Programming Languages
A Journey into Abstraction and Composition

Type Systems of Programming Languages

Prof. Dr. Guido Salvaneschi

Programming Group

School of Computer Science

University of St.Gallen

Errors in Programs

We write and execute programs.

We expect certain behaviors.

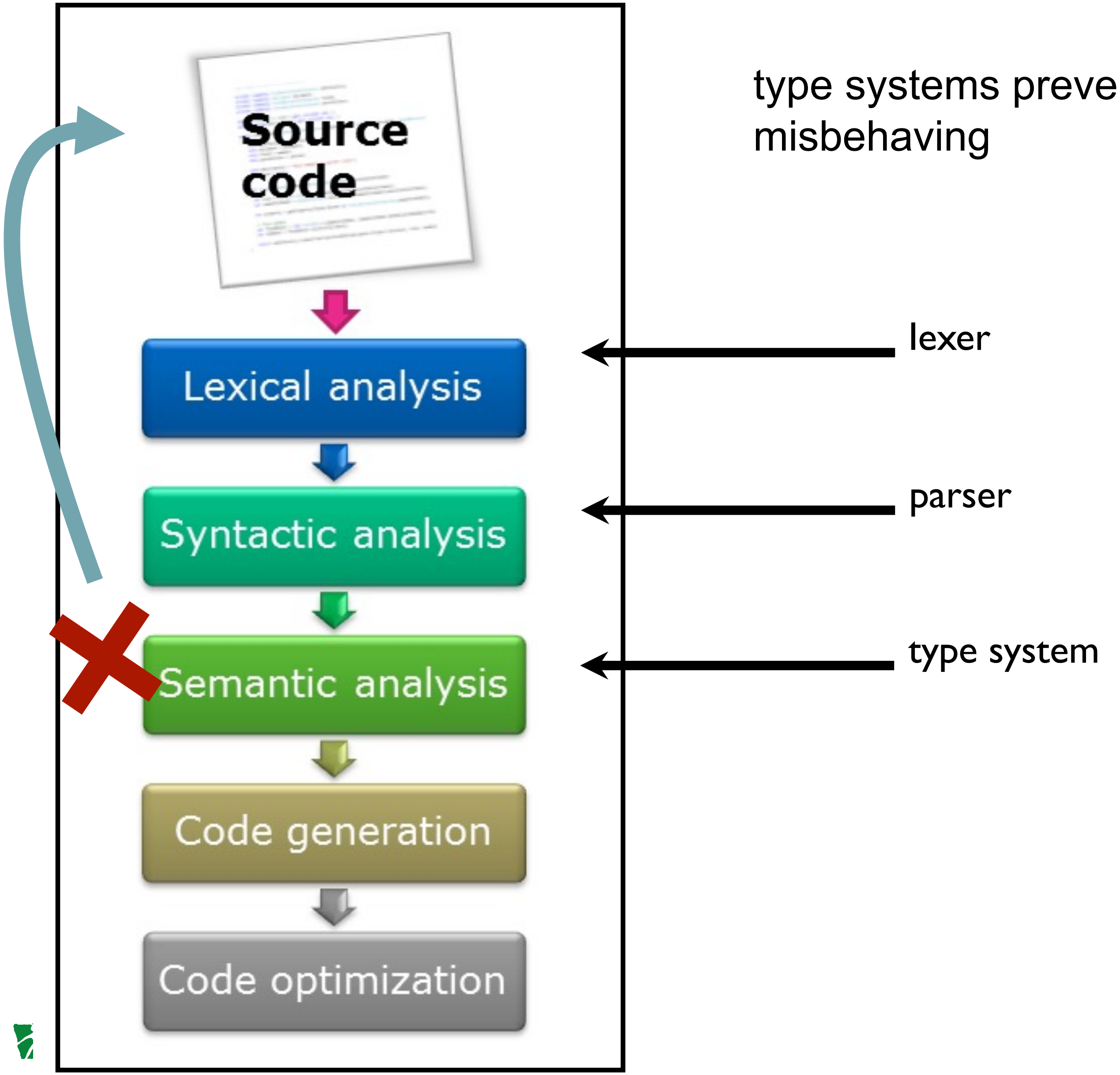
But programs can go wrong! you know?

```
*** STOP: 0x00000019 (0x00000000,0xC00E0FF0,0xFFFFEFD4,0xC0000000)
BAD_POOL_HEADER

CPUID:GenuineIntel 5.2.c irql:1f  SYSVER 0xf0000565

Dll Base DateStmp - Name                Dll Base DateStmp - Name
80100000 3202c07e - ntoskrnl.exe        80010000 31ee6c52 - hal.dll
80001000 31ed06b4 - atapi.sys           80006000 31ec6c74 - SCSIPORT.SYS
802c6000 31ed06bf - aic78xx.sys         802cd000 31ed237c - Disk.sys
802d1000 31ec6c7a - CLASS2.SYS          8037c000 31eed0a7 - Ntfs.sys
fc698000 31ec6c7d - Floppy.SYS          fc6a8000 31ec6ca1 - Cdrom.SYS
fc90a000 31ec6df7 - Fs_Rec.SYS          fc9c9000 31ec6c99 - Null.SYS
fc864000 31ed868b - KSecDD.SYS          fc9ca000 31ec6c78 - Beep.SYS
fc6d8000 31ec6c90 - i8042prt.sys        fc86c000 31ec6c97 - mouclass.sys
fc874000 31ec6c94 - kbdclass.sys        fc6f0000 31f50722 - VIDEOPORT.SYS
feffa000 31ec6c62 - mga_mil.sys         fc890000 31ec6c6d - vga.sys
fc708000 31ec6ccb - Msfs.SYS            fc4b0000 31ec6cc7 - Npfs.SYS
fefbc000 31eed262 - NDIS.SYS            a0000000 31f954f7 - win32k.sys
fefa4000 31f91a51 - mga.dll             fec31000 31eedd07 - Fastfat.SYS
feb8c000 31ec6e6c - TDI.SYS             feaf0000 31ed0754 - nbf.sys
feacf000 31f130a7 - tcpip.sys           feab3000 31f50a65 - netbt.sys
fc550000 31601a30 - el59x.sys           fc560000 31f8f864 - afd.sys
fc718000 31ec6e7a - netbios.sys         fc858000 31ec6c9b - Parport.sys
fc870000 31ec6c9b - Parallel.SYS        fc954000 31ec6c9d - ParVdm.SYS
fc5b0000 31ec6cb1 - Serial.SYS          fea4c000 31f5003b - rdr.sys
fea3b000 31f7a1ba - mup.sys             fe9da000 32031abe - srv.sys

Address  dword dump   Build [1381]                            - Name
fec32d84 80143e00 80143e00 80144000 ffdff000 00070b02          - KSecDD.SYS
801471c8 80144000 80144000 ffdff000 c03000b0 00000001          - ntoskrnl.exe
801471dc 80122000 f0003fe0 f030eee0 e133c4b4 e133cd40          - ntoskrnl.exe
80147304 803023f0 0000023c 00000034 00000000 00000000          - ntoskrnl.exe

Restart and set the recovery options in the system control panel
or the /CRASHDEBUG system start option.
```


NCR
has been detected and windows has been shut down to
your computer.
PAGE_FAULT_IN_NONPAGED_AREA
If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, follow
these steps:
Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any windows updates you might need.
problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup Options, and then
select Safe Mode.
information:
*** STOP: 0x00000050 (0x800005F2,0x00000000,0x804E83C8,0x00000000)
Beginning dump of physical memory
memory dump complete.
your system administrator or technical support group for further
assistance.
1
2 ABC
3 DEF
CANCEL
4 GHI
5 JKL
6 MNO
CLEAR

Lufthansa
Flug / flight
LH
nach-über /
Seoul
Z 50

Compiler



type systems prevent programs from misbehaving

Robin Milner

1934 - 2010

“Well-typed programs cannot go wrong”
-- Robin Milner, 1978

Robin Milner received Turing award 1991 for

- Logic for computable functions (LCF)
- Programming language ML
- Calculus of communicating systems (CCS), pi-calculus

Let’s Look at a Few Existing Type Systems

What kind of type systems do you know?

Which guarantees do the provide?

Let’s Look at a Few Existing Type Systems

```
class Marker {
    int position = 0;
    public void move() { position += 1; }
}

Marker m = new Marker();
m.move();
m.move();
m.move();
m.pause();
```

class types

Let’s Look at a Few Existing Type Systems

length :: [a] -> Int
length [] = 0
length (x : xs) = 1 + length xs

map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x : xs) = f x : map f xs

length [1,2,3,4]
length [“a”, “b”]

map int-to-string [1,2,3,4]
map int-to-string [“a”, “b”]

function types

Let’s Look at a Few Existing Type Systems

```
data Tree a =
        | Leaf a
        | Node (Tree a) (Tree a)

t = Node (Leaf 3) (Node (Leaf 4) (Leaf 19))

treesum :: Tree Int -> Int
treesum t = case t of
        Leaf a => a
        Node t1 t2 => treesum t1 + treesum t2

t2 = Node (Leaf 3) (Node “leaf” (Leaf 5))
```

algebraic data types

Let's Look at a Few Existing Type Systems

```
class FastMarker extends Marker {
        public void moveFast(int steps) {
                for (int i = 0; i < steps; i++) move();
    –   }
}

FastMarker m1 = new FastMarker();
m1.move();   m1.moveFast();

Marker m2 = new FastMarker();
m2.move();
m2.moveFast();
```

$\boxed{\textbf{subtyping}}$

Let's Look at a Few Existing Type Systems

```
String[] strings = new String {"a",  "b", "c"};
Object[] objects = strings;  //  'objects' and 'strings' are exactly the same

// Some legitimate operations
Object o = strings[0];
objects[0] = "abc";       //  strings[0] equals "abc"

objects[0] = new Integer(5);

int size = 0;
for (String s : strings)
        size += s.length();
```

run-time error


PANIC!
AAAARRGGH!
SCARED
DANGER
HAZARD!
EMERGENCY!
CRISIS!
URGENT!
FREAK OUT!
HELP!
DISTRESS!
CATASTROPHE!

Let’s Look at a Few Existing Type Systems

String[] strings = new String {“a”, “b”, “c”};

Object[] objects = strings; // ‘objects’ and ‘strings’ are exactly the same

Object o = strings[0];

objects[0] = “abc”;   // strings[0] == “abc”

objects[0] = new Integer(5);

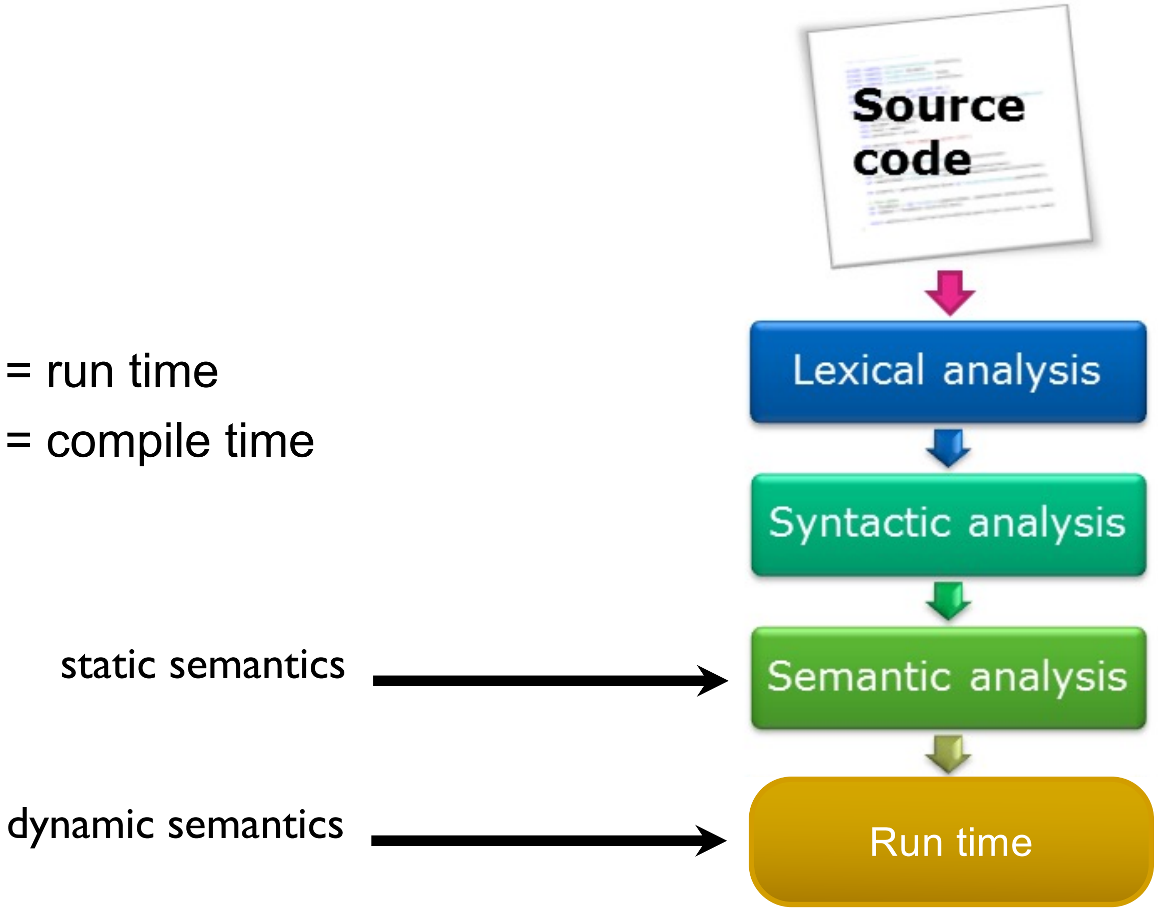
The type system accepts the program but program does go wrong at run time!

Types

Program behavior defined
by language semantics

Distinguish:

- dynamic semantics = run time
- static semantics = compile time



What is a Type?

A **type** is a collection of computable values that share some structural property

Examples
- Integers
- Strings
- int bool
- (int -> int) ->bool

“Non-examples”
- 3, true, x.x
- Even integers
- {f:int -> int | if x>3 then f(x) > x*(x+1)}

Distinction between sets that are types and sets that are not types is language-dependent

Strong vs. Weak Typing

A language is **strongly typed** if its type system allows all type errors in a program to be detected either at compile time or at run time.

- A strongly typed language can be statically or dynamically typed!

Compile vs. run-time checking

Type-checking at **compile** time: C, ML

Type-checking at **run** time: Perl, JavaScript

Java does both

- **Widening** conversions always valid: performed implicitly.
- **Narrowing** conversions. Validity cannot be determined at compile time; they require an explicit cast and may throw **`ClassCastException`**.
- Conversions between incompatible types are compile-time errors.

Basic tradeoffs

- Both prevent type errors
- Run-time checking slows down execution
- Compile-time checking restricts program flexibility
  - JavaScript array: elements can have different types
  - ML list: all elements must have same type

Typing

Static: type checking by analysis of program

- Compiler/interpreter verifies that type errors cannot occur
- C, C++, F#, Haskell, Java, OCaml

Dynamic: type checking done at run-time

- Runtime detects type errors and reports them.
- Usually requires extra tag information for values in memory.
- JavaScript, LISP, Matlab, PHP, Python, Ruby

Some mixed features, e.g., Java **`instanceof`**: most checking done at compile time, but also checking at run time


DYNAMIC
TYPING
STATIC
TYPING
DONE!

Typing

Manifest: type information supplied in source code

- C, C++, Java (Do not confuse with Scala Manifests)

Implicit: type information not supplied in source code

- Dynamic typing: LISP, Python, Ruby, PHP
- Type inference: Haskell, OCaml, ML, Scala

Usually a spectrum

No reasonable language requires to write the type of 5 in **`x: int = 5`**

Type Inference: Examples in Java

Type inference and instantiation of Generic classes

```
Map<String, List<String>> myMap =
        new HashMap<String, List<String>>();
```

You can substitute the parameterized type of the constructor with an empty set of type parameters (<>):

```
Map<String, List<String>> myMap = new HashMap<>();
```

Type Inference and Generic Methods

```
public static <U> void addBox(U u,
      java.util.List<Box<U>> boxes) {
    Box<U> box = new Box<>();
    box.set(u);
    boxes.add(box);
  }
BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);
BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);
```


http://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html

Type Inference

Best known in functional languages

- Especially useful in managing the types of higher-order functions
- But starting to appear in mainstream languages, e.g., C++11:

  ```
  auto x = e;
  ```

  declares variable x, initialized with expression e, and type of x is automatically inferred

Invented by Robin Milner for SML

(Hindley–Milner inference algorithm)

Type Inference

ML: Global Type inference

```
fun fac 0 = 1
  | fac n = n * fac (n - 1)
```

```
fun fac (0 : int) : int = 1
  | fac (n : int) : int = n * fac (n - 1)
```

Scala: Local type inference

```
def factorial(n: Int): Int = {
  if (n == 0)
      return 1
  else
      return n * factorial(n-1)
```

A Type Checker: First Steps

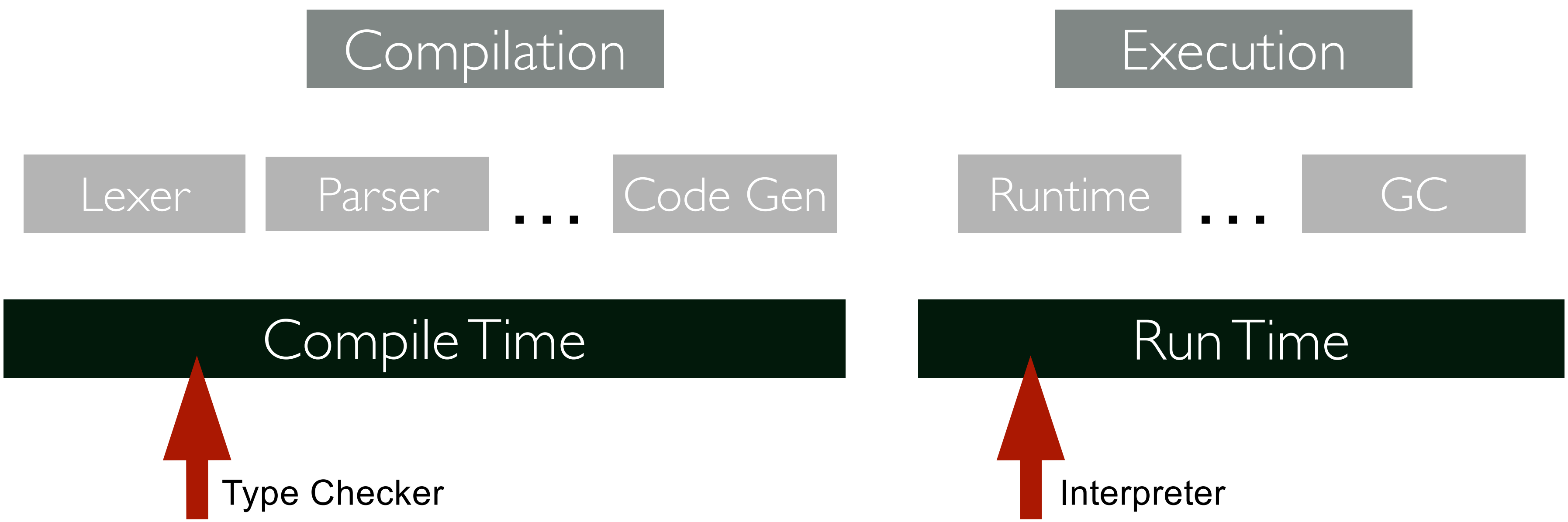
For arithmetic expressions AE.

For arithmetic expressions and functions FAE.

Where are we?

There is a conceptual difference:

- Interpreters model the **execution** semantics
- Type checking belongs to the compilation phase



A Type Checker: First Steps

An interpreter is a function, e.g., **`Expr -> Value`**

What about a type checker?

A Type Checker: First Steps

An interpreter is a function

- From a and expression to a value, e.g., FAE -> Value

What about a type checker?

- Conceptually: Expr -> Type, for example FAE -> Int or FAE -> Bool
- The type checker gets a program and returns its type.
- If the type checker cannot type the program i.e., the program is not correct (for what typing concerns) it throws an error.

In practice the type checker throws an Exception
if the type checking does not succeed.

A Type Checker for AE

What can go wrong in the typing of such programs?

```
def typeOf (e: FE): Type = e match {

  case Num(n) => TNum()

  case Add(lhs, rhs)
    if typeOf(lhs) == TNum() &&
       typeOf(rhs) == TNum() => TNum()
  …
  }
```

See the Interpreter

A Type Checker for BAE

Let's add Booleans

```
sealed abstract class FE
case class Num(n: Int) extends FE
case class Bool(b: Boolean) extends FE
case class Add(lhs: FE, rhs: FE) extends FE
case class Sub(lhs: FE, rhs: FE) extends FE
case class And(lhs: FE, rhs: FE) extends FE
case class Or(lhs: FE, rhs: FE) extends FE
case class Not(x: FE) extends FE
case class If(c: FE, ib: FE, eb: FE) extends FE
```

```
sealed abstract class Type
case class TNum() extends Type
case class TBool() extends Type
```

Now type errors can occur:

```
assert(typeOf(And(Num(1),Bool(false))) == TBool())
```

See the Interpreter

A Type Checker for FAE

How do we type functions?

```
App(
  Fun('n, Add(Id('n),Id('n))),
    Num(10)
))
```

The problem: how to give a type to the subexpression

```
Add(Id('n),Id('n))
```

The type of the Id identifier is not known at the time the type checking of such expression is performed.

A Type Checker for FAE

Ask the user to provide a type annotation for the identifiers in the function signature.

```
case class Add(lhs: FAE, rhs: FAE) extends FAE
case class App(funExpr: FAE, arg: FAE) extends FAE
case class Fun(param: Symbol, typ: Type, body: FAE) extends FAE
case class Id(id: Symbol) extends FAE
```

```
type Ctx = Map[Symbol, Type]
```

Propagate type information down in the expression tree using a **typing context**

- The typing context captures an assumption on the type of an identifier
- Use the typing context when typing subexpressions with identifiers

```
App(
  Fun('n, TNum(), Add(Id('n),Id('n))),
    Num(10)
))
```

```
Add(Id('n),Id('n))
```

```
case Id(x) => ctx(x)
```

Acknowledgments

Partially based on content from

Vitaly Shmatikov, Types and Parametric Polymorphism, CS 345

Based on material by Benyamin Pierce
CIS 500, Software Foundations

Programming Languages
A Journey into Abstraction and Composition

Introduction to Programming Language Formalization

Prof. Dr. Guido Salvaneschi

p.g Programming Group

School of Computer Science
University of St.Gallen

Simple Arithmetic Expressions

Here is a BNF grammar for a very simple language of arithmetic expressions:

`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*

Terminology: t here is a **metavariable** (or a **nonterminal**)

Terms, concretely

Define an infinite sequence of sets, S_0, S_1, S_2, . . . , as follows:

$$
\begin{aligned}
S_0 &= \emptyset \\
S_{i+1} &= \{\text{true}, \text{false}, 0\} \\
&\cup \; \{\text{succ } t_1, \text{pred } t_1, \text{iszero } t_1 \mid t_1 \in S_i\} \\
&\cup \; \{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \mid t_1, t_2, t_3 \in S_i\}
\end{aligned}
$$

Now let

$$S = \bigcup_i S_i$$

INDUCTION ON SYNTAX

Inductive Function Definitions

The set of constants appearing in a term t, written $Consts(t)$, is defined as follows:

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

Simple, right?

Another Inductive Definition

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

A proof by induction

Theorem: The number of distinct constants in a term is at most the size of the term. I.e., $|Consts\ (t)| \leq size(t)$.

Proof:

Another proof by induction

Theorem: The number of distinct constants in a term is at most the size of the term. I.e., $|\mathit{Consts}\ (t)| \leq \mathit{size}(t)$.

Proof: By induction on `t`.

Another proof by induction

Theorem: The number of distinct constants in a term is at most the size of the term. I.e., $|\mathit{Consts}\ (t)| \leq \mathit{size}(t)$.

Proof: By induction on `t`.
Assuming the desired property for immediate subterms of `t`, we must prove it for `t` itself.

Another proof by induction

Theorem: The number of distinct constants in a term is at most the size of the term. I.e., $|Consts\ (t)| \leq size(t)$.

Proof: By induction on `t`.
Assuming the desired property for immediate subterms of `t`, we must prove it for `t` itself.

There are "three" cases to consider:
Case: `t is a constant`
Immediate: $|Consts(t)| = |\{t\}| = 1 = size(t)$.

Another proof by induction

Theorem: The number of distinct constants in a term is at most the size of the term. I.e., $|Consts\ (t)| \leq size(t)$.

Proof: By induction on t.
Assuming the desired property for immediate subterms of t, we must prove it for t itself.

There are "three" cases to consider:
Case: `t is a constant`
Immediate: $|Consts(t)| = |\{t\}| = 1 = size(t)$.
Case: t = succ t_1, pred t_1, or iszero t_1
By the induction hypothesis, $|Consts\ (t)| \leq size(t)$. We now calculate as follows:
$|Consts\ (t)| = |Consts\ (t_1)| \leq size(t_1) < size(t)$.

Case: $\mathtt{t = if\ t_1\ then\ t_2\ else\ t_3}$

By the induction hypothesis, $|Consts\ (t_1)| \leq size(t_1)$, $|Consts\ (t_2)| \leq size(t_2)$, and $|Consts\ (t_3)| \leq size(t_3)$. We now calculate as follows:

$$
\begin{aligned}
|Consts\ (t)| &= |Consts\ (t_1) \cup Consts\ (t_2) \cup Consts\ (t_3)| \\
&\leq |Consts\ (t_1)| + |Consts\ (t_2)| + |Consts\ (t_3)| \\
&\leq size(t_1) + size(t_2) + size(t_3) \\
&< size(t).
\end{aligned}
$$

Based on material by Benyamin Pierce
CIS 500, Software Foundations

Programming Languages
A Journey into Abstraction and Composition

Operational Semantics

Prof. Dr. Guido Salvaneschi

p.g Programming Group

School of Computer Science
University of St.Gallen

Abstract Machines

An abstract machine consists of:

- a set of states
- a transition relation on states, written $\longrightarrow$

We read "$\mathtt{t} \longrightarrow \mathtt{t'}$" as "$\mathtt{t}$ evaluates to $\mathtt{t'}$ in one step"

A state records all the information in the machine at a given moment.

For example, an abstract-machine-style description of a conventional microprocessor would include the program counter, the contents of the registers, the contents of main memory, and the machine code program being executed.

Abstract Machines

For the very simple languages we are considering at the moment, however, the term being evaluated is the whole state of the abstract machine.

NB. Often, the transition relation is actually a **partial function** i.e., from a given state, there is **at most one** possible next state.

But in general, there may be many.

Operational semantics for Booleans

Syntax of terms and values

`t ::=`		***terms***
	`true`	*constant true*
	`false`	*constant false*
	`if t then t else t`	*conditional*
`v ::=`		***values***
	`true`	*true value*
	`false`	*false value*

Evaluation relation for Booleans

The evaluation relation $t \longrightarrow t'$ is the smallest relation closed under the following rules:

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

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

$$\frac{t_1 \longrightarrow t'1}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t'1 \texttt{ then } t_2 \texttt{ else } t_3} \qquad \text{(E-If)}$$

Terminology

Computation rules:

$$\texttt{if true then t2 else t3} \longrightarrow \texttt{t2} \quad \text{(E-IfTrue)}$$

$$\texttt{if false then t2 else t3} \longrightarrow \texttt{t3} \quad \text{(E-IfFalse)}$$

Congruence rule:

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1\texttt{'}}{\texttt{if t}_1\texttt{ then t}_2\texttt{ else t}_3 \longrightarrow \texttt{if t}_1\texttt{' then t}_2\texttt{ else t}_3} \quad \text{(E-IF)}$$

Computation rules perform "real" computation steps.

Congruence rules determine *where* computation rules can be applied next.

Evaluation, more explicitly

$\longrightarrow$ is the smallest two-place relation closed under the following rules:

$$((\texttt{if true then } t_2 \texttt{ else } t_3),\ t_2) \in \longrightarrow$$

$$((\texttt{if false then } t_2 \texttt{ else } t_3),\ t_3) \in \longrightarrow$$

$$\frac{(t_1,\ t'_1) \in \longrightarrow}{((\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3),\ (\texttt{if } t'_1 \texttt{ then } t_2 \texttt{ else } t_3)) \in \longrightarrow}$$

The notation $t \longrightarrow t'$ is short-hand for $(t,\ t') \in \longrightarrow$.

Simple Arithmetic Expressions

The set $\mathcal{T}$ of terms is defined by the following abstract grammar:

`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*

Inference Rule Notation

\More explicitly: The set $\mathcal{T}$ is the smallest set closed under the following rules.

$$\mathtt{true} \in \mathcal{T} \qquad \mathit{false} \in \mathcal{T} \qquad \mathtt{0} \in \mathcal{T}$$

$$\frac{t_1 \in \mathcal{T}}{succ\ t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T}}{succ\ t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T}}{succ\ t_1 \in \mathcal{T}}$$

$$\frac{t_1 \in \mathcal{T} \qquad t_2 \in \mathcal{T} \qquad t_3 \in \mathcal{T}}{if\ t_1\ then\ t_2\ else\ t_3 \in \mathcal{T}}$$

Recap: Operational Semantics

Computation rules:

$$\texttt{if true then t2 else t3} \longrightarrow \texttt{t2} \quad (\text{E-IfTrue})$$

$$\texttt{if false then t2 else t3} \longrightarrow \texttt{t3} \quad (\text{E-IfFalse})$$

Congruence rule:

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1\text{'}}{\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \longrightarrow \texttt{if } \texttt{t}_1\text{'} \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3} \quad (\text{E-IF})$$

Digression

Suppose we wanted to change our evaluation strategy so that the `then` and `else` branches of an `if` get evaluated (in that order) before the guard.

How would we need to change the rules?

Digression

Computation rules:

$$\texttt{if true then v2 else v3} \longrightarrow \texttt{v2} \quad \text{(E-IfTrue)}$$

$$\texttt{if false then v2 else v3} \longrightarrow \texttt{v3} \quad \text{(E-IfFalse)}$$

Congruence rule:

$$\frac{\texttt{t2} \longrightarrow \texttt{t2'}}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1 \texttt{ then } t_2\texttt{'} \texttt{ else } t_3}$$

$$\frac{\texttt{t3} \longrightarrow \texttt{t3'}}{\texttt{if } t_1 \texttt{ then } v_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1 \texttt{ then } v_2 \texttt{ else } t_3\texttt{'}}$$

$$\frac{\texttt{t1} \longrightarrow \texttt{t1'}}{\texttt{if } t_1 \texttt{ then } v_2 \texttt{ else } v_3 \longrightarrow \texttt{if } t_1\texttt{'} \texttt{ then v2 else } v_3}$$

Digression

Suppose, that if the evaluation of the `then` and `else` branches leads to the same value, we want to immediately produce that value ("short-circuiting" the evaluation of the guard).

How would we need to change the rules?

Of the rules we just invented, which are computation rules and which are congruence rules?

Digression

Computation rules:

$$\frac{\texttt{v} = \texttt{v}_2 = \texttt{v}_3}{\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{v}_2 \texttt{ else } \texttt{v}_3 \longrightarrow \texttt{v}}$$

$$\texttt{if true then v2 else v3} \longrightarrow \texttt{v2} \quad \text{(E-IfTrue)}$$

$$\texttt{if false then v2 else v3} \longrightarrow \texttt{v3} \quad \text{(E-IfFalse)}$$

Congruence rule:

$$\frac{\texttt{t2} \longrightarrow \texttt{t2'}}{\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \longrightarrow \texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2\texttt{' else } \texttt{t}_3}$$

$$\frac{\texttt{t3} \longrightarrow \texttt{t3'}}{\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{v}_2 \texttt{ else } \texttt{t}_3 \longrightarrow \texttt{if } \texttt{t}_1 \texttt{ then } \texttt{v}_2 \texttt{ else } \texttt{t}_3\texttt{'}}$$

$$\frac{\texttt{t1} \longrightarrow \texttt{t1'} \qquad \texttt{v}_2 \texttt{ != } \texttt{v}_3}{\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{v}_2 \texttt{ else } \texttt{v}_3 \longrightarrow \texttt{if } \texttt{t}_1\texttt{' then v2 else } \texttt{v}_3}$$

REASONING ABOUT EVALUATION

Simple Arithmetic Expressions

The set $\mathcal{T}$ of terms is defined by the following abstract grammar:

`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*

Normal forms

A **normal form** is a term that cannot be evaluated any further
i.e., a term $\mathtt{t}$ is a normal form (or "is in normal form") if there is no $\mathtt{t}'$ s.t. $\mathtt{t} \longrightarrow \mathtt{t}'$.

A normal form is a state where the abstract machine is halted
i.e., it can be regarded as a "result" of evaluation.

Recall that we intended the set of *values* (the boolean constants `true` and `false`) to be exactly the possible "results of evaluation."
Did we get this definition right?

Recap: Language with Booleans

Syntax of terms and values

`t ::=`		***terms***
	`true`	*constant true*
	`false`	*constant false*
	`if t then t else t`	*conditional*
`v ::=`		***values***
	`true`	*true value*
	`false`	*false value*

Values = normal forms (Language with Booleans)

Theorem: A term `t` is a value iff it is in normal form.

Numbers

Let's extend the language.
Will the results about values and normal forms still hold?

New syntactic forms

t ::= ...	terms
0	constant zero
succ t	successor
pred t	predecessor
iszero t	zero test
v ::= ...	Values
nv	numeric value
nv ::=	numeric values
0	zero value
succ nv	successor value

Numbers

New evaluation rules

$$\boxed{\mathtt{t}_1 \longrightarrow \mathtt{t'}_1}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t'}_1}{\mathtt{succ}\ \mathtt{t}_1 \longrightarrow \mathtt{succ}\ \mathtt{t'}_1} \quad \text{(E-Succ)}$$

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

$$\mathtt{pred}\ (\mathtt{succ}\ \mathtt{nv}_1) \longrightarrow \mathtt{nv}_1 \quad \text{(E-PredSucc)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t'}_1}{\mathtt{pred}\ \mathtt{t}_1 \longrightarrow \mathtt{pred}\ \mathtt{t'}_1} \quad \text{(E-Pred)}$$

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

$$\mathtt{iszero}\ (\mathtt{succ}\ \mathtt{nv}_1) \longrightarrow \mathtt{false} \quad \text{(E-IszeroSucc)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t'}_1}{\mathtt{iszero}\ \mathtt{t}_1 \longrightarrow \mathtt{iszero}\ \mathtt{t'}_1} \quad \text{(E-IsZero)}$$

Values are normal forms

Our observation a few slides ago that all values are in normal form still holds for the extended language.

Is the converse true? I.e., is every normal form a value?

Values are normal forms

Our observation a few slides ago that all values are in normal form still holds for the extended language.

Is the converse true? I.e., is every normal form a value?

No: some terms are *stuck*.

Formally, a stuck term is one that is a normal form but not a value.
What are some examples?

Stuck terms model **run-time errors**.

Multi-step evaluation.

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

I.e., it is the smallest relation closed under the following rules:

$$\frac{\mathtt{t} \longrightarrow \mathtt{t}'}{\mathtt{t} \longrightarrow^* \mathtt{t}'}$$

$$\mathtt{t} \longrightarrow^* \mathtt{t}$$

$$\frac{\mathtt{t} \longrightarrow^* \mathtt{t}' \qquad \mathtt{t}' \longrightarrow^* \mathtt{t}''}{\mathtt{t} \longrightarrow^* \mathtt{t}''}$$

Termination of evaluation

Theorem: For every `t` there is some normal form `t` such that $\mathtt{t} \longrightarrow^{*} \mathtt{t'}$.

Proof:

Termination of evaluation

Theorem: For every t there is some normal form t' such that $t \longrightarrow^* t'$.

Proof:

- First, recall that single-step evaluation strictly reduces the size of the term:
  if $t \longrightarrow t'$, *then* $size(t) > size(t')$
- Now, assume (for a contradiction) that

  $t_0,\ t_1,\ t_2,\ t_3,\ t_4,\ \ldots$

  is an infinite-length sequence such that

  $t_0 \longrightarrow t_1 \longrightarrow t_2 \longrightarrow t_3 \longrightarrow t_4 \longrightarrow \cdots.$
- Then

  $size(t_0) > size(t_1) > size(t_2) > size(t_3) > \ldots$
- But such a sequence cannot exist — contradiction!

Based on material by Benyamin Pierce
CIS 500, Software Foundations

Programming Languages
A Journey into Abstraction and Composition

Type Systems, Formally

Prof. Dr. Guido Salvaneschi

Programming Group

School of Computer Science
University of St.Gallen

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* $\mathtt{t : T}$ 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,
   a. if $\mathtt{t : T}$ and $\mathtt{t} \longrightarrow^* \mathtt{v}$, then $\mathtt{v : T}$
   b. if $\mathtt{t : T}$, then evaluation of $\mathtt{t}$ will not get stuck

Review: 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*

Evaluation Rules

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

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

$$\frac{t_1 \longrightarrow t'_1}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t'_1 \texttt{ then } t_2 \texttt{ else } t_3} \qquad \text{(E-If)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}'_1}{\mathtt{succ}\ \mathtt{t}_1 \longrightarrow \mathtt{succ}\ \mathtt{t}'_1} \qquad \text{(E-Succ)}$$

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

$$\mathtt{pred}\ (\mathtt{succ}\ \mathtt{nv}_1) \longrightarrow \mathtt{nv}_1 \qquad \text{(E-PredSucc)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}'_1}{\mathtt{pred}\ \mathtt{t}_1 \longrightarrow \mathtt{pred}\ \mathtt{t}'_1} \qquad \text{(E-Pred)}$$

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

$$\mathtt{iszero}\ (\mathtt{succ}\ \mathtt{nv}_1) \longrightarrow \mathtt{false} \qquad \text{(E-IszeroSucc)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}'_1}{\mathtt{iszero}\ \mathtt{t}_1 \longrightarrow \mathtt{iszero}\ \mathtt{t}'_1} \qquad \text{(E-Iszero)}$$

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

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

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

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

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

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

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

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

Typing Derivations

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

$$\dfrac{\dfrac{\dfrac{}{\texttt{0 : Nat}}\,\textsc{T-Zero}}{\texttt{iszero 0 : Bool}}\,\textsc{T-IsZero} \qquad \dfrac{}{\texttt{0 : Nat}}\,\textsc{T-Zero} \qquad \dfrac{\dfrac{}{\texttt{0 : Nat}}\,\textsc{T-Zero}}{\texttt{pred 0 : Nat}}\,\textsc{T-Pred}}{\texttt{if iszero 0 then 0 else pred 0 : Nat}}\,\textsc{T-If}$$

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

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.

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

Using this rule, we cannot assign a type to

```
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:

Progress: A well-typed term is not stuck

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

Preservation: Types are preserved by one-step evaluation

If $\mathtt{t} : \mathtt{T}$ *and* $\mathtt{t} \longrightarrow \mathtt{t}'$*, then* $\mathtt{t}' : \mathtt{T}$*.*

Recap: Type Systems

Very successful example of a **lightweight formal method**

Big topic in PL research

Enabling technology for all sorts of other things, e.g. language-based security

The skeleton around which modern programming languages are designed

Based on material by Benyamin Pierce
CIS 500, Software Foundations

A Type System for The Lambda Calculus

Programming Group

School of Computer Science

University of St.Gallen

Syntax

`t ::=`	*terms*
`x`	*variable*
`λx.t`	*abstraction*
`t t`	*application*

Terminology:

- terms in the pure λ-calculus are often called λ–*terms*
- terms of the form `λx. t` are called λ-*abstractions* or just abstractions

Syntactic conventions

Since λ-calculus provides only one-argument functions, all multi-argument functions must be written in curried style.

The following conventions make the linear forms of terms easier to read:

Application associates to the left
E.g., $t\ u\ v$ *means* $(t\ u)\ v$*, not* $t\ (u\ v)$

Bodies of λ-abstractions extend as far to the right as possible
E.g., $\lambda x.\lambda y.xy$ *means* $\lambda x.(\lambda y.xy)$*, not* $\lambda x.(\lambda y.x)y$

Scope

The λ-abstraction term `λx.t` *binds* the variable x.

The *scope* of this binding is the body `t`.

Occurrences of `x` inside `t` are said to be *bound* by the abstraction.

Occurrences of `x` that are *not* within the scope of an abstraction binding `x` are said to be *free*.

```
λx. λy. x y z
λx. (λy. z y) y
```

Values

`v ::=`	*values*
λ`x.t`	*abstraction value*

Operational Semantics

Computation rule:

$$(\lambda x.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \qquad \text{(E-AppAbs)}$$

Notation: $[x \mapsto v_2]t_{12}$ *is "the term that results from substituting free occurrences of* x *in* t_{12} *with* v_{12}*."*

Congruence rules:

$$\frac{t_1 \longrightarrow t'_1}{t_1\ t_2 \longrightarrow t'_1\ t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t'_2}{v_1\ t_2 \longrightarrow v_1\ t'_2} \qquad \text{(E-App2)}$$

Normal forms

Recall:

- A *normal form* is a term that cannot take an evaluation step.
- A *stuck* term is a normal form that is not a value.

Are there any stuck terms in the pure λ-calculus?
Prove it.

Normal forms

Recall:

- A *normal form* is a term that cannot take an evaluation step.
- A *stuck* term is a normal form that is not a value.

Are there any stuck terms in the pure λ-calculus?

Prove it.

Does every term evaluate to a normal form?

Prove it.

The simply typed lambda-calculus

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

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

- So, strictly speaking, there are *many* variants of $\lambda_{\rightarrow}$, 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*

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 typin rules simpler. Let’s take this choice for now.

Typing rules

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

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

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

Typing rules

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

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

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

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

Typing rules

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

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

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

$$\frac{\Gamma, \texttt{x}:\texttt{T}_1 \vdash \texttt{t}_2 : \texttt{T}_2}{\Gamma \vdash \lambda \texttt{x}:\texttt{T}_1.\texttt{t}_2 : \texttt{T}_1 \longrightarrow \texttt{T}_2} \qquad \text{(T-ABS)}$$

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

Typing rules

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

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

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

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

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

$$\frac{\Gamma \vdash t_1 : T_{11} \longrightarrow T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\, t_2 : T_{12}} \qquad \text{(T-App)}$$

Typing Derivations

What derivations justify the following typing statements?

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

$f{:}\mathrm{Bool} \longrightarrow \mathrm{Bool} \vdash f\ (\mathrm{if\ false\ then\ true\ else\ false}) : \mathrm{Bool}$

$f{:}\mathrm{Bool} \longrightarrow \mathrm{Bool} \vdash \lambda x{:}\mathrm{Bool}.\ f\ (\mathrm{if}\ x\ \mathrm{then\ false\ else}\ x) : \mathrm{Bool} \longrightarrow \mathrm{Bool}$

Properties of $\lambda_{\rightarrow}$

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

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

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

Preservation: Types are preserved by one-step evaluation

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