

# Static Type Analysis of Pattern Matching by Abstract Interpretation

Pietro Ferrara

ETH  
Zurich, Switzerland

FMOODS & FORTE, Amsterdam, Netherlands

# Pattern matching

- **Pattern matching**
  - > Well known and powerful feature
  - > Typical of functional programming languages
  - > Check if an expression matches a pattern
    - A list is empty or contains at least an element
    - The type of the variable is a String, Integer, ...
    - String like "a" followed by something else
    - Etc.
- **Our work is focused on type cases**

# New programming languages

- **Recent new programming languages**
  - > Scala (Java bytecode)
  - > F# (.Net)
- **Combine functional and OO features**
  - > Multi-paradigm programming languages
- **New open issues**
  - > Guarantee at compile time the exhaustiveness of
    - pattern matching
    - on type information

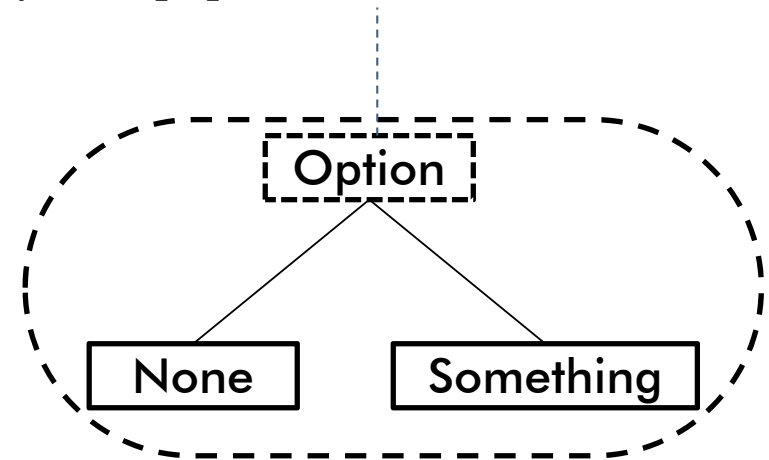
# Scala's approach

- Our approach is based on Scala
  - > But it is applicable to many other languages
- This language introduced new features
  - > sealed classes
    - Classes that cannot be extended by external code
- Pattern matching
  - > compiled to `if-then-else` statements
- Scala compiler
  - > Provide some feedback on pattern matching
    - Neither sound nor complete ☹️

Pietro Ferrara: "Static Type Analysis of Pattern Matching by Abstract Interpretation"  
FMOODS & FORTE, Amsterdam, Netherlands

# Running example

```
abstract sealed class Option[T]  
class Something[T](val y : T) extends Option[T]  
object None extends Option[Nothing]
```

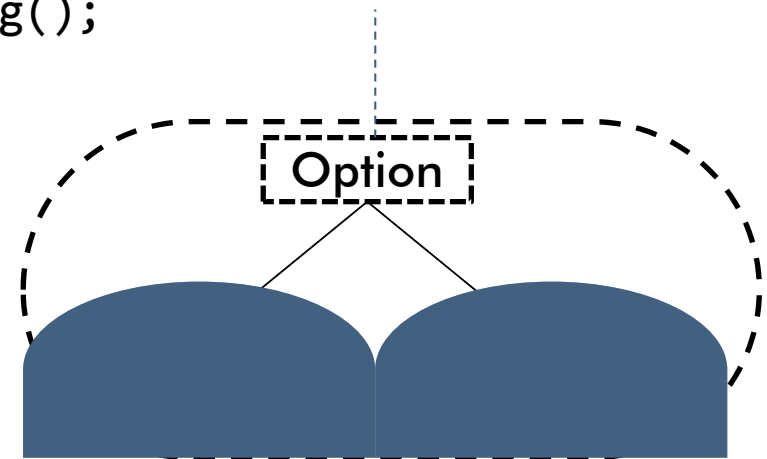


```
def extract[T](x : Option[T]) : String = x match {  
  case Something(y) => return y.toString();  
  case None => return "";  
}
```

# Running example

```
def extract[T](x : Option[T]) : String = x match {  
  case Something(y) => return y.toString();  
  case None => return "";  
}
```

Scala compiler



```
String extract(Option x) {  
→ if(x instanceof Something)  
  return ((Something) x).y.toString(); ✓ safe cast  
  else if(x instanceof None)  
    return "";  
    else throw new MatchError(); ✓ unreachable exception  
}
```

# Properties of interest

- We are interested in two properties
- Unreachability of MatchError exceptions
  - > Unreachable MatchError  $\Rightarrow$  Exhaustive
- Safety of casts
  - > Common pattern:
    - Check the type of a variable before casting it
- We need to track type information
  - > MatchError unreachable  $\Leftrightarrow$  Explored all types

# Sound static analysis

- **Prove these properties**
  - > at compile time (static)
  - > respected by all the executions (sound)
- **Abstract interpretation**
  - > Cousot&Cousot 77/79
  - > Mathematical framework to
    - Define the semantics
    - Soundly approximate it
  - > Ideal goal: fast and precise abstraction

# Contribution

- **New sound static type analysis**
- **Based on abstract interpretation**
  - > Two abstract domains
  - > Reduced product
- **Intra-procedural**
  - > Rely on pre and post conditions
- **Implemented**
  - > Applied to all Scala library 2.7.7

# Outline

## 1. Introduction

## 2. Abstract Semantics

- > Dynamic typing
- > Not-instance-of domain

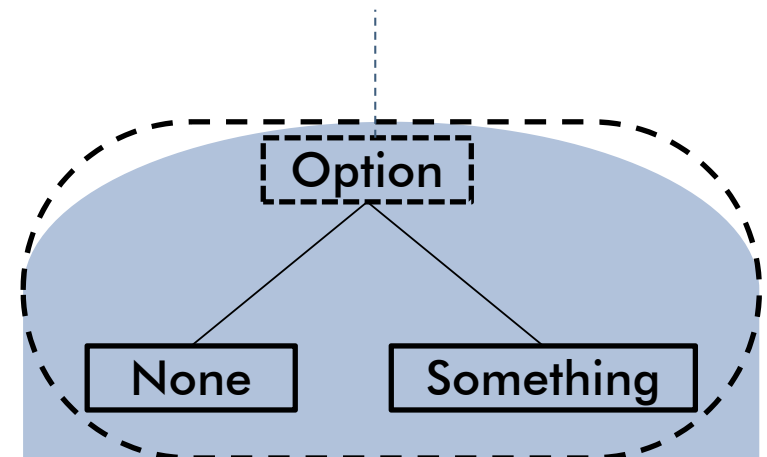
## 3. Experimental results

## 4. Conclusion and future work

# Static typing

- Variables and locations have static types
  - > Defined in the declaration of a variable/field
- Preserved by method calls
- Syntactic sugar

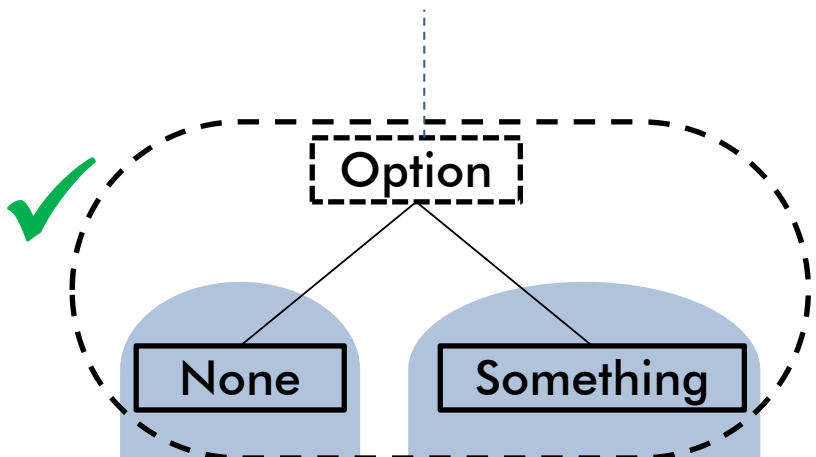
```
String extract(Option x) {  
  if(x instanceof Something)  
    return ((Something) x).y.toString();  
  else if(x instanceof None)  
    return "";  
  else throw new MatchError();  
}
```



# Dynamic typing

- Overapproximate the dynamic type
- Track information through
  - > Assignments
  - > Evaluation of instanceof conditions

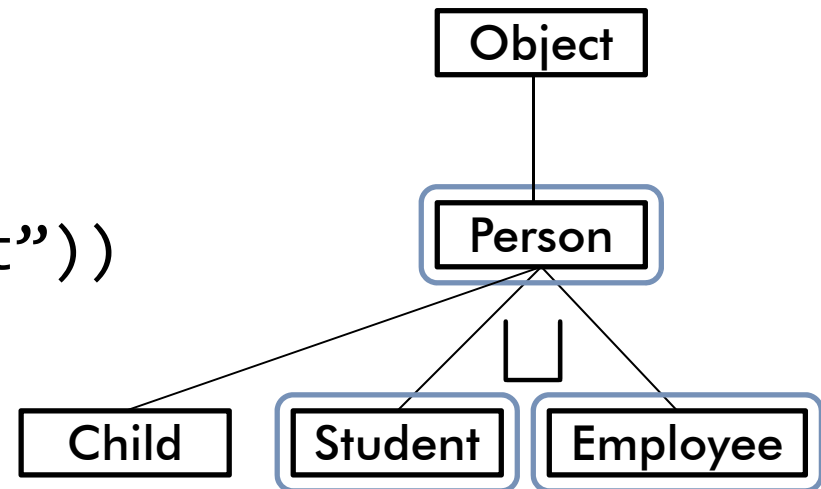
```
String extract(Option x) {  
→ if(x instanceof Something)  
    return ((Something) x).y.toString();  
else if(x instanceof None)  
    return "";  
else throw new MatchError();  
}
```



# Lattice structure

- Least upper bound
  - > Least common supertype

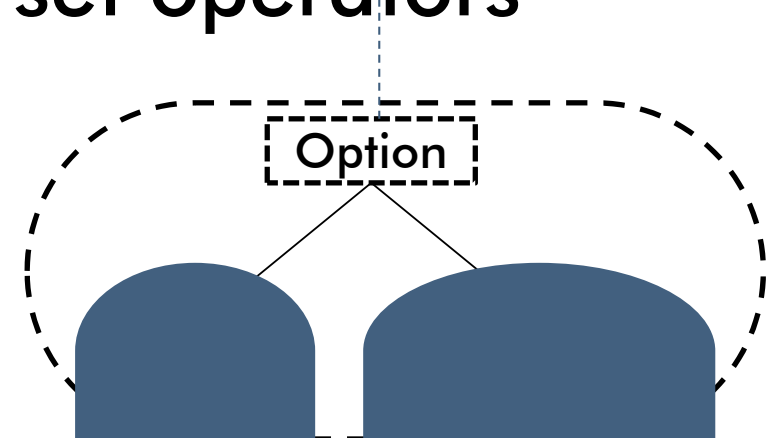
```
Object p;  
if(args[0].equals("Student"))  
    p=new Student();  
else p=new Employee();
```



# Not-instance-of domain

- Types a variable cannot be instance of
- Track information through
  - > Assignments
  - > Evaluation of `!instanceof` conditions
- Lattice structure: inverse set operators

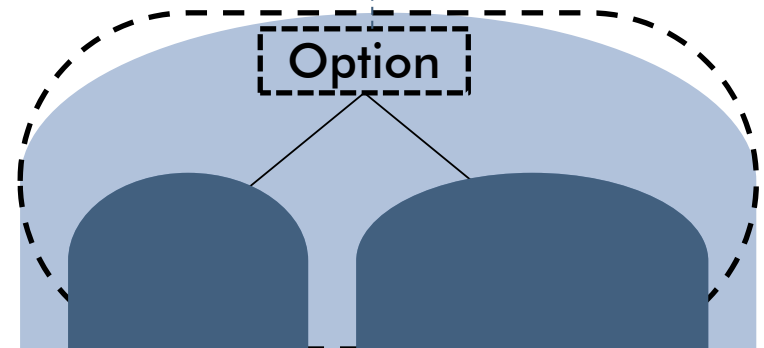
```
String extract(Option x) {  
→ if(x instanceof Something)  
    return ((Something) x).y.toString();  
else if(x instanceof None)  
    return "";  
else throw new MatchError();  
}
```




- Option is abstract
  - It cannot be instantiated
- Option is sealed
  - It cannot be extended by external code
- x cannot be instance of
  - None
  - Something

throw new MatchError() is unreachable

```
String extract(Option x) {  
→ if(x instanceof Something)  
    return ((Something) x).y.toString();  
else if(x instanceof None)  
    return "";  
else throw new MatchError();  
}
```



# Formal definition

$$\overline{red}_{\overline{FT}}((\overline{d}, \overline{n})) = \perp_{\overline{FT}}$$


$$\overline{red}_{\overline{FT}}((\overline{d}, \overline{n})) = \perp_{\overline{FT}}$$



throw new MatchError() is unreachable

```
else if (x instanceof None)
    return "";
else throw new MatchError();
```

None

Something

# Theoretical results

- **Definition of abstract and concrete**

- > **Domain**

- > **Semantics**

$$S[x = E, d] = d[x \mapsto \mathbb{E}[E, d]]$$

$$S[\text{declare } x : T, d] = d[x \mapsto T]$$

$$S[\text{assume}(B), d] = \begin{cases} \perp & \text{if } \mathbb{B}[B, d] = \text{false} \\ d & \text{if } \mathbb{B}[B, d] = \text{true} \end{cases}$$

$$\overline{NT}[x=y, \overline{nt}] = \overline{nt}[x \mapsto \overline{nt}(y)]$$

$$\overline{NT}[x=\text{new } T, \overline{nt}] = \overline{nt}[x \mapsto \emptyset]$$

$$\overline{NT}[\text{assume}(! x \text{ instanceof } T, \overline{nt})] = \overline{nt}[x \mapsto \overline{nt}(x) \cup \{T\}]$$

# Theoretical results

- **We prove formally the soundness**

Let  $S^\infty$  be the trace semantics defined relying on  $S$ , and  $\overline{FT}^\infty$  be the abstract trace semantics based on  $\overline{FT}$ . Then  $lfp \sqsubseteq S^\infty \subseteq \gamma(lfp \sqsubseteq_{\overline{FT}} \overline{FT}^\infty)$

- **The proof is based on**
  - > **Lattices as concrete and abstract domains**
  - > **Local soundness of the abstract semantics**
  - > **Monotonicity of the concretization function**

# Outline

## 1. Introduction

## 2. Abstract Semantics

- > Dynamic typing
- > Not-instance-of domain

## 3. Experimental results

## 4. Conclusion and future work

# Implementation

- **The analysis has been implemented**
  - > **Sample**
    - **Static Analyzer of Multiple Programming Languages**
    - **Generic static analyzer**
    - **Pluggable with different analysis**
  - > **Adopted a very simple heap abstraction**
- **Analyzed all the Scala library 2.7.7**
  - > **On a Intel Core 2 Quad CPU 2.83 GhZ**
  - > **4GB RAM, Windows 7**

# Times of execution

Lib.			Casts			MatchErrors		
	#m	†	✓	✗	%	✓	✗	%
actors	1626	3'56"	47	104	31%	20	17	54%
collection	14758	10'21"	183	509	26%	42	74	36%
util	4926	8'21"	126	508	11%	36	65	36%
xml	2786	6'21"	108	286	27%	5	21	19%
main	8218	12'02"	97	196	33%	37	12	76%
Scala lib	35732	54'02"	725	1951	27%	156	208	43%

- About 90 msec per method
- Proved the exhaustiveness of almost half of the pattern matchings

# Discussion (1/2)

- **Prove the exhaustiveness of 43% pattern matchings**
- **Not so precise**
  - > 57% (208) false alarms
- **Pattern matching deals not only with types**
  - > Structure of the data (e.g., list)
  - > Numerical information (e.g., constants)
  - > ... and so on!

# Discussion (2/2)

- Prove the safety of 27% of casts
  - > Not so precise
    - 73% (1951) false alarms
  - > No information about generic types
    - Like in Java bytecode
    - Dramatic when dealing with data structures

```
List<Integer> list=....;  
Integer a=list.apply(0);
```

Scala compiler

```
List list=....;  
Integer a=(Integer) list.apply(0);
```

- We cannot prove the safety of these casts!

# Outline

## 1. Introduction

## 2. Abstract Semantics

- > Dynamic typing
- > Not-instance-of domain

## 3. Experimental results

## 4. Conclusion and future work

# Conclusion

- **New sound static analysis**
- **Focused on**
  - > **Pattern matching**
  - > **Type cases**
  - > **Exhaustiveness**
  - > **Safety of type casts**
- **Implemented**
  - > **Scalable**
  - > **“Precise” (in some way 😊)**

# Future work

- **Combine the type analysis with others**
  - > Heap analysis
  - > Numerical analysis
- **Write contracts**
  - > Which level of expressiveness?
  - > How many contracts do we need?
  - > Can we automatically infer them?
- **Recover information about generic types**
  - > Improve the analysis of casts

# Question time

# Thank you!

Pietro Ferrara: "Static Type Analysis of Pattern Matching by Abstract Interpretation"  
FMOODS & FORTE, Amsterdam, Netherlands