Untyped λ-calculus
Nameless representation

Advanced Compiler Construction and Program Analysis

Lecture 1

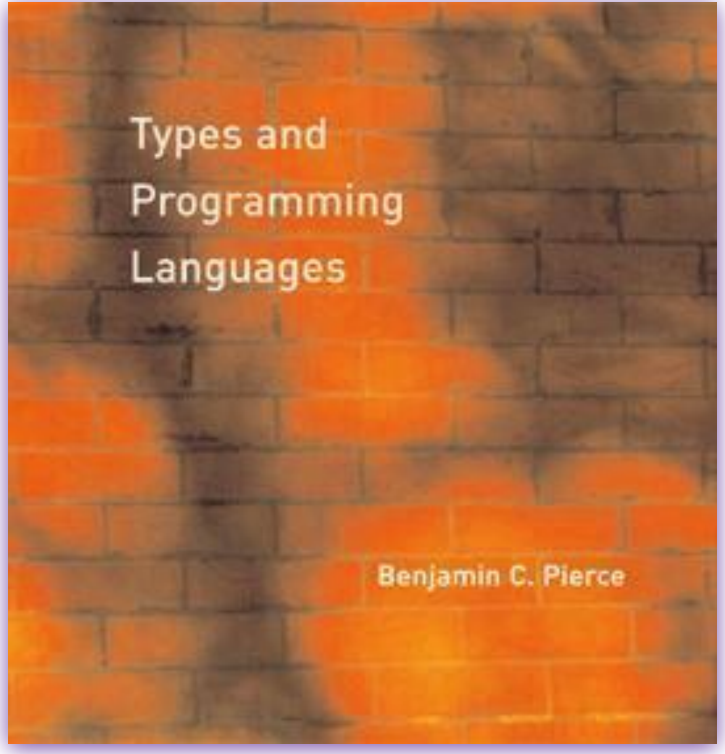
Innopolis University, Spring 2022

The topics of this lecture are covered in detail in...

Benjamin C. Pierce.

Types and Programming Languages

MIT Press 2002



Untyped Arithmetic Expressions. Syntax

```
t ::=                                  terms
    true                               constant true
    false                              constant false
    if t then t else t                 conditional
```

Untyped 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*

Untyped Expressions. Induction on terms

```
consts(true)        = {true}
```

Untyped Expressions. Induction on terms

```
consts(true)        = {true}
consts(false)       = {false}
consts(0)        = {0}
```

Untyped Expressions. Induction on terms

```
consts(true)        = {true}
consts(false)       = {false}
consts(0)       = {0}
consts(succ t)      = consts(t)
```

Untyped Expressions. Induction on terms

```
consts(true)       = {true}
consts(false)      = {false}
consts(0)        = {0}
consts(succ t)     = consts(t)
consts(pred t)     = consts(t)
consts(iszero t)  = consts(t)
```

Untyped Expressions. Induction on terms

$$\begin{aligned}
\text{consts}(\textbf{true}) &= \{\textbf{true}\} \\
\text{consts}(\textbf{false}) &= \{\textbf{false}\} \\
\text{consts}(\textbf{0}) &= \{\textbf{0}\} \\
\text{consts}(\textbf{succ } t) &= \text{consts}(t) \\
\text{consts}(\textbf{pred } t) &= \text{consts}(t) \\
\text{consts}(\textbf{iszero } t) &= \text{consts}(t) \\
\text{consts}(\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3) & \\
&= \text{consts}(t_1) \cup \text{consts}(t_2) \cup \text{consts}(t_3)
\end{aligned}$$

Untyped Expressions. Induction on terms

$$
\begin{aligned}
&\texttt{size}(\textbf{true}) = 1\\
&\texttt{size}(\textbf{false}) = 1\\
&\texttt{size}(\textbf{0}) = 1\\
&\texttt{size}(\textbf{succ } t) = \texttt{size}(t) + 1\\
&\texttt{size}(\textbf{pred } t) = \texttt{size}(t) + 1\\
&\texttt{size}(\textbf{iszero } t) = \texttt{size}(t) + 1\\
&\texttt{size}(\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3)\\
&\quad = \texttt{size}(t_1) + \texttt{size}(t_2) + \texttt{size}(t_3) + 1
\end{aligned}
$$

Untyped Expressions. Induction on terms

```
depth(true)         = 1
depth(false)        = 1
depth(0)            = 1
depth(succ t)       = depth(t) + 1
depth(pred t)       = depth(t) + 1
depth(iszero t)= depth(t) + 1
depth(if t₁ then t₂ else t₃)
  = max(depth(t₁), depth(t₂), depth(t₃)) + 1
```

Untyped Expressions. Induction on terms (proof)

Exercise 1.1. Prove the following statement:

The number of distinct constants in a term `t` *is no greater than the size of* `t`*:*

$$|\mathtt{consts(t)}| \le \mathtt{size(t)}$$

Untyped Expressions. Induction on terms (proof)

The number of distinct constants in a term `t` *is no greater than the size of* `t`*:* `|consts(t)| ≤ size(t)`

Proof.

Principles of induction

Theorem 1.2 (Induction on depth).

Suppose *P* is a predicate on terms.

If, for each term s,

given *P*(r) for all r such that depth(r) < depth(s)

we can show *P*(s),

then *P*(s) holds for all s.

$$(\forall s.(\forall r.(\mathsf{depth}(r) < \mathsf{depth}(s)) \implies P(r)) \implies P(s)) \implies \forall s.P(s)$$

Principles of induction

Theorem 1.3 (Induction on size).

Suppose *P* is a predicate on terms.

If, for each term s,

given *P*(r) for all r such that size(r) < size(s)

we can show *P*(s),

then *P*(s) holds for all s.

$$(\forall s.(\forall r.(\mathsf{size}(r) < \mathsf{size}(s)) \implies P(r)) \implies P(s)) \implies \forall s.P(s)$$

Principles of induction

Theorem 1.4 (Structural induction).

Suppose P is a predicate on terms.

If, for each term s,

given P(r) for all immediate subterms of s

we can show P(s),

then P(s) holds for all s.

Semantic styles

- **Operational semantics** specifies behaviour, typically by providing some machine that “executes” expressions.
- **Denotational semantics** provides some abstract interpretation (ignoring some details) in some domain.
- **Axiomatic semantics** focuses on reasoning about properties of programs (e.g. pre- and post-conditions and invariants).

Boolean Expressions

```
t ::=                                   terms
    true                                constant true
    false                               constant false
    if t then t else t                  conditional

v ::=                                   values
   true                                 constant true
   false                                constant false
```

Booleans. One-step evaluation

$$\texttt{t} \longrightarrow \texttt{t'}$$

Booleans. One-step evaluation

$$\texttt{t} \longrightarrow \texttt{t'}$$

$$\texttt{if true then } t_2 \texttt{ else } t_3 \quad \longrightarrow \quad t_2$$

Booleans. One-step evaluation

$\texttt{t} \longrightarrow \texttt{t'}$

$\texttt{if true then } t_2 \texttt{ else } t_3 \longrightarrow t_2$

$\texttt{if false then } t_2 \texttt{ else } t_3 \longrightarrow t_3$

Booleans. One-step evaluation

$\mathtt{t} \longrightarrow \mathtt{t'}$

$$\mathbf{if}\ \mathbf{true}\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \longrightarrow t_2$$

$$\mathbf{if}\ \mathbf{false}\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \longrightarrow t_3$$

$$\frac{t_1 \longrightarrow u_1}{\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \longrightarrow \mathbf{if}\ u_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3}$$

Boolean Expressions. Evaluation example

```
if false then true else
  (if true then false else true)
```

$\longrightarrow$?

Boolean Expressions. Evaluation example

```
if false then true else
  (if true then false else true)
```

$\longrightarrow$ `if true then false else true`

$\longrightarrow$ `?`

Boolean Expressions. Evaluation example

```
if false then true else
  (if true then false else true)
```

$\longrightarrow$ `if true then false else true`

$\longrightarrow$ `false`

Multi-step evaluation

Definition 1.5. The ***multi-step evaluation*** relation

$$\boxed{t \longrightarrow^* u}$$

is the reflexive, transitive closure of one-step evaluation.

That is, it is the smallest relation, such that

1. if $\boxed{t \longrightarrow u}$ then $\boxed{t \longrightarrow^* u}$
2. for any term t, we have $\boxed{t \longrightarrow^* t}$
3. if $\boxed{t \longrightarrow^* u}$ and $\boxed{u \longrightarrow^* s}$ then $\boxed{t \longrightarrow^* s}$

Multi-step evaluation example

```
if false then true else
  (if true then false else true)
```

$\longrightarrow^*$ `false`

Numbers. New syntactic forms

```
t ::= ...
   0
   succ t
   pred t
```

terms
constant zero
successor
predecessor
zero test

```
v ::= ... | nv
```

values

```
nv ::=
  0
  succ nv
```

numeric values
zero value
successor value

Numbers. New evaluation rules

$$\texttt{t} \longrightarrow \texttt{t'}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{u}_1}{\texttt{succ t}_1 \longrightarrow \texttt{succ u}_1}$$

Numbers. New evaluation rules

$$\mathtt{t} \longrightarrow \mathtt{t'}$$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{u_1}}{\mathtt{succ}\ \mathtt{t_1} \longrightarrow \mathtt{succ}\ \mathtt{u_1}}$$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{u_1}}{\mathtt{pred}\ \mathtt{t_1} \longrightarrow \mathtt{pred}\ \mathtt{u_1}}$$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{u_1}}{\mathtt{iszero}\ \mathtt{t_1} \longrightarrow \mathtt{iszero}\ \mathtt{u_1}}$$

Numbers. New evaluation rules

$$\mathtt{t} \longrightarrow \mathtt{t'}$$

$$\frac{t_1 \longrightarrow u_1}{\mathtt{succ}\ t_1 \longrightarrow \mathtt{succ}\ u_1}$$

$$\frac{t_1 \longrightarrow u_1}{\mathtt{pred}\ t_1 \longrightarrow \mathtt{pred}\ u_1}$$

$$\frac{t_1 \longrightarrow u_1}{\mathtt{iszero}\ t_1 \longrightarrow \mathtt{iszero}\ u_1}$$

$$\mathtt{iszero}\ 0 \longrightarrow \mathtt{true}$$

Numbers. New evaluation rules

$$\texttt{t} \longrightarrow \texttt{t'}$$

$$\frac{t_1 \longrightarrow u_1}{\textbf{succ}\ t_1 \longrightarrow \textbf{succ}\ u_1}$$

$$\textbf{iszero}\ 0 \longrightarrow \textbf{true}$$

$$\textbf{iszero}\ (\textbf{succ}\ t) \longrightarrow \textbf{false}$$

$$\frac{t_1 \longrightarrow u_1}{\textbf{pred}\ t_1 \longrightarrow \textbf{pred}\ u_1}$$

$$\frac{t_1 \longrightarrow u_1}{\textbf{iszero}\ t_1 \longrightarrow \textbf{iszero}\ u_1}$$

Numbers. New evaluation rules

$\texttt{t} \longrightarrow \texttt{t'}$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{u}_1}{\texttt{succ t}_1 \longrightarrow \texttt{succ u}_1}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{u}_1}{\texttt{pred t}_1 \longrightarrow \texttt{pred u}_1}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{u}_1}{\texttt{iszero t}_1 \longrightarrow \texttt{iszero u}_1}$$

$\texttt{iszero 0} \longrightarrow \texttt{true}$

$\texttt{iszero (succ t)} \longrightarrow \texttt{false}$

$\texttt{pred 0} \longrightarrow \texttt{0}$

$\texttt{pred (succ t)} \longrightarrow \texttt{t}$

Stuck terms

When formalizing semantics, we have to consider behaviour of *all terms*. In particular, we have to consider terms like **`pred 0`** and **`succ false`**.

If a term is not a value, but also cannot be reduced by any of the evaluation rules, we call this term a ***stuck term***.

Definition. A closed term **`t`** is ***stuck*** if it is in normal form, but is not a value.

Untyped λ-calculus. Syntax

```
t ::=
    x
    λx.t
    t t

v ::=
   λx.t
```

terms
variable
abstraction
application

values
abstraction value

Untyped λ-calculus. Evaluation rules

$$t \longrightarrow t'$$

$$\frac{t_1 \longrightarrow u_1}{t_1\ t_2 \longrightarrow u_1\ t_2}$$

$$\frac{t_2 \longrightarrow u_2}{t_1\ t_2 \longrightarrow t_1\ u_2}$$

$$(\lambda x.t_1)\ t_2 \longrightarrow [x \mapsto t_2]t_1$$

Untyped λ-calculus. Substitution

$[x \mapsto s]t$

$$[x \mapsto s]x = s$$

$$[x \mapsto s]y = y \quad \text{if } y \neq x$$

$$[x \mapsto s](\lambda x.t) = \lambda y.[x \mapsto s]t$$

$$\text{if } y \neq x \text{ and } y \text{ is not free in } s$$

$$[x \mapsto s](t_1\ t_2) = [x \mapsto s]t_1\ [x \mapsto s]t_2$$

Untyped λ-calculus. Alpha-equivalence

$$\lambda z.\ \lambda x.\ \lambda y.\ x\ (y\ z)$$

is the alpha-equivalent to

$$\lambda a.\ \lambda b.\ \lambda c.\ b\ (c\ a)$$

Names of bound variables do not matter!

Nameless representation of terms

Working “up to renaming of bound variables” is good when reasoning on paper, but is not very practical when implementing a compiler. Some options are:

1. Use symbolic names are perform automatic renaming whenever name conflicts arise.

2. Use symbolic names, but introduce a condition that all bound variables have to use unique names, different from each other and any free variables. *Barendregt convention.*

3. Devise “canonical” representation so that renaming is not required.

Nameless untyped λ-calculus. Syntax

```
t ::=
    n
    λt
    t t
```

terms
variable index
abstraction
application

```
v ::=
   λt
```

values
abstraction value

Nameless syntax. Example

```
λx.λy. x (y x)
```

corresponds to

```
λλ 1 (0 1)
```

Nameless syntax. Example

```
λx.λy. x (y x)
```

corresponds to

```
λλ 1 (0 1)
```

Nameless syntax. Exercise

Exercise 1.2. Write down nameless term corresponding to each of the following terms:

1. c0 = λs. λz. z
2. c2 = λs. λz. s (s z)
3. plus = λm. λn. λs. λz. m s (n z s)
4. fix = λf. (λx. f (λy. (x x) y)) (λx. f (λy. (x x) y))
5. foo = (λx. (λx. x)) (λx. x)

Nameless syntax. Exercise

```
1.  c0 = λs. λz. z
2.  c2 = λs. λz. s (s z)
3.  plus = λm. λn. λs. λz. m s (n z s)
4.  fix = λf. (λx. f (λy. (x x) y)) (λx. f (λy. (x x) y))
5.  foo = (λx. (λx. x)) (λx. x)
```

Nameless λ-calculus. Evaluation

$$t \longrightarrow t'$$

$$\frac{t_1 \longrightarrow u_1}{t_1\ t_2 \longrightarrow u_1\ t_2}$$

$$\frac{t_2 \longrightarrow u_2}{t_1\ t_2 \longrightarrow t_1\ u_2}$$

$$(\lambda t_1)\ t_2 \longrightarrow [0 \mapsto t_2]t_1$$

Nameless λ-calculus. Substitution

$[n \mapsto s]t$

$$
\begin{aligned}
[n \mapsto s]n &= s \\
[n \mapsto s]m &= m \quad \text{if } n \neq m \\
[n \mapsto s](\lambda t) &= \lambda[n+1 \mapsto \uparrow(s)]t \\
[n \mapsto s](t_1\ t_2) &= [n \mapsto s]t_1\ [n \mapsto s]t_2
\end{aligned}
$$

Nameless λ-calculus. Shifting

$\uparrow(t)$

$$
\begin{aligned}
\uparrow(k,\ n) &= n && \text{if } n < k \\
\uparrow(k,\ n) &= n+1 && \text{if } n \geq k \\
\uparrow(k,\ \lambda t) &= \lambda\uparrow(k+1, t) \\
\uparrow(k,\ t_1\ t_2) &= \uparrow(k,\ t_1)\ \uparrow(k,\ t_2)
\end{aligned}
$$

$$\uparrow(t) = \uparrow(0,\ t)$$

Example: inlining method call in Java

```
class A {
   int x, y;
   …
   bool f(int x) { return (x + y) > 0; }

   int g(int y) {
      for (int x = 0; x < 10; x++) {
         if (f(x + y)) { return x; }
      }
      return x;
   }
}
```

Example: inlining method call in Java

```
class A {
   int x, y;
   ...
   bool f(int x) { return (x + y) > 0; }

   int g(int y) {
      for (int x = 0; x < 10; x++) {
        if (f(x + y)) { return x; }
      }
      return x;
   }
}
```

Example: inlining method call in Java

```
class A {
   int x, y;
   ...
   bool f(int) { return (0 + y) > 0; }

   int g(int) {
      for (int = 0; 0 < 10; 0++) {
         if (f(0 + 1)) { return 0; }
      }
      return x;
   }
}
```

Example: inlining method call in Java

```
class A {
   int x, y;
   ...
   bool f = λ ((0 + y) > 0)

   int g(int) {
      for (int = 0; 0 < 10; 0++) {
         if (f(0 + 1)) { return 0; }
      }
      return x;
   }
}
```

Example: inlining method call in Java

```
class A {
    int x, y;
    ...
    bool f = λ ((0 + y) > 0)

    int g(int) {
      for (int = 0; 0 < 10; 0++) {
        if ((λ ((0 + y) > 0))(0 + 1)) { return 0; }
       }
       return x;
    }
}
```

Example: inlining method call in Java

```
class A {
    int x, y;
    ...
    bool f = λ ((0 + y) > 0)

    int g(int) {
        for (int = 0; 0 < 10; 0++) {
            if ([0 ↦ (0 + 1)]((0 + y) > 0)) { return 0; }
        }
        return x;
    }
}
```

Example: inlining method call in Java

```
class A {
   int x, y;
   ...
   bool f = λ ((0 + y) > 0)

   int g(int) {
      for (int = 0; 0 < 10; 0++) {
         if (((0 + 1) + y) > 0) { return 0; }
      }
      return x;
   }
}
```

Example: inlining method call in Java

```
class A {
   int x, y;
   ...
   bool f(int x) { return (x + y) > 0; }

   int g(int y1) {
      for (int x1 = 0; x1 < 10; x1++) {
        if (((x1 + y1) + y) > 0) { return x1; }
      }
      return x;
   }
}
```

Summary

- ❑ Untyped arithmetic expressions
- ❑ Principles of induction
- ❑ Untyped λ-calculus
- ❑ Nameless representation

Summary

- ❑ Untyped arithmetic expressions
- ❑ Principles of induction
- ❑ Untyped λ-calculus
- ❑ Nameless representation

See you next time!