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**

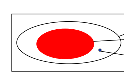
type sound = all well-typed programs are **free of execution errors**

→ 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!



→ keep the slack small!

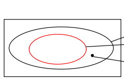
Introduce two types Bool and Nat, representing Booleans and Numbers.
Every Expr t will be of type Bool or Nat, or will have no type.

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!



→ keep the slack small!

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

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

A Type System for Arithmetic Expressions

typing rules: `true` : Bool `false` : Bool `zero` : Nat

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

$$\frac{t_1 : \text{Nat}}{\texttt{succ}\ t_1 : \text{Nat}} \qquad \frac{t_1 : \text{Nat}}{\texttt{pred}\ t_1 : \text{Nat}} \qquad \frac{t_1 : \text{Nat}}{\texttt{isZero}\ t_1 : \text{Bool}}$$

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:

$\texttt{true} : \text{Bool}$ $\qquad \texttt{false} : \text{Bool}$ $\qquad \dfrac{t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 : T}$

$\texttt{zero} : \text{Nat}$

$\dfrac{t_1 : \text{Nat}}{\texttt{succ}\ t_1 : \text{Nat}} \qquad \dfrac{t_1 : \text{Nat}}{\texttt{pred}\ t_1 : \text{Nat}} \qquad \dfrac{t_1 : \text{Nat}}{\texttt{isZero}\ t_1 : \text{Bool}}$

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

$$\dfrac{\dfrac{\texttt{zero} : \text{Nat}}{\texttt{isZero zero} : \text{Bool}} \qquad \texttt{zero} : \text{Nat} \qquad \dfrac{\texttt{zero} : \text{Nat}}{\texttt{pred zero} : \text{Nat}}}{\texttt{if isZero zero then zero else pred zero} : \text{Nat}}$$

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 deterimine 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

A Type System for Arithmetic Expressions

How to find a typing derivation?

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

INVERSION LEMMA

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!!

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 (**P**, **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 (**P**, **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 → NOT stuck → either value or
Progress
we can evaluate → 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 → NOT stuck → either value or
Progress
we can evaluate → result is well-typed
Preserve

Proving Type Safety

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

Observations: (1) if t : Bool is a value, then t = `true` or t = `false`
(2) if t : Nat is a value, then t = `succ`(… `succ` (`zero`) …) (≥ 0)

Proof. Induction on t.

t = `true` | `false` | `zero` → immediate.

t = `if` t_1 `then` t_2 `else` t_3 : R, then t_1 : 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 → t'.

Observations: (1) if t : Bool is a value, then t = `true` or t = `false`
(2) if t : Nat is a value, then t = `succ`(… `succ` (`zero`) …) (≥ 0)

Proof. Induction on t.

t = `true` | `false` | `zero` → immediate.

t = `if` t_1 `then` t_2 `else` t_3 : R, then t_1 : Bool, t_2 : R, and t_3: R (INV.L.)

- t_1 is value. By (1), t = `true` or t = `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 → t'.

Observations: (1) if t : Bool is a value, then t = `true` or t = `false`
(2) if t : Nat is a value, then t = `succ`(… `succ` (`zero`) …) (≥ 0)

Proof. Induction on t.

t = `true` | `false` | `zero` → immediate.

t = `if` t_1 `then` t_2 `else` t_3 : R, then t_1 : Bool, t_2 : R, and t_3: R (INV.L.)

- t_1 is value. By (1), t = `true` or t = `false`.
  Thus, t can evaluate to a t' (= t_2 or t_3)!

- t_1 is NOT value. By induction $\exists\, t_1'$ with $t_1 \to t_1'$.
  Thus, t can evaluate to a t' (= `if` t_1' `then` ..)!

Proving Type Safety

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

Observations: (1) if t : Bool is a value, then t = `true` or t = `false`
(2) if t : Nat is a value, then t = `succ`(… `succ` (`zero`) …) [≥ 0]

Proof. Induction on t.

t = `true` | `false` | `zero` → immediate.

t = `succ` t_1. By induction, t_1 is value or t_1 → t_1'. By INV.L., t_1 : Nat.

- t_1 is value. By (2), t_1 = `succ`(.. `zero` ..). Hence, t is also a value!
- t_1 is NOT value. Then t can evaluate to a t' (= `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 → t'.

Observations: (1) if t : Bool is a value, then t = `true` or t = `false`
(2) if t : Nat is a value, then t = `succ`(… `succ` (`zero`) …) [≥ 0]

Proof. Induction on t.

t = `true` | `false` | `zero` → immediate.

t = `pred` t_1. By induction, t_1 is value or t_1 → t_1'. By INV.L., t_1 : Nat.

- t_1 is value. By (2), t_1 = `succ`(.. `zero` ..). Thus, t can evaluate!
- t_1 is NOT value. Then t can evaluate to a t' (= `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 → t'.

Observations: (1) if t : Bool is a value, then t = `true` or t = `false`
(2) if t : Nat is a value, then t = `succ`(… `succ` (`zero`) …) [≥ 0]

Proof. Induction on t.

t = `true` | `false` | `zero` → immediate.

t = `iszero` t_1. By induction, t_1 is value or t_1 → t_1'. By INV.L., t_1 : Nat.

- t_1 is value. By (2), t_1 = `succ`(.. `zero` ..). Thus, t can evaluate!
- t_1 is NOT value. Then t can evaluate to a t' (= `iszero` t_1')

Proving Type Safety

Preservation Theorem: If t : T and t → t', then t' : T.

t = `if` t_1 `then` t_2 `else` t_3 : R, then t_1 : Bool, t_2 : R, and t_3: R (INV.L.)

t' = t_2 | t_3 | `if` t_1' `then` t_2 `else` t_3, where t_1 → t_1'

Proving Type Safety

Preservation Theorem: If t : T and t → t', then t' : T.

t = `if` t_1 `then` t_2 `else` t_3 : R, then t_1 : Bool, t_2 : R, and t_3: R (INV.L.)

t' = t_2 | t_3 | `if` t_1' `then` t_2 `else` t_3, where t_1 → t_1'
: R : R
By induction, t_1' : Bool. THUS, t' : R.

Proving Type Safety

Preservation Theorem: If t : T and t → t', then t' : T.

t = `if` t_1 `then` t_2 `else` t_3 : R, then t_1 : Bool, t_2 : R, and t_3: R (INV.L.)

t' = t_2 | t_3 | `if` t_1' `then` t_2 `else` t_3, where t_1 → t_1'
: R : R
By induction, t_1' : Bool. THUS, t' : R.

t = `succ` t_1. Thus, `succ` t_1 → `succ` t_1' and t_1 → t_1'. By INV.L., t_1 : Nat.

Proving Type Safety

Preservation Theorem: If t : T and t → t', then t' : T.

t = `if` t_1 `then` t_2 `else` t_3 : R, then t_1 : Bool, t_2 : R, and t_3: R (INV.L.)

t' = t_2 | t_3 | `if` t_1' `then` t_2 `else` t_3, where $t_1 \to t_1'$
: R : R

By induction, t_1' : Bool. THUS, t' : R.

t = `succ` t_1. Thus, `succ` t_1 → `succ` t_1' and $t_1 \to t_1'$. By INV.L., t_1 : Nat.

By induction, t_1' : Nat. THUS, also `succ` t_1' : Nat.

Proving Type Safety

Preservation Theorem: If t : T and t → t', then t' : T.

t = `if` t_1 `then` t_2 `else` t_3 : R, then t_1 : Bool, t_2 : R, and t_3: R (INV.L.)

t' = t_2 | t_3 | `if` t_1' `then` t_2 `else` t_3, where $t_1 \to t_1'$
: R : R

By induction, t_1' : Bool. THUS, t' : R.

t = `succ` t_1. Thus, `succ` t_1 → `succ` t_1' and $t_1 \to t_1'$. By INV.L., t_1 : Nat.

By induction, t_1' : Nat. THUS, also `succ` t_1' : Nat.

Cases t = `pred` t_1 | `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 (λx. x)
```

Simply Typed Lambda Calculus

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

1. Bool ∈ T
2. if $R_1, R_2 \in T$, then $R_1 \to R_2 \in T$

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

How to type `λx.t` ?
= what happens when `t` is applied to an argument?

But, what type of arguments to expect??

- annotate arguments explicitly. `λx:T₁.t` → written $\lambda x{:}T_1.t$ **explicitly typed langs.**
- deduce argument type from the body `t` of the abstraction — implicitly typed langs.

Simply Typed Lambda Calculus

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

determines a type environment for t

Type Environment $\Gamma = \{ (x_1, T_1), \ldots, (x_n, T_n) \}$ (finite function var→Types)

typing rule for lambda abstraction:

A ⊢ B = under the assumption A, B holds

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

Simply Typed Lambda Calculus

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

determines a type environment for t

Type Environment $\Gamma = \{ (x_1, T_1), \ldots, (x_n, T_n) \}$ (finite function var→Types)

typing rule for lambda abstraction:

A ⊢ B = under the assumption A, B holds

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

"making the assumption $x{:}T_1$ explicit"

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

Simply Typed Lambda Calculus

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

$$\frac{\Gamma \vdash t_1:T \to R \qquad \Gamma \vdash t_2:T}{\Gamma \vdash t_1\ t_2 : R}$$ function application

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

a derivation tree:

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

Simply Typed Lambda Calculus

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

$$\frac{\Gamma \vdash t_1:T \to R \qquad \Gamma \vdash t_2:T}{\Gamma \vdash t_1\ t_2 : R}$$ function application

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

a derivation tree:

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

Simply Typed Lambda Calculus

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

$$\frac{\Gamma \vdash t_1:T \to R \qquad \Gamma \vdash t_2:T}{\Gamma \vdash t_1\ t_2 : R}$$ function application

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

a derivation tree:

$$\frac{\text{abstraction}\ \dfrac{x : Bool \vdash x : Bool}{\vdash \lambda x:Bool.\ x : Bool \to Bool} \qquad \vdash true:Bool}{\vdash (\lambda x:Bool.\ x)\ true : Bool}\ \text{application}$$

Simply Typed Lambda Calculus

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

$$\frac{\Gamma \vdash t_1:T \to R \qquad \Gamma \vdash t_2:T}{\Gamma \vdash t_1\ t_2 : R}$$ function application

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

a derivation tree:

$$\frac{\text{abstraction}\ \dfrac{\dfrac{x : Bool \in x : Bool}{x : Bool \vdash x : Bool}}{\vdash \lambda x:Bool.\ x : Bool \to Bool} \qquad \vdash true:Bool}{\vdash (\lambda x:Bool.\ x)\ true : Bool}\ \text{application}$$

Proving Type Safety

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

INV. L.
1. If $\Gamma \vdash x : R$, then $x:R \in \Gamma$.
2. If $\Gamma \vdash \lambda x:T_1.t : R$, then $R = T_1 \to R_2$ 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. $\Gamma \vdash t_1: T \to R$ and $\Gamma \vdash t_2 : T$.

Observation (3) If v is a value of type $T_1 \to T_2$, 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 → t'.

Proof. t = true | false | 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).

INV. L.
1. If $\Gamma \vdash x : R$, then $x:R \in \Gamma$.
2. If $\Gamma \vdash \lambda x:T_1.t : R$, then $R = T_1 \to R_2$ 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. $\Gamma \vdash t_1: T \to R$ and $\Gamma \vdash t_2 : T$.

Observation (3) If v is a value of type $T_1 \to T_2$, 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 → t'.

Proof. t = true | false | if .. like before!

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

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

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

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

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

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** $\Gamma \vdash [x \to s]\, t : T$

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 $\Gamma \vdash [x \to s]\, x : T$

   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** $\Gamma \vdash [x \to s]\, t : T$

Proof.

induction on structure of t. 6 cases

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

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

   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** $\Gamma \vdash [x \to s]\, t : T$

Proof.

induction on structure of t. 6 cases

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

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

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

   By induction, $$\frac{\Gamma' \vdash [x \to s]\, t_1 : T_1.}{\Gamma \vdash \lambda y{:}T_2.\ [x \to s] t_1 : T_2 \to T_1.}\ \text{abstraction}$$

Proving Type Safety

Preservation of substitution:

If (1) $\Gamma \vdash s : S$ (2) $\Gamma,\ x{:}S \vdash t : T$ **then** $\Gamma \vdash [x \to s]\, t : T$

Proof.

induction on structure of t. 6 cases

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

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

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

   By induction, $$\frac{\Gamma' \vdash [x \to s]\, t_1 : T_1.}{\Gamma \vdash \lambda y{:}T_2.\ [x \to s] t_1 : T_2 \to T_1.}\ \text{abstraction}$$

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

Proving Type Safety

Preservation of substitution:

If (1) $\Gamma \vdash s : S$ (2) $\Gamma,\ x{:}S \vdash t : T$ **then** $\Gamma \vdash [x \to s]\, t : T$

Proof.

induction on structure of t. 6 cases

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

   $\Gamma,\ x{:}S \vdash t_1 : T_2 \to T_1$
   $\Gamma,\ x{:}S \vdash t_2 : T_2$ with $T = T_1$

   By induction (2x): $$\frac{\Gamma \vdash [x \to s]\, t_1 : T_2 \to T_1 \qquad \Gamma \vdash [x \to s]\, t_2 : T_2}{\Gamma \vdash [x \to s] t_1\ [x \to s] t_2 : T_1}\ \text{application}$$

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

Proving Type Safety

Preservation of substitution:

If (1) $\Gamma \vdash s : S$ (2) $\Gamma,\ x{:}S \vdash t : T$ **then** $\Gamma \vdash [x \to s]\, t : T$

Proof.

induction on structure of t. 6 cases

4. t = `true`. By INV.L., T = Bool.
   $[x \to s]\, t$ = true, and $\Gamma \vdash$ true : Bool ($\forall \Gamma$)

5. t = `false`. Same thing.

6. t = `if` t_1 `then` t_2 `else` t_3.

   by INV.L. $\Gamma,\ x{:}S \vdash t_1 : Bool$, $\Gamma,\ x{:}S \vdash t_2 : T$, $\Gamma,\ x{:}S \vdash t_3 : T$ induct. $$\frac{\Gamma,\ x{:}S \vdash [x \to s] t_1 : Bool \qquad \Gamma,\ x{:}S \vdash [x \to s] t_2 : T \qquad \Gamma,\ x{:}S \vdash [x \to s] t_3 : T}{\Gamma \vdash [x \to s]\ \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 : T}$$

Proving Type Safety

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

Proof. Induction on the structure of t.

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

$t =$ `if` t_1 `then` t_2 `else` t_3 exactly like before!

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

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

Proving Type Safety

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

Proof. Induction on the structure of t.

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

$t =$ `if` t_1 `then` t_2 `else` t_3 exactly like before!

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

(1) $t_1 \to t_1'$. By induction

$$\frac{\Gamma \vdash t_1' : T_2 \to T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1'\ t_2 : T_1}$$

Proving Type Safety

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

Proof. Induction on the structure of t.

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

$t =$ `if` t_1 `then` t_2 `else` t_3 exactly like before!

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

(1) $t_1 \to t_1'$. By induction

$$\frac{\Gamma \vdash t_1' : T_2 \to T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1'\ t_2 : T_1}$$

$t' \ : T$

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

Proving Type Safety

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

Proof. Induction on the structure of t.

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

$t =$ `if` t_1 `then` t_2 `else` t_3 exactly like before!

$t = t_1\ t_2$. By INV.L. $\Gamma \vdash t : T$ implies that $T=T_1$, $\Gamma \vdash t_1 : T_2 \to T_1$
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. $\Gamma, x{:}T_2 \vdash t_{12} : T_1$

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

Proving Type Safety

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

Proof. Induction on the structure of t.

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

$t =$ `if` t_1 `then` t_2 `else` t_3 exactly like before!

$t = t_1\ t_2$. By INV.L. $\Gamma \vdash t : T$ implies that $T=T_1$, $\Gamma \vdash t_1 : T_2 \to T_1$
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. $\Gamma, x{:}T_2 \vdash t_{12} : T_1$

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

Preserv. of subst.

$\downarrow$

$\Gamma \vdash [\, x \to t_2 \,]\, t : T_1$

Conclusions

TODAY: implement simply typed lambda caculus with `let/fix`
and types Bool and Nat.

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

`let` $x=t_1$ `in` t_2

similar to $(\lambda x{:}T_1.\, t_2)\ t_1 \to [\, x \to t_1 \,]\, t_2$

but this needs type T_1 explicitely!

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

evaluation easy:
(1) $t_1 \to t_1'$
(2) t_1 value: $[\, x \to t_1 \,]\, t_2$

Conclusions

TODAY: implement simply typed lambda caculus with `let/fix` and types Bool and Nat.

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

Note fix := λf. (λx. f (λy. x x y)) (λx. f (λy. x x y)) canNOT be typed in the simply typed lambda calculus. Can you find out WHY??

`fix (λfact. factdef) 3` →* 6

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

evaluation

(1) $t_1 \rightarrow t_1'$

(2) $t_1 = \lambda x{:}T_1 : t_2$ then $[\, x \rightarrow \texttt{fix}\ (\lambda x{:}T_1.\ t_2)\,]\ t_2$ — 'unroll'/expand once

Conclusions

TODAY: implement simply typed lambda caculus with `let/letrec` and types Bool and Nat.

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

`letrec x:T1=t1 in t2` := `let x=fix(λx:T1.t1) in t2`

(`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 $[\, x \rightarrow \texttt{fix}\ (\lambda x{:}T_1.\ t_2)\,]\ t_2$ — 'unroll'/expand once