

Verifying pattern matching with guards in Scala

Mirco Dotta, Philippe Suter

EPFL – SAV '07

June 21, 2007
version 1.1

Outline

Introduction

- Scala

- reasoning about pattern matching

- status in Scala

- motivation

- project overview

Turning patterns into formulas

- general idea

- formalization of concepts

- axioms

- patterns

- miscellaneous

Implementation

- current status

- future work

Scala¹

- ▶ Scala is an object-oriented and functional language which is completely interoperable with Java.
- ▶ It removes some of the more arcane constructs of these environments and adds instead:
 1. a uniform object model
 2. pattern matching and higher-order functions
 3. novel ways to abstract and compose programs

¹The Scala Experiment – Can We Provide Better Language Support for Component Systems?
<http://lamp.epfl.ch/~odersky/talks/google06.pdf>

Algebraic Data Types in Scala

- ▶ Consider the following ADT definition:

```
type Tree = Node of Tree * int * Tree  
          | EmptyTree
```

- ▶ In Scala:

```
abstract class Tree
```

```
case class Node (left: Tree, value: Int, right: Tree) extends Tree
```

```
case object EmptyTree extends Tree
```

Pattern matching in Scala

Consider the following search function on a sorted binary tree:

```
def search(tree: Tree, value: Int): Boolean = tree match {  
  case EmptyTree  $\Rightarrow$  false  
  case Node(_,v,_) if (v == value)  $\Rightarrow$  true  
  case Node(l,v,_) if (v < value)  $\Rightarrow$  search(l,v)  
  case Node(_,v,r) if (v > value)  $\Rightarrow$  search(r,v)  
  case _  $\Rightarrow$  throw new Exception("...")  
}
```

Pattern matching in Scala - cont'd

You can:

- ▶ match on objects
- ▶ use recursive patterns

case Node(Node(–,5,–),–,–) \Rightarrow output("5 on its left!")

- ▶ use type restrictions

case Node(left: Node,–,–) \Rightarrow output("node on its left!")

- ▶ use guards
- ▶ use wildcards

Pattern matching

In general, two interesting properties:

- ▶ completeness
- ▶ disjointness

(both \Rightarrow partitioning)

Enforcement of these properties varies among languages.

Status in Scala

In Scala:

- ▶ completeness is not required
 - ▶ `MatchException` raised if no match is found
- ▶ completeness can be checked to some extent
 - ▶ only for **sealed** classes
 - ▶ guards are taken into account very conservatively
- ▶ disjointness is neither required nor checkable
- ▶ unreachable patterns are forbidden

Scala status - cont'd

Current situation:

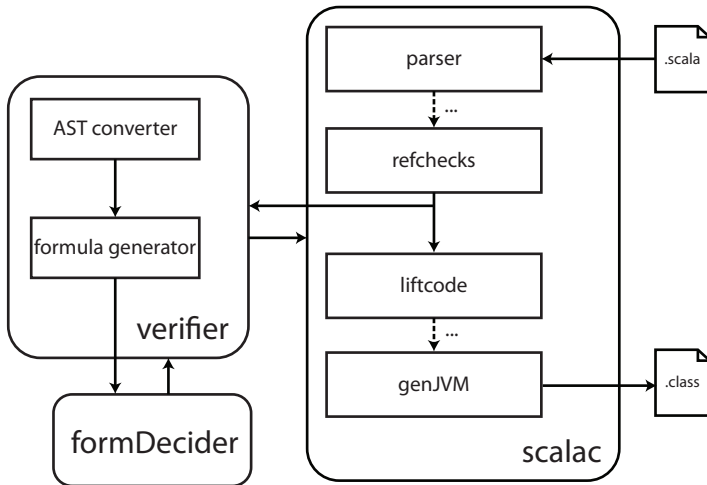
- ▶ little help from compiler
 - ▶ too conservative
 - ▶ Scala users keep asking for improved completeness checks
- ▶ ensuring disjointness is left to the developers

There is room for improvements using formal verification techniques.

Extending the Scala compiler

1. Analysis is implemented as an additional phase in the compiler.
2. Pattern matching subtrees and the related hierarchy are retrieved from the compiler environment and AST.
3. This information is used to generate an intermediate representation.
4. From there, formulas are constructed and fed to formDecider.
5. Based on the results, warning/error messages are sent back to the compiler.

The big picture



From patterns to formulas

- ▶ We want to create formulas – in *FOPL* – to prove completeness and disjointness.
- ▶ The process takes two aspects:
 - ▶ define a mapping from pattern expressions to formulas
 - ▶ how to represent types of classes and objects?
 - ▶ how to represent constructor parameters?
 - ▶ how to deal with recursive constructs?
 - ▶ how to include guards?
 - ▶ how about primitive types? and strings?
 - ▶ define completeness and disjointness
 - ▶ what axioms do we need?
 - ▶ how do formulas relate to each other?

Formalizing completeness and disjointness

Consider a pattern-matching expression E :

```
 $t$  match {  
  case  $p_1 \Rightarrow \dots$   
  ...  
  case  $p_i \Rightarrow \dots$   
}
```

Assume we have a predicate $\xi(t, p)$ such that $\forall i, \xi(t, p_i)$ is true iff the pattern p_i matches the expression t .

- ▶ E is complete $\iff \bigvee_i \xi(t, p_i)$
- ▶ E is disjoint $\iff \forall i, j, i \neq j \implies \neg(\xi(t, p_i) \wedge \xi(t, p_j))$

Formalizing patterns

Types can naturally be represented as sets

- ▶ $t: \text{Node} \mapsto t \in \text{Node}$

Subtyping can be seen as set inclusion

- ▶ **case class** `Node(...)` **extends** `Tree` $\mapsto \text{Node} \subseteq \text{Tree}$

Properties of ADT are used to generate axioms

- ▶ $\forall t \in \text{Tree}, t \in \text{Node}(...) \oplus t \in \text{EmptyTree}$

Formalizing patterns – cont'd

Objects are represented as singletons

► **case object** $\text{Leaf} \mapsto \text{Leaf} = \{\text{leaf}_0\}$

Types of constructor parameters are represented by functions

► **case class** $\text{Node}(\text{left}: \text{Tree}, \text{right}: \text{Tree}) \mapsto$
 $\forall n \in \text{Node} \ (\Psi_{\text{Node}, \text{left}}(n) \in \text{Tree} \wedge \Psi_{\text{Node}, \text{right}} \in \text{Tree})$

The above transformations, along with the information about the selector's type, define *axioms* about E .

Example – Axioms

```
abstract class Tree  
case class Node(left:Tree,right:Tree) extends Tree  
case object Leaf extends Tree
```

```
t: Tree match { ... }
```

$$t \in Tree$$
$$\wedge Node \subseteq Tree \wedge Leaf \subseteq Tree \wedge Leaf = \{leaf_0\}$$
$$\wedge \forall t_0 \in Tree, t_0 \in Node(...) \oplus t_0 \in Leaf$$
$$\wedge \forall n \in Node \ (\Psi_{Node, left}(n) \in Tree \wedge \Psi_{Node, right} \in Tree)$$

Axioms – cont'd

Recall that the formulas $\xi(t, p_i)$ correspond to the patterns p_i .

- ▶ Each of these formulas is in the form $A(t) \implies \Pi(p_i)$, where $A(t)$ are the axioms previously mentioned, and $\Pi(p_i)$ a formula depending on p_i .
- ▶ The formula for completeness $\bigvee_i \xi(t, p_i)$ hence becomes $\bigvee_i (A(t) \implies \Pi(p_i))$

Simplified, this becomes: $A(t) \implies \bigvee_i \Pi(p_i)$

Translation of patterns

The “root” type in the pattern is assigned to the selector

► $t \text{ match } \{ \text{case Node}(\dots) \Rightarrow \dots \} \mapsto t \in \text{Node}$

Aliases² are bound to fresh names

► $\text{case Node}(\text{left: Node}, \dots) \Rightarrow \dots$
 $\mapsto \text{left}_{\text{fresh}} = \Psi_{\text{Node}, \text{left}}(t) \wedge \text{left}_{\text{fresh}} \in \text{Node}$

Wildcards generate no constraints

► $\text{case } _ \Rightarrow \dots \mapsto \text{true}$

² the practical implementation slightly differs when proving completeness

Translation of patterns – cont'd

Guards are, to some extent, translated to formulas:

- ▶ equality and arithmetic operators are kept “as it”
- ▶ `equals` is always considered side-effect free
- ▶ dynamic type tests are converted to set membership
 - ▶ $o.\text{isInstanceOf}[Type] \mapsto o \in Type$
- ▶ other method calls are ignored

The result of the transformation is a predicate, whose parameters are the selector and the aliases defined in the pattern.

It is added as a conjunction to the main formula.

Matching on lists

Scala, as a language making an extensive use of lists, has a dedicated syntax for them:

```
z match {  
  case Nil  $\Rightarrow$  ...  
  case x :: xs  $\Rightarrow$  ...  
}
```

...but this is essentially syntactic sugar for the following hierarchy:

```
sealed abstract class List  
case final class ::(List, List) extends List  
case object Nil extends List
```

Implementation status

- ▶ large supported subset of Scala pattern matching expressions
- ▶ generation of formulas for completeness and disjointness
- ▶ integration with formDecider
- ▶ scalac integration under way ...

Future work

Some issues we want to address in the future:

- ▶ ... complete scalac integration :)
- ▶ allow matching on string constants
- ▶ improve support for primitive types
- ▶ implement limited support for external variables and functions
- ▶ ... oh, well, you always find something to do

Questions ?

One for the road...

```
sealed abstract class Arith
case class Sum(l: Arith, r: Arith) extends Arith
case class Prod(n: Num, f: Arith) extends Arith
case class Num(n: Int) extends Arith

def eval(a: Arith): Int = (a: @verified) match {
  case Sum(l, r) => eval(l) + eval(r)
  case Prod(Num(n), f) if(n == 0) => 0
  case Prod(Num(n), f) if(n != 0) => n * eval(f)
  case Num(n) => n
}
```


$$\begin{aligned}
 & a \in \text{Arith} \wedge \text{Sum} \subseteq \text{Arith} \wedge \text{Prod} \subseteq \text{Arith} \wedge \text{Num} \subseteq \text{Arith} \\
 & \wedge \forall a_0 \in \text{Arith}, ((a_0 \in \text{Sum} \oplus a_0 \in \text{Prod}) \wedge (a_0 \in \text{Sum} \oplus a_0 \in \text{Num}) \\
 & \wedge (a_0 \in \text{Prod} \oplus a_0 \in \text{Num})) \wedge \forall s_0 \in \text{Sum}, (\Psi_{\text{Sum},l}(s_0) \in \text{Arith} \\
 & \wedge \Psi_{\text{Sum},r}(s_0) \in \text{Arith}) \wedge \forall p_0 \in \text{Prod}, (\Psi_{\text{Prod},n}(p_0) \in \text{Num} \\
 & \wedge \Psi_{\text{Prod},f}(s_0) \in \text{Arith}) \wedge \forall n_0 \in \text{Num}, \Psi_{\text{Num},n}(n_0) \in \mathbb{N}
 \end{aligned}$$

\implies

$$\begin{aligned}
 & ((l_{\text{fresh}} = \Psi_{\text{Sum},l}(a) \wedge r_{\text{fresh}} = \Psi_{\text{Sum},r}(a)) \implies a \in \text{Sum}) \\
 & \vee ((f_{\text{fresh}} = \Psi_{\text{Prod},f}(a) \wedge n_{\text{fresh}} = \Psi_{\text{Num},n}(\Psi_{\text{Prod},l}(a))) \implies a \in \text{Prod} \\
 & \wedge \Psi_{\text{Prod},l}(a) \in \text{Num} \wedge n_{\text{fresh}} = 0) \\
 & \vee ((f_{\text{fresh}'} = \Psi_{\text{Prod},f}(a) \wedge n_{\text{fresh}'} = \Psi_{\text{Num},n}(\Psi_{\text{Prod},l}(a))) \implies a \in \text{Prod} \\
 & \wedge \Psi_{\text{Prod},l}(a) \in \text{Num} \wedge n_{\text{fresh}'} \neq 0) \\
 & \vee (n_{\text{fresh}''} = \Psi_{\text{Num},n}(a) \implies a \in \text{Num})
 \end{aligned}$$