01—Language Processing and Inductive Definitions

CS4215: Programming Language Implementation

Martin Henz

January 13, 2012
1 Brief Introduction to CS4215
2 Administrative Matters
3 Language Processing
4 Inductive Definitions
5 ePL Outlook
1 Brief Introduction to CS4215
   - Goal: Implementation Principles, Not “Hacking”
   - Method: “Learning by Programming”
   - Style: Incremental and Exploratory
   - Overview of Module Content

2 Administrative Matters

3 Language Processing

4 Inductive Definitions

5 ePL Outlook
Goal: Implementation Principles, Not “Hacking”

- Implementation of major programming language concepts
- As little “clutter” as possible
- Emphasis on the “what” of implementation: correctness w.r.t. given semantics
Learning By Programming

- **Goal:** get the insider’s view on programming languages
- You will implement a sequence of toy languages
- You will write interpreters in Java
- You will write virtual machines in Java
- You will write toy programs in the toy languages
- Extensive software support provided
Incremental² and Exploratory²

- Incremental: Sequence of programming languages, from simple expression-oriented to complex object-oriented
- Incremental: Sequence of implementation techniques, from the simplest interpreter-based implementation to realistic memory-aware virtual machines
- Exploratory: Plenty of scope for exploration, from the most basic to the most advanced topics in each section
- Exploratory: Opportunities for exploring related topics, hands-on, within module framework
Overview of Module Content

1. Programming language processing tools and inductive definitions (2 hours)
2. ePL: An Expression language (2 hours)
3. simPL: A simple functional language (6 hours)
4. rePL: Records for Functional Programming (2 hours)
5. imPL: A Simple Imperative Language (3 hours)
6. oPL: A Simple Object-oriented Language (3 hours)
7. Memory management, garbage collection (3 hours)
8. Implementation of type systems (3 hours)
9. Combining implementation techniques (2 hours): virtual machines and interpreters, virtual machines and just-in-time compilation
Use www.comp.nus.edu.sg/~cs4215 and IVLE
Notes and slides (www; no textbook)
Assignments (www; intensive work; marked; labs)
Discussion forums (IVLE)
Announcements (IVLE)
Webcast (IVLE)
No tutorials but labs; register!
1 Brief Introduction to CS4215

2 Administrative Matters

3 Language Processing
   - T-Diagrams
   - Translators
   - Interpreters
   - Combinations

4 Inductive Definitions

5 ePL Outlook
T-Diagrams

586

x86 Processor

C&C

Program “C&C” (x86 code)

x86

“C&C” running on x86

x86

x86
Translators

- Translator translates from one language—the *from-language*—to another language—the *to-language*
- Compiler translates from “high-level” language to “low-level” language
- De-compiler translates from “low-level” language to “high-level” language
Basic-to-C compiler written in x86 machine code
Compilation

Compiling “C&C” from Basic to C
Two-stage Compilation

Compiling “C&C” from Basic to C to x86 machine code
Compiling a Basic-to-x86 compiler from C to x86 machine code
Interpreter

- Interpreter is a program that executes another program.
- The interpreter’s source language is the language in which the interpreter is written.
- The interpreter’s target language is the language in which the programs are written which the interpreter can execute.
Interpreters

Basic
x86

Interpreter for Basic, written in x86 machine code
Interpreting a Program

Basic program “C&C” running on x86 using interpretation
“C&C” x86 executable running on a PowerPC using hardware emulation
Typical Execution of Java Programs

Compiling “C&C” from Java to JVM code, and running the JVM code on a JVM running on an x86
Excursion: Making these Slides

Compiling these slides from \texttt{\LaTeX} to DVI to PostScript to PDF on x86 (MBP)
Excursion: Viewing these Slides

Viewing the slides on a PC
Summary: Language Processing

- Components:
  programs, translators, interpreters, machines

- T-diagrams

- Combination of interpretation and compilation is common

- Interpretation and compilation are ubiquitous in computing
Brief Introduction to CS4215

Administrative Matters

Language Processing

Inductive Definitions

What are Inductive Definitions?
• Extremal Clause
• Proofs by Induction
• Defining Sets by Rules in Java

ePL Outlook
Inductive Definitions

- We will frequently define a set by a collection of rules that determine the elements of that set.
  Example: the set of machine code programs for a particular virtual machine
- What does it mean to define a set by a collection of rules?
Example: Numerals

Numerals, in unary (base-1) notation

- Zero is a numeral;
- if \( n \) is a numeral, then so is \( \text{Succ}(n) \).

Examples

- Zero
- \( \text{Succ}(\text{Succ}(\text{Succ}(\text{Zero}))) \)
Example: Binary Trees

**Binary trees (w/o data at nodes)**

- *Empty* is a binary tree;
- if *l* and *r* are binary trees, then so is *Node*(*l*, *r*).

**Examples**

- *Empty*
- *Node(Node(Empty, Empty), Node(Empty, Empty))*
Examples (more formally)

- **Numerals**: The set $\text{Num}$ is defined by the rules

  $\text{Zero}$
  
  $\overbrace{\text{Succ} (n)}$

- **Binary trees**: The set $\text{Tree}$ is defined by the rules

  $\text{Empty}$
  
  $\overbrace{\text{Node}(t_l, t_r)}$
Defining a Set by Rules

- Given a collection of rules, what set does it define?
  - What is the set of numerals?
  - What is the set of trees?
- Do the rules pick out a unique set?
Defining a Set by Rules

There can be many sets that satisfy a given collection of rules.

- \( \text{Num} = \{ \text{Zero}, \text{Succ(Zero)}, \ldots \} \)
- \( \text{StrangeNum} = \text{Num} \cup \{ \infty, \text{Succ(\infty)}, \ldots \} \), where \( \infty \) is an arbitrary symbol

- Both \( \text{Num} \) and \( \text{StrangeNum} \) satisfy the rules defining numerals (i.e., the rules are true for these sets). Really?
**Num Satisfies the Rules**

\[
\begin{align*}
\text{Zero} & \quad \text{Succ}(n) \\
\end{align*}
\]

\[\text{Num} = \{\text{Zero}, \text{Succ}(\text{Zero}), \text{Succ}(\text{Succ}(\text{Zero})), \ldots\}\]

Does Num satisfy the rules?

- Zero \(\in\) Num. \(\checkmark\)
- If \(n \in\) Num, then Succ\((n)\) \(\in\) Num. \(\checkmark\)
**StrangeNum Satisfies the Rules**

\[
\begin{align*}
\text{Zero} &\quad n \\
\text{Succ}(n) &\quad \text{Succ}(\text{Succ}(\text{Zero}))
\end{align*}
\]

\[
\text{StrangeNum} = \{\text{Zero}, \text{Succ}(\text{Zero}), \text{Succ}(\text{Succ}(\text{Zero})), \ldots \} \cup \{\infty, \text{Succ}(\infty), \ldots \}
\]

Does \text{StrangeNum} satisfy the rules?

- Zero ∈ StrangeNum. √
- If \( n \) ∈ StrangeNum, then Succ(\( n \)) ∈ StrangeNum. √
Defining Sets by Rules

- Both $\textit{Num}$ and $\textit{StrangeNum}$ satisfy all rules.
- It is not enough that a set satisfies all rules.
- Something more is needed: an \textit{extremal} clause.
  - “and nothing else”
  - “the least set that satisfies these rules”
An inductively defined set is the least set that satisfies a given set of rules.

Example: \( \text{Num} \) is the least set that satisfies these rules:

- \( \text{Zero} \in \text{Num} \)
- if \( n \in \text{Num} \), then \( \text{Succ}(n) \in \text{Num} \).
Inductive Definitions

Question: What do we mean by “least”?  
Answer: The smallest with respect to the subset ordering on sets.

- Contains no “junk”, only what is required by the rules.
- Since $\text{StrangeNum} \supseteq \text{Num}$, $\text{StrangeNum}$ is ruled out by the extremal clause.
- $\text{Num}$ is “ruled in” because it has no “junk”.

What’s the Big Deal?

- Inductively defined sets “come with” an induction principle.
- Suppose $I$ is inductively defined by rules $R$.
- To show that every $x \in I$ has property $P$, it is enough to show that $P$ satisfies the rules of $R$.
- Sometimes called *structural induction* or *rule induction*.
Example: Parity of Numerals

- The numeral Zero has parity 0.
- Any numeral Succ(n) has parity 1 − p if p is the parity of n.
- Let P be the following property:
  Every numeral has either parity 0 or parity 1.

- Does P satisfy the rules:
  \[
  \begin{align*}
  &\text{Zero} &\quad n \\
  &\text{Succ}(n)
  \end{align*}
  \]
Induction Principle

To show that every $n \in \text{Num}$ has property $P$, it is enough to show:

- Zero has property $P$.
- if $n$ has property $P$, then $\text{Succ}(n)$ has property $P$.

This is just ordinary mathematical induction!
Induction Principle

To show that every tree has property $P$, it is enough to show that

- $Empty$ has property $P$.
- if $l$ and $r$ have property $P$, then so does $Node(l, r)$.

We call this structural induction on trees.
Example: Height of a Tree

- To show: Every tree has a height, defined as follows:
  - The height of $\text{Empty}$ is 0.
  - If $l$ has height $h_l$ and the tree $r$ has height $h_r$, then the tree $\text{Node}(l, r)$ has height $1 + \max(h_l, h_r)$.

- Clearly, every tree has at most one height, but does it have a height at all?
Example: height

- It may seem obvious that every tree has a height, but notice that the justification relies on structural induction!
  - An “infinite tree” does not have a height!
  - But the extremal clause rules out the infinite tree!
Example: height

- Formally, we prove that for every tree $t$, there exists a number $h$ satisfying the specification of height.
- Proceed by induction on the rules defining trees, showing that the property “there exists a height $h$ for $t$” satisfies these rules.
Example: height

- Rule 1: *Empty* is a tree.
  Does there exist $h$ such that $h$ is the height of *Empty*?
  Yes! Take $h=0$.

- Rule 2: *Node*(l, r) is a tree if l and r are trees.
  Suppose that there exists $h_l$ and $h_r$, the heights of l and r, respectively.
  Does there exist $h$ such that $h$ is the height of *Node*(l, r)?
  Yes! Take $h = 1 + \max(h_l, h_r)$. 

interface Num {}

class Zero implements Num {}

class Succ implements Num {
    public Num pred;
    Succ(Num p) {pred = p;}
}

Num my_num = new Zero();
Num my_other_num =
    new Succ(new Succ(new Zero()));
Encoding Trees in Java

interface Tree {}
class Empty implements Tree {}
class Node implements Tree {
    public Tree left, right;
    Node(Tree l, Tree r) {
        left = l; right = r;
    }
}
Tree my_tree =
    new Node(new Empty(),
             new Node(new Node(new Empty(),
                              new Empty()),
                     new Empty()));
Constructors and Rules

The constructors of the classes correspond to the rules in the inductive definition.

Numerals

- `new Zero()` is of type `Num`
- If `n` is of type `Num`, then `new Succ(n)` is of type `Num`

Trees

- `new Empty()` is of type `Tree`
- If `l` and `r` are of type `Tree`, then `new Node(l, r)` is of type `Tree`
Analogy with Java

- We assume an implicit extremal clause: no other classes implement the interface.
- The associated induction principle may be used to prove termination and correctness of functions.
Example: Height in Java

```java
interface Tree {
    public int height();
}

class Empty implements Tree {
    public int height() {return 0;}
}

class Node implements Tree {
    public Tree left, right;
    Node(Tree l, Tree r) {left = l; right = r;}
    public int height() {
        return 1 + max(height(left), height(right));
    }
}
```
Why does \( \text{height}(t) \) terminate for every tree \( t \)?

- For every \( t \) of type Tree, does there exist \( h \) such that \( \text{height}(t) \) returns \( h \)?
- Proof similar to above!
Summary

- An inductively defined set is the least set that satisfies a collection of rules.
- Rules have the form: “If $x_1 \in X$ and \ldots and $x_n \in X$, then $x \in X$.”
- Notation: $x_1 \ldots x_n$
Summary

- Inductively defined sets admit proofs by rule induction.
- For each rule
  \[ x_1 \quad \cdots \quad x_n \]
  \[ \hline \]
  \[ x \]
  assume that \( x_1 \in P, \ldots, x_n \in P \), and show that \( x \in P \).
- Conclude that every element of the set is in \( P \).
Syntax of ePL

\[
n \quad \text{true} \quad \text{false}
\]

\[
E \quad E_1 \quad E_2
\]

\[
p_1[E] \quad p_2[E_1, E_2]
\]

\[
P_1 = \{\}\,
\]

\[
P_2 = \{|,\&,+, -,*,/, =,>,,<\}.
\]
Outlook to Week 2

- Syntax (parsing)
- Semantics (static and dynamic)
- Implementations:
  - typing
  - small step interpreter
  - big step interpreter

(compiler-based implementations in Week 3)