barkhausen institut

# Foundations of Certified Programming Language and Compiler Design

**Dr.-Ing. Sebastian Ertel**

Composable Operating Systems Group, Barkhausen Institute

# Announcements

- Let's reflect on our current knowledge.
- Please prepare presentations:
  - Reasoning in Holbert vs reasoning in Coq
  - Reasoning in Holbert vs reasoning in Lean
  - Take assignments 1 and 2 as a foundation
- Date: 12.12.2023

# 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 |

## Main Goals

Now that we entered the world of compile-time verification, i.e., type systems, let’s

- introduce types to our rigorous mathematical foundation of programming: the lambda calculus and
- observe what they can enforce.

# Recap: The untyped lambda calculus

*Syntax:*

$$
\begin{array}{lllll}
t & ::= & & \text{terms:} \\
 & | & x & \text{variable} \\
 & | & \lambda x.t & \text{abstraction} \\
 & | & t\ t & \text{application}
\end{array}
\qquad
\begin{array}{lllll}
v & ::= & & \text{values:} \\
 & | & \lambda x.t & \text{abstraction value}
\end{array}
$$

*Semantics:*

$\boxed{t \longrightarrow t'}$

$$
\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}\ \text{E-APP1}
\qquad
\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'}\ \text{E-APP2}
\qquad
\frac{}{(\lambda x.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12}}\ \text{E-APPABS}
$$

# Function Types

Desired goal: extension of our type system with types for functions.

# Function Types

Desired goal: extension of our type system with types for functions.

A first approach: Let's add a type for functions: $\lambda x.t : \rightarrow$.

# Function Types

Desired goal: extension of our type system with types for functions.

A first approach: Let's add a type for functions: $\lambda x.t :\to$.

- For our (arithmetic + booleans) example language, we would have:

# Function Types

Desired goal: extension of our type system with types for functions.

A first approach: Let's add a type for functions: $\lambda x.t : \rightarrow$.

- For our (arithmetic + booleans) example language, we would have:
  - $\lambda x.x : \rightarrow$ and

# Function Types

Desired goal: extension of our type system with types for functions.

A first approach: Let's add a type for functions: $\lambda x.t :\rightarrow$.

- For our (arithmetic + booleans) example language, we would have:
  - $\lambda x.x :\rightarrow$ and
  - $\texttt{if true then } (\lambda x.\texttt{true}) \texttt{ then } (\lambda x.\lambda y.y) :\rightarrow$.

# Function Types

Desired goal: extension of our type system with types for functions.

A first approach: Let's add a type for functions: $\lambda x.t :\rightarrow$.

- For our (arithmetic + booleans) example language, we would have:
  - $\lambda x.x :\rightarrow$ and
  - $\texttt{if true then } (\lambda x.\texttt{true}) \texttt{ then } (\lambda x.\lambda y.y) :\rightarrow$.

Consider the following abstractions: $\lambda x.\texttt{true} :\rightarrow$ and $\lambda x.\lambda y.y :\rightarrow$

# Function Types

Desired goal: extension of our type system with types for functions.

A first approach: Let's add a type for functions: $\lambda x.t :\rightarrow$.

- For our (arithmetic + booleans) example language, we would have:
  - $\lambda x.x :\rightarrow$ and
  - $\texttt{if true then } (\lambda x.\texttt{true}) \texttt{ then } (\lambda x.\lambda y.y) :\rightarrow$.

Consider the following abstractions: $\lambda x.\texttt{true} :\rightarrow$ and $\lambda x.\lambda y.y :\rightarrow$

What is the result of an application???

# Function Types

Desired goal: extension of our type system with types for functions.

A first approach: Let's add a type for functions: $\lambda x.t :\rightarrow$.

- For our (arithmetic + booleans) example language, we would have:
  - $\lambda x.x :\rightarrow$ and
  - $\texttt{if true then } (\lambda x.\texttt{true}) \texttt{ then } (\lambda x.\lambda y.y) :\rightarrow$.

Consider the following abstractions: $\lambda x.\texttt{true} :\rightarrow$ and $\lambda x.\lambda y.y :\rightarrow$

What is the result of an application???

Consider the following applications: $(\lambda x.\texttt{true})\ \texttt{false} :$ vs. $(\lambda x.\lambda y.y)\ \texttt{false}$

# Function Types

Desired goal: extension of our type system with types for functions.

A first approach: Let's add a type for functions: $\lambda x.t :\rightarrow$.

- For our (arithmetic + booleans) example language, we would have:
  - $\lambda x.x :\rightarrow$ and
  - $\texttt{if true then } (\lambda x.\texttt{true}) \texttt{ then } (\lambda x.\lambda y.y) :\rightarrow$.

Consider the following abstractions: $\lambda x.\texttt{true} :\rightarrow$ and $\lambda x.\lambda y.y :\rightarrow$

What is the result of an application???

Consider the following applications: $(\lambda x.\texttt{true})\ \texttt{false} :$ vs. $(\lambda x.\lambda y.y)\ \texttt{false}$

How can I "derive" the type $\texttt{Bool}$ for the first and $\rightarrow$ for the second?

# Function Types

Desired goal: extension of our type system with types for functions.

A first approach: Let's add a type for functions: $\lambda x.t :\rightarrow$.

- For our (arithmetic + booleans) example language, we would have:
  - $\lambda x.x :\rightarrow$ and
  - $\texttt{if true then } (\lambda x.\texttt{true}) \texttt{ then } (\lambda x.\lambda y.y) :\rightarrow$.

Consider the following abstractions: $\lambda x.\texttt{true} :\rightarrow$ and $\lambda x.\lambda y.y :\rightarrow$

What is the result of an application???

Consider the following applications: $(\lambda x.\texttt{true})\ \texttt{false} :$ vs. $(\lambda x.\lambda y.y)\ \texttt{false}$

How can I "derive" the type `Bool` for the first and $\rightarrow$ for the second?

In order to make a proper statement about the type of an application, we need to be more precise.

# Function Types

Desired goal: extension of our type system with types for functions.

A precise approach: We capture all information by defining an *infinite family of types*: $T_1 \to T_2$.

## Definition (Simple Types)

The set of *simple types* over the type `Bool` is generated by the following syntax:

$$
\begin{array}{lcll}
T & ::= & & \text{types:} \\
  & | & \texttt{Bool} & \text{type of booleans} \\
  & | & T \to T & \text{type of functions}
\end{array}
$$

The *type constructor* $\to$ is right-associative, i.e., $T_1 \to T_2 \to T_3$ stands for $T_1 \to (T_2 \to T_3)$.

# The Typing Context

## Challenge

$$\frac{t : T_2}{\lambda x.t : \textbf{???} \rightarrow T_2}\ \text{T-ABS}$$

# The Typing Context

## Challenge

$$\frac{t : T_2}{\lambda x.t : \textbf{???} \rightarrow T_2} \text{ T-ABS}$$

**Implicitly** typed languages compute this type via a type inference algorithm as part of the type checker. (Haskell)

# The Typing Context

## Challenge

$$\frac{t : T_2}{\lambda x.t : ??? \to T_2}\ \textsc{T-Abs}$$

**Implicitly** typed languages compute this type via a type inference algorithm as part of the type checker. (Haskell)

**Explicitly** typed languages request type annotations for variables from the developer. (In Coq, type inference is undeciable.)

$$\frac{t : T_2}{\lambda(x : T_1).t : T_1 \to T_2}\ \textsc{T-Abs} \xrightarrow{\text{assumptions}} \frac{x : T_1 \vdash t : T_2}{\vdash \lambda(x : T_1).t : T_1 \to T_2}\ \textsc{T-Abs}$$

# The Typing Context

## Challenge

$$\frac{t : T_2}{\lambda x.t : \text{???} \rightarrow T_2}\ \textsc{T-Abs}$$

**Implicitly** typed languages compute this type via a type inference algorithm as part of the type checker. (Haskell)

**Explicitly** typed languages request type annotations for variables from the developer. (In Coq, type inference is undeciable.)

$$\frac{t : T_2}{\lambda(x : T_1).t : T_1 \rightarrow T_2}\ \textsc{T-Abs} \xrightarrow{\text{assumptions}} \frac{x : T_1 \vdash t : T_2}{\vdash \lambda(x : T_1).t : T_1 \rightarrow T_2}\ \textsc{T-Abs}$$

**The typing context $\Gamma$** tracks the assumptions about the types of free variables.

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda(x : T_1).t : T_1 \rightarrow T_2}\ \textsc{T-Abs}$$

$$\boxed{t : T} \xrightarrow{\text{assumptions}} \boxed{\Gamma \vdash t : T}$$

## The Typing Context, Formally

- $\Gamma$ is a sequence of variable names and their according types, i.e., a function $\Gamma$ : Symbol $\rightarrow T$.
- "," adds a new binding.
- Ø is the empty context and can be omitted as $\vdash t : T$.
- $dom(\Gamma)$ is the set of variables bound by $\Gamma$.

# The simply typed lambda calculus

*Syntax:*

# The simply typed lambda calculus

*Syntax:*

$$
\begin{array}{lcll}
t & ::= & & \text{terms:} \\
  & |   & x & \text{variable} \\
  & |   & \lambda x : T . t & \text{abstraction} \\
  & |   & t \; t & \text{application} \\
v & ::= & & \text{values:} \\
  & |   & \lambda x : T . t & \text{abstraction value}
\end{array}
$$

# The simply typed lambda calculus

*Syntax:*

$$
\begin{array}{lllll}
t & ::= & & \text{terms:} \\
 & | & x & \text{variable} \\
 & | & \lambda x{:}T.t & \text{abstraction} \\
 & | & t\ t & \text{application} \\
v & ::= & & \text{values:} \\
 & | & \lambda x{:}T.t & \text{abstraction value}
\end{array}
$$

$$
\begin{array}{llll}
T & ::= & & \text{types:} \\
 & | & T \rightarrow T & \text{type of functions} \\
\Gamma & ::= & & \text{contexts:} \\
 & | & \emptyset & \text{empty context} \\
 & | & \Gamma, x : T & \text{term variable binding}
\end{array}
$$

# The simply typed lambda calculus

*Syntax:*

$$
\begin{array}{llll}
t & ::= & & \text{terms:} \\
 & | & x & \text{variable} \\
 & | & \lambda x : T . t & \text{abstraction} \\
 & | & t\ t & \text{application} \\
v & ::= & & \text{values:} \\
 & | & \lambda x : T . t & \text{abstraction value}
\end{array}
$$

$$
\begin{array}{llll}
T & ::= & & \text{types:} \\
 & | & T \to T & \text{type of functions} \\
\Gamma & ::= & & \text{contexts:} \\
 & | & \emptyset & \text{empty context} \\
 & | & \Gamma, x : T & \text{term variable binding}
\end{array}
$$

*Semantics:*

# The simply typed lambda calculus

*Syntax:*

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

| | | | |
|---|---|---|---|
| $T$ | $::=$ | | types: |
| | $\mid$ | $T \to T$ | type of functions |
| $\Gamma$ | $::=$ | | contexts: |
| | $\mid$ | $\varnothing$ | empty context |
| | $\mid$ | $\Gamma, x : T$ | term variable binding |

*Semantics:*

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

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

# The simply typed lambda calculus

*Syntax:*

$$
\begin{array}{lcll}
t & ::= & & \text{terms:} \\
 & \mid & x & \text{variable} \\
 & \mid & \lambda x{:}T.t & \text{abstraction} \\
 & \mid & t\ t & \text{application} \\
v & ::= & & \text{values:} \\
 & \mid & \lambda x{:}T.t & \text{abstraction value}
\end{array}
$$

$$
\begin{array}{lcll}
T & ::= & & \text{types:} \\
 & \mid & T \rightarrow T & \text{type of functions} \\
\Gamma & ::= & & \text{contexts:} \\
 & \mid & \emptyset & \text{empty context} \\
 & \mid & \Gamma, x : T & \text{term variable binding}
\end{array}
$$

*Semantics:*

$\boxed{t \longrightarrow t'}$

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

# The simply typed lambda calculus

*Syntax:*

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

| | | | |
|---|---|---|---|
| $T$ | $::=$ | | types: |
| | $\mid$ | $T \to T$ | type of functions |
| $\Gamma$ | $::=$ | | contexts: |
| | $\mid$ | $\emptyset$ | empty context |
| | $\mid$ | $\Gamma, x : T$ | term variable binding |

*Semantics:*

$\boxed{t \longrightarrow t'}$

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

# The simply typed lambda calculus

*Syntax:*

$$
\begin{array}{llll}
t & ::= & & \text{terms:} \\
 & | & x & \text{variable} \\
 & | & \lambda x{:}T.t & \text{abstraction} \\
 & | & t\ t & \text{application} \\
v & ::= & & \text{values:} \\
 & | & \lambda x{:}T.t & \text{abstraction value}
\end{array}
$$

$$
\begin{array}{llll}
T & ::= & & \text{types:} \\
 & | & T \to T & \text{type of functions} \\
\Gamma & ::= & & \text{contexts:} \\
 & | & \emptyset & \text{empty context} \\
 & | & \Gamma, x : T & \text{term variable binding}
\end{array}
$$

*Typing*

# The simply typed lambda calculus

*Syntax:*

$$
\begin{array}{llll}
t & ::= & & \text{terms:} \\
  & | & x & \text{variable} \\
  & | & \lambda x{:}T.t & \text{abstraction} \\
  & | & t\ t & \text{application} \\
v & ::= & & \text{values:} \\
  & | & \lambda x{:}T.t & \text{abstraction value}
\end{array}
$$

$$
\begin{array}{llll}
T & ::= & & \text{types:} \\
  & | & T \to T & \text{type of functions} \\
\Gamma & ::= & & \text{contexts:} \\
  & | & \varnothing & \text{empty context} \\
  & | & \Gamma, x : T & \text{term variable binding}
\end{array}
$$

*Typing*

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

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

# The simply typed lambda calculus

*Syntax:*

$$\begin{array}{llll} t & ::= & & \text{terms:} \\ & | & x & \text{variable} \\ & | & \lambda x : T.t & \text{abstraction} \\ & | & t\ t & \text{application} \\ v & ::= & & \text{values:} \\ & | & \lambda x : T.t & \text{abstraction value} \end{array}$$

$$\begin{array}{llll} T & ::= & & \text{types:} \\ & | & T \to T & \text{type of functions} \\ \Gamma & ::= & & \text{contexts:} \\ & | & \emptyset & \text{empty context} \\ & | & \Gamma, x : T & \text{term variable binding} \end{array}$$

*Typing*

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

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

# The simply typed lambda calculus

*Syntax:*

$$\begin{array}{llll} t & ::= & & \text{terms:} \\ & | & x & \text{variable} \\ & | & \lambda x : T.t & \text{abstraction} \\ & | & t\ t & \text{application} \\ v & ::= & & \text{values:} \\ & | & \lambda x : T.t & \text{abstraction value} \end{array}$$

$$\begin{array}{llll} T & ::= & & \text{types:} \\ & | & T \to T & \text{type of functions} \\ \Gamma & ::= & & \text{contexts:} \\ & | & \emptyset & \text{empty context} \\ & | & \Gamma, x : T & \text{term variable binding} \end{array}$$

*Typing*

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

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

# Derivation Trees

- Typing derviations:

# Derivation Trees

- Typing derviations:

$$\dfrac{\dfrac{\dfrac{x : \texttt{Bool} \in x : \texttt{Bool}}{x : \texttt{Bool} \vdash x : \texttt{Bool}}\ \textsc{T-Var}}{\lambda x : \texttt{Bool}.x : \texttt{Bool} \to \texttt{Bool}}\ \textsc{T-Abs} \qquad \dfrac{}{\vdash \texttt{true} : \texttt{Bool}}\ \textsc{T-True}}{\vdash (\lambda x : \texttt{Bool}.x)\ \texttt{true} : \texttt{Bool}}\ \textsc{T-App}$$

# Derivation Trees

- Typing derviations:

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

- Check: Show the derivation tree for
  $f : \texttt{Bool} \to \texttt{Bool} \vdash f\ (\texttt{if false then true else false}) : \texttt{Bool}$

# Derivation Trees

- Typing derviations:

$$\dfrac{\dfrac{\dfrac{x : \texttt{Bool} \in x : \texttt{Bool}}{x : \texttt{Bool} \vdash x : \texttt{Bool}}\ \textsc{T-Var}}{\lambda x : \texttt{Bool}.x : \texttt{Bool} \to \texttt{Bool}}\ \textsc{T-Abs} \qquad \dfrac{}{\vdash \texttt{true} : \texttt{Bool}}\ \textsc{T-True}}{\vdash (\lambda x : \texttt{Bool}.x)\ \texttt{true} : \texttt{Bool}}\ \textsc{T-App}$$

- Check: Show the derivation tree for
$f : \texttt{Bool} \to \texttt{Bool} \vdash f\ (\texttt{if false then true else false}) : \texttt{Bool}$

$$\dfrac{\dfrac{f : \texttt{Bool} \to \texttt{Bool} \in \Gamma}{\Gamma \vdash f : \texttt{Bool} \to \texttt{Bool}}\ \textsc{T-Var} \qquad \dfrac{\dfrac{}{\texttt{false} : \texttt{Bool}}\ \textsc{T-False} \quad \dfrac{}{\texttt{true} : \texttt{Bool}}\ \textsc{T-True} \quad \dfrac{}{\texttt{false} : \texttt{Bool}}\ \textsc{T-False}}{\Gamma \vdash \texttt{if false then true else false} : \texttt{Bool}}\ \textsc{T-If}}{f : \texttt{Bool} \to \texttt{Bool} \vdash f\ (\texttt{if false then true else false}) : \texttt{Bool}}\ \textsc{T-App}$$

# Derivation Trees

- Typing derviations:

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

- Check: Show the derivation tree for
$f : \texttt{Bool} \to \texttt{Bool} \vdash f\ (\texttt{if false then true else false}) : \texttt{Bool}$

$$\dfrac{\dfrac{f : \texttt{Bool} \to \texttt{Bool} \in \Gamma}{\Gamma \vdash f : \texttt{Bool} \to \texttt{Bool}}\,\text{T-Var} \qquad \dfrac{\dfrac{}{\texttt{false} : \texttt{Bool}}\,\text{T-False} \quad \dfrac{}{\texttt{true} : \texttt{Bool}}\,\text{T-True} \quad \dfrac{}{\texttt{false} : \texttt{Bool}}\,\text{T-False}}{\Gamma \vdash \texttt{if false then true else false} : \texttt{Bool}}\,\text{T-If}}{f : \texttt{Bool} \to \texttt{Bool} \vdash f\ (\texttt{if false then true else false}) : \texttt{Bool}}\,\text{T-App}$$

- Check: Find a context $\Gamma$ for $f\ x\ y : \texttt{Bool}$.

## Derivation Trees

- Typing derviations:

$$\dfrac{\dfrac{\dfrac{x : \texttt{Bool} \in x : \texttt{Bool}}{x : \texttt{Bool} \vdash x : \texttt{Bool}}\ \textsc{T-Var}}{\lambda x : \texttt{Bool}.x : \texttt{Bool} \to \texttt{Bool}}\ \textsc{T-Abs} \qquad \dfrac{}{\vdash \texttt{true} : \texttt{Bool}}\ \textsc{T-True}}{\vdash (\lambda x : \texttt{Bool}.x)\ \texttt{true} : \texttt{Bool}}\ \textsc{T-App}$$

- Check: Show the derivation tree for
  $f : \texttt{Bool} \to \texttt{Bool} \vdash f\ (\texttt{if false then true else false}) : \texttt{Bool}$

$$\dfrac{\dfrac{f : \texttt{Bool} \to \texttt{Bool} \in \Gamma}{\Gamma \vdash f : \texttt{Bool} \to \texttt{Bool}}\ \textsc{T-Var} \qquad \dfrac{\dfrac{}{\texttt{false} : \texttt{Bool}}\ \textsc{T-False} \quad \dfrac{}{\texttt{true} : \texttt{Bool}}\ \textsc{T-True} \quad \dfrac{}{\texttt{false} : \texttt{Bool}}\ \textsc{T-False}}{\Gamma \vdash \texttt{if false then true else false} : \texttt{Bool}}\ \textsc{T-If}}{f : \texttt{Bool} \to \texttt{Bool} \vdash f\ (\texttt{if false then true else false}) : \texttt{Bool}}\ \textsc{T-App}$$

- Check: Find a context $\Gamma$ for $f\ x\ y : \texttt{Bool}$.

$$\begin{array}{llll} \Gamma = & f : \texttt{Bool} \to \texttt{Bool} \to \texttt{Bool}, & f : \texttt{Nat} \to \texttt{Nat} \to \texttt{Bool}, & f : \texttt{T} \to \texttt{T} \to \texttt{Bool} \\ & x : \texttt{Bool}, y : \texttt{Bool} & x : \texttt{Nat}, y : \texttt{Nat} & x : \texttt{T}, y : \texttt{T} \end{array}$$