603AA - Principles of Programming Languages [PLP-2016]

Andrea Corradini
andrea@di.unipi.it
Department of Computer Science, Pisa
Academic Year 2016/17
Admins...

• Office hours: **Wednesday, 3-6 pm**
  – or by appointment <andrea@di.unipi.it>

• Page on Moodle platform:
  – Contains PDFs of relevant chapter

• Tutor: **Lillo Galletta**  [galletta@di.unipi.it](mailto:galletta@di.unipi.it)
  – Contact by email, indicating the topics you would like to discuss
  – Based on the requests, Lillo can meet you individually or in group

• No lesson tomorrow, September 23
  – Other possible slot?
• Programming languages and Abstract Machines
• Compilation and interpretation schemes
• Cross compilation
• Bootstrapping
• Compilers
Definition of Programming Languages

• A PL is defined via **syntax**, **semantics** and **pragmatics**

• The **syntax** is concerned with the form of programs: how expressions, commands, declarations, and other constructs must be arranged to make a well-formed program.

• The **semantics** is concerned with the meaning of (well-formed) programs: how a program may be expected to behave when executed on a computer.

• The **pragmatics** is concerned with the way in which the PL is intended to be used in practice. Pragmatics include the *paradigm(s)* supported by the PL.
Paradigms

A **paradigm** is a style of programming, characterized by a particular selection of key concepts

- **Imperative programming**: variables, commands, procedures.
- **Object-oriented (OO) programming**: objects, methods, classes.
- **Concurrent programming**: processes, communication.
- **Functional programming**: values, expressions, functions.
- **Logic programming**: assertions, relations.

In general, classification of languages according to paradigms is misleading.
Implementation of a Programming Language \( L \)

- Programs written in \( L \) must be executable
- Language \( L \) implicitly defines an Abstract Machine \( M_L \) having \( L \) as machine language
- Implementing \( M_L \) on an existing host machine \( M_o \) (via compilation, interpretation or both) makes programs written in \( L \) executable
Abstract Machine for a Language \( L \)

- Given a programming language \( L \), an Abstract Machine \( M_L \) for \( L \) is a collection of data structures and algorithms which can perform the storage and execution of programs written in \( L \)
- An abstraction of the concept of hardware machine
- Structure of an abstract machine:

<table>
<thead>
<tr>
<th>Memory</th>
<th>Interpreter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programs</td>
<td>Operations and Data Structures for:</td>
</tr>
<tr>
<td>Data</td>
<td>- Primitive Data processing</td>
</tr>
<tr>
<td></td>
<td>- Sequence control</td>
</tr>
<tr>
<td></td>
<td>- Data Transfer control</td>
</tr>
<tr>
<td></td>
<td>- Memory management</td>
</tr>
</tbody>
</table>
General structure of the Interpreter

Sequence control

Fetch next instruction

Decode

Fetch operands

Choose

Execute op_1

Execute op_2

...

Execute op_n

Execute HALT

Data control

Operations

Data control

Store the result

stop
The Machine Language of an AM

• Given and Abstract machine \( M \), the machine language \( L_M \) of \( M \)
  – includes all programs which can be executed by the interpreter of \( M \)
• Programs are particular data on which the interpreter can act
• The components of \( M \) correspond to components of \( L_M \), eg:
  – Primitive data types
  – Control structures
  – Parameter passing and value return
  – Memory management
• Every Abstract Machine has a unique Machine Language
• A programming language can have several Abstract Machines
An example: the Hardware Machine

- The language
- The memory
- The interpreter
- Operations and Data Structures for:
  - Primitive Data processing
  - Sequence control
  - Data Transfer control
  - Memory management
The Java Virtual Machine

- The language
- The memory
- The interpreter
- Operations and Data Structures for:
  - Primitive Data processing
  - Sequence control
  - Data Transfer control
  - Memory management

The core of a JVM interpreter is basically this:

```java
do {
    byte opcode = fetch an opcode;
    switch (opcode) {
        case opCode1:
            fetch operands for opCode1;
            execute action for opCode1;
            break;
        case opCode2:
            fetch operands for opCode2;
            execute action for opCode2;
            break;
        ...
    }
} while (more to do)
```

~ 160 opcodes
Implementing an Abstract Machine

• Each abstract machine can be implemented in **hardware** or in **firmware**, but if it is high-level this is not convenient in general.
• An abstract machine $M$ can be implemented over a **host machine** $M_0$, which we assume is already implemented.
• The components of $M$ are realized using data structures and algorithms implemented in the machine language of $M_0$.
• Two main cases:
  – The interpreter of $M$ coincides with the interpreter of $M_0$
    • $M$ is an **extension** of $M_0$
    • other components of the machines can differ
  – The interpreter of $M$ is different from the interpreter of $M_0$
    • $M$ is **interpreted** over $M_0$
    • other components of the machines may coincide
Hierarchies of Abstract Machines

• Implementation of an AM with another can be iterated, leading to a hierarchy (onion skin model)

• Example:

![Diagram of a hierarchy of abstract machines]

- Hardware machine
- Firmware machine
- Operating System machine
- Intermediate machine (Java Bytecode)
- HL machine (Java)
- Web machine (browser etc.)
- Web Service machine (languages for web services)
- E-Business machine (on-line commerce applications)
Implementing a Programming Language

• $L$ high level programming language
• $M_L$ abstract machine for $L$
• $M_o$ host machine

**Pure Interpretation**
- $M_L$ is interpreted over $M_o$
- Not very efficient, mainly because of the interpreter (fetch-decode phases)

**Pure Compilation**
- Programs written in $L$ are translated into equivalent programs written in $L_o$, the machine language of $M_o$
- The translated programs can be executed directly on $M_o$
  • $M_L$ is not realized at all
  • Execution more efficient, but the produced code is larger

• Two limit cases that almost never exist in reality
Pure Interpretation

• Program $P$ in $L$ as a partial function on $D$:
  \[ \mathcal{P}_L : D \rightarrow D \]

• Set of programs in $L$: $\mathcal{P}rog^L$

• The interpreter defines a function

\[ \mathcal{I}_L^0 : (\mathcal{P}rog^L \times D) \rightarrow D \quad \text{such that} \quad \mathcal{I}_L^0 (\mathcal{P}_L, Input) = \mathcal{P}_L (Input) \]
Pure [cross] Compilation

A compiler from $L$ to $LO$ defines a function

$$\mathcal{L}, \mathcal{L}_o : \text{Prog}^L \rightarrow \text{Prog}^{LO}$$

such that if

$$\mathcal{L}, \mathcal{L}_o (\mathcal{P}^L) = \mathcal{P}c^{LO},$$

then for every input we have

$$\mathcal{P}^L (\text{Input}) = \mathcal{P}c^{LO} (\text{Input})$$
Compilers versus Interpreters

• Compilers efficiently fix decisions that can be taken at compile time to avoid to generate code that makes this decision at runtime
  – Type checking at compile time vs. runtime
  – Static allocation
  – Static linking
  – Code optimization

• Compilation leads to better performance in general
  – Allocation of variables without variable lookup at run time
  – Aggressive code optimization to exploit hardware features

• Interpretation facilitates interactive debugging and testing
  – Interpretation leads to better diagnostics of a programming problem
  – Procedures can be invoked from command line by a user
  – Variable values can be inspected and modified by a user
Compilation + Interpretation

• All implementations of programming languages use both. At least:
  – Compilation (= translation) from external to internal representation
  – Interpretation for I/O operations (runtime support)

• Can be modeled by identifying an Intermediate Abstract Machine $M_I$ with language $L_I$
  – A program in $L$ is compiled to a program in $L_I$
  – The program in $L_I$ is executed by an interpreter for $M_I$
Compilation + Interpretation with Intermediate Abstract Machine

- The “pure” schemes as limit cases
- Let us sketch some typical implementation schemes...

**Diagram Description:**
- **Input data** to **Input** machine.
- **Program written in L** goes to **Compiler from L to Li**.
- **Program written in Li** goes to **Interpreter for Li written in Lo or RTS**.
- **Interpretation** of the program is done on **MO**.
- **Output data** from the interpreter.

**Flowchart Notes:**
- **Compilation on MA** for programs written in **L**.
- **Execution on MO** for programs written in **Li**.
- **Compiler from L to Li** transforms programs from **L** to **Li**.
- **Interpreter for Li written in Lo or RTS** executes programs written in **Li**.
Virtual Machines as Intermediate Abstract Machines

• Several language implementations adopt a compilation + interpretation schema, where the Intermediate Abstract Machine is called Virtual Machine

• Adopted by Pascal, Java, Smalltalk-80, C#, functional and logic languages, and some scripting languages
  – Pascal compilers generate P-code that can be interpreted or compiled into object code
  – Java compilers generate bytecode that is interpreted by the Java virtual machine (JVM)
  – The JVM may translate bytecode into machine code by just-in-time (JIT) compilation
Compilation and Execution on Virtual Machines

- Compiler generates intermediate program
- Virtual machine interprets the intermediate program
- Portability!
Pure Compilation and Static Linking

- Adopted by the typical Fortran systems
- Library routines are separately linked (merged) with the object code of the program

```
extern printf();
```
Compilation, Assembly, and Static Linking

• Facilitates debugging of the compiler
Compilation, Assembly, and Dynamic Linking

- Dynamic libraries (DLL, .so, .dylib) are linked at run-time by the OS (via stubs in the executable)

```c
extern printf();
```
Preprocessing

• Most C and C++ compilers use a preprocessor to import header files and expand macros

```
#include <stdio.h>
#define N 99
...
for (i=0; i<N; i++)
```

```
for (i=0; i<99; i++)
```

Source Program  -->  Preprocessor  -->  Modified Source Program  -->  Compiler  -->  Assembly or Object Code
The CPP Preprocessor

• Early C++ compilers used the CPP preprocessor to generated C code for compilation
Compilers, graphically

• Three languages involved in writing a compiler
  – Source Language (S)
  – Target Language (T)
  – Implementation Language (I)

• T-Diagram:

• If $I = T$ we have a **Host Compiler**
• If $S$, $T$, and $I$ are all different, we have a **Cross-Compiler**
Composing compilers

• Compiling a compiler we get a new one: the result is described by composing T-diagrams

• The shape of the basic transformation, in the most general case, is the following:

• Note: by writing this transformation, we implicitly assume that we can execute programs written in $M$
Composing compilers: an example

• A compiler of S to M can be written in any language having a host compiler for M.

Example:
S Pascal
I C
M 68000

• By compiling it we get a host compiler of S for M.
Bootstrapping

- **Bootstrapping**: techniques which use partial/inefficient compiler versions to generate complete/better ones
- Often compiling a translator programmed in its own language
- Why writing a compiler in its own language?
  - it is a non-trivial test of the language being compiled
  - compiler development can be done in the higher level language being compiled.
  - improvements to the compiler’s back-end improve not only general purpose programs but also the compiler itself
  - it is a comprehensive consistency check as it should be able to reproduce its own object code
Compilers: Portability Criteria

• Portability
  – Retargetability
  – Rehostability

• A **retargetable** compiler is one that can be modified easily to generate code for a new target language

• A **rehostable** compiler is one that can be moved easily to run on a new machine

• A portable compiler may not be as efficient as a compiler designed for a specific machine, because we cannot make any specific assumption about the target machine
Using Bootstrapping to port a compiler

- We have a host compiler/interpreter of L for M
- Write a compiler of L to N in language L itself

Example:
L Pascal
M P-code
Bootstrapping to optimize a compiler

• The efficiency of programs and compilers:
  – Efficiency of programs:
    • memory usage
    • runtime
  – Efficiency of compilers:
    • Efficiency of the compiler itself
    • Efficiency of the emitted code

• Idea: Start from a simple compiler (generating inefficient code) and develop more sophisticated version of it. We can use bootstrapping to improve performance of the compiler.
Bootstrapping to optimize a compiler

- We have a host compiler of ADA to M
- Write an optimizing compiler of ADA to M in ADA
Full Bootstrapping

• A full bootstrap is necessary when building a new compiler from scratch.

• **Example:**
  • We want to implement an **Ada** compiler for machine **M**. We don’t have access to any **Ada** compiler.
  • Idea: **Ada** is very large, we will implement the compiler in a subset of **Ada** (call it **Ada_0**) and bootstrap it from a subset of **Ada** compiler in another language (e.g. **C**).
Full Bootstrapping (2)

• **Step 1:** build a compiler of Ada₀ to M in another language, say C

```
Ada₀  M
  C
```

• **Step 2:** compile it using a host compiler of C for M

```
Ada₀  M
  C  C  M
     M
```

• **Note:** new versions would depend on the C compiler for M
Full Bootstrapping (3)

• **Step 3:** Build another compiler of \( \text{Ada}_0 \) in \( \text{Ada}_0 \)

  ![Diagram](image_url)

  - \( \text{Ada}_0 \)
  - M
  - \( \text{Ada}_0 \)

• **Step 4:** compile it using the \( \text{Ada}_0 \) compiler for M

  ![Diagram](image_url)

  - \( \text{Ada}_0 \)
  - M
  - \( \text{Ada}_0 \)
  - v1
  - M
  - M

• **Note:** C compiler is no more necessary
**Full Bootstrapping (4)**

- **Step 5:** Build a full compiler of Ada in Ada$_0$

![Diagram showing the process of bootstrapping](image)

- **Step 4:** Compile it using the second Ada$_0$ compiler for M

![Diagram showing the process of bootstrapping](image)

- Future versions of the compiler can be written directly in Ada