

# Formal Systems and their Applications

Dave Clarke

([Dave.Clarke@cs.kuleuven.be](mailto:Dave.Clarke@cs.kuleuven.be))

Acknowledgment: these slides are based in part on slides from Benjamin Pierce and Frank Piessens

# Course Overview

- Introduction
- Formal models of programming languages
  - Abstract syntax trees
  - Structural operational semantics
  - The lambda calculus
- Simple type systems
- Subtyping
- Conclusion

# Formal Models of Programming Languages

# Introduction

- To study programming languages formally, we need:
  - Formal account of their syntax
  - Formal definition of their semantics
- We introduce the techniques used for defining syntax and semantics by looking at a toy language

# Syntax

# Presentation of the syntax

- BNF-like notation

<code>t ::=</code>	<i>terms</i>
<code>  true</code>	<i>constant true</i>
<code>  false</code>	<i>constant false</i>
<code>  if t then t else t</code>	<i>conditional</i>
<code>  0</code>	<i>constant zero</i>
<code>  succ t</code>	<i>successor</i>
<code>  pred t</code>	<i>predecessor</i>
<code>  iszero t</code>	<i>zero test</i>

- Note the use of meta-variables (t)
- What does such a definition mean exactly?

# Inductive definition of syntax

- The BNF notation is considered a shorthand for:

The set  $\mathcal{T}$  of *terms* is the smallest set such that

1.  $\{\text{true}, \text{false}, 0\} \subseteq \mathcal{T}$ ;
2. if  $t_1 \in \mathcal{T}$ , then  $\{\text{succ } t_1, \text{pred } t_1, \text{iszero } t_1\} \subseteq \mathcal{T}$ ;
3. if  $t_1 \in \mathcal{T}$ ,  $t_2 \in \mathcal{T}$ , and  $t_3 \in \mathcal{T}$ , then  
if  $t_1$  then  $t_2$  else  $t_3 \in \mathcal{T}$ .

# Inductive definition of syntax

- Often alternatively presented as:

$$\frac{\text{true} \in \mathcal{T}}{\text{succ } t_1 \in \mathcal{T}} \quad \frac{\text{false} \in \mathcal{T}}{\text{pred } t_1 \in \mathcal{T}} \quad \frac{0 \in \mathcal{T}}{\text{iszero } t_1 \in \mathcal{T}}$$
$$\frac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T} \quad t_3 \in \mathcal{T}}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \in \mathcal{T}}$$

- Note:
  - Strings versus Abstract Syntax Trees (AST's)
  - Terminology: axiom, inference rule, rule schema



# Concrete definition of syntax

- A more constructive characterization:

Define an infinite sequence of sets,  $\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2, \dots$ , as follows:

$$\mathcal{S}_0 = \emptyset$$

$$\begin{aligned} \mathcal{S}_{i+1} = & \{ \text{true}, \text{false}, 0 \} \\ & \cup \{ \text{succ } t_1, \text{pred } t_1, \text{iszero } t_1 \mid t_1 \in \mathcal{S}_i \} \\ & \cup \{ \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \mid t_1, t_2, t_3 \in \mathcal{S}_i \} \end{aligned}$$

Now let

$$\mathcal{S} = \bigcup_i \mathcal{S}_i$$

- Theorem: the sets S and T are equal

# Induction on terms

- The constructive characterization gives us an important tool for proving things about terms, the *principle of induction on terms*

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  $t$ .

- Variants include: induction on depth and size

# Inductive definitions of functions

- It also justifies “recursive” definitions such as:

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

- Intuition: the definition is OK because “the recursive computation always terminates”

# Example

```
size(true)           = 1
size(false)          = 1
size(0)              = 1
size(succ t1)       = size(t1) + 1
size(pred t1)       = size(t1) + 1
size(iszero t1)     = size(t1) + 1
size(if t1 then t2 else t3) = size(t1) + size(t2) + size(t3) + 1
```

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

# Semantics

# Various semantic “styles”

- Operational semantics:
  - Define an abstract machine for the language
- Denotational semantics:
  - Define the meaning of a program to be a (intricate) mathematical structure
- Axiomatic semantics:
  - Define what you can know about the program state at each execution point of the program

We will only use operational semantics.  
In particular, we will use the so-called  
“small-step structural operational semantics”

# Small-Step Structural Operational Semantics

- We define an abstract machine, consisting of:
  - A set of states
  - A transition relation that defines how the state changes over time
- In the simple case we are considering now, the state is just the program
  - Computation is rewriting (“simplification”) of the program

# Semantics of Boolean Expressions

- Define “end-states” or values:

<code>t ::=</code>	<i>terms</i>
<code>  true</code>	<i>constant true</i>
<code>  false</code>	<i>constant false</i>
<code>  if t then t else t</code>	<i>conditional</i>
<code>v ::=</code>	<i>values</i>
<code>  true</code>	<i>true value</i>
<code>  false</code>	<i>false value</i>



# Semantics of Boolean Expressions

- Define the evaluation relation:

if true then  $t_2$  else  $t_3 \longrightarrow t_2$  (E-IFTRUE)

if false then  $t_2$  else  $t_3 \longrightarrow t_3$  (E-IFFALSE)

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

- Terminology:
  - E-IfTrue and E-IfFalse are *computation rules*
  - E-If is a *congruence rule*

# Derivations

- A pair  $(t, t')$  is only in the evaluation relation iff it is justified by the rules
- This justification can be made explicit as a “derivation tree”
- A powerful technique for proving properties of the evaluation relation is:
  - Induction on derivations
- Example: prove the determinacy of the evaluation relation

# Normal forms

- Def: A term is in **normal form** if it can not be evaluated further
- Thm: All values are normal forms
- Thm: All normal forms are values
  - By structural induction on terms
- Def: The *multi-step evaluation* relation,  $\longrightarrow^*$ , is the reflexive, transitive closure of single-step evaluation.
- Thm: Normal forms are unique
- Thm: Evaluation always terminates

# Booleans + Natural Numbers

*New syntactic forms*

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

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

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

*terms*

*constant zero*  
*successor*  
*predecessor*  
*zero test*

*values*

*numeric value*

*numeric values*

*zero value*  
*successor value*

*New evaluation rules*

$t \longrightarrow t'$

$$\frac{t_1 \longrightarrow t'_1}{\text{succ } t_1 \longrightarrow \text{succ } t'_1} \quad (\text{E-SUCC})$$

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

$$\text{pred } (\text{succ } nv_1) \longrightarrow nv_1 \quad (\text{E-PREDSUCC})$$

$$\frac{t_1 \longrightarrow t'_1}{\text{pred } t_1 \longrightarrow \text{pred } t'_1} \quad (\text{E-PRED})$$

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

$$\text{iszero } (\text{succ } nv_1) \longrightarrow \text{false} \quad (\text{E-ISZEROSUCC})$$

$$\frac{t_1 \longrightarrow t'_1}{\text{iszero } t_1 \longrightarrow \text{iszero } t'_1} \quad (\text{E-ISZERO})$$

# What properties remain true?

- Thm: Values are normal forms
  - OK
- Thm: all normal forms are values?
  - NO: succ false is in normal form and not a value
- Def: a term is **stuck** if it is in normal form and not a value
  - Stuck terms model runtime errors
  - A key goal of type systems will be to remove such runtime errors

# Wrap up

- Our model of programming languages:
  - Syntax is an inductively defined set of Abstract Syntax Trees, called **terms** (that can be rendered to and parsed from strings if needed by using parentheses)
  - Semantics consists of:
    - An inductively defined **evaluation relation** between terms
    - A definition of a subset of terms (the **values**) that are deemed to be end-results of computation
  - A program (term) leads to a run-time error if it gets **stuck**, i.e. evaluates to a normal form that is not a value

# The untyped lambda calculus



# Introduction

- We now switch to a more interesting programming language than the expression language we considered so far
- The lambda calculus is:
  - A Turing-complete language
  - And its key abstractions – function definition and application – are closely related to abstractions found in programming languages

# Syntax

$t ::=$	<i>terms</i>
$x$	<i>variable</i>
$\lambda x. t$	<i>abstraction</i>
$t t$	<i>application</i>

- Some syntactic conventions:

- ▶ Application associates to the left

*E.g.,  $t u v$  means  $(t u) v$ , not  $t (u v)$*

- ▶ Bodies of  $\lambda$ - abstractions extend as far to the right as possible

*E.g.,  $\lambda x. \lambda y. x y$  means  $\lambda x. (\lambda y. x y)$ , not  $\lambda x. (\lambda y. x) y$*

# Scope and free variables

- In the term  $\lambda x.t$ , the variable  $x$  is **bound** in  $t$  is the **scope** of the binding
- A variable is **free** if it is not bound by any enclosing abstraction
- A term without free variables is called a **closed term**

# Operational semantics (informal)

- Evaluating terms always boils down to performing function application:
  - An actual parameter (term) is substituted for the formal parameter in the body of a lambda abstraction
- Making this precise is more tricky than appears at first sight: we will give the formal definition later

# Multiple arguments

- Functions with more than one argument can be simulated using higher order functions:
  - Instead of  $\lambda(x,y).s$ , we write  $\lambda x. \lambda y.s$
  - Instead of  $f(a,b)$  we write  $f a b$

# The Church Booleans

```
tru  =  $\lambda t. \lambda f. t$   
fls  =  $\lambda t. \lambda f. f$ 
```

```
      tru v w  
=  $(\lambda t. \lambda f. t)$  v w  by definition  
→  $(\lambda f. v)$  w      reducing the underlined redex  
→ v                  reducing the underlined redex
```

```
      fls v w  
=  $(\lambda t. \lambda f. f)$  v w  by definition  
→  $(\lambda f. f)$  w      reducing the underlined redex  
→ w                  reducing the underlined redex
```

# Functions on Booleans

```
not  =  λb. b fls tru
```

```
and  =  λb. λc. b c fls
```

- Exercise: Compute a logical expression

# Encoding Pairs

```
pair = λf.λs.λb. b f s
fst  = λp. p tru
snd  = λp. p fls
```

$$\begin{aligned} & \text{fst (pair v w)} \\ = & \text{fst } (\underline{(\lambda f. \lambda s. \lambda b. b f s)} \text{ v w}) && \text{by definition} \\ \longrightarrow & \text{fst } (\underline{(\lambda s. \lambda b. b v s)} \text{ w}) && \text{reducing} \\ \longrightarrow & \text{fst } (\lambda b. b v w) && \text{reducing} \\ = & \underline{(\lambda p. p \text{ tru})} (\lambda b. b v w) && \text{by definition} \\ \longrightarrow & \underline{(\lambda b. b v w)} \text{ tru} && \text{reducing} \\ \longrightarrow & \text{tru v w} && \text{reducing} \\ \longrightarrow^* & \text{v} && \text{as before.} \end{aligned}$$



# Church Numerals

Idea: represent the number  $n$  by a function that “repeats some action  $n$  times.”

$$c_0 = \lambda s. \lambda z. z$$

$$c_1 = \lambda s. \lambda z. s z$$

$$c_2 = \lambda s. \lambda z. s (s z)$$

$$c_3 = \lambda s. \lambda z. s (s (s z))$$

That is, each number  $n$  is represented by a term  $c_n$  that takes two arguments,  $s$  and  $z$  (for “successor” and “zero”), and applies  $s$ ,  $n$  times, to  $z$ .

# Church Numerals

Successor:

$$\text{succ} = \lambda n. \lambda s. \lambda z. s (n s z)$$

Addition:

$$\text{plus} = \lambda m. \lambda n. \lambda s. \lambda z. m s (n s z)$$

Multiplication:

$$\text{times} = \lambda m. \lambda n. m (\text{plus } n) c_0$$

Zero test:

$$\text{iszro} = \lambda m. m (\lambda x. \text{fls}) \text{tru}$$

# Recursion

- Evaluation in the lambda calculus can diverge:

$$\text{omega} = (\lambda x. x x) (\lambda x. x x)$$

- More surprisingly: arbitrary recursive functions can be defined in the lambda calculus!
  - There exists a lambda term `fix`, that has the property that:
    - `fix f v` evaluates to: `f (fix f) v`, and hence to:
      - `f (f (fix f)) v`, and to:
        - `f (f (f (fix f))) v`

# Recursion

- We can use fix to define recursive functions:
  - First abstract out the recursive call:

```
f = λfct.  
    λn.  
        if n=0 then 1  
        else n * (fct (pred n))
```

- Then apply fix:
  - fix f n will compute the factorial of n

# Encoding versus extending the syntax

- The previous exercises show that various programming language features can be “encoded” in the lambda calculus
- But in our further study, we will introduce all these features by extending the syntax instead of encoding
  - This does not gain expressive power
  - But it does gain readability and typability
- The encoding we discussed will play no major role in the rest of this course

# Operational Semantics (formal)

$t ::=$	<i>terms</i>
$x$	<i>variable</i>
$\lambda x. t$	<i>abstraction</i>
$t t$	<i>application</i>

$v ::=$	<i>values</i>
$\lambda x. t$	<i>abstraction value</i>

Computation rule:

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

Congruence rules:

$$\frac{t_1 \longrightarrow t'_1}{t_1 t_2 \longrightarrow t'_1 t_2} \quad (\text{E-APP1})$$
$$\frac{t_2 \longrightarrow t'_2}{v_1 t_2 \longrightarrow v_1 t'_2} \quad (\text{E-APP2})$$

# Operational Semantics (formal)

- Free variables in a term:

$$FV(x) = \{x\}$$

$$FV(\lambda x. t_1) = FV(t_1) \setminus \{x\}$$

$$FV(t_1 t_2) = FV(t_1) \cup FV(t_2)$$

- Substitution:
  - Is tricky to define correct. We proceed in a number of attempts to the correct definition

# Substitution, take 1

Consider the following definition of substitution:

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

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

if  $x \neq y$

$$[x \mapsto s](\lambda y. t_1) = \lambda y. ([x \mapsto s]t_1)$$

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

What is wrong with this definition?

It substitutes for free and *bound* variables!

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

This is not what we want!



# Substitution, take 2

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

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

$$[x \mapsto s](\lambda y. t_1) = \lambda y. ([x \mapsto s]t_1) \quad \text{if } x \neq y$$

$$[x \mapsto s](\lambda x. t_1) = \lambda x. t_1$$

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

What is wrong with this definition?

It suffers from *variable capture*!

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

This is also not what we want.

# Substitution, take 3

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

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

if  $x \neq y$

$$[x \mapsto s](\lambda y. t_1) = \lambda y. ([x \mapsto s]t_1)$$

if  $x \neq y, y \notin FV(s)$

$$[x \mapsto s](\lambda x. t_1) = \lambda x. t_1$$

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

What is wrong with this definition?

Now substitution is a *partial function*!

E.g.,  $[x \mapsto y](\lambda y. x)$  is undefined.

But we want an result for every substitution.

# Alpha-equivalence

- To make substitution a total operation again, we observe that:
  - Names of bound variables do not matter
- Two terms are alpha-equivalent if they only differ in the choice of names for bound variables

# Substitution, final definition

Now consider substitution as an operation over *alpha-equivalence classes* of terms.

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

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

if  $x \neq y$

$$[x \mapsto s](\lambda y. t_1) = \lambda y. ([x \mapsto s]t_1)$$

if  $x \neq y, y \notin FV(s)$

$$[x \mapsto s](\lambda x. t_1) = \lambda x. t_1$$

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

Examples:

- ▶  $[x \mapsto y](\lambda y. x)$  must give the same result as  $[x \mapsto y](\lambda z. x)$ . We know the latter is  $\lambda z. y$ , so that is what we will use for the former.
- ▶  $[x \mapsto y](\lambda x. z)$  must give the same result as  $[x \mapsto y](\lambda w. z)$ . We know the latter is  $\lambda w. z$  so that is what we use for the former.

# Exercise

So what does

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

reduce to?

# Wrap-up

- The lambda-calculus is a simple, yet powerful model of computation
- We will use it as the core of most of the programming languages we study in this course
- It is also at the core of some real-life languages such as ML, Haskell, Lisp and Scheme