Abstract Machines, Interpreters, Compilers

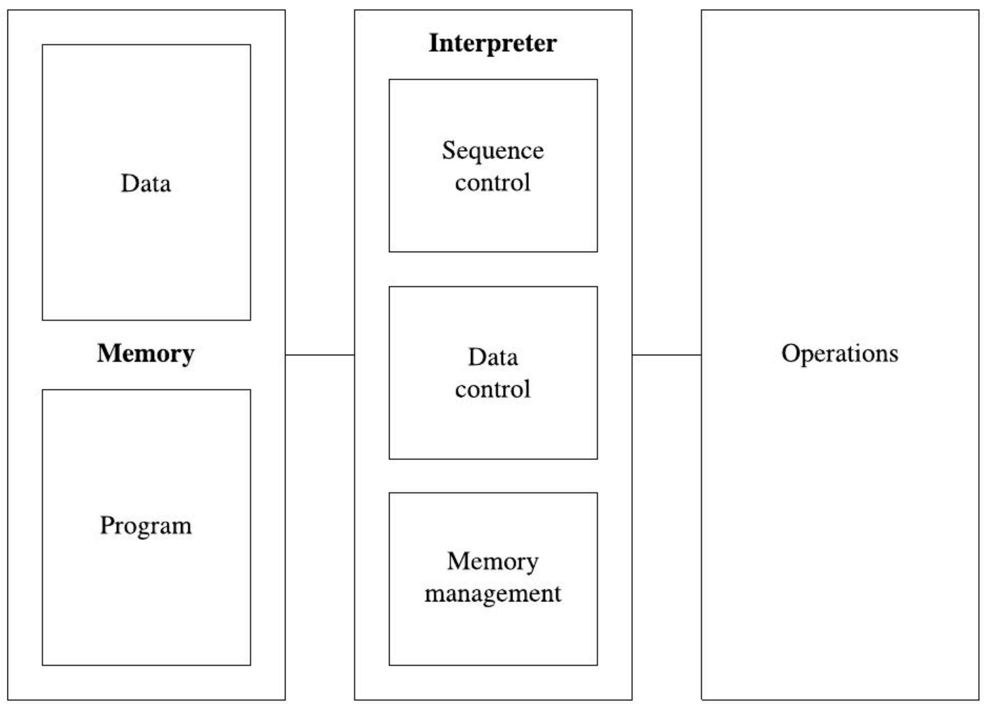
Based on the slides of Maurizio Gabbrielli

One Machine, One language

- One **physical machine** exists to run its **language**
- Language and machine come together
  - A machine corresponds to its language
  - A language can be run from multiple machines
- Heart of a physical machine:
  - The fundamental cycle *fetch-decode-execute*

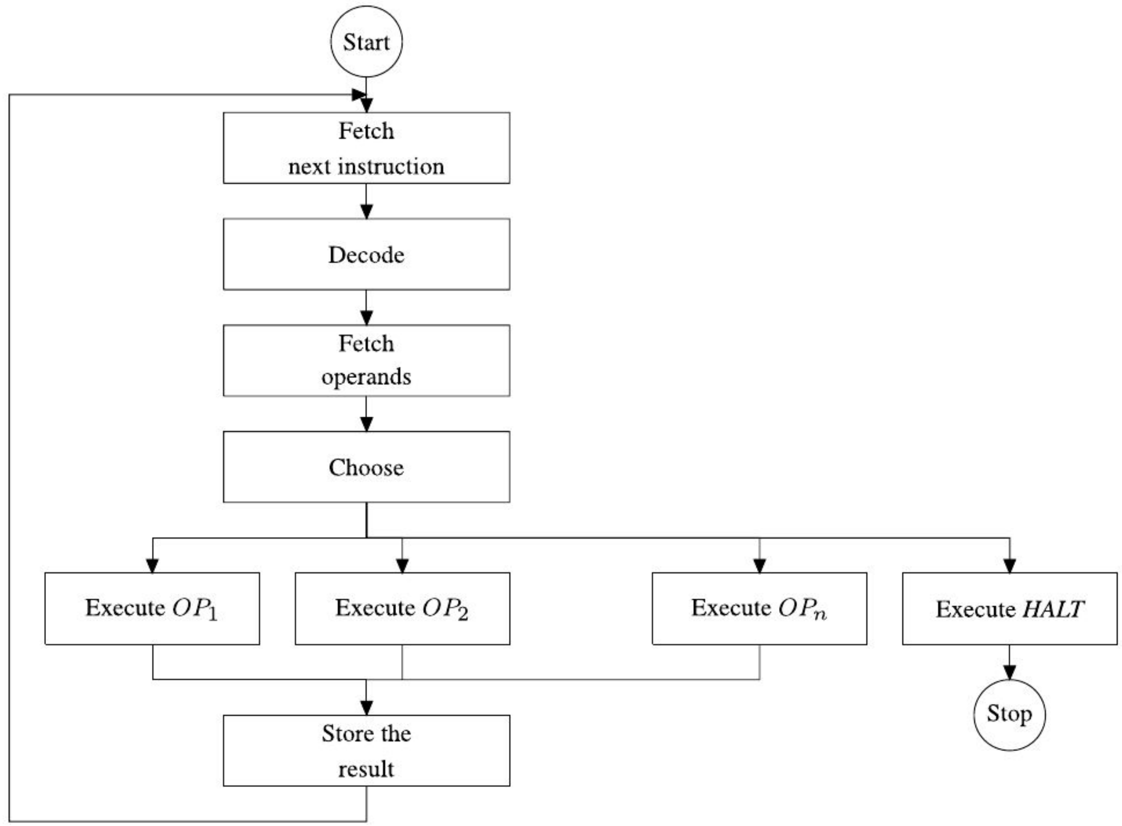
Abstract machine

- An Abstract Machine (AM) is a set of data structures and algorithms that can store and run programs
- Abstraction of the concept of a physical computer



Interpreter

- Component that interprets instructions
- The structure of the interpreter is the same for any AM
- AM ~ Store + Interpreter



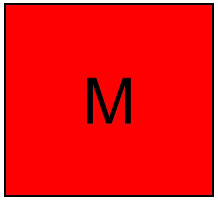
Machine language

- **M** → Abstract machine
- $\mathbf{L_M}$ → Machine language of M
- $\mathbf{L_M}$ It is the language that is "understood" by the interpreter of **M**
  - The programs are special primitive data on which the interpreter works

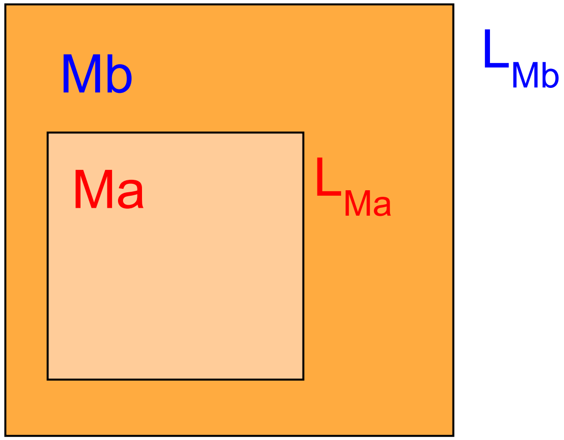
The Hardware Machine

- A conventional processor is a (very concrete form of) Abstract Machine
- Its language is the machine language

L_M



Chinese boxes



Making an abstract machine

Different ways:

1. Realization in **Hardware**
2. Emulation or simulation via **Firmware**
3. Interpretation or simulation via **Software**

1 is always theoretically possible but

- Used only for low-level machines or dedicated machines
- Maximum speed
- No flexibility

Making an abstract machine

Different ways:

1. Realization in **Hardware**
2. Emulation or simulation via **Firmware**
3. Interpretation or simulation via **Software**

2: data structures and algorithms but realized by micro-programs, residing in a read-only memory

- Microprogrammable (physical) machine
- High speed
- Greater flexibility than pure HW.

Making an abstract machine

Different ways:

1. Realization in **Hardware**
2. Emulation or simulation via **Firmware**
3. Interpretation or simulation via **Software**

3: data structures and algorithms of the abstract machine but realized through programs written in the language of the host machine

- Any Host machine works
- Less speed
- Maximum flexibility.

More Formally. What is a Program?

- Partial Function: f: A $\rightarrow$ B

  Correspondence between elements of A and B that can be undefined on some a $\in$ A

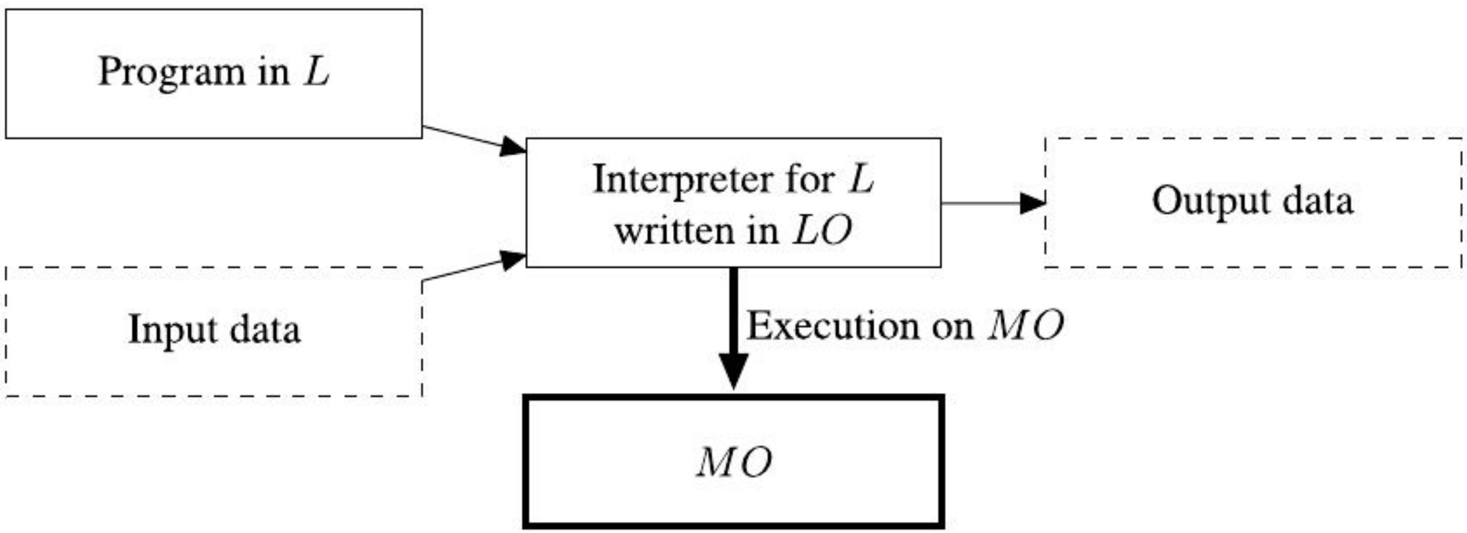
- $\mathcal{P}^{\mathcal{L}}$ indicates a program written in the language $\mathcal{L}$
- $\mathcal{P}^{\mathcal{L}}$ performs a partial function

$$\mathcal{P}^{\mathcal{L}} : \mathcal{D} \rightarrow \mathcal{D}$$

$$\mathcal{P}^{\mathcal{L}}(Input) = Output$$

Purely Interpretative Implementation

M_L is developed writing an interpreter of L that runs on M0



Definition 1.3 (Interpreter) An interpreter for language $\mathscr{L}$, written in language $\mathscr{L}o$, is a program which implements a partial function:

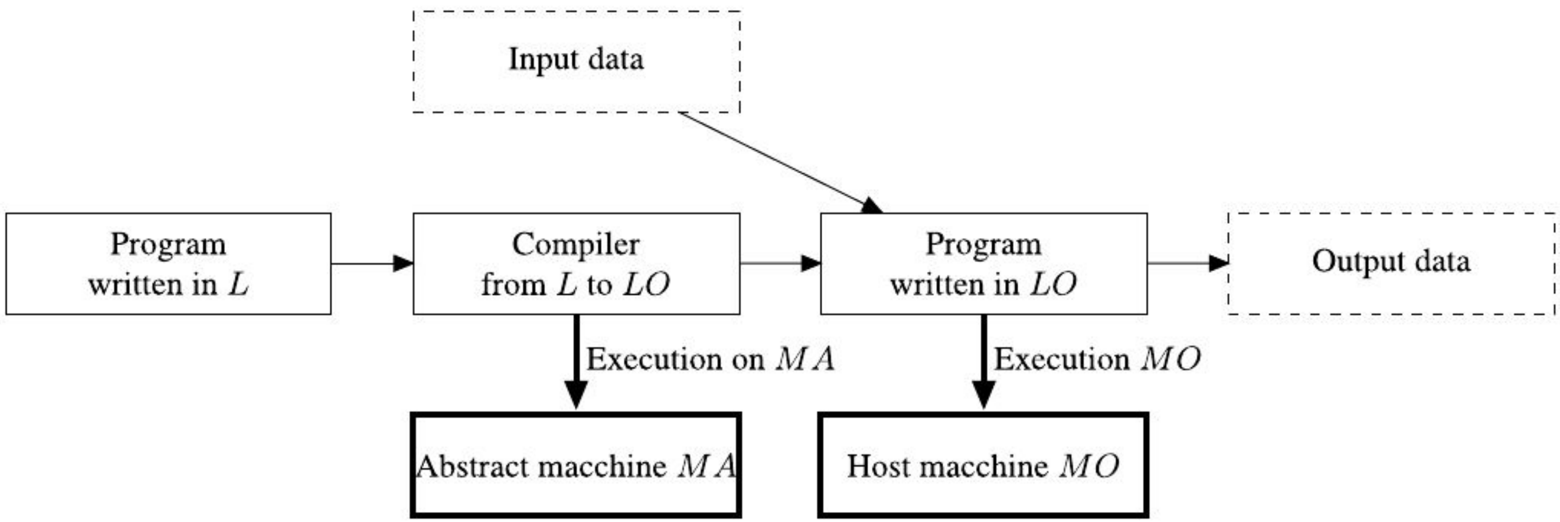
$$\mathscr{I}^{\mathscr{L}o}_{\mathscr{L}} : (\mathscr{P}rog^{\mathscr{L}} \times \mathscr{D}) \to \mathscr{D} \quad \text{such that } \mathscr{I}^{\mathscr{L}o}_{\mathscr{L}}(\mathscr{P}^{\mathscr{L}}, Input) = \mathscr{P}^{\mathscr{L}}(Input) \quad (1.1)$$

Purely Compilative Implementation

- The programs in L are translated to L0 programs
- Translation made by other program

$$\mathcal{C}^{La}_{L,\,LO}$$

The compiler from L To LO written in La



Pure compilative Implementation, 2

Definition 1.4 (Compiler) A compiler from $\mathscr{L}$ to $\mathscr{L}o$ is a program which implements a function:

$$\mathscr{C}_{\mathscr{L},\mathscr{L}o} : \mathscr{P}rog^{\mathscr{L}} \rightarrow \mathscr{P}rog^{\mathscr{L}o}$$

such that, given a program $\mathscr{P}^{\mathscr{L}}$, if

$$\mathscr{C}_{\mathscr{L},\mathscr{L}o}(\mathscr{P}^{\mathscr{L}}) = \mathscr{P}c^{\mathscr{L}o}, \tag{1.2}$$

then, for every *Input* $\in \mathscr{D}$[4]:

$$\mathscr{P}^{\mathscr{L}}(Input) = \mathscr{P}c^{\mathscr{L}o}(Input) \tag{1.3}$$

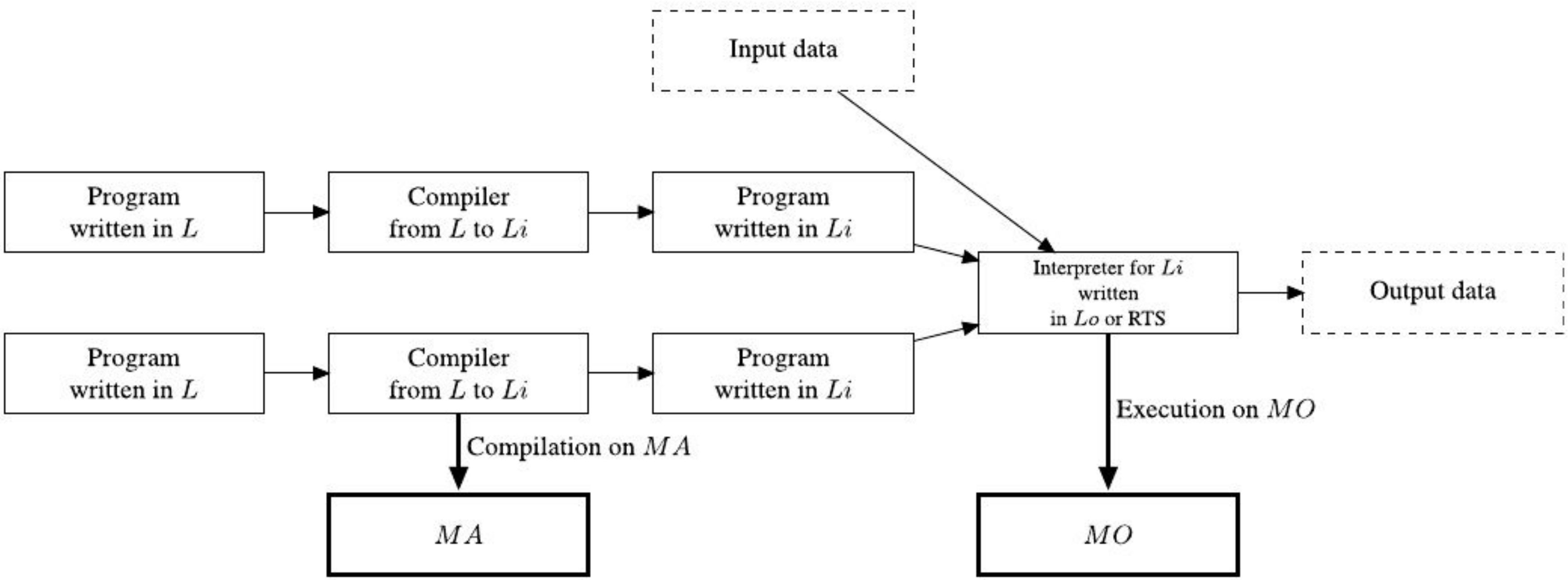
Compilation or interpretation?

- Pure Interpretative Implementation:
  - Poor machine efficiency $\mathbf{M}_{\mathcal{L}}$
  - Good flexibility and portability
  - Easy to run-time interaction (e.g. debugging)
- Pure Compilative Implementation:
  - Difficult, given the distance between $\mathcal{L}$ and **L0**
  - Good efficiency:
    - cost-decoding compiler load
    - Each instruction is translated only once
  - Low flexibility
  - Loss of information on the structure (abstraction) of the source program
  - More memory of product code (these days it does not matter that much)

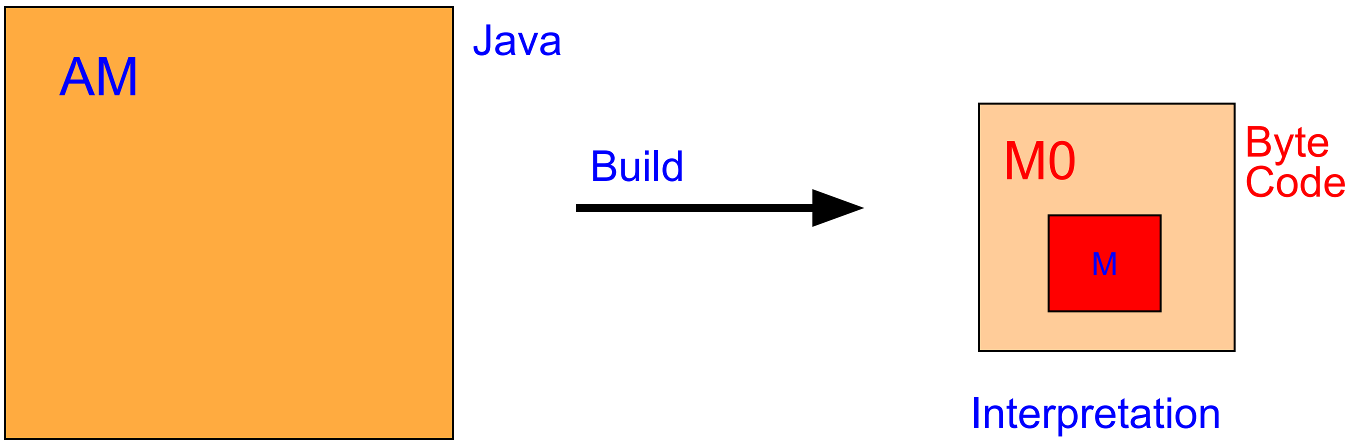
In the real world

Both techniques coexist

1) Some instructions (e.g. input/output) are always interpreted
2) Programs are translated in internal representation or intermediate code

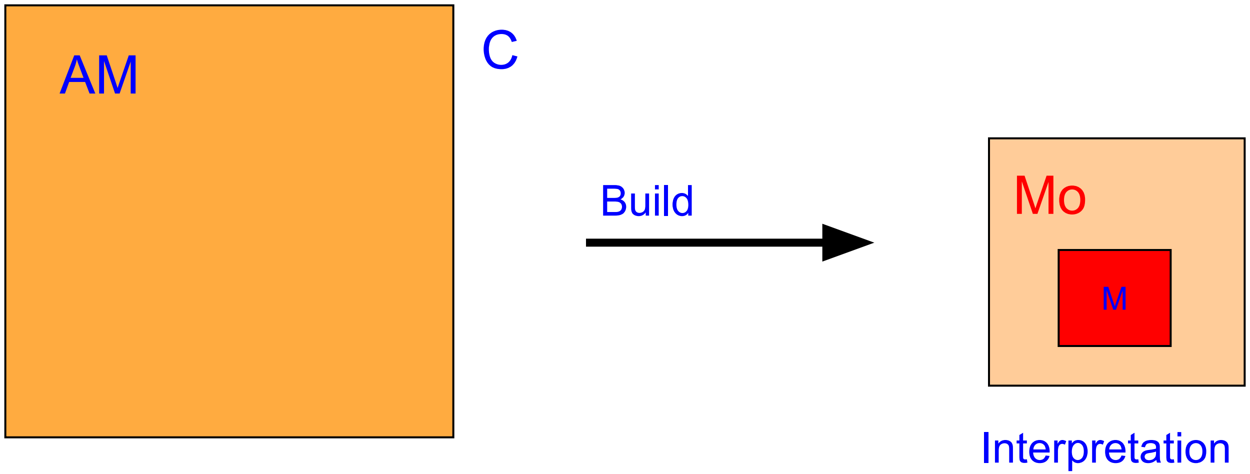


Example of interpretive type: Java



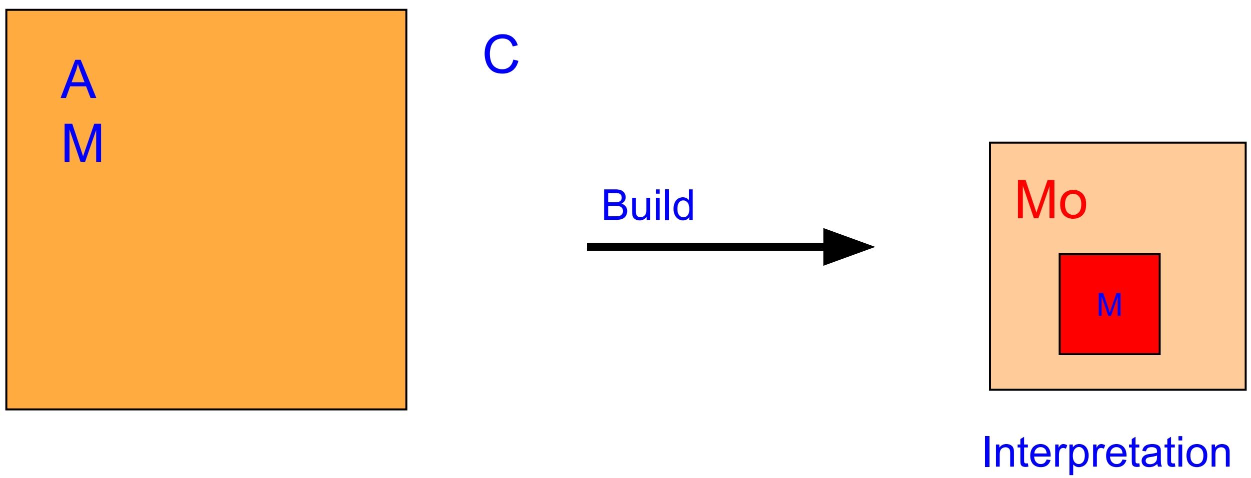
L_{AM} = Java
L_{M0} = Bytecode
M0 = JVM
M = suitable machine running the interpreter of the JVM

Example of compilative type: C



L_{AM} = C
L_{Mo} = Code generated by compiler
M0 =?
M seems pointless: it coincides with M0?

Implementation of compilative type: C



L_{AM} = C

L_{Mo} = Language generated by compiler + Support for memory management, I/O etc.

Mo = M + interpreter for run-time support calls

M = host Machine

What to translate and what to interpret

- In principle:
  - Translation for those constructs of $\mathcal{L}$ that correspond closely to constructs of $\mathcal{L}0$;
  - Simulation for others.

- Compilative Type Solution
  - Privileges efficiency
- Interpretive type Solution
  - Privileges flexibility

Real languages

- Languages typically implemented in a compilative way
  - **C, C++, Fortran, Pascal, ADA**
- Languages typically implemented in an interpretive way
  - **LISP, ML, Perl, Postscript, Prolog, Smalltalk, Java, Python**

Just-in-time compilation (not in the book!)

- AKA dynamic translation or run-time compilation
- Bytecode translation to machine code at run time (more difficult if from source code)
- JIT compiler analyses the code being executed and translates parts of the code where you can obtain a speedup
  - can be done per-file, per-function or arbitrary code fragment
  - code cached and reused later without needing to be recompiled
  - sometimes better performance than static compilation (some optimization only possible at runtime)
- Combines the speed of compiled code with the flexibility of interpretation
- JIT causes a delay in the initial execution (warm-up time)
- Classical example is the JIT compilers for Java, browsers for JavaScript

Partial Evaluation

If you know part of the input of a program **P** that program can be **Specialized.**

Conditional evaluation: (knowing we have X = 3 in input)

If x > 1 then A else B → A

Symbolic Evaluation Expressions:

20 + x-22 → x-2 (for each value of x)

20 + x-22 → 1 (knowing that x = 3 input)

Partial Evaluation

- **P^L(X, Y)** Program with input data X and Y, written in the language **L**

$$\mathscr{P}eval_{\mathscr{L}} : (\mathscr{P}rog^{\mathscr{L}} \times \mathscr{D}) \to \mathscr{P}rog^{\mathscr{L}}$$

$$\mathscr{P}eval_{\mathscr{L}}(P, D_1) = P'$$

$$\mathscr{I}_{\mathscr{L}}(P, (D_1, Y)) = \mathscr{I}_{\mathscr{L}}(P', Y)$$

First Futamura projection

- $\mathbf{Peval_{L0}}$ **(X, Y)** Partial evaluator for language **L0**
- $\mathbf{Int^{L0}_{L1}}$ Language interpreter for L1, written in **L0**
- $\mathbf{P^{L1}}$ Program written in **L1**
- D Program Data

Consider a specific program $\mathbf{P1^{L1}}$
If we do the partial evaluation of $\mathbf{Int^{L0}_{L1}}$ specializing it for P1 what happens?

$$\mathbf{Peval_{L0}\,(Int^{L0}_{L1},\ P1^{L1})\ ?}$$

First Futamura projection

- **$Peval_{L0}$ (X, Y)** Partial evaluator for language **L0**
- **Int^{L0}_{L1}** Language interpreter for L1, written in **L0**
- **P^{L1}** Program written in **L1**
- D Program Data

Consider a specific program **$P1^{L1}$**
If we do the partial evaluation of **Int^{L0}_{L1}** specializing it for P1 what happens?

$Peval_{L0}(Int^{L0}_{L1}, P1^{L1}) = P1'^{L0}$ And by definition of Peval you have
$Int^{L0}_{L1}(P1^{L1}, D) = P1'^{L0}(D)$

We have produced a program **$P1'^{L0}$ (D),** written in L0, which is functionally equivalent to **$Int^{L0}_{L1}(P1^{L1}, D)$** and then (by definition of interpreter) to **$P1^{L1}$ (D).**

We have compiled (and specialized) $P1_{L1}$ from L1 to L0.

Second projection of Futamura

- $\mathbf{Peval}^{\mathbf{L0}}_{\mathbf{L0}}$ Partial evaluator for language **L0** Written in **L0**
- $\mathbf{Int}^{\mathbf{L0}}_{\mathbf{L1}}$ Interpreter of **L1** Written in **L0**
- $\mathbf{P}^{\mathbf{L1}}$ Program written in **L1**

- As it $\mathbf{Peval}^{\mathbf{L0}}_{\mathbf{L0}}$ is written in **L0** we can apply it to itself
- We then make partial evaluation of the partial evaluator, specializing it for the interpreter $\mathbf{Peval}^{\mathbf{L0}}_{\mathbf{L0}}\ (\mathbf{Peval}^{\mathbf{L0}}_{\mathbf{L0}}, \mathbf{Int}^{\mathbf{L0}}_{\mathbf{L1}})$
  What happens?

Second projection of Futamura

- $\mathbf{Peval^{L0}_{L0}}$ Partial evaluator for language **L0** Written in **L0**
- $\mathbf{Int^{L0}_{L1}}$ Interpreter of **L1** Written in **L0**
- $\mathbf{P^{L1}}$ Program written in **L1**

- $\mathbf{Peval^{L0}_{L0}\ (Peval^{L0}_{L0}\ , Int^{L0}_{L1}) = R^{\ L0}}$
- And you have that

  $$\mathbf{R^{L0}_{L0}\,(P^{L1}\ ) = Peval^{L0}_{L0}\,(Int^{L0}_{L1}\ P^{L1}\ )}$$
- We have produced a program $\mathbf{R^{L0}}$ written in L0, which if applied to a program $\mathbf{P^{L1}}$ (written in in **L1**) produces a program written in **L0** which is equivalent to the partial evaluation of the interpreter on $\mathbf{P^{L1}}$ and so (for the first projection) is equivalent to $\mathbf{P^{L1}}$

We got a **Compiler** from L1 to L0!

Third projection of Futamura

$Peval^{L0}_{L0}$ Partial evaluator for language **L0**

...

$$\mathbf{Peval^{L0}_{L0}\,(Peval^{L0}_{L0}\,,\,Peval^{L0}_{L0})}$$

What did we get? → Homework :)

Exercises

Exercises from the PL book 1.3, 1.5, 1.6, 1.7

Hint: for exercise 6 if you do not know where to start see slides below

Pascal compiler generation: bootstrapping

- The first Pascal environments included
  - A Pascal compiler from Pascal to P-code: $\mathcal{C}^{\textbf{Pascal}}_{\textbf{Pascal, P-Code}}$
  - The same compiler, translated into P-code: $\mathcal{C}^{\textbf{P-Code}}_{\textbf{Pascal, P-Code}}$
  - An interpreter for P-code, written in Pascal: $\mathcal{I}^{\textbf{Pascal}}_{\textbf{P-code}}$
- To have a local implementation on a MO specification:
  - Produce (by hand) a translation of $\mathcal{I}^{\textbf{Pascal}}_{\textbf{P-code}}$ in the language of M0:

$$\mathcal{I}^{\mathcal{L}0}_{\textbf{P-code}}$$

- Run a Pascal P program on Mo:

$$\mathcal{I}^{\mathcal{L}0}_{\text{P-code}}(\mathcal{C}^{\text{P-Code}}_{\text{Pascal, P-Code}}, \text{P}) = \text{P '}, \text{ in P-code}$$
$$\mathcal{I}^{\mathcal{L}0}_{\text{P-code}}(\text{P '}, x) = \text{desired result}$$

Can we do better?

Compiler generation: bootstrapping

- Improve efficiency, with a compiler written in $L0$
- By hand, starting from $\mathcal{C}^{\textbf{Pascal}}_{\textbf{Pascal, P-Code}}$ produce $\mathcal{C}^{\textbf{Pascal}}_{\textbf{Pascal,L0}}$
- Now we have:
  - $\mathcal{C}^{\text{Pascal}}_{\text{Pascal, P-Code}}$
  - $\mathcal{C}^{\text{P-Code}}_{\text{Pascal, P-Code}}$
  - $I^{\text{Pascal}}_{\text{P-code}}$
  - $I^{L0}_{\text{P-code}}$
  - $\mathcal{C}^{\text{Pascal}}_{\text{Pascal,L0}}$
- Bootstrapping

$$I^{L0}_{\text{P-code}}(\mathcal{C}^{\text{P-Code}}_{\text{Pascal, P-Code}}, \mathcal{C}^{\text{Pascal}}_{\text{Pascal,L0}}) = \mathcal{C}^{\text{P-Code}}_{\text{Pascal,Lo}}$$

$$I^{L0}_{\text{P-code}}(\mathcal{C}^{\text{P-Code}}_{\text{Pascal,Lo}}, \mathcal{C}^{\text{Pascal}}_{\text{Pascal,L0}}) = \mathcal{C}^{L0}_{\text{Pascal,L0}}$$