

# Rethinking Scala Macros

Work in progress, not available yet

Eugene Burmako

École Polytechnique Fédérale de Lausanne  
<http://scalamacros.org/>

02 March 2014

# Outline

- ▶ What is Palladium?
- ▶ Planned features
- ▶ Planned deliverables
- ▶ Final words

What is Palladium?

# Project Palladium

- ▶ Successor of Project Kepler
- ▶ Goal of Project Kepler: bring macros to Scala
- ▶ Goal of Project Palladium: make macros in Scala easy to use

## Scala macros: the good parts

- ▶ Enable cool use cases that were previously impossible/impractical
- ▶ Have a significant community of research and production users
- ▶ A lot of popular libraries in Scala ecosystem use macros

## Scala macros: the bad parts

- ▶ Using macros is easy, developing macros is hard
- ▶ This contributes to the public image of metaprogramming
- ▶ Useful, but hacky and obscure

*I'm very envious of Racket macros, because it's very extensible.  
But I don't know how to do it for Haskell. TH is the nearest, but  
it's nowhere near.*

—Simon Peyton Jones

## Palladium goal #1: Being straightforward

```
coll.map(x => x + 1)
```



```
{  
  def fn(x: Int) = x + 1  
  val buf = Coll.newBuilder[T]  
  var i = 0  
  while (i < coll.length) { buf += fn(coll(i)); i += 1 }  
  buf.result  
}
```

- ▶ A canonical example that illustrates current problems with macros
- ▶ Currently possible, but prohibitively complex to get right
- ▶ To goal of Palladium is to make such macros writeable on autopilot

## Palladium goal #2: Being portable

The trick is to make this work with:

- ▶ Scala compilers other than `scalac`
- ▶ Integrated development environments
- ▶ Incremental compilation
- ▶ Interactive documentation
- ▶ Runtime reflection



## Summary

Palladium will make macros straightforward and portable

## Planned features

## Our running example

```
coll.map(x => x + 1)
```



```
{  
  def fn(x: Int) = x + 1  
  val buf = Coll.newBuilder[T]  
  var i = 0  
  while (i < coll.length) { buf += fn(coll(i)); i += 1 }  
  buf.result  
}
```

- ▶ Let's take another look at Paul's declosurify
- ▶ Possible but ridiculously hard at the moment
- ▶ How can Palladium help?

# Disclaimer

- ▶ What follows is just a sketch, nothing's implemented yet
- ▶ We might or might not be able to figure out everything
- ▶ But all in all, the plan seems reasonable enough
- ▶ After we have results, we'll see how/when this can be part of Scala

## Feature #1: Simple definitions

```
import scala.reflect._
import scala.language.macros

implicit class Mapper[Coll[_], A](coll: Coll[A]) {
  macro map[B](fn: A => B): Coll[B] = {
    val q(..$ps) => $body" = fn
    val newBuilder = t"Coll".companion.method("newBuilder")
    q"""
      def fn(..$ps) = $body
      val buf = $newBuilder[$A]
      var i = 0
      while (i < coll.length) { buf += fn(coll(i)); i += 1 }
      buf.result
    """
  }
}
```

- ▶ No longer necessary to split macro defs and macro impls
- ▶ No longer necessary to write tiresome `c.Expr` and `c.WeakTypeTag`

## Feature #2: Simple reflection

```
import scala.reflect._
import scala.language.macros

implicit class Mapper[Coll[_], A](coll: Coll[A]) {
  macro map[B](fn: A => B): Coll[B] = {
    val q(..$ps) => $body" = fn
    val newBuilder = t"Coll".companion.method("newBuilder")
    q"""
      def fn(..$ps) = $body
      val buf = $newBuilder[$A]
      var i = 0
      while (i < coll.length) { buf += fn(coll(i)); i += 1 }
      buf.result
    """
  }
}
```

- ▶ Explicit macro context will be gone, along with path dependencies
- ▶ Redesigned reflection API that makes introspection and codegen easy

## Feature #3: Simple trees

```
import scala.reflect._
import scala.language.macros

implicit class Mapper[Coll[_], A](coll: Coll[A]) {
  macro map[B](fn: A => B): Coll[B] = {
    val q(..$ps) => $body" = fn
    val newBuilder = t"Coll".companion.method("newBuilder")
    q"""
      def fn(..$ps) = $body
      val buf = $newBuilder[$A]
      var i = 0
      while (i < coll.length) { buf += fn(coll(i)); i += 1 }
      buf.result
    """
  }
}
```

- ▶ No more manual construction/deconstruction, reification, exprs
- ▶ Trees won't carry types or symbols, but will be typecheckable

## Feature #4: Simple types

```
import scala.reflect._
import scala.language.macros

implicit class Mapper[Coll[_], A](coll: Coll[A]) {
  macro map[B](fn: A => B): Coll[B] = {
    val q(..$ps) => $body" = fn
    val newBuilder = t"Coll".companion.method("newBuilder")
    q"""
      def fn(..$ps) = $body
      val buf = $newBuilder[$A]
      var i = 0
      while (i < coll.length) { buf += fn(coll(i)); i += 1 }
      buf.result
    """
  }
}
```

- ▶ Convenient notation to construct and deconstruct types
- ▶ No more tags, no more case `TypeRef(...)`, no more `appliedType`



## Feature #5: Simple symbols

```
import scala.reflect._
import scala.language.macros

implicit class Mapper[Coll[_], A](coll: Coll[A]) {
  macro map[B](fn: A => B): Coll[B] = {
    val q(..$ps) => $body" = fn
    val newBuilder = t"Coll".companion.method("newBuilder")
    q"""
      def fn(..$ps) = $body
      val buf = $newBuilder[$A]
      var i = 0
      while (i < coll.length) { buf += fn(coll(i)); i += 1 }
      buf.result
    """
  }
}
```

- ▶ Symbols as we know them should be gone for good
- ▶ Introspection serviced by Members, bindings handled by hygiene

## Feature #6: Inline expansion

- ▶ We can treat macro applications as folded regions of code
- ▶ When you press [+], a given macro application expands
- ▶ When you press [-], a given macro expansion collapses back

## Feature #7: Expansion error highlighting

- ▶ Inline expansion will provide long-awaited interactivity
- ▶ For one, errors in macro expansions are going to make sense
- ▶ Have an error? Click `[+]` and see what exactly causes it!

## Feature #8: Expansion error troubleshooting

- ▶ Quasiquotes can be smart, capturing locations they originate from
- ▶ That would enable tracking culprits of errors in generated code
- ▶ One could even imagine interactive fixes to codegen errors

## Feature #9: Inline debugging

- ▶ The concept of interactive expansion is also applicable to debugging
- ▶ Once a macro is expanded, you will be able to set breakpoints in expanded code

## Feature #10: Incremental compilation

SBT will correctly handle macro expansions:

- ▶ No more whole project recompilations on a tiny change in a macro
- ▶ Changes to macro arguments will recompile expansions
- ▶ Changes to macro bodies and their helpers will recompile expansions
- ▶ Changes to types introspected by macros will recompile expansions

# Summary

- ▶ Simple macro definitions
- ▶ Simple reflection API
- ▶ Interactive expansion
- ▶ Inline debugging
- ▶ Incremental compilation

Planned deliverables



# M1

- ▶ Aims to deliver a demoable prototype of the Palladium macro system
- ▶ That works nicely with the existing ecosystem of tools
- ▶ And is reasonably compatible with existing popular macros
- ▶ By ScalaDays 2014 (16-18 June)

## Component #1: New reflection API

- ▶ Reflection Core, a redesigned compile-time/runtime reflection library
- ▶ Interface shared between Scala, Dotty, Eclipse, IntelliJ, SBT, etc
- ▶ Spec'ed and developed independently of implementors

## Component #2: Hygienic quasiquotes

- ▶ Smart quasiquoting facility that respects hygiene and ref transparency
- ▶ Very much relies on getting trees right
- ▶ Denys will elaborate on that at Scala Days

## Component #3: AST interpretation

- ▶ Macros will run in an interpreter, ensuring portability and compatibility
- ▶ NB! Here we only need to interpret typed ASTs, relying on the fact that our host is going to provide a typechecking facility
- ▶ Having an AST interpreter is also useful beyond macro expansion
- ▶ For example, it will give us a nice, minimalistic REPL!

## Component #4: AST persistence

- ▶ In order to interpret macros, we need to store their ASTs
- ▶ And not only their ASTs, but also ASTs of their dependencies
- ▶ Ramping this up, how about we store ASTs for everything?!
- ▶ AST persistence is also useful beyond macro expansion

## Components #3+4: Runtime expansion

- ▶ AST interpretation and AST persistence work very well together
- ▶ Interpreted ASTs => we don't need the compiler to run macros
- ▶ Persistent ASTs => we don't need the compiler to setup environment
- ▶ As a result, we will be able to expand macros at runtime!!

## Component #5: Tooling infrastructure (SBT)

- ▶ At the moment, SBT doesn't know almost anything about macros
- ▶ A) If macro body changes, we've got to recompile, but we don't
- ▶ B) If macro data changes, we've got to recompile, but we don't
- ▶ With ASTs and interpretation traces, we can do so much better!

## Component #5: Tooling infrastructure (IDE)

- ▶ Not much can be done if macros are just arbitrary functions
- ▶ However with interpretation we can easily control expansions
- ▶ The model of `[+]`/`[-]` buttons for macro applications
- ▶ Both for interactive editing and debugging



# Summary

- ▶ Straightforward reflection API decoupled from compiler internals
- ▶ Hygienic quasiquotes which are essential for tree manipulations
- ▶ AST interpreter
- ▶ AST persistence
- ▶ Tooling infrastructure: incremental compilation and IDEs

Final words

# Status

- ▶ Palladium was kicked off just two weeks ago
- ▶ Most of the team is from EPFL with several external contributors
- ▶ It is a research platform for new metaprogramming technologies
- ▶ Targetting Scala and Dotty

## Feedback

- ▶ Your feedback and contributions are very much welcome
- ▶ Mailing list: [palladium-internals @ groups.google.com](mailto:palladium-internals@groups.google.com)
- ▶ Design documents: [Palladium Shared @ docs.google.com](https://docs.google.com)

# Summary

- ▶ Palladium will make macros straightforward and portable
- ▶ New reflection + AST interpretation + AST persistence + tooling
- ▶ Welcome to the future of Scala macros!