Foundations of Software
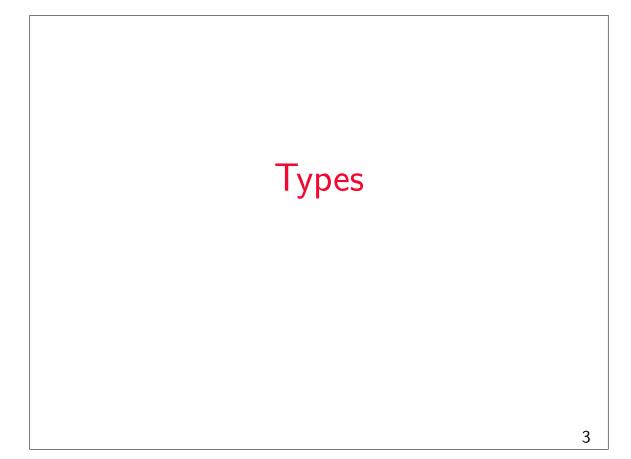# Fall 2023

Week 5

Plan

PREVIOUSLY: untyped lambda calculus

TODAY: types!!
1. Two example languages:
    1.1 typing arithmetic expressions
    1.2 simply typed lambda calculus (STLC)
2. For each:
    2.1 Define types
    2.2 Specify typing rules
    2.3 Prove soundness: *progress* and *preservation*

NEXT: lambda calculus extensions
NEXT: polymorphic typing

Types

Questions

What are your questions?

Problematic arithmetic expressions

Consider the term

$$t = \texttt{if true then (pred false) else 0}$$

Clicker question: What can we say about it? (multiple possible answers)

A. `t` is stuck

B. `t` is a closed term

C. `t` is typeable, i.e., there exists `T` such that `t : T`

D. there exists t' such that $t \longrightarrow^* t'$ and t' is stuck

URL: `ttpoll.eu`
Session ID: `cs452`

5

Safety for arithmetic expressions

Recall from last week: safety = progress + preservation, given some definition of "`t` is *valid*".

▶ Progress: if `t` is valid, then either `t` is a value, or $t \longrightarrow t'$.

▶ Preservation: if `t` is valid and $t \longrightarrow t'$, then t' is valid.

What is "is valid" for arithmetic expressions?

6

The type of variables in STLC

Consider the term

$$t = \lambda x : \texttt{Bool. if x then false else true}$$

Clicker question: What is `T` in `x : T`?

A. `T = Bool`

B. `T = Bool` $\rightarrow$ `Bool`

C. there are multiple such `T`'s

D. none of the above

URL: `ttpoll.eu`
Session ID: `cs452`

Safety for STLC

What is "is valid" for the Simply Typed Lambda Calculus with Booleans?

Reading for next week

▶ Chapter 11 until section 11.7 (Tuples) included

Outline

1. begin with a set of terms, a set of values, and an evaluation relation
2. define a set of *types* classifying values according to their "shapes"
3. define a *typing relation* `t : T` that classifies terms according to the shape of the values that result from evaluating them
4. check that the typing relation is *sound* in the sense that,
   4.1 if `t : T` and $t \longrightarrow^* v$, then `v : T`
   4.2 if `t : T`, then evaluation of `t` will not get stuck

Recall: Arithmetic Expressions – Syntax

```
t  ::=                                    terms
     true                                  constant true
     false                                constant false
     if t then t else t                   conditional
     0                                    constant zero
     succ t                               successor
     pred t                               predecessor
     iszero t                             zero test
v  ::=                                    values
     true                                 true value
     false                                false value
     nv                                   numeric value

nv ::=                                    numeric values
     0                                    zero value
     succ nv                              successor value
```

Recall: Arithmetic Expressions – Evaluation Rules

$$\text{if true then } t_2 \text{ else } t_3 \longrightarrow t_2 \qquad (\text{E-IfTrue})$$

$$\text{if false then } t_2 \text{ else } t_3 \longrightarrow t_3 \qquad (\text{E-IfFalse})$$

$$\text{pred } 0 \longrightarrow 0 \qquad (\text{E-PredZero})$$

$$\text{pred (succ } nv_1) \longrightarrow nv_1 \qquad (\text{E-PredSucc})$$

$$\text{iszero } 0 \longrightarrow \text{true} \qquad (\text{E-IszeroZero})$$

$$\text{iszero (succ } nv_1) \longrightarrow \text{false} \qquad (\text{E-IszeroSucc})$$

Recall: Arithmetic Expressions – Evaluation Rules

$$\frac{t_1 \longrightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3} \quad \text{(E-IF)}$$

$$\frac{t_1 \longrightarrow t_1'}{\text{succ } t_1 \longrightarrow \text{succ } t_1'} \quad \text{(E-SUCC)}$$

$$\frac{t_1 \longrightarrow t_1'}{\text{pred } t_1 \longrightarrow \text{pred } t_1'} \quad \text{(E-PRED)}$$

$$\frac{t_1 \longrightarrow t_1'}{\text{iszero } t_1 \longrightarrow \text{iszero } t_1'} \quad \text{(E-ISZERO)}$$

Types

In this language, values have two possible "shapes": they are either booleans or numbers.

```
T  ::=                                      types
       Bool                                     type of booleans
       Nat                                      type of numbers
```

Typing Rules

$$\text{true : Bool} \qquad\qquad \text{(T-TRUE)}$$

$$\text{false : Bool} \qquad\qquad \text{(T-FALSE)}$$

$$\frac{\text{t}_1 \text{ : Bool} \qquad \text{t}_2 \text{ : T} \qquad \text{t}_3 \text{ : T}}{\text{if t}_1 \text{ then t}_2 \text{ else t}_3 \text{ : T}} \qquad \text{(T-IF)}$$

$$\text{0 : Nat} \qquad\qquad \text{(T-ZERO)}$$

$$\frac{\text{t}_1 \text{ : Nat}}{\text{succ t}_1 \text{ : Nat}} \qquad \text{(T-SUCC)}$$

$$\frac{\text{t}_1 \text{ : Nat}}{\text{pred t}_1 \text{ : Nat}} \qquad \text{(T-PRED)}$$

$$\frac{\text{t}_1 \text{ : Nat}}{\text{iszero t}_1 \text{ : Bool}} \qquad \text{(T-ISZERO)}$$

15

Typing Derivations

Every pair (t, T) in the typing relation can be justified by a
derivation tree built from instances of the inference rules.

$$\cfrac{\cfrac{\overline{\text{0 : Nat}}\ \text{T-ZERO}}{\text{iszero 0 : Bool}}\ \text{T-ISZERO} \quad \cfrac{}{\text{0 : Nat}}\ \text{T-ZERO} \quad \cfrac{\cfrac{\overline{\text{0 : Nat}}\ \text{T-ZERO}}{\text{pred 0 : Nat}}\ \text{T-PRED}}{}}{\text{if iszero 0 then 0 else pred 0 : Nat}}\ \text{T-IF}$$

Proofs of properties about the typing relation often proceed by
induction on typing derivations.

16

Imprecision of Typing

Like other static program analyses, type systems are generally *imprecise*: they do not predict exactly what kind of value will be returned by every program, but just a conservative (safe) approximation.

$$\frac{\texttt{t}_1 : \texttt{Bool} \qquad \texttt{t}_2 : \texttt{T} \qquad \texttt{t}_3 : \texttt{T}}{\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 : \texttt{T}} \qquad (\text{T-I}_\text{F})$$

Using this rule, we cannot assign a type to

$$\texttt{if true then 0 else false}$$

even though this term will certainly evaluate to a number.

Type Safety

The safety (or soundness) of this type system can be expressed by two properties:

1. *Progress:* A well-typed term is not stuck
   If $t : T$, then either t is a value or else $t \longrightarrow t'$ for some t'.

2. *Preservation:* Types are preserved by one-step evaluation
   If $t : T$ and $t \longrightarrow t'$, then $t' : T$.

Inversion

Lemma:

1. If `true : R`, then $R = \text{Bool}$.
2. If `false : R`, then $R = \text{Bool}$.
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 `0 : R`, then $R = \text{Nat}$.
5. If `succ `t_1` : R`, then $R = \text{Nat}$ and $t_1 : \text{Nat}$.
6. If `pred `t_1` : R`, then $R = \text{Nat}$ and $t_1 : \text{Nat}$.
7. If `iszero `t_1` : R`, then $R = \text{Bool}$ and $t_1 : \text{Nat}$.

Inversion

Lemma:

1. If $\text{true} : R$, then $R = \text{Bool}$.
2. If $\text{false} : R$, then $R = \text{Bool}$.
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 $0 : R$, then $R = \text{Nat}$.
5. If $\text{succ } t_1 : R$, then $R = \text{Nat}$ and $t_1 : \text{Nat}$.
6. If $\text{pred } t_1 : R$, then $R = \text{Nat}$ and $t_1 : \text{Nat}$.
7. If $\text{iszero } t_1 : R$, then $R = \text{Bool}$ and $t_1 : \text{Nat}$.

Proof: ...

This leads directly to a recursive algorithm for calculating the type of a term...

Typechecking Algorithm

```
typeof(t) = if t = true then Bool
            else if t = false then Bool
            else if t = if t1 then t2 else t3 then
              let T1 = typeof(t1) in
              let T2 = typeof(t2) in
              let T3 = typeof(t3) in
              if T1 = Bool and T2=T3 then T2
              else "not typable"
            else if t = 0 then Nat
            else if t = succ t1 then
              let T1 = typeof(t1) in
              if T1 = Nat then Nat else "not typable"
            else if t = pred t1 then
              let T1 = typeof(t1) in
              if T1 = Nat then Nat else "not typable"
            else if t = iszero t1 then
              let T1 = typeof(t1) in
              if T1 = Nat then Bool else "not typable"
```

Properties of the Typing Relation

Recall: Typing Rules

$$\texttt{true : Bool} \qquad \text{(T-TRUE)}$$

$$\texttt{false : Bool} \qquad \text{(T-FALSE)}$$

$$\frac{\texttt{t}_1 \texttt{ : Bool} \qquad \texttt{t}_2 \texttt{ : T} \qquad \texttt{t}_3 \texttt{ : T}}{\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 \texttt{ : T}} \qquad \text{(T-IF)}$$

$$\texttt{0 : Nat} \qquad \text{(T-ZERO)}$$

$$\frac{\texttt{t}_1 \texttt{ : Nat}}{\texttt{succ t}_1 \texttt{ : Nat}} \qquad \text{(T-SUCC)}$$

$$\frac{\texttt{t}_1 \texttt{ : Nat}}{\texttt{pred t}_1 \texttt{ : Nat}} \qquad \text{(T-PRED)}$$

$$\frac{\texttt{t}_1 \texttt{ : Nat}}{\texttt{iszero t}_1 \texttt{ : Bool}} \qquad \text{(T-ISZERO)}$$

Recall: Inversion

Lemma:

1. If $\texttt{true} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
2. If $\texttt{false} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
3. If $\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : \texttt{R}$, then $t_1 : \texttt{Bool}$, $t_2 : \texttt{R}$, and $t_3 : \texttt{R}$.
4. If $\texttt{0} : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$.
5. If $\texttt{succ } t_1 : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$ and $t_1 : \texttt{Nat}$.
6. If $\texttt{pred } t_1 : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$ and $t_1 : \texttt{Nat}$.
7. If $\texttt{iszero } t_1 : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$ and $t_1 : \texttt{Nat}$.

Canonical Forms

Lemma:

1. If $\texttt{v}$ is a value of type $\texttt{Bool}$, then $\texttt{v}$ is either $\texttt{true}$ or $\texttt{false}$.
2. If $\texttt{v}$ is a value of type $\texttt{Nat}$, then $\texttt{v}$ is a numeric value.

Proof:

Canonical Forms

Lemma:

1. If `v` is a value of type `Bool`, then `v` is either `true` or `false`.
2. If `v` is a value of type `Nat`, then `v` is a numeric value.

Proof: Recall the syntax of values:

`v`	`::=`		*values*
	`true`		*true value*
	`false`		*false value*
	`nv`		*numeric value*
`nv`	`::=`		*numeric values*
	`0`		*zero value*
	`succ nv`		*successor value*

For part 1,

Canonical Forms

Lemma:

1. If `v` is a value of type `Bool`, then `v` is either `true` or `false`.
2. If `v` is a value of type `Nat`, then `v` is a numeric value.

Proof: Recall the syntax of values:

`v`	`::=`		*values*
	`true`		*true value*
	`false`		*false value*
	`nv`		*numeric value*
`nv`	`::=`		*numeric values*
	`0`		*zero value*
	`succ nv`		*successor value*

For part 1, if `v` is `true` or `false`, the result is immediate.

Canonical Forms

Lemma:

1. If `v` is a value of type `Bool`, then `v` is either `true` or `false`.
2. If `v` is a value of type `Nat`, then `v` is a numeric value.

Proof: Recall the syntax of values:

`v`	`::=`		*values*
	`true`		*true value*
	`false`		*false value*
	`nv`		*numeric value*
`nv`	`::=`		*numeric values*
	`0`		*zero value*
	`succ nv`		*successor value*

For part 1, if `v` is `true` or `false`, the result is immediate. But `v` cannot be `0` or `succ nv`, since the inversion lemma tells us that `v` would then have type `Nat`, not `Bool`.

24

Progress

Theorem: Suppose t is a well-typed term (that is, $t : T$ for some type T). Then either t is a value or else there is some t' with $t \longrightarrow t'$.

Progress

Theorem: Suppose t is a well-typed term (that is, $t : T$ for some type T). Then either t is a value or else there is some t' with $t \longrightarrow t'$.

Proof:

Progress

Theorem: Suppose t is a well-typed term (that is, $t : T$ for some type T). Then either t is a value or else there is some t' with $t \longrightarrow t'$.

Proof: By induction on a derivation of $t : T$.

Progress

Theorem: Suppose t is a well-typed term (that is, $t : T$ for some type T). Then either t is a value or else there is some t' with $t \longrightarrow t'$.

Proof: By induction on a derivation of $t : T$.

The T-TRUE, T-FALSE, and T-ZERO cases are immediate, since t in these cases is a value.

Progress

Theorem: Suppose t is a well-typed term (that is, $t : T$ for some type T). Then either t is a value or else there is some t' with $t \longrightarrow t'$.

Proof: By induction on a derivation of $t : T$.

The T-TRUE, T-FALSE, and T-ZERO cases are immediate, since t in these cases is a value.

Case T-IF: $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$
                 $t_1 : \text{Bool}$ $t_2 : T$ $t_3 : T$

25

Progress

Theorem: Suppose t is a well-typed term (that is, $t : T$ for some type T). Then either t is a value or else there is some t' with $t \longrightarrow t'$.

Proof: By induction on a derivation of $t : T$.

The T-TRUE, T-FALSE, and T-ZERO cases are immediate, since t in these cases is a value.

Case T-IF: $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$
                 $t_1 : \text{Bool}$ $t_2 : T$ $t_3 : T$

By the induction hypothesis, either t_1 is a value or else there is some t_1' such that $t_1 \longrightarrow t_1'$. If t_1 is a value, then the canonical forms lemma tells us that it must be either `true` or `false`, in which case either E-IFTRUE or E-IFFALSE applies to t. On the other hand, if $t_1 \longrightarrow t_1'$, then, by E-IF, $t \longrightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3$.

25

Progress

Theorem: Suppose t is a well-typed term (that is, $t : T$ for some type T). Then either t is a value or else there is some t' with $t \longrightarrow t'$.

Proof: By induction on a derivation of $t : T$.

The cases for rules T-ZERO, T-SUCC, T-PRED, and T-ISZERO are similar.

(Recommended: Try to reconstruct them.)

26

Preservation

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

27

Preservation

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

Proof: By induction on the given typing derivation.

Preservation

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

Proof: By induction on the given typing derivation.

Case T-TRUE: $t = true$ $T = Bool$

Then t is a value.

Preservation

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

Proof: By induction on the given typing derivation.

Case T-IF:
$\quad$ $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \quad t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T$

There are three evaluation rules by which $t \longrightarrow t'$ can be derived: E-IFTRUE, E-IFFALSE, and E-IF. Consider each case separately.

Preservation

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

Proof: By induction on the given typing derivation.

Case T-IF:
$\quad$ $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \quad t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T$

There are three evaluation rules by which $t \longrightarrow t'$ can be derived: E-IFTRUE, E-IFFALSE, and E-IF. Consider each case separately.

Subcase E-IFTRUE: $\quad t_1 = \text{true} \quad t' = t_2$

Immediate, by the assumption $t_2 : T$.

(E-IFFALSE subcase: Similar.)

Preservation

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

Proof: By induction on the given typing derivation.

Case T-IF:
   $t = $ if t_1 then t_2 else t_3 $t_1 : $ Bool $t_2 : T$ $t_3 : T$

There are three evaluation rules by which $t \longrightarrow t'$ can be derived:
E-IFTRUE, E-IFFALSE, and E-IF. Consider each case separately.

Subcase E-IF: $t_1 \longrightarrow t_1'$ $t' = $ if t_1' then t_2 else t_3

Applying the IH to the subderivation of $t_1 : $ Bool yields
$t_1' : $ Bool. Combining this with the assumptions that $t_2 : T$ and
$t_3 : T$, we can apply rule T-IF to conclude that
if t_1' then t_2 else $t_3 : T$, that is, $t' : T$.

Messing With It

Messing with it: Remove a rule

What if we remove E-PREDZERO ?

29

Messing with it: Remove a rule

What if we remove E-PREDZERO ?

Then `pred 0` type checks, but it is stuck and is not a value. Thus the progress theorem fails.

29

Messing with it: If

What if we change the rule for typing `if`'s to the following ?:

$$\frac{t_1 : \text{Bool} \qquad t_2 : \text{Nat} \qquad t_3 : \text{Nat}}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \text{Nat}} \quad \text{(T-IF)}$$

The system is still sound. Some `if`'s do not type, but those that do are fine.

Messing with it: adding bit

t	::=		*terms*
	...		
	$bit(t)$		*boolean to natural*

1. evaluation rule
2. typing rule
3. progress and preservation updates

The Simply Typed Lambda-Calculus

The simply typed lambda-calculus

The system we are about to define is commonly called the *simply typed lambda-calculus*, or $\lambda_\to$ for short.

Unlike the untyped lambda-calculus, the "pure" form of $\lambda_\to$ (with no primitive values or operations) is not very interesting; to talk about $\lambda_\to$, we always begin with some set of "base types."

- ▶ So, strictly speaking, there are *many* variants of $\lambda_\to$, depending on the choice of base types.
- ▶ For now, we'll work with a variant constructed over the booleans.

Untyped lambda-calculus with booleans

```
t  ::=                                   terms
    x                                      variable
    λx.t                                   abstraction
    t t                                    application
    true                                   constant true
    false                                  constant false
    if t then t else t                     conditional

v  ::=                                   values
    λx.t                                    abstraction value
    true                                   true value
    false                                  false value
```

"Simple Types"

```
T ::=                              types
    Bool                               type of booleans
    T→T                                types of functions
```

What are some examples?

Type Annotations

We now have a choice to make. Do we...

► annotate lambda-abstractions with the expected type of the argument

$$\lambda x:T_1. \ t_2$$

(as in most mainstream programming languages), or

► continue to write lambda-abstractions as before

$$\lambda x. \ t_2$$

and ask the typing rules to "guess" an appropriate annotation (as in OCaml)?

Both are reasonable choices, but the first makes the job of defining the typing rules simpler. Let's take this choice for now.

Typing rules

$$\text{true : Bool} \qquad \text{(T-True)}$$

$$\text{false : Bool} \qquad \text{(T-False)}$$

$$\frac{\text{t}_1 \text{ : Bool} \qquad \text{t}_2 \text{ : T} \qquad \text{t}_3 \text{ : T}}{\text{if t}_1 \text{ then t}_2 \text{ else t}_3 \text{ : T}} \qquad \text{(T-If)}$$

Typing rules

$$\text{true : Bool} \qquad \text{(T-True)}$$

$$\text{false : Bool} \qquad \text{(T-False)}$$

$$\frac{\text{t}_1 \text{ : Bool} \qquad \text{t}_2 \text{ : T} \qquad \text{t}_3 \text{ : T}}{\text{if t}_1 \text{ then t}_2 \text{ else t}_3 \text{ : T}} \qquad \text{(T-If)}$$

$$\frac{\text{???}}{\lambda \text{x:T}_1.\text{t}_2 \text{ : T}_1 \rightarrow \text{T}_2} \qquad \text{(T-Abs)}$$

Typing rules

$$\text{true : Bool} \qquad \text{(T-TRUE)}$$

$$\text{false : Bool} \qquad \text{(T-FALSE)}$$

$$\frac{t_1 : \text{Bool} \qquad t_2 : T \qquad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \qquad \text{(T-IF)}$$

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

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(T-VAR)}$$

37

Typing rules

$$\Gamma \vdash \text{true : Bool} \qquad \text{(T-TRUE)}$$

$$\Gamma \vdash \text{false : Bool} \qquad \text{(T-FALSE)}$$

$$\frac{\Gamma \vdash t_1 : \text{Bool} \qquad \Gamma \vdash t_2 : T \qquad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \qquad \text{(T-IF)}$$

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

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} {\to} T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \; t_2 : T_{12}} \qquad \text{(T-APP)}$$

37

Typing Derivations

What derivations justify the following typing statements?

- ⊢ (λx:Bool.x) true : Bool
- f:Bool→Bool ⊢
    f (if false then true else false) : Bool
- f:Bool→Bool ⊢
    λx:Bool. f (if x then false else x) : Bool→Bool

Properties of $\lambda_\rightarrow$

The fundamental property of the type system we have just defined is *soundness* with respect to the operational semantics.

1. *Progress:* A closed, well-typed term is not stuck

    If ⊢ t : T, *then either* t *is a value or else* $t \longrightarrow t'$ *for some* t'.

2. *Preservation:* Types are preserved by one-step evaluation

    If Γ ⊢ t : T *and* $t \longrightarrow t'$, *then* Γ ⊢ t' : T.

Proving progress

Same steps as before...

Proving progress

Same steps as before...
- ▶ inversion lemma for typing relation
- ▶ canonical forms lemma
- ▶ progress theorem

Inversion

Lemma:

1. If $\Gamma \vdash \texttt{true} : R$, then $R = \texttt{Bool}$.

2. If $\Gamma \vdash \texttt{false} : R$, then $R = \texttt{Bool}$.

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

Inversion

Lemma:

1. If $\Gamma \vdash \texttt{true} : R$, then $R = \texttt{Bool}$.

2. If $\Gamma \vdash \texttt{false} : R$, then $R = \texttt{Bool}$.

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

4. If $\Gamma \vdash x : R$, then

Inversion

Lemma:

1. If $\Gamma \vdash \texttt{true} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
2. If $\Gamma \vdash \texttt{false} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
3. If $\Gamma \vdash \texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 : \texttt{R}$, then $\Gamma \vdash \texttt{t}_1 : \texttt{Bool}$ and $\Gamma \vdash \texttt{t}_2, \texttt{t}_3 : \texttt{R}$.
4. If $\Gamma \vdash \texttt{x} : \texttt{R}$, then $\texttt{x:R} \in \Gamma$.

Inversion

Lemma:

1. If $\Gamma \vdash \texttt{true} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
2. If $\Gamma \vdash \texttt{false} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
3. If $\Gamma \vdash \texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 : \texttt{R}$, then $\Gamma \vdash \texttt{t}_1 : \texttt{Bool}$ and $\Gamma \vdash \texttt{t}_2, \texttt{t}_3 : \texttt{R}$.
4. If $\Gamma \vdash \texttt{x} : \texttt{R}$, then $\texttt{x:R} \in \Gamma$.
5. If $\Gamma \vdash \lambda\texttt{x:T}_1.\texttt{t}_2 : \texttt{R}$, then

Inversion

Lemma:

1. If $\Gamma \vdash \texttt{true} : R$, then $R = \texttt{Bool}$.
2. If $\Gamma \vdash \texttt{false} : R$, then $R = \texttt{Bool}$.
3. If $\Gamma \vdash \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : R$, then $\Gamma \vdash t_1 : \texttt{Bool}$ and $\Gamma \vdash t_2, t_3 : R$.
4. If $\Gamma \vdash x : R$, then $x{:}R \in \Gamma$.
5. If $\Gamma \vdash \lambda x{:}T_1.t_2 : R$, then $R = T_1 {\rightarrow} R_2$ for some R_2 with $\Gamma, x{:}T_1 \vdash t_2 : R_2$.

41

Inversion

Lemma:

1. If $\Gamma \vdash \texttt{true} : R$, then $R = \texttt{Bool}$.
2. If $\Gamma \vdash \texttt{false} : R$, then $R = \texttt{Bool}$.
3. If $\Gamma \vdash \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : R$, then $\Gamma \vdash t_1 : \texttt{Bool}$ and $\Gamma \vdash t_2, t_3 : R$.
4. If $\Gamma \vdash x : R$, then $x{:}R \in \Gamma$.
5. If $\Gamma \vdash \lambda x{:}T_1.t_2 : R$, then $R = T_1 {\rightarrow} R_2$ for some R_2 with $\Gamma, x{:}T_1 \vdash t_2 : R_2$.
6. If $\Gamma \vdash t_1 \; t_2 : R$, then

41

Inversion

Lemma:

1. If $\Gamma \vdash \texttt{true} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.

2. If $\Gamma \vdash \texttt{false} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.

3. If $\Gamma \vdash \texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 : \texttt{R}$, then $\Gamma \vdash \texttt{t}_1 : \texttt{Bool}$ and $\Gamma \vdash \texttt{t}_2, \texttt{t}_3 : \texttt{R}$.

4. If $\Gamma \vdash \texttt{x} : \texttt{R}$, then $\texttt{x}:\texttt{R} \in \Gamma$.

5. If $\Gamma \vdash \lambda\texttt{x}:\texttt{T}_1.\texttt{t}_2 : \texttt{R}$, then $\texttt{R} = \texttt{T}_1 \rightarrow \texttt{R}_2$ for some $\texttt{R}_2$ with $\Gamma, \texttt{x}:\texttt{T}_1 \vdash \texttt{t}_2 : \texttt{R}_2$.

6. If $\Gamma \vdash \texttt{t}_1 \ \texttt{t}_2 : \texttt{R}$, then there is some type $\texttt{T}_{11}$ such that $\Gamma \vdash \texttt{t}_1 : \texttt{T}_{11} \rightarrow \texttt{R}$ and $\Gamma \vdash \texttt{t}_2 : \texttt{T}_{11}$.

Canonical Forms

Lemma:

Canonical Forms

Lemma:

1. If `v` is a value of type `Bool`, then

Canonical Forms

Lemma:

1. If `v` is a value of type `Bool`, then `v` is either `true` or `false`.

Canonical Forms

Lemma:

1. If v is a value of type `Bool`, then v is either `true` or `false`.
2. If v is a value of type $T_1 \rightarrow T_2$, then

Canonical Forms

Lemma:

1. If v is a value of type `Bool`, then v is either `true` or `false`.
2. If v is a value of type $T_1 \rightarrow T_2$, then v has the form $\lambda x{:}T_1.t_2$.

Progress

Theorem: Suppose t is a closed, well-typed term (that is, $\vdash t : T$ for some T). Then either t is a value or else there is some t' with $t \longrightarrow t'$.

Proof: By induction

43

Progress

Theorem: Suppose t is a closed, well-typed term (that is, $\vdash t : T$ for some T). Then either t is a value or else there is some t' with $t \longrightarrow t'$.

Proof: By induction on typing derivations.

43

Progress

Theorem: Suppose t is a closed, well-typed term (that is, $\vdash t : T$ for some T). Then either t is a value or else there is some t' with $t \longrightarrow t'$.

Proof: By induction on typing derivations. The cases for boolean constants and conditions are the same as before. The variable case is trivial (because t is closed). The abstraction case is immediate, since abstractions are values.

Progress

Theorem: Suppose t is a closed, well-typed term (that is, $\vdash t : T$ for some T). Then either t is a value or else there is some t' with $t \longrightarrow t'$.

Proof: By induction on typing derivations. The cases for boolean constants and conditions are the same as before. The variable case is trivial (because t is closed). The abstraction case is immediate, since abstractions are values.
Consider the case for application, where $t = t_1\ t_2$ with $\vdash t_1 : T_{11} {\rightarrow} T_{12}$ and $\vdash t_2 : T_{11}$.

Progress

Theorem: Suppose t is a closed, well-typed term (that is, $\vdash t : T$ for some T). Then either t is a value or else there is some t' with $t \longrightarrow t'$.

Proof: By induction on typing derivations. The cases for boolean constants and conditions are the same as before. The variable case is trivial (because t is closed). The abstraction case is immediate, since abstractions are values.

Consider the case for application, where $t = t_1\ t_2$ with $\vdash t_1 : T_{11} \to T_{12}$ and $\vdash t_2 : T_{11}$. By the induction hypothesis, either t_1 is a value or else it can make a step of evaluation, and likewise t_2.

43

Progress

Theorem: Suppose t is a closed, well-typed term (that is, $\vdash t : T$ for some T). Then either t is a value or else there is some t' with $t \longrightarrow t'$.

Proof: By induction on typing derivations. The cases for boolean constants and conditions are the same as before. The variable case is trivial (because t is closed). The abstraction case is immediate, since abstractions are values.

Consider the case for application, where $t = t_1\ t_2$ with $\vdash t_1 : T_{11} \to T_{12}$ and $\vdash t_2 : T_{11}$. By the induction hypothesis, either t_1 is a value or else it can make a step of evaluation, and likewise t_2. If t_1 can take a step, then rule E-App1 applies to t. If t_1 is a value and t_2 can take a step, then rule E-App2 applies. Finally, if both t_1 and t_2 are values, then the canonical forms lemma tells us that t_1 has the form $\lambda x{:}T_{11}.t_{12}$, and so rule E-AppAbs applies to t.

43