskell Curry** – mathematician

- Correspondence between the implicational fragement of NJ and the simply typed lambda calculus (STLC).
- Curry and Feys: correspondence not only between propositions and types but also between proofs and terms.

1969 **William A. Howard** – logician

- Correspondence extends to the other propositional connectives of NJ and the STLC with product, sum and unit types.
- Proof simplification corresponds to term evaluation!
- The correspondence extends to first-order logic!

[1] Philip Wadler. “Propositions as Types”. In: *Commun. ACM* (2015).

Polymorphism and First-Order Logic

Typing relation

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

First-order logic

Polymorphism and First-Order Logic

Typing relation

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

First-order logic

$$\frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[x \longmapsto t]}\ (\forall_E) \quad \equiv \quad \frac{\Gamma \vdash \forall X.T_{12}}{\Gamma \vdash T_{12}[X \longmapsto T_2]}\ (\forall_E)$$

Polymorphism and First-Order Logic

Typing relation

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

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

First-order logic

$$\frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[x \longmapsto t]}\ (\forall_E) \quad \equiv \quad \frac{\Gamma \vdash \forall X.T_{12}}{\Gamma \vdash T_{12}[X \longmapsto T_2]}\ (\forall_E)$$

Polymorphism and First-Order Logic

Typing relation

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

$$\frac{\Gamma \vdash t_2 : T_2}{\Gamma \vdash \lambda X.t_2 : \forall X.T_2}\ \text{T-TAbs}$$

First-order logic

$$\frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[x \longmapsto t]}\ (\forall_E) \quad \equiv \quad \frac{\Gamma \vdash \forall X.T_{12}}{\Gamma \vdash T_{12}[X \longmapsto T_2]}\ (\forall_E)$$

$$\frac{\Gamma \vdash T_2}{\Gamma \vdash \forall X.T_2}\ (\forall_I)$$

Polymorphism and First-Order Logic

Typing relation

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

$$\frac{\Gamma \vdash t_2 : T_2}{\Gamma \vdash \lambda X.t_2 : \forall X.T_2}\ \text{T-TAbs}$$

$$\frac{\Gamma \vdash t_1 : \{\exists X, T_{12}\} \quad \Gamma, X, x : T_{12} \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ \{X, x\} = t_1\ \texttt{in}\ t_2 : T_2}\ \text{T-Unpack}$$

First-order logic

$$\frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[x \longmapsto t]}\ (\forall_E) \quad \equiv \quad \frac{\Gamma \vdash \forall X.T_{12}}{\Gamma \vdash T_{12}[X \longmapsto T_2]}\ (\forall_E)$$

$$\frac{\Gamma \vdash T_2}{\Gamma \vdash \forall X.T_2}\ (\forall_I)$$

Polymorphism and First-Order Logic

Typing relation

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

$$\frac{\Gamma \vdash t_2 : T_2}{\Gamma \vdash \lambda X.t_2 : \forall X.T_2}\ \text{T-TAbs}$$

$$\frac{\Gamma \vdash t_1 : \{\exists X, T_{12}\} \quad \Gamma, X, x : T_{12} \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ \{X, x\} = t_1\ \texttt{in}\ t_2 : T_2}\ \text{T-Unpack}$$

First-order logic

$$\frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[x \longmapsto t]}\ (\forall_E) \quad \equiv \quad \frac{\Gamma \vdash \forall X.T_{12}}{\Gamma \vdash T_{12}[X \longmapsto T_2]}\ (\forall_E)$$

$$\frac{\Gamma \vdash T_2}{\Gamma \vdash \forall X.T_2}\ (\forall_I)$$

$$\frac{\Gamma \vdash \exists X.T_{12} \quad \Gamma, T_{12} \vdash T_2}{\Gamma \vdash T_2}\ (\exists_E)$$

Polymorphism and First-Order Logic

Typing relation

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

$$\frac{\Gamma \vdash t_2 : T_2}{\Gamma \vdash \lambda X.t_2 : \forall X.T_2} \text{ T-TAbs}$$

$$\frac{\Gamma \vdash t_1 : \{\exists X, T_{12}\} \quad \Gamma, X, x : T_{12} \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ \{X, x\} = t_1\ \texttt{in}\ t_2 : T_2} \text{ T-Unpack}$$

$$\frac{\Gamma \vdash t_2 : [X \mapsto U]T_2}{\Gamma \vdash \{*U, t_2\}\ \texttt{as}\ \{\exists X, T_2\} : \{\exists X, T_2\}} \text{ T-Pack}$$

First-order logic

$$\frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[x \longmapsto t]}\ (\forall_E) \quad \equiv \quad \frac{\Gamma \vdash \forall X.T_{12}}{\Gamma \vdash T_{12}[X \longmapsto T_2]}\ (\forall_E)$$

$$\frac{\Gamma \vdash T_2}{\Gamma \vdash \forall X.T_2}\ (\forall_I)$$

$$\frac{\Gamma \vdash \exists X.T_{12} \quad \Gamma, T_{12} \vdash T_2}{\Gamma \vdash T_2}\ (\exists_E)$$

Polymorphism and First-Order Logic

Typing relation

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

$$\frac{\Gamma \vdash t_2 : T_2}{\Gamma \vdash \lambda X.t_2 : \forall X.T_2} \text{ T-TAbs}$$

$$\frac{\Gamma \vdash t_1 : \{\exists X, T_{12}\} \quad \Gamma, X, x : T_{12} \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ \{X, x\} = t_1\ \texttt{in}\ t_2 : T_2} \text{ T-Unpack}$$

$$\frac{\Gamma \vdash t_2 : [X \mapsto U]T_2}{\Gamma \vdash \{*U, t_2\}\ \texttt{as}\ \{\exists X, T_2\} : \{\exists X, T_2\}} \text{ T-Pack}$$

First-order logic

$$\frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[x \longmapsto t]}\ (\forall_E) \quad \equiv \quad \frac{\Gamma \vdash \forall X.T_{12}}{\Gamma \vdash T_{12}[X \longmapsto T_2]}\ (\forall_E)$$

$$\frac{\Gamma \vdash T_2}{\Gamma \vdash \forall X.T_2}\ (\forall_I)$$

$$\frac{\Gamma \vdash \exists X.T_{12} \quad \Gamma, T_{12} \vdash T_2}{\Gamma \vdash T_2}\ (\exists_E)$$

$$\frac{\Gamma \vdash T_2[X \longmapsto U]}{\Gamma \vdash \exists X.T_2}\ (\exists_I)$$

Term Substitution and Proof Substitution

- We extend the cut elimination procedure with the following cases:

Term Substitution and Proof Substitution

- We extend the cut elimination procedure with the following cases:

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash A(x)}}{\Gamma \vdash \forall x. A(x)}\ (\forall_I)}{\Gamma \vdash A(t)}\ (\forall_E) \quad \rightsquigarrow \quad \dfrac{\pi[x \longmapsto t]}{\Gamma \vdash A(t)}$$

Term Substitution and Proof Substitution

- We extend the cut elimination procedure with the following cases:

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash A(x)}}{\Gamma \vdash \forall x.A(x)}\,(\forall_I)}{\Gamma \vdash A(t)}\,(\forall_E) \quad \rightsquigarrow \quad \dfrac{\pi[x \longmapsto t]}{\Gamma \vdash A(t)}$$

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash A_1(t)}}{\Gamma \vdash \exists x.A_1(x)}\,(\exists_I) \qquad \dfrac{\pi'}{\Gamma, A_1(x) \vdash A_2}}{\Gamma \vdash A_2}\,(\exists_E) \quad \rightsquigarrow \quad \dfrac{\pi'[x \longmapsto t][A_1 \longmapsto \pi]}{\Gamma \vdash A_2}$$

- Universal quantification:

- Universal quantification:

Cut elimination:

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash t_{12} : T_{12}}}{\Gamma \vdash (\lambda X.t_{12}) : \forall X.T_{12}}\ (\text{T-TAbs},\forall_I)}{\Gamma \vdash (\lambda X.t_{12})\ [T_2] : [X \mapsto T_2]T_{12}}\ (\text{T-TApp},\forall_E) \quad \rightsquigarrow \quad \dfrac{\pi[X \longmapsto T_2]}{\Gamma \vdash [X \mapsto T_2]t_1 : [X \mapsto T_2]T_{12}}$$

- Universal quantification:

Cut elimination:

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash t_{12} : T_{12}}\;(\text{T-TAbs},\forall_I)}{\Gamma \vdash (\lambda X.t_{12}) : \forall X.T_{12}}\;(\text{T-TApp},\forall_E)}{\Gamma \vdash (\lambda X.t_{12})\;[T_2] : [X \mapsto T_2]T_{12}} \quad \rightsquigarrow \quad \dfrac{\pi[X \longmapsto T_2]}{\Gamma \vdash [X \mapsto T_2]t_1 : [X \mapsto T_2]T_{12}}$$

β-Reduction:

$$(\lambda X.t_{12})\;[T_2] \xrightarrow{\text{E-TAPPTABS}} [X \mapsto T_2]t_1$$

- Universal quantification:

Cut elimination:

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash t_{12} : T_{12}}}{\Gamma \vdash (\lambda X.t_{12}) : \forall X.T_{12}}\ (\text{T-TAbs},\forall_I)}{\Gamma \vdash (\lambda X.t_{12})\ [T_2] : [X \mapsto T_2]T_{12}}\ (\text{T-TApp},\forall_E) \quad \rightsquigarrow \quad \dfrac{\pi[X \longmapsto T_2]}{\Gamma \vdash [X \mapsto T_2]t_1 : [X \mapsto T_2]T_{12}}$$

β-Reduction:

$$(\lambda X.t_{12})\ [T_2] \xrightarrow{\text{E-TAPPTABS}} [X \mapsto T_2]t_1$$

- The existential case is analogous.

Overview

Logic **Programming Languages**

Overview

Logic
propositions

Programming Languages
types

Overview

Logic	**Programming Languages**
propositions	types
proposition $P \Rightarrow Q$	type $P \rightarrow Q$

Overview

Logic	**Programming Languages**
propositions	types
proposition $P \Rightarrow Q$	type $P \to Q$
proof of proposition P	term t of type P

Overview

Logic	**Programming Languages**
propositions	types
proposition $P \Rightarrow Q$	type $P \to Q$
proof of proposition P	term t of type P
proposition P is provable	type P is inhabited (by some term)

Overview

Logic	**Programming Languages**
propositions	types
proposition $P \Rightarrow Q$	type $P \rightarrow Q$
proof of proposition P	term t of type P
proposition P is provable	type P is inhabited (by some term)
cut elimination	β-reduction

Overview

Logic	**Programming Languages**
propositions	types
proposition $P \Rightarrow Q$	type $P \rightarrow Q$
proof of proposition P	term t of type P
proposition P is provable	type P is inhabited (by some term)
cut elimination	β-reduction
cut-free proof	term in normal form

Overview

Logic	**Programming Languages**
propositions	types
proposition $P \Rightarrow Q$	type $P \to Q$
proof of proposition P	term t of type P
proposition P is provable	type P is inhabited (by some term)
cut elimination	β-reduction
cut-free proof	term in normal form
proposition $P \wedge Q$	type $P \times Q$

Overview

Logic	**Programming Languages**
propositions	types
proposition $P \Rightarrow Q$	type $P \to Q$
proof of proposition P	term t of type P
proposition P is provable	type P is inhabited (by some term)
cut elimination	β-reduction
cut-free proof	term in normal form
proposition $P \wedge Q$	type $P \times Q$
proposition $P \vee Q$	type $P + Q$

Overview

Logic	**Programming Languages**
propositions	types
proposition $P \Rightarrow Q$	type $P \rightarrow Q$
proof of proposition P	term t of type P
proposition P is provable	type P is inhabited (by some term)
cut elimination	β-reduction
cut-free proof	term in normal form
proposition $P \wedge Q$	type $P \times Q$
proposition $P \vee Q$	type $P + Q$
$\top$	type `Unit`

Overview

Logic	Programming Languages
propositions	types
proposition $P \Rightarrow Q$	type $P \rightarrow Q$
proof of proposition P	term t of type P
proposition P is provable	type P is inhabited (by some term)
cut elimination	β-reduction
cut-free proof	term in normal form
proposition $P \land Q$	type $P \times Q$
proposition $P \lor Q$	type $P + Q$
$\top$	type `Unit`
$\bot$	type 0 (which has no term syntax, i.e., impossible to construct)

Overview

Logic	**Programming Languages**
propositions	types
proposition $P \Rightarrow Q$	type $P \rightarrow Q$
proof of proposition P	term t of type P
proposition P is provable	type P is inhabited (by some term)
cut elimination	β-reduction
cut-free proof	term in normal form
proposition $P \wedge Q$	type $P \times Q$
proposition $P \vee Q$	type $P + Q$
$\top$	type `Unit`
$\bot$	type 0 (which has no term syntax, i.e., impossible to construct) This is also called an uninhabited type.

The Curry-Howard Correspondence

History[1]

1934 **Haskell Curry** – mathematician

- Correspondence between the implicational fragement of NJ and the simply typed lambda calculus (STLC).
- Curry and Feys: correspondence not only between propositions and types but also between proofs and terms.

1969 **William A. Howard** – logician

- Correspondence extends to the other propositional connectives of NJ and the STLC with product, sum and unit types.
- Proof simplification corresponds to term evaluation!
- The correspondence extends even to **higher-order logic**!

[1] Philip Wadler. "Propositions as Types". In: *Commun. ACM* (2015).

Trusted Computing Base – TCB

Definition (Trusted Computing Base – TCB)

The *trusted computing base (TCB)* is the set of hardware and software components that a system(/platform) relies upon to perform correct (according to its specification – often secure and reliable) computations. A bug in the TCB can compromise the whole system.

Trusted Computing Base – TCB

Definition (Trusted Computing Base – TCB)

The *trusted computing base (TCB)* is the set of hardware and software components that a system(/platform) relies upon to perform correct (according to its specification – often secure and reliable) computations. A bug in the TCB can compromise the whole system.

- Current approaches try to minimize the size of the TCB
  - to reduce the complexity of the TCB and therewith the probability of bugs and
  - to make the TCB amenable to formal verification.

Trusted Computing Base – TCB

Definition (Trusted Computing Base – TCB)

The *trusted computing base (TCB)* is the set of hardware and software components that a system(/platform) relies upon to perform correct (according to its specification – often secure and reliable) computations. A bug in the TCB can compromise the whole system.

- Assume the TCB of a computing system is fully formally verified ... then there is a new TCB left: the "formal verification algorithm" in the proof assistant:
  - When propositions are types and proof are programs then this algorithm is the called *the type checker*.
  - Type checking is a relatively small and straightforward:
    - Check the argument types for function applications.
    - Make sure `match` expressions are exhaustive.
    - Guarantee termination.
  - Type inference undeciable for the rich types in proof assistants. (Coq vs. Agda).