# Linter

## static analysis for Scala

(Scala Workshop, 02.07.2013)

Matic Potočnik - @HairyFotr
University of Ljubljana

# **Linter** – static analysis for Scala

## What is Linter?

Scala compiler plug-in
for doing static analysis
started by Jorge Ortiz

```
Linter then...  6 checks
Linter now... ~70 checks
```

# **Linter** – static analysis for Scala

## How does it work?

Pattern matching
on the Scala AST

Abstract interpretation,
mostly for Int and String

# **Linter** – static analysis for Scala

## Option and collection related checks:

- Replace `if(opt.isDefined) opt.get else alternative`
  with `opt.getOrElse(alternative)`

- Replace `col.find(x => cond).isDefined`
  with `col.exists(x => cond)`

- Replace `col.flatMap(x => if(cond) Some(x) else None)`
  and `col.flatMap(x => if(cond) List(x) else Nil)`
  with `col.filter(x => cond)`

- Using `opt.size` is practically always by mistake

- ...

# **Linter** – static analysis for Scala

**Numeric math checks:**

- Loss of precision
  - Use log1p(x) instead of log(1 + x)
  - Use expm1(x) instead of exp(x) – 1
  - BigDecimal(0.55555555555555555555)
    ... we get 0.5555555555555556

- Manually calculating functions like

  abs, signum, isNaN, ...

# **Linter** – static analysis for Scala

**Repeated part of condition:**

```
if(data.isEmpty && arg == “shortname” || arg == “name”) {

    ...

} else if(arg == “shortname” || arg == “name”) {

    ...

} else ...
```

**Warning: This condition has appeared earlier in the if-else chain, and will never hold here.**

# **Linter** – static analysis for Scala

**String length approximation:**

```
val msg = getMessage(arg)

...

if(msg.isEmpty) {

   ...

}


// So far so good...
```

# **Linter** – static analysis for Scala

**String length approximation:**

```
val msg = getTimeStamp + getMessage(arg)

...

if(msg.isEmpty) {
   ...
}
```

is Long, which means its String length is from 1 to 20

**Warning: This condition will never hold.**

# **Linter** – static analysis for Scala

## Detecting runtime exceptions early:

```
val foo = bar.replaceAll("?", ".")
```

**Warning:** **Regex pattern syntax error:**
**Dangling meta character '?'**

## Example of Int abstract interpretation:

```
for (i ← 1 to 10) {
  val div = 1/(i-1)

  if(i > 15) ...
}
```

**Warning:** **You will likely divide by zero here.**
**Warning:** **This condition will never hold.**

# **Linter** – static analysis for Scala

Unused method parameters

“Suspicious” code

Unused sealed traits

**Many more checks**

Unnecessary ifs

Pattern matching checks

# **Linter** – static analysis for Scala

## How to get Linter?

Source code released under
the Apache License, ver 2.0

https://github.com/HairyFotr/linter

To use Linter in your project
follow the instructions there

# Future work

- Add more checks ← Everyone has ideas or a favorite bug
- Improve (or replace) abstract interpreters and support more types (esp. Boolean, collections)
- Auto-generate parts of docs, config, and tests (all three are incomplete at the moment)

Matic Potočnik - @HairyFotr
University of Ljubljana

# InSynth: Type-Driven Interactive Synthesis of Code Snippets

Tihomir Gvero

# Motivation

- Large APIs and libraries
  - $\sim 4000$ classes in Java 6.0 standard library
- Using those APIs (for the first time) can be
  - Tedious
  - Time consuming
- Developers should focus on solving creative tasks
- Manual Solution
  - Read Documentation
  - Inspect Examples
- Automation = Code synthesis + Code completion

# Our Solution

- **InSynth**: Interactive Synthesis of Code Snippets
- Input:
  - Scala partial program
  - Cursor point
- We automatically extract:
  - Declarations in scope (with/without statistics from corpus)
  - Desired type
- Algorithm
  - Complete
  - Efficient – output N expressions in less than T ms
  - Effective – favor useful expressions over obscure ones
  - Generates expressions with higher order functions
- Output
  - Ranked list of expressions

# Sequence of Streams

```
def main(args:Array[String]) = {
    var body:String = "email.txt"
    var sig:String = "signature.txt"
    var inStream:SeqInStr =
    …
}
```

# Sequence of Streams

```
def main(args:Array[String]) = {
    var body:String = "email.txt"
    var sig:String = "signature.txt"
    var inStream:SeqInStr =
    …
}
```

```
new SeqInStr(new FileInStr(sig), new FileInStr(sig))
new SeqInStr(new FileInStr(sig), new FileInStr(body))
new SeqInStr(new FileInStr(body), new FileInStr(sig))
new SeqInStr(new FileInStr(body), new FileInStr(body))
new SeqInStr(new FileInStr(sig), System.in)
```

# Sequence of Streams

```
def main(args:Array[String]) = {
    var body:String = "email.txt"
    var sig:String = "signature.txt"
    var inStream:SeqInStr =
    …
}
```

- new SeqInStr(new FileInStr(sig), new FileInStr(sig))
- new SeqInStr(new FileInStr(sig), new FileInStr(body))
- new SeqInStr(new FileInStr(body), new FileInStr(sig))
- new SeqInStr(new FileInStr(body), new FileInStr(body))
- new SeqInStr(new FileInStr(sig), System.in)

# Sequence of Streams

```
def main(args:Array[String]) = {
    var body:String = "email.txt"
    var sig:String = "signature.txt"
    var inStream:SeqInStr = new SeqInStr(new FileInStr(sig), new FileInStr(body))
    …
}
```

# Sequence of Streams

```
def main(args:Array[String]) = {
    var body:String = "email.txt"
    var sig:String = "signature.txt"
    var inStream:SeqInStr = new SeqInStr(new FileInStr(sig), new FileInStr(body))
    …
}
```

Imported over 3000 declarations

Executed in less than 250ms

# TreeFilter (HOF)

```
def filter(p: Tree => Boolean): List[Tree] = {
   val ft:FilterTreeTraverser =
   ft.traverse(tree)
   ft.hits.toList
}
```

# TreeFilter (HOF)

```
def filter(p: Tree => Boolean): List[Tree] = {
   val ft:FilterTreeTraverser =
   ft.traverse(tree)
   ft.hits.toList
}
```

```
new FilterTreeTraverser(x => p(x))
new FilterTreeTraverser(x => isType)
new FilterTreeTraverser(x => p(tree))
new FilterTreeTraverser(x => new Wrapper(x).isType)
new FilterTreeTraverser(x => p(new Wrapper(x).tree))
```

# TreeFilter (HOF)

```
def filter(p: Tree => Boolean): List[Tree] = {
  val ft:FilterTreeTraverser =
  ft.traverse(tree)
  ft.hits.toList
}
```

- new FilterTreeTraverser(x => p(x))
- new FilterTreeTraverser(x => isType)
- new FilterTreeTraverser(x => p(tree))
- new FilterTreeTraverser(x => new Wrapper(x).isType)
- new FilterTreeTraverser(x => p(new Wrapper(x).tree))

# TreeFilter (HOF)

```
def filter(p: Tree => Boolean): List[Tree] = {
   val ft:FilterTreeTraverser = new FilterTreeTraverser(x => p(x))
   ft.traverse(tree)
   ft.hits.toList
}
```

# TreeFilter (HOF)

```
def filter(p: Tree => Boolean): List[Tree] = {
   val ft:FilterTreeTraverser = new FilterTreeTraverser(x => p(x))
   ft.traverse(tree)
   ft.hits.toList
}
```

Imported over 4000 declarations

Executed in less than 300ms

- “Complete Completion using Types and Weights” (PLDI ’13)
- InSynth is Eclipse plugin (part of Scala IDE EcoSystem)

http://lara.epfl.ch/w/insynth

# Questions? Opinions?

Meet me in the break.

p.giarrusso@gmail.com

Details in blog post at:
blaisorbladeprog.blogspot.com

# Flexible Implicits (from Agda) for Scala

Paolo G. Giarrusso

2 July 2013, Scala 2013

# Problem statement

Methods can take implicit parameters

after the other ones.

# Problem statement

Methods can take implicit parameters
after the other ones.

This is irregular and restrictive!

# Problem statement

Methods can take implicit parameters
after the other ones.

This is irregular and restrictive!

With dependent types, even more restrictive.

# Solution

Methods can take implicit parameters ~~after the other ones.~~ anywhere!

# Example 1

Dependent methods (example in macros):

```
def method(c: Context)(arg1: c.Tree,
  arg2: c.Tree)
```

`Trees` are tagged by `Contexts`.

# Example 1

Dependent methods (example in macros):

```
def method(c: Context)(arg1: c.Tree,
  arg2: c.Tree)
```

`Trees` are tagged by `Contexts`.

Uses:

- *cake pattern*
- `Trees` with different `Contexts` should be incompatible (useful for typed DSLs!)

# Example 1: Caller-side

```
def method (  c: Context  ) (arg1: c.Tree,
  arg2: c.Tree)

val v1: Context = …
val v2: v1.Tree = …
val v3: v1.Tree = …

method (  v1  ) (v2, v3)
```

Passing `v1` is redundant!

# Solution

```
def method (  c: Context  ) (arg1: c.Tree,
  arg2: c.Tree)

val v1: Context = …
val v2: v1.Tree = …
val v3: v1.Tree = …

method (  v1  ) (v2, v3)
```

# Solution

```
def method (  c: Context  ) (arg1: c.Tree,
  arg2: c.Tree)

val v1: Context = …
val v2: v1.Tree = …
val v3: v1.Tree = …

method (  v1  ) (v2, v3)
```

# Solution

```
def method [[ c: Context ]] (arg1: c.Tree,
  arg2: c.Tree)

val v1: Context = …
val v2: v1.Tree = …
val v3: v1.Tree = …

method [[ v1 ]] (v2, v3)
```

# Solution

```
def method [[ c: Context ]] (arg1: c.Tree,
  arg2: c.Tree)

val v1: Context = …
val v2: v1.Tree = …
val v3: v1.Tree = …

method [[ v1 ]] (v2, v3)
```

# Solution

```
def method [[ c: Context ]] (arg1: c.Tree,
  arg2: c.Tree)

val v1: Context = …
val v2: v1.Tree = …
val v3: v1.Tree = …

method          (v2, v3)
```

# Solution

```
def method [[ c: Context ]] (arg1: c.Tree,
  arg2: c.Tree)

val v1: Context = …
val v2: v1.Tree = …
val v3: v1.Tree = …

method              (v2, v3)
```

Use current implicit inference.

# Solution

```
def method [[ c: Context ]] (arg1: c.Tree,
  arg2: c.Tree)

val v1: Context = …
val v2: v1.Tree = …
val v3: v1.Tree = …

method              (v2, v3)
```

Use current implicit inference.

Add rule for dependent method types.

# Example 2

```
def method (  implicit arg: T  ) : U ⇒ V

val u: U = …
val res = method
res(u)                          → Works

method(u)                       → Error: supply 'arg'!

method (implicitly) (u) → Correct version
```

A (common) pitfall

# Example 2

```
def method [[ implicit arg: T ]] : U ⇒ V

val u: U = …
val res = method
res(u)                          → Works

method(u)                       → Works too!

method [[  …  ]] (u)            → Specify 'arg'
```

A (common) pitfall, solved