Static Type Analysis of Pattern Matching by Abstract Interpretation

Pietro Ferrara

ETH Zurich, Switzerland
pietro.ferrara@inf.ethz.ch

Abstract. Pattern matching is one of the most attractive features of functional programming languages. Recently, pattern matching has been applied to programming languages supporting the main current object oriented features. In this paper, we present a static type analysis based on the abstract interpretation framework aimed at proving the exhaustiveness of pattern matchings and the safety of type casts. The analysis is composed by two distinct abstract domains. The first domain collects information about dynamic typing, while the second one tracks the types that an object cannot be instance of. The analysis has been implemented and applied to all the Scala library. The experimental results underline that our approach scales up since it analyzes a method in 90 msec in average. In addition, the analysis is precise in practice as well, since we prove the exhaustiveness of 42% of pattern matchings and 27% of the type casts without any manual annotation on the code.

Key words: Abstract interpretation, static analysis, pattern matching.

1 Introduction

Pattern matching is recognized to be one of the most expressive features of functional programming languages. Extending its full expressiveness to programming languages supporting the current object oriented features is not straightforward [12]. In fact, features like inheritance and information hiding are not supported by the functional pattern matching, since it dealt with algebraic data types. Nevertheless, recent work [14, 26] introduced pattern matching in programming languages supporting the main current object oriented features as F# [1] and Scala [23]. In addition, some extensions of Java supporting pattern matching appeared in the last few years [16, 24]. Pattern matching checks if an expression respects a given pattern looking at an ordered list of case expressions. One of the most common case expressions is the type case: we select the case to apply following the dynamic type of the expression. Consider the Scala program in Figure 1. The content of argument x is matched with respect to its type: if it is an instance of Something, the method returns the string representing the contained value. Otherwise, the type of x is None, and extract returns the empty string.

Languages like F# and Scala are compiled into a bytecode language (for instance, MSIL [11] or Java bytecode [18]) that supports the main imperative features. Thus type cases in pattern matching are translated into type tests and casts. For instance, the body of method extract in Figure 1 is translated into the code in Figure 2 by the Scala compiler [13]. Note that information about generics is erased. This code can be easily optimized adopting a specific static

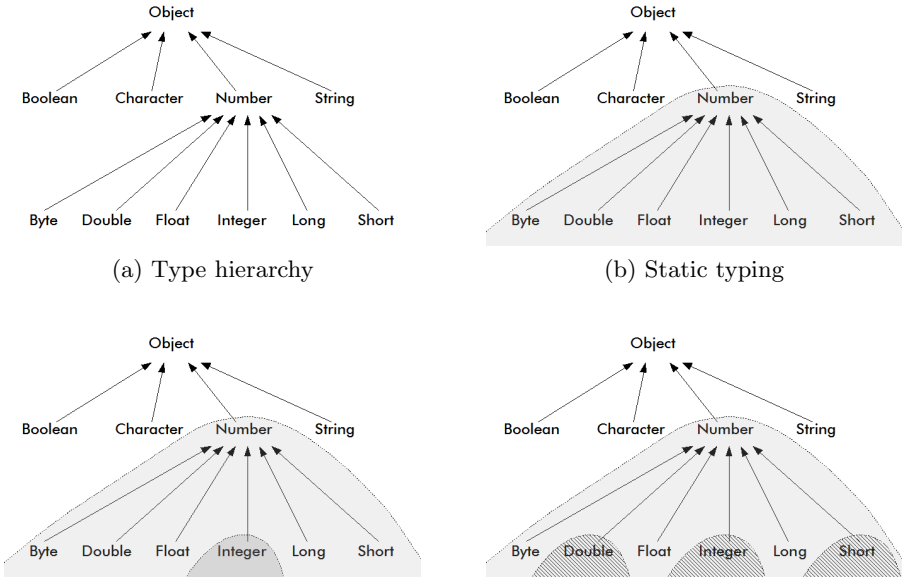
```
abstract sealed class Option[T]
 final case class Something[T](val y : T) extends Option[T]
 final case object None extends Option[Nothing]

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

Fig. 1: Pattern matching with type cases

```
1 String extract (Option x) {
2         if (x instanceOf Something)
3                 return ((Something) x).y. toString ();
4         else if (x instanceOf None)
5                         return "";
6                 else throw new MatchError ();
7 }
```

Fig. 2: Results of Scala compilation

analysis. When x instanceof Something is false, we know that x instanceof None is always true. In fact, the static type of x is Option, this class is abstract, and the only two classes that extend it are Something and None. Note that Option is declared as sealed. In Scala, a sealed class can be extended only by classes that are defined in the same source file. Therefore we know that Option cannot be extended by external code, and we can conclude that the if statement at line 4 is always evaluated to true. In addition, we know that the statement throw new MatchError() is unreachable, that is the pattern matching contained by the original Scala program is exhaustive. Finally, the type cast at line 3 is safe.



(a) Type hierarchy

(b) Static typing

(c) Dynamic typing $\overline{\mathsf{DT}}$

(d) Not-instance-of domain $\overline{\mathsf{NT}}$

Fig. 3: Type system and abstract domains

1.1 Contribution

The main contribution of this paper is the introduction of two new abstract domains in order to capture precise information on the types of variables. The two main goals of our analysis are precision and efficiency. Precision is achieved through the development of abstract domains focused on the properties of interest, that is, exhaustiveness of pattern matching and safety of casts. Efficiency is achieved through a modular analysis. For this reason, our approach is based on the Design-by-Contract (DbC) methodology [21]. This means that when we analyze a method call we rely on class invariants, pre and post conditions. Thanks to this approach, we do not need to analyze the whole program, but we can analyze a method alone relying on its contracts. In a similar way, we rely on language constructs like sealed in Scala in order to be sure that a class cannot be extended by external code.

In this paper, we do not deal with issues related to the heap and its analysis. Hence the language on which we are going to define our analysis does not contain field accesses, aliasing, etc., but it deals only with variables. The object oriented programs we analyzed are preprocessed by a sound heap analysis that replaces heap accesses with variables. This means that method calls may have side effects on these variables. The heap analysis we perform is quite rough: intuitively, we approximate all the instances of the same class with the same variable, and we perform weak updates [4] as these abstract references may represent several

concrete references. Then the number of abstract references is bounded to the number of classes of the analyzed program. In this way we preserve the scalability of our approach. On the other hand, it may seem that this approach is too much coarse since we approximate all the fields having the same static type with the same variable. Indeed, this does not particularly affect the precision of our approach because, for the most part of the programs we analyzed, the content of fields is first stored in a local variable (and in this way we obtain a specific variable identifier), and then type checking and casts are performed on this local variable.

Consider now the type hierarchy depicted in Figure 3a. The class Object is extended by several subclasses that represent different types in a programming language, e.g., Boolean. The abstract class Number is extended by several subclasses, each representing one possible numerical type, e.g., Integer. For instance, Figure 3b represents a situation in which the static type of a variable is Number. The static type information guarantees that the objects assigned to a variable are instances of its static type or of one of its subclasses. Thus instances of Long, Float, etc. may be assigned to a variable whose static type is Number.

The first abstract domain we are going to define is $\overline{\mathsf{DT}}$. It is aimed at approximating information about the dynamic type of variables. Figure 3c depicts a case in which we assign an object instance of the Integer class (darker area) to a variable of static type Number (light grey area). The information about dynamic typing comes mainly from assignments, type casts, new statement, and assumption of instanceof conditions. Since we rely on DbC, if a method has no postcondition about the dynamic type of variables (e.g., the returned value), we would abstract its type information using its static type after the method call.

The second domain we are going to define is $\overline{\mathsf{NT}}$. It abstracts the types that a variable cannot be instance of. For instance, Figure 3d depicts a situation in which a variable of static type Number cannot be instance of Double, Integer, or Short. In this way we can discard some leaves of the type tree. This information is collected when we assume the negation of an instanceof boolean condition. We could have adopted a different approach capturing a set of types for each variable representing all the possible types a variable can have at a given program point. In this way, we may model that an object cannot be instance of a type simply removing the type from the set of types related to it. Unluckily this solution is not feasible for two main reasons. Imagine to be after a method call, where we know only the static type of a variable. In this context, assuming that an object cannot be instance of a given type (i) may affect the performance (e.g., if its static type is Object and we have to exclude String we should relate the variable to a set of types of huge dimensions), (ii) cannot be represented since our analysis is modular and external code usually defines and instantiates new classes. One may argue that, since we are interested in proving the exhaustiveness of pattern matching, we can restrict the analysis only on types that cannot be extended by external code. With such restriction the approach based on set of possible types would be feasible, and performance could be preserved supposing that we have few subclasses. First of all, we want to track information also on classes

that can be externally extended, in order to eventually provide information to developers like "if this class would be sealed, then the pattern matching would be exhaustive". This solution would not collect the information necessary to provide this output. In addition, we do not expect in general that sealed classes (or more generally classes that cannot be extended by external code) have only few subclasses, since each subclass may be used to represent a particular case of a complex data structure.

Combining $\overline{\mathsf{DT}}$ and $\overline{\mathsf{NT}}$, we are in position to obtain precise information about types and prove the safety of some cast operations, and the unreachability of some code, in particular when dealing with pattern matching.

The analysis works on a simple object oriented language. We designed it in order to translate the main object oriented programming languages into this language. In this way, we want to apply our analysis to several languages. Up to now, we have developed the translation of the Scala programming language. Thus our analysis is focused on the code obtained from the Scala pattern matching statements. In this context, if we prove that a MatchError exception is unreachable, this means that the pattern matching is exhaustive.

The analysis has been implemented and the experimental results underline that our approach scales up. We are in position to analyze Scala libraries (more than 30.000 methods) in less than one hour, and in average we require 90 milliseconds to analyze a single method. In addition, it turns out that the analysis is precise in practice. In fact, we are able to prove the exhaustiveness of 42% of pattern matchings (that may contain not only type cases but also other cases whose information is not captured by our analysis) and 27% of the type casts without any manual annotation of the code.

The following of this section introduces some preliminaries on abstract interpretation. Section 2 presents the language that we analyze. Section 3 introduces the concrete domain and semantics, while Section 4 formalizes the two domains adopted by our analysis, and explains how they work on an example. Section 5 presents the experimental results, and Section 6 introduces the related work. Finally, Section 7 concludes and discusses the future work.

1.2 Abstract Interpretation

Our approach is based on abstract interpretation [7, 8], a theory for defining and soundly approximating the semantics of a program. A concrete semantics, aimed at specifying the runtime properties of interest, is defined. Then it is approximated to obtain an abstract semantics computable but still precise enough to capture the property of interest, and specified by an abstract domain and an abstract transition function.

Formally, the concrete domain D is a lattice $\langle \mathsf{D}, \sqsubseteq_{\mathsf{C}}, \bot_{\mathsf{C}}, \top_{\mathsf{C}}, \sqcup_{\mathsf{C}}, \sqcap_{\mathsf{C}} \rangle$. The concrete elements are related to the abstract domain $\langle \overline{\mathsf{D}}, \sqsubseteq_{\mathsf{A}}, \bot_{\mathsf{A}}, \top_{\mathsf{A}}, \sqcup_{\mathsf{A}}, \sqcap_{\mathsf{A}} \rangle$ by a concretization function γ and an abstraction function α. In order to obtain a sound analysis, we require that they form a Galois connection; we denote it by $\langle \mathsf{D}, \sqsubseteq_{\mathsf{C}}, \bot_{\mathsf{C}}, \top_{\mathsf{C}}, \sqcup_{\mathsf{C}}, \sqcap_{\mathsf{C}} \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \overline{\mathsf{D}}, \sqsubseteq_{\mathsf{A}}, \bot_{\mathsf{A}}, \top_{\mathsf{A}}, \sqcup_{\mathsf{A}}, \sqcap_{\mathsf{A}} \rangle$.

C ::=	x = E	(C1)	E ::= new T()	(E1)	B ::= x instanceof T (B1)
	declare x : T	(C2)	x	(E2)	! x instanceof T (B2)
	assume(B)	(C3)	(T) x	(E3)	
	x.M(y1, ..., yn)	(C4)			

Table 1: OO core language

A semantics is defined on the abstract and concrete domains. Given a state of the domain and a statement, it produces the result of the execution of the statement starting from the given state. The abstract semantics $\overline{\mathbb{S}}$ has to soundly approximate the concrete one, that is $\forall \overline{\mathsf{d}} \in \overline{\mathsf{D}} : \mathbb{S}[\![\gamma(\overline{\mathsf{d}})]\!] \sqsubseteq_{\mathsf{A}} \gamma \circ \overline{\mathbb{S}}[\![\overline{\mathsf{d}}]\!]$. Usually, the semantics is defined as the computation of a fixpoint [8].

Different domains can be combined in Cartesian products. Let $\overline{\mathsf{A}} \times \overline{\mathsf{B}}$ be the Cartesian product of abstract domains $\overline{\mathsf{A}}$ and $\overline{\mathsf{B}}$. The lattice operators are defined as the pointwise application on the components of the pairs of the lattice operator on each abstract domain. The reduced product $\overline{\mathsf{A}} \otimes \overline{\mathsf{B}}$ is a Cartesian product on which a *reduce* function is provided: given a state of the Cartesian product, it mutually refines the information contained in each domain using the information provided by the other domain.

Generic analyzers Abstract interpretation can be applied in order to develop generic analyzers [5]. In particular, this theory allows one to define a compositional analysis, e.g., an analysis that can be instantiated with different numerical domains, and in order to analyze different properties. Many different generic analyzers have been proposed recently [15, 19, 25]. The type analysis we present in this paper has been implemented in Sample (Static Analyzer of Multiple Programming LanguagEs). This generic analyzer can be plugged with different numerical domains and heap analysis. In addition, it can be extended with new analyses like our type analysis.

2 Language

We introduce a core language in order to make explicit the main characteristics of our abstract domains. It consists of assignments (C1), declarations of variables (C2), assumptions of boolean conditions (C3), and method calls (C4). An expression that can be assigned to a variable is the instantiation of a class (E1), another variable (E2), or the cast of a variable (E3). A condition can be the check that the dynamic type is instance of (B1) or it is not instance of (B2) a type.

We focus our attention only on the main aspects of the type analysis we are interested in, that is, casts and runtime checks of dynamic types. Nevertheless, our implementation covers all the features of current programming languages, as numerical operations, field accesses, etc. Intuitively, we represent the body of a

$$\mathbb{E}[\![\mathsf{new\ T}(), \mathsf{d}]\!] = \mathsf{T}$$
$$\mathbb{E}[\![\mathsf{x}, \mathsf{d}]\!] = \mathsf{d}(\mathsf{x})$$
$$\mathbb{E}[\![(\mathsf{T})\ \mathsf{x}, \mathsf{d}]\!] = = \begin{cases} \mathsf{d}(\mathsf{x}) & \text{if } \mathsf{d}(\mathsf{x}) \sqsubseteq_{\overline{\mathsf{T}}} \mathsf{T} \\ \bot & \text{otherwise} \end{cases}$$

$$\mathbb{B}[\![\mathsf{x\ instanceof\ T}, \mathsf{d}]\!] = = \mathsf{d}(\mathsf{x}) \sqsubseteq_{\overline{\mathsf{T}}} \mathsf{T}$$
$$\mathbb{B}[\![!\mathsf{B}, \mathsf{d}]\!] = \neg\mathbb{B}[\![\mathsf{B}, \mathsf{d}]\!]$$

$$\mathbb{S}[\![\mathsf{x} = \mathsf{E}, \mathsf{d}]\!] = \mathsf{d}[\mathsf{x} \mapsto \mathbb{E}[\![\mathsf{E}, \mathsf{d}]\!]]$$
$$\mathbb{S}[\![\mathsf{declare\ x\ :\ T}, \mathsf{d}]\!] = \mathsf{d}[\mathsf{x} \mapsto \mathsf{T}]$$
$$\mathbb{S}[\![\mathsf{assume(B)}, \mathsf{d}]\!] = = \begin{cases} \bot & \text{if } \mathbb{B}[\![\mathsf{B}, \mathsf{d}]\!] = \mathsf{false} \\ \mathsf{d} & \text{if } \mathbb{B}[\![\mathsf{B}, \mathsf{d}]\!] = \mathsf{true} \end{cases}$$
$$\mathbb{S}[\![\mathsf{x.M(y1, ..., yn)}, \mathsf{d}]\!] = = \mathit{execute}(\mathsf{x.M(y1, ..., yn)}, \mathsf{d})$$

Table 2: Collecting semantics

method through a Control Flow Graph (CFG). Each block in the CFG contains a list of statements. Different blocks are connected through edges that could eventually contain a boolean condition to represent conditional jumps. In this way, we support control structures like if statements and while loops.

3 Concrete Domain and Collecting Semantics

The concrete domain is composed by environments that collect the dynamic type for each variable. Formally, $\mathsf{Env} : [\mathsf{Var} \rightarrow \overline{\mathsf{T}}]$. The concrete domain is composed by sets of environments. Each environment represents one possible execution of the program in a given point. The lattice is obtained using set operators. Formally, $\langle \wp(\mathsf{Env}), \subseteq, \cup, \cap, \mathsf{Env}, \emptyset \rangle$.

We define the collecting semantics as a function that, given an environment and a statement, returns the environment resulting from the execution of the given statement on the given environment. Table 2 reports the formal definition of this collecting semantics[1]. $\sqsubseteq_{\overline{\mathsf{T}}}$ corresponds to the subtype relation, while $\bot$ is used to represent a situation in which the execution is stopped because of an unsafe dynamic cast. The collecting semantics is aimed at formalizing the runtime behaviors focusing on the information we are interested. The extension of this semantics to our concrete domain (that is made by set of environments) is the application of this semantics on each environment in the initial state.

$\mathbb{S}$ is the basic step adopted in order to compute the semantics of a method on its CFG representation. Intuitively, we build up incrementally the traces representing the executions of a method relying on $\mathbb{S}$ to perform single computational steps [6]. Since non-deterministic behaviors are possible (e.g., because of inputs from the user) and these may also affect the type information, we consider sets of traces.

4 Abstract Domain and Semantics

In this section, we present and formalize the two abstract domains of our analysis, how we combine them, and we explain how they work in practice on the example

[1] $\mathit{execute}(\mathsf{x.M(y1, ..., yn)}, \mathsf{d})$ resolves and executes method M on d

presented in Figure 2. Note that we will define the abstract semantics only on single statements. Its extension to blocks and CFGs is obtained as usual in approaches based on abstract interpretation and trace semantics relying on the upper bound operators [7].

4.1 Static Typing

The programming language introduced in Section 2 provides some information on the static types of variables. We suppose that a subtype relation $\sqsubseteq_{\overline{\mathsf{T}}}$ is provided, and that this is a partial ordering. Given two types $\bar{\mathsf{t}}_1, \bar{\mathsf{t}}_2 \in \overline{\mathsf{T}}$, $\bar{\mathsf{t}}_1 \sqsubseteq_{\overline{\mathsf{T}}} \bar{\mathsf{t}}_2$ if and only if $\bar{\mathsf{t}}_1$ is subtype of $\bar{\mathsf{t}}_2$. We denote by $\sqcup_{\overline{\mathsf{T}}}$ and $\sqcap_{\overline{\mathsf{T}}}$ respectively the upper and lower bound operators univocally identified by $\sqsubseteq_{\overline{\mathsf{T}}}$. These operators form a lattice. In this way, we support the most part of common object oriented type systems [3], like the ones of Java, C#, Scala, and F#.

Each type represents itself and all the types that are its subtype. Note that this is exactly the semantics of a static type: at runtime a variable of static type $\bar{\mathsf{t}}$ can have a type that is $\bar{\mathsf{t}}$ or one of its subtypes. Formally, the concretization of a type $\bar{\mathsf{t}}$ is defined as $\gamma_{\overline{\mathsf{T}}}(\bar{\mathsf{t}}) = \{\overline{\mathsf{t}'} : \overline{\mathsf{t}'} \sqsubseteq_{\overline{\mathsf{T}}} \bar{\mathsf{t}}\}$. Since information about static typing can be considered as syntactic sugar, we suppose that a function *statictype* : $[\mathsf{Var} \to \overline{\mathsf{T}}]$ is provided. Given a variable, it returns its static type.

4.2 $\overline{\mathsf{DT}}$: Dynamic Typing

$\overline{\mathsf{DT}}$ abstracts the dynamic type of variables. Relying on $\overline{\mathsf{T}}$ we build up the domain $\overline{\mathsf{DT}} : [\mathsf{Var} \to \overline{\mathsf{T}}]$ that relates each variable to its dynamic type. The operators on the lattice $\langle \overline{\mathsf{DT}}, \sqsubseteq_{\overline{\mathsf{DT}}}, \sqcup_{\overline{\mathsf{DT}}}, \sqcap_{\overline{\mathsf{DT}}}, \top_{\overline{\mathsf{DT}}}, \bot_{\overline{\mathsf{DT}}} \rangle$ are obtained as the functional extension of the ones of $\overline{\mathsf{T}}$. The information inferred by $\overline{\mathsf{DT}}$ is refined with the static type information provided by the *statictype* function: if the type of a variable v is not yet defined, the dynamic type domain returns its static type, that is, *statictype*(v). The concretization of $\overline{\mathsf{DT}}$ is defined as the functional extension of $\gamma_{\overline{\mathsf{T}}}$ as well. Formally, $\gamma_{\overline{\mathsf{DT}}}(\bar{\mathsf{f}}) = \{[\mathsf{v} \mapsto \bar{\mathsf{t}}] : \mathsf{v} \in dom(\bar{\mathsf{f}}) \wedge \bar{\mathsf{t}} \in \gamma_{\overline{\mathsf{T}}}(\bar{\mathsf{f}}(\mathsf{v}))\}$.

Semantics

$$\overline{\mathbb{DT}}[\![\mathsf{x} = \mathsf{y}, \overline{\mathsf{dt}}]\!] = \overline{\mathsf{dt}}[\mathsf{x} \mapsto \overline{\mathsf{dt}}(\mathsf{y})]$$
$$\overline{\mathbb{DT}}[\![\mathsf{x} = \mathsf{new}\ \mathsf{T}(), \overline{\mathsf{dt}}]\!] = \overline{\mathsf{dt}}[\mathsf{x} \mapsto \mathsf{T}]$$
$$\overline{\mathbb{DT}}[\![\mathsf{x} = (\mathsf{T})\mathsf{y}, \overline{\mathsf{dt}}]\!] = \overline{\mathsf{dt}}[\mathsf{x} \mapsto \mathsf{T} \sqcap_{\overline{\mathsf{T}}} \overline{\mathsf{dt}}(\mathsf{y}), \mathsf{y} \mapsto \mathsf{T} \sqcap_{\overline{\mathsf{T}}} \overline{\mathsf{dt}}(\mathsf{y})]$$
$$\overline{\mathbb{DT}}[\![\mathsf{assume}(\mathsf{x}\ \mathsf{instanceof}\ \mathsf{T}), \overline{\mathsf{dt}}]\!] = \overline{\mathsf{dt}}[\mathsf{x} \mapsto \overline{\mathsf{dt}}(\mathsf{x}) \sqcap_{\overline{\mathsf{T}}} \mathsf{T}]$$
$$\overline{\mathbb{DT}}[\![\mathsf{x.M}(\mathsf{y1}, ..., \mathsf{yn}), \overline{\mathsf{dt}}]\!] = \overline{\mathbb{DT}}[\![\mathsf{assume}(\mathsf{rename}(\mathsf{postcondition}_{\mathsf{M}})), \top_{\overline{\mathsf{DT}}}]\!]$$

When a variable is assigned to another one ($\mathsf{x} = \mathsf{y}$), we capture that the dynamic type of x is the type of y. Similarly, when we assign to a variable the instance of a fresh object, we relate this variable to the type of the new object. The dynamic type of the assignment of a cast to type T of a variable y is the lower bound between the dynamic type of y and type T. Thus, it may be the case in

which it contains a type that is more approximated than T even if the cast is safe. After the cast, we know that its dynamic type will be surely T or one of its subtypes. This is why we take the lower bound between the dynamic type of the variable and T. In a similar way, when we test to true a boolean condition like x instanceof T, we track that the dynamic type of x is the lower bound of its type and T. When we analyze a method call, we assume the postconditions of this method call, that may contain some type information, on the top state of the $\overline{\mathsf{DT}}$ domain. Since we do not consider framing information, we suppose that potentially a method may access and assign all the variables of the current state of computation. Then we assume the postcondition on a top state in order to preserve the soundness of our analysis. In addition, we suppose that a rename function is provided, and it renames all the variables contained in the postcondition matching the calling environment.

Running Example At the beginning of the analysis of method extract we have no information about dynamic typing, but we know that the static type of x is Option. When we analyze the condition of the if branch at line 2 we calculate that the state leading to the then statement of our dynamic type domain is $[\mathsf{x} \mapsto \mathsf{Something}]$, since the condition is evaluated to true. In the same way, at line 4 we obtain that $[\mathsf{x} \mapsto \mathsf{None}]$. Thus we prove that the cast at line 3 is safe, while we are not able yet to detect that line 6 is unreachable.

4.3 $\overline{\mathsf{NT}}$: Not-instance-of Domain

The second domain we introduce is $\overline{\mathsf{NT}}$. This domain is aimed at collecting the types an object cannot be instance of at a given point of the program.

First of all, we define a lattice that collects sets of types. The intuition behind them is that they contain all the types of which a variable cannot be instance of. Thus let $\overline{\mathsf{NT}}$ be this domain, i.e., $\overline{\mathsf{NT}} = \wp(\overline{\mathsf{T}})$. Its concretization contains all the types that are not in the given set or that are subtypes of one of the types in the set. Formally, $\gamma_{\overline{\mathsf{NT}}}(\bar{\mathsf{f}}) = \{[\mathsf{v} \mapsto \bar{\mathsf{t}} : \mathsf{v} \in dom(\bar{\mathsf{f}}) \wedge \nexists \overline{\mathsf{t}'} \in \bar{\mathsf{f}}(\mathsf{v}) : \bar{\mathsf{t}} \sqsubseteq_{\overline{\mathsf{T}}} \overline{\mathsf{t}'}]\}$. The ordering operator on $\overline{\mathsf{NT}}$ is not the subset operator, as more types we have, more precise we are. On the other hand, it is neither the superset operator, as several types may be subtypes of the same type, and they are different elements of the set. Thus the ordering operator is defined as

$$\overline{\mathsf{nt}_1} \sqsubseteq_{\overline{\mathsf{NT}}} \overline{\mathsf{nt}_2} \Leftrightarrow \forall \mathsf{v} \in dom(\overline{\mathsf{nt}_2}) : \mathsf{v} \in dom(\overline{\mathsf{nt}_1}) \wedge \forall \overline{\mathsf{t}_2} \in \overline{\mathsf{nt}_2}(\mathsf{v}) : \exists \overline{\mathsf{t}_1} \in \overline{\mathsf{nt}_1}(\mathsf{v}) : \overline{\mathsf{t}_2} \sqsubseteq_{\overline{\mathsf{T}}} \overline{\mathsf{t}_1}$$

The other lattice operators are the ones induced by $\sqsubseteq_{\overline{\mathsf{NT}}}$. The bottom element for a single variable is the set containing only the top type (e.g., Object in Java or Any in Scala). The concretization of this element would be all the types that are not subtype of the top type, i.e., the empty set. We denote by $\bot_{\overline{\mathsf{NT}}}$ a function that relates all the variables to $\{\top_{\overline{\mathsf{T}}}\}$. Formally, $\bot_{\overline{\mathsf{NT}}} = \lambda \mathsf{x}.\{\top_{\overline{\mathsf{T}}}\}$. The top element is the empty function. Formally, $\langle \overline{\mathsf{NT}}, \sqsubseteq_{\overline{\mathsf{NT}}}, \sqcup_{\overline{\mathsf{NT}}}, \sqcap_{\overline{\mathsf{NT}}}, \bot_{\overline{\mathsf{NT}}}, \emptyset \rangle$.

Semantics

$$\overline{\mathbb{NT}}[\![\mathsf{x} = \mathsf{y}, \overline{\mathsf{nt}}]\!] = \overline{\mathsf{nt}}[\mathsf{x} \mapsto \overline{\mathsf{nt}}(\mathsf{y})]$$
$$\overline{\mathbb{NT}}[\![\mathsf{x} = \mathsf{new}\ \mathsf{T}, \overline{\mathsf{nt}}]\!] = \overline{\mathsf{nt}}[\mathsf{x} \mapsto \emptyset]$$
$$\overline{\mathbb{NT}}[\![\mathsf{x} = (\mathsf{T})\mathsf{y}, \overline{\mathsf{nt}}]\!] = \overline{\mathsf{nt}}[\mathsf{x} \mapsto \overline{\mathsf{nt}}(\mathsf{y})]$$
$$\overline{\mathbb{NT}}[\![\mathsf{assume}(!\ \mathsf{x}\ \mathsf{instanceof}\ \mathsf{T}, \overline{\mathsf{nt}})]\!] = \overline{\mathsf{nt}}[\mathsf{x} \mapsto \overline{\mathsf{nt}}(\mathsf{x}) \cup \{\mathsf{T}\}]$$
$$\overline{\mathbb{NT}}[\![\mathsf{x.M}(\mathsf{y1}, ..., \mathsf{yn}), \overline{\mathsf{dt}}]\!] = \overline{\mathbb{DT}}[\![\mathsf{assume}(\mathsf{reaname}(\mathsf{postcondition_M})), \top_{\overline{\mathsf{NT}}}]\!]$$

We are interested in collecting which types a variable is not instance of. When a variable y is assigned to x (with or without cast), we track that the types x cannot be instance of are the same of y. When we assign a new instance of a class to a variable, we forget everything we knew about that variable. In this way, we lose some information: we could consider that the variable cannot be instance of all the types that are not subtype of the instantiated class. On the other hand, this would be computationally expensive, the same information is already contained in $\overline{\mathsf{DT}}$, and we are going to combine this domain with $\overline{\mathsf{NT}}$. $\overline{\mathsf{NT}}$ is also interested to approximate information when a boolean condition like x instanceof List is tested to false. For instance, when we analyze the if statement if(x instanceof List) C1; else C2, we know that in the else branch x cannot be instance of List, and we add List to the state of the $\overline{\mathsf{NT}}$ domain before analyzing C2. The method call is managed as in $\overline{\mathsf{DT}}$: method calls may have side effects on variables, arbitrarily changing their dynamic type. In order to preserve the soundness of our approach, the only information we have after the method call is that the postcondition of the called method holds.

Running Example The $\overline{\mathsf{NT}}$ domain models information on the example presented in Figure 2 when we test to false conditions containing instanceof. Thus at line 4 (i.e., the else branch of the if statement at line 2) its state is $[\mathsf{x} \mapsto \{\mathsf{Something}\}]$. At line 6 (i.e., the else branch of the if statement at line 4) its state is $[\mathsf{x} \mapsto \{\mathsf{Something}, \mathsf{None}\}]$. Unluckily, $\overline{\mathsf{NT}}$ alone is not enough in order to prove that line 6 is unreachable, as it knows nothing about the static or dynamic types of x.

4.4 $\overline{\mathsf{FT}}$: Reduced Product of $\overline{\mathsf{DT}}$ and $\overline{\mathsf{NT}}$

The $\overline{\mathsf{DT}}$ domain infers the dynamic type of a variable. $\overline{\mathsf{NT}}$ collects the types a variable cannot be instance of. Thus these domains model different types of information, but we can mutually refine them. In particular, we are interested in checking if the information contained by $\overline{\mathsf{NT}}$ about a variable is not compatible with the one contained in $\overline{\mathsf{DT}}$. In this case, we know that this point of the program is unreachable, and thus we can refine our domain to the bottom state. For instance, consider an abstract sealed class I (thus it cannot be instantiated nor extended by external code) that is implemented by two classes O1 and O2. If we know from $\overline{\mathsf{DT}}$ that the dynamic type of a variable x is I, but from $\overline{\mathsf{NT}}$ we know that x cannot be instance of O1 nor O2, then there is no possible type for x.

Line	$\overline{\mathsf{DT}}$	$\overline{\mathsf{NT}}$	Reduced $\overline{\mathsf{FT}}$
2	$[\mathsf{x} \mapsto \mathsf{Option}]$	$\emptyset$	
3	$[\mathsf{x} \mapsto \mathsf{Something}]$	$\emptyset$	
4	$[\mathsf{x} \mapsto \mathsf{Option}]$	$[\mathsf{x} \mapsto \{\mathsf{Something}\}]$	
5	$[\mathsf{x} \mapsto \mathsf{None}]$	$[\mathsf{x} \mapsto \{\mathsf{Something}\}]$	
6	$[\mathsf{x} \mapsto \mathsf{Option}]$	$[\mathsf{x} \mapsto \{\mathsf{Something}, \mathsf{None}\}]$	$\bot_{\overline{\mathsf{FT}}}$

Table 3: Results of the analysis of the example

Formally, let be $\overline{\mathsf{FT}}$ the reduced product of $\overline{\mathsf{DT}}$ and $\overline{\mathsf{NT}}$, i.e., $\overline{\mathsf{FT}} = \overline{\mathsf{DT}} \otimes \overline{\mathsf{NT}}$. The lattice operators of $\langle \overline{\mathsf{FT}}, \sqsubseteq_{\overline{\mathsf{FT}}}, \sqcup_{\overline{\mathsf{FT}}}, \sqcap_{\overline{\mathsf{FT}}}, \top_{\overline{\mathsf{FT}}}, \bot_{\overline{\mathsf{FT}}} \rangle$ are defined as usual when dealing with the product of domains. The reduction function $\overline{red}_{\overline{\mathsf{FT}}}$ is defined as

$$\overline{red}_{\overline{\mathsf{FT}}} : [\overline{\mathsf{FT}} \to \overline{\mathsf{FT}}]$$

$$\overline{red}_{\overline{\mathsf{FT}}}((\overline{\mathsf{d}}, \overline{\mathsf{n}})) = \begin{cases} \bot_{\overline{\mathsf{FT}}} & \text{if } \exists \mathsf{x} \in dom(\overline{\mathsf{d}}) : \overline{getType}(\mathsf{x}, \overline{\mathsf{d}}) = \overline{\mathsf{t}} \wedge \\ & \forall \overline{\mathsf{t_1}} \in \gamma_{\overline{\mathsf{T}}}(\overline{\mathsf{t}}) : \overline{abstract}(\overline{\mathsf{t_1}}) = \mathsf{false} \wedge \exists \overline{\mathsf{t_2}} \in \overline{\mathsf{n}} : \overline{\mathsf{t_1}} \sqsubseteq_{\overline{\mathsf{T}}} \overline{\mathsf{t_2}} \wedge \\ & \qquad \overline{\mathsf{t}} \text{ cannot be extended by external code} \\ (\overline{\mathsf{d}}, \overline{\mathsf{n}}) & \text{otherwise} \end{cases}$$

where $\overline{abstract} : [\overline{\mathsf{T}} \to \{\mathsf{true}, \mathsf{false}\}]$ is a function that, given a type, checks if the given type cannot be instantiated, e.g., it is an interface or an abstract class in Java, or a trait in Scala.

The reduction function is aimed at discovering if the information contained in $\overline{\mathsf{NT}}$ is not compatible with the information contained in $\overline{\mathsf{DT}}$. It returns $\bot_{\overline{\mathsf{FT}}}$ if and only if (i) all the types a variable can be instance of following the information contained in $\overline{\mathsf{DT}}$ are excluded at least by one type contained in $\overline{\mathsf{NT}}$, (ii) the type related to the variable in $\overline{\mathsf{DT}}$ cannot be extended by external code (e.g., it is declared as sealed in Scala). In this way we achieve the modularity in our analysis. Note that this reduction is partial, as there may be other ways to refine the information contained in the different domains. Anyway, we did not find useful other reductions of the information contained in this domain in order to improve the precision of our analysis.

The concretization of $\overline{\mathsf{FT}}$ is defined as the intersection of the pointwise application of $\gamma_{\overline{\mathsf{DT}}}$ and $\gamma_{\overline{\mathsf{NT}}}$. Formally, $\gamma_{\overline{\mathsf{FT}}}((\overline{\mathsf{d}}, \overline{\mathsf{n}})) = \gamma_{\overline{\mathsf{DT}}}(\overline{\mathsf{d}}) \cap \gamma_{\overline{\mathsf{NT}}}(\overline{\mathsf{n}})$. The semantics $\overline{\mathbb{FT}}$ is defined as the pointwise application of $\overline{\mathbb{DT}}$ and $\overline{\mathbb{NT}}$.

Theorem 1. *The fixpoint trace semantics based on $\overline{\mathbb{FT}}$ is sound with respect to the one based on $\mathbb{S}$.*

Running Example Table 3 reports the results of the $\overline{\mathsf{FT}}$ domain when applied to method extract of the example presented in Figure 2. We already pointed out how the information is captured by the abstract domains. The reduced product refines the information contained in the two domains at line 6 discovering that

it is not possible for variable x to be instance of Option, and not to be instance of Something and None at the same time. In fact, these two classes represent all the possible instances of Option. Note that Option is declared as sealed, hence it cannot be extended by external code. Therefore the reduced product is able to discover that line 6 is unreachable, and MatchError cannot be thrown. Previously, we discovered that $\overline{\mathsf{DT}}$ proved that the cast at line 3 is safe, thus, in a Java environment, a ClassCastException cannot be thrown. Using this information, we may improve the runtime performance (for instance removing the last if statement since the else branch is unreachable) and provide useful information to developers in order to debug programs, since we guarantee that pattern matching is exhaustive in the original Scala code.

5 Experimental Results

				Casts			MatchErrors		
	#m	**t**	**tm**	**v**	**nv**	**p**	**v**	**nv**	**p**
actors	1626	3'56"	145.35	47	104	31.13%	20	17	54.05%
collection	14578	10'21"	42.63	183	509	26.45%	42	74	36.21%
util	4926	8'21"	101.75	126	508	10.87%	36	65	35.64%
xml	2786	6'21"	136.59	108	286	27.41%	5	21	19.23%
mainlib	8218	12'02"	87.82	97	196	33.11%	37	12	75.51%
scala lib	35732	54'02"	90.72	725	1951	27.09%	156	208	42.86%

We applied our analysis to all the Scala library v. 2.7.7. We executed the analysis on an Intel Code 2 Quad CPU 2.83 GHz with 4 GB of RAM, running Windows 7, and the Java SE Runtime Environment 1.6.0_16-b01. The preceding table reports the experimental results. Column **#m** reports the number of analyzed methods. Column **t** reports the time required to analyze these methods[2], while **tm** reports the time required in average to analyze a single method in milliseconds. We report the number of validated cases (column **v**), not validated (**nv**) cases, and the overall precision (**p**) of our analysis when analyzing both type casts and reachability of MatchError exceptions.

The Scala library contains more than 35.000 methods. In average we analyze a method in 90 milliseconds. Thus our analysis is quite efficient, and it scales up. When we analyze the type casts, we are not able to distinguish between casts due to the compilation of pattern matchings, and other casts, since our analysis works on the code obtained after the transformations and simplifications performed by the Scala compiler. Hence our analysis takes into account all the casts contained in the program. We are able to automatically prove safe more than 27% of all the casts. We think that this result goes beyond our initial goal (that was, to precisely analyze pattern matching), since we proved the safety of more than 700 casts while there were only 364 MatchError exceptions (that means that there were 364 pattern matchings). On the other hand, in general a precision less than 30% is not particularly satisfying. We analyzed these warnings, and it turns out that

[2] This time takes into account also the heap analysis

the most part is due to erasure of type generics of Scala compiler. For instance, when we access the next element of a list of Integer objects, this is compiled into a program that takes the next element of a list of Object instances and casts it to Integer. Obviously, since we do not have information about generics, we cannot prove the safety of this code. We plan to apply our analysis to languages that preserve generic type information in order to study how this feature can improve the precision of our analysis. A minor part of the warnings is induced by the lack of contracts. Since the Scala library is not annotated, each time we have a method call we forget everything is contained in $\overline{\mathsf{DT}}$ and $\overline{\mathsf{NT}}$. We expect the we may improve the precision of our analysis adding some contracts concerning framing information in particular.

About the reachability of MatchError, we proved that about 43% of these exceptions is unreachable. This result is particular encouraging: even if we took into account a part of the information that can be used in pattern matching and we do not annotate the code with contracts, we proved that almost half of the existing pattern matchings is safe, and we can remove the MatchError exception. Usually we were not able to prove the exhaustiveness of pattern matchings when they contain some checks on numerical information. In a minor part of the cases, we failed to prove the exhaustiveness because the pattern matching was not exhaustive since it expected to receive a particular type of expression. Contracts would be the solution in order to express that a method expects that an argument can have only some specific types (e.g., not all the ones that are subtype of its static type), thus we expect that we could improve the precision adding this annotation.

6 Related Work

The related work is addressed mainly into two directions: the static analysis of type information in order to optimize virtual calls, and static analysis of pattern matching.

The first topic has been studied during the last 15 years. Object oriented languages introduced the idea of virtual calls: we do not know at compile time exactly which method we are calling, but we need some runtime information (for instance, the dynamic type of the object on which we are calling the method). Thus the binding of the called method is dynamic, and this step may require an overhead. Applying static analysis to reduce and bound the number of virtual calls improves the runtime performance of programs. In this context, many approaches were proposed [2]. Some of them performs an inter-procedural analysis [17], while our approach relies on contracts. Other approaches analyze the class hierarchy statically [9], and check if the virtual call can refer only to one implementation of the method in this context. These approaches usually consider information only about dynamic types.

A recent work [20] proposed a type analysis based on the abstract interpretation theory in order to optimize JavaScript code. This analysis is focused on the numerical information that can be assigned to a variable in order to infer

if we can adopt an Int32 type instead of a Float64 type. Using this information the code is optimized. This approach is focused on the numerical information and adopts this information to infer the types of variables, while our approach approximates information on the types of variables to check which casts are safe and if some statements are unreachable.

Recent work applied static analysis techniques to prove properties of pattern matching. Mitchell and Runciman [22] proposed an analysis to check if non-exhaustive pattern matching in Haskell could lead to failures. In particular, it infers preconditions that are strong enough to prove that the pattern matching never fails. Our approach is aimed at discovering which pattern matchings are non-exhaustive, instead of inferring which constraints are necessary in order to make exhaustive a pattern matching that is not.

Dotta *et al.* [10] introduced a verification system that checks the disjointness, reachability, and exhaustiveness of Scala pattern matching. Global information about the program is inferred through different kinds of formulas. The authors do not formalize how these formulas are inferred, and there is no proof about the soundness of the approach. Instead, we fully formalized our approach and we proved its soundness. On the other hand, their analysis tracks more information than ours. Their implementation requires up to some seconds to prove the exhaustiveness of some case studies. Our analysis tracks information only of type-based pattern matchings, but it is scales up.

7 Conclusion and Future Work

In this paper we presented a static type analysis focused on pattern matching. We introduced and combined two distinct abstract domains, each abstracting different information (dynamic typing, and not-instance-of information). We proved the soundness of our approach relying on the abstract interpretation framework. The analysis has been implemented in Sample, and the experimental results proved the scalability and the precision of our approach.

As future work, we plan to combine our analysis with some numerical domains in order to study if considering this information improves the precision of our analysis. In addition, we plan to exploit the information captured by our analysis to optimize the programs resulting from the Scala compiler, and to study how much this optimization may improve the runtime performance of some Scala benchmarks.

References

1. F#. http://research.microsoft.com/fsharp.
2. D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA '96*, pages 324–341. ACM, 1996.
3. L. Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 97. CRC Press, 2004.

4. D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *PLDI '90*. ACM, 1990.
5. P. Cousot. The calculational design of a generic abstract interpreter. In *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.
6. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. In *Theoretical Computer Science*, volume 277, pages 47–103. Elsevier Science publishers, 2002.
7. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*. ACM, 1977.
8. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL '79*. ACM, 1979.
9. J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95*. Springer-Verlag, 1995.
10. M. Dotta, P. Suter, and V. Kuncak. On static analysis for expressive pattern matching. Technical report, EPFL, 2008.
11. S. ECMA-335. *Common Language Infrastructure (CLI)*. ECMA, 4th edition, June 2006.
12. B. Emir. *Object-oriented pattern matching*. PhD thesis, EPFL, 2007.
13. B. Emir, Q. Ma, and M. Odersky. Translation correctness for first-order object-oriented pattern matching. In *APLAS '07*, LNCS, pages 54–70. Springer, 2007.
14. B. Emir, M. Odersky, and J. Williams. Matching objects with patterns. In *ECOOP '07*, pages 273–298. Springer, 2007.
15. P. Ferrara. Checkmate: a generic static analyzer of java multithreaded programs. In *Proceedings of SEFM '09*. IEEE Computer Society, 2009.
16. M. Hirzel, N. Nystrom, B. Bloom, and J. Vitek. Matchete: Paths through the pattern matching jungle. In *PADL '08*, LNCS. Springer, 2008.
17. R. Kumar and S. S. Chakraborty. Precise static type analysis for object oriented programs. *SIGPLAN Not.*, 42:17–26, 2007.
18. T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
19. F. Logozzo and M. Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In *Proceedings of CC '08*, LNCS. Springer-Verlag, 2008.
20. F. Logozzo and H. Venter. Rata: Rapid atomic type analysis by abstract interpretation. Application to JavaScript optimization. In *CC '10*, LNCS. Springer, 2010.
21. B. Meyer. *Object-Oriented Software Construction (2nd Edition)*. Prentice Hall, 1997.
22. N. Mitchell and C. Runciman. Not all patterns, but enough: an automatic verifier for partial but sufficient pattern matching. *SIGPLAN Not.*, 44:49–60, 2009.
23. M. Odersky. *The Scala Language Specification*, 2008.
24. A. Richard and O. Lhotak. Oomatch: pattern matching as dispatch in Java. In *OOPSLA '07*, pages 771–772. ACM, 2007.
25. F. Spoto. Julia: A Generic Static Analyser for the Java Bytecode. In *Proceedings of FTfJP'2005*, 2005.
26. D. Syme, G. Neverov, and J. Margetson. Extensible pattern matching via a lightweight language extension. *SIGPLAN Not.*, 42(9):29–40, 2007.