



Type Systems

Lecture 3 Nov. 3rd, 2004

Sebastian Maneth

<http://lampwww.epfl.ch/teaching/typeSystems/2004>

Today: ... into the **types** ...

1. A **Type System** for Arithmetic Expressions
2. Proving Type Safety
3. **Simply Typed Lambda Calculus**
4. Proving Type Safety
5. Conclusions

A Type System for Arithmetic Expressions

Expr ::= true | false | zero

Expr ::= if Expr then Expr else Expr

Expr ::= succ (Expr)

Expr ::= pred (Expr)

Expr ::= isZero (Expr)

Val ::= true | false | NVal

NVal ::= zero | succ NVal

“Stuck” terms: succ(true)
isZero(false)
if zero then true else false

Cannot rewrite, but are not values. → no semantics = **execution error**

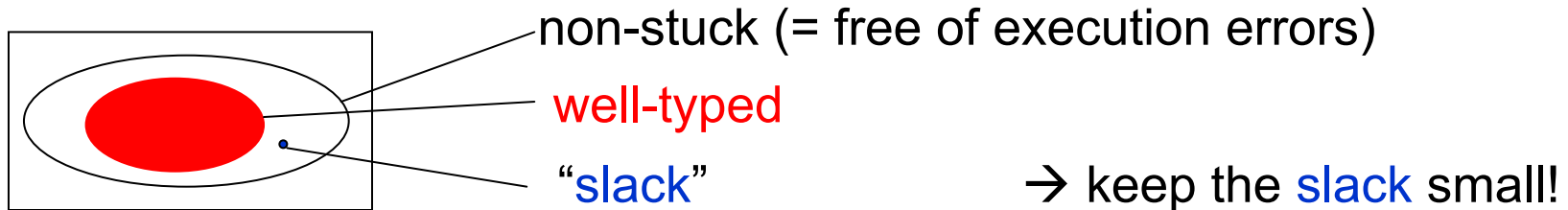
type sound = all well-typed programs are **free of execution errors**

→ find a **Type System** for Expr, so that well-typed terms do NOT get stuck!

A Type System for Arithmetic Expressions

→ find a **Type System** for Expr, so that well-typed terms do NOT get stuck!

The converse will NOT be true: `if true then zero else succ(true)`
is not stuck (evaluates to `zero`), but will not be well-typed!



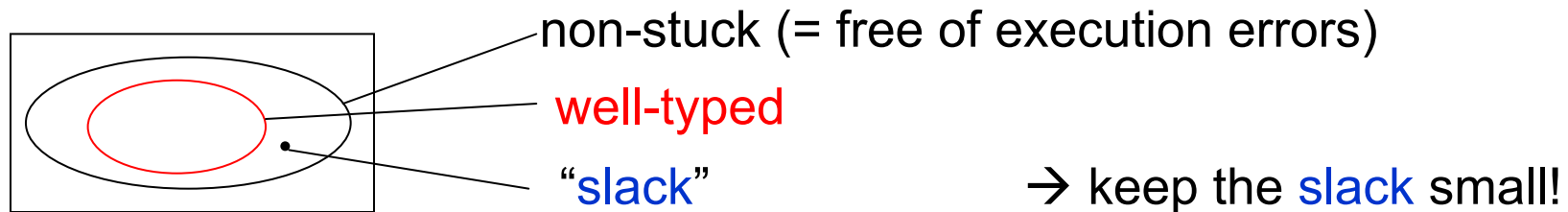
Introduce two types **Bool** and **Nat**, representing Booleans and Numbers.
Every Expr `t` will be of type **Bool** or **Nat**, or will have no type.

`t : Bool` = "t has type **Bool**"

A Type System for Arithmetic Expressions

→ find a **Type System** for Expr, so that well-typed terms do NOT get stuck!

The converse will NOT be true: `if true then zero else false`
is not stuck (evaluates to `zero`), but will not be well-typed!



Introduce two types **Bool** and **Nat**, representing Booleans and Numbers.
Every Expr `t` will be of type **Bool** or **Nat**, or will have no type.

`t : Bool` = "t has type **Bool**"

typing rules (**Type System**): `true : Bool` `false : Bool`

$$\frac{t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$$

A Type System for Arithmetic Expressions

typing rules:

$\text{true} : \text{Bool}$	$\text{false} : \text{Bool}$	$\frac{t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$
$\text{zero} : \text{Nat}$		
$\frac{t_1 : \text{Nat}}{\text{succ } t_1 : \text{Nat}}$	$\frac{t_1 : \text{Nat}}{\text{pred } t_1 : \text{Nat}}$	$\frac{t_1 : \text{Nat}}{\text{isZero } t_1 : \text{Bool}}$

Note: this type system is VERY simple.

→ it can be incorporated into the syntax definition (EBNF).

do you see how?

A Type System for Arithmetic Expressions

typing rules:

$\text{true} : \text{Bool}$	$\text{false} : \text{Bool}$	$\frac{t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$
$\text{zero} : \text{Nat}$		
$\frac{t_1 : \text{Nat}}{\text{succ } t_1 : \text{Nat}}$	$\frac{t_1 : \text{Nat}}{\text{pred } t_1 : \text{Nat}}$	$\frac{t_1 : \text{Nat}}{\text{isZero } t_1 : \text{Bool}}$

typing derivation for `if isZero zero then zero else pred zero`

$\frac{\text{zero} : \text{Nat}}{\text{isZero zero} : \text{Bool}}$	$\text{zero} : \text{Nat}$	$\frac{\text{zero} : \text{Nat}}{\text{pred zero} : \text{Nat}}$
$\text{if isZero zero then zero else pred zero} : \text{Nat}$		

A Type System for Arithmetic Expressions

How to find a typing derivation?

→ assume the Expr has some type R; then determine backwards the required types of the subexpressions, and check them!

1. If `true` : R or `false` : R, then R = `Bool`.
2. If `zero` : R, then R = `Nat`.

A Type System for Arithmetic Expressions

How to find a typing derivation?

→ assume the Expr has some type R; then determine backwards the required types of the subexpressions, and check them!

1. If `true` : R or `false` : R, then $R = \text{Bool}$.
2. If `zero` : R, then $R = \text{Nat}$.
3. If `if` t_1 `then` t_2 `else` t_3 : R, then $t_1 : \text{Bool}$, $t_2 : R$, and $t_3 : R$
4. If `succ` t_1 : R or `pred` t_1 : R, then $R = \text{Nat}$
5. If `isZero` t_1 : R, then $R = \text{Bool}$ and $t_1 : \text{Nat}$

A Type System for Arithmetic Expressions

How to find a typing derivation?

→ assume the Expr has some type R; then determine backwards the required types of the subexpressions, and check them!

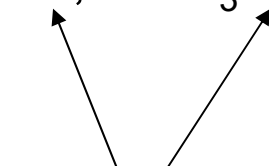
1. If `true` : R or `false` : R, then R = **Bool**.

2. If `zero` : R, then R = **Nat**.

3. If `if` t_1 `then` t_2 `else` t_3 : R, then t_1 : **Bool**, t_2 : R, and t_3 : R

4. If `succ` t_1 : R or `pred` t_1 : R, then R = **Nat**

5. If `isZero` t_1 : R, then R = **Bool** and t_1 : **Nat**



must be the *same* R!!

A Type System for Arithmetic Expressions

How to find a typing derivation?

→ assume the Expr has some type R; then determine backwards the required types of the subexpressions, and check them!

INVERSION LEMMA

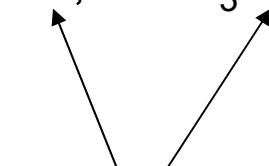
1. If $\text{true} : R$ or $\text{false} : R$, then $R = \text{Bool}$.

2. If $\text{zero} : R$, then $R = \text{Nat}$.

3. If $\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : R$, then $t_1 : \text{Bool}$, $t_2 : R$, and $t_3 : R$

4. If $\text{succ } t_1 : R$ or $\text{pred } t_1 : R$, then $R = \text{Nat}$

5. If $\text{isZero } t_1 : R$, then $R = \text{Bool}$ and $t_1 : \text{Nat}$



must be the same R!!

Theorem: Every term has at most one type (with unique derivation).

Proof by induction, using **INV.L.**

What you will learn in this course:

- how to **define** a type system **T** (to allow for unambiguous implementations)
- how to formally **prove** that (\mathbf{P}, \mathbf{T}) is type sound
- how to **implement** a typechecker for **T**

What you will learn in this course:

- how to **define** a type system **T** (to allow for unambiguous implementations)
- how to formally **prove** that (\mathbf{P}, \mathbf{T}) is **type sound**
= type safe
- how to **implement** a typechecker for **T**

Proving Type Safety

“well-typed terms do not go wrong”

Safety = **Progress** + **Preservation**

Progress = A well-typed term is NOT stuck

Preservation = evaluation preserves well-typedness

well-typed $\xrightarrow{\text{Progress}}$ NOT stuck \rightarrow either value or

Progress

we can evaluate $\xrightarrow{\text{Preserve}}$ result is well-typed

Preserve

Proving Type Safety

“well-typed terms do not go wrong”

Safety = **Progress** + **Preservation**

Progress = A well-typed term is NOT stuck

Preservation = evaluation preserves well-typedness

well-typed $\xrightarrow{\text{Progress}}$ NOT stuck \rightarrow either value or
we can evaluate $\xrightarrow{\text{Preserve}}$ result is well-typed

Proving Type Safety

Progress Theorem: If t is well-typed, then it is either a value or there exists a t' such that $t \rightarrow t'$.

Observations: (1) if $t : \text{Bool}$ is a value, then $t = \text{true}$ or $t = \text{false}$

(2) if $t : \text{Nat}$ is a value, then $t = \underbrace{\text{succ}(\dots \text{succ}(\text{zero}) \dots)}_{\geq 0}$

Proof. Induction on t .

$t = \text{true} \mid \text{false} \mid \text{zero} \rightarrow$ immediate.

$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : R$, then $t_1 : \text{Bool}$, $t_2 : R$, and $t_3 : R$ (INV.L.)

Proving Type Safety

Progress Theorem: If t is well-typed, then it is either a value or there exists a t' such that $t \rightarrow t'$.

Observations: (1) if $t : \text{Bool}$ is a value, then $t = \text{true}$ or $t = \text{false}$
(2) if $t : \text{Nat}$ is a value, then $t = \underbrace{\text{succ}(\dots \text{succ}(\text{zero}) \dots)}_{\geq 0}$

Proof. Induction on t .

$t = \text{true} \mid \text{false} \mid \text{zero} \rightarrow$ immediate.

$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : R$, then $t_1 : \text{Bool}$, $t_2 : R$, and $t_3 : R$ (INV.L.)

- t_1 is value. By (1), $t = \text{true}$ or $t = \text{false}$.

Thus, t can evaluate to a t' ($= t_2$ or t_3)!

Proving Type Safety

Progress Theorem: If t is well-typed, then it is either a value or there exists a t' such that $t \rightarrow t'$.

Observations: (1) if $t : \text{Bool}$ is a value, then $t = \text{true}$ or $t = \text{false}$
(2) if $t : \text{Nat}$ is a value, then $t = \underbrace{\text{succ}(\dots \text{succ}(\text{zero}) \dots)}_{\geq 0}$

Proof. Induction on t .

$t = \text{true} \mid \text{false} \mid \text{zero} \rightarrow$ immediate.

$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : R$, then $t_1 : \text{Bool}$, $t_2 : R$, and $t_3 : R$ (**INV.L.**)

- t_1 is value. By (1), $t = \text{true}$ or $t = \text{false}$.

Thus, t can evaluate to a t' ($= t_2$ or t_3)!

- t_1 is NOT value. By induction $\exists t_1'$ with $t_1 \rightarrow t_1'$.

Thus, t can evaluate to a t' ($= \text{if } t_1' \text{ then } \dots$)!

Proving Type Safety

Progress Theorem: If t is well-typed, then it is either a value or there exists a t' such that $t \rightarrow t'$.

Observations: (1) if $t : \text{Bool}$ is a value, then $t = \text{true}$ or $t = \text{false}$
(2) if $t : \text{Nat}$ is a value, then $t = \underbrace{\text{succ}(\dots \text{succ}(\text{zero}) \dots)}_{\geq 0}$

Proof. Induction on t .

$t = \text{true} \mid \text{false} \mid \text{zero} \rightarrow$ immediate.

$t = \text{succ } t_1$. By induction, t_1 is value or $t_1 \rightarrow t_1'$. By **INV.L.**, $t_1 : \text{Nat}$.

- t_1 is value. By (2), $t_1 = \text{succ}(\dots \text{zero} \dots)$. Hence, t is also a value!
- t_1 is NOT value. Then t can evaluate to a t' ($= \text{succ } t_1'$)

Proving Type Safety

Progress Theorem: If t is well-typed, then it is either a value or there exists a t' such that $t \rightarrow t'$.

Observations: (1) if $t : \text{Bool}$ is a value, then $t = \text{true}$ or $t = \text{false}$
(2) if $t : \text{Nat}$ is a value, then $t = \underbrace{\text{succ}(\dots \text{succ}(\text{zero}) \dots)}_{\geq 0}$

Proof. Induction on t .

$t = \text{true} \mid \text{false} \mid \text{zero} \rightarrow$ immediate.

$t = \text{pred } t_1$. By induction, t_1 is value or $t_1 \rightarrow t_1'$. By **INV.L.**, $t_1 : \text{Nat}$.

- t_1 is value. By (2), $t_1 = \text{succ}(\dots \text{zero} \dots)$. Thus, t can evaluate!
- t_1 is NOT value. Then t can evaluate to a t' ($= \text{pred } t_1'$)

Proving Type Safety

Progress Theorem: If t is well-typed, then it is either a value or there exists a t' such that $t \rightarrow t'$.

Observations: (1) if $t : \text{Bool}$ is a value, then $t = \text{true}$ or $t = \text{false}$
(2) if $t : \text{Nat}$ is a value, then $t = \underbrace{\text{succ}(\dots \text{succ}(\text{zero}) \dots)}_{\geq 0}$

Proof. Induction on t .

$t = \text{true} \mid \text{false} \mid \text{zero} \rightarrow$ immediate.

$t = \text{isZero } t_1$. By induction, t_1 is value or $t_1 \rightarrow t_1'$. By **INV.L.**, $t_1 : \text{Nat}$.

- t_1 is value. By (2), $t_1 = \text{succ}(\dots \text{zero} \dots)$. Thus, t can evaluate!
- t_1 is NOT value. Then t can evaluate to a t' ($= \text{isZero } t_1'$)

Proving Type Safety

Preservation Theorem: If $t : T$ and $t \rightarrow t'$, then $t' : T$.

$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : R$, then $t_1 : \text{Bool}$, $t_2 : R$, and $t_3 : R$ (INV.L.)

$t' = t_2 \mid t_3 \mid \text{if } t_1' \text{ then } t_2 \text{ else } t_3$, where $t_1 \rightarrow t_1'$

Proving Type Safety

Preservation Theorem: If $t : T$ and $t \rightarrow t'$, then $t' : T$.

$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : R$, then $t_1 : \text{Bool}$, $t_2 : R$, and $t_3 : R$ (INV.L.)

$t' = t_2 \mid t_3 \mid \text{if } t_1' \text{ then } t_2 \text{ else } t_3$, where $t_1 \rightarrow t_1'$
:R :R

By induction, $t_1' : \text{Bool}$. THUS, $t' : R$.

Proving Type Safety

Preservation Theorem: If $t : T$ and $t \rightarrow t'$, then $t' : T$.

$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : R$, then $t_1 : \text{Bool}$, $t_2 : R$, and $t_3 : R$ (INV.L.)

$t' = t_2 \mid t_3 \mid \text{if } t_1' \text{ then } t_2 \text{ else } t_3$, where $t_1 \rightarrow t_1'$
:R :R

By induction, $t_1' : \text{Bool}$. THUS, $t' : R$.

$t = \text{succ } t_1$. Thus, $\text{succ } t_1 \rightarrow \text{succ } t_1'$ and $t_1 \rightarrow t_1'$. By INV.L., $t_1 : \text{Nat}$.

Proving Type Safety

Preservation Theorem: If $t : T$ and $t \rightarrow t'$, then $t' : T$.

$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : R$, then $t_1 : \text{Bool}$, $t_2 : R$, and $t_3 : R$ (INV.L.)

$t' = t_2 \mid t_3 \mid \text{if } t_1' \text{ then } t_2 \text{ else } t_3$, where $t_1 \rightarrow t_1'$
:R :R

By induction, $t_1' : \text{Bool}$. THUS, $t' : R$.

$t = \text{succ } t_1$. Thus, $\text{succ } t_1 \rightarrow \text{succ } t_1'$ and $t_1 \rightarrow t_1'$. By INV.L., $t_1 : \text{Nat}$.

By induction, $t_1' : \text{Nat}$. THUS, also $\text{succ } t_1' : \text{Nat}$.

Proving Type Safety

Preservation Theorem: If $t : T$ and $t \rightarrow t'$, then $t' : T$.

$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : R$, then $t_1 : \text{Bool}$, $t_2 : R$, and $t_3 : R$ (INV.L.)

$t' = t_2 \mid t_3 \mid \text{if } t_1' \text{ then } t_2 \text{ else } t_3$, where $t_1 \rightarrow t_1'$
: R : R

By induction, $t_1' : \text{Bool}$. THUS, $t' : R$.

$t = \text{succ } t_1$. Thus, $\text{succ } t_1 \rightarrow \text{succ } t_1'$ and $t_1 \rightarrow t_1'$. By INV.L., $t_1 : \text{Nat}$.

By induction, $t_1' : \text{Nat}$. THUS, also $\text{succ } t_1' : \text{Nat}$.

Cases $t = \text{pred } t_1 \mid \text{isZero } t_1$

Try yourself!!

Simply Typed Lambda Calculus

Imagine the small language λ -Bool, consisting of lambda terms together with Boolean primitives.

→ How to define a **Type System** that is **safe** (= “well-typed programs do not go wrong”)

i.e., we need **typing rules** for variables, abstraction, application, in such a way that we can prove **Progress** and **Preservation**.

... and in such a way that the “**slack**” is small! ...

BUT, lambda calculus is Turing complete → nontrivial properties canNOT be decided!!! (Rice’s Theorem)

if <long and tricky computation> **then true else** $(\lambda x. x)$

Simply Typed Lambda Calculus

The **set of simple types** over **Bool** is the smallest set T such that

1. **Bool** $\in T$

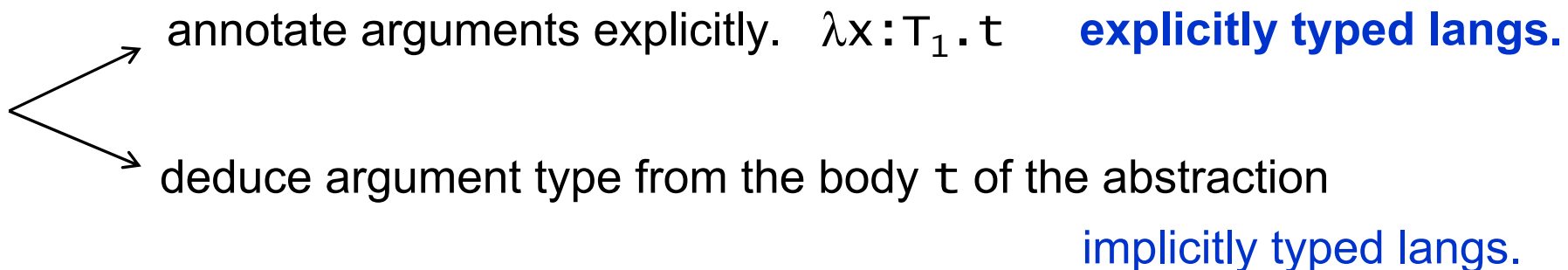
2. if $R_1, R_2 \in T$, then $R_1 \rightarrow R_2 \in T$

\rightarrow binds to the right. Thus, $R_1 \rightarrow R_2 \rightarrow R_3$ means $R_1 \rightarrow (R_2 \rightarrow R_3)$.

How to type $\lambda x. t$?

= what happens when t is applied to an argument?

But, **what type of arguments to expect??**



Simply Typed Lambda Calculus

We do **explicitly typed langs!** Syntax change: $\lambda x:T_1. t$

determines a type environment for t

Type Environment $\Gamma = \{ (x_1, T_1), \dots, (x_n, T_n) \}$ (finite function $\text{var} \rightarrow \text{Types}$)

typing rule for lambda abstraction:

$A \vdash B$ = under the assumption A , B holds

$$\frac{\Gamma, x:T_1 \vdash t:T_2}{\Gamma \vdash \lambda x:T_1. t : T_1 \rightarrow T_2}$$

Simply Typed Lambda Calculus

We do **explicitly typed langs!** Syntax change: $\lambda x:T_1. t$

determines a type environment for t

Type Environment $\Gamma = \{ (x_1, T_1), \dots, (x_n, T_n) \}$ (finite function $\text{var} \rightarrow \text{Types}$)

typing rule for lambda abstraction:

$A \vdash B$ = under the assumption A , B holds

$$\frac{\Gamma, x:T_1 \vdash t:T_2}{\Gamma \vdash \lambda x:T_1. t : T_1 \rightarrow T_2}$$

“making the assumption $x:T_1$ explicit”

Note: renaming of x in t is needed if x appears in Γ !

Simply Typed Lambda Calculus

$$\frac{\Gamma, x:T_1 \vdash t:T_2}{\Gamma \vdash \lambda x:T_1. t : T_1 \rightarrow T_2} \quad \text{lambda abstraction}$$

$$\frac{\Gamma \vdash t_1:T \rightarrow R \quad \Gamma \vdash t_2:T}{\Gamma \vdash t_1 t_2 : R} \quad \text{function application}$$

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

a derivation tree:

$$\vdash (\lambda x:\text{Bool}. x) \text{ true} : \text{Bool}$$

Simply Typed Lambda Calculus

$$\frac{\Gamma, x:T_1 \vdash t:T_2}{\Gamma \vdash \lambda x:T_1. t : T_1 \rightarrow T_2} \quad \text{lambda abstraction}$$

$$\frac{\Gamma \vdash t_1:T \rightarrow R \quad \Gamma \vdash t_2:T}{\Gamma \vdash t_1 t_2 : R} \quad \text{function application}$$

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

a derivation tree:

$$\frac{\vdash \lambda x:\text{Bool}. x : \text{Bool} \rightarrow \text{Bool} \quad \vdash \text{true}:\text{Bool}}{\vdash (\lambda x:\text{Bool}. x) \text{true} : \text{Bool}} \quad \text{application}$$

Simply Typed Lambda Calculus

$$\frac{\Gamma, x:T_1 \vdash t:T_2}{\Gamma \vdash \lambda x:T_1. t : T_1 \rightarrow T_2} \quad \text{lambda abstraction}$$

$$\frac{\Gamma \vdash t_1:T \rightarrow R \quad \Gamma \vdash t_2:T}{\Gamma \vdash t_1 t_2 : R} \quad \text{function application}$$

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

a derivation tree:

$$\text{abstraction } \frac{x : \text{Bool} \vdash x : \text{Bool}}{\Gamma \vdash \lambda x:\text{Bool}. x : \text{Bool} \rightarrow \text{Bool}} \quad \text{application } \frac{\Gamma \vdash \lambda x:\text{Bool}. x : \text{Bool} \rightarrow \text{Bool} \quad \Gamma \vdash \text{true}:\text{Bool}}{\Gamma \vdash (\lambda x:\text{Bool}. x) \text{ true} : \text{Bool}}$$

Simply Typed Lambda Calculus

$$\frac{\Gamma, x:T_1 \vdash t:T_2}{\Gamma \vdash \lambda x:T_1. t : T_1 \rightarrow T_2} \quad \text{lambda abstraction}$$

$$\frac{\Gamma \vdash t_1:T \rightarrow R \quad \Gamma \vdash t_2:T}{\Gamma \vdash t_1 t_2 : R} \quad \text{function application}$$

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

a derivation tree:

$$\text{abstraction} \frac{\frac{x : \text{Bool} \in x : \text{Bool}}{x : \text{Bool} \vdash x : \text{Bool}}}{\Gamma \vdash \lambda x:\text{Bool}. x : \text{Bool} \rightarrow \text{Bool}} \quad \Gamma \vdash \text{true}:\text{Bool}}{\Gamma \vdash (\lambda x:\text{Bool}. x) \text{ true} : \text{Bool}} \quad \text{application}$$

Proving Type Safety

Theorem: Every term has at most one type (with unique derivation).

- ⊥ 1. If $\Gamma \vdash x : R$, then $x:R \in \Gamma$.
- ⊃ 2. If $\Gamma \vdash \lambda x:T_1. t : R$, then $R = T_1 \rightarrow R_2$ for some R_2 with $\Gamma, x:T_1 \vdash t:R_2$.
- ⊆ 3. If $\Gamma \vdash t_1 t_2 : R$, then $\exists T$ s.t. $\Gamma \vdash t_1: T \rightarrow R$ and $\Gamma \vdash t_2 : T$.

Observation (3) If v is a value of type $T_1 \rightarrow T_2$, then $v = \lambda x: T_1. t_2$.

Progress Theorem: If t is closed and well-typed, then it is either a value or there exists a t' such that $t \rightarrow t'$.

Proof. $t = \text{true} \mid \text{false} \mid \text{if} ..$ like before!

$t = \lambda x:T_1. t_1$ is a value!

Proving Type Safety

Theorem: Every term has at most one type (with unique derivation).

- ⌋ 1. If $\Gamma \vdash x : R$, then $x:R \in \Gamma$.
- ⌋ 2. If $\Gamma \vdash \lambda x:T_1. t : R$, then $R = T_1 \rightarrow R_2$ for some R_2 with $\Gamma, x:T_1 \vdash t:R_2$.
- ⌋ 3. If $\Gamma \vdash t_1 t_2 : R$, then $\exists T$ s.t. $\Gamma \vdash t_1 : T \rightarrow R$ and $\Gamma \vdash t_2 : T$.

Observation (3) If v is a value of type $T_1 \rightarrow T_2$, then $v = \lambda x: T_1. t_2$.

Progress Theorem: If t is closed and well-typed, then it is either a value or there exists a t' such that $t \rightarrow t'$.

Proof. $t = \text{true} \mid \text{false} \mid \text{if} ..$ like before!

$t = \lambda x:T_1. t_1$ is a value!

$t = t_1 t_2 : R$, then $\exists T$ s.t. $\vdash t_1 : T \rightarrow R$ and $\vdash t_2 : T$.

by induction for t_1 and t_2 : either a value or can take a step.

If $t_1 \rightarrow t_1'$ then $t \rightarrow t'$ ($= t_1' t_2$)

If t_1 value and $t_2 \rightarrow t_2'$ then $t \rightarrow t'$ ($= t_1 t_2'$)

If both are values, then t_1 is abstraction, so can be applied!

Proving Type Safety

Preservation of substitution:

If (1) $\Gamma \vdash s : S$
(2) $\Gamma, x:S \vdash t : T$ then $\Gamma \vdash [x \rightarrow s]t : T$

Proof.

induction on structure of t . 6 cases

1. $t = z$. If $z=x$ then $\Gamma, x:S \vdash x : T$ implies that $T=S$.
And $\Gamma \vdash s : S$ means that $\Gamma \vdash [x \rightarrow s]x : T$

If $z \neq x$ then $\Gamma, x:S \vdash z : T$ implies that $z:T \in \Gamma$.
Thus $\Gamma \vdash z : T$.

Proving Type Safety

Preservation of substitution:

If (1) $\Gamma \vdash s : S$
(2) $\Gamma, x:S \vdash t : T$ then $\Gamma \vdash [x \rightarrow s]t : T$

Proof.

induction on structure of t . 6 cases

2. $t = \lambda y:T_2. t_1$. By **INV.L.** $T = T_2 \rightarrow T_1$ and $\Gamma, y:T_2 \vdash t_1 : T_1$.

Since $x \notin \text{dom}(\Gamma)$ and $x \neq y$, weaken $\Gamma, y:T_2 \vdash t_1 : T_1$ to $\Gamma, y:T_2, x:S \vdash t_1 : T_1$

and weaken $\Gamma \vdash s : S$ to $\Gamma' \vdash s : S$

Proving Type Safety

Preservation of substitution:

If (1) $\Gamma \vdash s : S$
(2) $\Gamma, x:S \vdash t : T$ then $\Gamma \vdash [x \rightarrow s]t : T$

Proof.

induction on structure of t . 6 cases

2. $t = \lambda y:T_2. t_1$. By **INV.L.** $T = T_2 \rightarrow T_1$ and $\Gamma, y:T_2 \vdash t_1 : T_1$.

Since $x \notin \text{dom}(\Gamma)$ and $x \neq y$, weaken $\Gamma, y:T_2 \vdash t_1 : T_1$
to $\Gamma, y:T_2, x:S \vdash t_1 : T_1$
 Γ'

and weaken $\Gamma \vdash s : S$ to $\Gamma' \vdash s : S$

By induction, $\Gamma' \vdash [x \rightarrow s]t_1 : T_1$.

$\Gamma \vdash \lambda y:T_2. [x \rightarrow s]t_1 : T_2 \rightarrow T_1$ abstraction

Proving Type Safety

Preservation of substitution:

If (1) $\Gamma \vdash s : S$
(2) $\Gamma, x:S \vdash t : T$ then $\Gamma \vdash [x \rightarrow s]t : T$

Proof.

induction on structure of t . 6 cases

2. $t = \lambda y:T_2. t_1$. By **INV.L.** $T = T_2 \rightarrow T_1$ and $\Gamma, y:T_2 \vdash t_1 : T_1$.

Since $x \notin \text{dom}(\Gamma)$ and $x \neq y$, weaken $\Gamma, y:T_2 \vdash t_1 : T_1$
to $\Gamma, y:T_2, x:S \vdash t_1 : T_1$

and weaken $\Gamma \vdash s : S$ to $\Gamma' \vdash s : S$

By induction, $\Gamma' \vdash [x \rightarrow s]t_1 : T_1$.

$\Gamma \vdash \lambda y:T_2. [x \rightarrow s]t_1 : T_2 \rightarrow T_1$ abstraction

$= \Gamma \vdash [x \rightarrow s]t : T$

Proving Type Safety

Preservation of substitution:

If (1) $\Gamma \vdash s : S$
(2) $\Gamma, x:S \vdash t : T$ then $\Gamma \vdash [x \rightarrow s]t : T$

Proof.

induction on structure of t . 6 cases

3. $t = t_1 t_2$. By **INV.L.** $\Gamma, x:S \vdash t : T$ implies

$$\begin{array}{l} \Gamma, x:S \vdash t_1 : T_2 \rightarrow T_1 \\ \Gamma, x:S \vdash t_2 : T_2 \end{array} \quad \text{with } T = T_1$$

By induction (2x):

$$\frac{\begin{array}{l} \Gamma \vdash [x \rightarrow s]t_1 : T_2 \rightarrow T_1 \\ \Gamma \vdash [x \rightarrow s]t_2 : T_2 \end{array}}{\Gamma \vdash [x \rightarrow s]t_1 [x \rightarrow s]t_2 : T_1} \text{ application}$$

$$= \Gamma \vdash [x \rightarrow s]t : T$$

Proving Type Safety

Preservation of substitution:

If (1) $\Gamma \vdash s : S$
(2) $\Gamma, x:S \vdash t : T$ then $\Gamma \vdash [x \rightarrow s]t : T$

Proof.

induction on structure of t . 6 cases

4. $t = \text{true}$. By **INV.L.**, $T = \text{Bool}$.
 $[x \rightarrow s]t = \text{true}$, and $\Gamma \vdash \text{true} : \text{Bool} \quad (\forall \Gamma)$

5. $t = \text{false}$. Same thing.

6. $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$.

by INV.L.	$\Gamma, x:S \vdash t_1 : \text{Bool}$		$\Gamma, x:S \vdash [x \rightarrow s]t_1 : \text{Bool}$
	$\Gamma, x:S \vdash t_2 : T$	induct.	$\Gamma, x:S \vdash [x \rightarrow s]t_2 : T$
	$\Gamma, x:S \vdash t_3 : T$		$\Gamma, x:S \vdash [x \rightarrow s]t_3 : T$
<hr/>			
			$\Gamma \vdash [x \rightarrow s] \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T$

Proving Type Safety

Preservation. If $\Gamma \vdash t : T$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T$.

Proof. Induction on the structure of t .

$t = z \mid \lambda y:T_1. t_1 \mid \mathbf{true} \mid \mathbf{false}$ nothing to be done ($\nexists t'$)

$t = \mathbf{if} t_1 \mathbf{then} t_2 \mathbf{else} t_3$ exactly like before!

$t = t_1 t_2$. By **INV.L.** $\Gamma \vdash t : T$ implies that $T = T_1$, $\Gamma \vdash t_1 : T_2 \rightarrow T_1$
and $\Gamma \vdash t_2 : T_2$

(1) $t_1 \rightarrow t_1'$. By induction $\Gamma \vdash t_1' : T_2 \rightarrow T_1$

Proving Type Safety

Preservation. If $\Gamma \vdash t : T$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T$.

Proof. Induction on the structure of t .

$t = z \mid \lambda y:T_1. t_1 \mid \mathbf{true} \mid \mathbf{false}$ nothing to be done ($\nexists t'$)

$t = \mathbf{if} t_1 \mathbf{then} t_2 \mathbf{else} t_3$ exactly like before!

$t = t_1 t_2$. By **INV.L.** $\Gamma \vdash t : T$ implies that $T = T_1$, $\Gamma \vdash t_1 : T_2 \rightarrow T_1$
and $\Gamma \vdash t_2 : T_2$

(1) $t_1 \rightarrow t_1'$. By induction $\Gamma \vdash t_1' : T_2 \rightarrow T_1$

 $\Gamma \vdash t_1' t_2 : T_1$

Proving Type Safety

Preservation. If $\Gamma \vdash t : T$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T$.

Proof. Induction on the structure of t .

$t = z \mid \lambda y:T_1. t_1 \mid \text{true} \mid \text{false}$ nothing to be done ($\nexists t'$)

$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$ exactly like before!

$t = t_1 t_2$. By **INV.L.** $\Gamma \vdash t : T$ implies that $T = T_1$, $\Gamma \vdash t_1 : T_2 \rightarrow T_1$
and $\Gamma \vdash t_2 : T_2$

(1) $t_1 \rightarrow t_1'$. By induction $\Gamma \vdash t_1' : T_2 \rightarrow T_1$

$\Gamma \vdash t_1' t_2 : T_1$
 $t' : T$

(2) t_1 value, $t_2 \rightarrow t_2'$. Same as (1)!

Proving Type Safety

Preservation. If $\Gamma \vdash t : T$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T$.

Proof. Induction on the structure of t .

$t = z \mid \lambda y:T_1. t_1 \mid \mathbf{true} \mid \mathbf{false}$ nothing to be done ($\nexists t'$)

$t = \mathbf{if} t_1 \mathbf{then} t_2 \mathbf{else} t_3$ exactly like before!

$t = t_1 t_2$. By **INV.L.** $\Gamma \vdash t : T$ implies that $T = T_1$, $\Gamma \vdash t_1 : T_2 \rightarrow T_1$
and $\Gamma \vdash t_2 : T_2$

(3) t_1, t_2 values. Then $t_1 = \lambda x:T_2. t_{12}$. By **INV.L.** $\Gamma, x:T_2 \vdash t_{12} : T_1$

$t \rightarrow t' = [x \rightarrow t_2] t_{12}$

Proving Type Safety

Preservation. If $\Gamma \vdash t : T$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T$.

Proof. Induction on the structure of t .

$t = z \mid \lambda y:T_1. t_1 \mid \text{true} \mid \text{false}$ nothing to be done ($\nexists t'$)

$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$ exactly like before!

$t = t_1 t_2$. By **INV.L.** $\Gamma \vdash t : T$ implies that $T = T_1$, $\Gamma \vdash t_1 : T_2 \rightarrow T_1$
and $\Gamma \vdash t_2 : T_2$

(3) t_1, t_2 values. Then $t_1 = \lambda x:T_2. t_{12}$. By **INV.L.** $\Gamma, x:T_2 \vdash t_{12} : T_1$

$t \rightarrow t' = [x \rightarrow t_2] t_{12}$

Preserv. of **subst.**



$\Gamma \vdash [x \rightarrow t_2] t : T_1$

Conclusions

TODAY: implement **simply typed lambda calculus** with **let/fix** and types **Bool** and **Nat**.

To avoid repetitions and to increase readability:
give names to subexpressions!

let $x=t_1$ **in** t_2

similar to $(\lambda x:T_1. t_2) t_1 \rightarrow [x \rightarrow t_1] t_2$

↑
but this needs type T_1 explicitly!

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x:T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x=t_1 \text{ in } t_2 : T_2}$$

evaluation easy:

- (1) $t_1 \rightarrow t_1'$
- (2) t_1 value: $[x \rightarrow t_1'] t_2$

Conclusions

TODAY: implement **simply typed lambda calculus** with **let/fix** and types **Bool** and **Nat**.

To be able to type recursive functions: add **fix** to the language.

Note $\text{fix} := \lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y))$ canNOT be typed in the simply typed lambda calculus. **Can you find out WHY??**

fix $(\lambda \text{fact}. \text{factdef})$ 3 \rightarrow^* 6

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_1}{\Gamma \vdash \text{fix } t_1 : T_1}$$

evaluation

(1) $t_1 \rightarrow t_1'$

(2) $t_1 = \lambda x:T_1. t_2$ then $[x \rightarrow \text{fix } (\lambda x:T_1. t_2)] t_2$

'unroll'/expand once

Conclusions

TODAY: implement **simply typed lambda calculus** with **let/letrec** and types **Bool** and **Nat**.

To be able to type recursive functions: add **letrec** to the language.

letrec $x:T_1=t_1$ **in** t_2 := **let** $x=fix(\lambda x:T_1.t_1)$ **in** t_2

(**fix**: only internally, for typing!)

```
let rec fact:Num->Num =  
  \x:Num. if (isZero x) then (succ zero) else ...
```

language
of
today

evaluation

(1) $t_1 \rightarrow t_1'$

(2) $t_1 = \lambda x:T_1. t_2$ then $[x \rightarrow fix(\lambda x:T_1. t_2)] t_2$

'unroll'/expand once