Verifying pattern matching with guards in Scala

Mirco Dotta, Philippe Suter

EPFL – SAV '07

June 20, 2007
version 0.1a

Outline

Scala[1]

- Scala is an object-oriented and functional language which is completely interoperable with Java.

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

Scala[1]

- 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

[1] The 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:

$$\textbf{type}\ \text{Tree} = \text{Node of Tree} * \text{int} * \text{Tree} \\ \qquad\qquad\quad \mid \text{EmptyTree}$$

Algebraic Data Types in Scala

- Consider the following ADT definition:

$$\textbf{type}\ \text{Tree} = \text{Node of Tree} * \text{int} * \text{Tree} \\ \mid \text{EmptyTree}$$

- In Scala:

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:

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 ⇒ false
    case Node(_,v,_) if(v == value) ⇒ true
    case Node(l,v,_) if(v < value) ⇒ search(l,v)
    case Node(_,v,r) if(v > value) ⇒ search(r,v)
    case _ ⇒ throw new Exception("...")
}
```

Pattern matching in Scala - cont'd

You can:

Pattern matching in Scala - cont'd

You can:

- match on objects

Pattern matching in Scala - cont'd

You can:

- match on objects
- use recursive patterns

case Node(Node(_,5,_),_,_) ⇒ output("5 on its left!")

Pattern matching in Scala - cont'd

You can:

- match on objects
- use recursive patterns

  case Node(Node(_,5,_),_,_) ⇒ output("5 on its left!")

- use type restrictions

  case Node(left: Node,_,_) ⇒ output("node on its left!")

Pattern matching in Scala - cont'd

You can:

- match on objects
- use recursive patterns

  case Node(Node(_,5,_),_,_) ⇒ output("5 on its left!")

- use type restrictions

  case Node(left: Node,_,_) ⇒ output("node on its left!")

- use guards

Pattern matching in Scala - cont'd

You can:

- match on objects
- use recursive patterns

  case Node(Node(_,5,_),_,_) ⇒ output("5 on its left!")

- use type restrictions

  case Node(left: Node,_,_) ⇒ output("node on its left!")

- use guards
- use wildcards

In general, two interesting properties:

In general, two interesting properties:

- completeness
- disjointness

In general, two interesting properties:

- completeness
- disjointness

(both $\Rightarrow$ partitioning)

In general, two interesting properties:

- completeness
- disjointness

(both $\Rightarrow$ partitioning)

Enforcement of these properties varies among languages.

Status in Scala

In Scala:

Status in Scala

In Scala:

- completeness is not required

Status in Scala

In Scala:

- completeness is not required
  - `MatchException` raised if no match is found

Status in Scala

In Scala:

- completeness is not required
  - `MatchException` raised if no match is found
- completeness can be checked to some extent

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

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

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

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

Project goals

Current situation:

Project goals

Current situation:

- little help from compiler
  - too conservative
  - Scala users keep asking for improved completeness checks

Project goals

Current situation:

- little help from compiler
  - too conservative
  - Scala users keep asking for improved completeness checks
- ensuring disjointness is left to the developers
  - apparently, a less sought-after property

Project goals

Current situation:

- little help from compiler
  - too conservative
  - Scala users keep asking for improved completeness checks
- ensuring disjointness is left to the developers
  - apparently, a less sought-after property

There is room for improvements using formal verification techniques.

Extending the Scala compiler

1. Analysis is implemented as an additional phase in the compiler.

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.

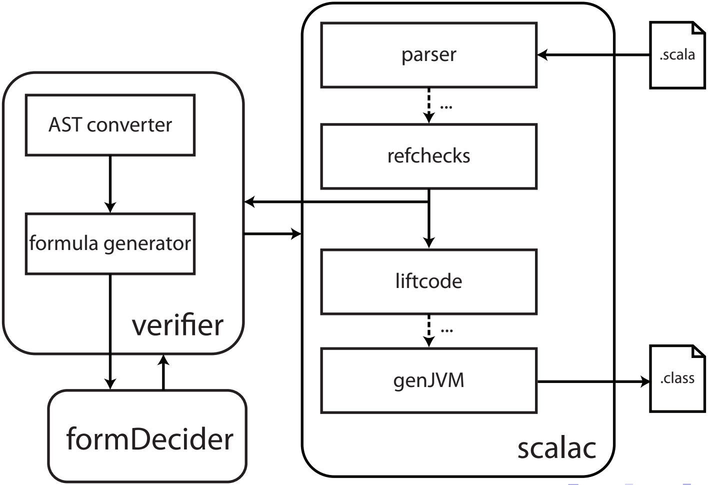
Extending the Scala compiler

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.

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.

From patterns to formulas

- We want to create formulas – in *FOPL* – to prove completeness and disjointness.
- The process can be split as follows:

From patterns to formulas

- We want to create formulas – in *FOPL* – to prove completeness and disjointness.
- The process can be split as follows:
  1. define a mapping from pattern expressions to formulas

From patterns to formulas

- We want to create formulas – in *FOPL* – to prove completeness and disjointness.
- The process can be split as follows:
  1. 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?

From patterns to formulas

- We want to create formulas – in *FOPL* – to prove completeness and disjointness.
- The process can be split as follows:
  1. 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?
  2. define completeness and disjointness

From patterns to formulas

- We want to create formulas – in *FOPL* – to prove completeness and disjointness.
- The process can be split as follows:
  1. 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?
  2. 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 $p_1 \Rightarrow \ldots$
    $\ldots$
    case $p_i \Rightarrow \ldots$
}

Formalizing completeness and disjointness

Consider a pattern-matching expression E:

t **match** $\{$
  case $p_1 \Rightarrow \ldots$
  $\ldots$
  case $p_i \Rightarrow \ldots$
$\}$

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.

Formalizing completeness and disjointness

Consider a pattern-matching expression E:

t **match** {
  case $p_1 \Rightarrow \ldots$
  $\ldots$
  case $p_i \Rightarrow \ldots$
}

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 $\iff \bigvee_i \xi(t, p_i)$

Formalizing completeness and disjointness

Consider a pattern-matching expression E:

t **match** $\{$
 case $p_1 \Rightarrow \ldots$
 $\ldots$
 case $p_i \Rightarrow \ldots$
$\}$

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: Node $\longmapsto$ $t \in Node$

Formalizing patterns

Types can naturally be represented as sets

- t: Node $\longmapsto t \in Node$

Subtyping can be seen as set inclusion

- **case class** Node(...) **extends** Tree $\longmapsto Node \subseteq Tree$

Formalizing patterns

Types can naturally be represented as sets

- t: Node $\longmapsto t \in Node$

Subtyping can be seen as set inclusion

- **case class** Node(...) **extends** Tree $\longmapsto Node \subseteq Tree$

Properties of ADT are used to generate axioms

- $\forall t \in Tree, t \in Node(\ldots) \oplus t \in EmptyTree$

Formalizing patterns – cont'd

Objects are represented as singletons

- **case object** Leaf $\longmapsto Leaf = \{leaf_0\}$

Formalizing patterns – cont'd

Objects are represented as singletons

- **case object** Leaf $\longmapsto$ $Leaf = \{leaf_0\}$

Types of constructor parameters are represented by functions

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

Formalizing patterns – cont'd

Objects are represented as singletons

- **case object** Leaf $\longmapsto$ $\mathit{Leaf} = \{\mathit{leaf}_0\}$

Types of constructor parameters are represented by functions

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

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** { ... }

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

Axioms – cont'd

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

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.

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 $\bigvee_i (A(t) \implies \Pi(p_i))$

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 **match** { **case** Node(...) ⇒ ... } $\longmapsto t \in Node$

[2] the practical implementation slightly differs when proving completeness

Translation of patterns

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

- t **match** { **case** Node(...) ⇒ ...} $\longmapsto t \in Node$

Aliases[2] are bound to fresh names

- **case** Node(left: Node, ...) ⇒ ...
  $\longmapsto left_{\text{fresh}} = \Psi_{\text{Node,left}}(t) \wedge left_{\text{fresh}} \in Node$

[2] the practical implementation slightly differs when proving completeness

Translation of patterns

The "root" type in the pattern is assigned to the selector

- t **match** { **case** Node(...) ⇒ ...} $\longmapsto t \in Node$

Aliases[2] are bound to fresh names

- **case** Node(left: Node, ...) ⇒ ...
  $\longmapsto left_{\mathsf{fresh}} = \Psi_{\mathsf{Node,left}}(t) \wedge left_{\mathsf{fresh}} \in Node$

Wildcards generate no constraints

- **case** _ ⇒ ... $\longmapsto true$

[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.isInstanceOf[`*Type*`]` $\longmapsto o \in \mathit{Type}$
- other method calls are ignored

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.isInstanceOf[`*Type*`]` $\longmapsto o \in \mathit{Type}$
- 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 ⇒ ...
    case x :: xs ⇒ ...
}

... 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

Future work

Some issues we want to address in the future:

- Actually plug it into scalac :)
- Allow matching on string constants.
- Improve support for primitive types.
- Implement limited support for external variables and functions

Future work

Some issues we want to address in the future:

- Actually plug it into scalac :)
- 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
}
```

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

$$\Longrightarrow$$

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