Scala Macros: Let Our Powers Combine!

On How Rich Syntax and Static Types Work with Metaprogramming

Eugene Burmako
EPFL, Switzerland
eugene.burmako@epfl.ch

Abstract

Compile-time metaprogramming has been proven immensely useful enabling programming techniques such as language virtualization, embedding of external DSLs, self-optimization, and boilerplate generation amongst many others.

In the recent production release of Scala 2.10 we have introduced macros, an experimental facility which gives its users compile-time metaprogramming powers. Alongside of the mainline release of Scala Macros, we have also introduced other macro flavors, which provide their users with different interfaces and capabilities for interacting with the Scala compiler.

In this paper, we show how the rich syntax and static types of Scala synergize with macros, through a number of real case studies using our macros (some of which are production systems) such as language virtualization, type providers, materialization of type class instances, type-level programming, and embedding of external DSLs We explore how macros enable new and unique ways to use pre-existing language features such as implicits, dynamics, annotations, string interpolation and others, showing along the way how these synergies open up new ways of dealing with software development challenges.

1. Introduction

Compile-time metaprogramming can be thought of as the algorithmic construction of programs at compile-time. It's often used with the intent of allowing programmers to generate parts of their programs rather than having to write these program portions themselves. Thus, metaprograms are programs who have a knowledge of other programs, and which can manipulate them.

Across languages and paradigms, this sort of metaprogramming has been proven immensely useful, acting as an enabling force behind a number of programming techniques, such as: language virtualization (overloading/overriding semantics of the original programming language) [11], embedding of external domain-specific languages (tight integration of external DSLs into the host language) [39, 47], self-optimization (self-application of optimizations based on analysis of the program's own code) [34], and boilerplate generation (automatizing repetitive patterns which cannot be readily abstracted away by the underlying language) [29, 35].

In the recent production release of Scala 2.10 we have introduced Scala Macros [5] as a new experimental language feature– Scala's realization of compile-time metaprogramming. This new feature enables the compiler to recognize certain methods in Scala programs as metaprograms, or *macros*, which are then themselves invoked at certain points of compilation. When invoked, macros are provided with a compiler context, which exposes the compiler's representation of the program being compiled along with an API providing certain compiler functionality such as parsing, typechecking and error reporting. Using the API available in the context, macros can influence compilation by, for example, changing the code being compiled or affecting type inference performed by the typechecker.

The most basic form of compile-time metaprogramming in our system is achieved by *def macros*, plain methods whose invocations are expanded during compilation. In addition to these def macros, we have identified, implemented, and experimented with different *macro flavors*: dynamic macros, string interpolation, implicit macros, type macros, and macro annotations. Each of which encompasses some different way in which macros are presented to, and can be used by users. We will go on to explore a number of applications, which have proven markedly difficult or impossible to achieve via other means, each of which exercise one of these macro flavors.

Our contributions are as follows:

- The design and implementation of a number of macro flavors, which are integrated in a principled way alongside of Scala's rich syntax and strong static type system.

- A comprehensive validation of the utility of these macro flavors through a number of real case studies. We show that macros (a) enable language virtualization, (b) can implement a form of type providers, (c) can be used to automatically generate type class instances, (d) simplify type-level programming, and (e) enable embedding of external domain-specific languages. We additionally go on to show that macros can re-implement non-trivial language features such as code lifting and materialization of type class instances.

The rest of the paper is organized as follows. Section 2 provides a basic introduction to Scala macros. Section 3 introduces the macro flavors we have experimented with, setting the stage for Section 4, which outlines some of the use cases that these flavors enable and discusses alternative ways of achieving similar functionality. Throughout the paper we deliberately avoid going into the details of our macro system (expansion semantics, integration with the typechecker, handling of hygiene, interplay between macros, etc) in order to focus specifically on how macros work together with Scala's rich syntax and static type system.

2. Intuition

To get acquainted with metaprogramming in Scala, let us explore the simplest flavor of Scala macros, def macros, which were inspired by macros in Lisp [13, 16] and Nemerle [35].

Def macros are methods, whose calls are expanded at compile time. Here, expansion means transformation into a code snippet derived from the method being called and its arguments. When such macros are expanded, they operate with a context, which exposes the code to be expanded and routines to manipulate code snippets.

The def macro context provides the opaque type `Code` representing untyped code snippets, exposes the `macroApplication` method, which returns the call being expanded, and defines the `q` string interpolator, which makes it possible to create and pattern match snippets using the convenient string literal syntax. For example, `q"$x + $y"` creates a snippet which represents addition of two arguments specified by snippets `x` and `y`, and `val q"$x + $y" = z` pattern matches `z` as addition and binds `x` and `y` to summands.

Asserts are the canonical example of familiar functionality, which can be enhanced with def macros. The `assert` function evaluates the provided boolean expression and raises an error if the result of evaluation is `false`. The listing below shows a possible implementation of the `assert` macro:

```
def assert(cond: Boolean, msg: String) = macro assertImpl
def assertImpl(c: Context) = {
  import c.universe._
  val q"assert($cond, $msg)" = c.macroApplication
  q"if (!$cond) raise($msg)"
}
```

Here the `assert` function serves as a façade for the `assertImpl` metaprogram, which takes applications of `assert` and transforms them into equivalent conditionals. For example, `assert(2 + 2 == 4, "does not compute")` would be replaced with `if (!(2 + 2 == 4)) raise("does not compute")`.

Even in this simple form, the macro is arguably more useful than a corresponding function in an eager language such as Scala, because it does not calculate the message unless the asserted condition is violated. The necessity to shield the evaluation of the message for performance reasons usually produces noticeable amounts of boilerplate, which cannot be easily abstracted away. Scala does support lazy evaluation with by-name parameters, but the inner workings of their internal representation might also degrade performance. Macros are able to address the performance problem without downsides.

In addition to def macros, we've conceived, implemented, and experimented with a number of other macro flavors– macros which provide different interfaces and capabilities for interacting with the Scala compiler.

3. Hammers: The Macro Flavors

Macros realize the notion of textual abstraction [16], which consists of recognizing pieces of text that match a specification and replacing them according to a procedure. In Lisp, the origin of macros, programs are represented in a homogeneous way with S-expressions. Therefore recognition and replacement of program fragments can be done uniformly, regardless of whether a transformed fragment represents e.g. an arithmetic expression or a function declaration.

In Scala, a language with rich syntax and static types, compile-time transformations of code naturally distinguish terms and types, expressions and declarations, following the architecture of `scalac`, the Scala compiler. Therefore it makes sense to recognize the following three realizations of textual abstraction in Scala: *term macros*, which expand terms, *type macros*, which expand types, and *macro annotations*, which expand definitions. In this section we will highlight these three kinds of macros along with their flavors, which appear on the intersection with other language features.

3.1 Def macros

The most natural flavor of term macros are def macros, briefly covered in the Section 2. To the programmer, def macros look like regular Scala methods with an unusual property– when a method call in a Scala program is resolved to represent an application of a def macro, that macro definition is expanded by invoking a corresponding metaprogram, called *macro implementation*. As a convenience, the macro engine automatically destructures the method call being expanded and binds type and value arguments of the call to the corresponding parameters of the metaprogram. The param-

eters and return type of the macro implementation may be typed, as is the case in the snippet below. In this case, the types of the parameters and the return type will be used to typecheck the arguments and the result of the macro:

```
def printf(format: String, params: Any*): Unit = macro impl
def impl(c: Context)(format: c.Expr[String],
  params: c.Expr[Any]*): c.Expr[Unit] = ...
printf("hello %s", "world")
```

Just like regular methods, def macros can be declared either inside or outside of classes, can be monomorphic or polymorphic, and can participate in type inference and implicit search. The only fundamental difference with regular methods is that macros are resolved at compile time, which precludes dynamic dispatch and eta expansion.

Outside of the context of macros, many existing Scala features are typically desugared to method calls– either to calls to methods with special names like `selectDynamic` and `applyDynamic`, or to methods with special meaning like implicits. Existing features that are desugared to method calls are thus unchanged with the exception of the added capability that the inserted method calls may additionally be expanded at compile time. This makes it possible to retain the same user interface and semantics for all of these existing Scala features, while also gaining code generation and compile-time programmability powers provided by macros.

3.2 Dynamic macros

Beginning with version 2.9, Scala has provided a static proxying-like facility by rewriting operations with non-existent fields and calls to non-existent methods on targets extending the `Dynamic` trait, into corresponding calls to `selectDynamic`, `updateDynamic` and `applyDynamic`.

For example, the following code snippet will print `hello world`.

```
class JsonObject(fields: Map[String, Any]) extends Dynamic {
  def selectDynamic(fieldName: String) = fields(fieldName)
}
val jo = new JsonObject(Map("greeting" -> "hello world"))
println(jo.greeting)
```

If one turns a `doDynamic` method into a def macro, it becomes possible to perform on-demand code generation. For example, dynamic macros can be used to reduce the amount of generated code for the situations when comprehensive code generation is impractical [36].

3.3 String interpolation

String interpolation is a new feature in Scala 2.10, which introduces extensible string literal syntax and establishes desugaring rules to standardize programmability.

```
val world = "world"
s"hello $world"
// desugars into: StringContext("hello ", "").s(world)
```

String interpolation was specifically designed with macros in mind. Defined as a regular method, a string interpolator has to perform potentially costly parsing, validation and interpolation at runtime. On the other hand, implementing an interpolator as a macro allows the programmer to optimize these typical tasks of handling external domain-specific languages.

3.4 Implicit macros

Implicit macros have been discovered to enable materialization of type class instances encoded with implicits [8], enabling boilerplate-free generic programming [14].

In the example below, in order to overcome type erasure, the `generic` method requires an instance of the `Manifest` type class to be passed along with the `x` argument. Manifests exist so as to carry the information about static types prior to erasure at compile-time, along to runtime. This makes it possible to know, at runtime, what `x`'s static type is.

```
def foo[T](x: T)(implicit m: Manifest[T]) = ...
foo(2) // what the user writes
foo(2)(Manifest.Int) // what happens under the covers
```

Of course, having to manually provide manifests to call generic methods that need to work around erasure is not an option. Therefore, since version 2.8, implicit search in `scalac` is hardcoded to automatically synthesize instances of the `Manifest` type class when no suitable implicit value can be found in scope

By declaring an appropriate implicit def as a macro, as described in Section 4, it becomes possible to unhardcode the part of the compiler that performs materialization of implicits, simplifying the language and reducing maintenance efforts. An important point about this technique is that it naturally scales to arbitrary type classes and target types.

3.5 Type macros

Type macros are to types as def macros are to terms. Whenever `scalac` encounters an occurrence of a type macro, possibly applied, it expands this occurrence according to the underlying macro implementation. In a sense, type macros generalize upon type aliases (which are already capable of type-level expansion), by allowing not only type arguments, but also value arguments, and supporting arbitrary expansion logic.

In the example below, the `Db` object extends `H2Db("...")`. `H2Db` is a type macro, therefore its application expands by taking a database connection string, generating a trait containing classes and values corresponding to the tables in the database, and returning a reference to the generated trait. As a result, `Db` ends up inheriting from a synthetic trait, which encapsulates the given database.

```
type H2Db(url: String) = macro impl
object Db extends H2Db("jdbc:h2:coffees.h2.db")
// expands into:
```

```
// @synthetic trait CoffeesH2Db$1 {
//   case class Coffee(...)
//   val Coffees: Table[Coffee] = ...
//   ...
// }
// object Db extends CoffeesH2Db$1
println(Db.Coffees.all)
```

The main use case of type macros is to enable code generation from a schema that's fully available at compile time. Another useful application of type macros is giving terms a promotion to the type level [46], either explicitly or triggered by implicit conversions.

3.6 Macro annotations

Despite being able to support a fair share of use cases, term macros and type macros alone are not enough to exhaustively cover the syntax of Scala. Along with the need for expression-level and type-level rewritings, there is a necessity in macros that transform definitions.

Note that the notion of a definition transformer, even though usually in a different form, is available in other languages. For example, Python has decorators [4] that alter the functionality of functions, methods and classes they are attached to, .NET languages have custom attributes which provide static metadata for their annottees, and the JVM also supports something like custom attributes under the name of annotations.

Inspired by Nemerle, which makes it possible for specially defined .NET macro attributes to transform annotated definitions, we have developed the notion of *macro annotations*, definition-transforming macros.

```
@case class C(x: Int)
// expands into:
// class C(x: Int) {
//   /* standard case class methods like toString */
// }
// object C {
//   /* standard case companion methods like unapply */
// }
```

A motivational use case for macro annotations is the modularization of the implementations of lazy vals and case classes so as to be able to migrate from the compiler, to the Scala standard library as macros.

Another use of macro annotations involves transformations necessary to support other macros. For instance, serialization macros in the `scala-pickling` project [25] can sometimes benefit from helper methods defined in serialized classes. As another example, LINQ-like techniques [7, 24, 41] that rely on compile-time code lifting often have problems with externally defined methods, because such methods might have already been compiled without lifting support. In such cases macro annotations can be used to generate the necessary boilerplate.

4. Nails: The Macro Applications

In the previous section, we introduced macro flavors exposed to Scala programmers, and now we elaborate on the use cases and techniques enabled by these available macro flavors.

4.1 Language virtualization

Language virtualization is historically the first use case for Scala macros and also the direct motivator for adding macros to the language. Since macros have access to code snippets representing their arguments, it becomes possible to analyze these snippets and then overload/override the usual semantics of Scala for them, achieving language virtualization and enabling deep embedding of internal domain-specific languages [2, 6, 15, 41, 43]

In particular, language virtualization with macros enables language-integrated queries without the necessity to introduce additional language features such as type-directed lifting [24] or quotations [7].

By implementing query combinators as def macros, data providers can obtain code snippets representing queries at compile-time (like `pred` in the example), remember them for runtime (by using a lifting function, which takes a snippet and generates its runtime representation) and then translate lifted queries to a datasource-specific representation (like in `toList` in the example).

```
class Queryable[T](val query: Query) {
  def filter(pred: T => Boolean): Queryable[T] =
    macro QueryableMacros.filter[T]
  ...
  def toList: List[T] = {
    val translatedQuery = query.translate
    translatedQuery.execute.asInstanceOf[List[T]]
  }
}

object QueryableMacros {
  def filter[T: c.WeakTypeTag](c: Context)(pred: c.Code) = {
    import c.universe._
    val T: c.Type = weakTypeOf[T]
    val callee: c.Code = c.prefix
    val lifted: c.Code = QueryableMacros.lift(pred)
    q"new Queryable[$T]($callee.query.filter($lifted))"
  }
  ...
}
```

Related work. The comparison of staged approaches [7, 30, 37] with language virtualization is quite interesting, as both techniques have interesting strengths, which we illustrate below for the use case of language-integrated queries.

On the one hand, macros allow for earlier error detection (query fragments can be partially validated at compile-time) and have simpler syntax (lifting of queries is done automatically due to the fact that macros operate on code snippets,

which are already lifted, and that makes stage annotations unnecessary).

On the other hand, staging provides superior composability, because macro-based query translation can only transparently lift code inside DSL blocks (i.e. in our case, only the arguments to query combinators). In the example below, the second invocation of the `filter` macro will only see `Ident(TermName("isAffordable"))`, but not the body of the exernally defined `isAffordable` function.

```
case class Coffee(name: String, price: Double)
val coffees: Queryable[Coffee] = Db.coffees

// closed world
coffees.filter(c => c.price < 10)

// open world
def isAffordable(c: Coffee) = c.price < 10
coffees.filter(isAffordable)
```

It is for that reason that the authors of Slick [41], a macro-powered data access framework for Scala, support both macro-based and staged query embeddings, with the former being concise and the latter being extensible.

There are also middle-ground approaches, which try to get the best of two worlds. Yin-Yang [15] uses macros to transparently rewrite shallow DSL programs into equivalent deep DSL programs. Lancet [31] employs bytecode interpretation and symbolic execution to achieve staging within a JIT compiler. This approach allows to sometimes omit stage annotations.

4.2 Type providers

Type providers [36] are a strongly-typed type-bridging mechanism, which enables information-rich programming in F# 3.0. A type provider is a compile-time facility, which is capable of generating definitions and their implementations based on static parameters describing datasources. In the example below taken from [36], the programmer uses the OData type provider, supplying it with a URL pointing to the data schema, creating a strongly-typed representation of the datasource, which is then used to write a strongly-typed query.

```
type NetFlix = ODataService<"...">
let netflix = NetFlix.GetDataContext()
let avatarTitles =
  query { for t in netflix.Titles do
          where (t.Name.Contains "Avatar") sortBy t.Name
          take 100 }
```

In Scala, type macros provide a way to generate traits, classes and objects containing arbitrary Scala code. Generated definitions can, for example, contain inner classes that represent database table schemas and lazy values that represent tables themselves. When encapsulated in an object, generated inner definitions can then be made visible to the outer world using the standard import mechanism.

An important feature of type providers in F# is that they generate datasource representations lazily, providing types and their members only when explicitly requested by the compiler. This becomes crucial when generating strongly-typed wrappers for datasource entities is either redundant (from performance and/or reflection standpoints) or infeasible (authors of [36] mention cases where the generated code is too large for the limits of a .NET process).

The notion of erased type providers cannot be readily implemented with Scala macros, but there are ways to avoid some of the undesired code generation burden. Instead of generating a class per each entity in a datasource it might be enough to generate a single class for all the entities powered by dynamic macros. As described in Section 3, extending the `Dynamic` trait and implementing corresponding `doDynamic` methods with macros allows for on-demand code generation.

4.3 Materialization of type class instances

Type classes, originally introduced in [45] as a principled approach to ad-hoc polymorphism, have proven to be useful to support such techniques as retroactive extension, generic programming and type-level computations.

As codified in [8], type classes can be expressed in Scala using a type-directed implicit parameter passing mechanism. In fact, type classes are very popular in Scala, used to work around erasure [27], express generic numeric computations [28], support generic programming [32], implement serialization [25, 44], and so on.

The example below defines the `Showable` type class, which abstracts over a prettyprinting strategy. The accompanying `show` method takes two parameters: an explicit one, the target, and an implicit one, which carries the instance of `Showable`. After being declared like that, `show` can be called with only the target provided, and `scalac` will try to infer the corresponding type class instance from the scope of the call site based on the type of the target. If there is a matching implicit value in scope, it will be inferred and compilation will succeed, otherwise a compilation error will occur.

```
trait Showable[T] { def show(x: T): String }
def show[T](x: T)(implicit s: Showable[T]) = s.show(x)

implicit object IntShowable { def show(x: Int) = x.toString }
show(42) // "42"
show("42") // compilation error
```

One of the well-known problems with type classes, in general and in particular in Scala, is that instance definitions for similar types are frequently very similar, which leads to proliferation of boilerplate code.

For example, for a lot of objects prettyprinting means printing the name of their class and the names and values of the fields. Even though this and similar recipes are very concise, in practice it is often impossible to implement them

concisely, so the programmer is forced to repeat himself over and over again. This very use case can be implemented with runtime reflection, which is available in the Java Virtual Machine, but oftentimes reflection is either too imprecise because of erasure or too slow because of the overhead it imposes.

```
class C(x: Int)
implicit def cShowable = new Showable[C] {
  def show(c: C) = "C(" + c.x + ")"
}

class D(x: Int)
implicit def dShowable = new Showable[D] {
  def show(d: D) = "D(" + d.x + ")"
}
```

With implicit macros it becomes possible to eliminate the boilerplate by completely removing the need to manually define type class instances.

```
trait Showable[T] { def show(x: T): String }
object Showable {
  implicit def materializeShowable[T]: Showable[T] = macro ...
}
```

Instead of writing multiple instance definitions, the programmer defines a single `materializeShowable` macro in the companion object of the `Showable` type class. Members of a companion object belong to implicit scope of an associated type class, which means that in cases when the programmer does not provide an explicit instance of `Showable`, the materializer will be called. Upon being invoked, the materializer can acquire a representation of `T` and generate the appropriate instance of the `Showable` type class.

A nice thing about implicit macros is that they seamlessly meld into the pre-existing infrastructure of implicit search. Such standard features of Scala implicits as multi-parametricity and overlapping instances are available to implicit macros without any special effort from the programmer. For example, it is possible to define a non-macro prettyprinter for lists of prettyprintable elements and have it transparently integrated with the macro-based materializer.

```
implicit def listShowable[T](implicit s: Showable[T]) =
  new Showable[List[T]] {
    def show(x: List[T]) = {
      x.map(s.show).mkString("List(", ", ", ")")
    }
  }

show(List(42)) // prints: List(42)
```

In this case, the required instance `Showable[Int]` would be generated by the materializing macro defined above. Thus, by making macros implicit, they can be used to automate the materializtion of type class instances, while at the same time seamlessly integrating with non-macro implicits.

Related work. It it interesting to compare the macro-based materialization approach with a generic deriving mechanism proposed for Haskell in [19].

Given that the programmer defines an isomorphism between datatypes in the program and their type representations, the deriving mechanism makes it possible to write a generic function that provides an implementation of a derived type class and works across arbitrary isomorphic datatypes. This eliminates most of the boilerplate associated with type class instantiations, and the rest (auto-generation of isomorphisms and the necessity to define trivial type class instances, which delegate to the generic implementation) can be implemented in the compiler.

If we compare the two aforementioned techniques of type class instance generation, it can be seen that in the isomorphism-based approach derived instances are interpreted, relying on a generic programming framework to execute the underlying generic function while traversing the representation of the underlying type. To the contrast, in the macro-based approach the instances are compiled, being specialized to the underlying type at compile time, removing the overhead of interpretation. This brings a natural question of whether it is possible to automatically produce compiled instances from interpreted ones.

[1] describes a manual translation technique based on compile-time metaprogramming capabilities of Haskell and a collection of code generating combinators, while [20] outlines a semi-automatic facility that leverages advanced optimization features of Glasgow Haskell Compiler. However, to the best of our knowledge, the question of fully automatic translation remains open.

4.4 Type-level programming

Type-level programming is a technique which involves writing functions that operate on types and using these functions to encode advanced type constraints and achieve precision in type signatures. With this technique it is, for example, possible to express functional dependencies [8], something which cannot be achieved in typical variations of System F_ω.

While type-level programming has proven to be useful in Scala, being a fundamental feature enabling the design of standard collections [27], its applications remain limited.

In our opinion one of the reasons for this is that type-level functions can only be written using implicits, which provide a clever yet awkward domain-specific language [8] for expressing general-purpose computations. With implicits being traditionally underspecified and relying on multiple typechecker features playing in concert to express non-trivial computations, it is hard to write robust and portable type-level functions. Finally there is a problem of performance, which is a consequence of the fact that implicit-based type functions are interpreted, and that interpretation is done by a launching a series of implicit searches, which repeatedly scan the entire implicit scope.

Compile-time metaprogramming provides an alternative approach to type-level computations, allowing the programmer to encode type manipulations in macros, written in full-fledged Scala, which has simpler semantics and predictable performance in comparison with the language of implicits.

As an example, we now explore how type-level computations help to verify communication between actors in distributed systems. In Akka [42], a toolkit and runtime for realizing message-passing concurrency on the JVM, actors typically interact using an untyped `tell` method. Since actors are able to send messages of arbitrary types to one another, type information is lost on the receiver side, and can typically only be recovered using pattern matching, loosening type guarantees.

```
abstract class ActorRef {
  ...
  def tell(msg: Any, sender: ActorRef): Unit = ...
  ...
}
```

To address the type unsafety problem, Akka provides a channel abstraction and introduces type specifications for channels [18]. As actors sometimes need to work with multiple message types (e.g. a main communications channel might support both payload messages forwarded to workers and administrative messages overseeing routing or throttling), a simple `Channel[Input, Output]` signature is not enough. Type specification of a channel should essentially be a type-level multimap from request types to response types. In Akka such multimaps are represented as heterogenerous lists of tuples. For example, the `(A, B) :+: (C, D) :+: TNil` type specifies a channel, which can receive messages of types `A` and `C`, responding with messages of types `B` and `D` correspondingly.

The challenge in the design of typed channels is to devise a mechanism of typechecking `tell`, which would typecheck its arguments against the multimap describing the receiver. The facts about the arguments that one might wish to verify range from simple ones such as "does the receiver support a given message type?" and "does the sender support any possible reply type from the receiver?" (the reply problem) to more complex ones like "is there a guarantee that on each step of the communication between the sender and the receiver, any possible message type can be handled by the corresponding actor?" (the ping-pong problem).

In order to implement the required type-level predicates, typed channels turn the `tell` method into a macro. Being able to get a hold of compile-time representations of the types in question (the type of the message and specifications of the sender and the receiver), the `tell` macro analyzes these types using the straightforward pattern matching and collection operations.

For example, the `replyChannels` function presented below takes a channel specification along with a message type and returns a list of possible reply types. `replyChannels` is then used in ping-pong analysis; if at a certain iteration of the analysis the resulting list is empty, the `tell` macro reports an error.

```
def replyChannels(list: Type, msg: Type): List[Type] = {
  def rec(l: Type, acc: List[Type]): List[Type] = {
    l match {
      case TypeRef(_, _, TypeRef(_, _, in :: out :: Nil) ::
                         tail :: Nil)
      if msg <:< in =>
        rec(tail, if (acc contains out) acc else out :: acc)
      case TypeRef(_, _, _ :: tail :: Nil) =>
        rec(tail, acc)
      case _ => acc.reverse
    }
  }
  val n = typeOf[Nothing]
  if (msg =:= n) List(n) else rec(list, Nil)
}
```

Related work. Despite being straightforward to implement and debug in comparison with implicits, macros as they stand now are however not necessarily the ultimate type-level programming technique. On the one hand, type-level computations with macros are more natural and more powerful than when written with implicits. Also, an important practical advantage of the macro-based approach is the quality of error messages, which can be tailored to precisely identify and present the problem to the user, in comparison with variations on the generic "could not find implicit value of type X" error. But on the other hand, naïve manipulations with raw type representations (such as e.g. `TypeRef` deconstructions in the implementation of `replyChannels`) are quite low-level. The balance between declarativeness of implicits and simplicity of macros has yet to be found.

An alternative approach to simplification of type-level programming involves incorporating some of the features present in dependently-typed languages such as Coq [38] and Agda [26] to make certain term-level constructs usable on the type level. In [46] authors present an extension to Haskell, which automatically promotes value and type constructors to become type and kind constructors, offering considerable gains in expressiveness. It would be interesting to see whether it is possible to use macros, which already know their way around the typechecker, as a vehicle for implementing a similar extension to Scala.

4.5 External domain-specific languages

External domain-specific languages are relevant even in languages like Scala which were designed to be friendly to internal DSLs. Regular expressions, XML, JSON, HTML, SQL, text templates; all of these can be succinctly represented as programs in external DSLs.

Without special language or tool support, programs view external DSLs as passive strings, which can be parsed and interpreted, but cannot communicate with the main program.

Compile-time metaprogramming provides a way to animate external DSLs, making them able to analyze and possibly influence the enclosing program [39].

In Scala, external DSLs can be embedded into code by the virtue of string interpolation, which standardizes extensible string literals and the notion of interpolation both for construction and pattern matching purposes.

For example, with string interpolation it is possible to define a domain-specific language for JSON, having the convenient `json"..."` syntax for JSON objects.

```
implicit class JsonHelper(val sc: StringContext) {
  def json(args: Any*): JSONObject = {
    val strings = sc.parts.iterator
    val expressions = args.iterator
    var buf = new StringBuffer(strings.next)
    while(strings.hasNext) {
      buf append expressions.next
      buf append strings.next
    }
    parseJson(buf)
  }
}
```

After the programmer defines the `StringContext.json` extension method, as shown on the snippet above, `scalac` will desugar `json"..."` and `json"""..."""` literals into calls to that method. Static parts of literals (like brackets and commas in `json"[$foo, $bar]"`) are then available in `StringContext.parts`), while interpolated parts (like `foo` and `bar` in the previous example) are passed as arguments to the extension method. String interpolation additionally supports pattern matching.

Turning the `json` method into a macro opens a number of possibilities to the DSL author. First of all, it allows to move the cost of parsing to compile-time and to report previously runtime errors at compile time. Secondly, it is often possible to statically validate interpolated expressions against the locations they are interpolated into. For example, the `json` macro can catch the following typo at compile time by figuring out that it does not make sense to interpolate a number into a location that expects a string:

```
val name = "answer"
val value = 42
json"{$value: $value}"
```

Moreover, by the virtue of being run inside the compiler, interpolation macros can interact with the typechecker, asking it for information and even influencing typing. For example, the quasiquoting interpolator [33] uses the types of its arguments to resolve occasional ambiguities in the grammar of interpolated Scala and also conveys the exact types of the variables bound during pattern matching to the typechecker.

Integration with the typechecker of the host program can be used to typecheck external DSLs (this direction of research is very appealing in the context of quasiquoting, but one could also imagine interpolations used to generate programs in other programming languages also benefitting from a typechecking facility). This is however non-trivial, because unlike MetaML-like quotations [7, 37], interpolation-based quasiquotes do not come with the guarantees of typeability (e.g. with such quasiquotes a well-typed program can be assembled from smaller fragments that do not make sense on their own) and can have holes that preclude typechecking.

The example provided below receives a definition of a method, takes it apart and creates its asynchronous analog by changing the name, wrapping the body in a future and adjusting the return type accordingly.

```
val q"def $name($params): $tpt = $body" = meth
val tpt1 = if (tpt.isEmpty) tpt else q"Future[$tresult]"
val name1 = TermName("async" + name.capitalize)
q"def $name1($params): $tpt1 = future { $body }"
```

Note that none of the quasiquotes in the example are typeable as is, yet there is still some room for typechecking by validating whether the `future` function and the `Future` type constructor are imported and have appropriate type signatures.

Related work. Similar approaches to embedding domain-specific languages have been explored in Haskell [21] and Ocaml [47], which also provide ways to declare isolated blocks of code written in an external language and use those blocks for construction and deconstruction of domain-specific objects.

However it remains to be explored how to integrate external languages into the type system of the host. As shown by the experiences of Template Haskell [34] and Nemerle [35], this is possible for the particular case of quasiquotes, though both approaches are tightly integrated into the corresponding macro systems, so it is not immediately obvious how to generalize them. [9] introduces SoundExt, an automated and modular mechanism to specify typing rules for domain-specific languages in a framework for language extensibility.

A promising direction of research into integration of external domain-specific languages involves syntactic language extensibility. Replacing the string interpolation frontend with a more sophisticated mechanism akin to the one described in [10], would retain the power of compile-time programmability and integration with the typechecker and also gain the flexibility and modularity of the recent developments in parsers.

4.6 Language extensibility

One man's language feature is another man's macro. This principle is well-known to the practitioners of lisps [11, 12], and we have also put it in action in Scala. There already is a handful of pre-existing or suggested language features that we were able or plan to implement with macros, simplifying the Scala compiler: type-directed lifting, autogeneration of type manifests, xml literals, source locations, case classes,

lazy values, enumerations, implicit classes and asynchronous computations [40, 43].

In the example below we illustrate idiom brackets [23] for Scala, implemented in a library [17] with a pair of macros named `idiom` and `$`.

The `idiom` macro takes a type constructor, which represents an idiom instance, and a code block, which will get some of its subexpressions transformed. The `$` macro demarcates transformation boundaries. Inside it, all non-idiom values are wrapped in the `pure` function defined by the current idiom introduced by the enclosing `idiom` macro, and all function applications are routed to go through `app`, again defined by the current idiom (the actual transformation rules are a bit more complicated, but this does not matter for the purposes of the demonstration). Implementations of `pure` and `app` are looked up by the `$` macro via an implicit search for an instance of the `Idiom` type class for the current idiom type constructor.

```
implicit val option = new Idiom[Option] {
  def pure[A](a: => A): Option[A] = Option(a)
  def app[A, B](ff: Option[A => B]): Option[A] => Option[B] =
    aa => for (f <- ff; a <- aa) yield f(a)
}

idiom[Option] {
  $(Some(42) + 1) should equal (Some(43))
  $(Some(10) + Some(5) * Some(2)) should equal (Some(20))
}
```

Note how macros have been able to implement functionality, which requires extensions to Haskell [22] and Idris [3], and how implicits make the macro modular and configurable by the user.

5. Conclusion and Future Work

We have designed and implemented a compile-time metaprogramming system for Scala, a language with rich syntax and static types.

The most important aspect of our design is that we have been able to naturally integrate macros with pre-existing language features, preserving their familiar interface and semantics, while at the same time empowering them with code generation and compile-time programmability capabilities.

Case studies show that with Scala macros it is possible to achieve language virtualization, emulate type providers, materialize type class instances, simplify type-level programming, embed external domain-specific languages and implement non-trivial language features.

Having experimented with a number of macro flavors and use cases that these flavors enable, we now plan to proceed with finding a minimalistic system which would retain the useful properties of the current design in a flexible and tractable framework. Among the design issues that need to be solved along the way are determining the appropriate detail of the compiler API, specifying hygiene and referential transparency of code manipulations, standardizing the protocol of communication between macros, and coming up with a predictable yet flexible execution model for macros.

Another interesting direction of future research is exploration of compile-time capabilities that don't quite fit into the model of textual abstraction. For instance, we would like to find a robust way for macros to interact with the compiler's symbol table in order to provide macro writers with predictable APIs to add new classes, change existing ones, introduce variables shared between macro expansions, etc. And yet another topic is finding a robust and predictable way to deeply integrate macros with the typechecker, e.g. granting macros powers to influence type inference.

References

[1] M. D. Adams and T. DuBuisson. Template your boilerplate: using template haskell for efficient generic programming. In *Haskell 2012, Copenhagen, Denmark, 13 September 2012*, pages 13–24, 2012.

[2] S. Behnke. Scala macros use case: Teaching scala. `https://www.learnscala.de/2013/01/28/en/scala-macros-use-case-teaching-scala`, 2013.

[3] E. Brady. Implementation of a general purpose programming language with dependent types. Technical report, 2013.

[4] R. Brewer. Optimal syntax for python decorators. `http://www.aminus.org/rbre/python/pydec.html`, Aug. 2004.

[5] E. Burmako and M. Odersky. Scala macros, a technical report. In *Third International Valentin Turchin Workshop on Metacomputation*, 2012.

[6] P. Butcher. Scalamock, native scala mocking framework. `https://github.com/paulbutcher/ScalaMock`, 2012.

[7] J. Cheney, S. Lindley, and P. Wadler. The essence of language-integrated query. Technical report, Mar. 2013.

[8] B. C. d. S. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *OOPSLA*, pages 341–360, 2010.

[9] S. Erdweg and F. Lorenzen. Modular and automated type-soundness verification for language extensions, 2013. To appear.

[10] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: library-based syntactic language extensibility. *ACM SIGPLAN Notices*, 46(10):391–406, Oct. 2011. OOPSLA '11 proceedings.

[11] M. Felleisen. Adding types to untyped languages. In A. Kennedy and N. Benton, editors, *TLDI 2010, Madrid, Spain, January 23, 2010*, pages 1–2. ACM, 2010.

[12] M. Flatt, R. B. Findler, and M. Felleisen. Scheme with classes, mixins, and traits. In *APLAS 2006, Sydney, Australia, November 8-10, 2006*, volume 4279, pages 270–289, 2006.

[13] M. Flatt, R. Culpepper, D. Darais, and R. B. Findler. Macros that work together - compile-time bindings, partial expansion, and definition contexts. *J. Funct. Program*, 22(2):181–216, 2012.

[14] S. L. P. Jones and R. Lämmel. Scrap your boilerplate. In *APLAS 2003, Beijing, China, November 27-29, 2003*, 2003.

[15] V. Jovanovic, V. Nikolaev, N. D. Pham, V. Ureche, S. Stucki, C. Koch, and M. Odersky. Yin-yang: Transparent deep embedding of dsls. Technical Report EPFL-REPORT-185832, EPFL, Lausanne, Switzerland, 2013.

[16] E. M. Kohlbecker. *Syntactic Extensions in the Programming Language Lisp*. PhD thesis, Indiana University, Bloomington, IN, Aug. 1986.

[17] E. Kotelnikov. Scala-idioms, idiom brackets for scala. `https://github.com/aztek/scala-idioms`, 2013.

[18] R. Kuhn. Akka, typed channels (experimental), version 2.2-snapshot. `http://doc.akka.io/docs/akka/snapshot/scala/typed-channels.html`, 2013.

[19] J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löh. A generic deriving mechanism for haskell. In *Haskell 2010, Baltimore, MD, USA, 30 September 2010*, pages 37–48. ACM, 2010.

[20] J. P. Magalhães, S. Holdermans, J. Jeuring, and A. Löh. Optimizing generics is easy! In *PEPM 2010, Madrid, Spain, January 18-19, 2010*, pages 33–42, 2010.

[21] G. Mainland. Why it's nice to be quoted: quasiquoting for haskell. In *Haskell 2007, Freiburg, Germany, September 30, 2007*, pages 73–82, 2007.

[22] C. McBride. The strathclyde haskell enhancement. `https://personal.cis.strath.ac.uk/conor.mcbride/pub/she/`.

[23] C. McBride and R. Paterson. Applicative programming with effects. *J. Funct. Program*, 18(1):1–13, 2008.

[24] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In S. Vansummeren, editor, *PODS 2006*, page 706, Chicago, Illinois, June 2006.

[25] H. Miller, P. Haller, E. Burmako, and M. Odersky. Object-oriented pickler combinators and an extensible generation framework, Mar. 2013. To appear.

[26] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 1997.

[27] M. Odersky and A. Moors. Fighting bit rot with types (experience report: Scala collections). In *FSTTCS 2009, December 15-17, 2009, IIT Kanpur, India*, volume 4, pages 427–451, 2009.

[28] E. Osheim. Spire, powerful new number types and numeric abstractions for scala. `https://github.com/non/spire`, 2013.

[29] Postsharp Technologies. Producing high-quality software with aspect-oriented programming. Technical report, July 2011.

[30] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *ACM SIGPLAN Notices*, 46(2):127–136, Feb. 2011.

[31] T. Rompf, A. K. Sujeeth, K. J. Brown, H. Lee, H. Chafi, K. Olukotun, and M. Odersky. Project lancet: Surgical precision jit compilers, Mar. 2013. To appear.

[32] M. Sabin and E. Burmako. Datatype generic programming in scala with shapeless and inference driving macros, Apr. 2013. To appear.

[33] D. Shabalin, E. Burmako, and M. Odersky. Quasiquotes for scala. Technical Report EPFL-REPORT-185242, EPFL, Lausanne, Switzerland, 2013.

[34] T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In *ACM SIGPLAN Haskell Workshop 02*, pages 1–16, Oct. 2002.

[35] K. Skalski. Syntax-extending and type-reflecting macros in an object-oriented language. Master's thesis, University of Warsaw, Poland, 2005.

[36] D. Syme, K. Battocchi, K. Takeda, D. Malayeri, J. Fisher, J. Hu, T. Liu, B. McNamara, D. Quirk, M. Taveggia, W. Chae, U. Matsveyeu, and T. Petricek. F# 3.0 - strongly-typed language support for internet-scale information sources. Technical Report MSR-TR-2012-101, Microsoft Research, september 2012.

[37] W. Taha. *Multi-Stage Programming : Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, Portland, OR, 1999.

[38] T. C. Team. The coq proof assistant. `http://coq.inria.fr`, 2012.

[39] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. *PLDI*, 46(1993):132–141, 2010.

[40] M. Torgersen. Asynchronous programming in c# and vb.net. Technical report, 2010.

[41] Typesafe Inc. Scala language integrated connection kit. `https://github.com/slick/slick`, 2012.

[42] Typesafe Inc. Akka. `http://akka.io/`, 2013.

[43] Typesafe Inc. An asynchronous programming facility for scala. `https://github.com/scala/async`, 2013.

[44] P. Voitot. Unveiling play 2.1 json api - bonus : Json inception (based on scala 2.10 macros). `http://mandubian.com/2012/11/11/JSON-inception/`, 2012.

[45] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *POPL*, pages 60–76, 1989.

[46] S. Weirich, B. A. Yorgey, J. Cretin, S. P. Jones, D. Vytiniotis, and J. P. Magalhaes. Giving haskell a promotion. Jan. 28 2012.

[47] H. Zhang and S. Zdancewic. Fan: compile-time metaprogramming for ocaml, Apr. 2013. To appear.