01—Language Processing and Inductive Definitions

CS4215: Programming Language Implementation

Razvan Voicu
razvan@comp.nus.edu.sg

Week 1 (Jan 9 - 13, 2016)

Goal: Implementation Principles

- Implementation of major programming language concepts
- As “concise” as possible (with little *clutter*)
- 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 OCaml (previously Java)
- You will write virtual machines in OCaml (previously Java)
- You will get to learn how to build a domain-specific language.
- You will write toy programs in the toy languages
- Why OCaml? One of the most expressive languages.
- Extensive software support provided

Incremental and Exploratory

- 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 virtual machines
- Exploratory: Plenty of scope for exploration, from the most basic to the most advanced topics in each section
- Exploratory: Opportunities for exploring building domain-specific languages with a mini-project.

Overview of Module Content

1. Programming language processing tools and inductive definitions (1 hour)
2. OCaml as an Implementation Language
3. ePL: An Expression language
4. simPL: A simple functional language
5. polyPL: Adding Polymorphism and Exception
6. dPL: Algebraic Data Types
7. imPL: A Simple Imperative Language
8. oPL: A Simple Object-oriented Language
9. Domain-Specific Languages

Instructor

Răzvan Voicu

- Adjunct Associate Professor
- Currently a data scientist with Teralytics Pte Ltd (www.teralytics.net)
- Former full time lecturer in SoC, NUS

- razvan.voicu@teralytics.net (use for fast response)
  - Better yet, message me on Google Hangouts
- razvan@comp.nus.edu.sg (use for official purposes)

Administrative Matters

- Use IVLE
- Notes and slides (www; no textbook)
- Assignments (www; intensive work; marked; labs)
- Discussion forums (IVLE)
- Announcements (IVLE)
- Will have tutorial cum laboratory to focus on practical aspects.

Assessments

- Assignments 30%
- Mini-Project on DSL 25%
- Exam 45%

What's in a Programming Language

- Over 1000 programming languages recorded in Wikipedia
- Many approaches to developing languages
- Addressing software engineering concerns such as reuse, modularity
- Addressing many types of programmer backgrounds
- Addressing many types of project management concerns
- Addressing corporate branding needs
- Creators are often highly opinionated and with a very strong vision

Programming Paradigms

- Classical approach: a programming language follows a computational paradigm
- Imperative: instruct the machine what to do at each step
  - Distinguishing feature: assignment
  - Fortran, Algol68, Pascal, C
- Object-Oriented: extension of imperative, encapsulates data into objects
  - Distinguishing feature: inheritance
  - Smalltalk, C++, Java
- Functional Programming: computation based on mathematical function
  - Distinguishing feature: higher order programming
  - Lisp, Scheme, ML, Ocaml, Haskell, Scala
- Logic Programming: logic inference as computation
  - Distinguishing features: unification, backtracking
  - Prolog, Mercury, Oz

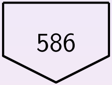
Programming Paradigms

- Concurrency
  - Distinguishing feature: concurrent constructs (threads, actors) as first class citizens
  - Erlang, Go

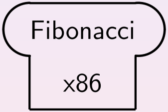
Other Well-Known Categories

- Scripting Languages: Python, Ruby, Javascript
- Domain Specific Languages
  - Query languages (SQL, GraphQL)
  - Document description languages (PDF, Postscript, Latex, HTML, SGML)
  - Hardware description languages (VHDL, Verilog)
  - UI description languages (XAML)
  - Software testing languages (Cucumber)
  - Financial product description languages
  - Risk modelling languages
  - Transaction description languages

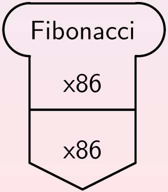
T-Diagrams



x86 Processor



Program “Fibonacci” (x86 code)

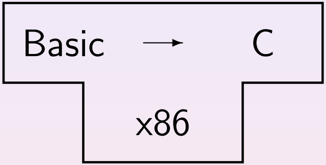


“Fibonacci” running on x86

Translators

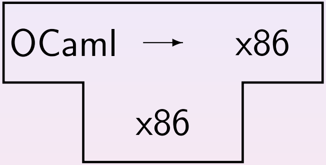
- Translator translates from one language—the *from-language*—to another language—the *to-language*
- Compiler translates from “high-level” language to “low-level” language
- De-compiler translates from “low-level” language to “high-level” language

T-Diagram of Translator



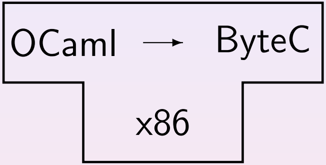
Basic-to-C compiler implemented in x86 machine code

OCaml Native Compiler



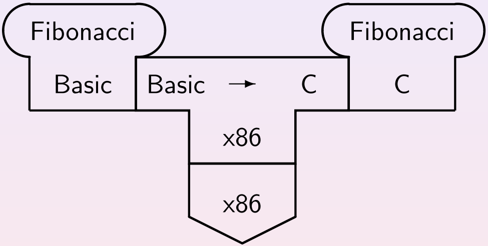
An OCaml native compiler, called `ocamlopt`, implemented in x86 machine code

OCaml ByteCodeCompiler



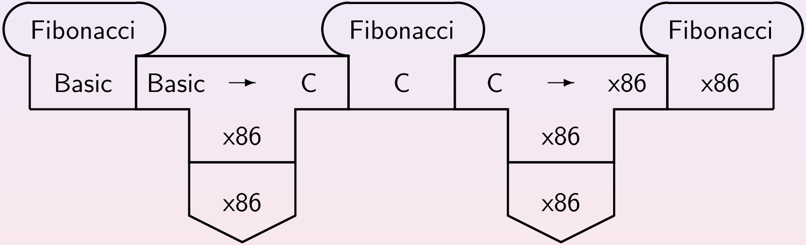
An OCaml bytecode compiler, called `ocamlc`, implemented in x86 machine code

Compilation



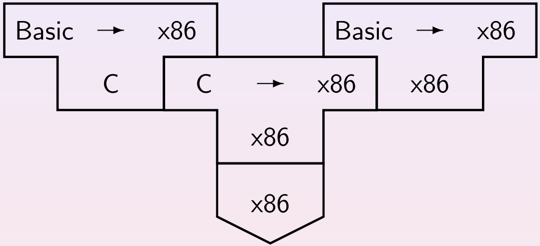
Compiling “Fibonacci” from Basic to C

Two-stage Compilation



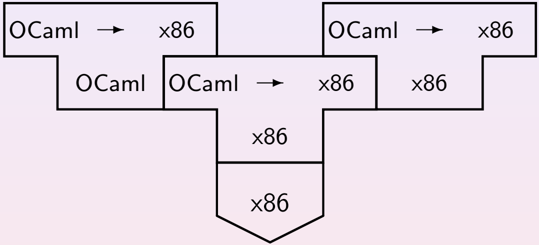
Compiling “Fibonacci” from Basic to C to x86 machine code

Compiling a Compiler



Compiling a Basic-to-x86 compiler from C to x86 machine code

Bootstrapping a Compiler



Compiling a OCaml-to-x86 compiler implemented in OCaml to run natively on x86 machine code

Interpreter

- Interpreter is 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

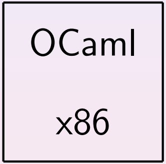
Interpreters

Basic

x86

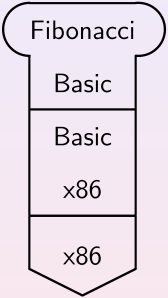
Interpreter for Basic, implemented in x86 machine code

Interpreter for OCaml



Interpreter for OCaml, implemented in x86 machine code that can be executed either interactively or in batch mode.

Interpreting a Program

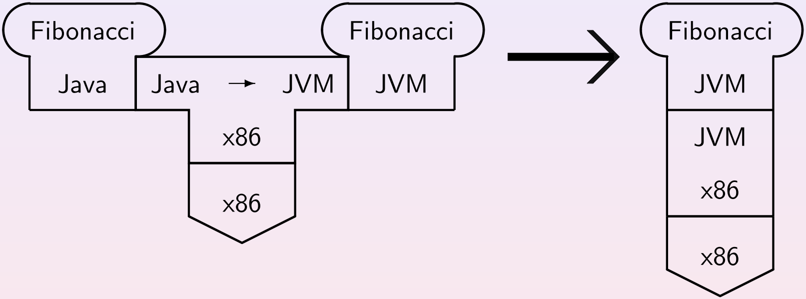


Basic program “Fibonacci”
running on x86 using interpretation

Hardware Emulation

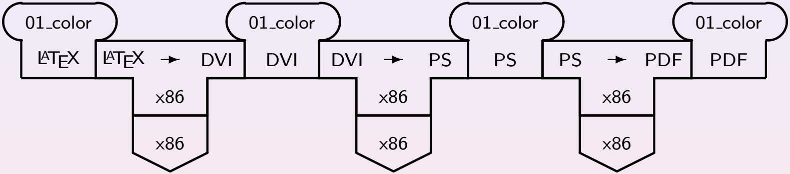
"Fibonacci" x86 executable running on a PowerPC using hardware emulation

Typical Execution of Java Programs



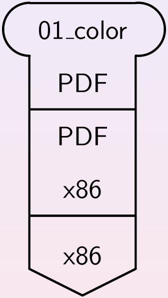
Compiling “Fibonacci” from Java to JVM code, and running the JVM code on a JVM running on an x86

Excursion: Making these Slides



Compiling these slides
from LaTeX to DVI to PostScript to PDF on x86 (PC)

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

Inductive Definitions

- We will frequently define a set by a collection of rules that determine the elements of that set.
  Example: the set of programs for a particular programming language
- 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 $Succ(n)$.

Examples

- *Zero*
- $Succ(Succ(Succ(Zero)))$

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 *Num* is defined by the rules

$$\frac{}{Zero \in Num} \qquad \frac{n \in Num}{Succ(n) \in Num}$$

- Binary trees: The set *Tree* is defined by the rules

$$\frac{}{Empty \in Tree} \qquad \frac{t_l \in Tree \qquad t_r \in Tree}{Node(t_l, t_r) \in Tree}$$

Examples (formally and implicitly)

- Numerals: The set *Num* is defined by the rules

$$\frac{}{Zero} \qquad\qquad \frac{n}{Succ(n)}$$

- Binary trees: The set *Tree* is defined by the rules

$$\frac{}{Empty} \qquad\qquad \frac{t_l \qquad t_r}{Node(t_l, t_r)}$$

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.
  - $Num = \{Zero, Succ(Zero), \ldots\}$
  - $StrangeNum = Num \cup \{\infty, Succ(\infty), \ldots\}$, where ∞ is an arbitrary symbol
- Both *Num* and *StrangeNum* satisfy the rules defining numerals (i.e., the rules are true for these sets). Really?

Num Satisfies the Rules

$$\frac{}{Zero \in Num} \qquad \frac{n \in Num}{Succ(n) \in Num}$$

$Num = \{Zero, Succ(Zero), Succ(Succ(Zero)), \ldots\}$
Does *Num* satisfy the rules?

- $Zero \in Num$. $\surd$
- If $n \in Num$, then $Succ(n) \in Num$. $\surd$

StrangeNum Satisfies the Rules

$$\frac{}{Zero \in Num} \qquad \frac{n \in Num}{Succ(n) \in Num}$$

$StrangeNum =$
$\{Zero, Succ(Zero), Succ(Succ(Zero)), \ldots\} \cup \{\infty, Succ(\infty), \ldots\}$
Does *StrangeNum* satisfy the rules?

- $Zero \in StrangeNum$. $\surd$
- If $n \in StrangeNum$, then $Succ(n) \in StrangeNum$. $\surd$

This is despite the fact that ∞ not explicitly mentioned in the rules.

Defining Sets by Rules

- Both *Num* and *StrangeNum* satisfy all rules.
- It is not enough that a set satisfies all rules.
- Something more is needed: an *extremal* clause.
    - "and nothing else"
    - "the least set that satisfies these rules"

Inductive Definitions

- An inductively defined set is the least set that satisfies a given set of rules.
- Example: *Num* is the least set that satisfies these rules:
  - $\mathit{Zero} \in \mathit{Num}$
  - if $n \in \mathit{Num}$, then $\mathit{Succ}(n) \in \mathit{Num}$.

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 $\mathit{StrangeNum} \supsetneq \mathit{Num}$, *StrangeNum* is ruled out by the extremal clause.
- *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 $\dfrac{}{P(Zero)}$ $\dfrac{P(n)}{P(Succ(n))}$?

Induction Principle

- To show that every $n \in \mathit{Num}$ has property P, it is enough to show:
  - Zero has property P.
  - if n has property P, then $\mathit{Succ}(n)$ 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 *Empty* is 0.
  - If l has height h_l and the tree r has height h_r, then the tree $Node(l, r)$ 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 $h = 1 + max(h_l, h_r)$.

Encoding Numerals in Java

```
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 Numerals in OCaml

```
type num =
  | Zero
  | Succ of num

let my_num = Zero

let my_other_num:num = Succ (Succ 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()));
```

Encoding Trees in OCaml

```
type tree =
  | Empty
  | Node of tree * tree

let my_tree =
  Node(Empty,
       Node(Node(Empty,Empty),
            Empty))
```

Constructors and Rules in OCaml

- The algebraic data construction corresponds directly to the rules in the inductive definition.
- Numerals
  - `Zero` is of type `Num`
  - if `n` is of type `Num`, then `Succ(n)` is of type `Num`
- Trees
  - `Empty` is of type `Tree`
  - if `l` and `r` are of type `Tree`, then `Node(l,r)` is of type `Tree`

Extremal Clause with Java/OCaml

- We assume an implicit extremal clause: no other kinds of objects/values can be constructed for each given type.
- The associated induction principle may be used to prove termination and correctness of functions.

Example: Height in OCaml

```
let rec height (t:tree) : int =
  match t with
    | Empty -> 0
    | Node (l,r) -> 1 + (max (height l) (height r))

let h = height my_tree

let _ = print_endline ("height of my_tree is "
                                    ^(string_of_int h))
```

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 ... and $x_n \in X$, then $x \in X$."
- Notation:

$$\frac{x_1 \in X \qquad \cdots \qquad x_n \in X}{x \in X}$$

Summary

- Inductively defined sets admit proofs by rule induction.
- For each set, with rules of the form:

$$\frac{x_1 \in X \qquad \cdots \qquad x_n \in X}{x \in X}$$

  We can proof this property inductively using:

$$\frac{P(x_1) \qquad \cdots \qquad P(x_n)}{P(x)}$$

- Conclude that every element of the set satisfies P.