Types and Programming Languages

Lecture 1. Introduction & Untyped arithmetic expressions

Xiaojuan Cai
cxj@sjtu.edu.cn

BASICS Lab, Shanghai Jiao Tong University

Fall, 2016
Why study this course?

For credits?

To meet the requirement that every master student should take at least TWO English courses?

Being curious about Programming Languages.
Why study this course?

- For credits?
- To meet the requirement that every master student should take at least TWO English courses?
Why study this course?

- For credits?
- To meet the requirement that every master student should take at least TWO English courses?
- Being curious about Programming Languages.
Programming languages

- How many programming languages in the world?
Programming languages

- How many programming languages in the world?
  - https://en.wikipedia.org/wiki/List_of_programming_languages
  - http://99-bottles-of-beer.net/
Programming languages

- How many programming languages in the world?
  - https://en.wikipedia.org/wiki/List_of_programming_languages
  - http://99-bottles-of-beer.net/

- Which programming language is the best?
Programming languages

- How many programming languages in the world?
  - https://en.wikipedia.org/wiki/List_of_programming_languages
  - http://99-bottles-of-beer.net/

- Which programming language is the best?

- Which type of cars is the best?
Programming languages

- How many programming languages in the world?
  - https://en.wikipedia.org/wiki/List_of_programming_languages
  - http://99-bottles-of-beer.net/

- Which programming language is the best?
- Which type of cars is the best?
  - Winning an F1 race
  - Taking kids to the kindergarten
Where do we start?

Which programming language should we start with?

- A popular one or a classical one?
- We will focus mostly on functional programming languages because
  - mutation is discouraged which makes programs safe.
  - higher-order functions are very convenient.
  - ML-like languages: SML, OCaml, Haskell, ...
Where do we start?

Which programming language should we start with? A popular one or a classical one?
Where do we start?

Which programming language should we start with? A popular one or a classical one?

We will focus mostly on functional programming languages because

- mutation is discouraged which makes programs safe.
- higher-order functions are very convenient.
Where do we start?

Which programming language should we start with? A popular one or a classical one?

We will focus mostly on functional programming languages because
  ▶ mutation is discouraged which makes programs safe.
  ▶ higher-order functions are very convenient.

ML-like languages: SML, OCaml, Haskell, ...
Why types?

There are many excellent textbooks on Programming languages, such as:

- *Programming Language Pragmatics*, by Michael L. Scott
- *Practical Foundations of Programming Languages*, by Robert Harper
- *Programming Languages, Principles and Paradigms*, by Allen Tucker and Robert Noonan
- ...
Why types?

There are many excellent textbooks on Programming languages, such as:

▶ *Programming Language Pragmatics*, by Michael L. Scott
▶ *Practical Foundations of Programming Languages*, by Robert Harper
▶ *Programming Languages, Principles and Paradigms*, by Allen Tucker and Robert Noonan
▶ ...

We will focus on types because

▶ *most language features can be discussed in the framework of type theory, and*
▶ *language design goes hand-in-hand with type system design.*
Course overview

We will discuss in this course:

1. Structural induction
2. $\lambda$-calculus
3. Simple type systems and safety
4. Production types, sum types
5. Reference types, exceptions
6. Subtyping
7. Recursive types
8. Type reconstruction
9. Universal type and polymorphism
10. Type operators...
We will discuss in this course:
1. Structural induction
2. $\lambda$-calculus
3. simple type systems and safety
4. Production types, sum types
5. Reference types, exceptions
6. Subtyping
7. Recursive types
8. Type reconstruction
9. Universal type and polymorphism
10. type operators ...
Course overview

We will discuss in this course:
1. Structural induction
2. $\lambda$-calculus
3. simple type systems and safety
4. Production types, sum types
5. Reference types, exceptions
6. Subtyping
7. Recursive types
8. Type reconstruction
9. Universal type and polymorphism
10. type operators ...
After this course, you can ...

- Know how to judge a language.
- Learn to think about software in “PL” way and become a better programmer.
- Confidently pick up new languages for your lifetime.
- ...
Course policy

- Final exam: 70%
- Homework: 30%

TA: Yuwei Wang, wangywgg@sjtu.edu.cn

Course policy

- Final exam: 70%
- Homework: 30%
- TA:
  Yuwei Wang, wangywgg@sjtu.edu.cn
- Course homepage:
Outline

Introduction

Preliminaries

Untyped arithmetic expressions
  Abstract syntax
  Induction on terms
Semantics
  Booleans
  Numbers and Booleans
Types in Computer Science

Type systems is the most popular and best established lightweight formal methods.
Types in Computer Science

Type systems is the most popular and best established lightweight formal methods.

Definition
A *type system* is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.
Brief history

Types system (type theory) refers to a much broader field.

- **1900.** Formalized, Russell’s paradox
- **1925.** Simple theory of types, Ramsey
- **1940.** Simply typed $\lambda$-calculus, Church
- **1973.** Constructive type theory, Martin Löf
- **1992.** Pure type theory, Barendregt
- ...
Some definitions

- **Static type system.** Type checking during compile-time
- **Dynamic type system.** Type checking during run-time
Some definitions

- **Static type system.** Type checking during compile-time
- **Dynamic type system.** Type checking during run-time
- Static $\Rightarrow$ Conservative $\Rightarrow$ prove the absence of bad behaviours
Some definitions

- **Static type system.** Type checking during compile-time
- **Dynamic type system.** Type checking during run-time
- Static $\Rightarrow$ Conservative $\Rightarrow$ prove the absence of bad behaviours
- Incapable of finding all undesired program behaviours, e.g. divide by zero
Some definitions

- **Static type system.** Type checking during compile-time
- **Dynamic type system.** Type checking during run-time
- Static $\Rightarrow$ Conservative $\Rightarrow$ prove the absence of bad behaviours
- Incapable of finding all undesired program behaviours, e.g. divide by zero
- **Type checkers**
  - automatic: no manual interaction
  - type annotations
What types good for

- Detecting errors early.
- Maintenance tools.
- Abstracting
- Documentation
- Efficiency

Applications: network security, program analysis, theorem prover, database, xml, ...

Language design goes hand-in-hand with type system design.
Outline

Introduction

Preliminaries

Untyped arithmetic expressions
  Abstract syntax
  Induction on terms
Semantics
  Booleans
  Numbers and Booleans
An n-place relation is a set $R \subseteq S_1 \times S_2 \times \cdots \times S_n$.

A two-place relation $R$ on sets $S$ and $T$ is called a binary relation. We often write $s R t$ instead of $(s, t) \in R$. 
Relations

- An $n$-place relation is a set $R \subseteq S_1 \times S_2 \times \cdots \times S_n$.
- A two-place relation $R$ on sets $S$ and $T$ is called a binary relation. We often write $s R t$ instead of $(s, t) \in R$.
- The “mixfix” concrete syntax, e.g., $\Gamma \vdash s : T$ means “the triple $(\Gamma, s, T)$ in the typing relation”.
- $P$ is preserved by $R$ if whenever we have $s R t$ and $P(s)$, we also have $P(t)$.
Functions

- \( \text{dom}(R) \): the domain of a relation \( R \) on sets \( S \) and \( T \) is the set of elements \( s \in S \) such that \( (s, t) \in R \) for some \( t \).

- A relation \( R \) on sets \( S \) and \( T \) is called a partial function if, whenever \( (s, t_1) \in R \) and \( (s, t_2) \in R \), we have \( t_1 = t_2 \). If \( \text{dom}(R) = S \), then \( R \) is a total function.

- We write \( f(x) \uparrow \) to mean “\( f \) is undefined on \( x \),” and \( f(x) \downarrow \) to mean “\( f \) is defined on \( x \).”
Ordered sets

A binary relation $R$ on a set $S$ is

- **Reflexive**: $\forall x \in S. x R x$.
- **Transitive**: $x R y \land y R z$ implies $x R z$.
- **Symmetric**: $x R y$ implies $y R x$.
- **Antisymmetric**: $x R y \land y R x$ implies $x = y$. 

1. **Preorder** (or Quasi order): Reflexive + Transitive
2. **Equivalence**: Preorder + Symmetric
3. **Partial order**: Preorder + Antisymmetric
4. **Total order**: Partial order + ($\forall x, y \in S. x R y \lor y R x$)
5. **Well quasi order**: Preorder + (Any infinite sequence contains an increasing pair)
6. **Well founded order**: Preorder + (No infinite decreasing sequences)

Quiz: 1. Can Transitivity + Symmetry indicate Reflexivity?
2. Give examples of these orders.
Ordered sets

A binary relation $R$ on a set $S$ is

- **Reflexive**: $\forall x \in S. x R x$.
- **Transitive**: $x R y \land y R z$ implies $x R z$.
- **Symmetric**: $x R y$ implies $y R x$.
- **Antisymmetric**: $x R y \land y R x$ implies $x = y$.

1. **Preorder** (or Quasi order): Reflexive $+$ Transitive
2. **Equivalence**: Preorder $+$ Symmetric
3. **Partial order**: Preorder $+$ Antisymmetric
4. **Total order**: Partial order $+$ ($\forall x, y \in S. x R y \lor y R x$)
Ordered sets

A binary relation $R$ on a set $S$ is

- **Reflexive**: $\forall x \in S. x R x$.
- **Transitive**: $x R y \land y R z$ implies $x R z$.
- **Symmetric**: $x R y$ implies $y R x$.
- **Antisymmetric**: $x R y \land y R x$ implies $x = y$.

1. **Preorder** (or Quasi order): Reflexive + Transitive
2. **Equivalence**: Preorder + Symmetric
3. **Partial order**: Preorder + Antisymmetric
4. **Total order**: Partial order + $(\forall x, y \in S. x R y \lor y R x)$
5. **Well quasi order**: Preorder + (Any infinite sequence contains an increasing pair)
6. **Well founded order**: Preorder + (No infinite decreasing sequences)
Ordered sets

A binary relation $R$ on a set $S$ is

- **Reflexive**: $\forall x \in S. x R x$.
- **Transitive**: $x R y \land y R z$ implies $x R z$.
- **Symmetric**: $x R y$ implies $y R x$.
- **Antisymmetric**: $x R y \land y R x$ implies $x = y$.

1. **Preorder** (or Quasi order): Reflexive + Transitive
2. **Equivalence**: Preorder + Symmetric
3. **Partial order**: Preorder + Antisymmetric
4. **Total order**: Partial order + $(\forall x, y \in S. x R y \lor y R x)$
5. **Well quasi order**: Preorder + (Any infinite sequence contains an increasing pair)
6. **Well founded order**: Preorder + (No infinite decreasing sequences)

**Quiz**: 1. Can Transitivity + Symmetry indicate Reflexivity?
2. Give examples of these orders.
Inductions

- Ordinary induction on natural numbers
  If $P(0)$
  and for all $i$, $P(i)$ implies $P(i + 1)$,
  then $P(n)$ holds for all $n$.

- Complete induction on natural numbers
  If, for each natural number $k$,
  given $P(i)$ for all $i < k$
  we can show $P(k)$
  then $P(n)$ holds for all $n$. 
Outline

Introduction

Preliminaries

Untyped arithmetic expressions
  Abstract syntax
  Induction on terms
Semantics
  Booleans
  Numbers and Booleans
Untyped systems

- Untyped arithmetic expressions
- Untyped $\lambda$-calculus
Introduction

\[ t ::= \begin{array}{ll}
\text{true} & \text{constant true} \\
\text{false} & \text{constant false} \\
\text{if } t \text{ then } t \text{ else } t & \text{conditional} \\
0 & \text{constant zero} \\
\text{succ } t & \text{successor} \\
\text{pred } t & \text{predecessor} \\
\text{iszero } t & \text{zero test}
\end{array} \]

- BNF grammar
- \( t \) is metavariable.
- For simplicity, we use arabic numbers, e.g. 3 stands for \((\text{succ } (\text{succ } (\text{succ } 0)))\)
- Currently, \( \text{if } (\text{succ } 0) \text{ then true else } (\text{pred } 0) \) is a valid term.
Other ways to give syntax definition

The set of *terms* is the smallest set $T$ such that

- Inductively.
  - $\{true, false, 0\} \subseteq T$;
  - if $t_1 \in T$, then $\{\text{succ } t_1, \text{pred } t_1, \text{iszero } t_1\} \subseteq T$;
  - if $t_1, t_2, t_3 \in T$, then if $t_1$ then $t_2$ else $t_3 \in T$

- By inference rules

\[
\begin{align*}
\text{true} & \in T \\
\text{false} & \in T \\
0 & \in T \\
\hline
\text{succ } t_1 & \in T \\
\text{pred } t_1 & \in T \\
\text{iszero } t_1 & \in T \\
\hline
\end{align*}
\]

\[
\begin{align*}
t_1, t_2, t_3 & \in T \\
\hline
\text{if } t_1 \text{ then } t_2 \text{ else } t_3 & \in T
\end{align*}
\]
Other ways to give syntax definition, cont’d

▶ Concretely.
For each natural number \( i \), define \( S_i \) as follows:

\[
\begin{align*}
S_0 &= \emptyset \\
S_{i+1} &= \{\text{true, false, 0}\} \\
&\quad \cup \{\text{suc}\ t_1, \text{pred}\ t_1, \text{iszero}\ t_1 \mid t_1 \in S_i\} \\
&\quad \cup \{\text{if}\ t_1 \text{ then } t_2 \text{ else } t_3 \mid t_1, t_2, t_3 \in S_i\}
\end{align*}
\]

\[
S = \bigcup_i S_i
\]
Other ways to give syntax definition, cont’d

Concretely.

For each natural number \(i\), define \(S_i\) as follows:

\[
S_0 = \emptyset \\
S_{i+1} = \{\text{true, false, 0}\} \\
\quad \cup \{\text{succ } t_1, \text{pred } t_1, \text{iszero } t_1 \mid t_1 \in S_i\} \\
\quad \cup \{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \mid t_1, t_2, t_3 \in S_i\}
\]

\[
S = \bigcup_i S_i
\]

Lemma. \(S = T\).

Quiz. What if we change the concrete definition of \(S\) to

\[
S_0 = \{\text{true, false, 0}\} \\
S_{i+1} = \{\text{succ } t_1, \text{pred } t_1, \text{iszero } t_1 \mid t_1 \in S_i\} \\
\quad \cup \{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \mid t_1, t_2, t_3 \in S_i\}
\]
Inductive structure

For any $t \in T$, one of three things must be true about $t$:

1. $t$ is constant
2. $t$ has form $\text{succ } t_1, \text{pred } t_1$, or $\text{iszero } t_1$
3. $t$ has form $\text{if } t_1 \text{ then } t_2 \text{ else } t_3$.

Two ways to use this observation: inductive definition and inductive proof.
Inductive definition

\[
\begin{align*}
\text{consts}(\text{true}) &= \{\text{true}\} \\
\text{consts}(\text{false}) &= \{\text{false}\} \\
\text{consts}(0) &= \{0\} \\
\text{consts}(\text{succ } t_1) &= \text{consts}(t_1) \\
\text{consts}(\text{pred } t_1) &= \text{consts}(t_1) \\
\text{consts}(\text{iszero } t_1) &= \text{consts}(t_1) \\
\text{consts}(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) &= \text{consts}(t_1) \cup \text{consts}(t_2) \cup \text{consts}(t_3)
\end{align*}
\]
Inductive definition

\[
\begin{align*}
\text{consts}(\text{true}) &= \{ \text{true} \} \\
\text{consts}(\text{false}) &= \{ \text{false} \} \\
\text{consts}(0) &= \{ 0 \} \\
\text{consts}(\text{succ } t_1) &= \text{consts}(t_1) \\
\text{consts}(\text{pred } t_1) &= \text{consts}(t_1) \\
\text{consts}(\text{iszero } t_1) &= \text{consts}(t_1) \\
\text{consts}(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) &= \text{consts}(t_1) \cup \text{consts}(t_2) \cup \text{consts}(t_3)
\end{align*}
\]

**Quiz.**
1. Give an inductive definition of *size*, which is the size of the syntax tree of a term \( t \).
2. Give an inductive definition of *depth*, which is the height of the syntax tree of a term \( t \).
\[
\begin{align*}
size(\text{true}) &= 1 \\
size(\text{false}) &= 1 \\
size(0) &= 1 \\
size(\text{succ } t_1) &= size(t_1) + 1 \\
size(\text{pred } t_1) &= size(t_1) + 1 \\
size(\text{iszero } t_1) &= size(t_1) + 1 \\
size(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) &= size(t_1) + size(t_2) + size(t_3) + 1
\end{align*}
\]

**Lemma.** \(|consts(t)| \leq size(t)\).

**Proof.** By induction on the structure of \(t\).
Principles of induction on terms

- **Induction on depth:**
  If, for each term $s$, given $P(r)$ for all $r$ such that $\text{depth}(r) < \text{depth}(s)$, we can show $P(s)$, then $P(t)$ holds for all $t$.

- **Induction on size:**
  If, for each term $s$, given $P(r)$ for all $r$ such that $\text{size}(r) < \text{size}(s)$, we can show $P(s)$, then $P(t)$ holds for all $t$.

- **Structural Induction:**
  If, for each term $s$, given $P(r)$ for all immediate subterms $r$ of $s$, we can show $P(s)$, then $P(t)$ holds for all $t$. 
Outline

Introduction

Preliminaries

Untyped arithmetic expressions
  Abstract syntax
  Induction on terms

Semantics
  Booleans
  Numbers and Booleans
Semantics of languages

- **Operational semantics.** It specifies the behavior of PL by defining an *abstract machine*.
- **Denotational semantics.** The meaning of a term is taken to be some mathematical object (a number or a function).
- **Axiomatic semantics.** It takes the laws themselves as the definition of the language.
A toy language – Booleans

Syntax

\[ t ::= \]
\[ \text{true} \quad \text{constant true} \]
\[ \text{false} \quad \text{constant false} \]
\[ \text{if } t \text{ then } t \text{ else } t \quad \text{conditional} \]

\[ v ::= \]
\[ \text{true} \quad \text{true value} \]
\[ \text{false} \quad \text{false value} \]
Evaluation rules for Booleans

Evaluation

\[ \text{E-IfTrue} \quad \frac{\text{if true then } t_2 \text{ else } t_3}{t_2} \]

\[ \text{E-IfFalse} \quad \frac{\text{if false then } t_2 \text{ else } t_3}{t_3} \]

\[ \text{E-If} \quad \frac{t_1}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3} \]

Quiz.

Evaluate the following terms:

- \( \text{true} \)
- \( \text{if} (\text{if false then false else false}) \text{ then true else false} \)
Evaluation rules for Booleans

Evaluation

\[
\text{E-IfTrue} \quad \text{if true then } t_2 \text{ else } t_3 \rightarrow t_2
\]

\[
\text{E-IfFalse} \quad \text{if false then } t_2 \text{ else } t_3 \rightarrow t_3
\]

\[
\text{E-If} \quad t_1 \rightarrow t'_1
\]

\[
\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3
\]

\text{E-IfTrue} \text{ and } \text{E-IfFalse} \text{ are also called computation rules and } \text{E-If} \text{ is called congruence rule.}
Evaluation rules for Booleans

Evaluation

\[
\begin{align*}
\text{E-IfTrue} & \quad \text{if true then } t_2 \text{ else } t_3 \longrightarrow t_2 \\
\text{E-IfFalse} & \quad \text{if false then } t_2 \text{ else } t_3 \longrightarrow t_3 \\
\text{E-If} & \quad t_1 \longrightarrow t'_1 \\
& \quad \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3
\end{align*}
\]

E-IfTrue and E-IfFalse are also called computation rules and E-If is called congruence rule.

Quiz. Evaluate the following terms:

- true
- if (if false then false else false) then true else false
Derivation tree of One-step evaluation

Deﬁnitions:

\[ s \overset{\text{def}}{=} \text{if true then false else false} \]
\[ t \overset{\text{def}}{=} \text{if s then false else false} \]
\[ u \overset{\text{def}}{=} \text{if false then false else false} \]

Theorem 3.5.4: [Determinacy of one-step evaluation]

If \( t \overset{\rightarrow}{\rightarrow} t' \) and \( t \overset{\rightarrow}{\rightarrow} t'' \), then \( t' = t'' \).

Proof. By induction on the depth of the derivation tree.
Derivation tree of One-step evaluation

\[ s \overset{\text{def}}{=} \text{if true then false else false} \]
\[ t \overset{\text{def}}{=} \text{if s then false else false} \]
\[ u \overset{\text{def}}{=} \text{if false then false else false} \]

\[
\begin{array}{c}
\text{E-IfTrue} \quad \frac{s \rightarrow \text{false}}{\text{E-If}} \quad \frac{t \rightarrow u}{\text{E-If}} \\
\text{if t then false else false} \rightarrow \text{if u then false else false}
\end{array}
\]

**Theorem 3.5.4 [Determinacy of one-step evaluation]:** If \( t \rightarrow t' \) and \( t \rightarrow t'' \), then \( t' = t'' \).
Derivation tree of One-step evaluation

\[ s \overset{\text{def}}{=} \text{if true then false else false} \]
\[ t \overset{\text{def}}{=} \text{if } s \text{ then false else false} \]
\[ u \overset{\text{def}}{=} \text{if false then false else false} \]

\[
\begin{align*}
\text{E-If} & \quad \text{E-IfTrue} \\
& \quad \text{E-If} \\
& \quad \text{E-If} \\
\end{align*}
\]

Theorem 3.5.4 [Determinacy of one-step evaluation]: If \( t \rightarrow t' \) and \( t \rightarrow t'' \), then \( t' = t'' \).
Proof. By induction on the depth of the derivation tree.
Normal form and multi-step evaluation

- A term $t$ is in normal form if no evaluation rule can apply to it.

**Theorem 3.5.7:** Every value is in normal form.

**Theorem 3.5.8:** If $t$ is in normal form, then it is a value.
Normal form and multi-step evaluation

- A term $t$ is in normal form if no evaluation rule can apply to it.

**Theorem 3.5.7:** Every value is in normal form.

**Theorem 3.5.8:** If $t$ is in normal form, then it is a value.

- The multi-step evaluation relation $\rightarrow^*$ is the reflexive, transitive closure of $\rightarrow$.

**Theorem 3.5.11** [Uniqueness of normal forms]: If $t \rightarrow^* u$ and $t \rightarrow^* u'$ where $u, u'$ are normal forms, then $u = u'$.

**Theorem 3.5.12** [Termination of evaluation]: For every term $t$ there is some normal form $u$ such that $t \rightarrow^* u$. 
Arithmetic Expression

Syntax

\[ t ::= \begin{align*}
& \text{terms} \\
& \ldots \\
& 0 \quad \text{constant zero} \\
& \text{succ } t \quad \text{successor} \\
& \text{pred } t \quad \text{predecessor} \\
& \text{iszero } t \quad \text{zero test}
\end{align*} \]

\[ v ::= \begin{align*}
& \text{values} \\
& \ldots \\
& \text{nv} \quad \text{numeric value}
\end{align*} \]

\[ nv ::= \begin{align*}
& \text{numeric values} \\
& 0 \quad \text{zero value} \\
& \text{succ } nv \quad \text{successor value}
\end{align*} \]
Evaluation rules

Quiz. Give the definition of evaluation rules to guarantee

**Determinacy of one-step evaluation**: If $t \rightarrow t'$ and $t \rightarrow t''$, then $t' = t''$. 

### Evaluation Rules

- **E-PredZero**
  - $\text{pred } 0 \rightarrow 0$

- **E-IszeroZero**
  - $\text{iszero } 0 \rightarrow \text{true}$

- **E-PredSucc**
  - $\text{pred}(\text{succ } n) \rightarrow n$

- **E-IszeroSucc**
  - $\text{iszero}(\text{succ } n) \rightarrow \text{false}$

- **E-Pred**
  - $t_1 \rightarrow t_1'$
    - $\text{pred } t_1 \rightarrow \text{pred } t_1'$

- **E-Iszero**
  - $t_1 \rightarrow t_1'$
    - $\text{iszero } t_1 \rightarrow \text{iszero } t_1'$

- **E-Succ**
  - $t_1 \rightarrow t_1'$
    - $\text{succ } t_1 \rightarrow \text{succ } t_1'$
Evaluation rules

Quiz. Give the definition of evaluation rules to guarantee

[Determinacy of one-step evaluation]:
If \( t \rightarrow t' \) and \( t \rightarrow t'' \), then \( t' = t'' \).

Evaluation

\[
\begin{align*}
E-PredZero & \quad \text{pred } 0 \rightarrow 0 \\
E-IszeroZero & \quad \text{iszero } 0 \rightarrow \text{true} \\
E-PredSucc & \quad \text{pred } (\text{succ } \text{nv}) \rightarrow \text{nv} \\
E-IszeroSucc & \quad \text{iszero } (\text{succ } \text{nv}) \rightarrow \text{false} \\
E-Pred & \quad t_1 \rightarrow t'_1 \\
& \quad \text{pred } t_1 \rightarrow \text{pred } t'_1 \\
E-Iszero & \quad t_1 \rightarrow t'_1 \\
& \quad \text{iszero } t_1 \rightarrow \text{iszero } t'_1 \\
E-Succ & \quad t_1 \rightarrow t'_1 \\
& \quad \text{succ } t_1 \rightarrow \text{succ } t'_1
\end{align*}
\]
Normal form and stuckness

What if we change \( E \text{-PREDSUCC} \) to \( \text{pred} (\text{succ } t) \longrightarrow t \)? Does it still satisfy [Determinacy of one-step transition]?
Normal form and stuckness

- What if we change $E\text{-PredSucc}$ to $\text{pred (succ } t) \rightarrow t$? Does it still satisfy [Determinacy of one-step transition]?
- Note there are meaningless terms, such as if 0 then (succ true) else (iszeoro false).
Normal form and stuckness

- What if we change $E$-PredSucc to $\text{pred} \ (\text{succ} \ t) \rightarrow t$? Does it still satisfy [Determinacy of one-step transition]?
- Note there are meaningless terms, such as
  if 0 then (succ true) else (iszeoro false).
- A term $t$ is stuck if it is in normal form but not a value.
Conclusion

- Types are very important for PLs.
- This course will give a full view of type systems from the simplest one to full-fledged one.
- Fundamental concepts for PLs:
  - syntax, defined inductively, concretely, ...
  - inductive proofs are very important for PLs, especially, structural induction.
  - operational semantics plays more and more important roles. We define evaluation rules by using operational transitions.
- Properties such as [Determinacy of one-step evaluation], [Uniqueness of normal forms], and [Termination] are important for a good language design.
Homework

- 3.3.4, 3.5.10, 3.5.13, 3.5.18.
- Install OCaml and get familiar with this language.
  http://ocaml.org/