INF108: Compilation

Louis Jachiet

Interpretation

Examples

You all know examples of interpreted programs: python, js, ocaml (sort of), bash

Examples

You all know examples of interpreted programs: python, js, ocaml (sort of), bash

and examples of compiled programs: latex, C, ocaml

A mathematical definition

A compiler is a function

$c : \mathcal{L}_1 \rightarrow \mathcal{L}_2$ such that:

$$\forall P, \text{input},\ sem_1(P)(\text{input}) = sem_2(c(P))(\text{input})$$

A mathematical definition

A compiler is a function

$c : \mathcal{L}_1 \to \mathcal{L}_2$ such that:

$$\forall P, \text{input},\ \ sem_1(P)(\text{input}) = sem_2(c(P))(\text{input})$$

An interpreter is a function

$i : \mathcal{L}_1 \to$ execution such that:

$$\forall P, \text{input},\ \ sem_1(P, \text{input}) = i(P, \text{input})$$

A mathematical definition

A compiler is a function

$c : \mathcal{L}_1 \rightarrow \mathcal{L}_2$ such that:

$$\forall P, \text{input}, \ \mathit{sem}_1(P)(\text{input}) = \mathit{sem}_2(c(P))(\text{input})$$

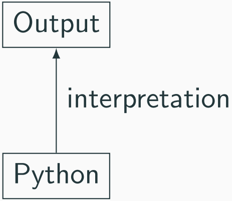
An interpreter is a function

$i : \mathcal{L}_1 \rightarrow$ execution such that:

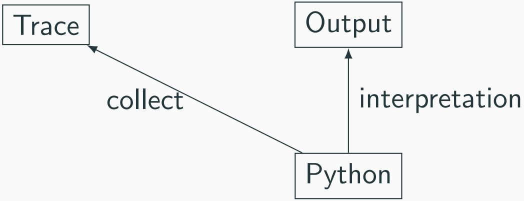
$$\forall P, \text{input}, \ \mathit{sem}_1(P, \text{input}) = i(P, \text{input})$$

Note: an interpreter runs for every input, a compiler runs once.

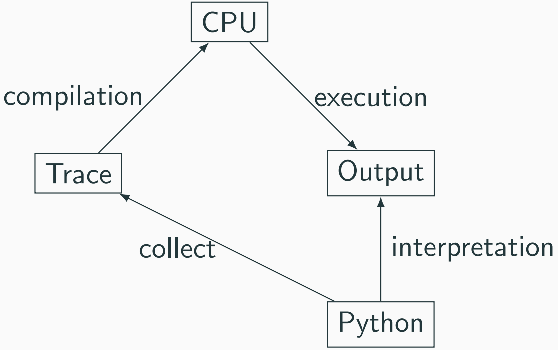
Actually...



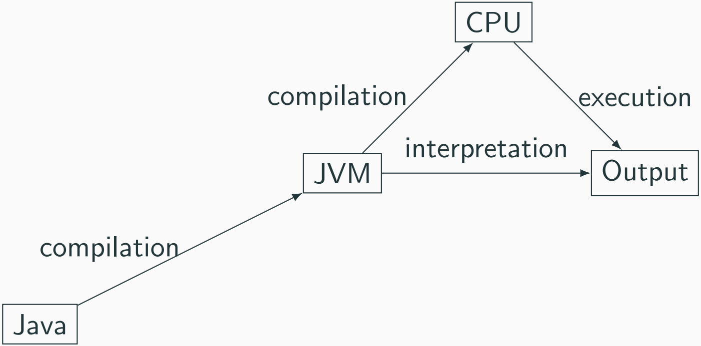
Actually...



Actually...


CPU
compilation
execution
Trace
Output
collect
interpretation
Python

Actually...


CPU
compilation
execution
interpretation
JVM
Output
compilation
Java

Benefits of interpreted vs compiled languages

Cons

- (somewhat) slower
- memory hungry
- no static verification (especially types)
- close to the machine

Pros

- more advanced features
- full abstraction of the execution
- automatic garbage collection
- reflection
- allows quick & dirty code

Type systems

What are types?

Types are used for a variety of reasons:

- to help understand what is happening

What are types?

Types are used for a variety of reasons:

- to help understand what is happening
- to ensure the safety of programs

What are types?

Types are used for a variety of reasons:

- to help understand what is happening
- to ensure the safety of programs
- to help the compiler select the appropriate structures

What are types?

Types are used for a variety of reasons:

- to help understand what is happening
- to ensure the safety of programs
- to help the compiler select the appropriate structures
- to help programmers with some design patterns

What are types?

Types are used for a variety of reasons:

- to help understand what is happening
- to ensure the safety of programs
- to help the compiler select the appropriate structures
- to help programmers with some design patterns

Different paradigms have very different interpretation of what types are...

Strong vs Weak Typing

Strong typing

Verifies a lot of things at compile time and thus “ensures” nothing wrong will happen at the execution.

Strong vs Weak Typing

Strong typing

Verifies a lot of things at compile time and thus “ensures” nothing wrong will happen at the execution.

Weak typing

Allows the use to bypass type safety, allows implicit conversion, etc.

Strong vs Weak Typing

Strong typing

Verifies a lot of things at compile time and thus “ensures” nothing wrong will happen at the execution.

Weak typing

Allows the use to bypass type safety, allows implicit conversion, etc.

Not a clear binary distinction! Most languages fall in between...

Dynamic vs Static Typing

Dynamic typing

All the type checking and the type information is managed at the execution.

Dynamic vs Static Typing

Dynamic typing

All the type checking and the type information is managed at the execution.

Static typing

All the type checking and the type information is managed at the compilation

Dynamic vs Static Typing

Dynamic typing

All the type checking and the type information is managed at the execution.

Static typing

All the type checking and the type information is managed at the compilation

Not a clear binary distinction! Most languages fall in between...

Manifest vs Inferred vs Latent Typing

Manifest typing

The user declares the types of the variables

Manifest vs Inferred vs Latent Typing

Manifest typing

The user declares the types of the variables

Inferred typing

The system deduces the types of variables

Manifest vs Inferred vs Latent Typing

Manifest typing

The user declares the types of the variables

Inferred typing

The system deduces the types of variables

Latent typing

Variables do not have types, values have types

Manifest vs Inferred vs Latent Typing

Manifest typing

The user declares the types of the variables

Inferred typing

The system deduces the types of variables

Latent typing

Variables do not have types, values have types

Not a clear binary distinction! Most languages fall in between...

Nominal vs Structural Typing

Nomimal typing

The user declares names for types two types are different if they have different names

Nominal vs Structural Typing

Nomimal typing

The user declares names for types two types are different if they have different names

Structural typing

Two values have the same types if they have the same "properties".

Nominal vs Structural Typing

Nomimal typing

The user declares names for types two types are different if they have different names

Structural typing

Two values have the same types if they have the same "properties".

Duck typing

An object can be used as long we only need fields or methods that the object has.

Nominal vs Structural Typing

Nomimal typing

The user declares names for types two types are different if they have different names

Structural typing

Two values have the same types if they have the same "properties".

Duck typing

An object can be used as long we only need fields or methods that the object has.

Not a clear binary distinction...

How to compute the types?

It depends A LOT on the type systems:

- In Python, everything needs to be checked at the execution

How to compute the types?

It depends A LOT on the type systems:

- In Python, everything needs to be checked at the execution
- In C, all variables are explicitly typed, we just need to apply implicit typing

How to compute the types?

It depends A LOT on the type systems:

- In Python, everything needs to be checked at the execution
- In C, all variables are explicitly typed, we just need to apply implicit typing
- In Ocaml, we need infer all the type information in the “most general” way

How to compute the types?

It depends A LOT on the type systems:

- In Python, everything needs to be checked at the execution
- In C, all variables are explicitly typed, we just need to apply implicit typing
- In Ocaml, we need infer all the type information in the "most general" way $\Rightarrow$ how to do that?

Typing a fragment of OCaml

Our types are inductively defined as:

- some basic types (int, char, string, etc.)
- functions types ($\tau \to \tau'$)
- product types ($\tau_1 \times \tau_2$)
- variable types (e.g. α)

All type variables are quantified globally!

Typing a fragment of OCaml

An algorithm to infer types:

- start with a type variables for all language variables and expressions
- then add constraints
  - if $x = \text{cst}$ then add $t(x) = t(\text{cst})$
  - if $(x, y) = z$ then add $(t(x) \times t(y)) = t(z))$
  - if $y = fx$ is used then add
    - $t(f) = \tau_1 \to \tau_2$
    - $t(y) = \tau_2$
    - $t(x) = \tau_1$
  - for let $x = e_1$ in e_2 then add $t(x) = t(e_1)$

Limits of this type system

What is the type of this ?

```
let create_store () =
    let data  = ref None in
    let get () = match !data with Some x -> x in
    let set x = data := Some x in
    (get,set)
```

Limits of this type system

What is the type of this ?

```
let create_store () =
   let data  = ref None in
   let get () = match !data with Some x -> x in
   let set x = data := Some x in
   (get,set)

let myStore = create_store ()
```

A correct type system?

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash_D x : \sigma} \quad [\texttt{Var}]$$

$$\frac{\Gamma \vdash_D e_0 : \tau \to \tau' \qquad \Gamma \vdash_D e_1 : \tau}{\Gamma \vdash_D e_0\ e_1 : \tau'} \quad [\texttt{App}]$$

$$\frac{\Gamma,\ x : \tau \vdash_D e : \tau'}{\Gamma \vdash_D \lambda\, x\ .\ e : \tau \to \tau'} \quad [\texttt{Abs}]$$

$$\frac{\Gamma \vdash_D e_0 : \sigma \qquad \Gamma,\ x : \sigma \vdash_D e_1 : \tau}{\Gamma \vdash_D \texttt{let}\ x = e_0\ \texttt{in}\ e_1 : \tau} \quad [\texttt{Let}]$$

$$\frac{\Gamma \vdash_D e : \sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash_D e : \sigma} \quad [\texttt{Inst}]$$

$$\frac{\Gamma \vdash_D e : \sigma \quad \alpha \notin \text{free}(\Gamma)}{\Gamma \vdash_D e : \forall\, \alpha\ .\ \sigma} \quad [\texttt{Gen}]$$

Can we have a strong and powerful type system?

Can we have a strong and powerful type system?

You will see in P2 :)