Formal Systems and their Applications

Dave Clarke
(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**

\[
t ::= \\
true \\
false \\
\text{if } t \text{ then } t \text{ else } t \\
0 \\
\text{succ } t \\
\text{pred } t \\
\text{iszero } t \\
\]

**terms**
- constant true
- constant false
- conditional
- constant zero
- successor
- predecessor
- zero test

- 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 $T$ of terms is the smallest set such that

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

• Often alternatively presented as:

\[
\begin{align*}
\text{true} & \in \mathcal{T} & \text{false} & \in \mathcal{T} & 0 & \in \mathcal{T} \\
\frac{t_1 \in \mathcal{T}}{\text{succ } t_1 \in \mathcal{T}} & \frac{t_1 \in \mathcal{T}}{\text{pred } t_1 \in \mathcal{T}} & \frac{t_1 \in \mathcal{T}}{\text{iszero } t_1 \in \mathcal{T}} \\
\frac{t_1 \in \mathcal{T} & t_2 \in \mathcal{T} & t_3 \in \mathcal{T}}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \in \mathcal{T}}
\end{align*}
\]

• 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, $S_0, S_1, S_2, \ldots$, as follows:

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

Now let

\[
S = \bigcup_i S_i
\]

- 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*

\[
\text{If, for each term } s, \\
\text{given } P(r) \text{ for all immediate subterms } r \text{ of } s \\
\text{we can show } P(s), \\
\text{then } P(t) \text{ holds for all } t.
\]

• Variants include: induction on depth and size
Inductive definitions of functions

- It also justifies “recursive” definitions such as:

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

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

\[
\begin{align*}
\text{size}(\text{true}) & = 1 \\
\text{size}(\text{false}) & = 1 \\
\text{size}(0) & = 1 \\
\text{size}(\text{succ } t_1) & = \text{size}(t_1) + 1 \\
\text{size}(\text{pred } t_1) & = \text{size}(t_1) + 1 \\
\text{size}(\text{iszero } t_1) & = \text{size}(t_1) + 1 \\
\text{size}(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) & = \text{size}(t_1) + \text{size}(t_2) + \text{size}(t_3) + 1
\end{align*}
\]

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

\[
t ::= \begin{align*}
true \\
false \\
\text{if } t \text{ then } t \text{ else } t 
\end{align*}
\]

\[
v ::= \begin{align*}
true \\
false 
\end{align*}
\]

\text{terms}
\begin{align*}
\text{constant true} \\
\text{constant false} \\
\text{conditional}
\end{align*}

\text{values}
\begin{align*}
\text{true value} \\
\text{false value}
\end{align*}
Semantics of Boolean Expressions

- Define the evaluation relation:

  \[
  \text{if true then } t_2 \text{ else } t_3 \rightarrow t_2 \quad (E-\text{IfTrue})
  \]
  \[
  \text{if false then } t_2 \text{ else } t_3 \rightarrow t_3 \quad (E-\text{IfFalse})
  \]
  \[
  t_1 \rightarrow t_1' \quad \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3 \quad (E-\text{If})
  \]

- 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

\[ t ::= \ldots \]
\[ 0 \]
\[ \text{succ } t \]
\[ \text{pred } t \]
\[ \text{iszero } t \]

\[ v ::= \ldots \]
\[ \text{nv} \]

\[ \text{nv} ::= \]
\[ 0 \]
\[ \text{succ } \text{nv} \]

terms
constant zero
successor
predecessor
zero test

values
numeric value

numeric values
zero value
successor value
New evaluation rules

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

\[ t ::= \]
\[ x \]
\[ \lambda x.t \]
\[ t \quad t \]

- Some syntactic conventions:
  - Application associates to the left
    \[ E.g., t \ u \ v \text{ means } (t \ u) \ v, \text{ not } t \ (u \ v) \]
  - Bodies of \( \lambda \)-abstractions extend as far to the right as possible
    \[ E.g., \lambda x. \lambda y. \ x \ y \text{ means } \lambda x. (\lambda y. x \ y), \text{ not } \lambda x. (\lambda y. x) \ y \]
Scope and free variables

• In the term $\lambda x.t$, the variable $x$ is **bound** in $t$.
  $t$ 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

\[
\text{tru} = \lambda t. \lambda f. t \\
\text{fals} = \lambda t. \lambda f. f
\]

\[
\text{tru} \ v \ w \\
= (\lambda t. \lambda f. t) \ v \ w \quad \text{by definition} \\
\longrightarrow (\lambda f. \ v) \ w \quad \text{reducing the underlined redex} \\
\longrightarrow v \quad \text{reducing the underlined redex}
\]

\[
\text{fals} \ v \ w \\
= (\lambda t. \lambda f. f) \ v \ w \quad \text{by definition} \\
\longrightarrow (\lambda f. \ f) \ w \quad \text{reducing the underlined redex} \\
\longrightarrow w \quad \text{reducing the underlined redex}
\]
Functions on Booleans

\[ \text{not} \ = \ \lambda b. \ b \ fls \ tru \]

\[ \text{and} \ = \ \lambda b. \ \lambda c. \ b \ c \ fls \]

- Exercise: Compute a logical expression
Encoding Pairs

\[
\begin{align*}
\text{pair} &= \lambda f. \lambda s. \lambda b. b \ f \ s \\
\text{fst} &= \lambda p. p \ \text{tru} \\
\text{snd} &= \lambda p. p \ \text{fls}
\end{align*}
\]

\[
\begin{align*}
\text{fst} \ (\text{pair} \ v \ w) \\
&= \text{fst} \ ((\lambda f. \lambda s. \lambda b. b \ f \ s) \ v \ w) \quad \text{by definition} \\
&\quad \rightarrow \text{fst} \ ((\lambda s. \lambda b. b \ v \ s) \ w) \quad \text{reducing} \\
&\quad \rightarrow \text{fst} \ (\lambda b. b \ v \ w) \quad \text{reducing} \\
&\quad = (\lambda p. p \ \text{tru}) \ (\lambda b. b \ v \ w) \quad \text{by definition} \\
&\quad \rightarrow (\lambda b. b \ v \ w) \ \text{tru} \quad \text{reducing} \\
&\quad \rightarrow \ \text{tru} \ v \ w \quad \text{reducing} \\
&\quad \rightarrow^* \ v \quad \text{as before.}
\end{align*}
\]
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 \( \text{fix} \), that has the property that:
    • \( \text{fix} f v \) evaluates to: \( f (\text{fix} f) v \), and hence to:
    • \( f (f (\text{fix} f)) v \), and to:
    • \( f (f (f (\text{fix} f))) v \)
Recursion

• *We can use fix to define recursive functions:*  
  – First abstract out the recursive call:

\[
\begin{align*}
f &= \lambda fct. \\
    &\quad \lambda n. \\
    &\quad \text{if } n=0 \text{ then } 1 \\
    &\quad \text{else } n \times (fct(\text{pred } n))
\end{align*}
\]

– Then apply fix:
  • \(\text{fix } f n\) 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)

\[
t ::= \quad \text{terms} \\
x \quad \lambda x.t \quad tt
\]

\[
v ::= \quad \text{values} \\
\lambda x.t
\]

**Computation rule:**

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

**Congruence rules:**

\[
\frac{t_1 \rightarrow t'_1}{t_1 t_2 \rightarrow t'_1 t_2} \quad (E\text{-APP1})
\]

\[
\frac{t_2 \rightarrow t'_2}{v_1 t_2 \rightarrow v_1 t'_2} \quad (E\text{-APP2})
\]
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
Consider the following definition of substitution:

\[
\begin{align*}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y & \text{if } x \neq y \\
[x \mapsto s](\lambda y. t_1) &= \lambda y. ([x \mapsto s]t_1) \\
[x \mapsto s](t_1 \ t_2) &= ([x \mapsto s]t_1)([x \mapsto s]t_2)
\end{align*}
\]

What is wrong with this definition?

It substitutes for free and \textit{bound} variables!

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

This is not what we want!
Substitution, take 2

\[
\begin{align*}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y & \text{if } x \neq y \\
[x \mapsto s](\lambda y. t_1) &= \lambda y. ([x \mapsto s]t_1) & \text{if } x \neq y \\
[x \mapsto s](\lambda x. t_1) &= \lambda x. t_1 \\
[x \mapsto s](t_1 \ t_2) &= ([x \mapsto s]t_1)([x \mapsto s]t_2)
\end{align*}
\]

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

\[
\begin{align*}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y & \text{if } x \neq y \\
[x \mapsto s](\lambda y \cdot t_1) &= \lambda y \cdot ([x \mapsto s]t_1) & \text{if } x \neq y, y \not\in \text{FV}(s) \\
[x \mapsto s](\lambda x \cdot t_1) &= \lambda x \cdot t_1 \\
[x \mapsto s](t_1 \ t_2) &= ([x \mapsto s]t_1)([x \mapsto s]t_2)
\end{align*}
\]

What is wrong with this definition?

Now substitution is a \textit{partial function}!

E.g., \([x \mapsto y](\lambda y \cdot x)\) 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.

\[
\begin{align*}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y & \text{if } x \neq y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y. ([x \mapsto s]t_1) & \text{if } x \neq y, y \not\in FV(s) \\
[x \mapsto s](\lambda x.t_1) &= \lambda x. t_1 \\
[x \mapsto s](t_1 \ t_2) &= ([x \mapsto s]t_1)([x \mapsto s]t_2)
\end{align*}
\]

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