PROJECT KEPLER: WHAT'S UP?

Eugene Burmako

LAMP, EPFL

2011-10-18

Quick recap

Goal: Implement macros, a type-safe compile-time metaprogramming facility for Scala.

Inspiration: Mostly Nemerle, though it was enlightening to learn about certain aspects of Scheme, MetaML and Template Haskell.

Previously on Kepler: Unlike infamous C macros, state of the art macros work with ASTs and are deeply integrated with compilers, which makes them safe, expressive and composable. Read more on macrology in my previous talk.

Status update

Use-case: Kepler got a testbed – another pet project inside our lab, a LINQ-inspired datasource connectivity kit.

People: The experiment is being conducted by Christopher Vogt (the Scala Integrated Query guy) and Stefan Zeiger (the ScalaQuery guy). Christopher, Stefan and I will be shaping the face of macros, which provide the language integration part of LINQ.

Status: Designing macro-related Scala improvement documents (also known as SIDs). By the way, have you seen the new Scala improvement process site?

Outline

Today's talk will explore the approaches to bringing macros to Scala. The talk will realize the vague notions from the previous talk into concrete extensions to Scala language.

Topics covered:

- What's new
- Lightweight macros
- Macro annotations

We will discuss a lot of ideas today, but don't worry about feature bloat. Right now we're just brainstorming, but in subsequent installments we will leave only the absolutely necessary minimum of features.

Before we start

Everything mentioned below is highly experimental and can be changed at any time. By the time you're reading this I might have already abandoned this project, deleted the GitHub repo and cleaned up Google's cache.

I'm open to all comments, suggestions and criticisms. The design of macros is still in flux, so they can be easily adjusted to embrace your personal use-cases and make the world a better place to live. Share your thoughts with me at eugene.burmako@epfl.ch.

We have cookies

For a long time, macros in mainstream programming languages have been associated with preprocessor macros of C and C++. This has rightfully given bad connotations to the word “macro”.

To the contrast, macros proposed by Project Kepler:

- Are written in full-fledged Scala
- Operate on high-level and type-safe expression trees
- Run inside the compiler and can use all the semantic information it has
- Are not about syntax changes

Read up more to see this with your own eyes.

What’s new

The use-case

In a few days after my previous talk, I've overheard Martin saying that it would be fun to experiment with LINQ for Scala. At first, I didn't give much importance to that, but then it struck me – what a nice testbed for macros!

LINQ is a brilliant concept. Being released five years ago, even now it remains state of the art. However, it has its weaknesses: 1) relying on a clever but rigid compiler hardcode, 2) lack of composability, 3) unclear semantics of calls to external code.

With macros we can solve all these problems and ~~introduce new ones~~ do even better than that!

Macrology incoming

Before proceeding with this presentation, make sure you've read the slides from my previous talk.

The PDF linked above introduces the notions of compile-time metaprogramming, macro expansion, quasi-quoting and splicing. We're going to utilize all these concepts starting from the very next slide.

LINQ, the macro way

```
class Queryable[T, Repr](query: Query) {
  macro def filter(p: T => Boolean): Repr = <[
      val b = $this.newBuilder
      b.query = Filter($this.query, $reify(p))
      b.result
  ]>
}

val products = db.products
products.filter(p => p.startsWith(“foo”)).toList
```

Lightweight macros

As you can see, the invocation of the “filter” macro is indistinguishable from a regular method invocation.

The signature of the macro is also seamlessly integrated into the language. Macros get executed during the compile-time, so they process and return (possibly untyped) expression trees, however, compiler typechecks the invocations according to declared macro signatures and generates typeful binary representations for them.

Inside a macro one can access all the values that constitute the lexical scope, including “this” and implicits. For the nitpickers: yes, we also overload “implicitly[T]”.

Lightweight, but not crippled

Macros cannot access or modify run-time values, however, this does not make them second-class citizens.

Macro programmer can use the entire Scala library and whatever functions and objects he wishes. Certain functionality can be reused by both compile-time and run-time worlds. After all, macros are written in full-blown Scala!

See that call to “$reify(p)”? Reify is a regular function that takes an AST and produces an AST. By the way, we could also write “reify($p)”. Can you guess what would that mean? Hint: take a look at one of the draft SIDs.

Type providers

Compile-time support for language-integrated queries and run-time query translation facilities are cool in themselves, but they need one more component to form a practically useful solution.

We need to somehow map datasource entities onto the classes of our programming language.

That’s a well-known design problem of the O/R ecosystem. Some people go for textual code generation, others advocate manually written classes. We will explore a different route that is rapidly gaining momentum: type providers, courtesy of Microsoft Research.

Type providers, the macro way

Type providers are all about generating the representation of datasource entities (tables, XSD entities, LDAP entries) during the compile-time. Well, that “compile-time” part certainly rings a bell.

```
macro class MySqlDb(connectionString: String) = ...
type MyDb = MySqlDb(“Server=127.0.0.1;Database=Bar;”)
new MyDb().filter(p => p.startsWith(“foo”)).toList
```

From the perspective of the user, the code didn’t become more complex! That’s good – we’re still lightweight.

Macro types

…are nothing more than a fancy sugar for the macros we've seen before.

This flavor of macros also takes ASTs and returns an AST, though the return value represents not just an expression, but a full-blown class. Really, it's that simple.

We can immediately enjoy seamless integration and orthogonality. First of all, macro developers can reuse and override each others' logic, since macros are first-class language entities. Secondly, users can inherit from macro types and adjust their functionality in a familiar way.

Pushing this even further

After fleshing out the notion of macro defs, and adopting a Martin's brilliant idea of macro types, I was happy enough and wanted to cut down the brainstorming, though deep inside there was something that still bugged me.

Here's the deal. Here at LAMP we have a fun project that uses lightweight modular staging to crosscompile Scala programs into JavaScript.

Nada and Greg are developing that stuff and are constantly struggling with the problem that is inherent to most polymorphic languages.

Variadic templates

Say, you want to write the code that deals with functions? No problem – just define a gazillion of overloads: one for Function0[R], one for Function1[T1, R], one for… well, you get the drill.

Even Scala spec commemorates this meme by saying that "A tuple type is an alias for the class Tuple*n*".

Okay, we already know how to generate individual classes with tedious content, but how do we generate a bunch of classes without resorting to declaring all of them by hand?

Macro packages

Generalizing the idea of macro classes, we go further and introduce macro packages:

```
macro package scala.tuples(n: Int) = …
import scala.tuples(22)._
```

As usual, this macro takes an AST and produces another AST, nothing more. And again, with macros we enjoy perfect composability: for example, we can generate a package that includes the contents of another package and adds something of its own.

Lightweight macros

Summary

So far we've seen the following flavors of lightweight macros (lightweight = transparently integrated into Scala):

1) Macro defs rewrite calls to macro methods by generating ASTs during compilation and inlining them into call sites.

2) Macro types generate new classes/traits that are either directly instantiated or mixed into other classes/traits.

3) Macro packages dynamically produce a bunch of boilerplate declarations that can be imported and used in a normal fashion.

Generalization

Okay, we have macro defs, macro types and macro packages. Let's explore the design space a bit more to look for other sensible applications of macros.

The gist of all these macro XXX thingies is defining something that can be used in place of those XXXs. For example, macro types can be used wherever you use regular types (e.g. in generic type arguments).

To put it in a nutshell, macros virtualize definitions, i.e. XXXs that are covered in Chapter 4 "Basic Declarations and Definitions". Namely: vals, vars, defs, types, classes, traits (but not implicits – that would be too much, lol).

Macro constructors

For example, we can make use of macro constructors. That's a feature inspired by Dart's factories. By simply adding the "macro" prefix to a constructor definition, we can transform constructors into factories without having to change the call sites.

All in all, it was after I thought about macro constructors when I realized that macros can be generalized to affect not only defs, but all other definitions as well (e.g. packages).

Glitches

To be honest, I would be glad if I didn't have to write this slide. The point is that language integration is not 100% seamless.

The main issue involves dynamic dispatch. It's virtually impossible to mix compile-time metaprogramming and run-time polymorphism. This is doable, but all implementations horrify me.

Another issue is that you cannot really use macro types wherever you wish. Say, what does "M(1)(2)" stand for? Is it a single-argument constructor applied to the type generated by M(1)? Or is it a curried constructor of the type M? Though, that's not a showstopper, since type aliases totally address this issue.

Macro annotations

Out of thin air

All right, various flavors of lightweight macros are based on the very same idea of virtualizing language definitions, but macro annotations are just different.

The idea is simple: you define a special macro, macro annotation, and then ascribe whatever (an expression, a parameter of a method, a type parameter of a class, etc.) with a corresponding annotation (hi, Nemerle!).

These beasts are, in some sense, dual to lightweight macros. The latter are transparent to the caller, while the prior require an explicit trigger.

Usefulness

Why bother with explicit annotations when we already have transparent macros? Well, macro annotations can achieve something that is impossible for lightweight macros – they can modify pre-existing methods and classes.

Say, for some reason you want to persist ASTs of certain methods across compilations (this is useful for the infrastructure of LINQ, though, I won't go into the details right now).

It's completely unclear how to do this right off the bat. Well, maybe decompilation could be an option, but it's complex, and it isn't 100% reliable.

Pimp my methods/classes

If only we could tell the compiler to take the methods of interest, take their expression trees and serialize them into whatever form.

The first part of this wish, marking the methods of interest, can be achieved by plain old annotations, while the second part, making the compiler do something custom, is definitely about macros.

If we combine these two concepts, we come up with the notion of macro annotations.

Making it even more useful

We've seen that macro annotations provide unique functionality, though this comes at a cost – all methods and classes that need to be pimped are to be annotated manually. What've even worse – if some third-party library was compiled without those annotations, there's nothing we can do.

Package annotations to the rescue! We can define a special macro, global to our program, that, during the compile-time, will automatically ascribe the entities to be pimped with relevant macro annotations (yep, that's a macro-generating macro).

For marking the entities to be affected we could use explicit configuration (AOP frameworks, anyone?) or conventions.

Wrapping up

Links

- Project Kepler, Compile-Time Metaprogramming for Scala
  https://github.com/xeno-by/kepler
- My blog that hosts fine-grained status updates
  http://blog-en.xeno.by
- A collection of enlightening papers about macros
  http://macros.xeno.by
- Nemerle Programming Language
  http://nemerle.org/

Acknowledgements

Since the first announcement of Project Kepler, I've been receiving a lot of feedback. Thanks for that, folks. This project wouldn't be even half that interesting without your ideas and suggestions. Please, keep it going!

Once again, I'd like to praise the Nemerle programming language. Nemerle is a living proof that it's possible and practical to do language-integrated metaprogramming in a statically-typed language with syntax. These guys natively support .NET platform, have a mature compiler, sport IDE integration and do frequent releases. Take a close look at that project!

Your feedback

…is extremely important!

During this talk you've seen several examples of feedback that made macros more powerful and orthogonal.

Do you want Scala macros to do more? Do you have a use-case that needs to be covered by macros?

Nothing is set in stone yet. You have a unique opportunity to influence the design of a potential language feature in the way that will be useful for you and your future projects!

Summary

- Macros for Scala come in two major flavors: lightweight macros and macro annotations. The prior can virtualize arbitrary definitions, while the latter are about pimping already existing definitions.

- Project Kepler now powers the Scala LINQ experiment. Our collaboration with Christopher and Stefan will shape the face of Scala macros.

- We can tailor macros to fit your particular use case. Just drop me an email (see the next slide for contacts), and we'll address your suggestions. Yep, it's as easy as that!

Questions and answers

eugene.burmako@epfl.ch

Bonus slides

First-class citizens, eh?

That was a bold statement, but it's not entirely true. The point is that macros are special and, as of such, they don't directly compose.

Say, you have m1, which has the Expr ⇒ Expr type, and would like to reuse its AST transformation logic in m2.

But how do you do that? If you just write m1(foo) inside the body of m2 that would trigger macro expansion, which is definitely not what you wanted. If you try to distinguish between macro-like and regular invocations using parameter types, then what do you do when the macro itself operates on Exprs? What do you do if it has no parameters at all?

Macros and run-time values

When I mentioned that macros cannot use run-time values, I wasn't telling the whole truth. Well, of course, macros cannot, say, read or update them, but what they can do is referencing those values, and that's enough.

```
macro class MySqlDb(connectionString: String) = ...
type MyDb = MySqlDb("Server=127.0.0.1;Database=Bar;")
```

For example, here we could provide a runtime string (i.e. a variable or a string-typed expression) and be fine with that, since MySqlDb would take an expression tree that, say, represents a variable and just burn that AST into the resulting class.

However, by the virtue of closures, that class would reference the correct run-time value, i.e. stuff would work as expected.

Implicit parameters for macro defs

In macros implicit parameters can be used for the same purpose as in regular methods, e.g. for configuration. I believe, this is quite useful, and we should support that. So, I propose that:

- All implicit parameters (except special ones) are passed in reified form. We also overload implicitly[T] to produce expression trees.
- Manifests are passed as runtime objects, since, if we think about that, runtime and reified forms of manifests is the same.
- Compiler contexts and similar stuff are "magic" implicit parameters that are always passed in runtime form.

Binary format of macros

Certainly, macros need to be compiled into some form that can be reused between compilations.

It can be a class (one per each macro), named macrodeclaringclass$macro$macroname$n that hosts:

1) The original signature with all static annotations retained (can we achieve the latter?),

2) The AST => AST transformer (the meat of the macro),

3) Possible stubs for default parameters.

Macro traits

By allowing macro traits access already existing members of the class they’re mixed into, we can enable class pimping that is more predictable than class annotations.

Annotations just take a class AST and replace it with another one. Macro traits add the functionality iteratively and in predictable order determined by linearization rules.

Macro packages

Why do we need macro packages, when we could be fine just with macro objects? First of all, due to orthogonality reasons.

But, mainly, we need a syntax to import generated stuff right into our current namespace, so that we can put generated Tuples into scala, not scala.tuples.

“macro package object = …” seems to be ok, but we might need to write several declarations of this kind, which will do what? Create multiple package objects with the same name? That’s a tough question.

Transitive imports

However, macro packages cannot close over run-time values (you cannot nest packages inside a definition), while macro objects surely can.

This means that we cannot simultaneously enjoy both automatic import of generated entities and run-time macro parameters.

Do transitive imports make a good SID? I've seen quite a few StackOverflow questions that ask for that.

On the use of macro annotations

With them, we can implement lazy parameters. One just annotates a parameter with @Lazy, and that macro annotation does the AST rewriting for the affected method.

ExpressionContext

We need some way to not only pass arguments and implicits into a macro def, but to also indicate the context the macro is being used in.

Say, for macro vals we need to know whether they are read or written. Also, for the annotation on a, say, method parameter, we most likely want to know its context, i.e. method declaration/definition.